

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.

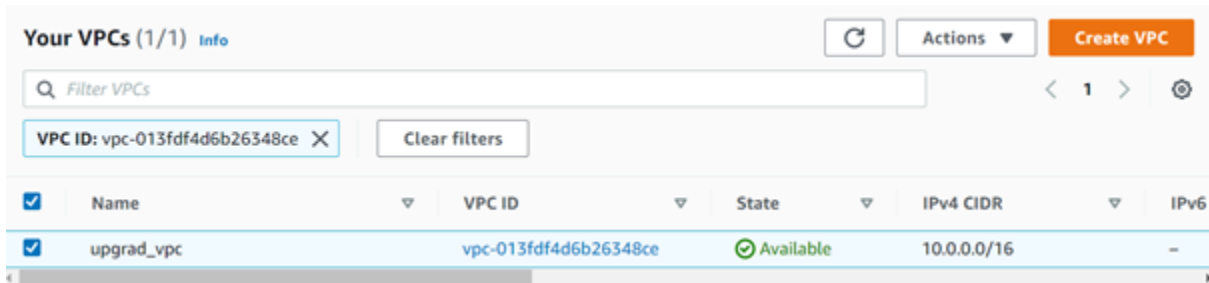


Fig 1.1: VPCs setup

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.

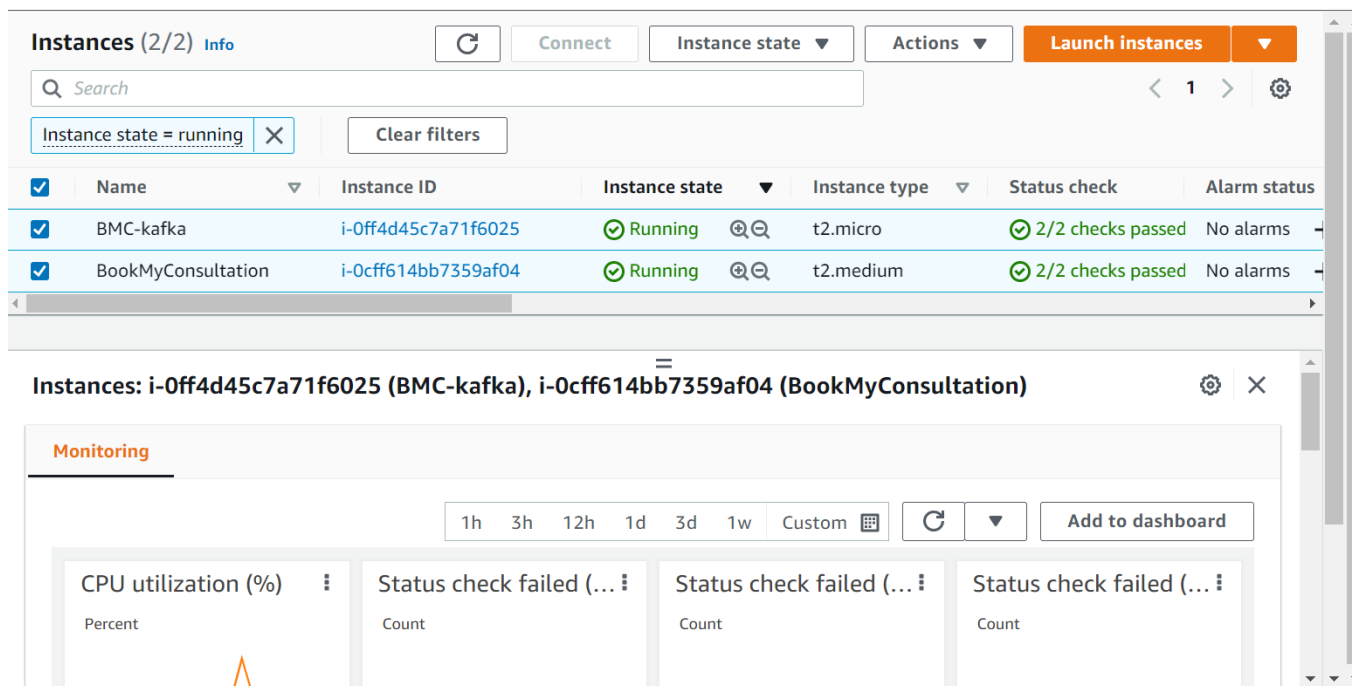
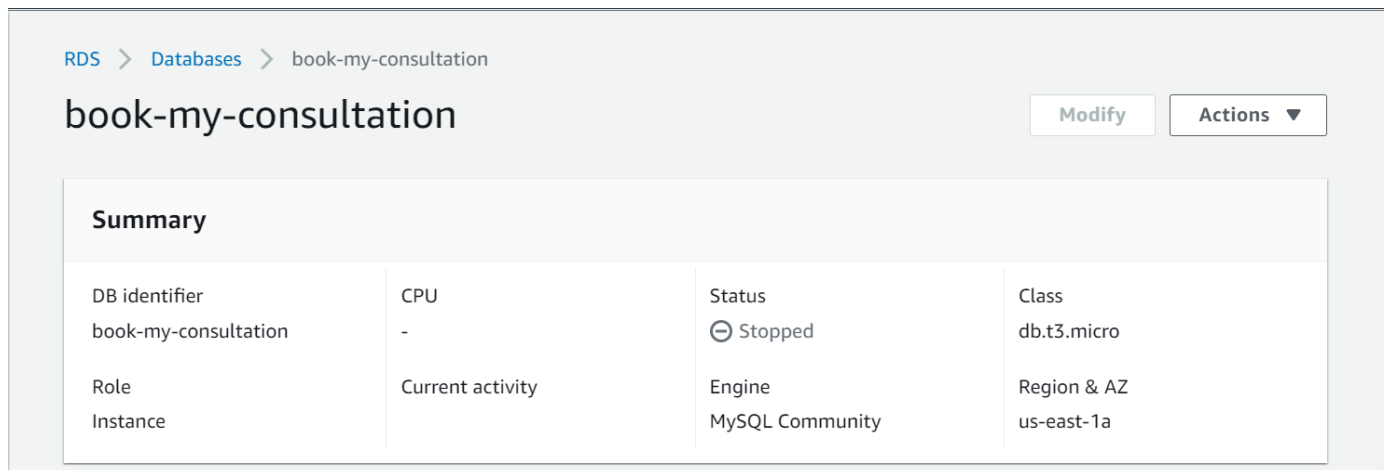


Fig 1.2: EC2 instances setup

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

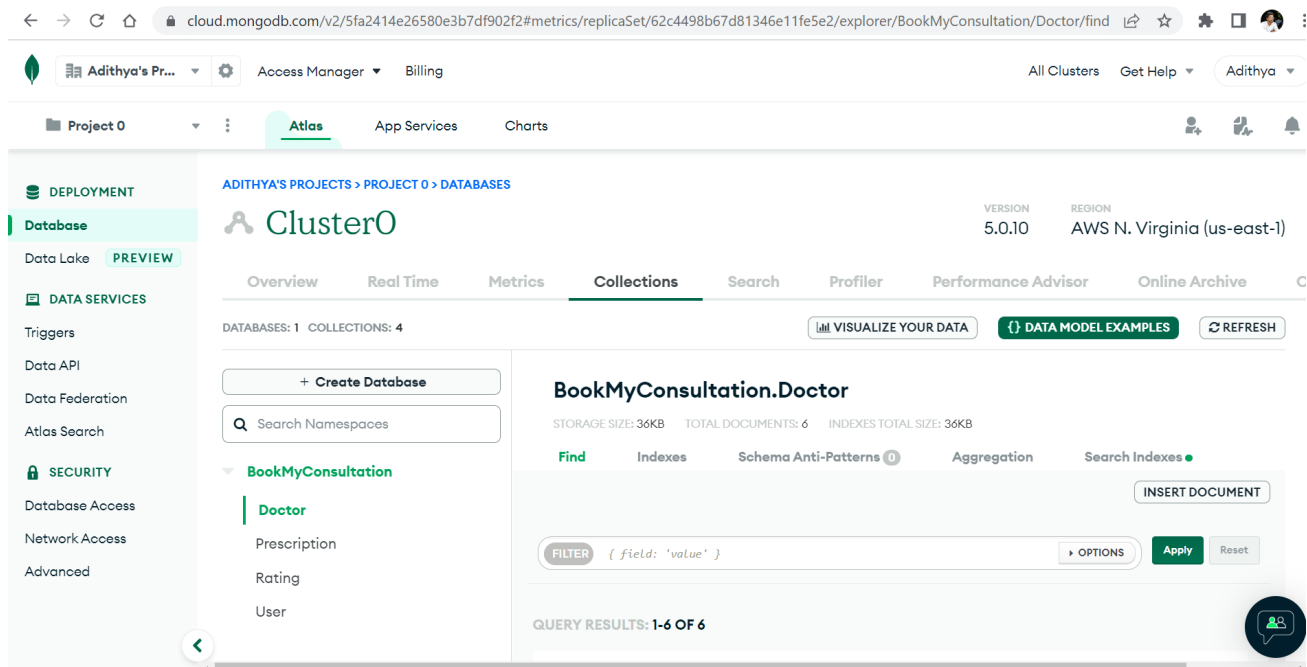


The screenshot shows the AWS RDS console for the instance 'book-my-consultation'. The breadcrumb navigation is 'RDS > Databases > book-my-consultation'. The instance name 'book-my-consultation' is displayed at the top with 'Modify' and 'Actions' buttons. Below is a 'Summary' section with a table of instance details.

Summary			
DB identifier book-my-consultation	CPU -	Status ⊖ Stopped	Class db.t3.micro
Role Instance	Current activity	Engine MySQL Community	Region & AZ us-east-1a

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.



The screenshot shows the MongoDB Atlas console. The breadcrumb navigation is 'ADITHYA'S PROJECTS > PROJECT 0 > DATABASES'. The main heading is 'Cluster0' with 'VERSION 5.0.10' and 'REGION AWS N. Virginia (us-east-1)'. The left sidebar shows 'DEPLOYMENT' and 'Database' (with 'PREVIEW' selected). The 'DATA SERVICES' section lists 'Triggers', 'Data API', 'Data Federation', and 'Atlas Search'. The 'SECURITY' section lists 'Database Access', 'Network Access', and 'Advanced'. The main content area shows 'Overview', 'Real Time', 'Metrics', 'Collections' (selected), 'Search', 'Profiler', 'Performance Advisor', and 'Online Archive'. Below this, it says 'DATABASES: 1 COLLECTIONS: 4'. A '+ Create Database' button and a 'Search Namespaces' input are present. The 'BookMyConsultation' database is expanded, showing collections: 'Doctor', 'Prescription', 'Rating', and 'User'. The 'Doctor' collection is selected, showing 'STORAGE SIZE: 36KB', 'TOTAL DOCUMENTS: 6', and 'INDEXES TOTAL SIZE: 36KB'. The 'Find' tab is active, showing a filter bar with '{ field: 'value' }' and an 'INSERT DOCUMENT' button. The bottom of the page shows 'QUERY RESULTS: 1-6 OF 6'.

