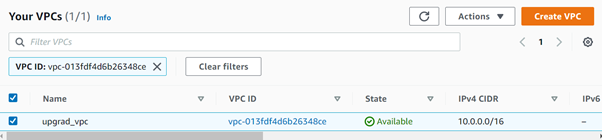
# **Installation and setup**

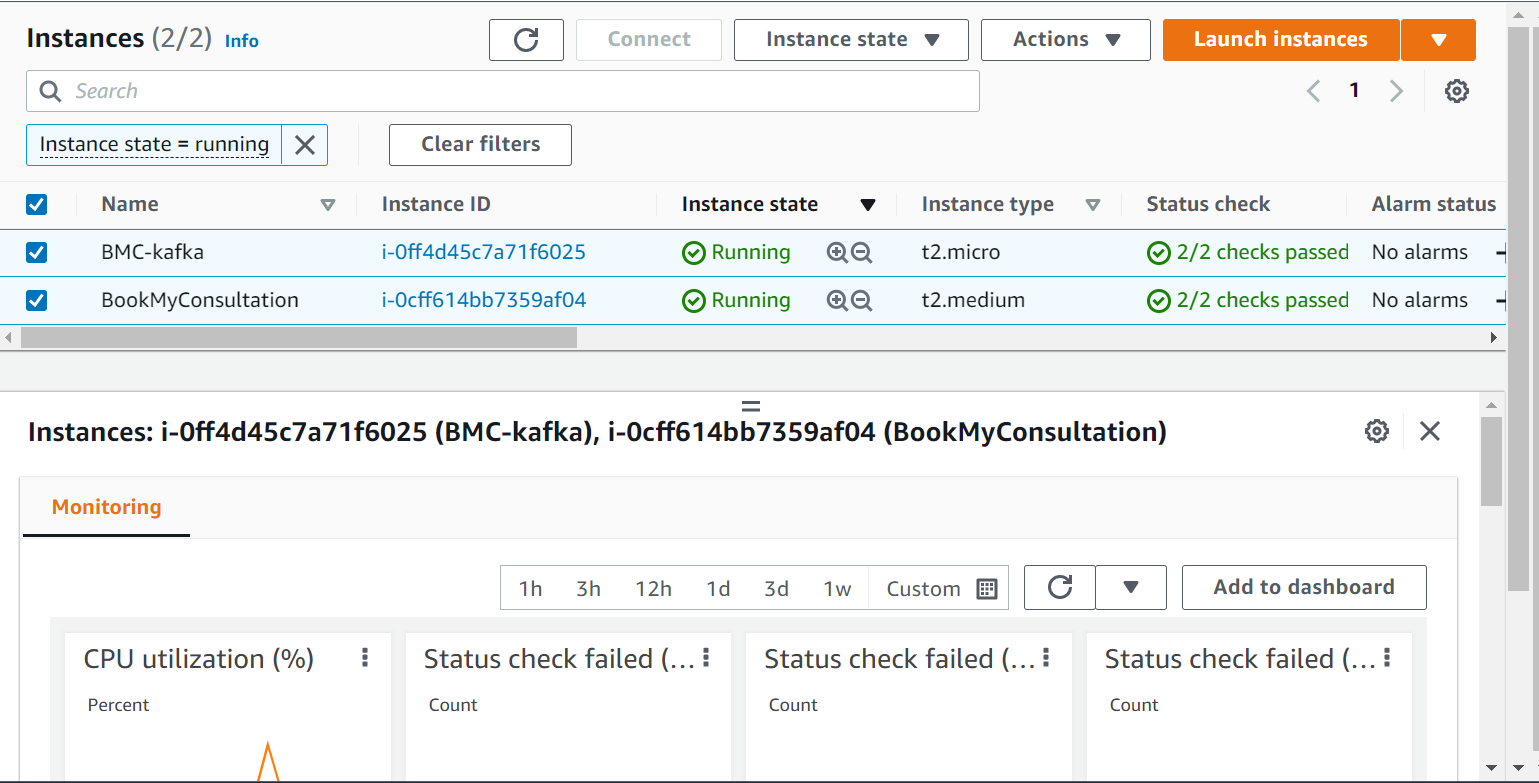
### **Setting up AWS**

To connect to our project, we set up a VPC to define the security groups for connecting to our application.



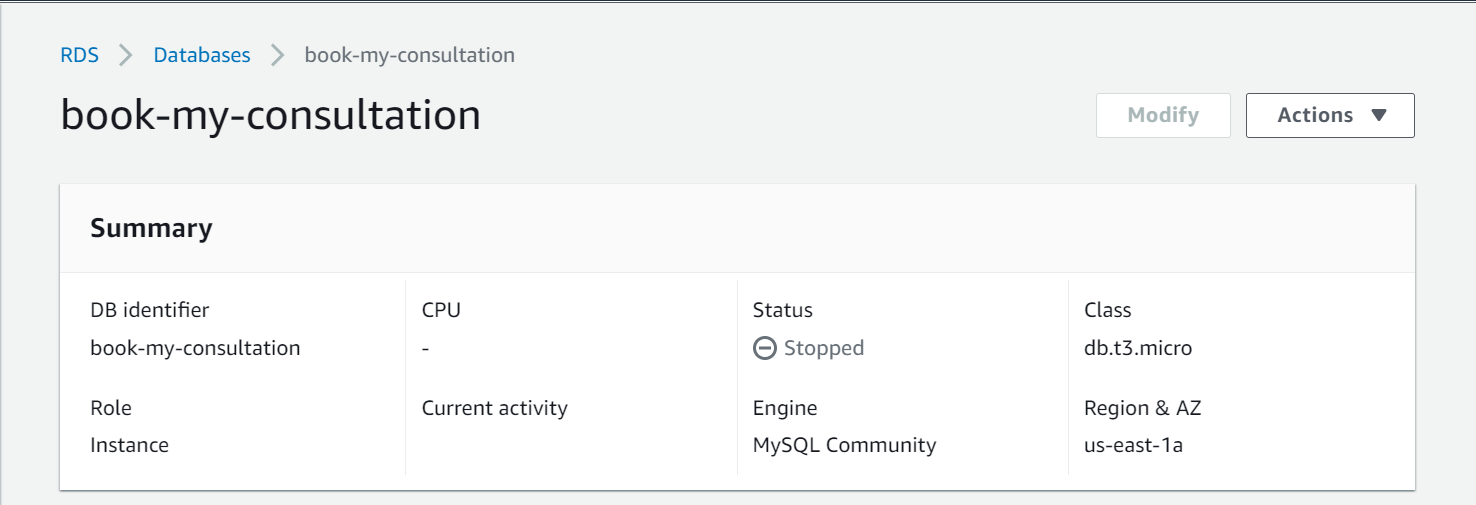
*Fig 1.1: VPCs setup*

To run our project, we are going to use two of AWS’ EC2 instances. An EC2 instance to run the containerized application, and another to run the Kafka-based notification queue.



*Fig 1.2: EC2 instances setup*

We use an RDS instance to store our SQL databases, BookMyConsultation. An RDS instance was created in my AWS account as below.



*Fig 1.3: RDS instances setup*

We also need to create a Mongo DB to store our NoSQL data of the Doctor, User, Prescriptions and Rating collections. I used the Atlas MongoDB for my collections as below.

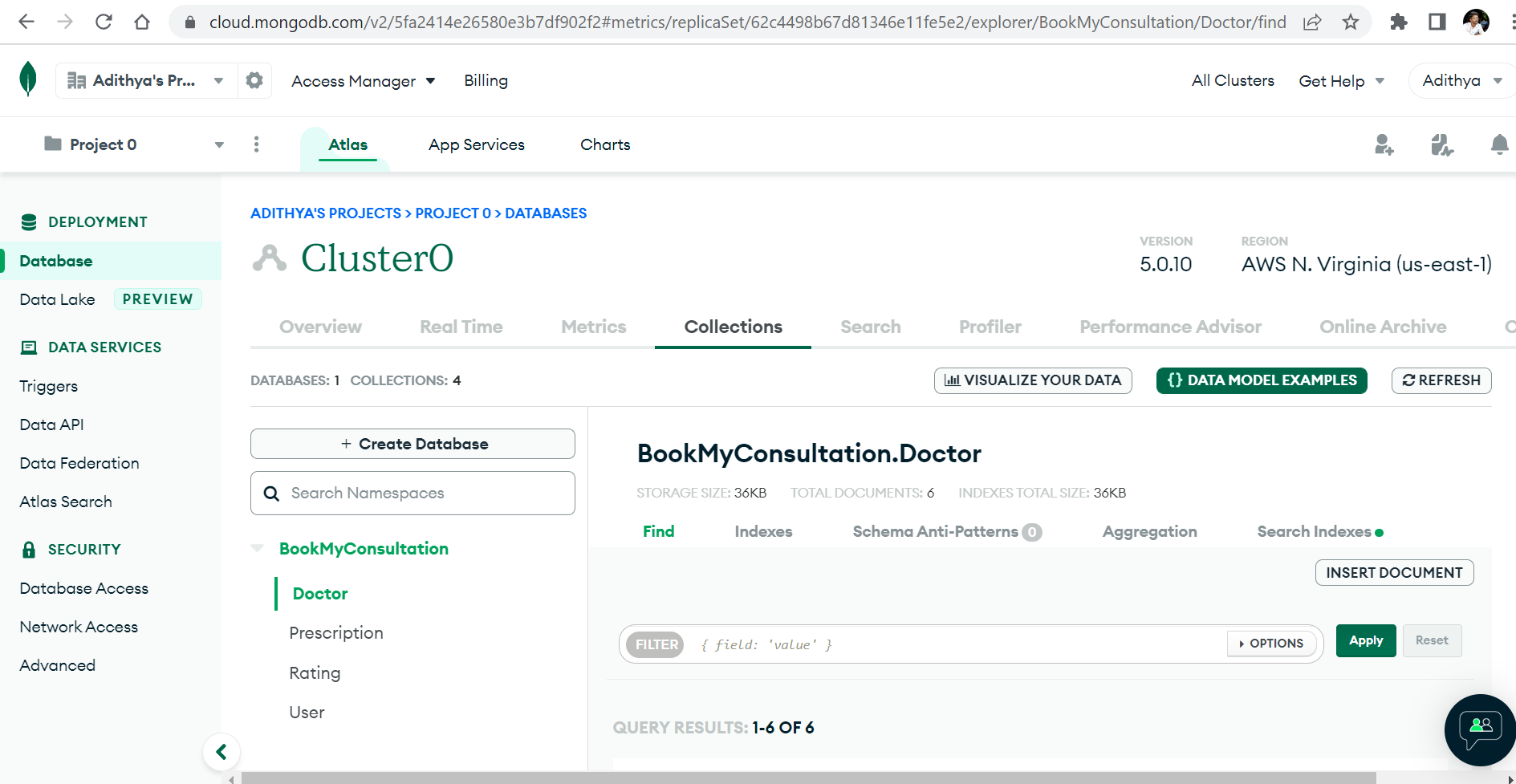


Fig 1.4: MongoDB setup

### **Setting up Application**

#### **1. Application Instance**

To connect to our application instance, we ssh into our instance using the key-value pair generated during creation.

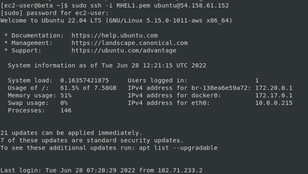


Fig 2.1a: Login to EC2 instance

Once inside, we install docker and docker-compose in the EC2 instance for running our application

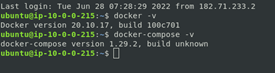


Fig 2.1b: Docker versions

Next, we move the application codebase into the ec2 instance using WinSCP

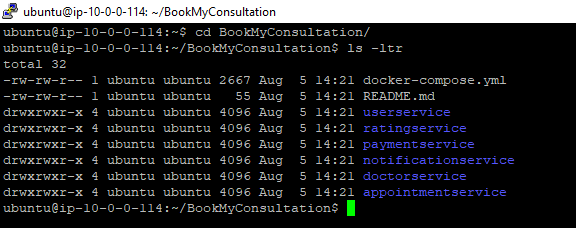


Fig 2.1c: Uploaded codebase

#### **2. Kafka Instance**

For the Kafka instance, we connect to the instance using ssh and the key-value pair generated/selected during creation. We download Java onto the instance and then follow the instructions given in the manual to download and set up Kafka and zookeeper.

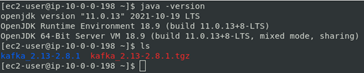


Fig 2.2: Kafka server setup

# Implementation

### **Creating Dockerfiles**

Once the code base is in the AWS instance, we create Dockerfiles for each application in order to containerize each application.

The dockerfile automatically creates a jar file for the service using a Maven image to compile and package the service.

Once the jar file is generated, we use the official java image to upload the jar file and set an entry point to run the jar file on startup. This now forms our service image.

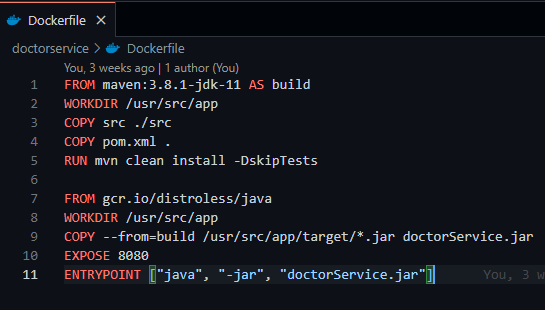


Fig 3.1: Dockerfile for DoctorService

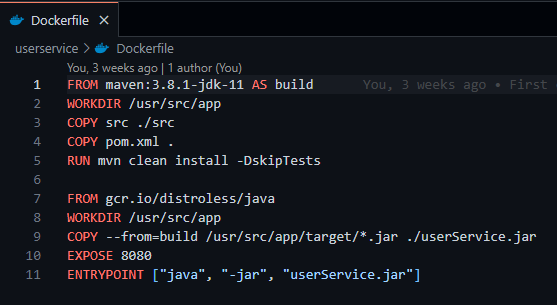


Fig 3.2: Dockerfile for UserService

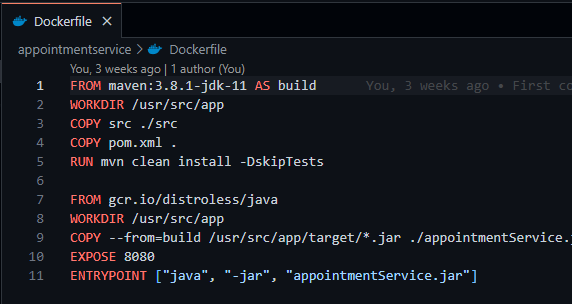


Fig 3.3: Dockerfile for AppointmentService

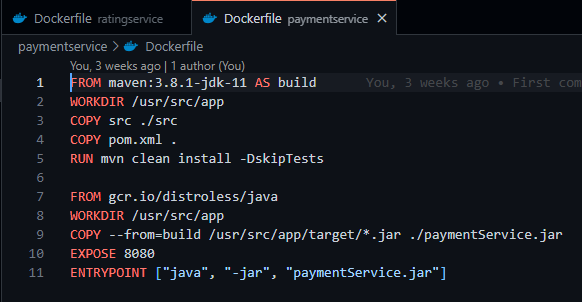


Fig 3.4: Dockerfile for PaymentService

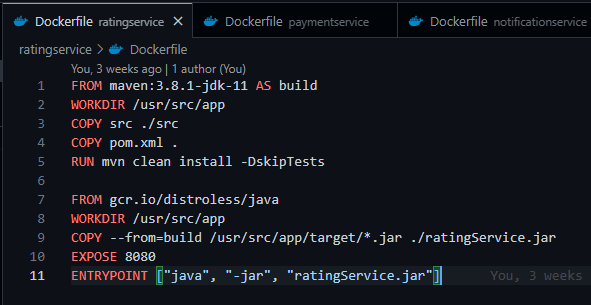


Fig 3.5: Dockerfile for ratingService

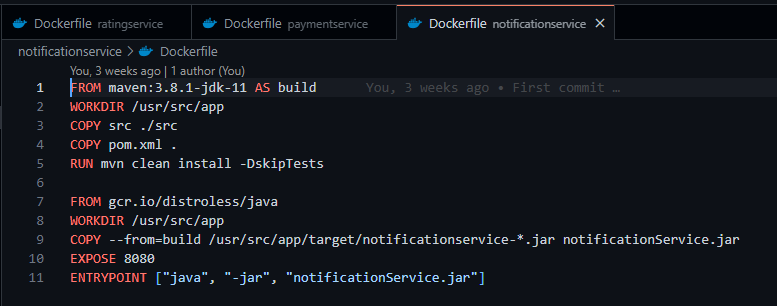
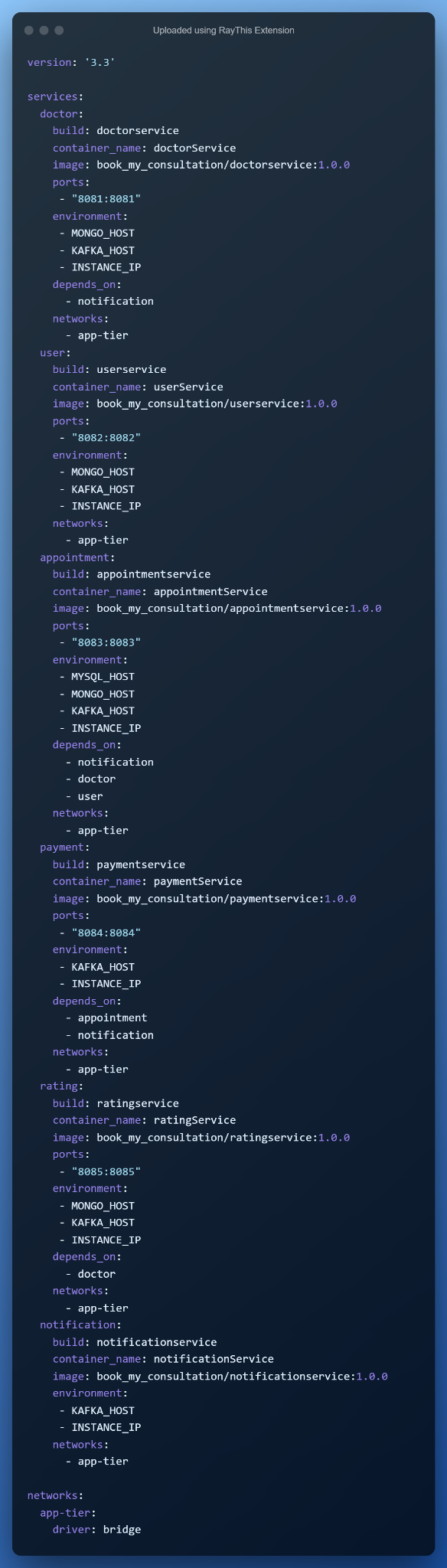


Fig 3.6: Dockerfile for NotificationService

### **Creating Docker-Compose File**

We also set up the docker-compose file to build and deploy the services through the Dockerfiles. 

