

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.

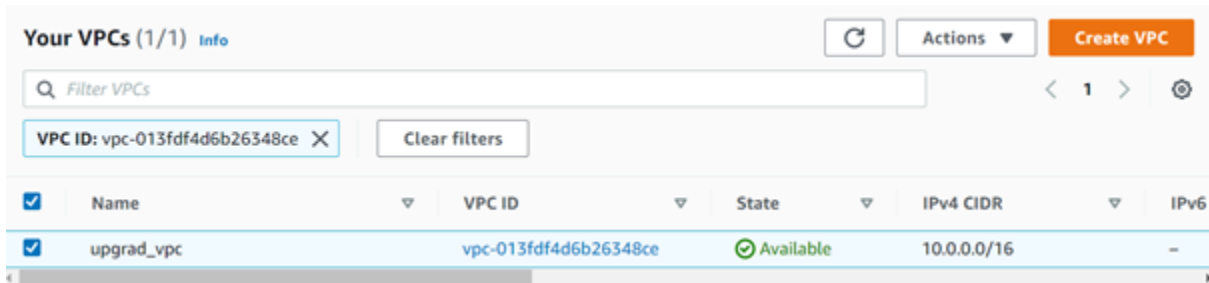


Fig 1.1: VPCs setup

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.

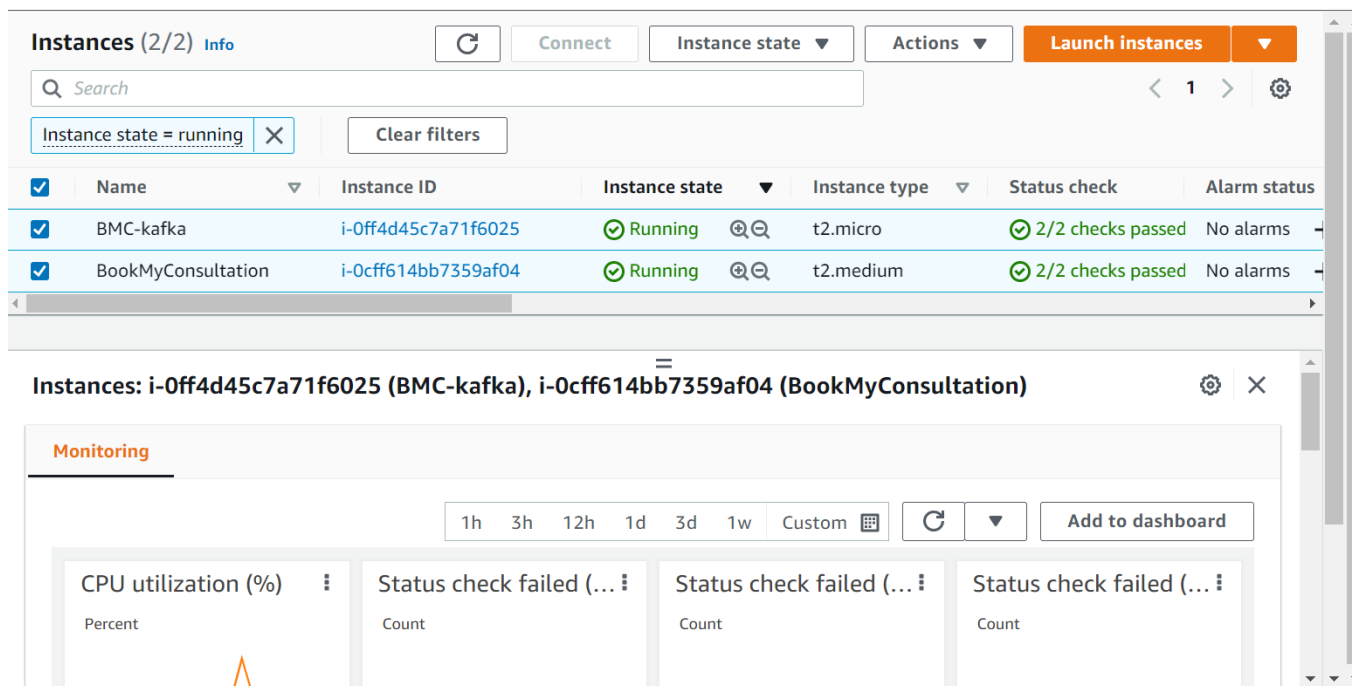
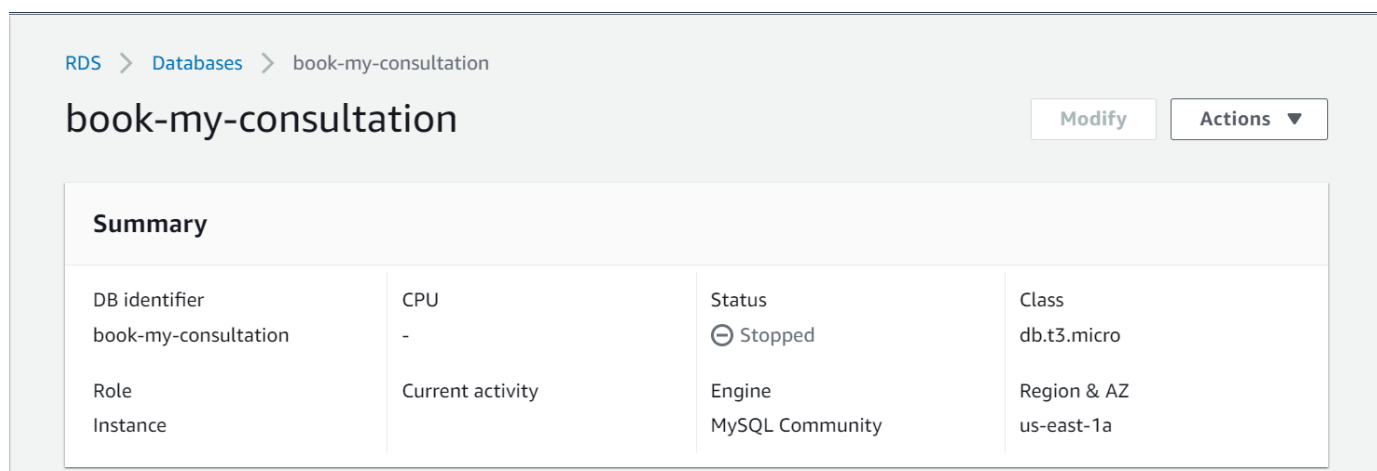


Fig 1.2: EC2 instances setup

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

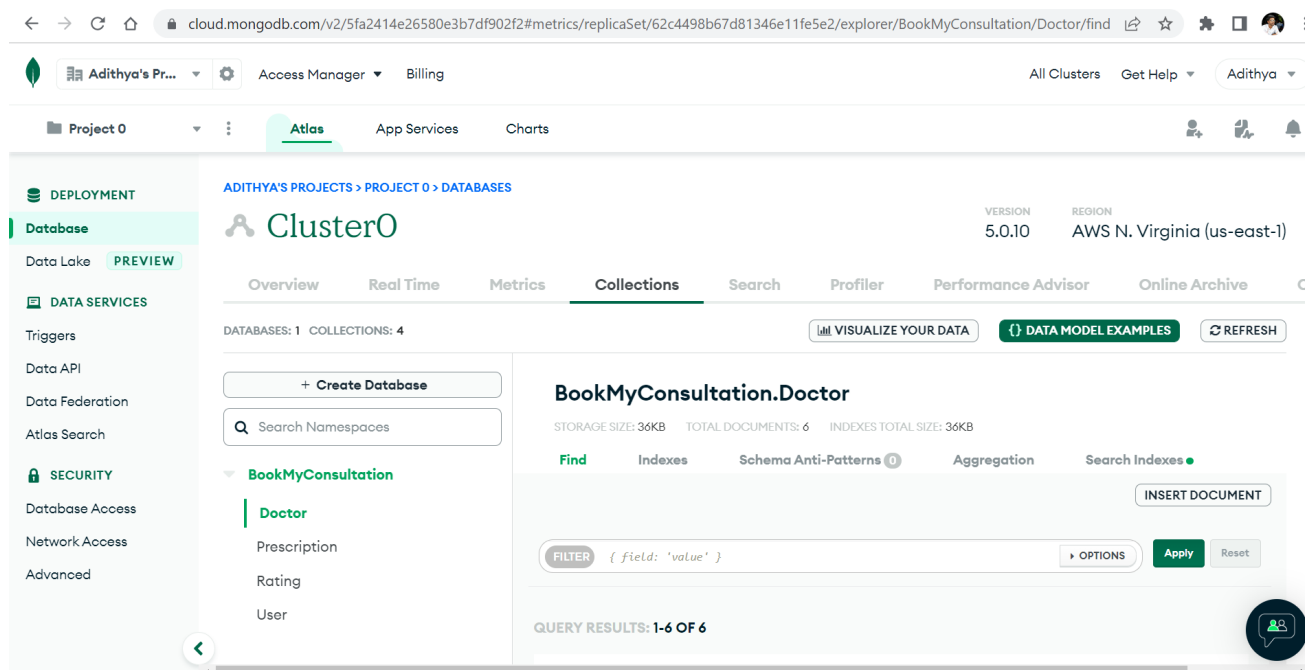


The screenshot shows the AWS RDS console for the instance 'book-my-consultation'. The breadcrumb navigation is 'RDS > Databases > book-my-consultation'. The instance name 'book-my-consultation' is displayed at the top with 'Modify' and 'Actions' buttons. Below is a 'Summary' section with a table of instance 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 cluster 'Cluster0' is selected, with version 5.0.10 and region AWS N. Virginia (us-east-1). The 'Collections' tab is active, showing 'BookMyConsultation.Doctor'. The collection has a storage size of 36KB, 6 total documents, and 36KB of indexes. The 'Find' tab is selected, showing a filter bar with the query '{ field: 'value' }'. The 'QUERY RESULTS' section shows '1-6 OF 6' results. The left sidebar contains navigation links for Deployment, Database, Data Services, and Security.