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.

```
[ec2-user@beta ~]$ sudo ssh -i RHEL1.pem ubuntu@94.158.61.152
[sudo] password for ec2-user:
Welcome to Ubuntu 22.04 LTS (GNU/Linux 5.15.0-1011-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue Jun 28 12:21:15 UTC 2022

System load:  0.16357421875   Users logged in:      1
Usage of /:   61.5% of 7.58GB   IPv4 address for br-138ea6e59a72: 172.20.0.1
Memory usage: 51%             IPv4 address for docker0: 172.17.0.1
Swap usage:   0%              IPv4 address for eth0:   10.0.0.215
Processes:   146

21 updates can be applied immediately.
7 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

Last login: Tue Jun 28 07:28:29 2022 from 182.71.233.2
```

Fig 2.1a: Login to EC2 instance

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

```
Last login: Tue Jun 28 07:28:29 2022 from 182.71.233.2
ubuntu@ip-10-0-0-215:~$ docker -v
Docker version 20.10.17, build 100c701
ubuntu@ip-10-0-0-215:~$ docker-compose -v
docker-compose version 1.29.2, build unknown
ubuntu@ip-10-0-0-215:~$
```

Fig 2.1b: Docker versions

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

```
ubuntu@ip-10-0-0-114: ~/BookMyConsultation
ubuntu@ip-10-0-0-114:~$ cd BookMyConsultation/
ubuntu@ip-10-0-0-114:~/BookMyConsultation$ ls -ltr
total 32
-rw-rw-r-- 1 ubuntu ubuntu 2667 Aug  5 14:21 docker-compose.yml
-rw-rw-r-- 1 ubuntu ubuntu  55 Aug  5 14:21 README.md
drwxrwxr-x 4 ubuntu ubuntu 4096 Aug  5 14:21 userservice
drwxrwxr-x 4 ubuntu ubuntu 4096 Aug  5 14:21 ratingservice
drwxrwxr-x 4 ubuntu ubuntu 4096 Aug  5 14:21 paymentservice
drwxrwxr-x 4 ubuntu ubuntu 4096 Aug  5 14:21 notificationservice
drwxrwxr-x 4 ubuntu ubuntu 4096 Aug  5 14:21 doctorservice
drwxrwxr-x 4 ubuntu ubuntu 4096 Aug  5 14:21 appointmentservice
ubuntu@ip-10-0-0-114:~/BookMyConsultation$
```

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.

```
[ec2-user@ip-10-0-0-198 ~]$ java -version
openjdk version "11.0.13" 2021-10-19 LTS
OpenJDK Runtime Environment 18.9 (build 11.0.13+8-LTS)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.13+8-LTS, mixed mode, sharing)
[ec2-user@ip-10-0-0-198 ~]$ ls
kafka_2.13-2.8.1  kafka_2.13-2.8.1.tgz
[ec2-user@ip-10-0-0-198 ~]$
```

Fig 2.2: Kafka server setup

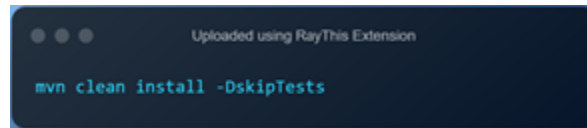
Implementation

Creating Dockerfiles

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

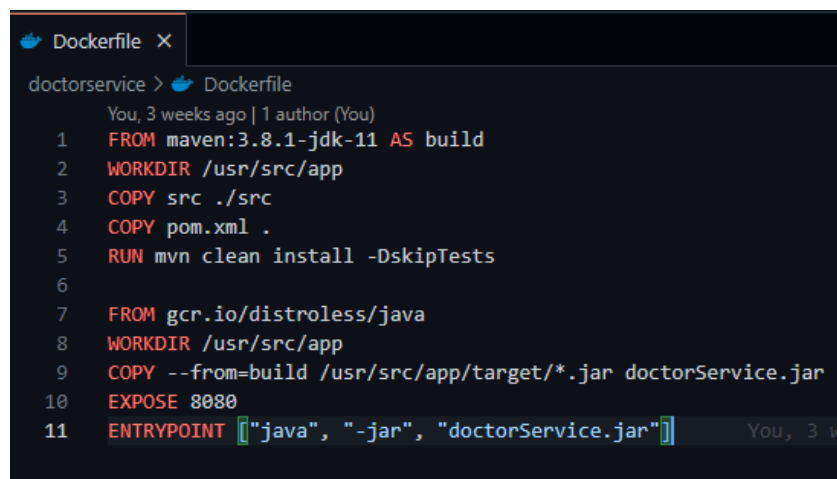
Each component needs to be compiled and pushed into a jar file, and that jar file needs to be uploaded into a java image for running.

To be compiled, must navigate into each service folder and run the following command to generate the jar file



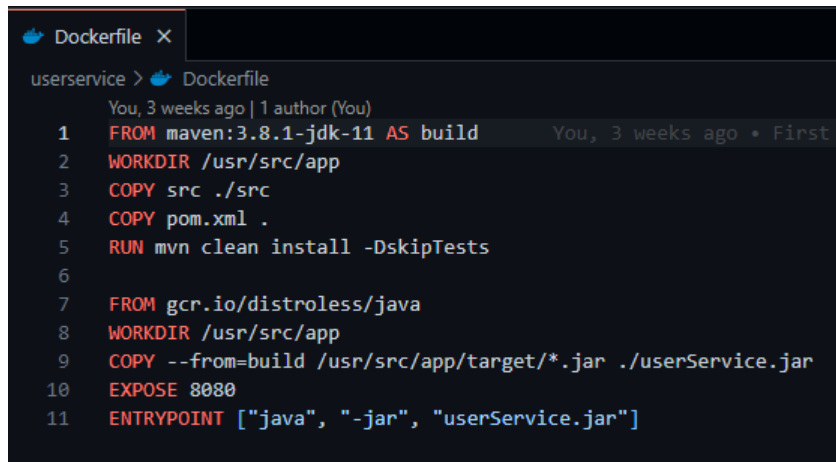
```
Uploaded using RayThis Extension  
mvn clean install -DskipTests
```

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



```
Dockerfile X  
doctorservice > Dockerfile  
You, 3 weeks ago | 1 author (You)  
1 FROM maven:3.8.1-jdk-11 AS build  
2 WORKDIR /usr/src/app  
3 COPY src ./src  
4 COPY pom.xml .  
5 RUN mvn clean install -DskipTests  
6  
7 FROM gcr.io/distroless/java  
8 WORKDIR /usr/src/app  
9 COPY --from=build /usr/src/app/target/*.jar doctorService.jar  
10 EXPOSE 8080  
11 ENTRYPOINT ["java", "-jar", "doctorService.jar"]  
You, 3 w
```

Fig 3.1: Dockerfile for DoctorService



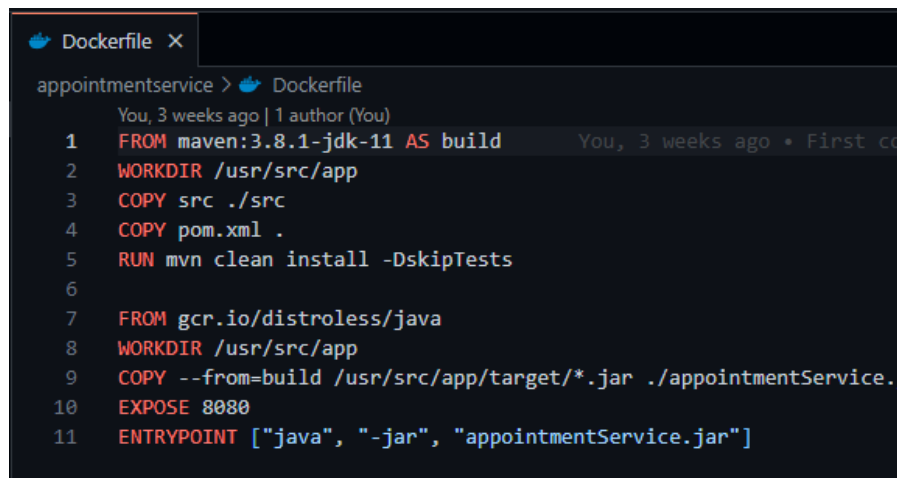
The screenshot shows a code editor with a tab labeled 'Dockerfile'. The content is a Dockerfile for a service named 'userservice'. It starts with a 'FROM' instruction using 'maven:3.8.1-jdk-11' as the base image. The 'WORKDIR' is set to '/usr/src/app'. Source files are copied from the local directory. A 'RUN' instruction executes 'mvn clean install -DskipTests'. A second 'FROM' instruction uses 'gcr.io/distroless/java' as the base image. The 'WORKDIR' remains '/usr/src/app'. The application JAR is copied from the build output to 'userService.jar'. The 'EXPOSE' instruction is set to '8080'. The 'ENTRYPOINT' is configured to run 'java' with arguments '-jar' and 'userService.jar'.

```

1 FROM maven:3.8.1-jdk-11 AS build
2 WORKDIR /usr/src/app
3 COPY src ./src
4 COPY pom.xml .
5 RUN mvn clean install -DskipTests
6
7 FROM gcr.io/distroless/java
8 WORKDIR /usr/src/app
9 COPY --from=build /usr/src/app/target/*.jar ./userService.jar
10 EXPOSE 8080
11 ENTRYPOINT ["java", "-jar", "userService.jar"]

```

Fig 3.2: Dockerfile for UserService



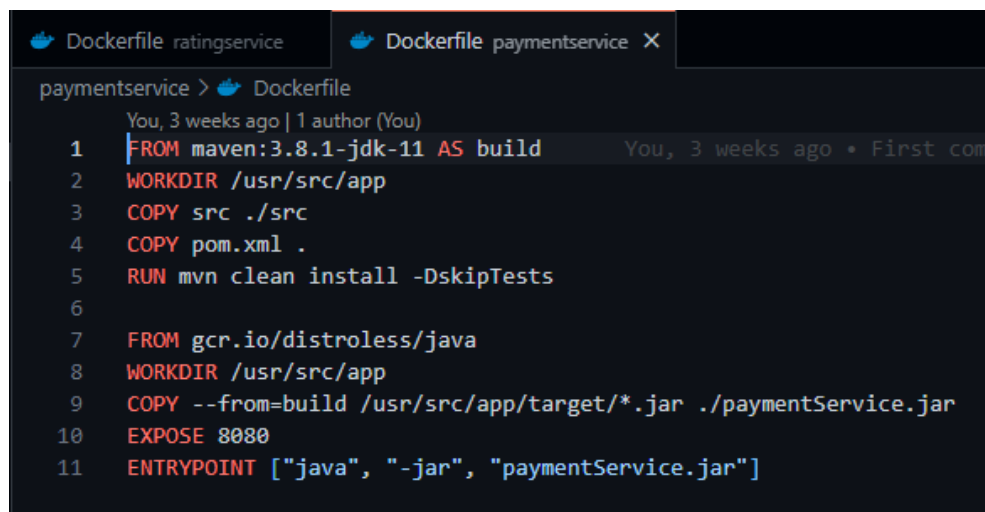
The screenshot shows a code editor with a tab labeled 'Dockerfile'. The content is a Dockerfile for a service named 'appointmentservice'. It follows a similar structure to the previous one, starting with 'FROM maven:3.8.1-jdk-11 AS build'. The 'WORKDIR' is '/usr/src/app'. Source files are copied, and 'mvn clean install -DskipTests' is run. The second 'FROM' instruction uses 'gcr.io/distroless/java'. The application JAR is copied to 'appointmentService.jar'. The 'EXPOSE' instruction is '8080'. The 'ENTRYPOINT' is 'java' with arguments '-jar' and 'appointmentService.jar'.

```

1 FROM maven:3.8.1-jdk-11 AS build
2 WORKDIR /usr/src/app
3 COPY src ./src
4 COPY pom.xml .
5 RUN mvn clean install -DskipTests
6
7 FROM gcr.io/distroless/java
8 WORKDIR /usr/src/app
9 COPY --from=build /usr/src/app/target/*.jar ./appointmentService.jar
10 EXPOSE 8080
11 ENTRYPOINT ["java", "-jar", "appointmentService.jar"]

```

Fig 3.3: Dockerfile for AppointmentService



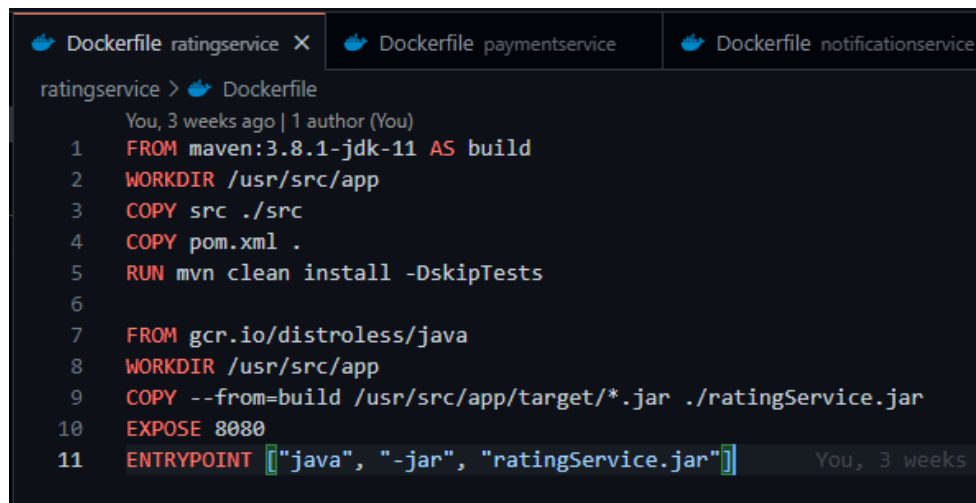
The screenshot shows a code editor with a tab labeled 'Dockerfile'. The content is a Dockerfile for a service named 'paymentservice'. It follows the same pattern, starting with 'FROM maven:3.8.1-jdk-11 AS build'. The 'WORKDIR' is '/usr/src/app'. Source files are copied, and 'mvn clean install -DskipTests' is run. The second 'FROM' instruction uses 'gcr.io/distroless/java'. The application JAR is copied to 'paymentService.jar'. The 'EXPOSE' instruction is '8080'. The 'ENTRYPOINT' is 'java' with arguments '-jar' and 'paymentService.jar'.

```

1 FROM maven:3.8.1-jdk-11 AS build
2 WORKDIR /usr/src/app
3 COPY src ./src
4 COPY pom.xml .
5 RUN mvn clean install -DskipTests
6
7 FROM gcr.io/distroless/java
8 WORKDIR /usr/src/app
9 COPY --from=build /usr/src/app/target/*.jar ./paymentService.jar
10 EXPOSE 8080
11 ENTRYPOINT ["java", "-jar", "paymentService.jar"]