### **Running the application**

To begin, let’s start the Kafka server in order for it to be ready when we start our service application.

To do that, we log into our Kafka instance and navigate to the kafka folder. Within the folder, We first have to set the ip of the kafka server within the server properties.

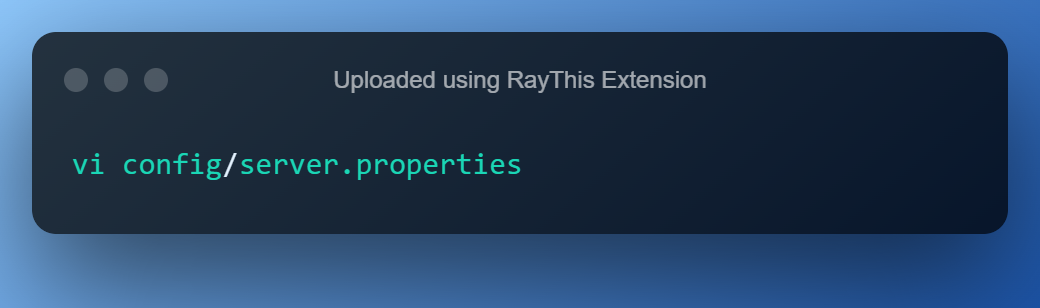


Fig 5.1: Update kafka server properties

Following that, we run the following commands.



Fig 5.2: start Kafka server

Before we start the services application, We need to set the various configurations to be used within the project. Navigate to the .env file in the project folder and update the variables with the current config.

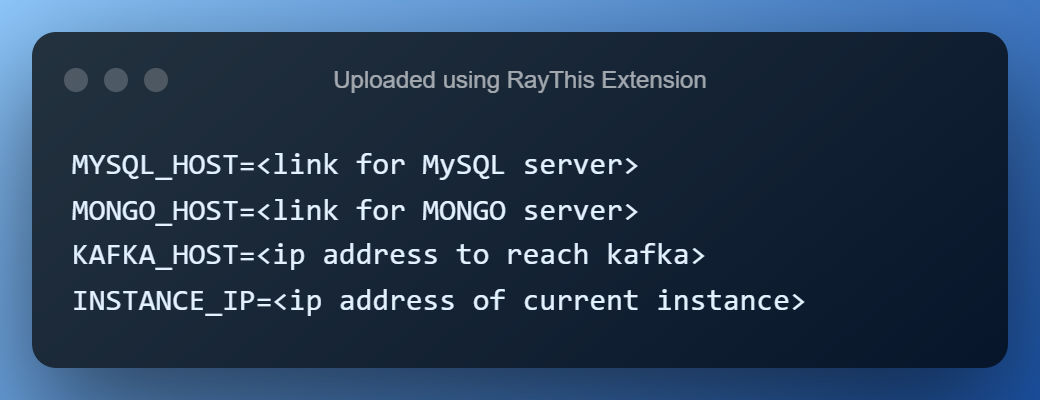


Fig 5.3: Update env variables

We need to build the images for each of the services we are deploying. To build the image for each service and tag them accordingly, we use



Fig 5.4: build service images

Once the build is done, we can start all the applications simultaneously using



Fig 5.5: start services

Once all the applications are up, the docker images and containers should be up and running.

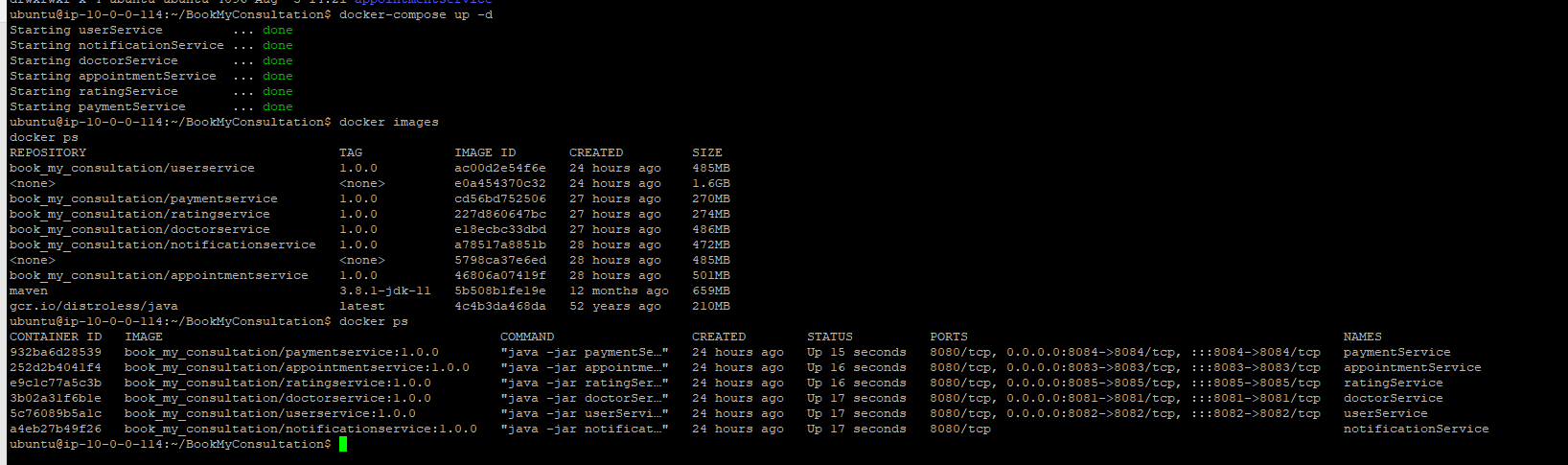


Fig 5.6: docker images and containers

### Testing with APIs

#### DoctorService

##### CreateDoctor

For this API, we validate the details of the Doctor including the first name, last name, email Id and PAN ID. If any of the validations fail, we return an error elaborating on the same.