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

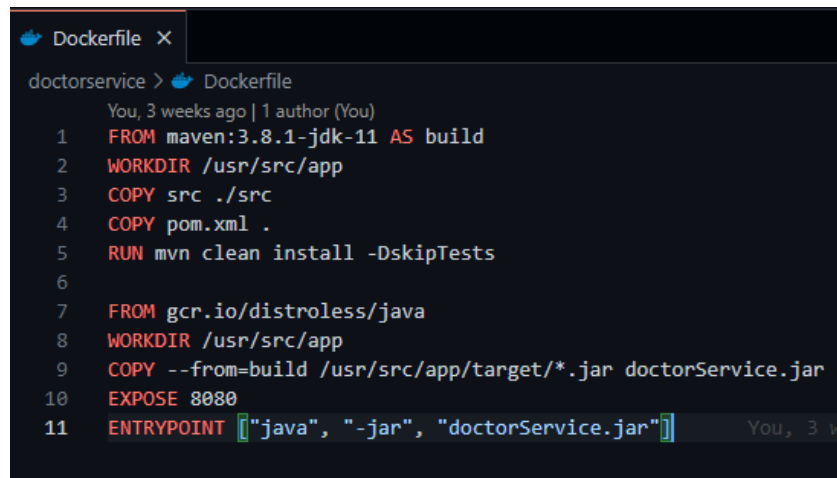
Implementation

Creating Dockerfiles

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

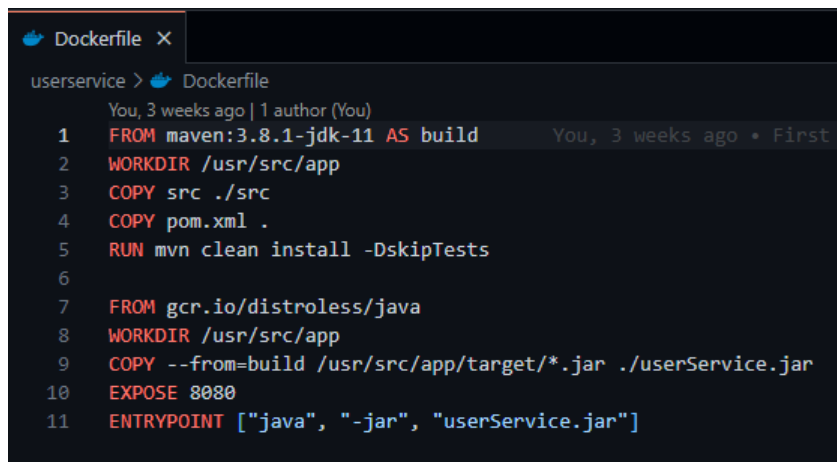
The dockerfile automatically creates a jar file for the service using a Maven image to compile and package the service.

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.



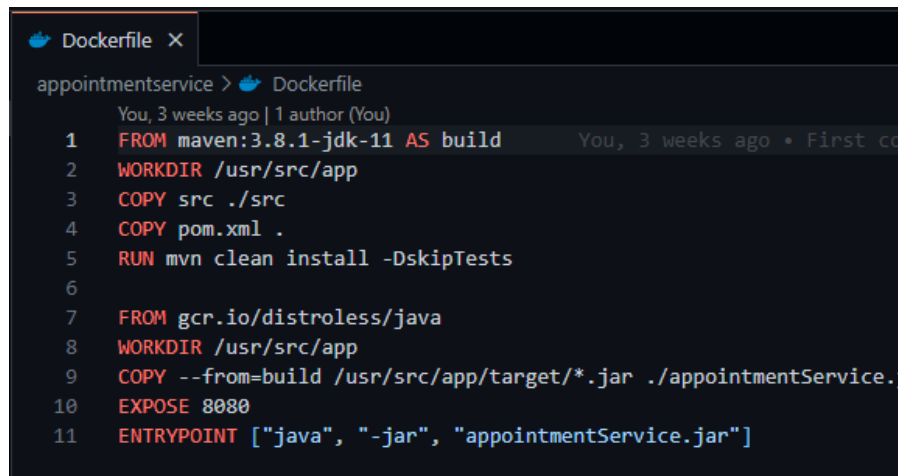
```
Dockerfile X
doctorservice > Dockerfile
You, 3 weeks ago | 1 author (You)
1 FROM maven:3.8.1-jdk-11 AS build
2 WORKDIR /usr/src/app
3 COPY src ./src
4 COPY pom.xml .
5 RUN mvn clean install -DskipTests
6
7 FROM gcr.io/distroless/java
8 WORKDIR /usr/src/app
9 COPY --from=build /usr/src/app/target/*.jar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

Fig 3.1: Dockerfile for DoctorService



```
Dockerfile X
userservice > 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 ./userService.jar
10 EXPOSE 8080
11 ENTRYPOINT ["java", "-jar", "userService.jar"]
```

Fig 3.2: Dockerfile for UserService



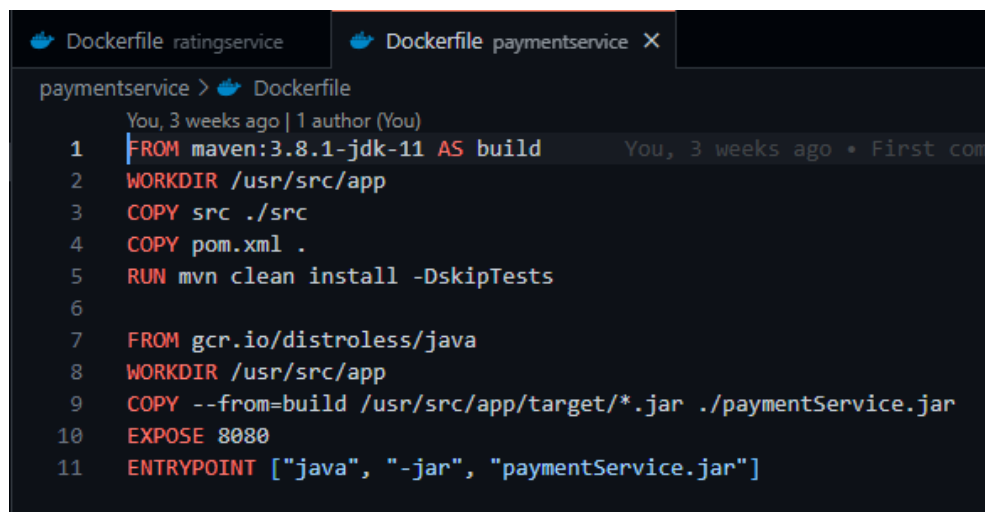
The screenshot shows a code editor with a tab labeled 'Dockerfile'. The content of the Dockerfile is as follows:

```

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



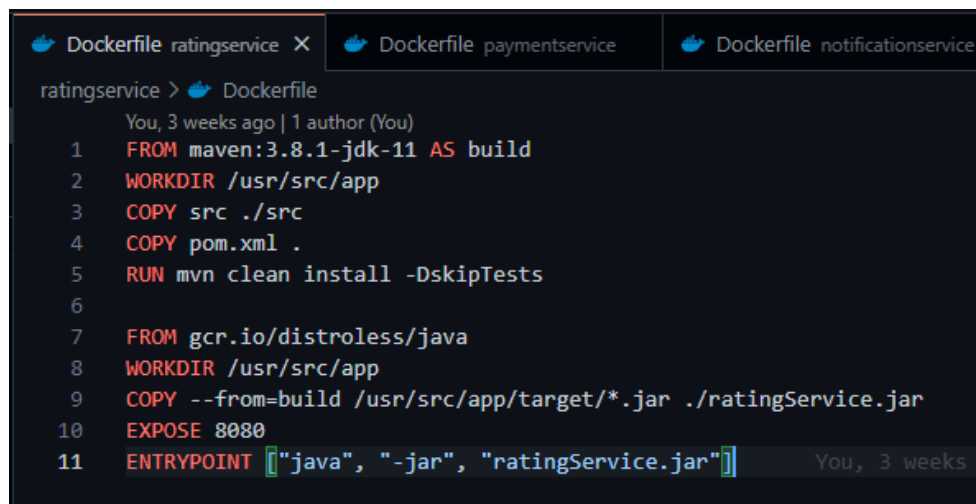
The screenshot shows a code editor with a tab labeled 'Dockerfile paymentservice'. The content of the Dockerfile is as follows:

```

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



The screenshot shows a code editor with a tab labeled 'Dockerfile ratingservice'. The content of the Dockerfile is as follows:

```

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 You, 3 weeks ago • First commit ...
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
      - KAFKA_HOST
      - INSTANCE_IP
    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
      - KAFKA_HOST
      - INSTANCE_IP
    networks:
      - app-tier
  appointment:
    build: appointmentService
    container_name: appointmentService
    image: book_my_consultation/appointmentService:1.0.0
    ports:
      - "8083:8083"
    environment:
      - MYSQL_HOST
      - MONGO_HOST
      - KAFKA_HOST
      - INSTANCE_IP
    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"
    environment:
      - KAFKA_HOST
      - INSTANCE_IP
    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
      - KAFKA_HOST
      - INSTANCE_IP
    depends_on:
      - doctor
    networks:
      - app-tier
  notification:
    build: notificationService
    container_name: notificationService
    image: book_my_consultation/notificationService:1.0.0
    environment:
      - KAFKA_HOST
      - INSTANCE_IP
    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 first have to set the ip of the kafka server within the server properties.

A terminal window with a dark blue background and light blue text. At the top, it says "Uploaded using RayThis Extension". Below that, the command `vi config/server.properties` is entered in green text.

```
Uploaded using RayThis Extension
vi config/server.properties
```

Fig 5.1: Update kafka server properties

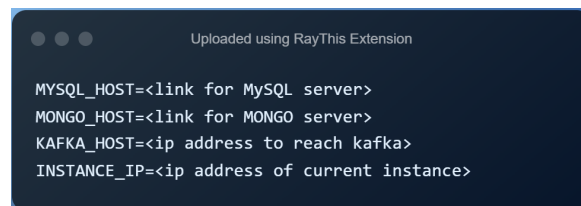
Following that, we run the following commands.

A terminal window with a dark blue background and light blue text. At the top, it says "Uploaded using RayThis Extension". Below that, two commands are entered in green text: `bin/zookeeper-server-start.sh config/zookeeper.properties` and `bin/kafka-server-start.sh config/server.properties`.

```
Uploaded using RayThis Extension
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
```

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.