```

Fig 3.4: Dockerfile for PaymentService



```
ratingService > Dockerfile
You, 3 weeks ago | 1 author (You)
1 FROM maven:3.8.1-jdk-11 AS build
2 WORKDIR /usr/src/app
3 COPY src ./src
4 COPY pom.xml .
5 RUN mvn clean install -DskipTests
6
7 FROM gcr.io/distroless/java
8 WORKDIR /usr/src/app
9 COPY --from=build /usr/src/app/target/*.jar ./ratingService.jar
10 EXPOSE 8080
11 ENTRYPOINT ["java", "-jar", "ratingService.jar"]
```

Fig 3.5: Dockerfile for ratingService



```
notificationService > Dockerfile
You, 3 weeks ago | 1 author (You)
1 FROM maven:3.8.1-jdk-11 AS build
2 WORKDIR /usr/src/app
3 COPY src ./src
4 COPY pom.xml .
5 RUN mvn clean install -DskipTests
6
7 FROM gcr.io/distroless/java
8 WORKDIR /usr/src/app
9 COPY --from=build /usr/src/app/target/notificationService-*.jar notificationService.jar
10 EXPOSE 8080
11 ENTRYPOINT ["java", "-jar", "notificationService.jar"]
```

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.

```
version: '3.3'

services:
  doctor:
    build: doctorservice
    container_name: doctorService
    image: book_my_consultation/doctorService:1.0.0
    ports:
      - "8081:8081"
    environment:
      MONGO_HOST: <mongodb-connection-address>
      KAFKA_HOST: 3.87.198.243
    depends_on:
      - notification
    networks:
      - app-tier
  user:
    build: userservice
    container_name: userService
    image: book_my_consultation/userservice:1.0.0
    ports:
      - "8082:8082"
    environment:
      MONGO_HOST: <mongodb-connection-address>
    networks:
      - app-tier
  appointment:
    build: appointmentService
    container_name: appointmentService
    image: book_my_consultation/appointmentService:1.0.0
    ports:
      - "8083:8083"
    environment:
      MYSQL_HOST: book-my-consultation.c4zicvaqizuv.us-east-1.rds.amazonaws.com
      MONGO_HOST: <mongodb-connection-address>
    depends_on:
      - notification
      - doctor
      - user
    networks:
      - app-tier
  payment:
    build: paymentService
    container_name: paymentService
    image: book_my_consultation/paymentService:1.0.0
    ports:
      - "8084:8084"
    depends_on:
      - appointment
      - notification
    networks:
      - app-tier
  rating:
    build: ratingService
    container_name: ratingService
    image: book_my_consultation/ratingService:1.0.0
    ports:
      - "8085:8085"
    environment:
      MONGO_HOST: <mongodb-connection-address>
    depends_on:
      - doctor
    networks:
      - app-tier
  notification:
    build: notificationService
    container_name: notificationService
    image: book_my_consultation/notificationService:1.0.0
    environment:
      KAFKA_HOST: 3.87.198.243
    networks:
      - app-tier

networks:
  app-tier:
    driver: bridge
```

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 run the following commands.

```
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
```

Fig 5.1: start Kafka server

Before we start the services application, 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

```
docker-compose build
```

Fig 5.2: build service images

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

```
docker-compose up
```

Fig 5.3: start services

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

```
ubuntu@ip-10-0-0-114:~/BookMyConsultation$ docker-compose up -d
Starting userService ... done
Starting notificationService ... done
Starting doctorService ... done
Starting appointmentService ... done
Starting ratingsService ... done
Starting paymentService ... done
ubuntu@ip-10-0-0-114:~/BookMyConsultation$ docker images
docker ps
REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
book_my_consultation/userService    1.0.0        a200d254f6e  24 hours ago  485MB
<none>                        <none>       e0a454370c32  24 hours ago  1.6GB
book_my_consultation/paymentService 1.0.0        cd56bd752506  27 hours ago  270MB
book_my_consultation/ratingsService 1.0.0        227d860647bc  27 hours ago  274MB
book_my_consultation/doctorService  1.0.0        e18ecbc33dbd  27 hours ago  486MB
book_my_consultation/notificationService 1.0.0        a78517a8851b  28 hours ago  478MB
<none>                        <none>       5792ca37e6ed  28 hours ago  485MB
book_my_consultation/appointmentService 1.0.0        46806a07419f  28 hours ago  501MB
maven                        3.8.1-jdk-11 5b508b1fe19e  12 months ago  659MB
gcr.io/distroless/java      latest       4c4b3da468da  52 years ago   210MB
ubuntu@ip-10-0-0-114:~/BookMyConsultation$ docker ps
CONTAINER ID   NAME                                COMMAND                  CREATED      STATUS      PORTS                                NAMES
332ba6d28539  book_my_consultation/paymentService:1.0.0  "java -jar paymentSe..." 24 hours ago Up 15 seconds 8080/tcp, 0.0.0.0:8084->8084/tcp, :::8084->8084/tcp  paymentService
252d2b4041f4  book_my_consultation/appointmentService:1.0.0  "java -jar appointme..." 24 hours ago Up 16 seconds 8080/tcp, 0.0.0.0:8083->8083/tcp, :::8083->8083/tcp  appointmentService
e8c1c77a5c3b  book_my_consultation/ratingsService:1.0.0    "java -jar ratingSer..." 24 hours ago Up 16 seconds 8080/tcp, 0.0.0.0:8085->8085/tcp, :::8085->8085/tcp  ratingsService
3b02a31f6b1e  book_my_consultation/doctorService:1.0.0    "java -jar doctorSer..." 24 hours ago Up 17 seconds 8080/tcp, 0.0.0.0:8081->8081/tcp, :::8081->8081/tcp  doctorService
3c7089b9b5a1c  book_my_consultation/userService:1.0.0      "java -jar userServ..." 24 hours ago Up 17 seconds 8080/tcp, 0.0.0.0:8082->8082/tcp, :::8082->8082/tcp  userService
a1eb27b4e426  book_my_consultation/notificationService:1.0.0  "java -jar notificat..." 24 hours ago Up 17 seconds 8080/tcp  notificationService
```

Fig 5.4: docker images and containers

Testing with APIs

DoctorService

1. 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.