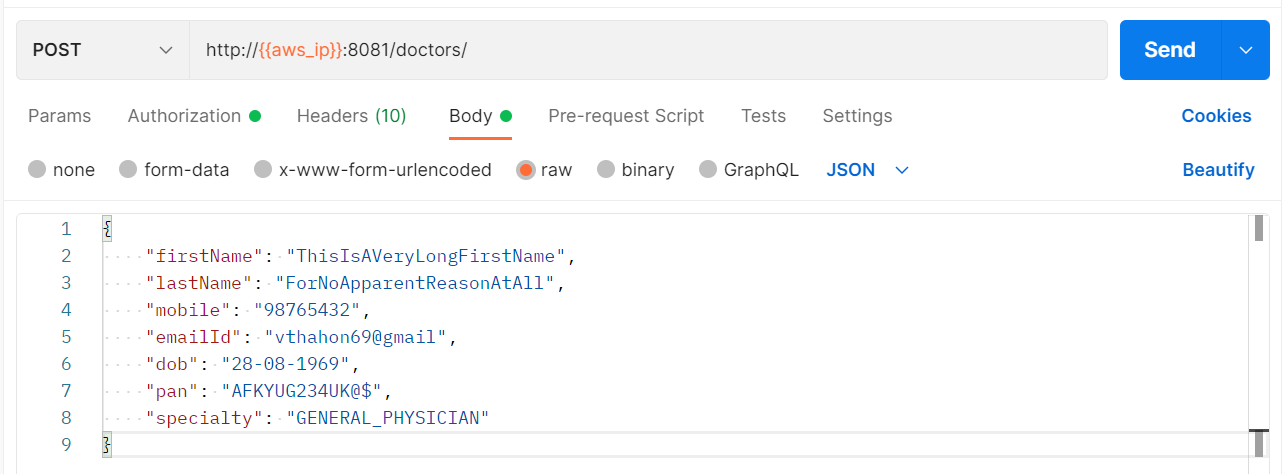


Fig 6.1a: Incorrect Details to create doctor

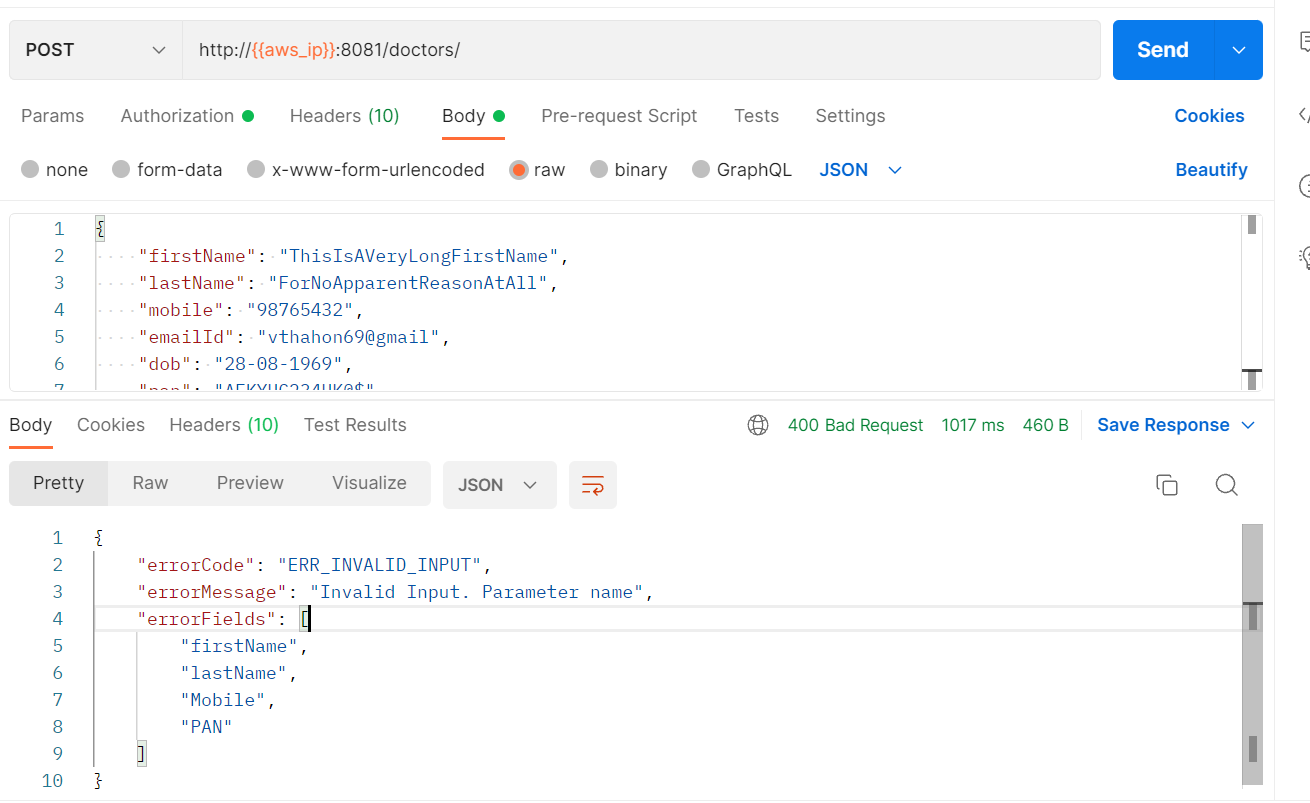


Fig 6.1b: Response for incorrect details to create Doctor

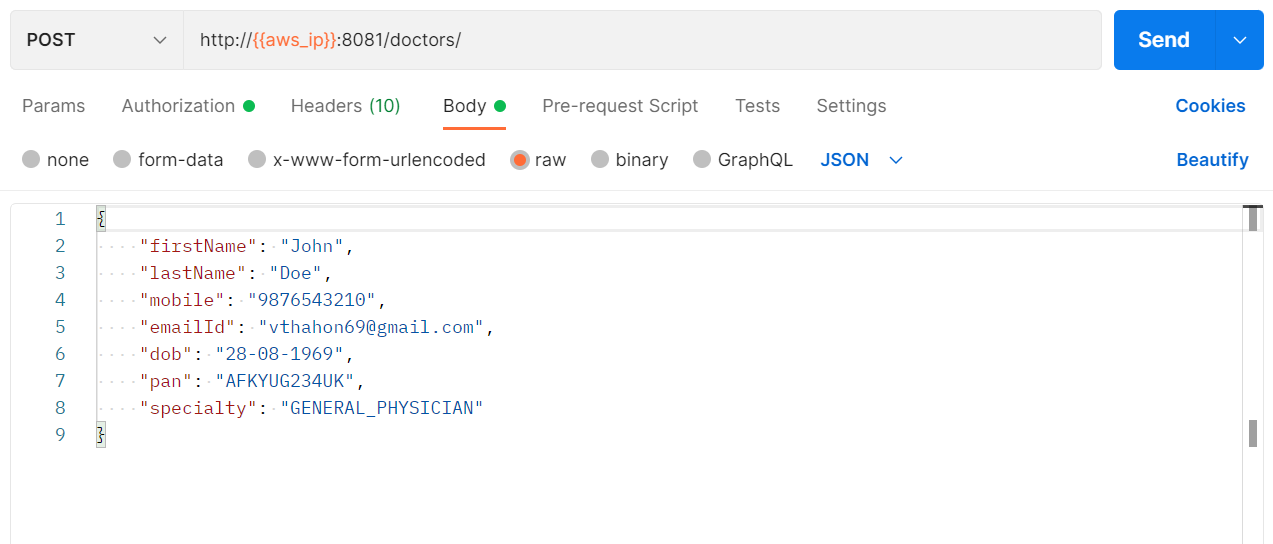


Fig 6.1c: Correct details to create doctor

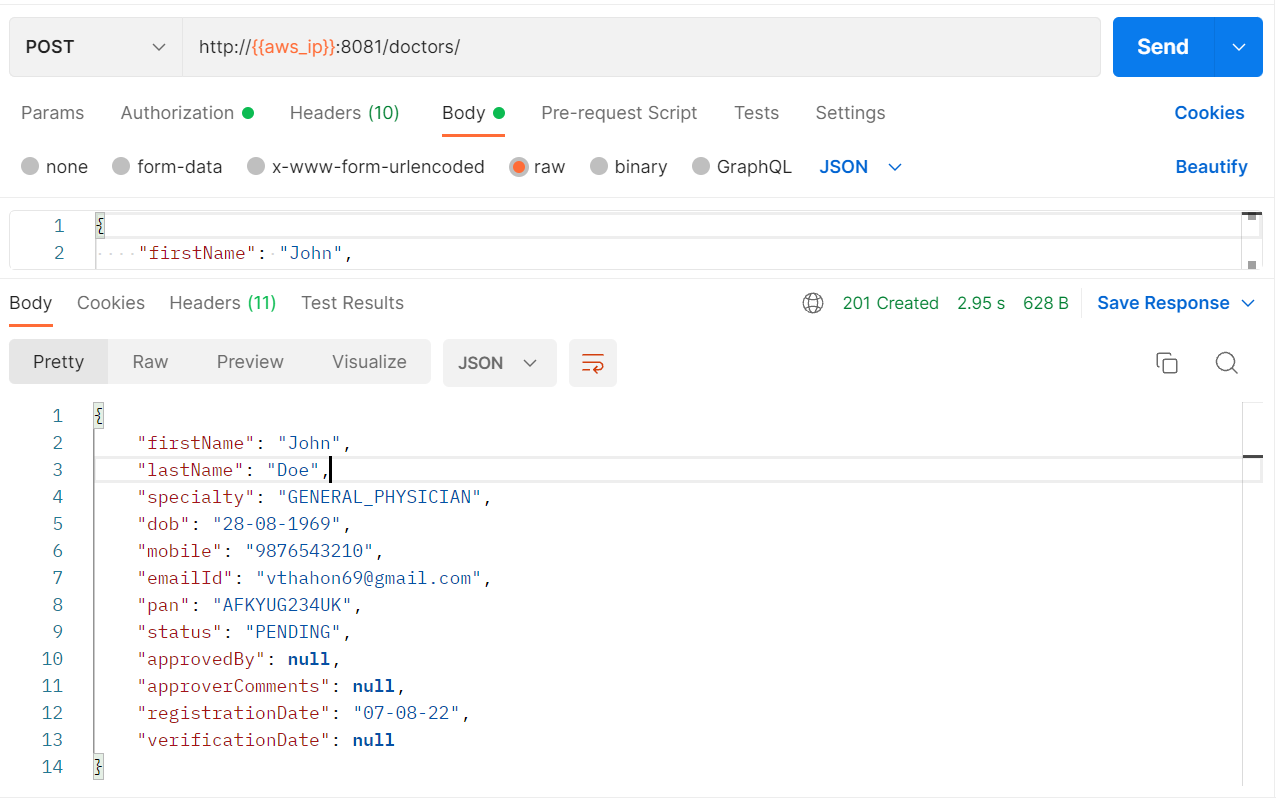


Fig 6.1d: Response for correct details to create doctor



Fig 6.1e: Notification sent to Kafka server for Doctor creation

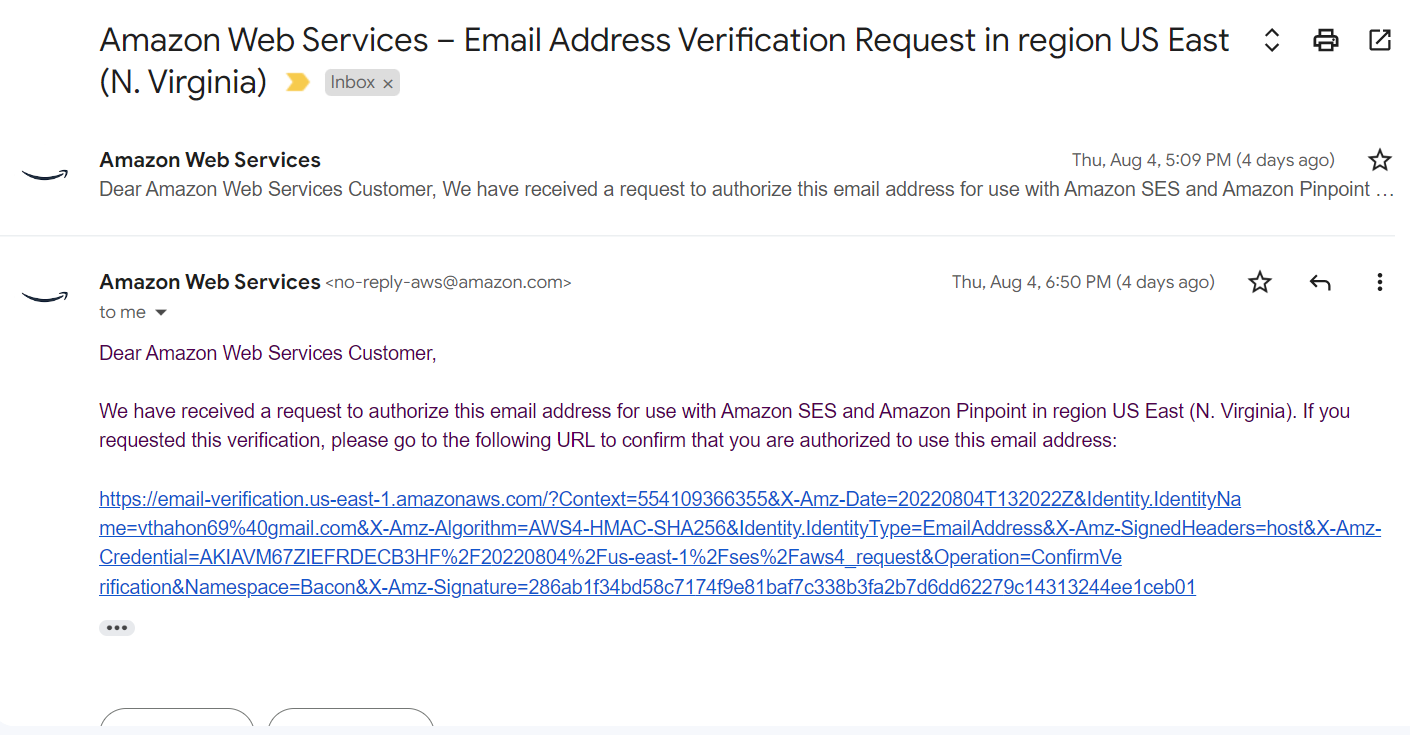


Fig 6.1f: Verification email sent to Doctor Email

##### Upload Documents to Doctor S3 bucket

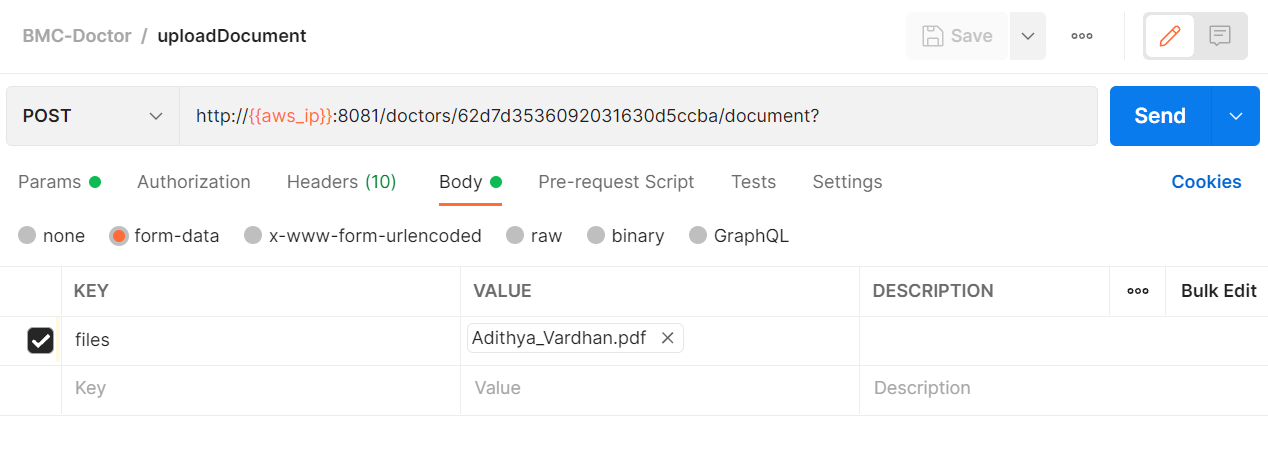


Fig 6.2a: Request to upload documents for Doctor

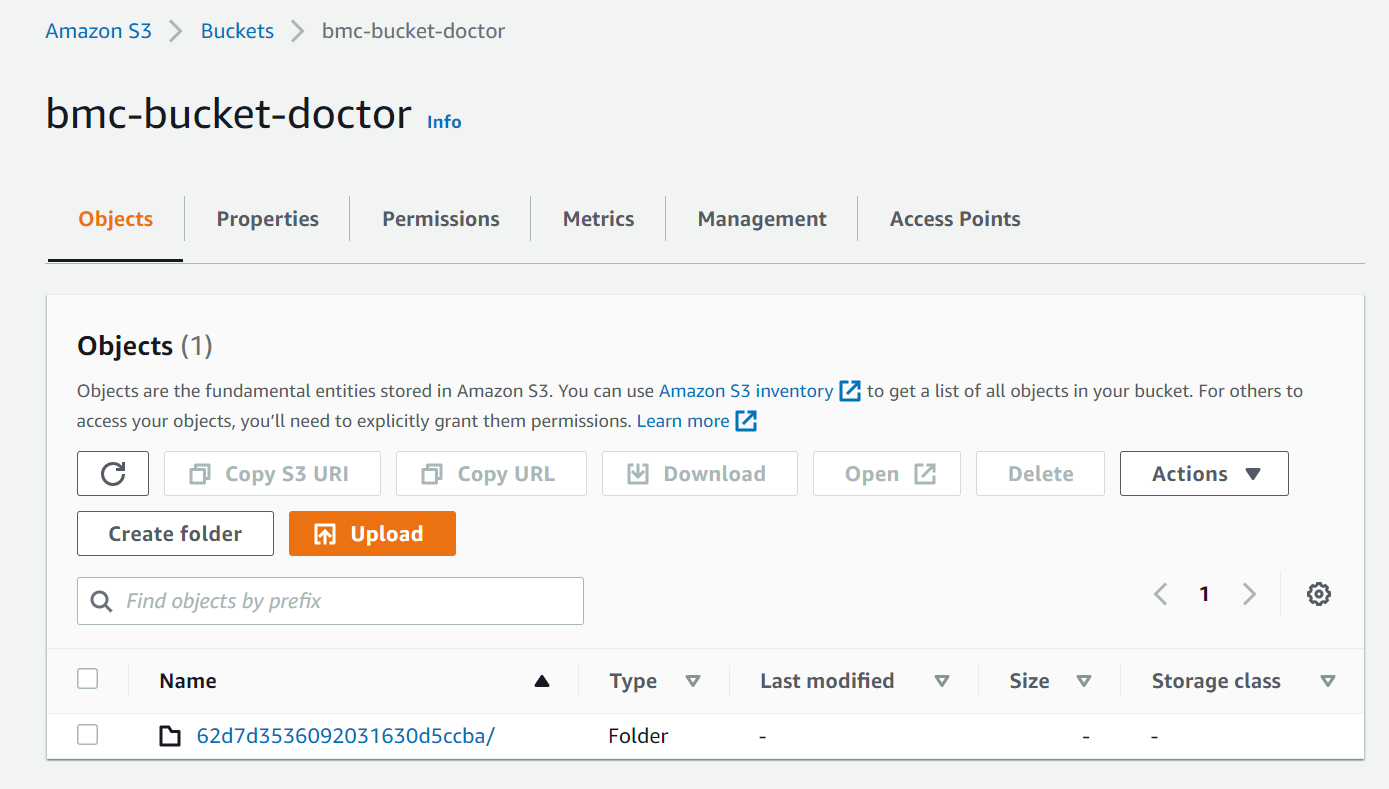


Fig 6.2b: Documents uploaded to Amazon S3

##### Approve Doctor

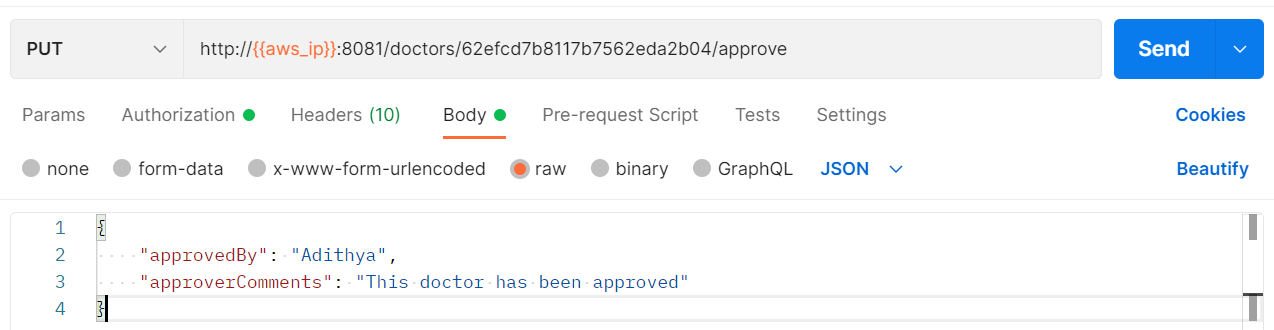