A terminal window with a dark blue background and light blue text. At the top, it says "Uploaded using RayThis Extension". Below that, four environment variables are listed in green text: `MYSQL_HOST=<link for MySQL server>`, `MONGO_HOST=<link for MONGO server>`, `KAFKA_HOST=<ip address to reach kafka>`, and `INSTANCE_IP=<ip address of current instance>`.

```
Uploaded using RayThis Extension
MYSQL_HOST=<link for MySQL server>
MONGO_HOST=<link for MONGO server>
KAFKA_HOST=<ip address to reach kafka>
INSTANCE_IP=<ip address of current instance>
```

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

A terminal window with a dark blue background and light blue text. At the top, it says "Uploaded using RayThis Extension". Below that, the command `docker-compose build` is entered in green text.

```
Uploaded using RayThis Extension
docker-compose build
```

Fig 5.4: build service images

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

A terminal window with a dark blue background and light blue text. At the top, it says "Uploaded using RayThis Extension". Below that, the command `docker-compose up` is entered in green text.

```
Uploaded using RayThis Extension
docker-compose up
```

Fig 5.5: 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 ratingService ... 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              ac00d2e54f6e       24 hours ago       485MB
<none>                                <none>             e0a454370c32       24 hours ago       1.6GB
book_my_consultation/paymentService 1.0.0              cd56cd792d506       27 hours ago       270MB
book_my_consultation/ratingService  1.0.0              227d860647bc       27 hours ago       274MB
book_my_consultation/doctorService  1.0.0              e19ecbc33dbd       27 hours ago       486MB