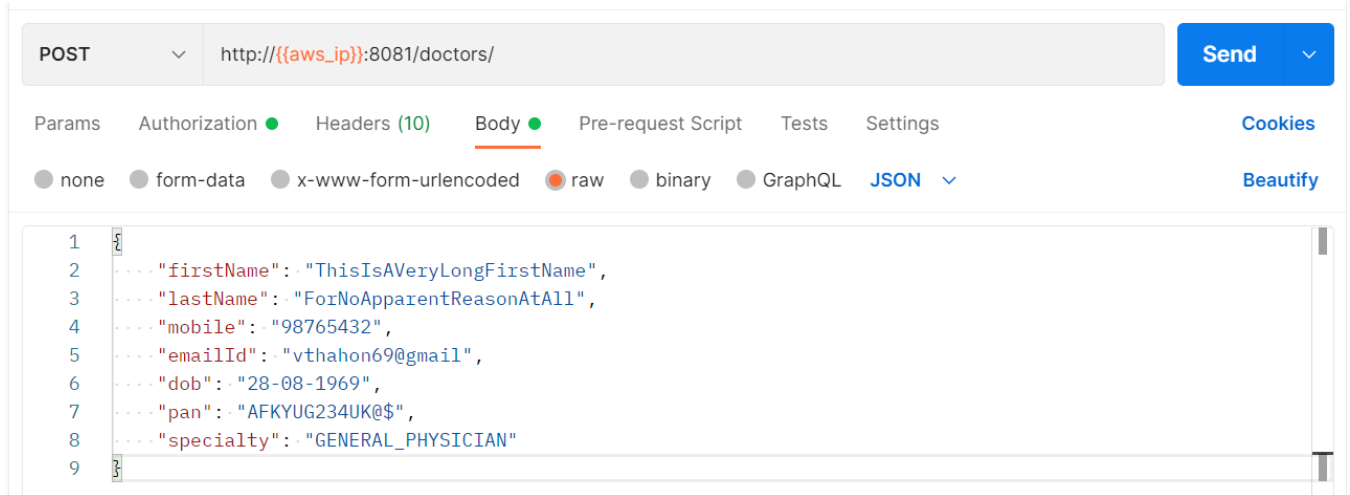


Fig 6.1a: Incorrect Details to create doctor

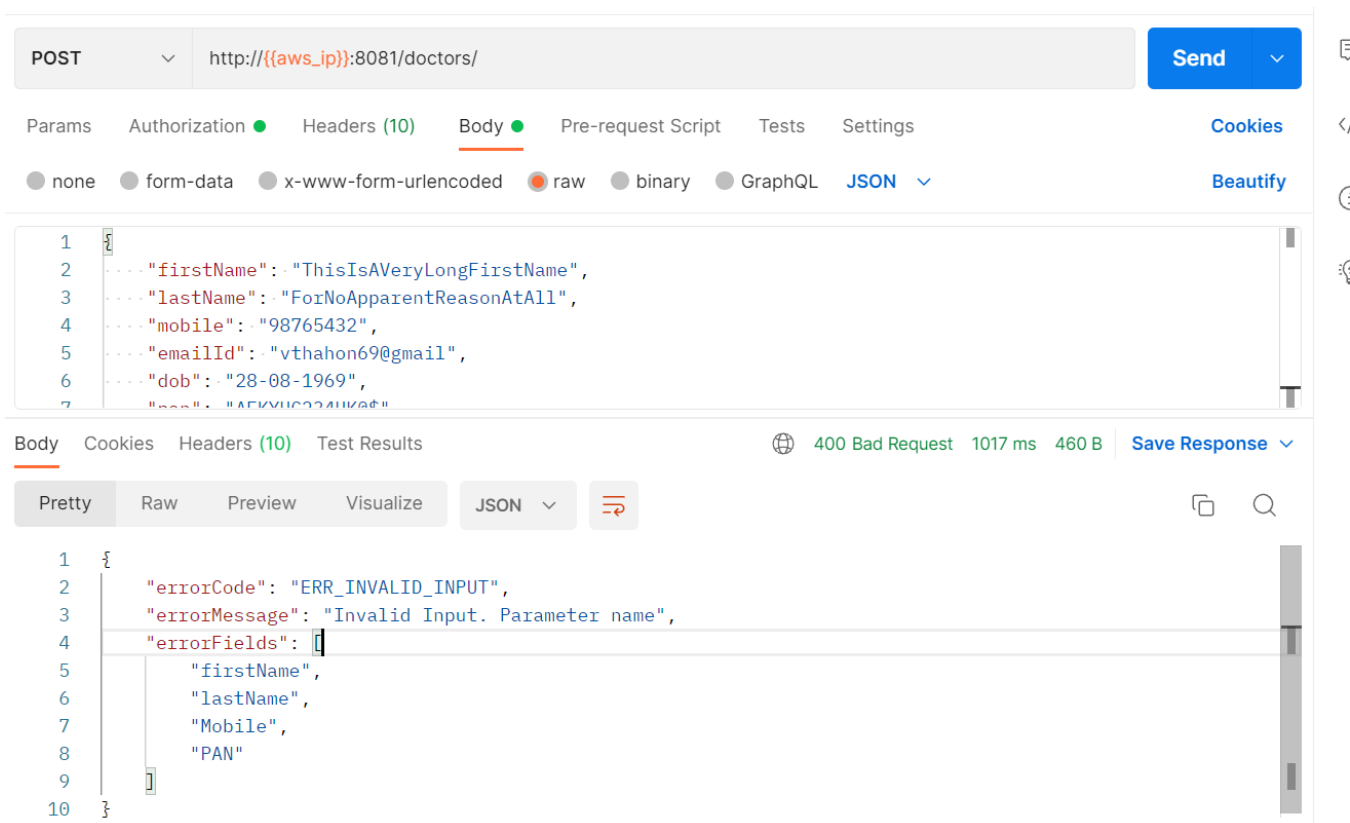


Fig 6.1b: Response for incorrect details to create Doctor

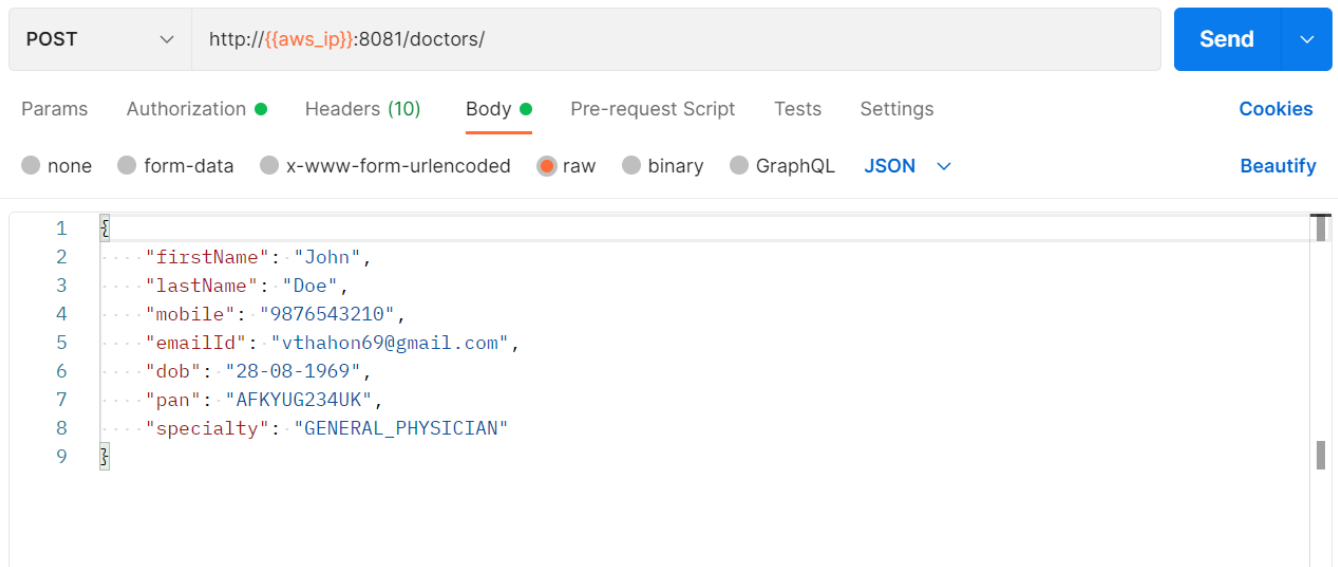


Fig 6.1c: Correct details to create doctor

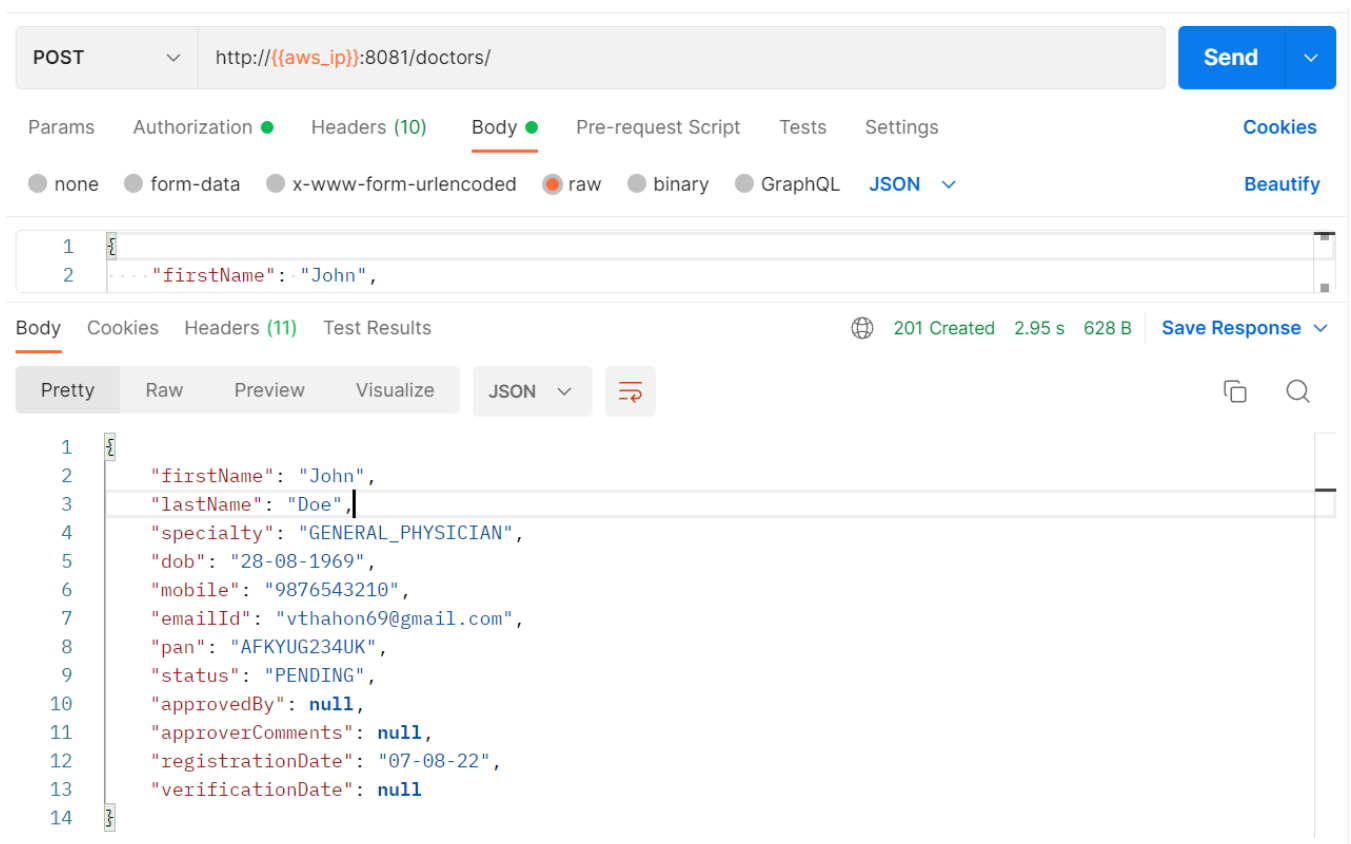


Fig 6.1d: Response for correct details to create doctor

```
14:34:37.842 [main] INFO org.apache.kafka.clients.consumer.internals.Fetcher - [Consumer clientId=consumer-bookMyConsultation-1, groupId=bookMyConsultation] Built incremental fetch (sessionId=14318758, epoch=1) for mode 0: added 0 partiti
on(s), altered 1 partition(s), removed 0 partition(s), replaced 0 partition(s) out of 1 partition(s)
14:34:37.842 [main] INFO org.apache.kafka.clients.consumer.internals.Fetcher - [Consumer clientId=consumer-bookMyConsultation-1, groupId=bookMyConsultation] Sending READ_UNCOMMITTED IncrementalFetchRequest(toSend=(message=0), toForget=
), toReplace=(), implied=(), canUseTopicId=True) to broker 3.84.55.47:9092 (id: 0 rack: null)
14:34:37.842 [main] INFO org.apache.kafka.clients.NetworkClient - [Consumer clientId=consumer-bookMyConsultation-1, groupId=bookMyConsultation] Sending FETCH request with header RequestHeader(apiKey=FETCH, apiVersion=12, clientId=consum
er-bookMyConsultation-1, correlationId=9230) and timeout 30000 to node 0: FetchRequestData(clusterId=null, replicaId=1, maxWaitMs=500, minBytes=1, maxBytes=5242880, isolationLevel=0, sessionId=182157535, sessionEpoch=113, topics=[Fet
chTopic(topic="message", topicId=bookMyConsultation-1QFao10qkq1d8, partitions=[FetchPartition(partition=0, currentLeaderEpoch=0, fetchOffset=0, lastFetchedEpoch=1, logStartOffset=1, partitionMaxBytes=1048576)])], forgottenTopicsData=[], rackId=
'')
=====
RECORD: doctorCreate = Dr.Doe has been created. Please find the details below:
Doctor{id='62efcd7b8117b7562eda2b04', firstName='John', lastName='Doe', specialty='GENERAL_PHYSICIAN', dob='28-08-1969', mobile='9876543210', emailId='vthahon69@gmail.com', pan='AFKYUG234UK', status='PENDING', approvedBy='null', approve
dComments='null', registrationDate='07-08-22', verificationDate='null'}
```

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

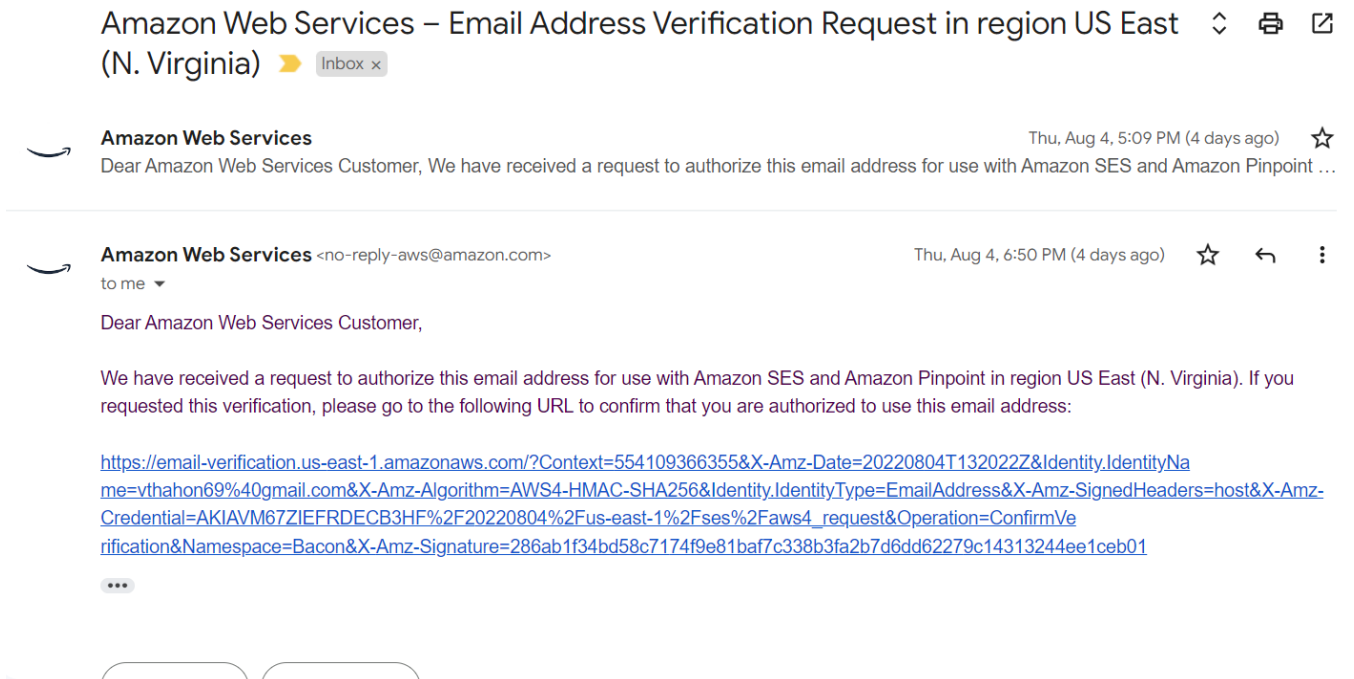


Fig 6.1f: Verification email sent to Doctor Email

2. Upload Documents to Doctor S3 bucket

BMC-Doctor / uploadDocument Save ... 🔗 💬

POST ⌵ http://{{aws_ip}}:8081/doctors/62d7d3536092031630d5ccba/document? Send ⌵

Params ● Authorization Headers (10) ● Body ● Pre-request Script Tests Settings ● Cookies