Fig 6.3a: Request to approve doctor

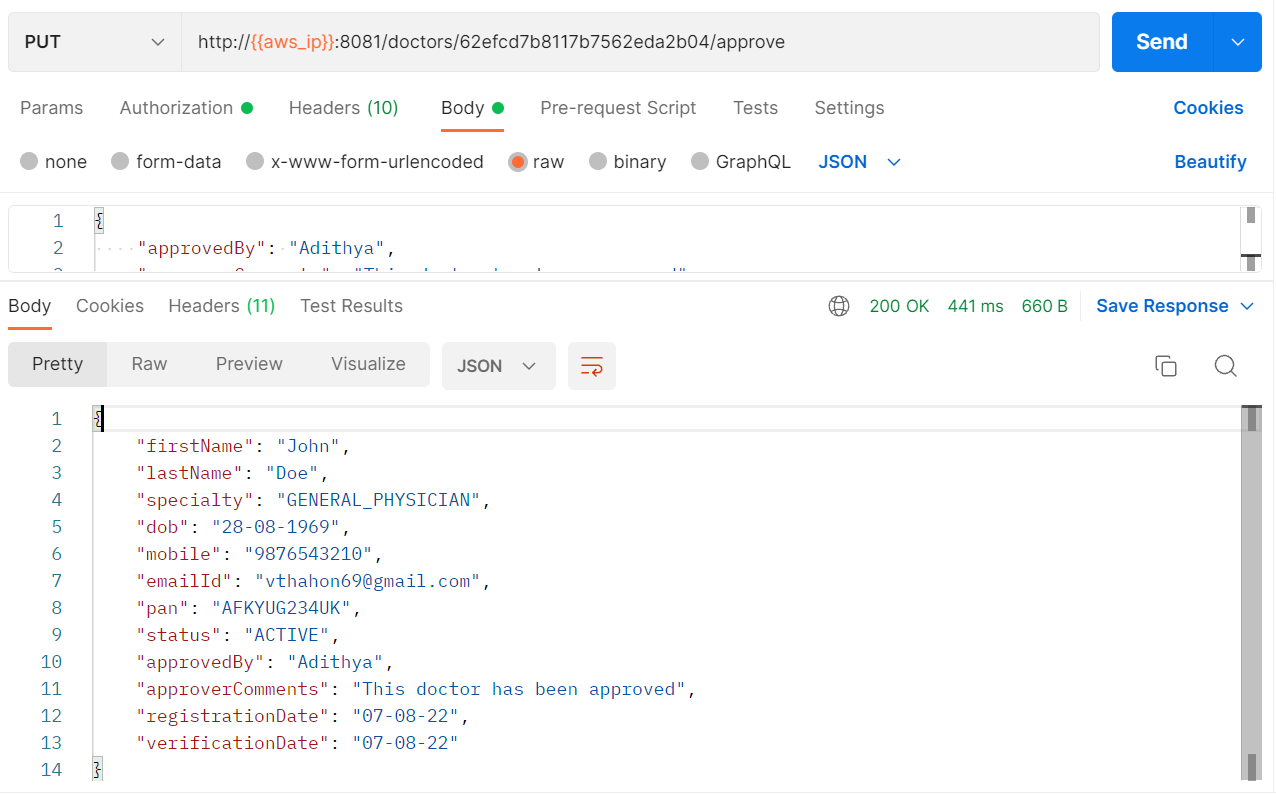


Fig 6.3b: Response for approving doctor

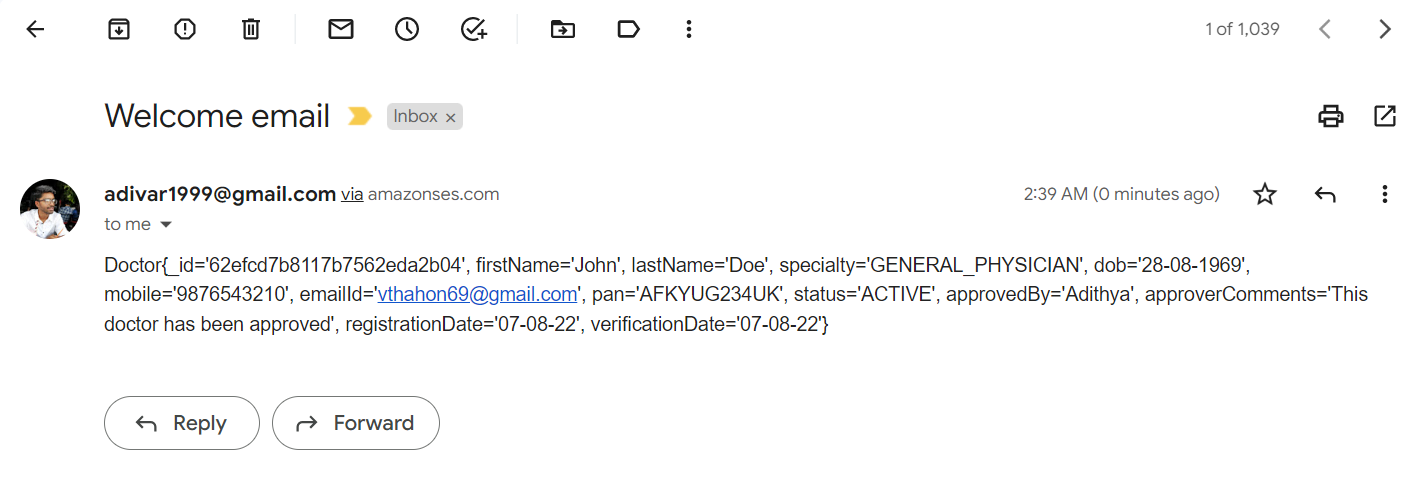


Fig 6.3c: Email for Doctor Approval

##### Reject Doctor

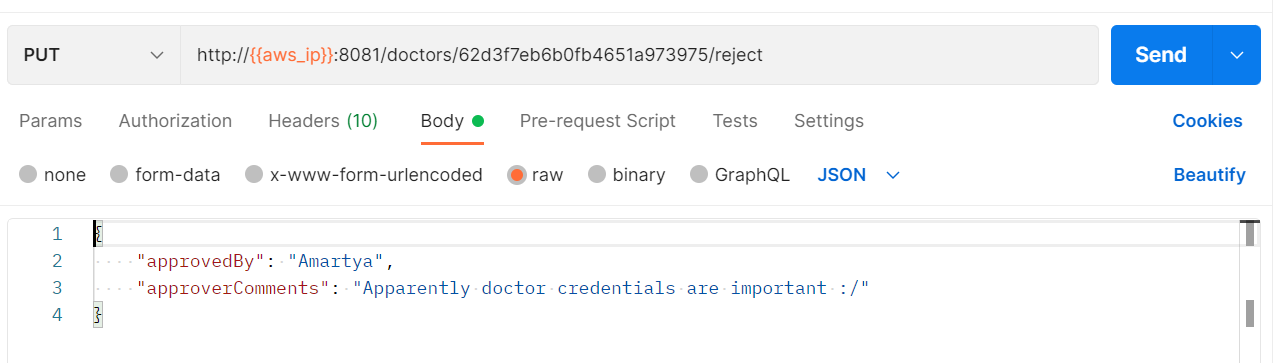


Fig 6.4a: Request to reject Doctor

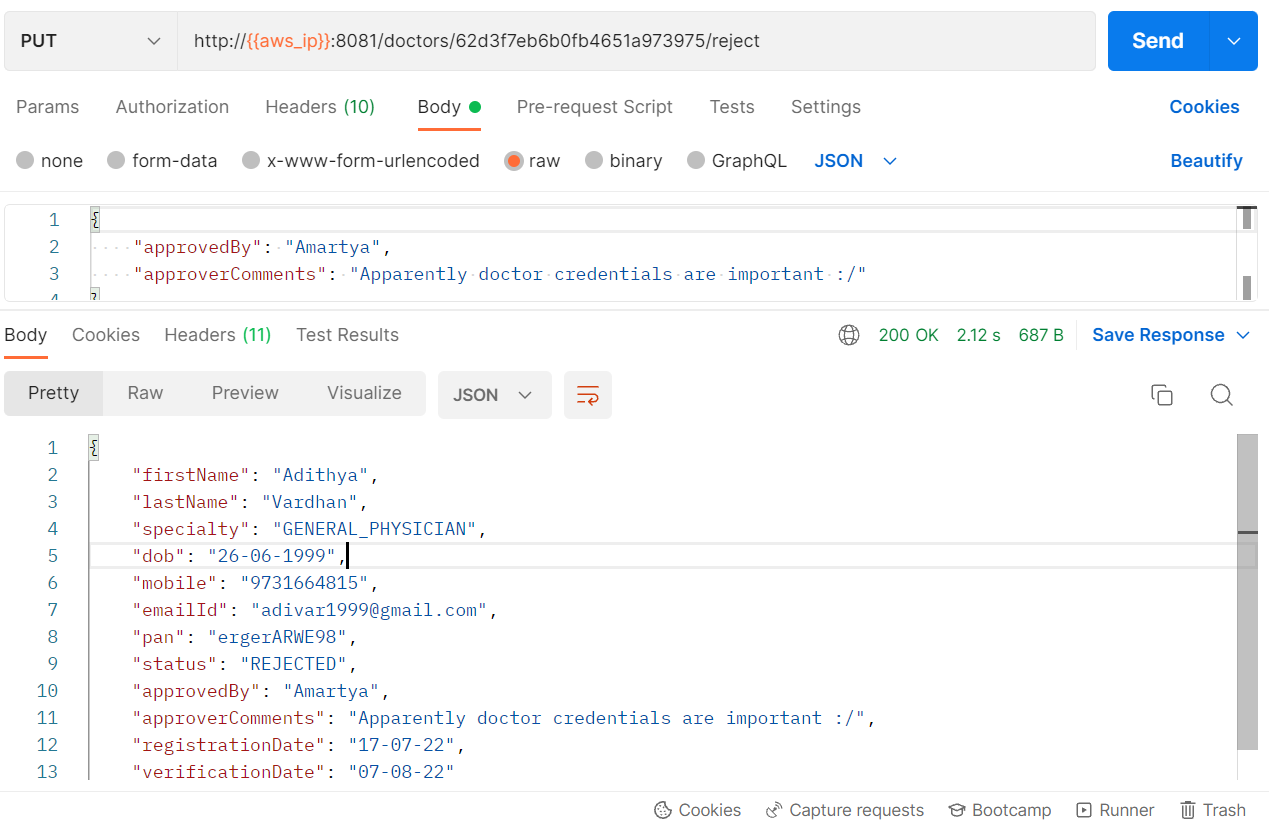


Fig 6.4b: Response to reject Doctor

##### Return a list of 20 Doctors sorted by rating

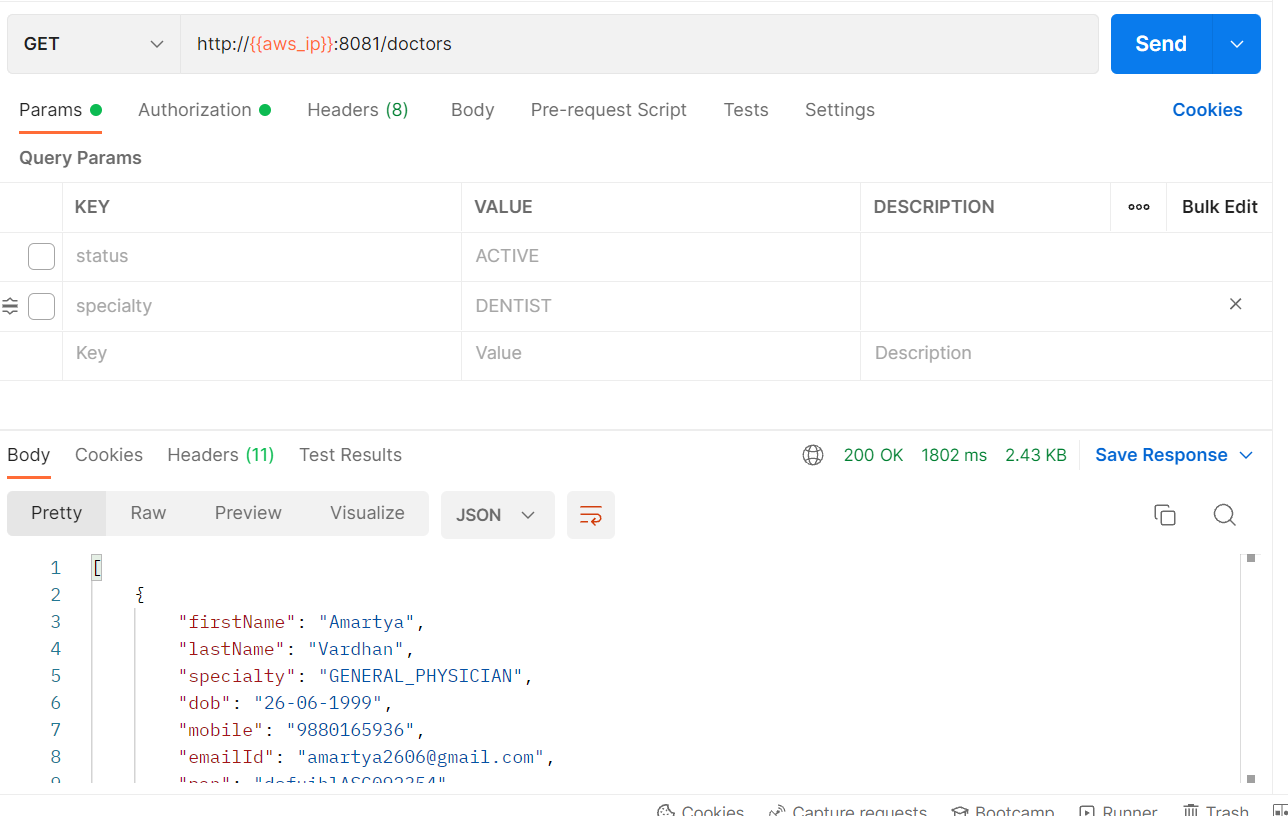


Fig 6.5a: Request and response for all doctors sorted by rating

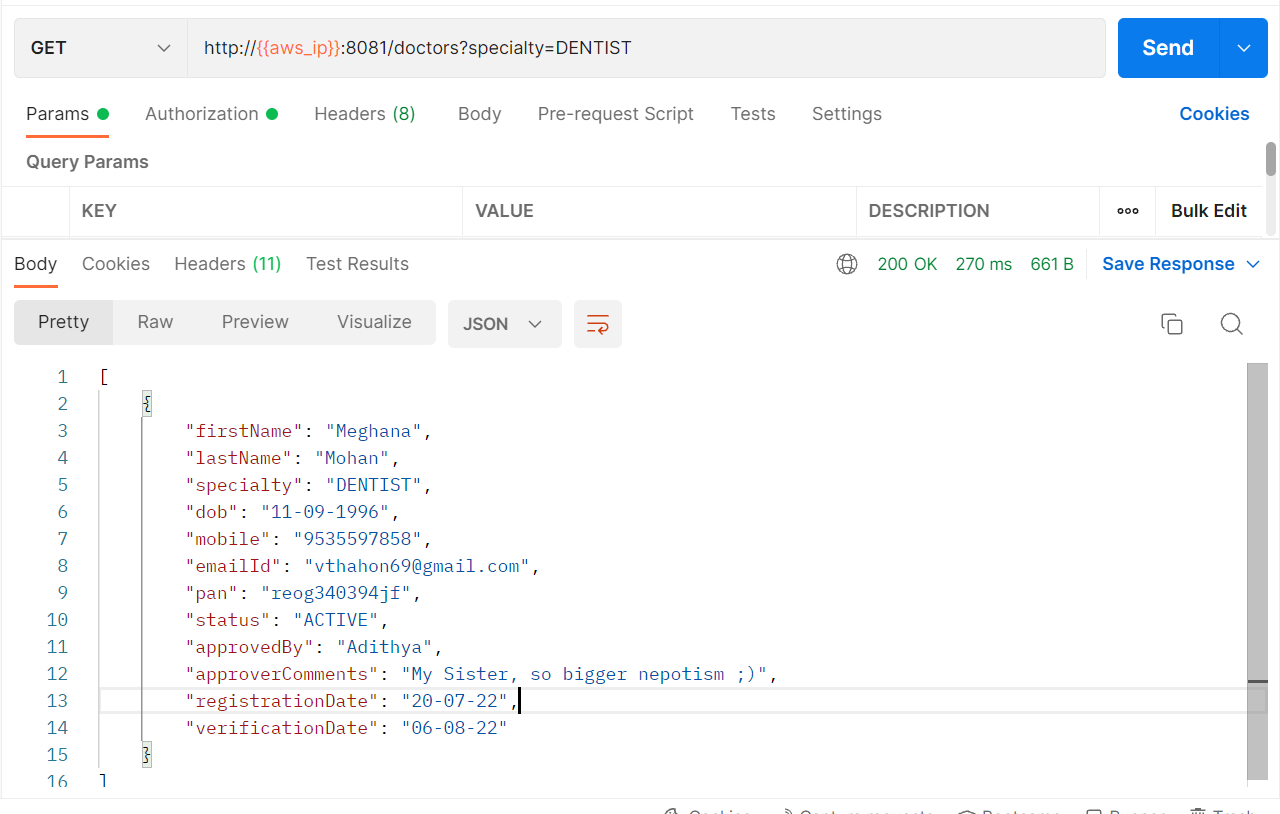


Fig 6.5b: Request and response for all doctors filtered by specialty

##### Return details of a Doctor by DoctorId

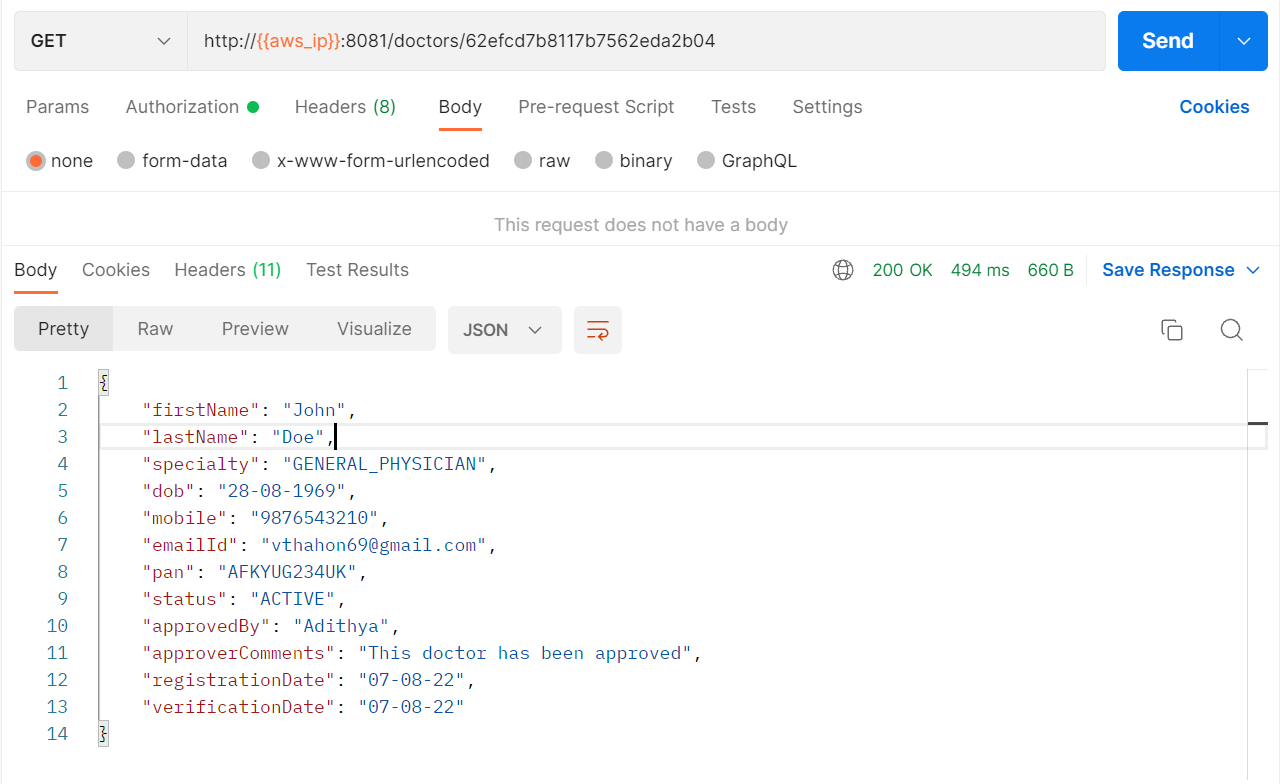