book_my_consultation/notificationService 1.0.0              a78517a8851b       28 hours ago       472MB
<none>                                <none>             5798ca37e6ed       28 hours ago       485MB
book_my_consultation/appointmentService 1.0.0              46806a07419f       28 hours ago       501MB
maven                               3.8.1-jdk-11       5b508b1fa19e       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   IMAGE                                     COMMAND                  CREATED        STATUS        PORTS                                     NAMES
932ba6d28539   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
e0e1c77a5c3b   book_my_consultation/ratingService:1.0.0     "java -jar ratingSer..." 24 hours ago   Up 16 seconds   8080/tcp, 0.0.0.0:8085->8085/tcp, :::8085->8085/tcp   ratingService
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
5c76089b5a1c   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
a4eb27b49f26   book_my_consultation/notificationService:1.0.0 "java -jar notificat..." 24 hours ago   Up 17 seconds   8080/tcp                                     notificationService
ubuntu@ip-10-0-0-114:~/BookMyConsultation$

```

Fig 5.6: 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.

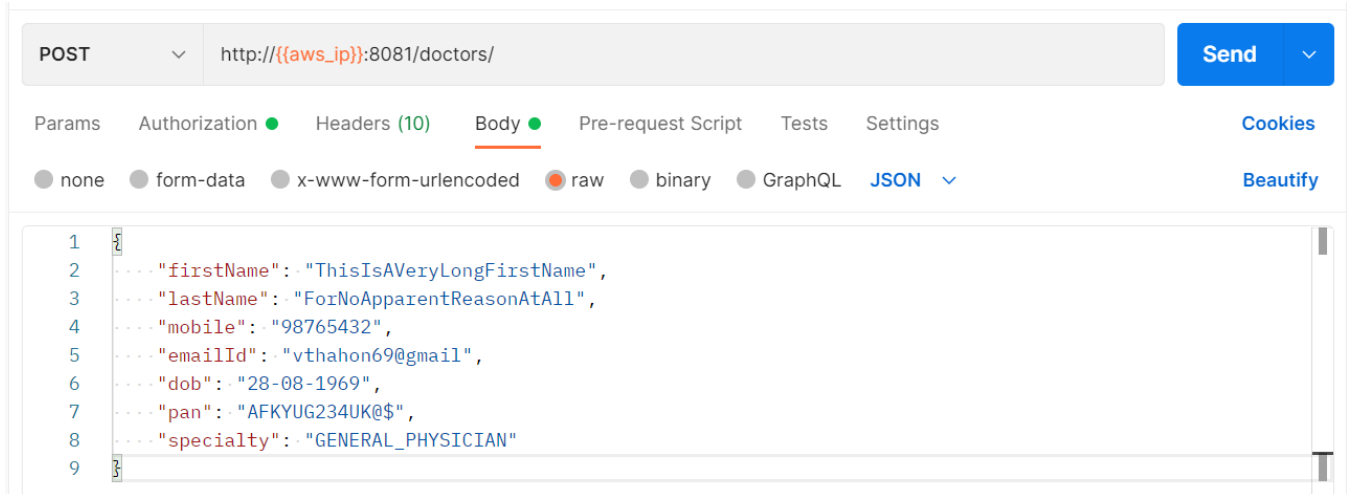


Fig 6.1a: Incorrect Details to create doctor

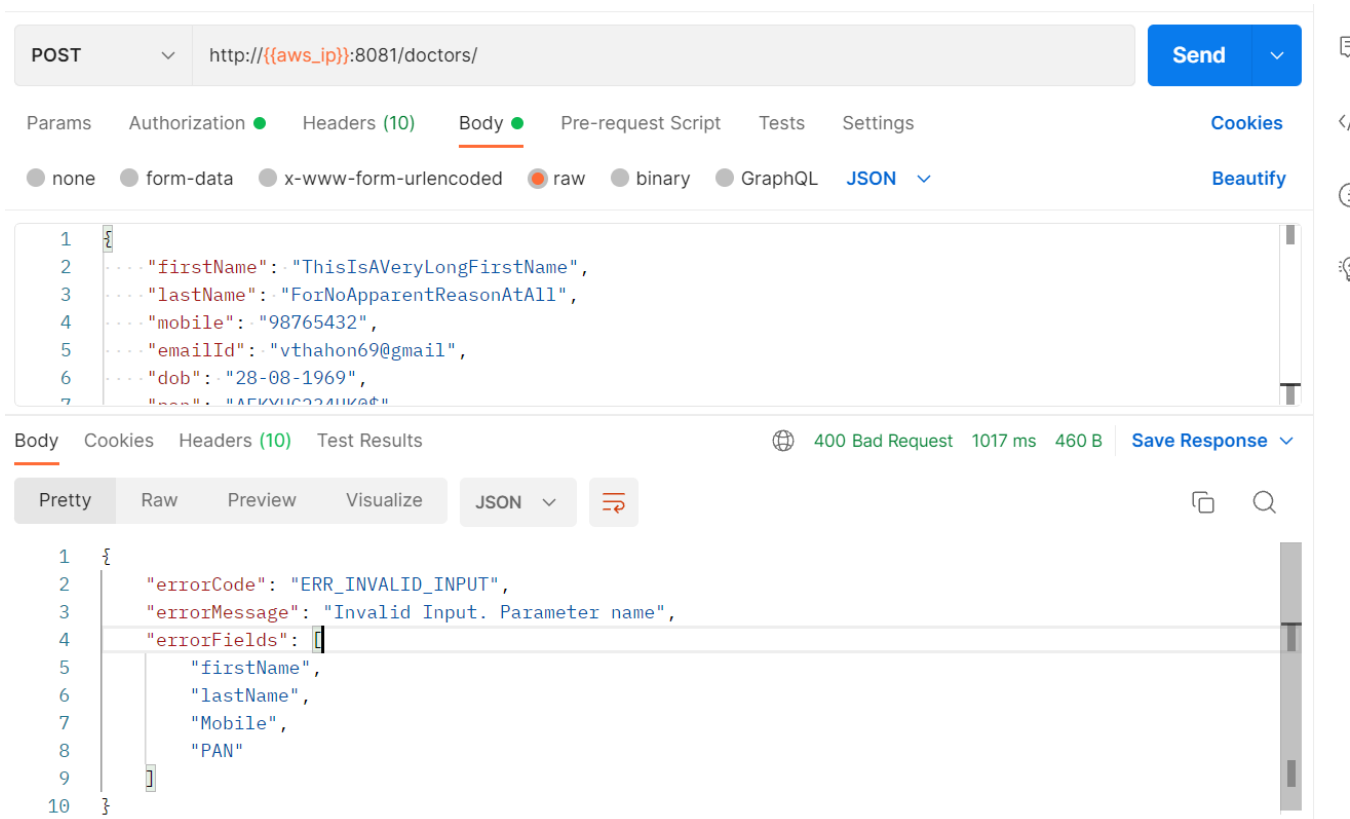


Fig 6.1b: Response for incorrect details to create Doctor

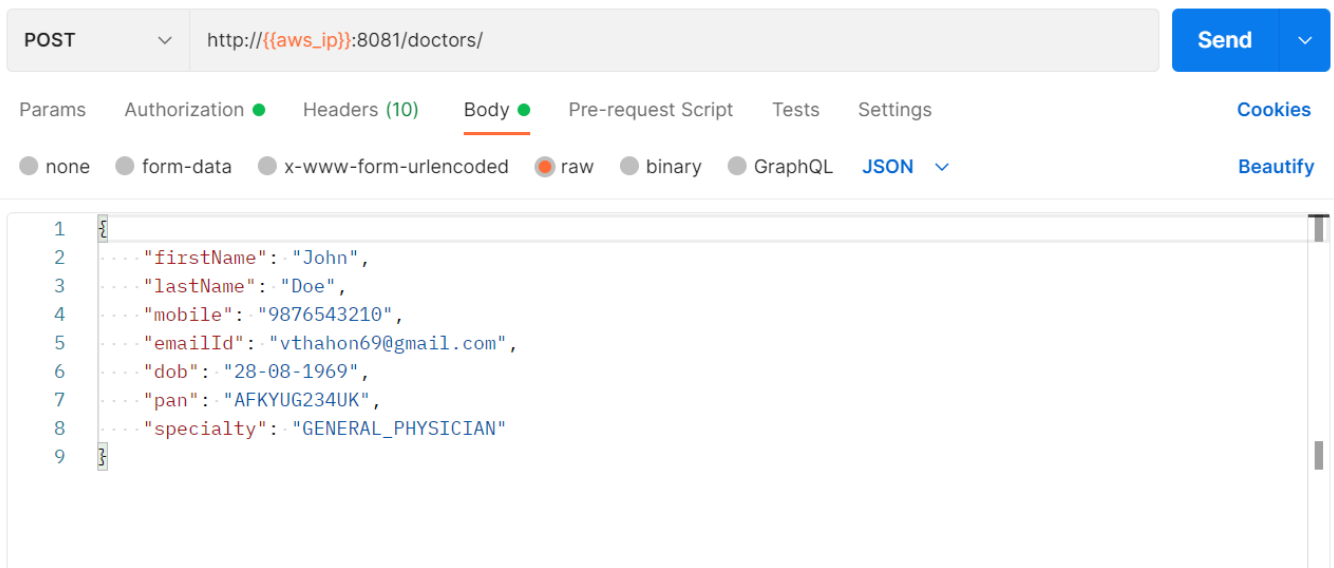


Fig 6.1c: Correct details to create doctor

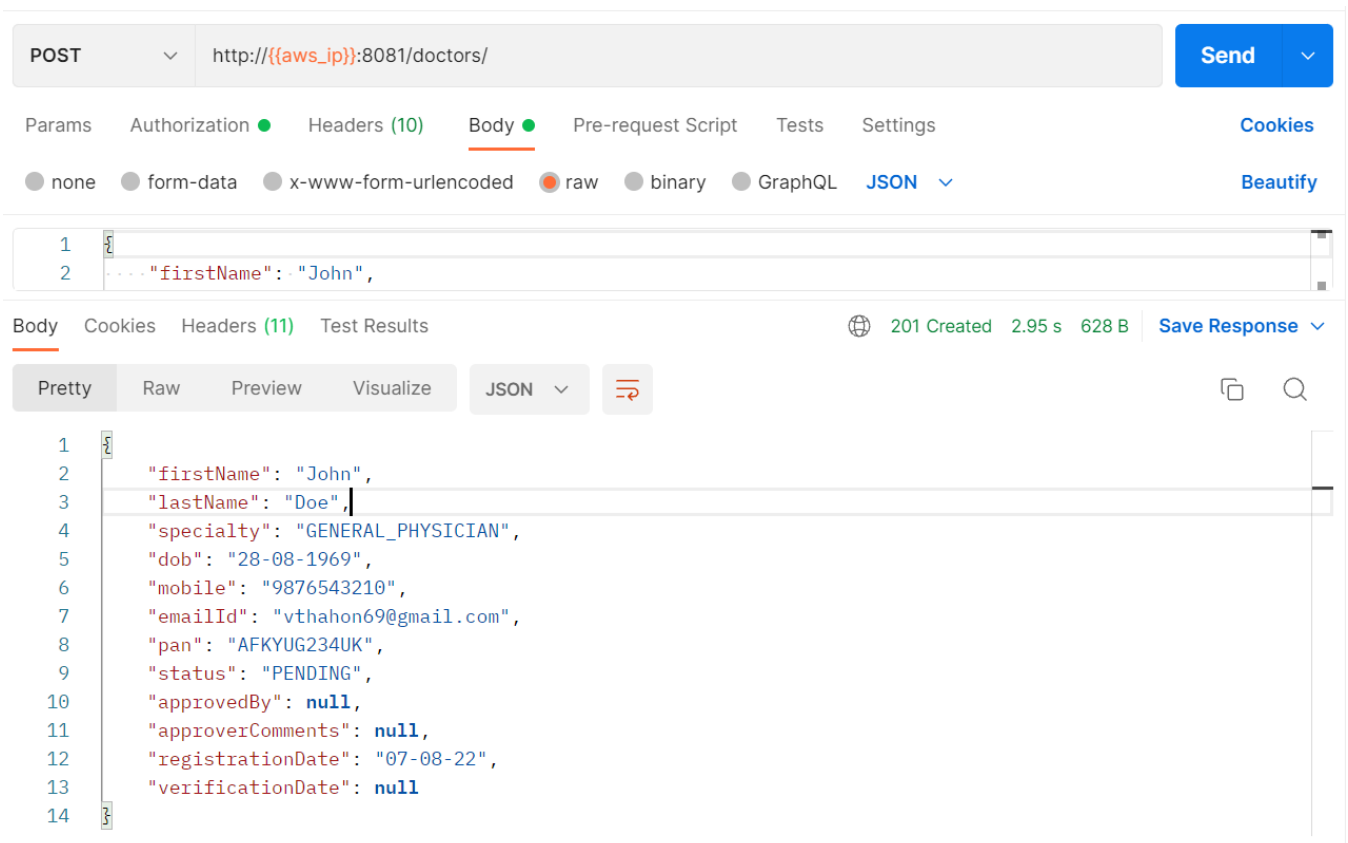


Fig 6.1d: Response for correct details to create doctor

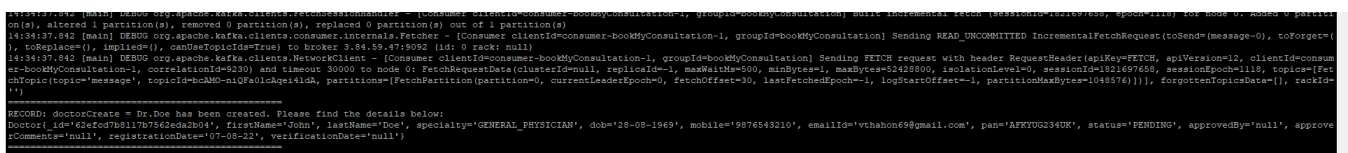




Fig 6.1e: Notification sent to Kafka server for Doctor creation

Amazon Web Services – Email Address Verification Request in region US East (N. Virginia)