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	files	Adithya_Vardhan.pdf ×			
	Key	Value	Description		

Fig 6.2a: Request to upload documents for Doctor

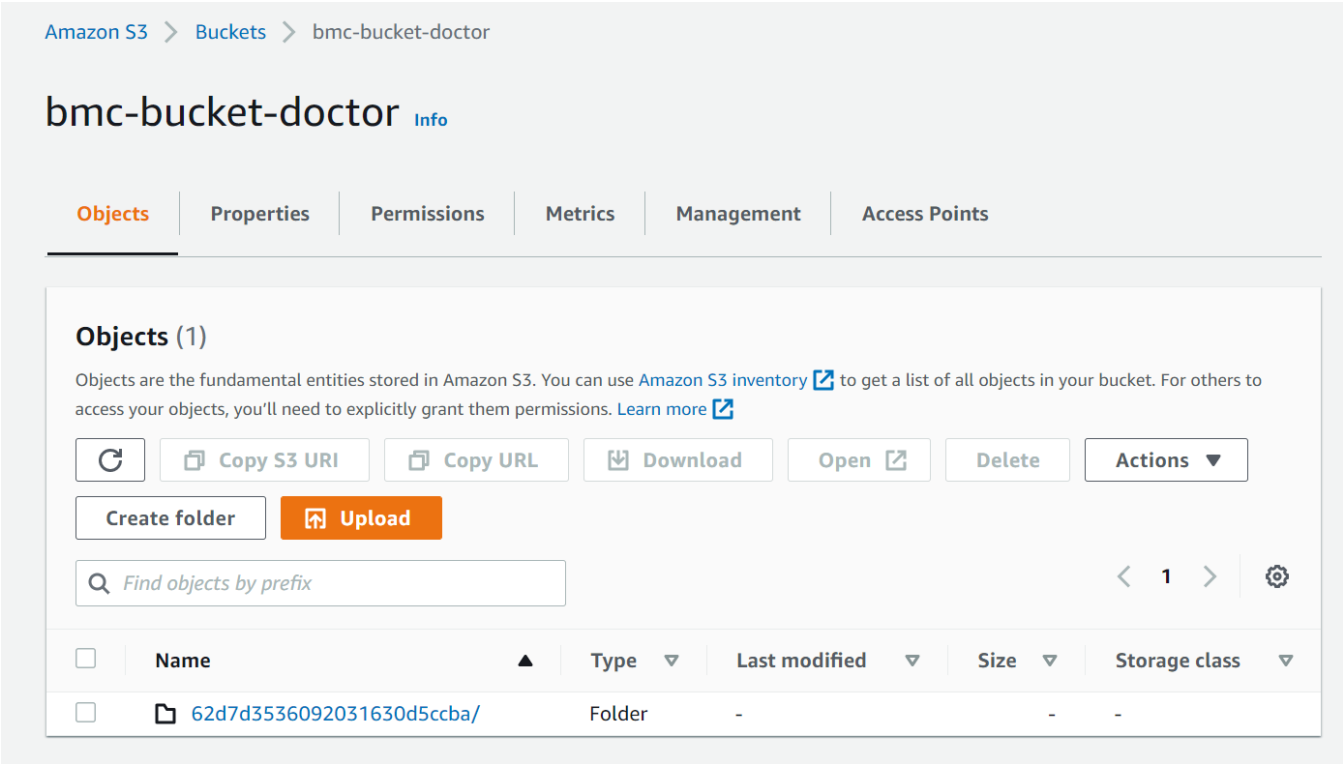


Fig 6.2b: Documents uploaded to Amazon S3

3. Approve Doctor

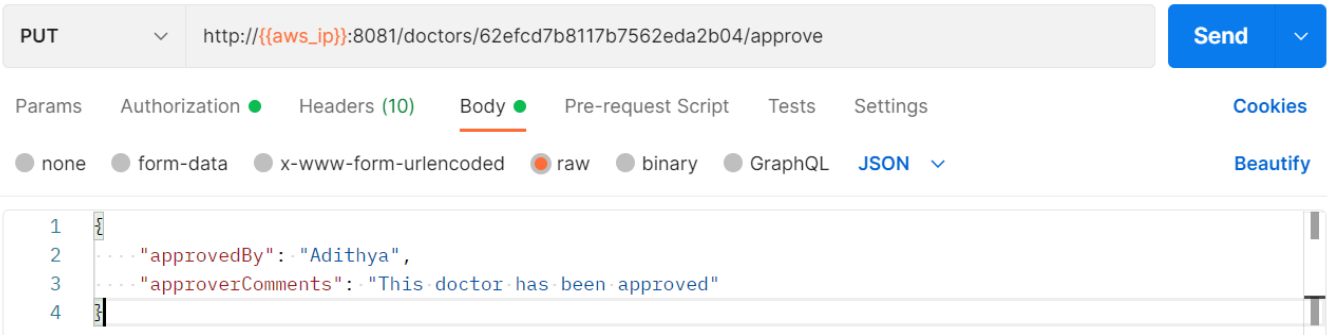


Fig 6.3a: Request to approve doctor

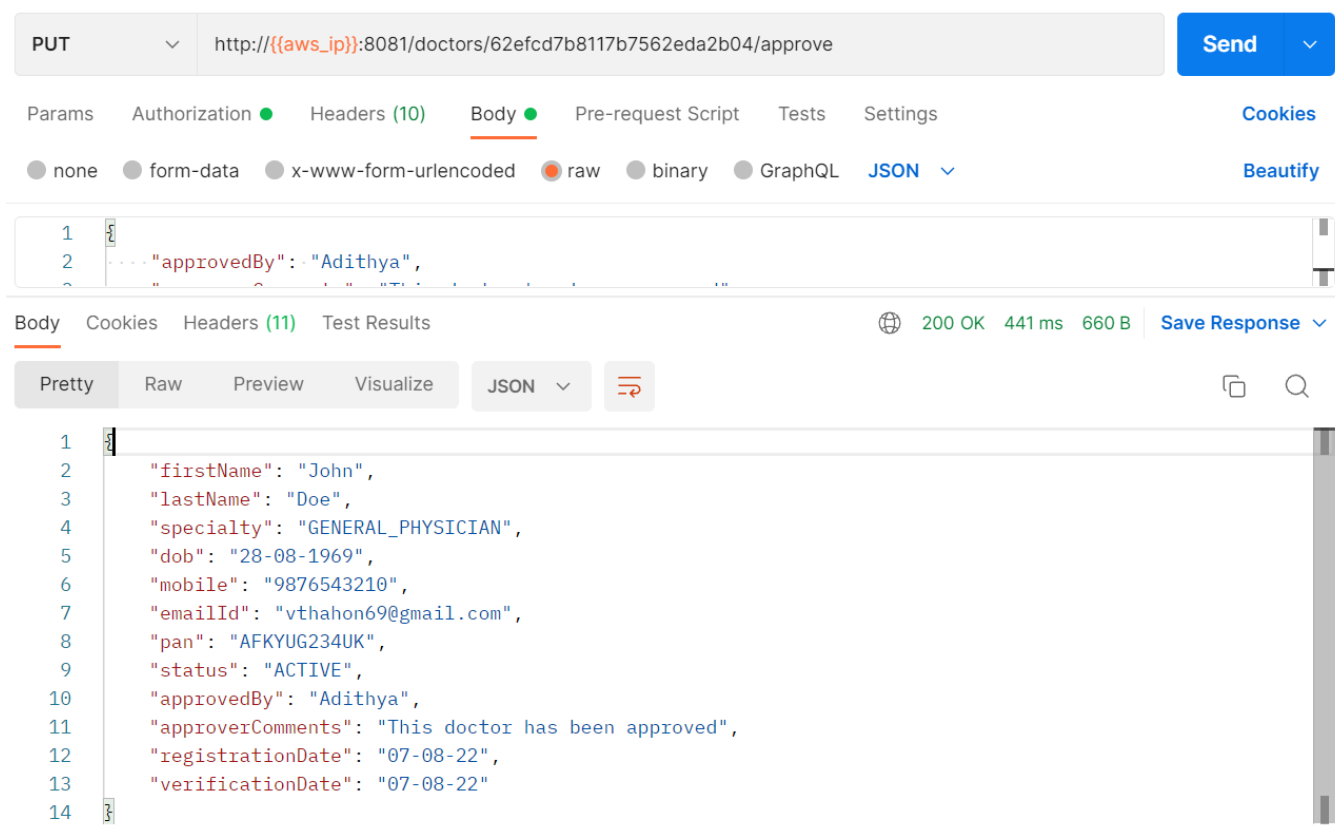


Fig 6.3b: Response for approving doctor

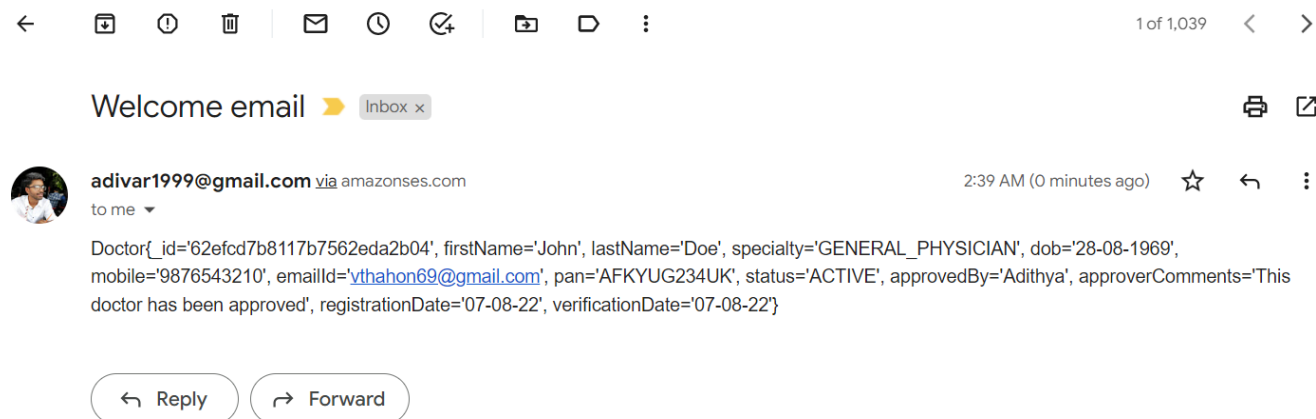


Fig 6.3c: Email for Doctor Approval

4. Reject Doctor

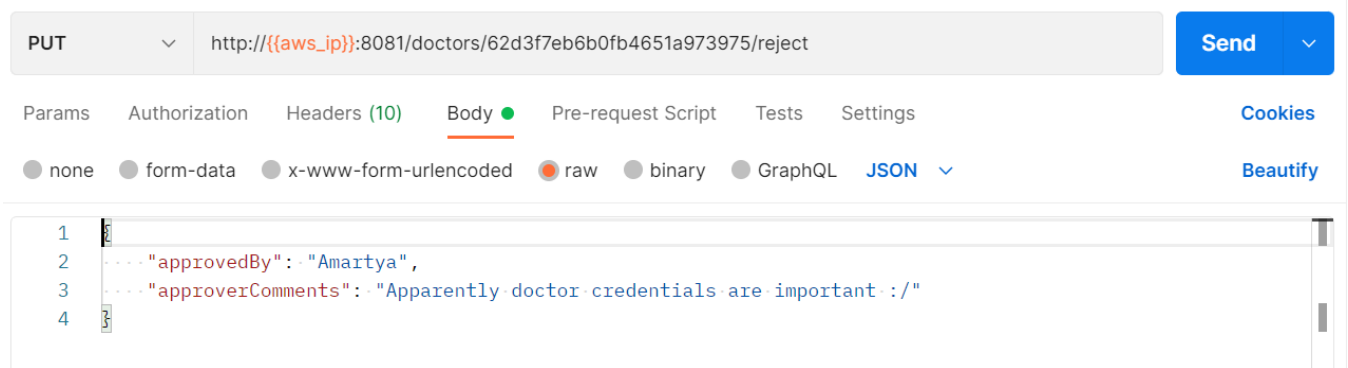


Fig 6.4a: Request to reject Doctor

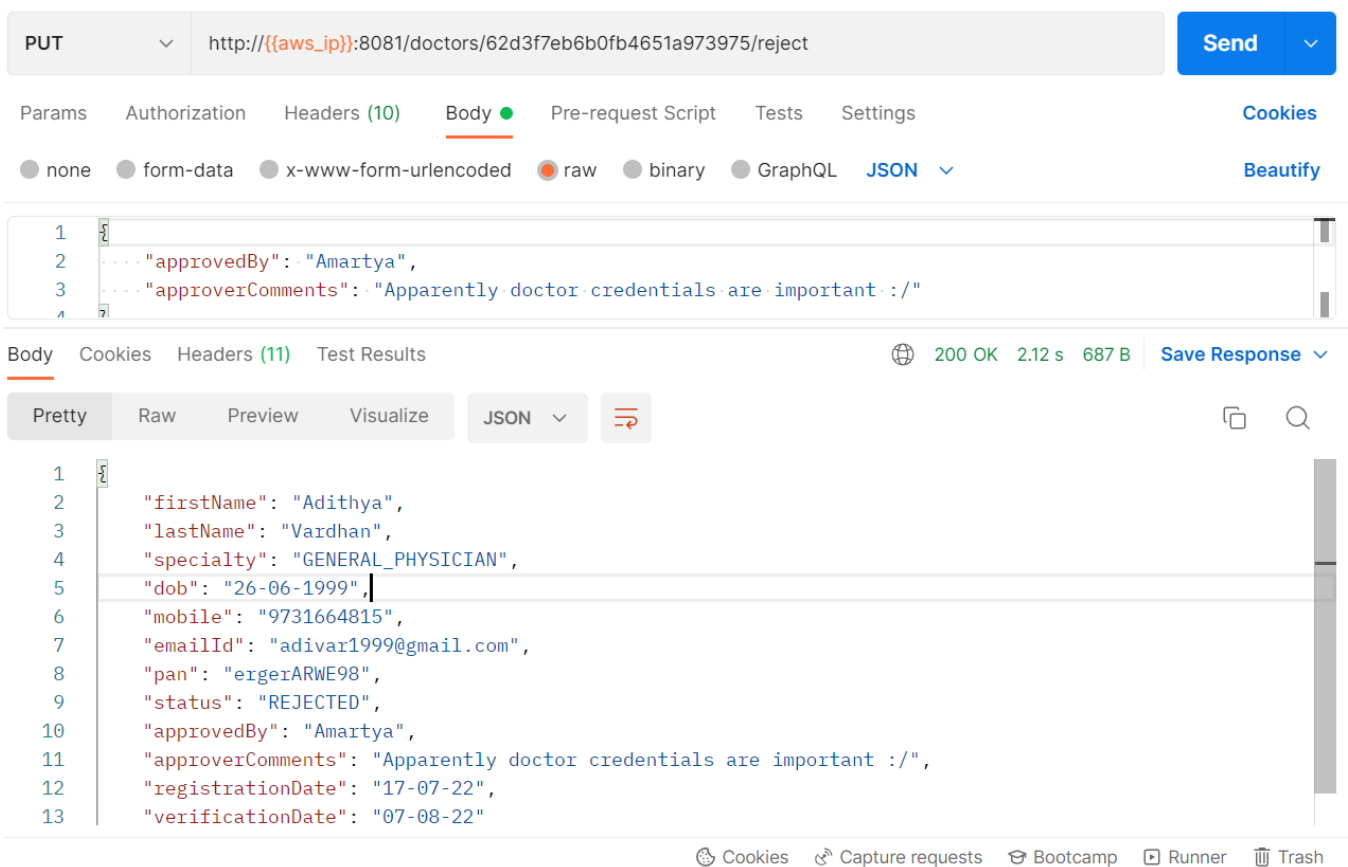


Fig 6.4b: Response to reject Doctor

5. Return a list of 20 Doctors sorted by rating

The screenshot shows a REST client interface with a GET request to `http://{{aws_ip}}:8081/doctors`. The response status is 200 OK, with a response time of 1802 ms and a body size of 2.43 KB. The response body is displayed in JSON format, showing a single doctor object:

```
{
  "firstName": "Amartya",
  "lastName": "Vardhan",
  "specialty": "GENERAL_PHYSICIAN",
  "dob": "26-06-1999",
  "mobile": "9880165936",
  "emailId": "amartya2606@gmail.com",
  "pan": "defghikjACD002254"
}
```

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

The screenshot shows a REST client interface with a GET request to `http://{{aws_ip}}:8081/doctors?specialty=DENTIST`. The response status is 200 OK, with a response time of 270 ms and a body size of 661 B. The response body is displayed in JSON format, showing a single doctor object:

```
[
  {
    "firstName": "Meghana",
    "lastName": "Mohan",
    "specialty": "DENTIST",
    "dob": "11-09-1996",
    "mobile": "9535597858",
    "emailId": "vtahon69@gmail.com",
    "pan": "reog340394jf",
    "status": "ACTIVE",
    "approvedBy": "Adithya",
    "approverComments": "My Sister, so bigger nepotism ;)",
    "registrationDate": "20-07-22",
    "verificationDate": "06-08-22"
  }
]
```

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

6. Return details of a Doctor by DoctorId