Fig 6.6: Request and response to get doctor details

#### UserService

##### Create User

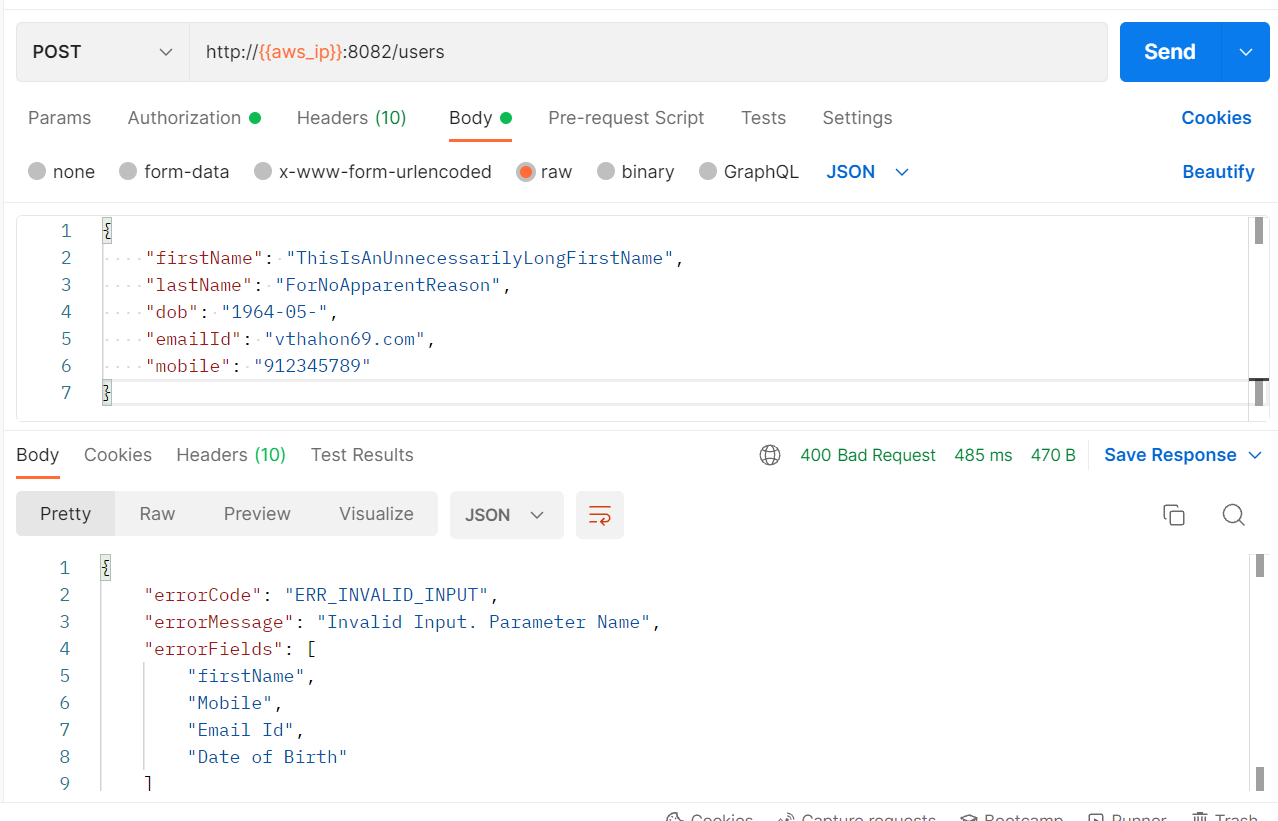


Fig 7.1a: Request and response for incorrect entries for Create User

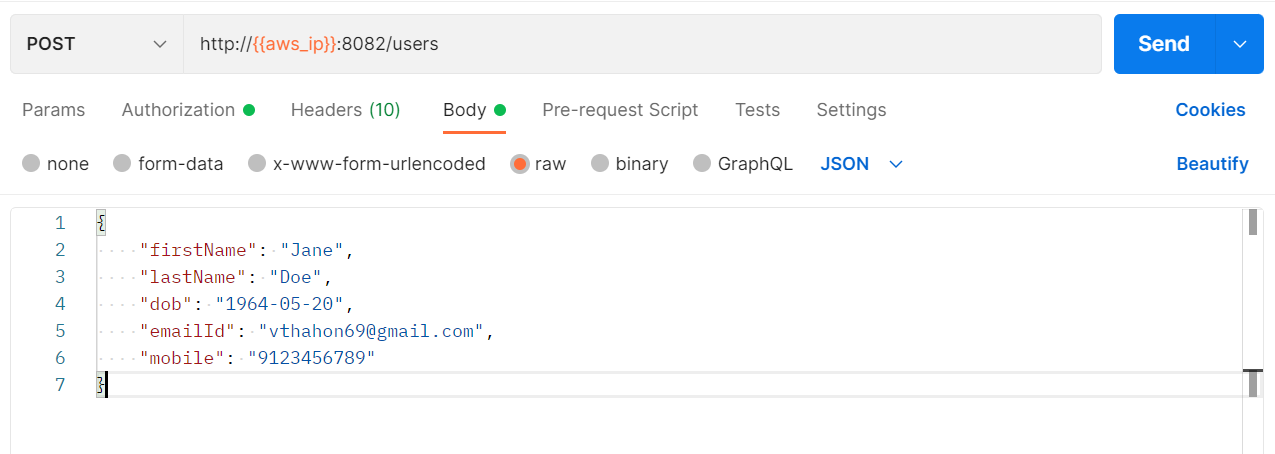


Fig 7.1b: Correct Request for Create User

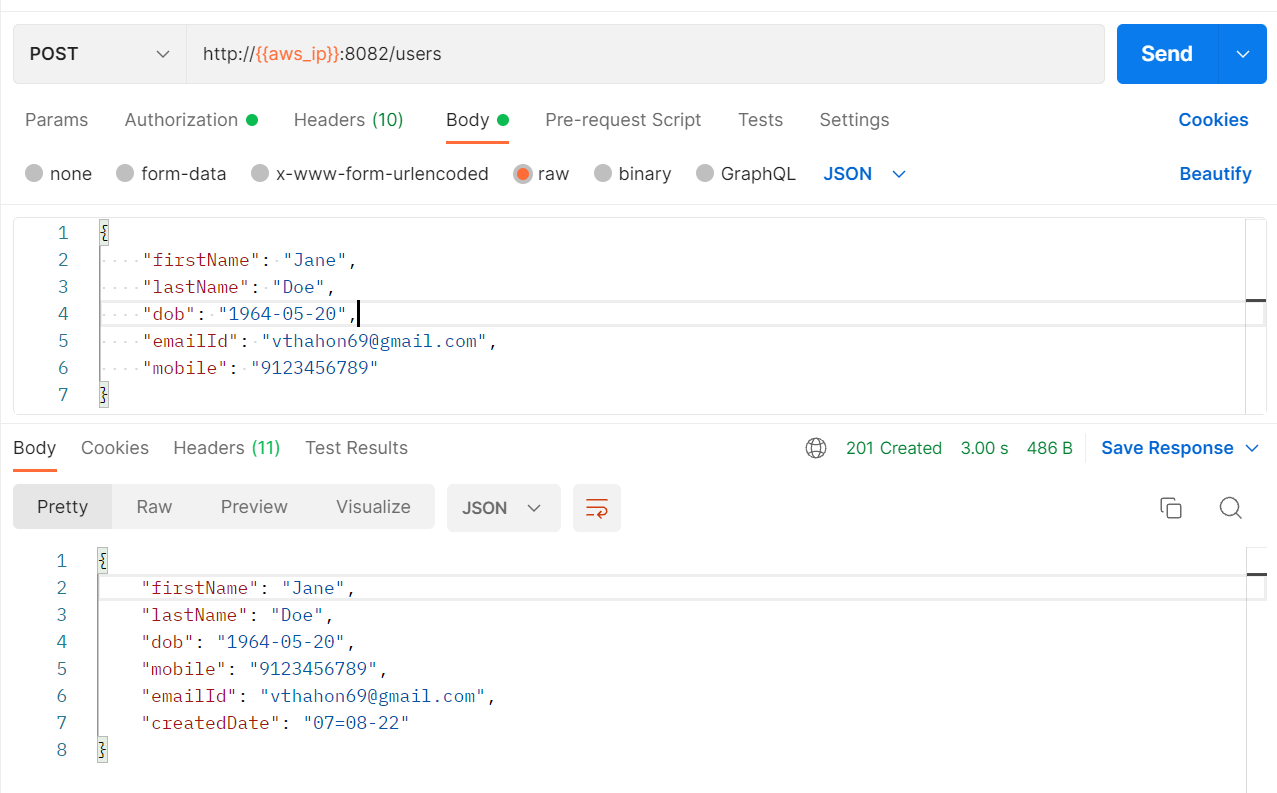


Fig 7.1c: Response for correct Create User

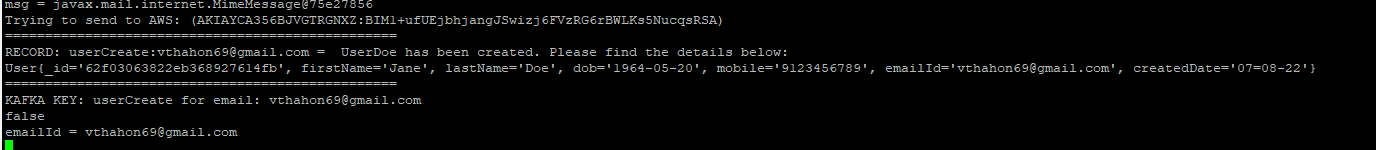


Fig7.1d: Notification in Kafka for creating user

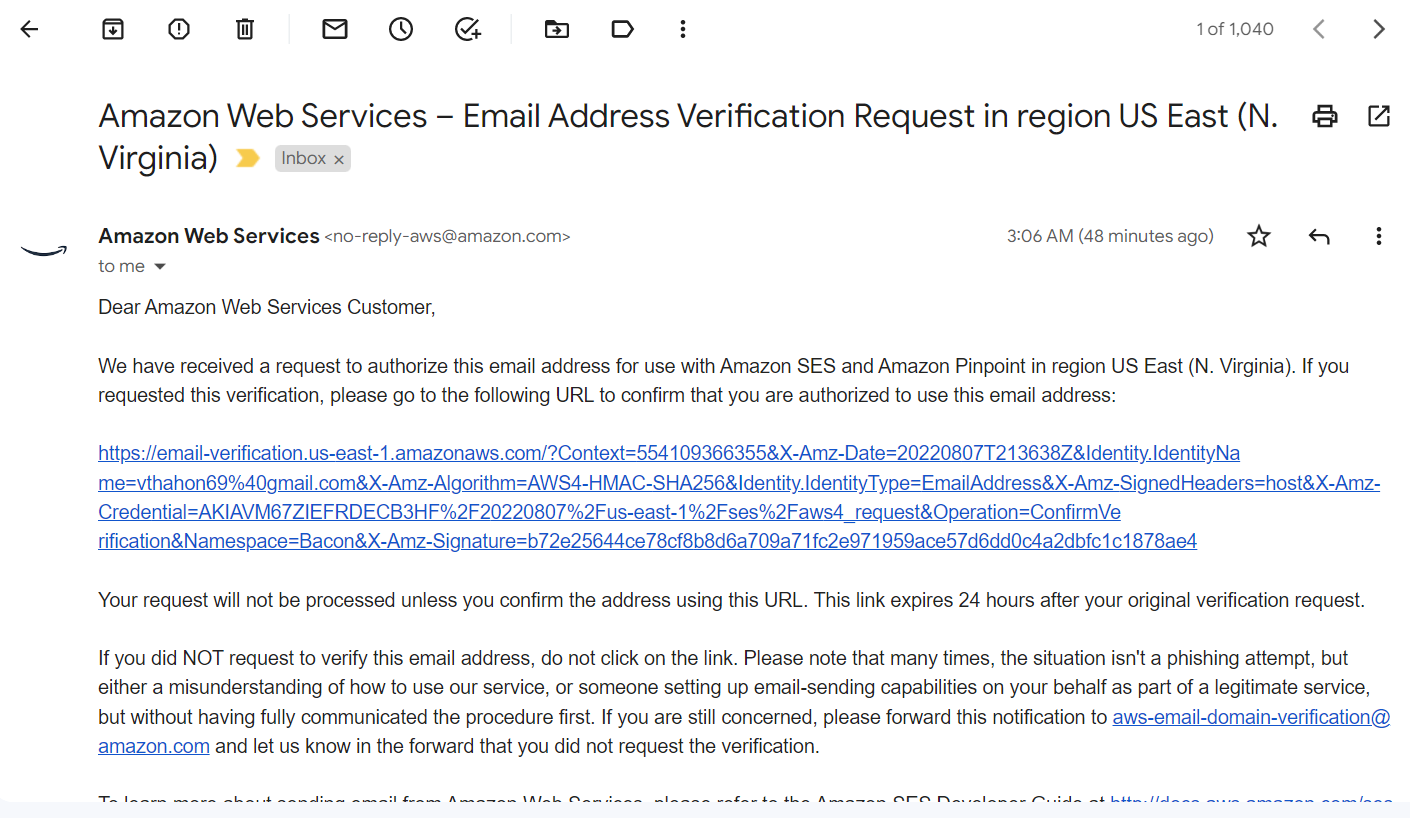


Fig 7.1e: Verification Email for Create user

##### getUser



Fig 7.2: Request and response for getUser

##### Upload Documents for User

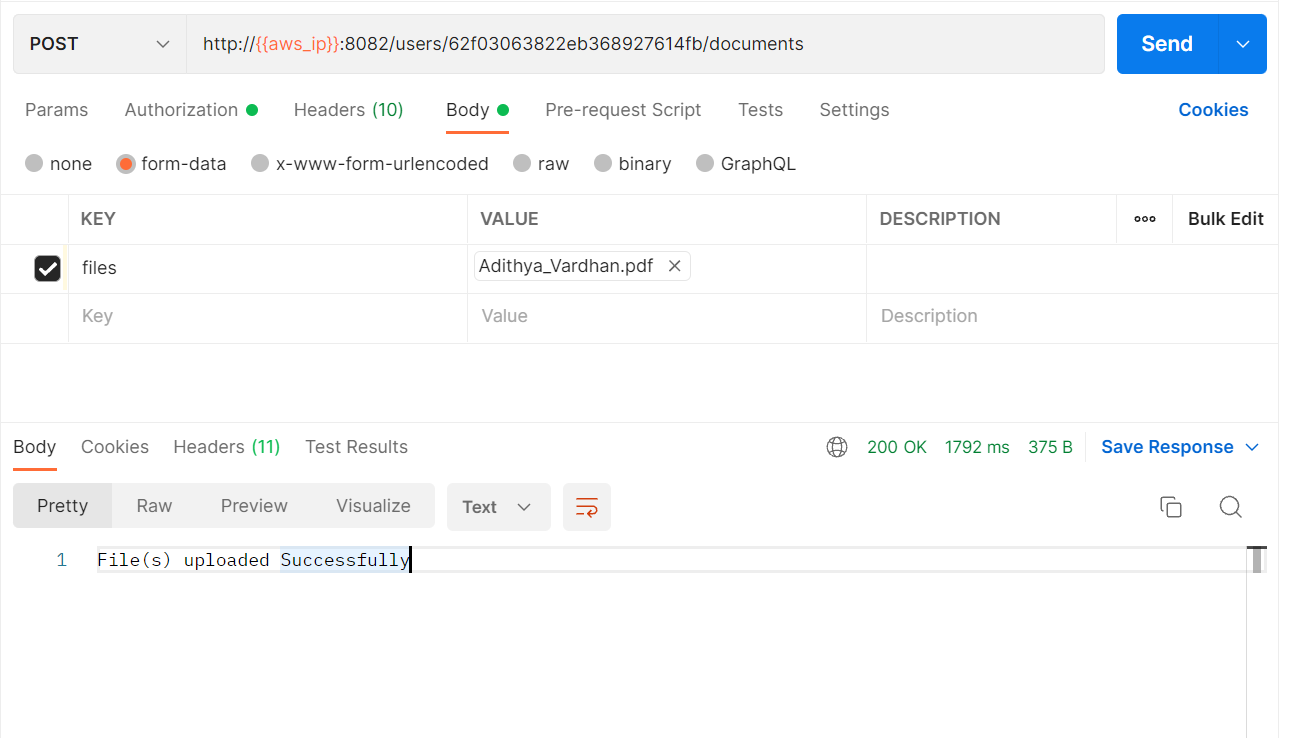


Fig 7.3a: Request and response for Upload Documents for User

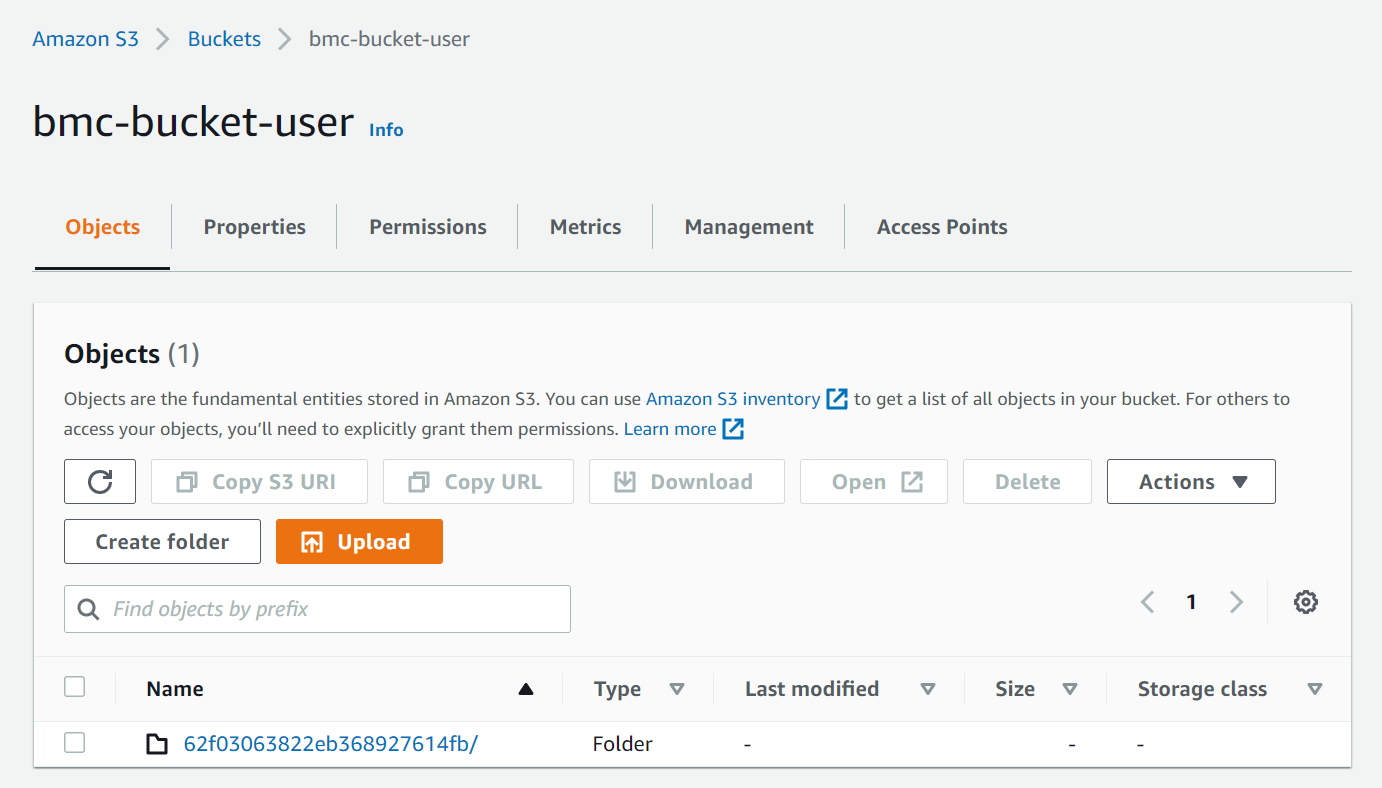


Fig 7.3b: File uploaded to Amazon S3