 **Amazon Web Services** Thu, Aug 4, 5:09 PM (4 days ago) ☆
Dear Amazon Web Services Customer, We have received a request to authorize this email address for use with Amazon SES and Amazon Pinpoint ...





 **Amazon Web Services** <no-reply-aws@amazon.com> Thu, Aug 4, 6:50 PM (4 days ago) ☆ ← ⋮
to me ▾
Dear Amazon Web Services Customer,




We have received a request to authorize this email address for use with Amazon SES and Amazon Pinpoint in region US East (N. Virginia). If you requested this verification, please go to the following URL to confirm that you are authorized to use this email address:



https://email-verification.us-east-1.amazonaws.com/?Context=554109366355&X-Amz-Date=20220804T132022Z&Identity.IdentityName=vtahon69%40gmail.com&X-Amz-Algorithm=AWS4-HMAC-SHA256&Identity.IdentityType=EmailAddress&X-Amz-SignedHeaders=host&X-Amz-Credential=AKIAVM67ZIEFRDECB3HF%2F20220804%2Fus-east-1%2Fses%2Faws4_request&Operation=ConfirmVerification&Namespace=Bacon&X-Amz-Signature=286ab1f34bd58c7174f9e81baf7c338b3fa2b7d6dd62279c14313244ee1ceb01
...







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

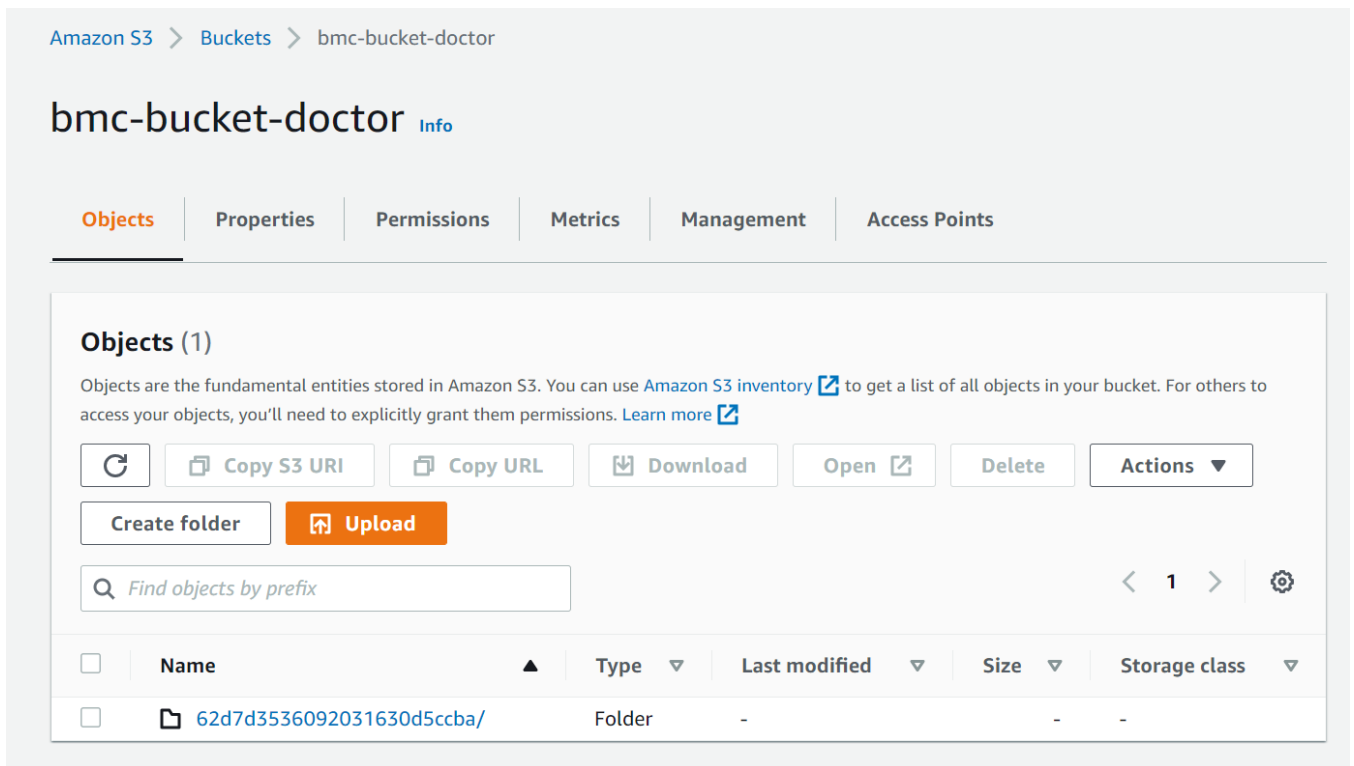


Fig 6.2b: Documents uploaded to Amazon S3

3. Approve Doctor

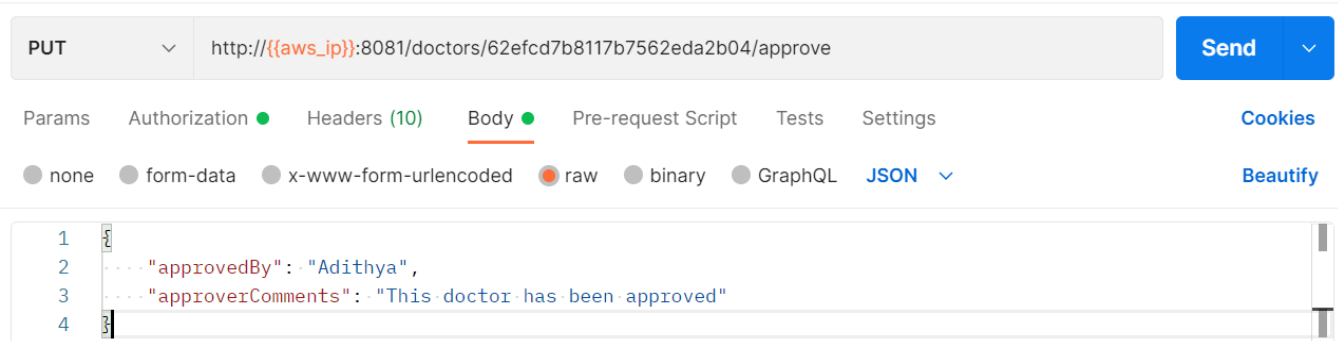


Fig 6.3a: Request to approve doctor

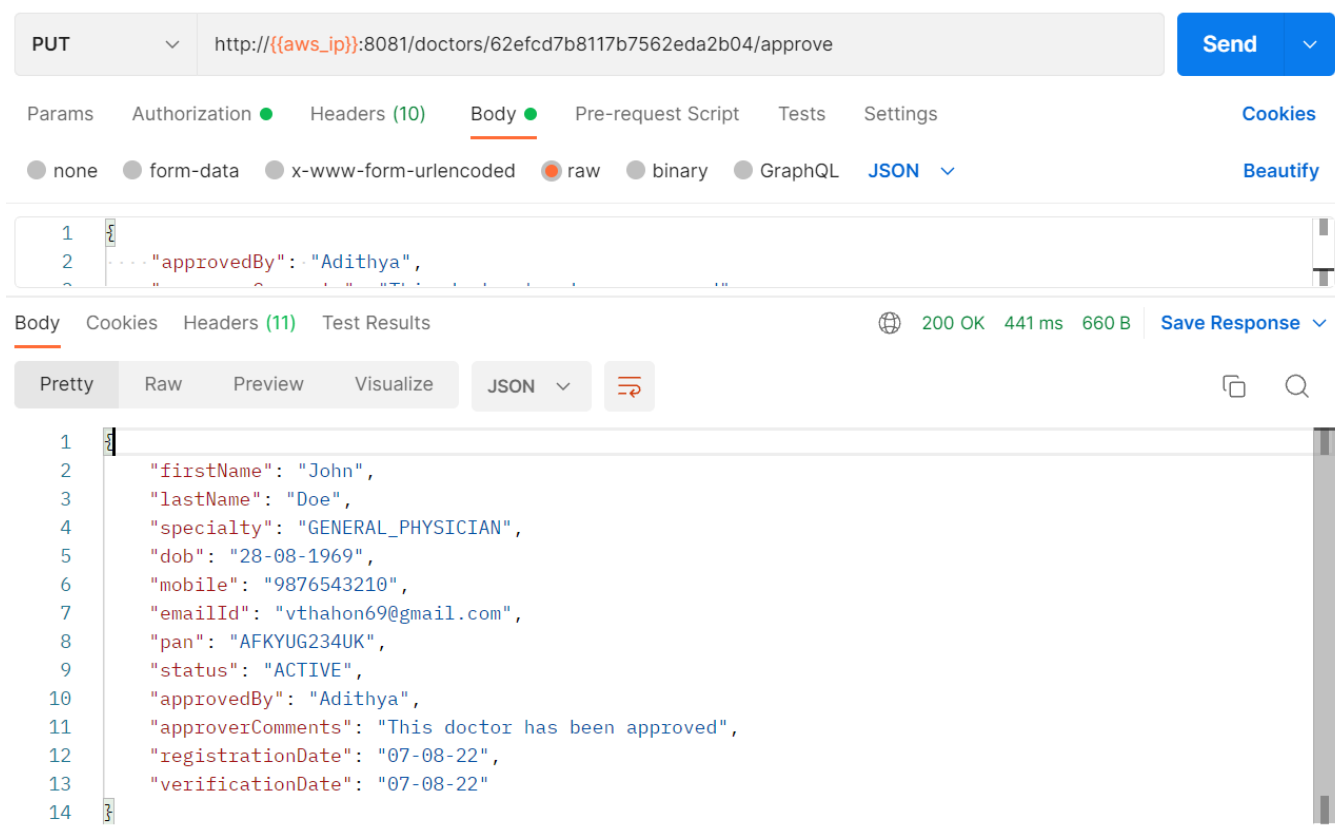


Fig 6.3b: Response for approving doctor

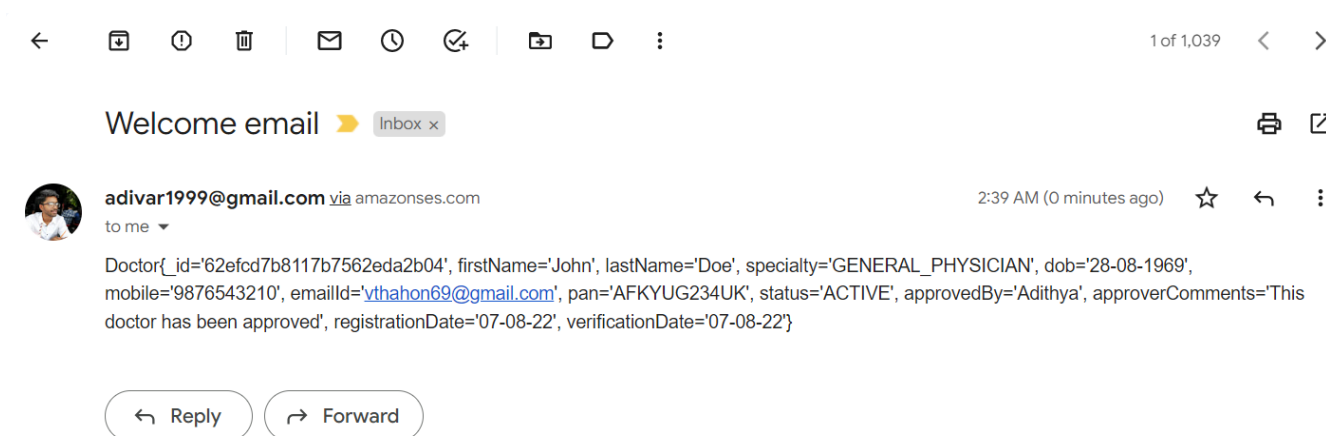


Fig 6.3c: Email for Doctor Approval

4. Reject Doctor

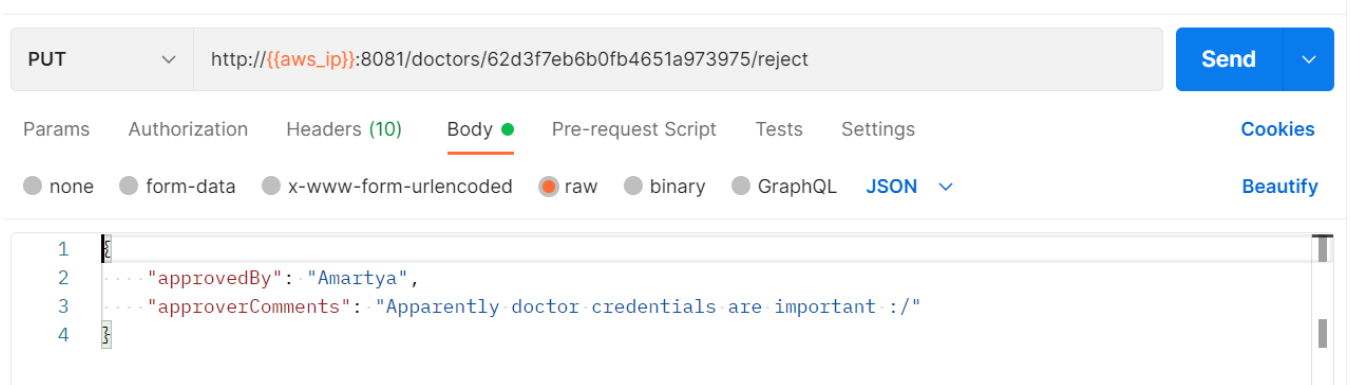


Fig 6.4a: Request to reject Doctor

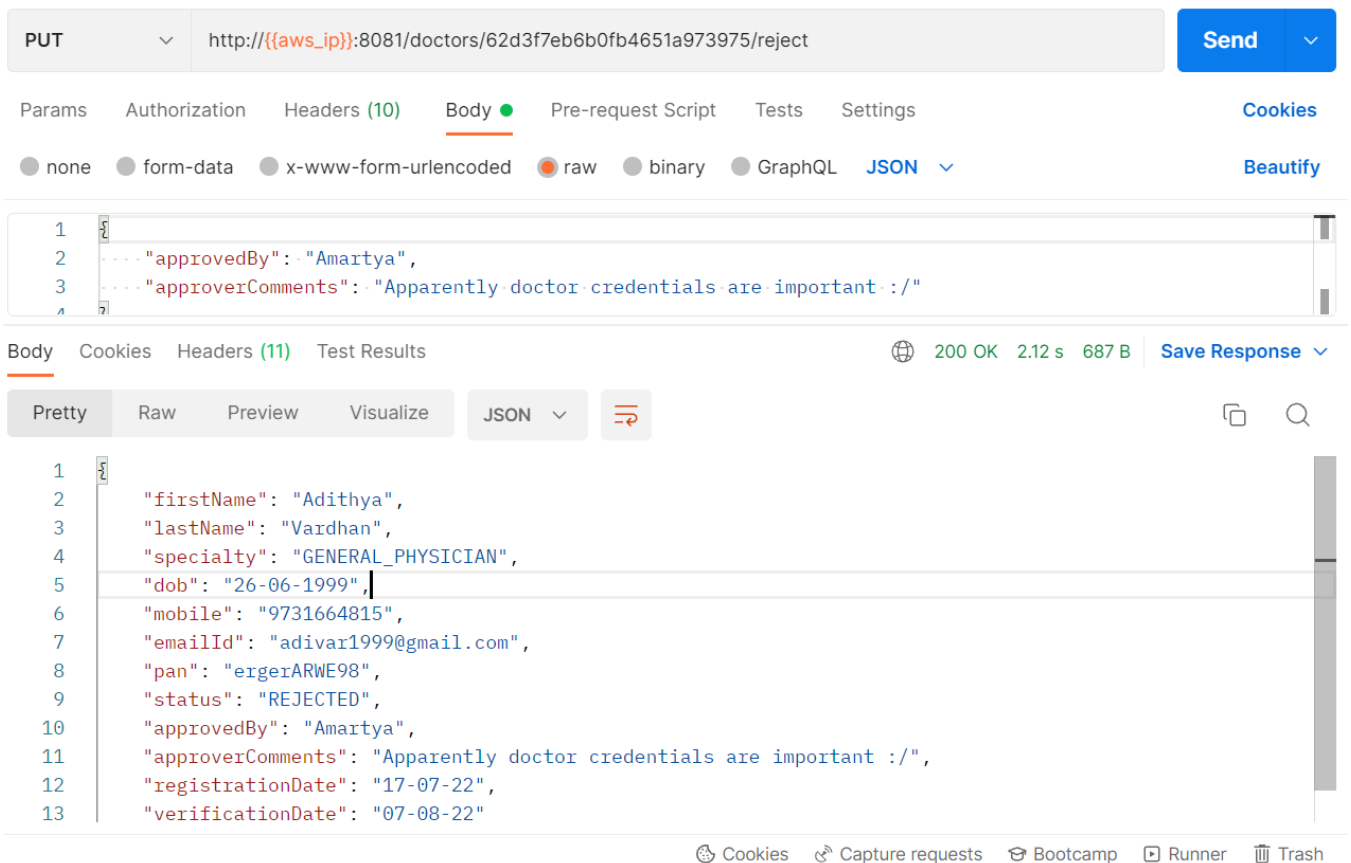


Fig 6.4b: Response to reject Doctor

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 is a JSON array containing one doctor object. The response status is 200 OK, with a response time of 1802 ms and a body size of 2.43 KB.

KEY	VALUE	DESCRIPTION
status	ACTIVE	
specialty	DENTIST	
Key	Value	Description