The screenshot displays a REST client interface with a GET request to the endpoint `http://{{aws_ip}}:8081/doctors/62efcd7b8117b7562eda2b04`. The response is a 200 OK status with a 494 ms response time and 660 B of data. The response body is shown in JSON format, detailing a doctor's information.

Request Details:

- Method: GET
- URL: `http://{{aws_ip}}:8081/doctors/62efcd7b8117b7562eda2b04`
- Body: This request does not have a body

Response Details:

- Status: 200 OK
- Time: 494 ms
- Size: 660 B
- Save Response: [Save Response](#)

Response Body (JSON):

```
1 {
2   "firstName": "John",
3   "lastName": "Doe",
4   "specialty": "GENERAL_PHYSICIAN",
5   "dob": "28-08-1969",
6   "mobile": "9876543210",
7   "emailId": "vthahon69@gmail.com",
8   "pan": "AFKYUG234UK",
9   "status": "ACTIVE",
10  "approvedBy": "Adithya",
11  "approverComments": "This doctor has been approved",
12  "registrationDate": "07-08-22",
13  "verificationDate": "07-08-22"
14 }
```

Fig 6.6: Request and response to get doctor details

UserService

1. Create User

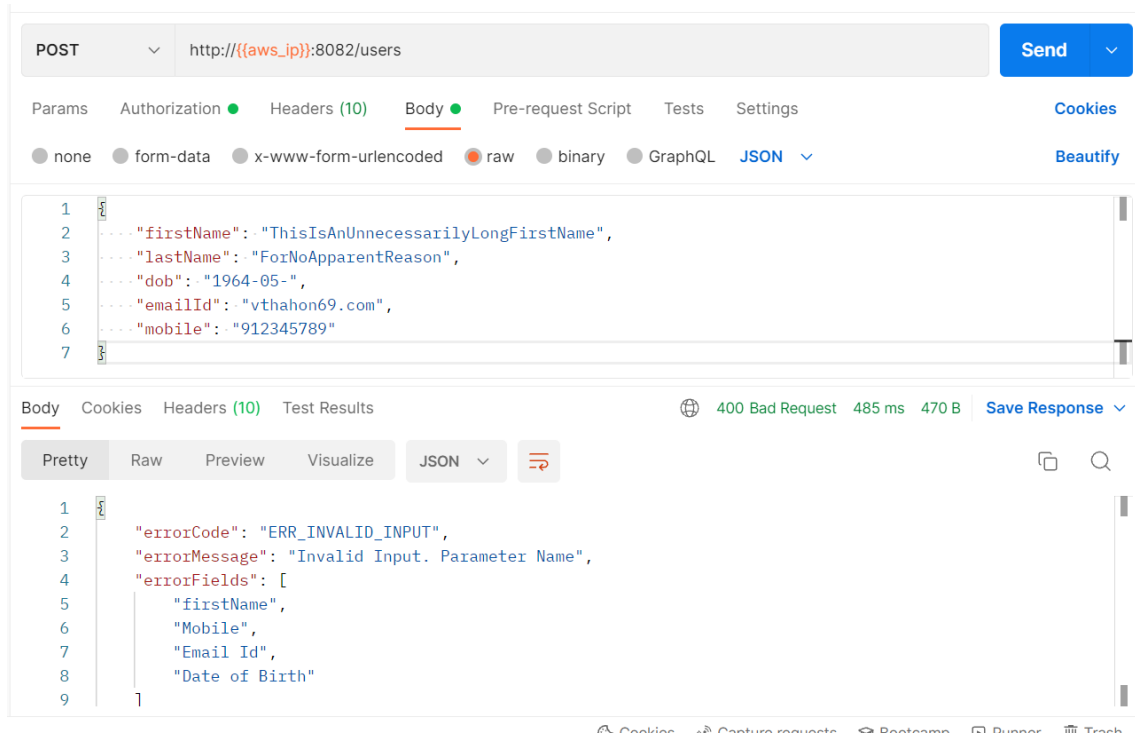


Fig 7.1a: Request and response for incorrect entries for Create User

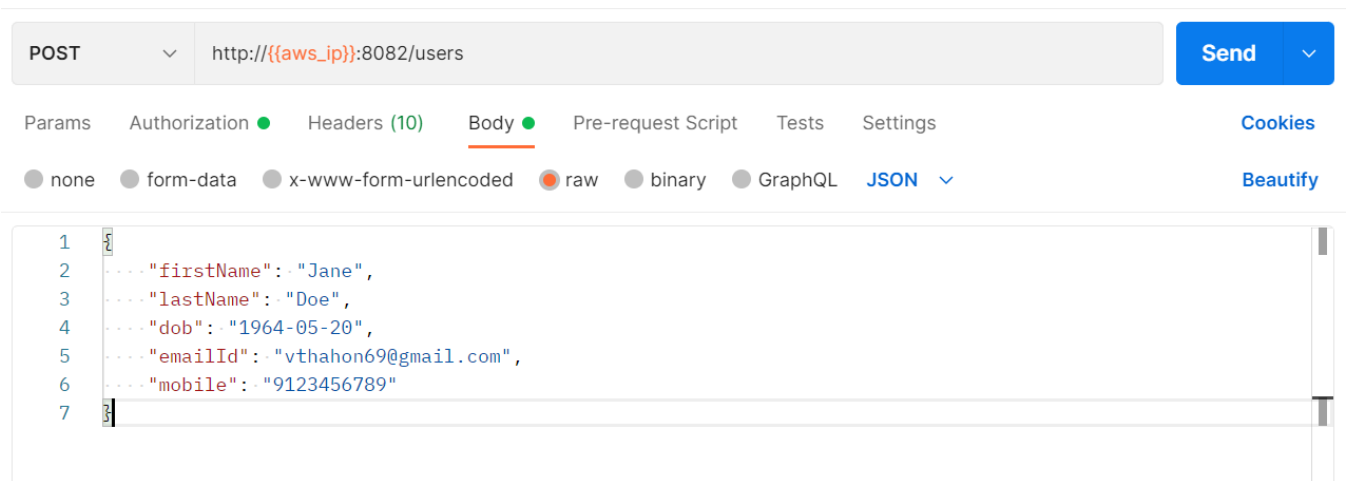


Fig 7.1b: Correct Request for Create User

POST http://{{aws_ip}}:8082/users Send

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "firstName": "Jane",
3   "lastName": "Doe",
4   "dob": "1964-05-20",
5   "emailId": "vthahon69@gmail.com",
6   "mobile": "9123456789"
7 }
```

Body Cookies Headers (11) Test Results 201 Created 3.00 s 486 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "firstName": "Jane",
3   "lastName": "Doe",
4   "dob": "1964-05-20",
5   "mobile": "9123456789",
6   "emailId": "vthahon69@gmail.com",
7   "createdDate": "07=08-22"
8 }
```

Fig 7.1c: Response for correct Create User

```
msg = javax.mail.internet.MimeMessage@7522783e
Trying to send to AWS: (AKIAYCA356BJVGTRGNXZ:BIM1+ufUEjbbjjangU5wizj6FVzRG6rBWLKs5NucqsRSA)
=====
RECORD: userCreate:vthahon69@gmail.com = UserDoe has been created. Please find the details below:
User(_id='62f03063822eb368927614fb', firstName='Jane', lastName='Doe', dob='1964-05-20', mobile='9123456789', emailId='vthahon69@gmail.com', createdDate='07=08-22')
KAFKA KEY: userCreate for email: vthahon69@gmail.com
false
emailId = vthahon69@gmail.com
```