#### AppointmentService

##### createAvailability

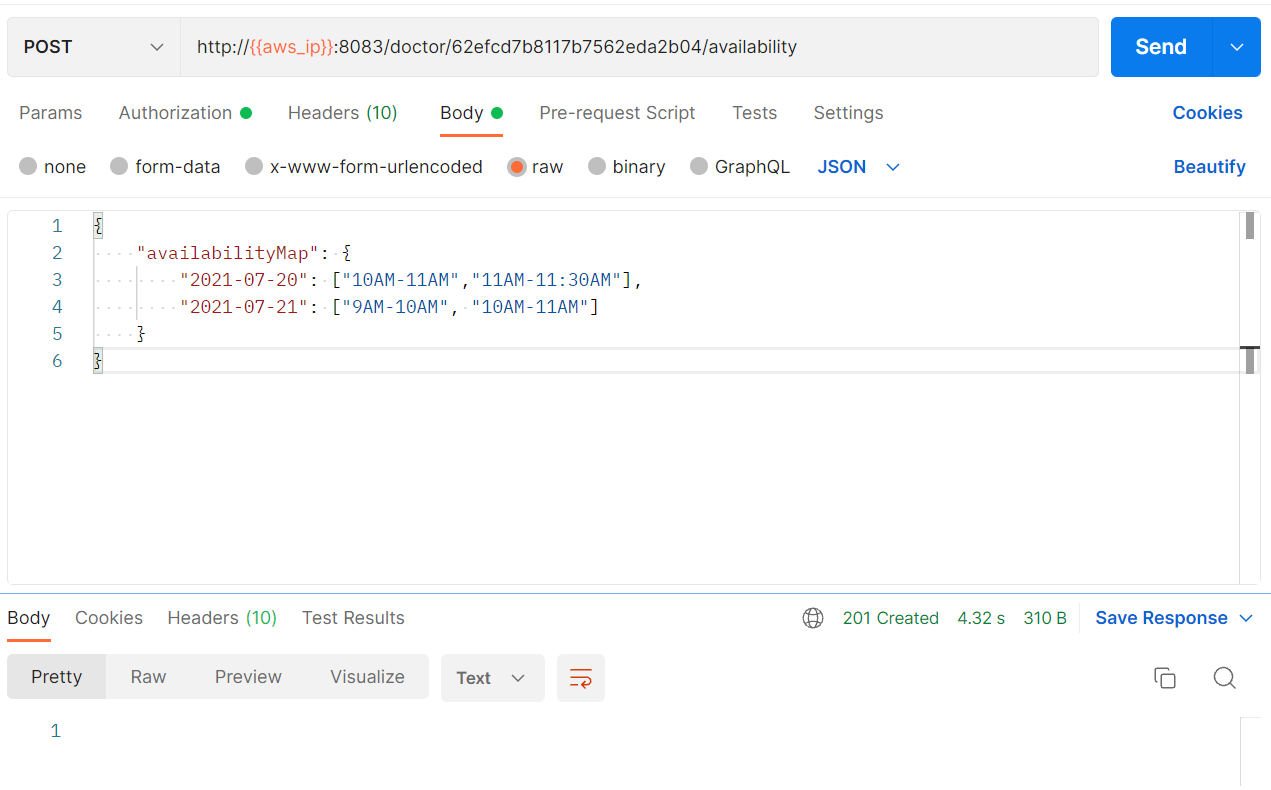


Fig 8.1: Request and response for setAvailability

##### getAvailability

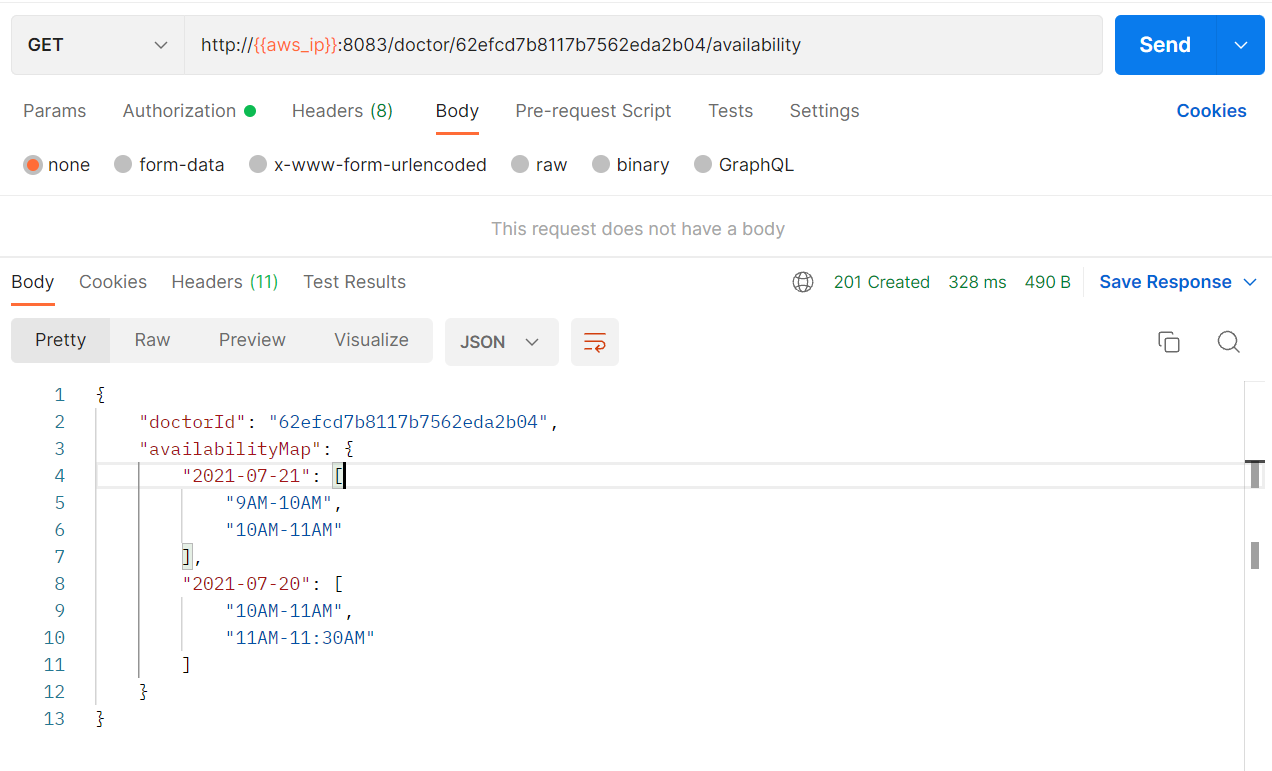


Fig 8.2: Request and response for getAvailability

##### createAppointment

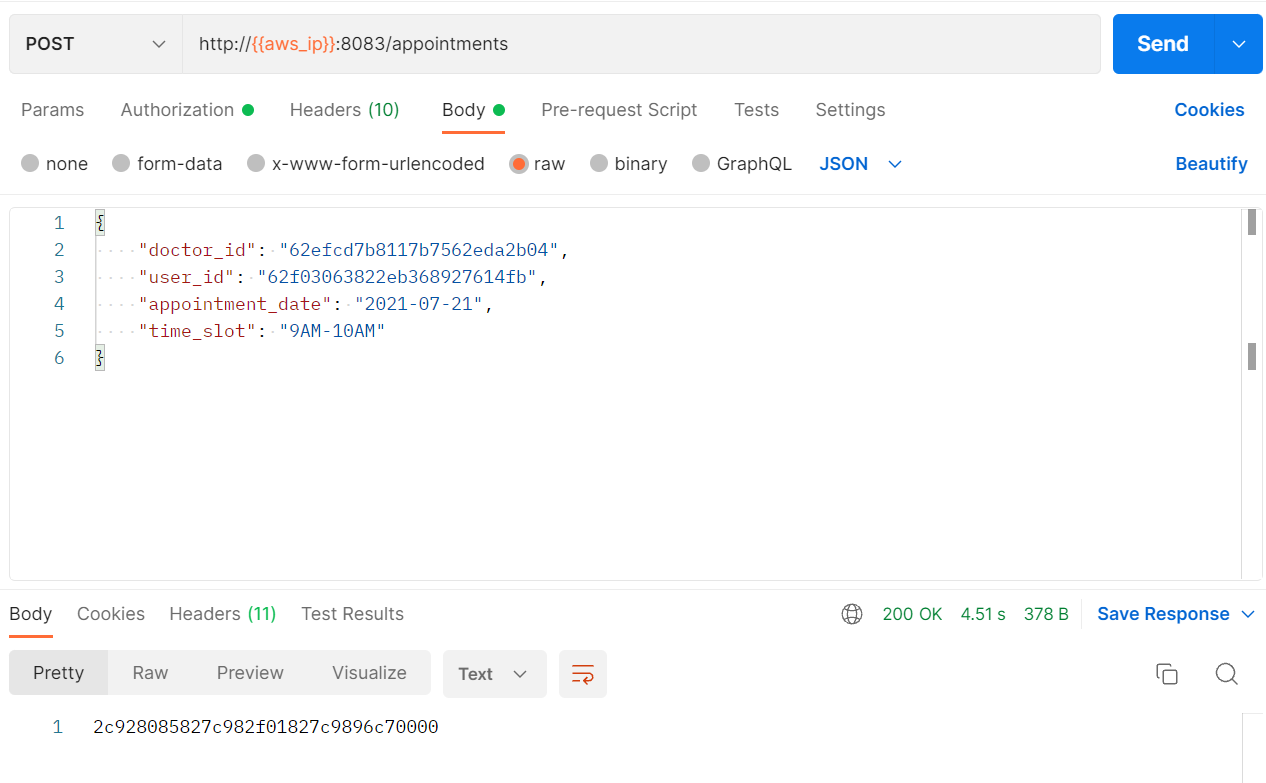


Fig 8.3a: Request and response for create Appointment

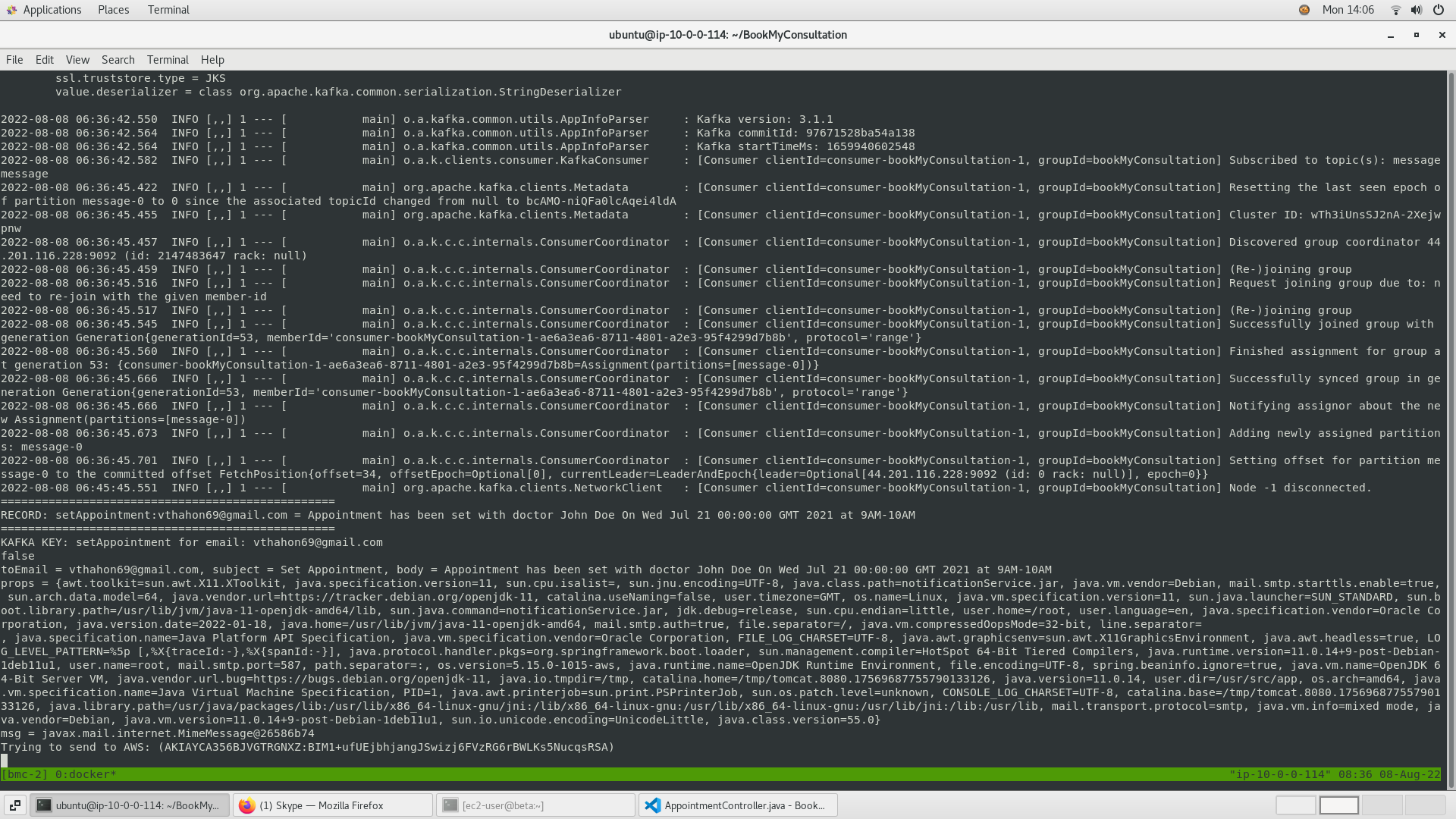


Fig 8.3b: Kafka Notification for create Appointment

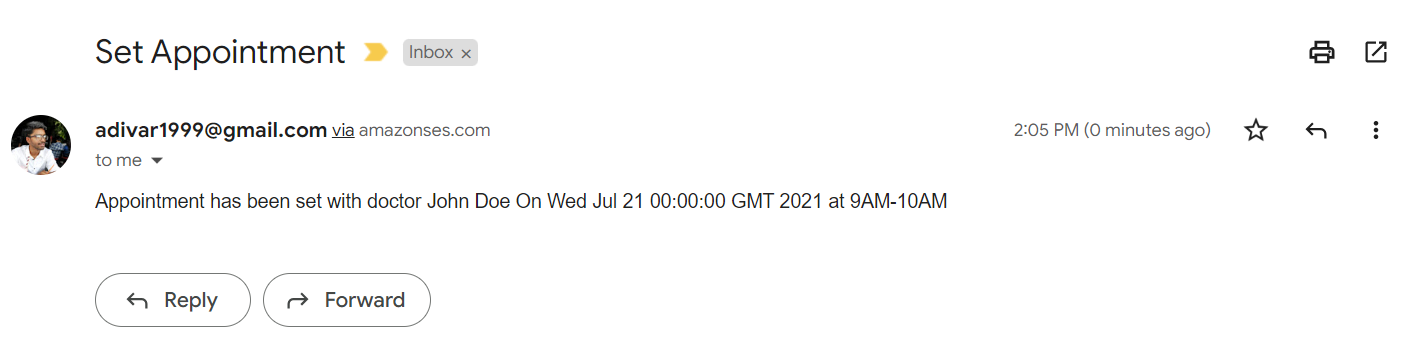


Fig 8.3c: Email confirmation for create Appointment

##### getAppointment

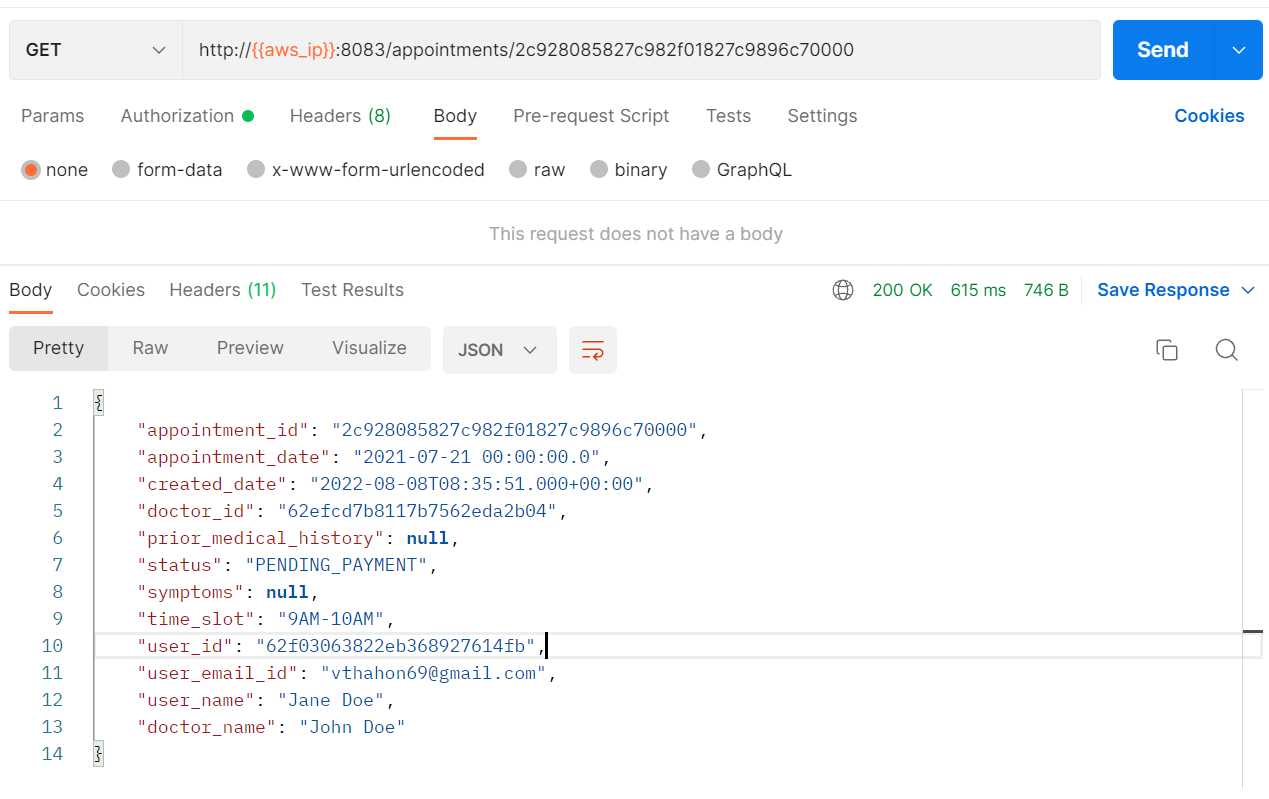


Fig 8.4: Request and response for get appointment

##### getAppointments

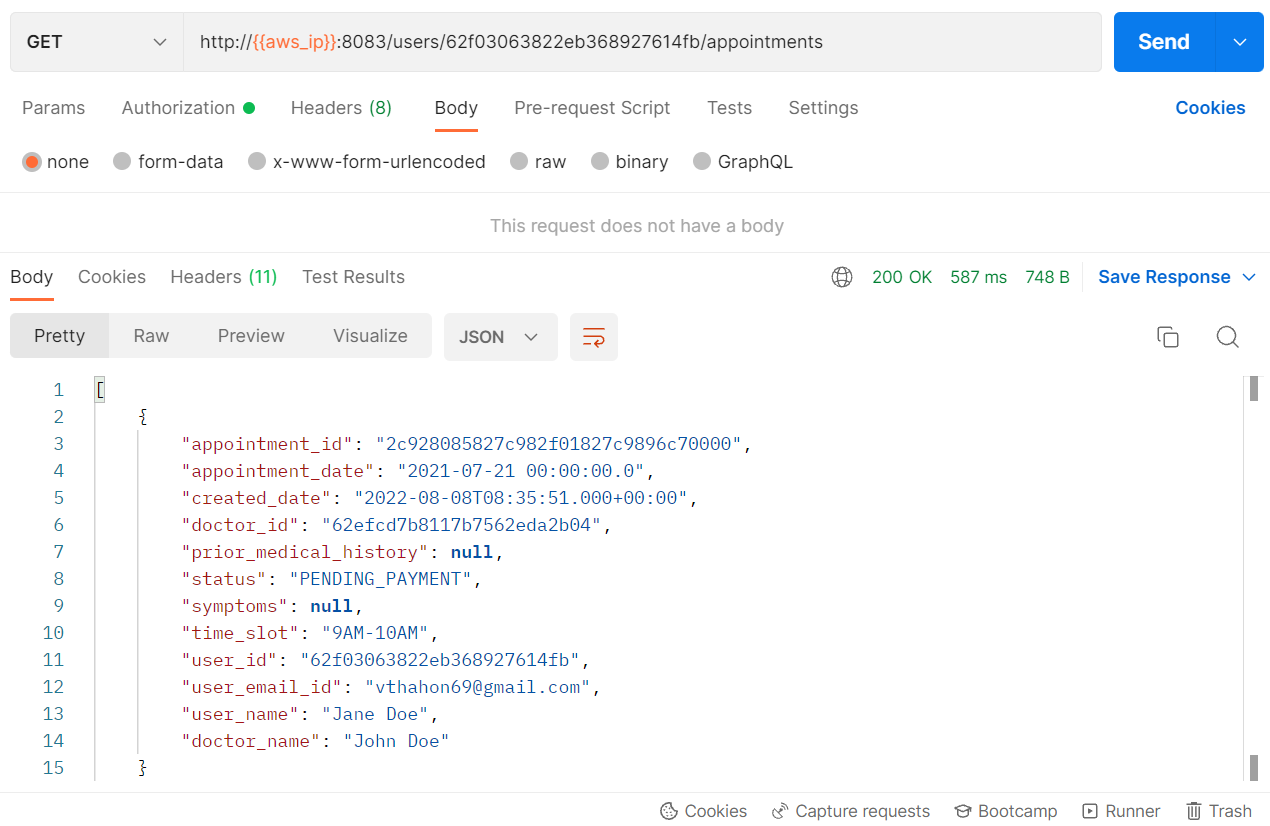