```
1 [
2   {
3     "firstName": "Amartya",
4     "lastName": "Vardhan",
5     "specialty": "GENERAL_PHYSICIAN",
6     "dob": "26-06-1999",
7     "mobile": "9880165936",
8     "emailId": "amartya2606@gmail.com",
9     "pan": "4456341AC000354"
```

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 is a JSON array containing one doctor object. The response status is 200 OK, with a response time of 270 ms and a body size of 661 B.

KEY	VALUE	DESCRIPTION
status	ACTIVE	
specialty	DENTIST	
Key	Value	Description

```
1 [
2   {
3     "firstName": "Meghana",
4     "lastName": "Mohan",
5     "specialty": "DENTIST",
6     "dob": "11-09-1996",
7     "mobile": "9535597858",
8     "emailId": "vthahon69@gmail.com",
9     "pan": "reog340394jf",
10    "status": "ACTIVE",
11    "approvedBy": "Adithya",
12    "approverComments": "My Sister, so bigger nepotism ;)",
13    "registrationDate": "20-07-22",
14    "verificationDate": "06-08-22"
15  }
16 ]
```

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. At the top, a GET request is configured with the URL `http://{{aws_ip}}:8081/doctors/62efcd7b8117b7562eda2b04`. The 'Body' tab is selected, showing the message 'This request does not have a body'. Below the request, the response is shown in the 'Body' tab, displaying a JSON object with 11 headers. The response status is 200 OK, with a response time of 494 ms and a body size of 660 B. The JSON response is as follows:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "specialty": "GENERAL_PHYSICIAN",
  "dob": "28-08-1969",
  "mobile": "9876543210",
  "emailId": "vthahon69@gmail.com",
  "pan": "AFKYUG234UK",
  "status": "ACTIVE",
  "approvedBy": "Adithya",
  "approverComments": "This doctor has been approved",
  "registrationDate": "07-08-22",
  "verificationDate": "07-08-22"
}
```