Fig7.1d: Notification in Kafka for creating user

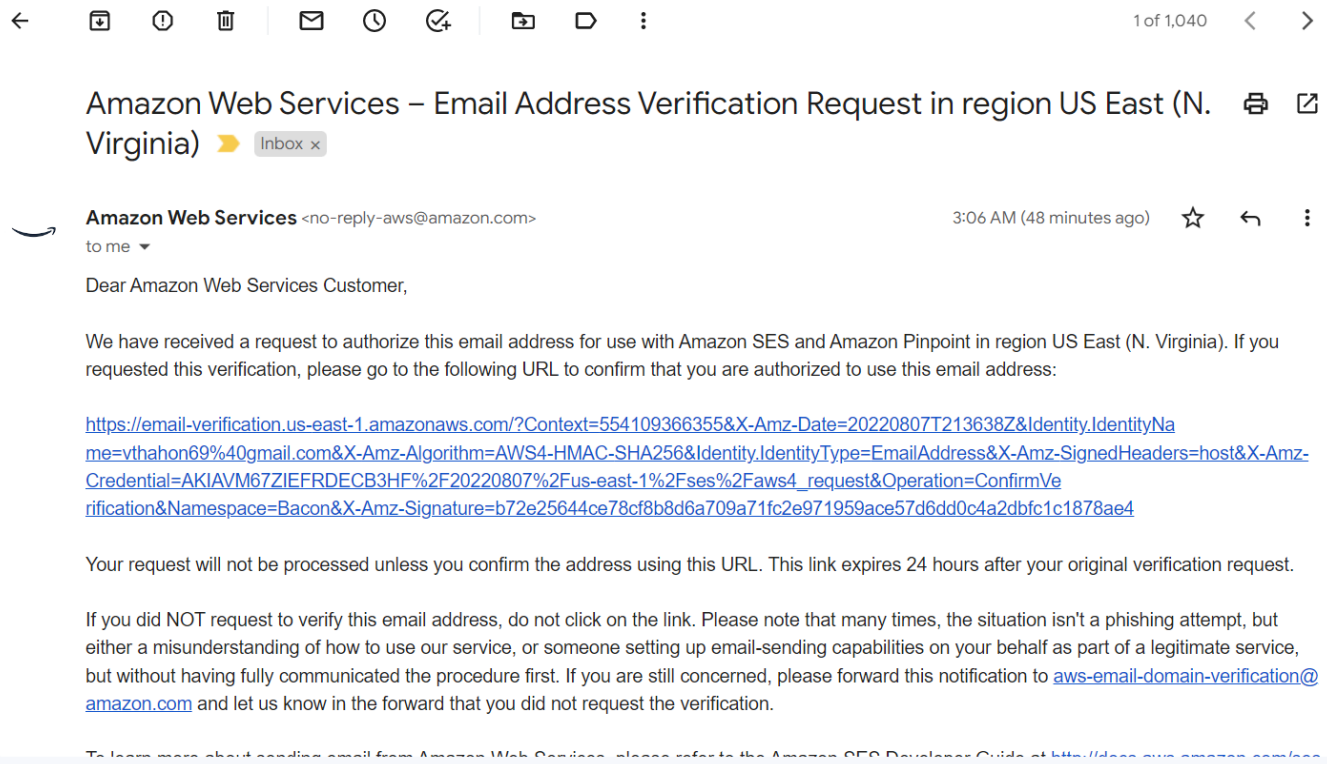


Fig 7.1e: Verification Email for Create user

2. getUser

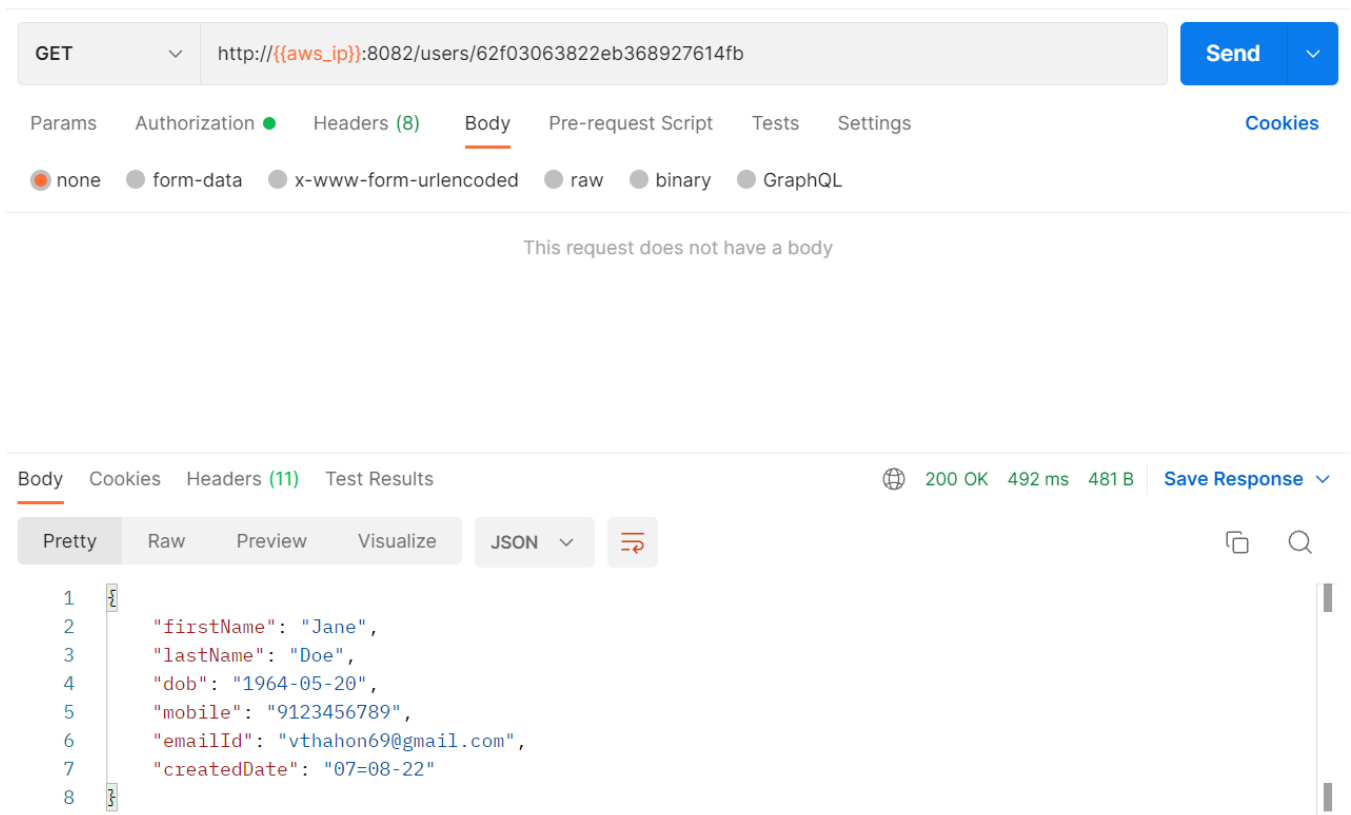


Fig 7.2: Request and response for getUser

3. Upload Documents for User

The screenshot shows a REST client interface with a POST request to `http://({aws_ip}):8082/users/62f03063822eb368927614fb/documents`. The request body is a multipart form with a file named `Adithya_Vardhan.pdf`. The response is a 200 OK status with a body containing the text `File(s) uploaded Successfully`.

KEY	VALUE	DESCRIPTION
files	Adithya_Vardhan.pdf	
Key	Value	Description

Response: 200 OK, 1792 ms, 375 B. Body: File(s) uploaded Successfully.

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

The screenshot shows the Amazon S3 console for the bucket `bmc-bucket-user`. Under the `Objects` tab, a folder named `62f03063822eb368927614fb/` is listed. The folder is shown with a folder icon, its name, type (Folder), and last modified date (-).

Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Actions: Copy S3 URI, Copy URL, Download, Open, Delete, Actions

Create folder, Upload

Find objects by prefix

Name	Type	Last modified	Size	Storage class
62f03063822eb368927614fb/	Folder	-	-	-

Fig 7.3b: File uploaded to Amazon S3

AppointmentService

1. createAvailability

The screenshot shows a REST client interface for a POST request. The URL is `http://{{aws_ip}}:8083/doctor/62efcd7b8117b7562eda2b04/availability`. The request body is a JSON object:

```
1 {
2   "availabilityMap": {
3     "2021-07-20": ["10AM-11AM", "11AM-11:30AM"],
4     "2021-07-21": ["9AM-10AM", "10AM-11AM"]
5   }
6 }
```

The response status is 201 Created, with a response time of 4.32 s and a size of 310 B. The response body is empty.

Fig 8.1: Request and response for setAvailability

2. getAvailability

The screenshot shows a REST client interface for a GET request. The URL is `http://{{aws_ip}}:8083/doctor/62efcd7b8117b7562eda2b04/availability`. The response status is 201 Created, with a response time of 328 ms and a size of 490 B. The response body is a JSON object:

```
1 {
2   "doctorId": "62efcd7b8117b7562eda2b04",
3   "availabilityMap": {
4     "2021-07-21": [
5       "9AM-10AM",
6       "10AM-11AM"
7     ],
8     "2021-07-20": [
9       "10AM-11AM",
10      "11AM-11:30AM"
11     ]
12   }
13 }
```

Fig 8.2: Request and response for getAvailability

3. createAppointment

The screenshot shows a REST client interface with a POST request to `http://{{aws_ip}}:8083/appointments`. The request body is a JSON object with the following fields:

```
{
  "doctor_id": "62efcd7b8117b7562eda2b04",
  "user_id": "62f03063822eb368927614fb",
  "appointment_date": "2021-07-21",
  "time_slot": "9AM-10AM"
}
```

The response is a 200 OK status with a response time of 4.51s and a body size of 378 B. The response body is a long alphanumeric string: `2c928085827c982f01827c9896c70000`.

Fig 8.3a: Request and response for create Appointment

The screenshot shows a Kafka log entry with the following message:

```
RECORD: setAppointment:vthahon09@gmail.com = Appointment has been set with doctor John Doe On Wed Jul 21 00:00:00 GMT 2021 at 9AM-10AM
```

Fig 8.3b: Kafka Notification for create Appointment

The screenshot shows an email titled "Set Appointment" from `adivar1999@gmail.com` via `amazonses.com`. The email content states: "Appointment has been set with doctor John Doe On Wed Jul 21 00:00:00 GMT 2021 at 9AM-10AM". The email was received at 2:05 PM (0 minutes ago).

Fig 8.3c: Email confirmation for create Appointment

4. getAppointment

The screenshot shows a REST client interface with a GET request to `http://{{aws_ip}}:8083/appointments/2c928085827c982f01827c9896c70000`. The response is a JSON object with the following fields:

```
1 {
2   "appointment_id": "2c928085827c982f01827c9896c70000",
3   "appointment_date": "2021-07-21 00:00:00.0",
4   "created_date": "2022-08-08T08:35:51.000+00:00",
5   "doctor_id": "62efcd7b8117b7562eda2b04",
6   "prior_medical_history": null,
7   "status": "PENDING_PAYMENT",
8   "symptoms": null,
9   "time_slot": "9AM-10AM",
10  "user_id": "62f03063822eb368927614fb",
11  "user_email_id": "vthahon69@gmail.com",
12  "user_name": "Jane Doe",
13  "doctor_name": "John Doe"
14 }
```

Fig 8.4: Request and response for get appointment

5. getAppointments

The screenshot shows a REST client interface with a GET request to `http://{{aws_ip}}:8083/users/62f03063822eb368927614fb/appointments`. The response is a JSON object with the following fields:

```
1 {
2   "appointment_id": "2c928085827c982f01827c9896c70000",
3   "appointment_date": "2021-07-21 00:00:00.0",
4   "created_date": "2022-08-08T08:35:51.000+00:00",
5   "doctor_id": "62efcd7b8117b7562eda2b04",
6   "prior_medical_history": null,
7   "status": "PENDING_PAYMENT",
8   "symptoms": null,
9   "time_slot": "9AM-10AM",
10  "user_id": "62f03063822eb368927614fb",
11  "user_email_id": "vthahon69@gmail.com",
12  "user_name": "Jane Doe",
13  "doctor_name": "John Doe"
14 }
```