Fig 8.5: Request and response for get Appointments by user

##### createPrescription

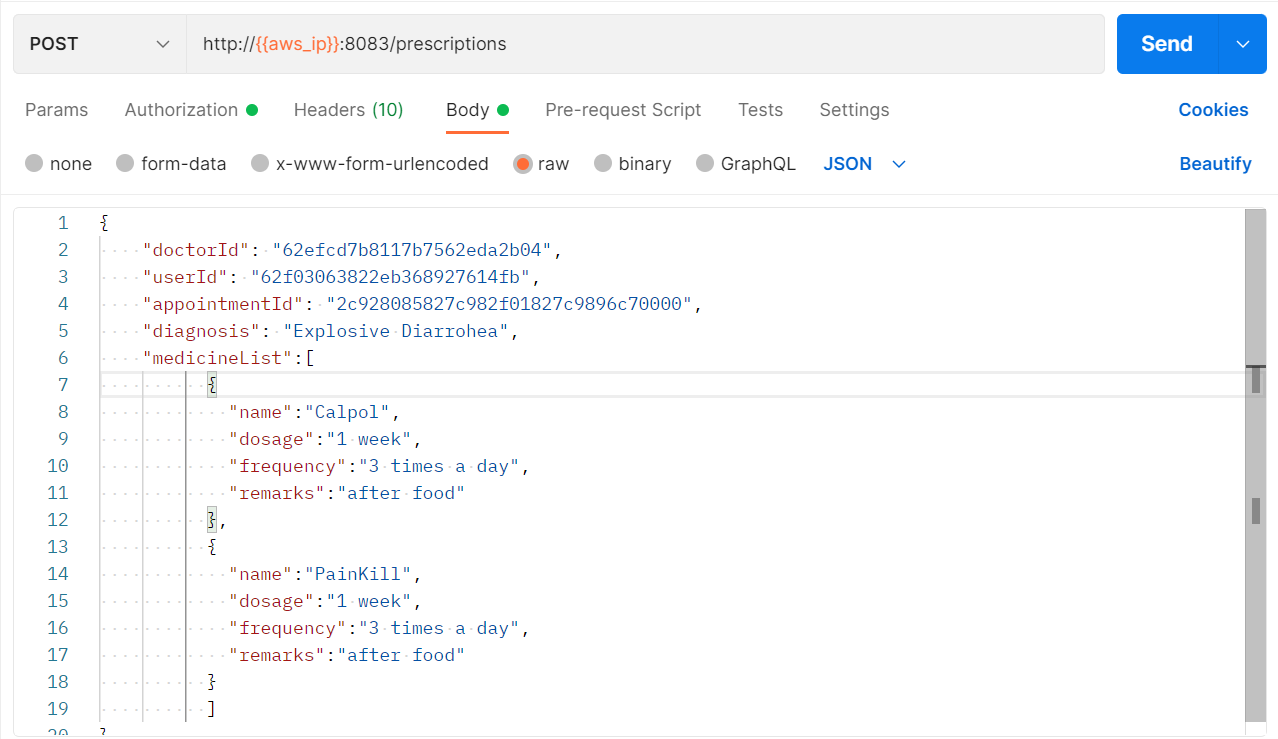


Fig 8.6a: Request for create Prescription

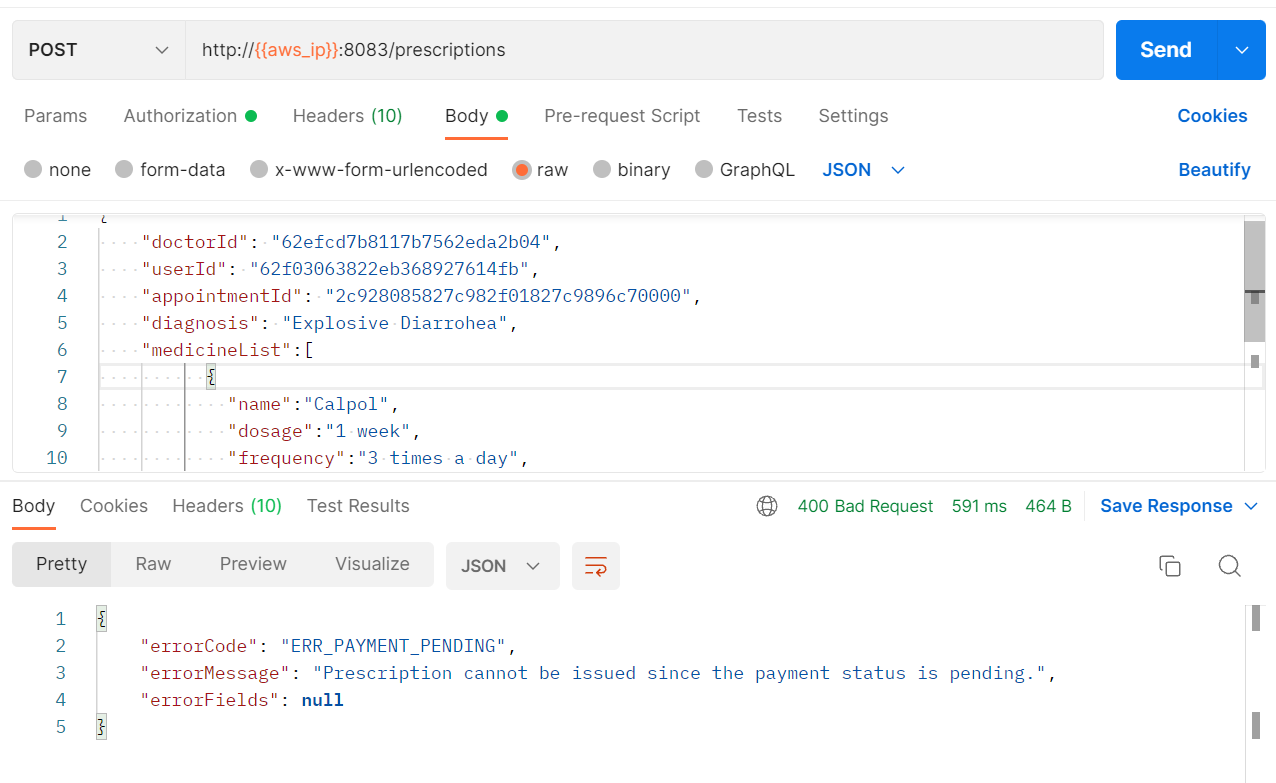


Fig 8.6b: Response when the Status of Appointment is PENDING

#### PaymentService

##### createPayment

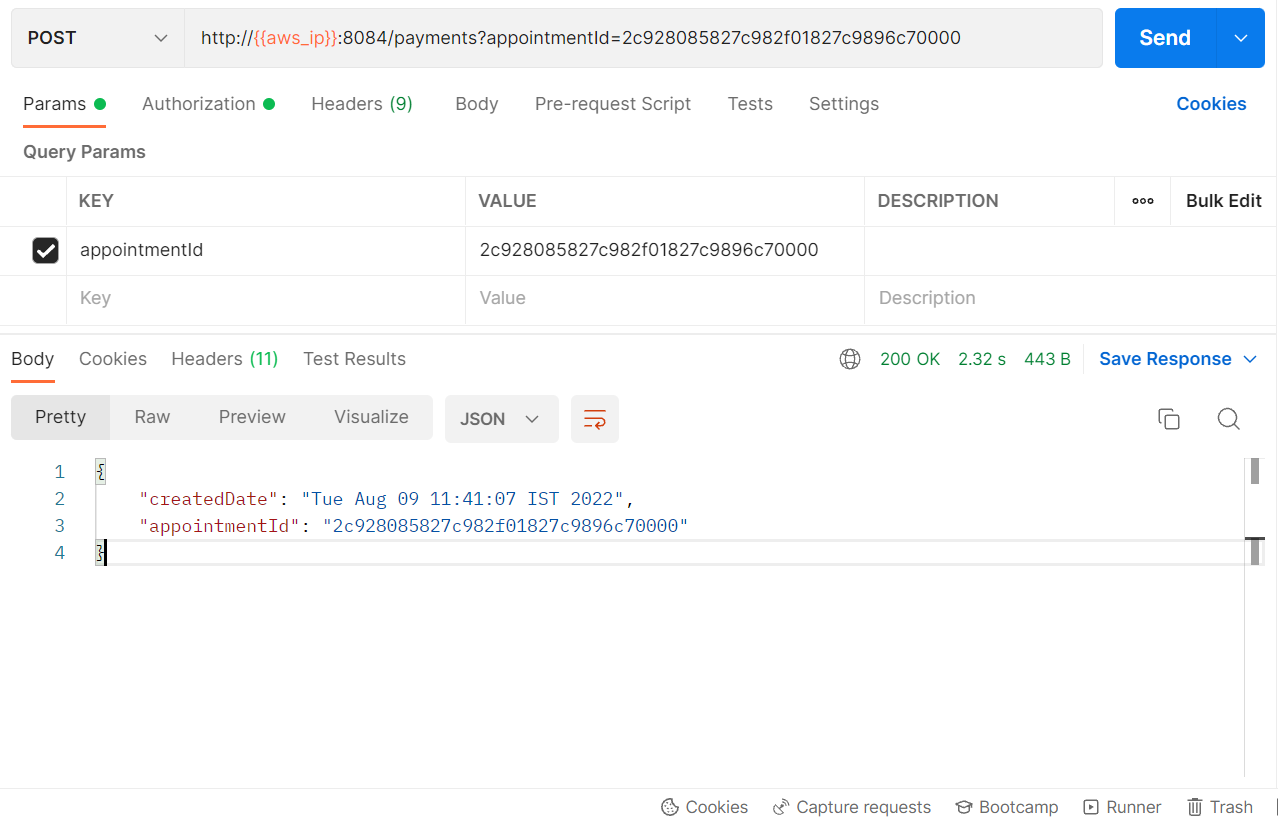


Fig 9.1: Request and response for create Payment

#### RatingService

##### createRating

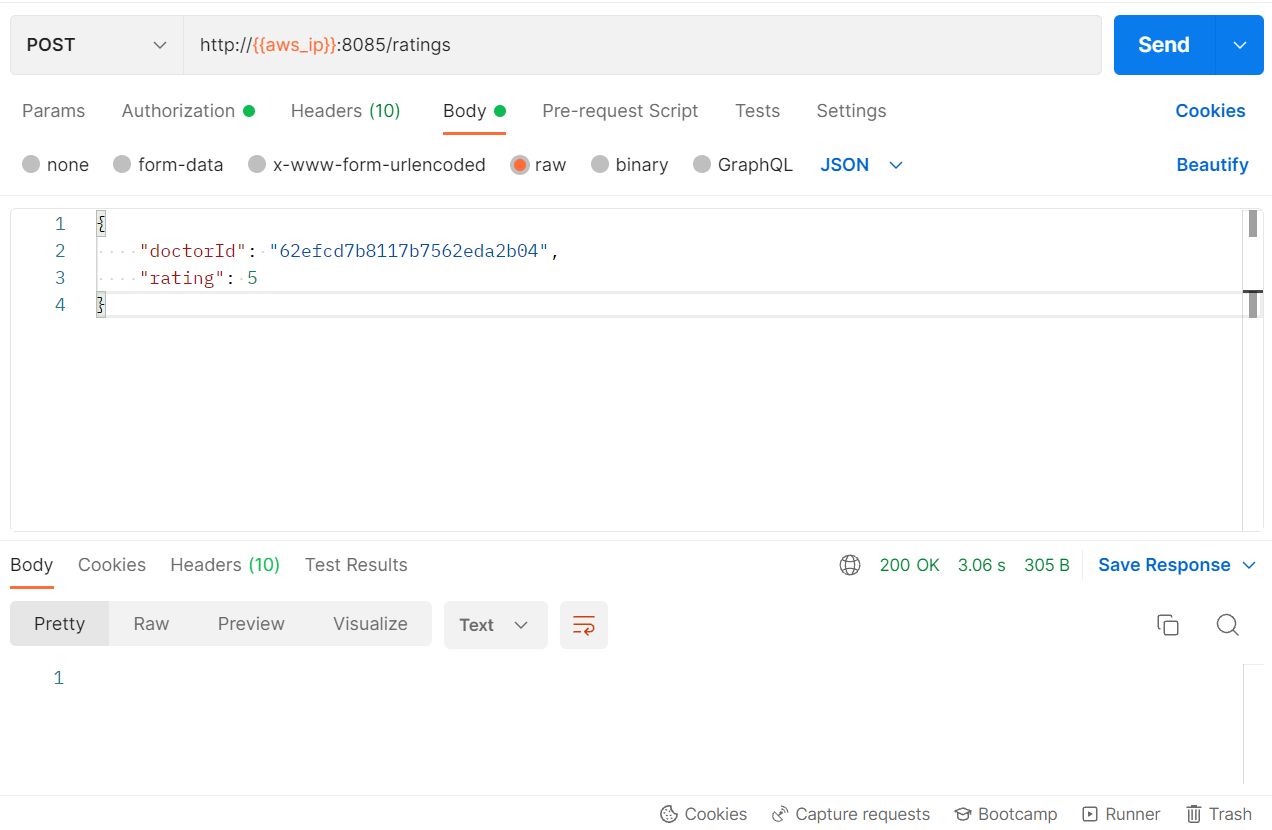


Fig 10.1: Request and response for creating Rating

#### NotificationService

The notification service uses Kafka to receive notifications sent from each service. For particular services, we receive confirmations to send emails using Amazon SES. The functionality has been implemented as shown in each of the required APIs.

The [***CreateDoctor***](#_xy9bcykvmcmp) and [***createUser***](#_wbod3ealisn1) API require SES to send a verification email to the respective email ID to confirm that the email exists.

The [***approveDoctor***](#_a0un6h5v36hr), [***rejectDoctor***](#_y6i4brrbdvnd), [***setAppointment***](#_2mahbi1nnswi) and [***setPrescription***](#_xy5kgsxh8be)APIs request SES to send custom messages to the respective email ID of the Doctor or the User.

Other services send kafka notifications to print by themselves, as shown in each API

#### Authentication

Authentication in the form of JWT token validation has been implemented in each of the services that contain API endpoints. To generate the JWT tokens, I used a separate project developed during the Security module.

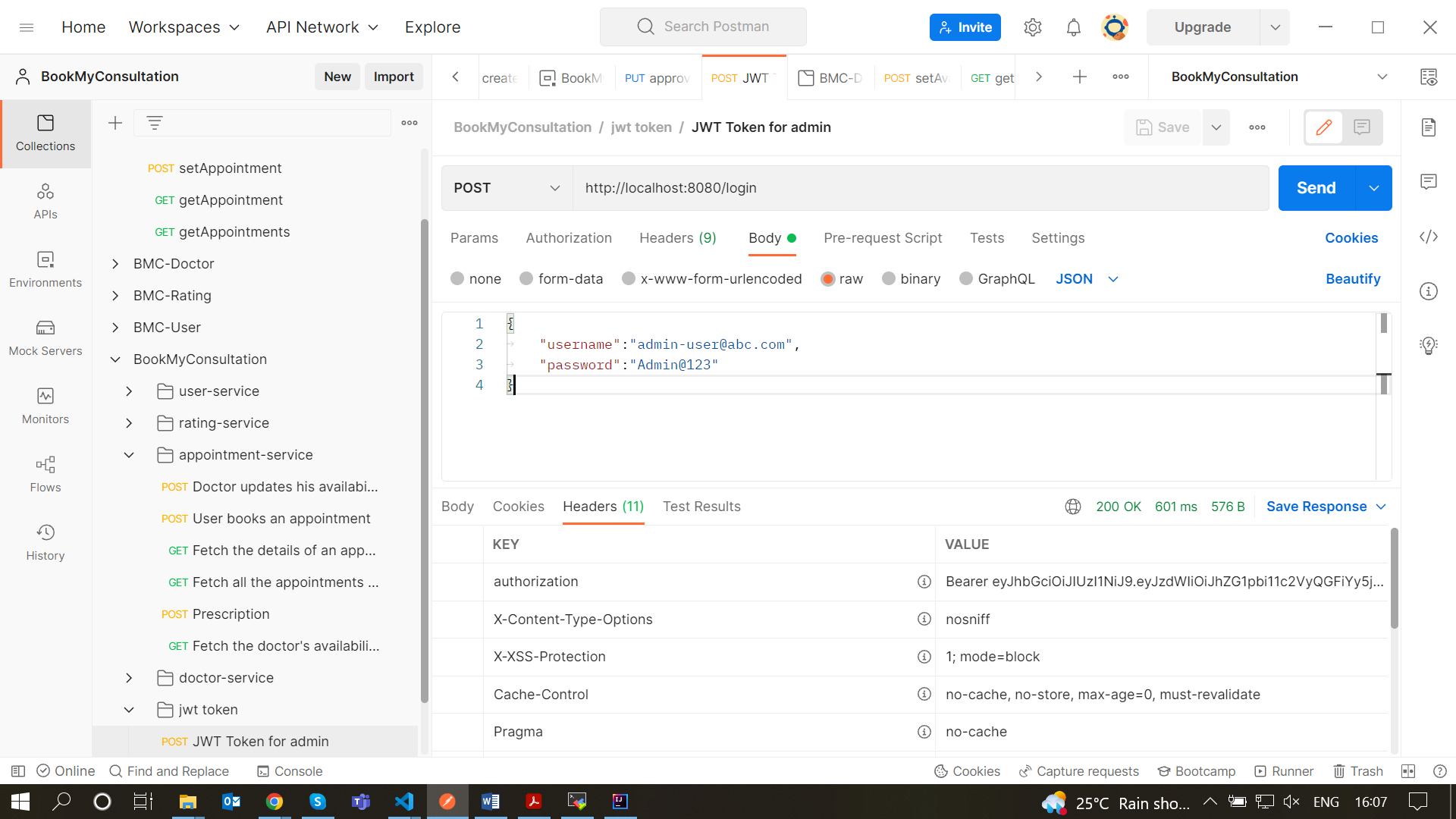


Fig 12.1: Request and response for Admin JWT token generation

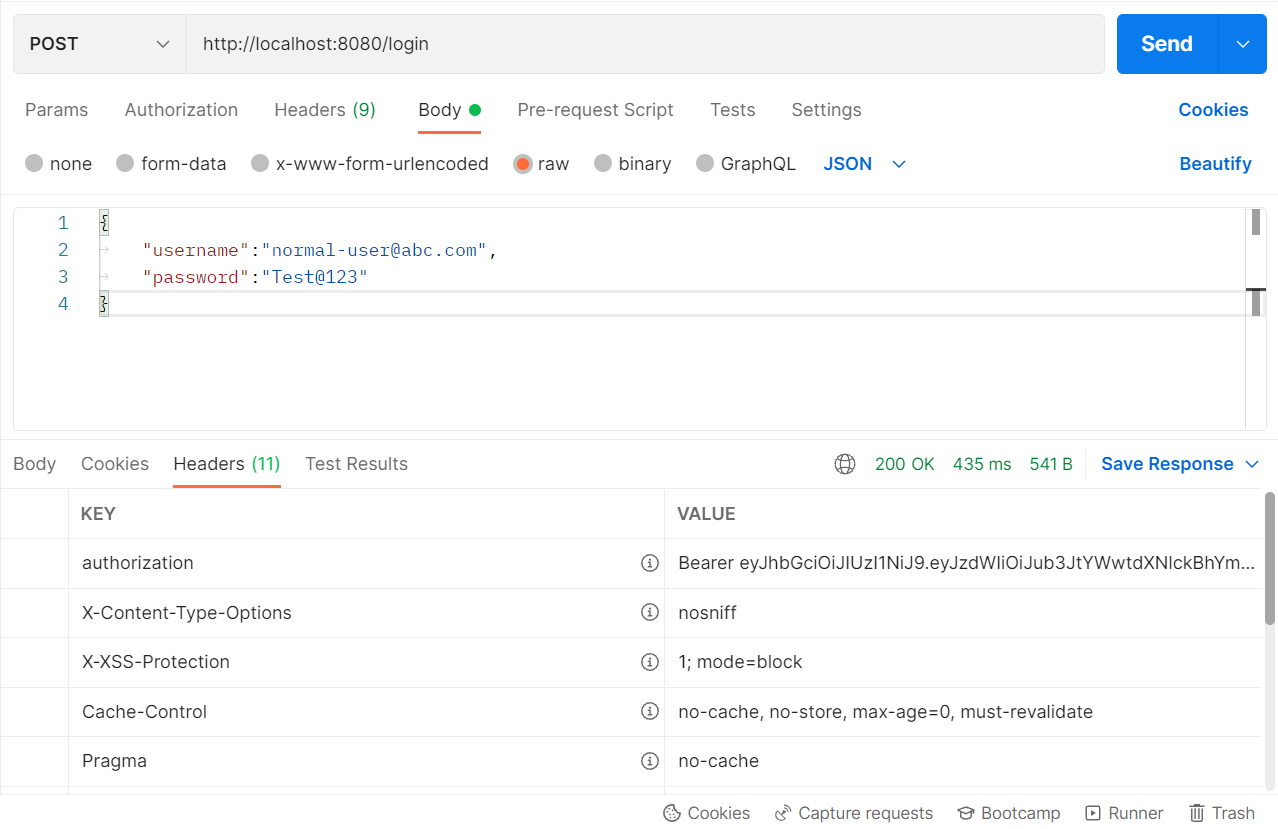


Fig 12.2: Request and response for User JWT token generation

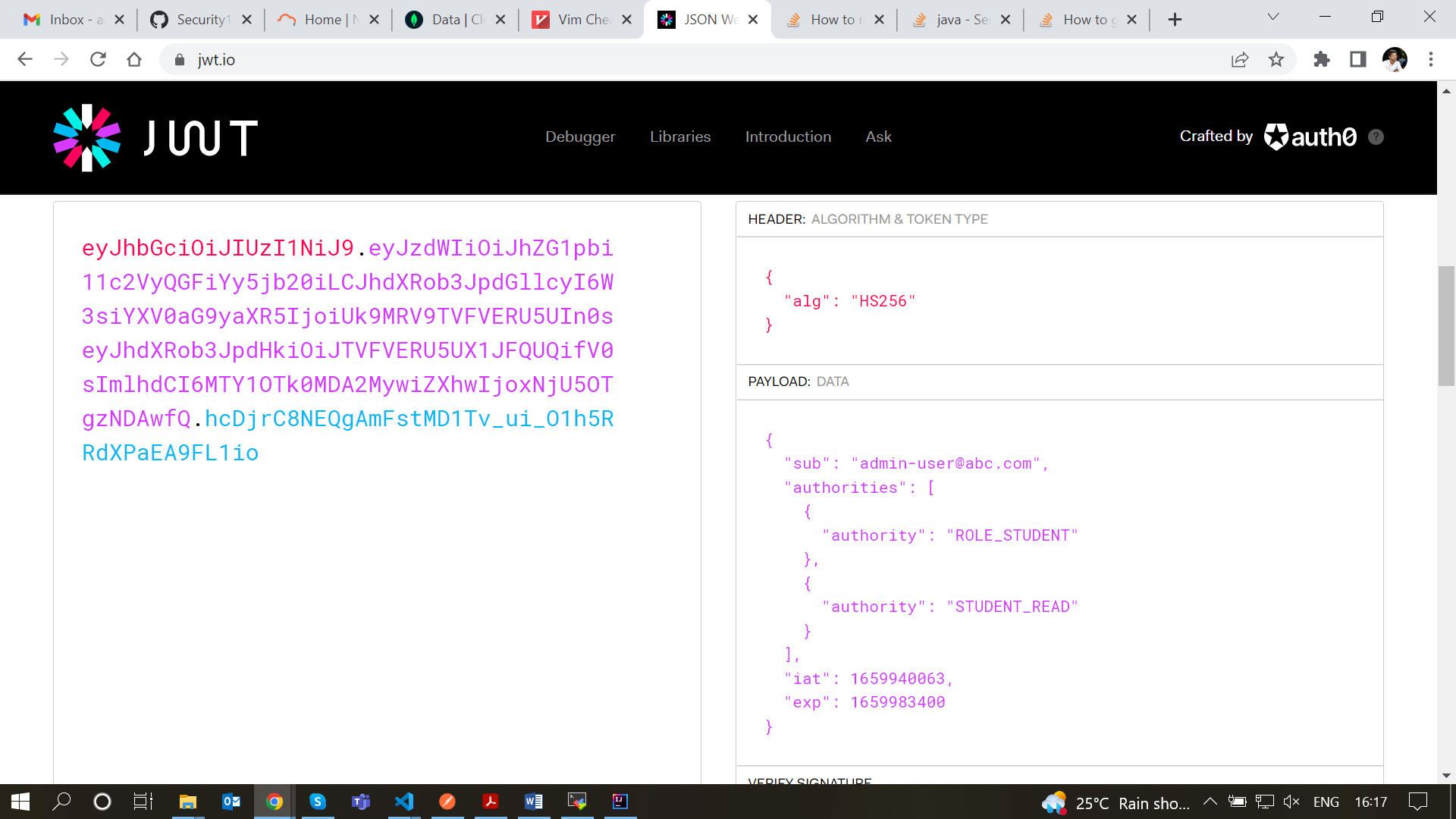


Fig 12.3: JWT token decrypted

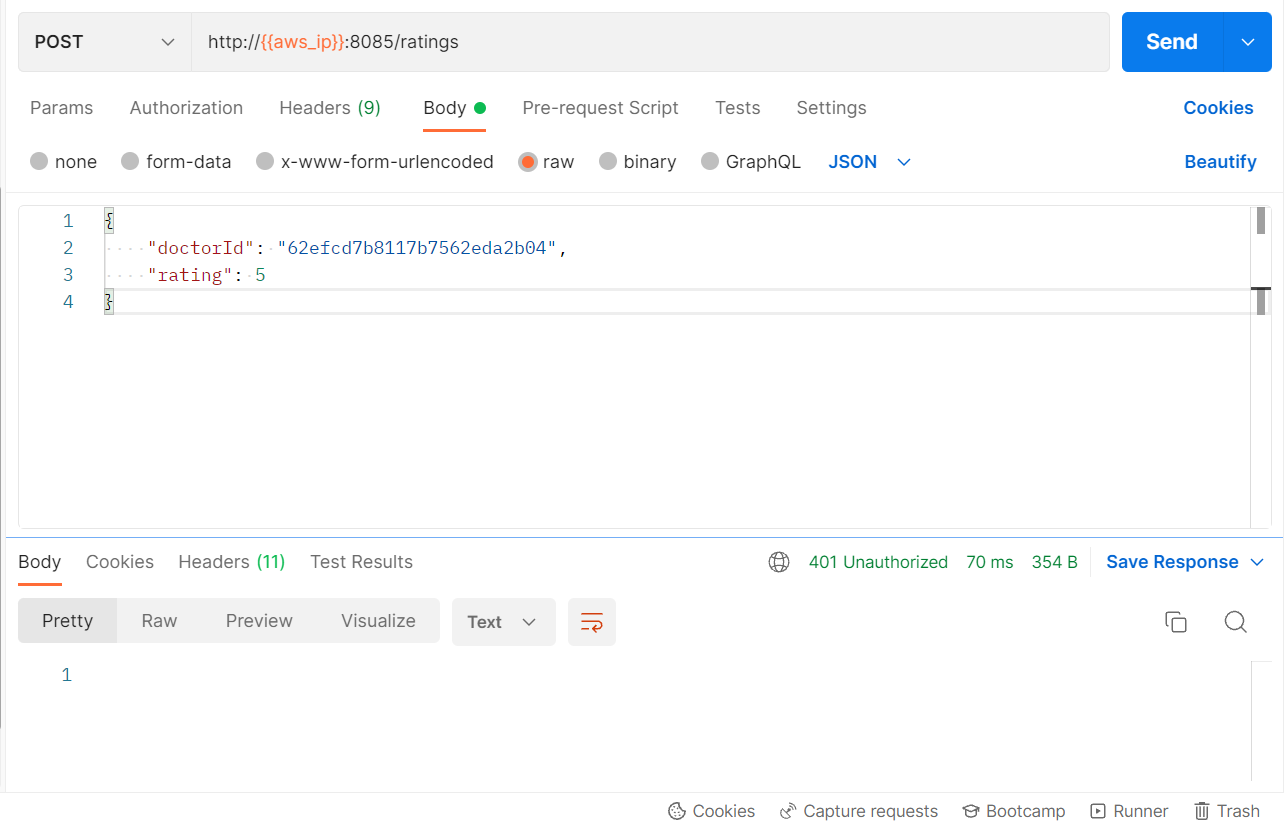


Fig 12.4: Response of an API without authorization

# Closing the Project

At the end, once we have implemented all the use cases of the application, we need to close all parts of the project.

We start by stopping all running instances of the various services running in our project, by the command



Fig 9.1: Stop running instances

Then we close and terminate the RDS instance that we have running on the AWS cloud as well the EC2 instance running the services and the Kafka server.