Fig 8.5: Request and response for get Appointments by user

6. createPrescription

The screenshot shows a REST client interface with a POST request to `http://{{aws_ip}}:8083/prescriptions`. The request body is a JSON object with the following structure:

```
1 {
2   "doctorId": "62efcd7b8117b7562eda2b04",
3   "userId": "62f03063822eb368927614fb",
4   "appointmentId": "2c928085827c982f01827c9896c70000",
5   "diagnosis": "Explosive Diarrhoea",
6   "medicineList": [
7     {
8       "name": "Calpol",
9       "dosage": "1 week",
10      "frequency": "3 times a day",
11      "remarks": "after food"
12    },
13    {
14      "name": "PainKill",
15      "dosage": "1 week",
16      "frequency": "3 times a day",
17      "remarks": "after food"
18    }
19  ]
20 }
```

Fig 8.6a: Request for create Prescription

The screenshot shows the same REST client interface, but now displaying a 400 Bad Request response. The status bar indicates "400 Bad Request", "591 ms", and "464 B". The response body is a JSON object with the following structure:

```
1 {
2   "errorCode": "ERR_PAYMENT_PENDING",
3   "errorMessage": "Prescription cannot be issued since the payment status is pending.",
4   "errorFields": null
5 }
```

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

PaymentService

1. createPayment

POST

http://{{aws_ip}}:8084/payments?appointmentId=2c928085827c982f01827c9896c70000

Send

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	appointmentId	2c928085827c982f01827c9896c70000			
	Key	Value	Description		

Body

Cookies

Headers (11)

Test Results

200 OK 2.32 s 443 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "createdDate": "Tue Aug 09 11:41:07 IST 2022",
3   "appointmentId": "2c928085827c982f01827c9896c70000"
4 }
```

Cookies

Capture requests

Bootcamp

Runner

Trash

Fig 9.1: Request and response for create Payment

RatingService

1. createRating

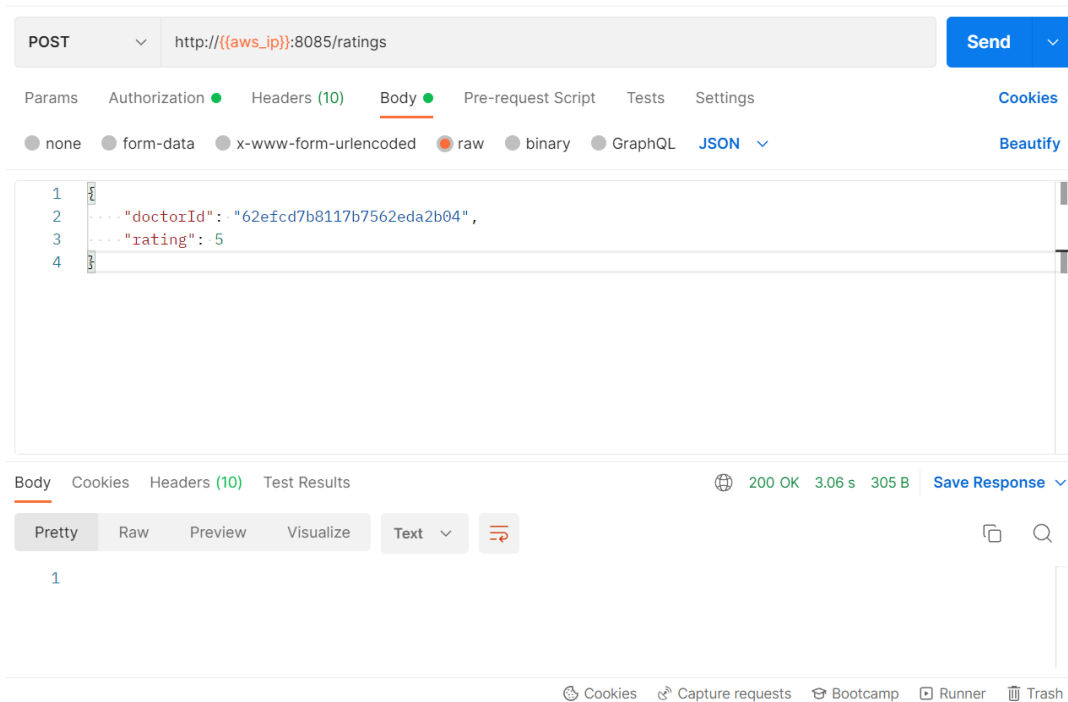


Fig 10.1: Request and response for creating Rating

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

The [approveDoctor](#), [rejectDoctor](#), [setAppointment](#) and [setPrescription](#) 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.

POST http://localhost:8080/login Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```

1 {
2   "username": "admin-user@abc.com",
3   "password": "Admin@123"
4 }

```

Body Cookies Headers (11) Test Results 200 OK 601 ms 576 B Save Response

KEY	VALUE
authorization	Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWiiOiJhZG1pb11c2VyQGFiYy5j...
X-Content-Type-Options	nosniff
X-XSS-Protection	1; mode=block
Cache-Control	no-cache, no-store, max-age=0, must-revalidate
Pragma	no-cache

Cookies Capture requests Bootcamp Runner Trash

Fig 12.1: Request and response for Admin JWT token generation

POST http://localhost:8080/login Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```

1 {
2   "username": "normal-user@abc.com",
3   "password": "Test@123"
4 }

```

Body Cookies Headers (11) Test Results 200 OK 435 ms 541 B Save Response

KEY	VALUE
authorization	Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWiiOiJub3JtYWwtdXNlckBhYm...
X-Content-Type-Options	nosniff
X-XSS-Protection	1; mode=block
Cache-Control	no-cache, no-store, max-age=0, must-revalidate
Pragma	no-cache

Cookies Capture requests Bootcamp Runner Trash

Fig 12.2: Request and response for User JWT token generation

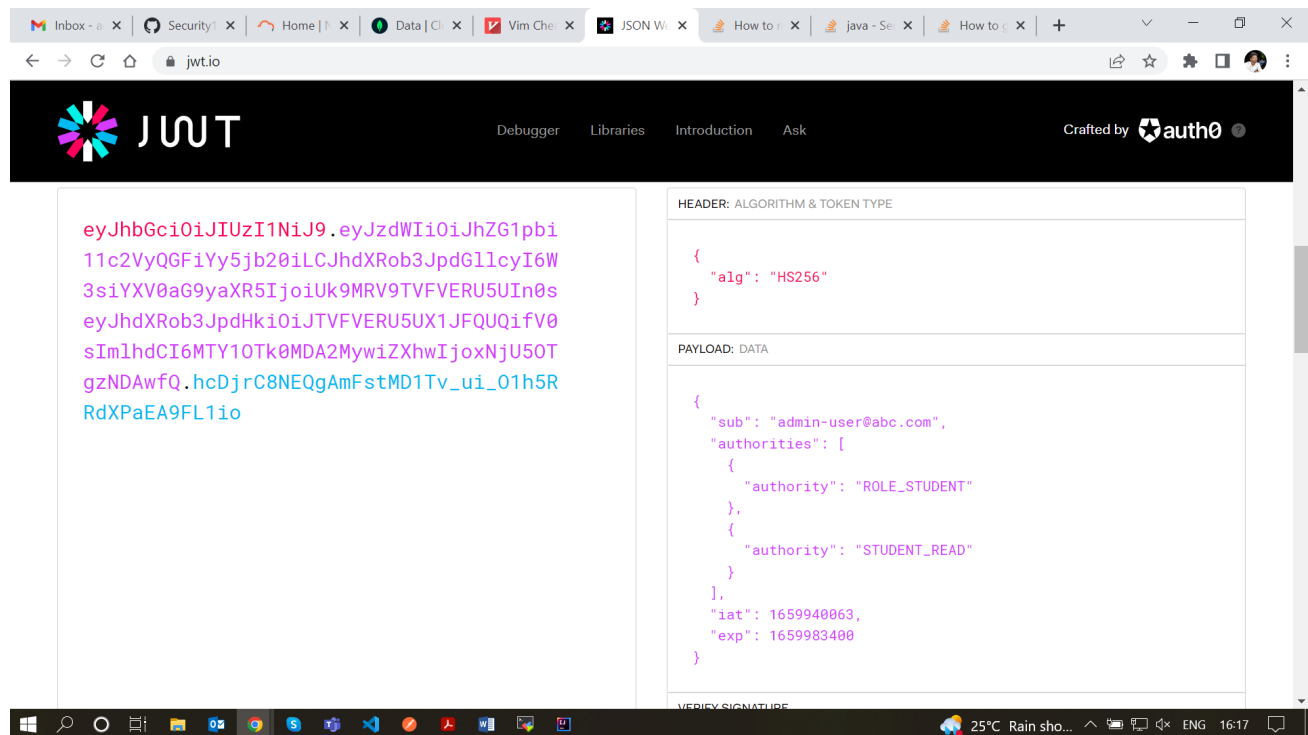


Fig 12.3: JWT token decrypted

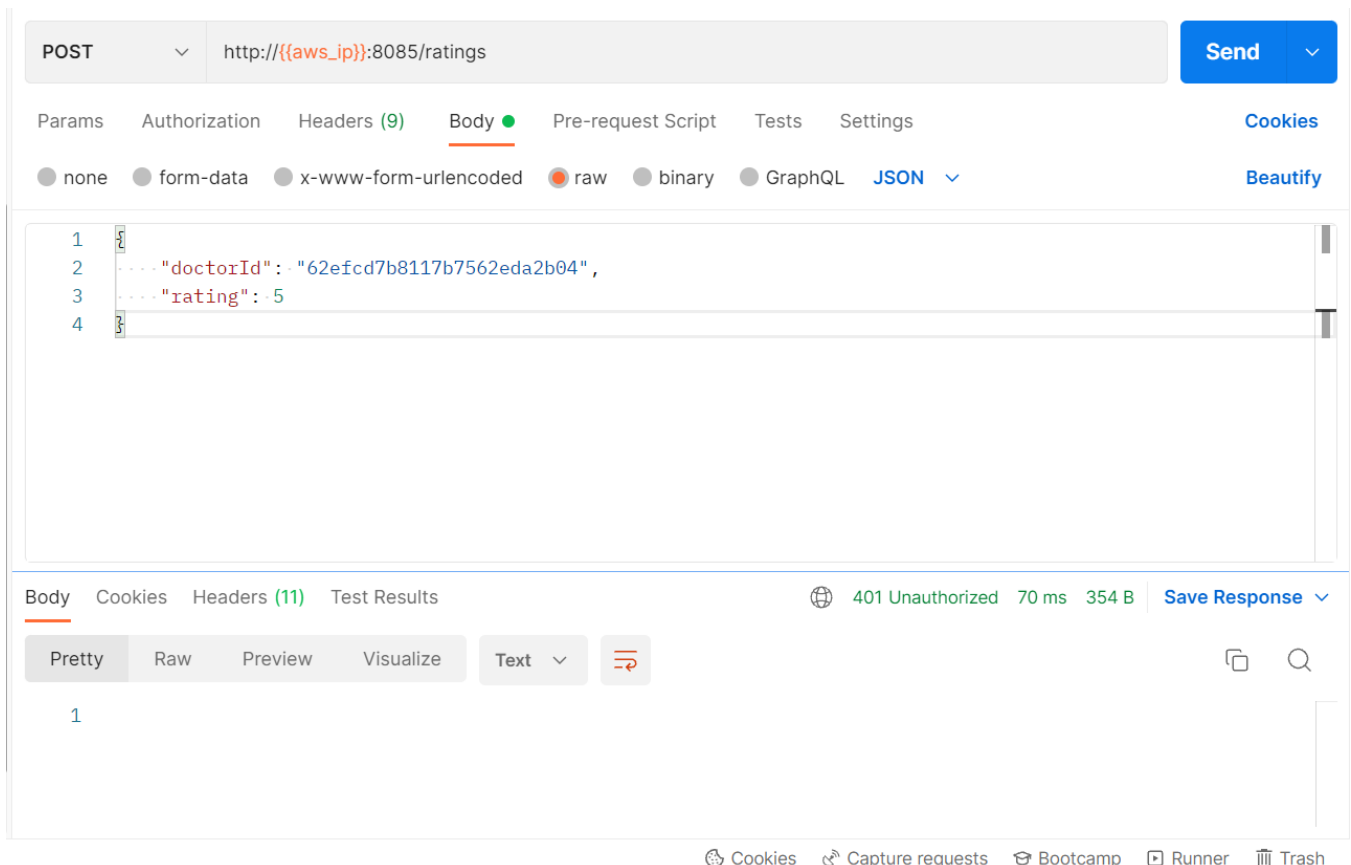


Fig 12.4: Response of an API without authorization

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.