Fig 6.6: Request and response to get doctor details

UserService

1. Create User

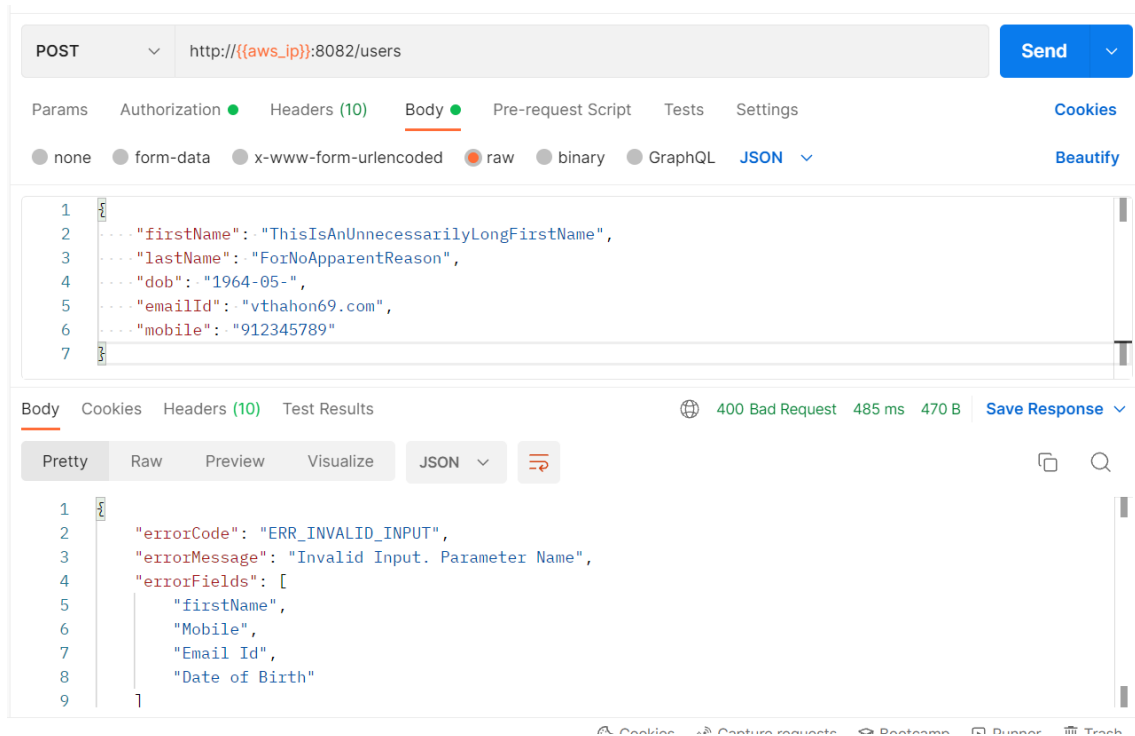


Fig 7.1a: Request and response for incorrect entries for Create User

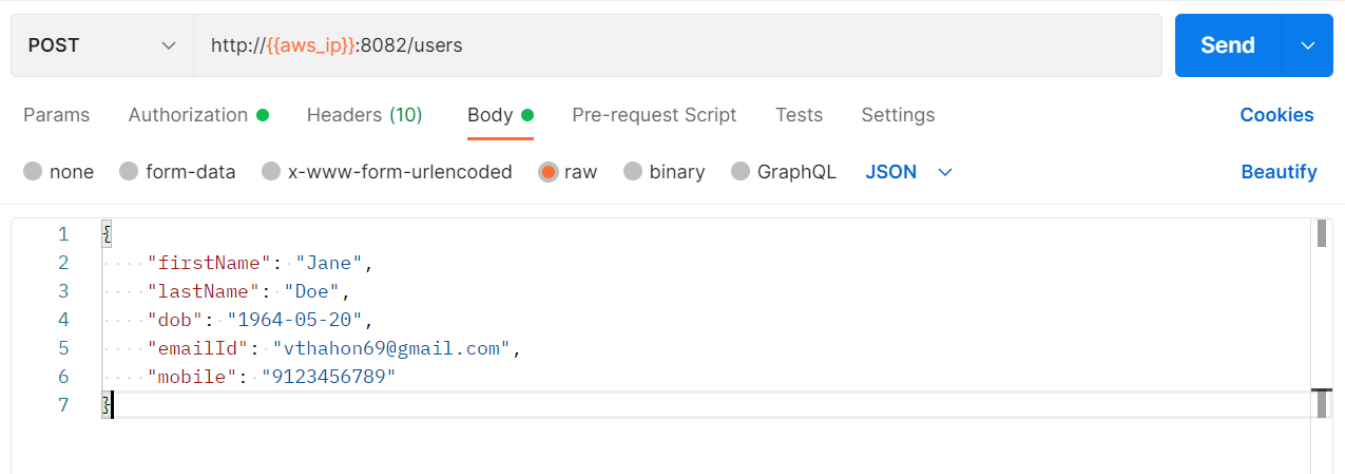


Fig 7.1b: Correct Request for Create User

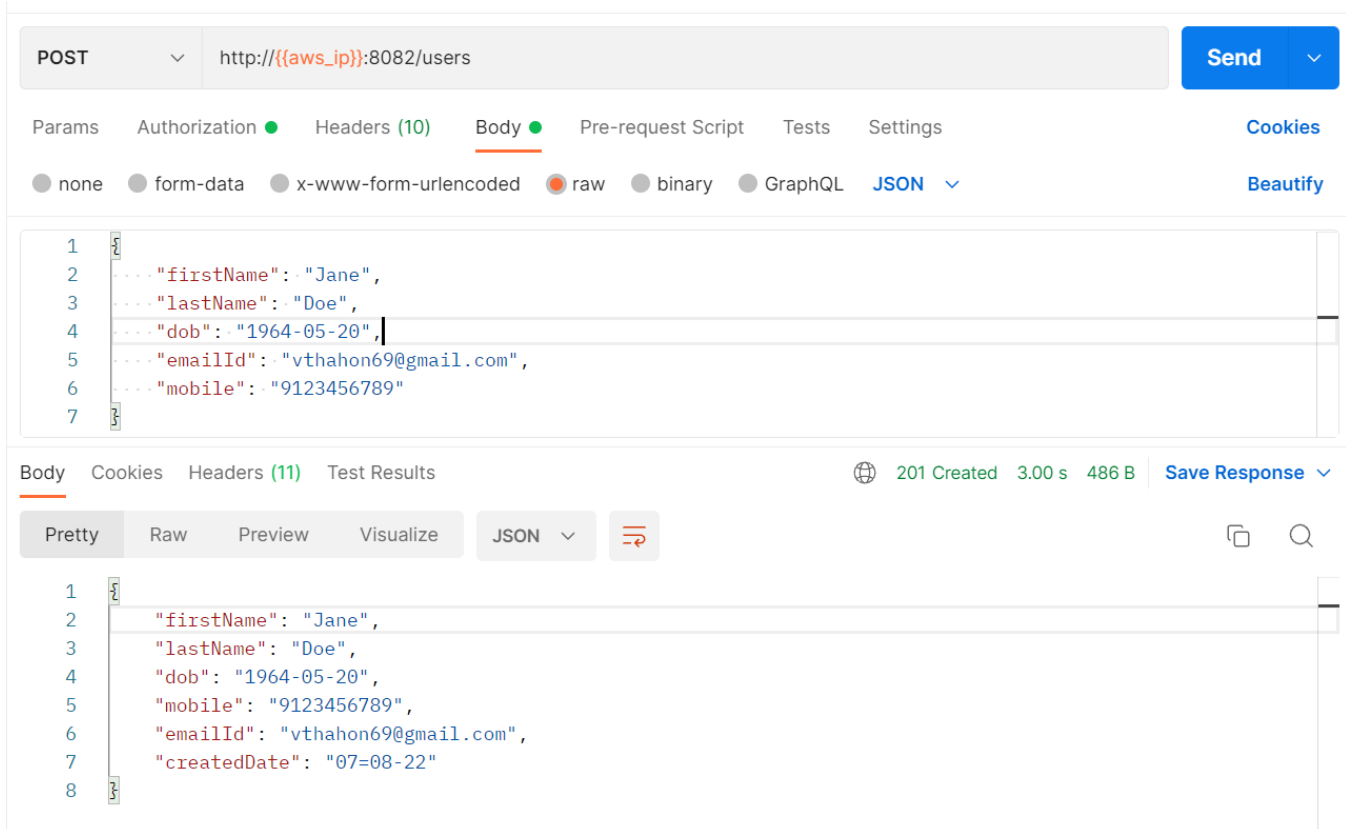


Fig 7.1c: Response for correct Create User

```


msg = javax.mail.internet.MimeMessage@5e2783e
Trying to send to AWS: (AKIAYCA356BJVGTRGNXZ:BIM1+ufUEjbbjanguJswizj6FVzRG6rBWLKs5NucqsRSA)
=====
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

Amazon Web Services – Email Address Verification Request in region US East (N. Virginia) 🖨️ 🔗

Inbox x



Amazon Web Services
<no-reply-aws@amazon.com>

3:06 AM (48 minutes ago)
☆
↶
⋮

to me ▾

Dear Amazon Web Services Customer,

We have received a request to authorize this email address for use with Amazon SES and Amazon Pinpoint in region US East (N. Virginia). If you requested this verification, please go to the following URL to confirm that you are authorized to use this email address:

https://email-verification.us-east-1.amazonaws.com/?Context=554109366355&X-Amz-Date=20220807T213638Z&Identity.IdentityName=vthahon69%40gmail.com&X-Amz-Algorithm=AWS4-HMAC-SHA256&Identity.IdentityType=EmailAddress&X-Amz-SignedHeaders=host&X-Amz-Credential=AKIAVM67ZIEFRDECB3HF%2F20220807%2Fus-east-1%2Fses%2Faws4_request&Operation=ConfirmVerification&Namespace=Bacon&X-Amz-Signature=b72e25644ce78cf8b8d6a709a71fc2e971959ace57d6dd0c4a2dbfc1c1878ae4

Your request will not be processed unless you confirm the address using this URL. This link expires 24 hours after your original verification request.

If you did NOT request to verify this email address, do not click on the link. Please note that many times, the situation isn't a phishing attempt, but either a misunderstanding of how to use our service, or someone setting up email-sending capabilities on your behalf as part of a legitimate service, but without having fully communicated the procedure first. If you are still concerned, please forward this notification to aws-email-domain-verification@amazon.com and let us know in the forward that you did not request the verification.

To learn more about sending email from Amazon Web Services, please refer to the Amazon SES Developer Guide at <https://docs.aws.amazon.com/ses/>

Fig 7.1e: Verification Email for Create user

2. getUser

GET

⌵

http://{{aws_ip}}:8082/users/62f03063822eb368927614fb

Send

⌵

Params

Authorization ●

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

● none

● form-data

● x-www-form-urlencoded

● raw

● binary

● GraphQL

This request does not have a body

Body

Cookies

Headers (11)

Test Results

🌐

200 OK

492 ms

481 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 7.2: 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 set to `form-data` and contains a file named `Adithya_Vardhan.pdf`. The response status is `200 OK` with a response time of `1792 ms` and a size of `375 B`. The response body is `File(s) uploaded Successfully`.

KEY	VALUE	DESCRIPTION
files	Adithya_Vardhan.pdf	
Key	Value	Description

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

The screenshot shows the Amazon S3 console for the bucket `bmc-bucket-user`. The `Objects` tab is selected, showing a list of objects. The object `62f03063822eb368927614fb/` is listed as a `Folder` with a size of `-` and a storage class of `-`.

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

Fig 7.3b: File uploaded to Amazon S3

AppointmentService

1. createAvailability

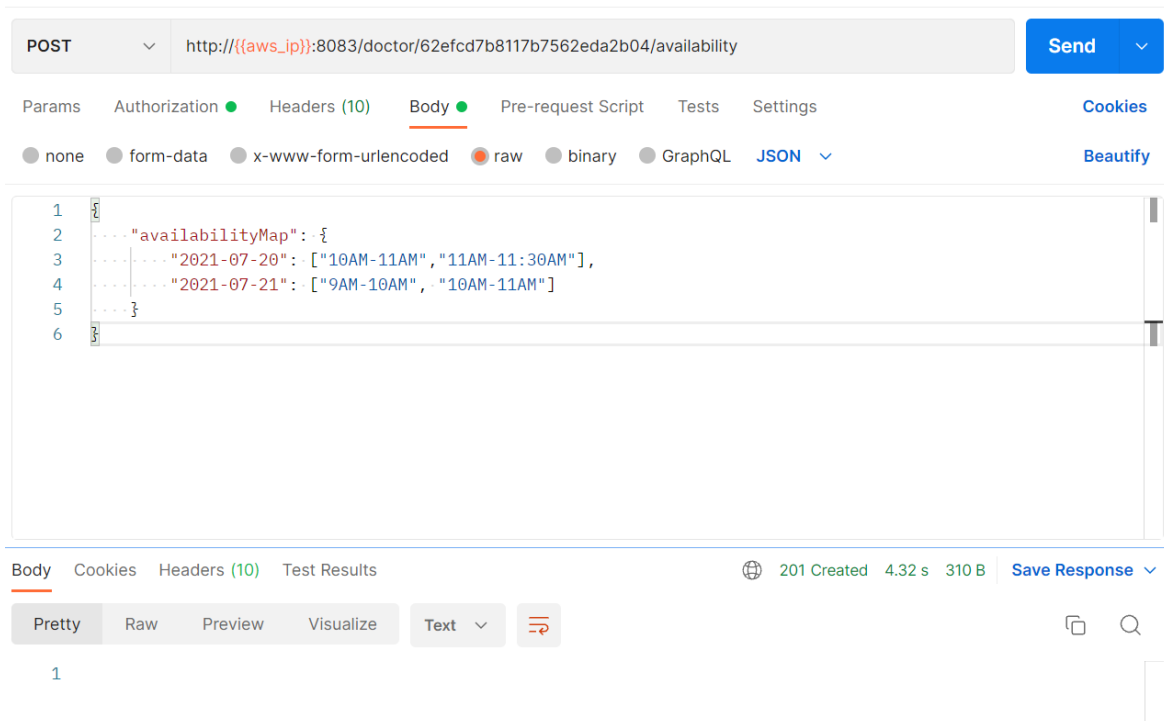


Fig 8.1: Request and response for setAvailability

2. getAvailability

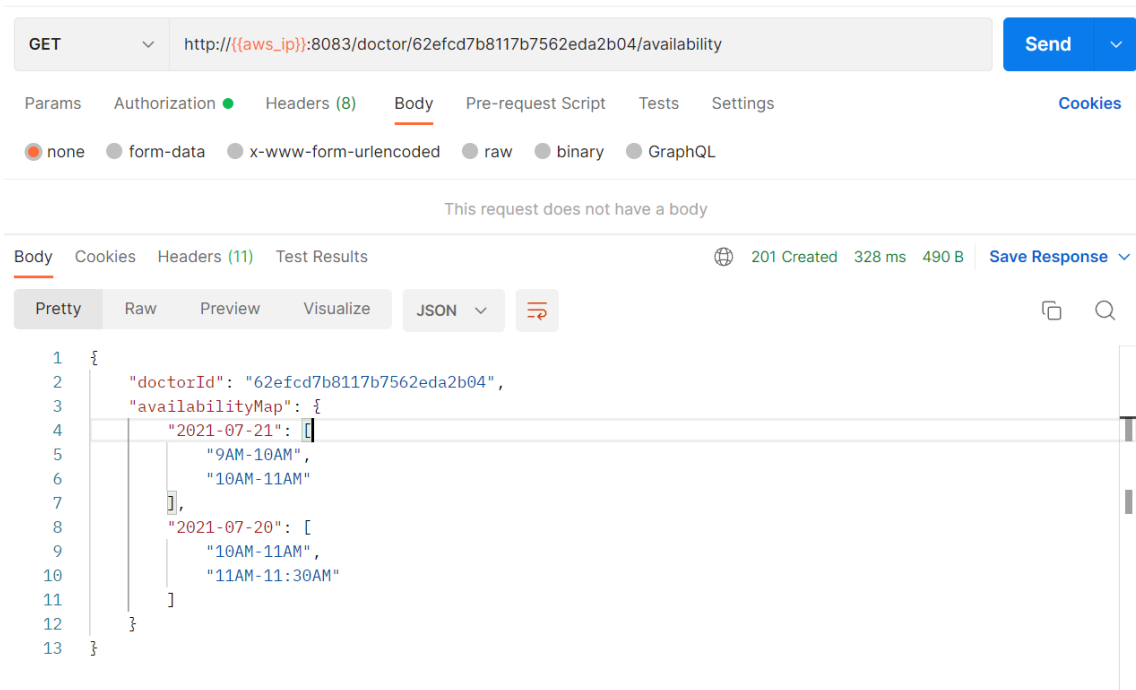


Fig 8.2: Request and response for getAvailability

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:

```
1 {
2   ... "doctor_id": "62efcd7b8117b7562eda2b04",
3   ... "user_id": "62f03063822eb368927614fb",
4   ... "appointment_date": "2021-07-21",
5   ... "time_slot": "9AM-10AM"
6 }
```

The response is a JSON object with a single field `id` containing the value `2c928085827c982f01827c9896c70000`.

Fig 8.3a: Request and response for create Appointment

The screenshot shows a Kafka log entry for a message record. The record contains the following information:

```
2022-08-08 06:36:45.701 INFO [...] 1 --- [main] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-bookMyConsultation-1, groupId=bookMyConsultation] Adding newly assigned partition
s: message-0
2022-08-08 06:36:45.701 INFO [...] 1 --- [main] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-bookMyConsultation-1, groupId=bookMyConsultation] Setting offset for partition me
ssage-0 to the committed offset FetchPosition[offset=34, offsetEpoch=Optional[0], currentLeader=LeaderAndEpoch{leader=Optional[44.201.116.228:9092 (id: 0 rack: null)], epoch=0}}
2022-08-08 06:45:45.551 INFO [...] 1 --- [main] org.apache.kafka.clients.NetworkClient : [Consumer clientId=consumer-bookMyConsultation-1, groupId=bookMyConsultation] Node -1 disconnected.
=====
RECORD: setAppointment:vthahon6@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 confirmation for creating an appointment. The email is titled "Set Appointment" and is 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 interface includes a "Reply" button and a "Forward" button.

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   {
3     "appointment_id": "2c928085827c982f01827c9896c70000",
4     "appointment_date": "2021-07-21 00:00:00.0",
5     "created_date": "2022-08-08T08:35:51.000+00:00",
6     "doctor_id": "62efcd7b8117b7562eda2b04",
7     "prior_medical_history": null,
8     "status": "PENDING_PAYMENT",
9     "symptoms": null,
10    "time_slot": "9AM-10AM",
11    "user_id": "62f03063822eb368927614fb",
12    "user_email_id": "vthahon69@gmail.com",
13    "user_name": "Jane Doe",
14    "doctor_name": "John Doe"
15  }
```

Fig 8.5: Request and response for get Appointments by user

6. createPrescription

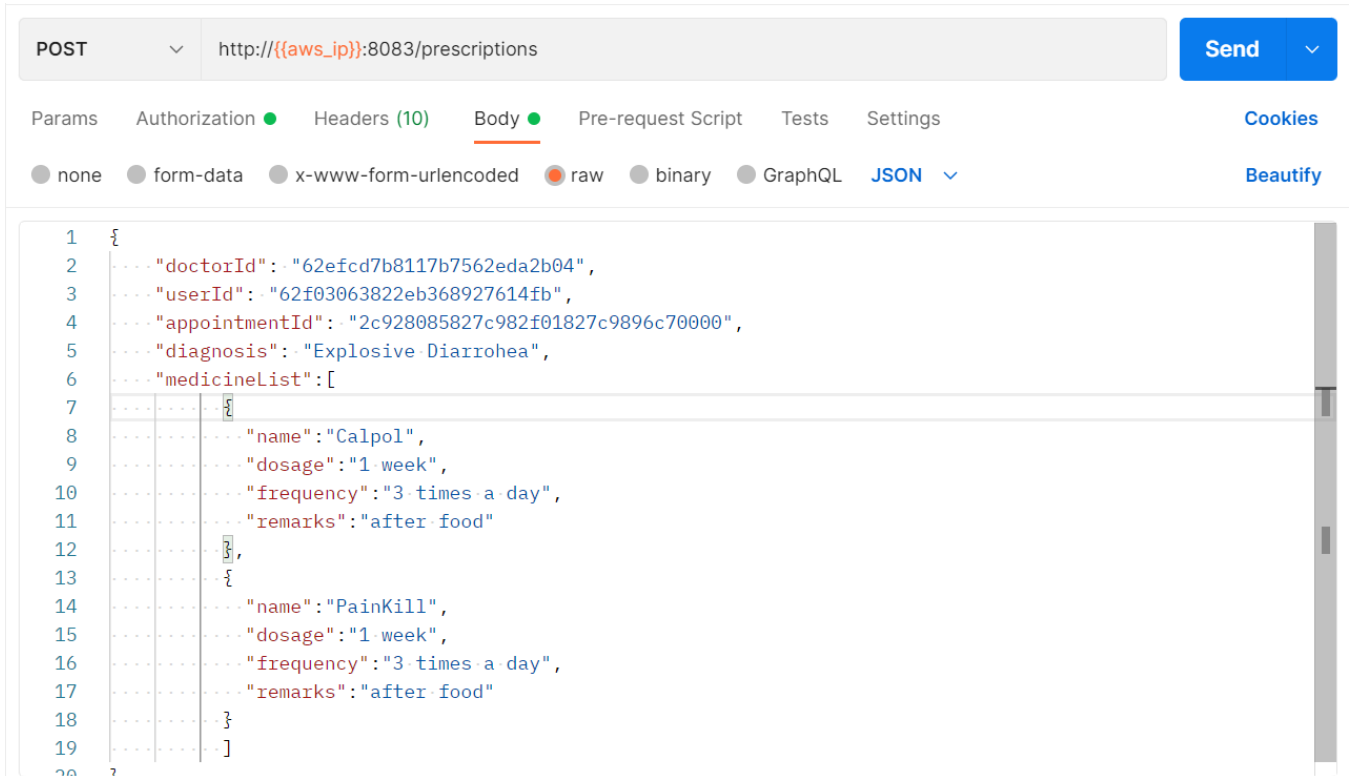


Fig 8.6a: Request for create Prescription

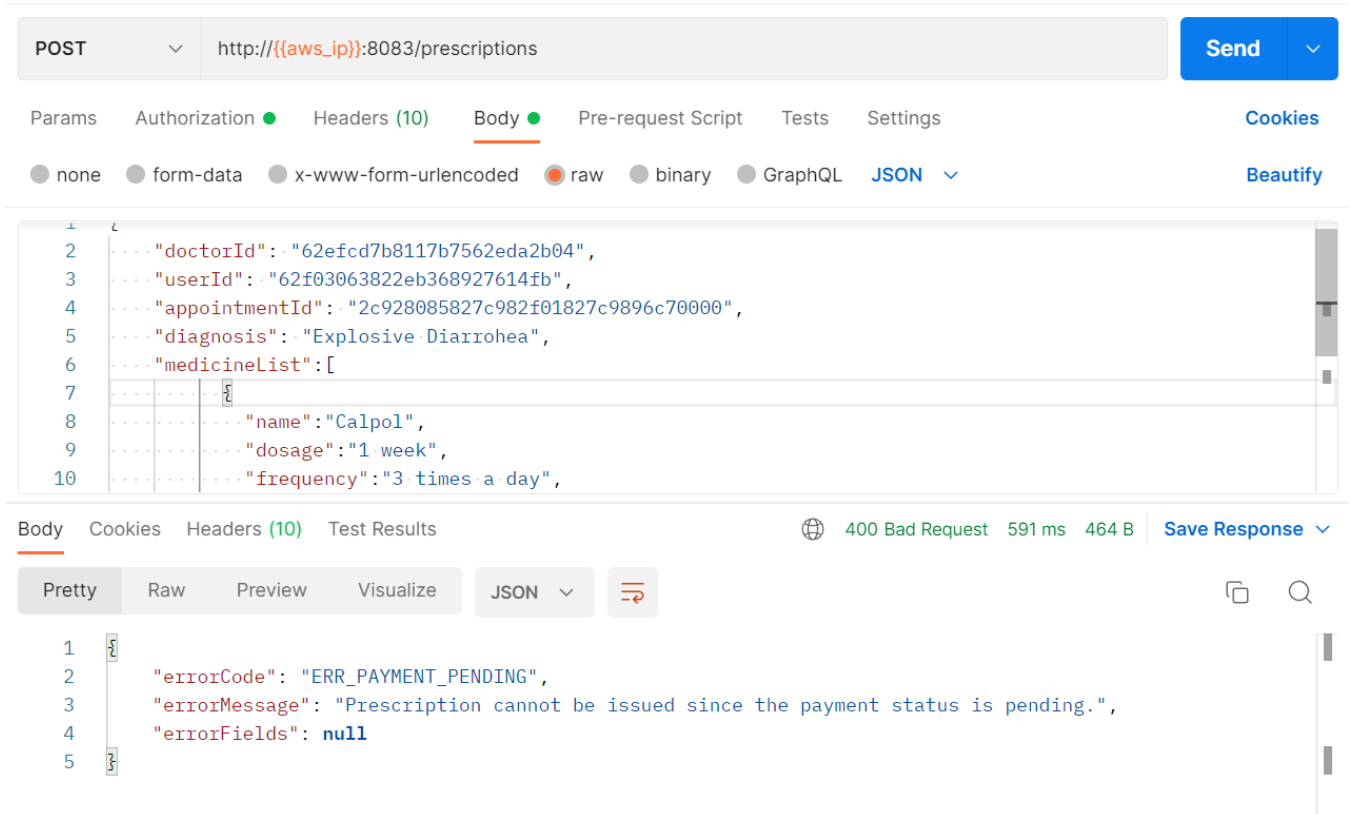


Fig 8.6b: Response when the Status of Appointment is PENDING

PaymentService

1. createPayment

The screenshot shows a REST client interface for a POST request to `http://{{aws_ip}}:8084/payments?appointmentId=2c928085827c982f01827c9896c70000`. The request is configured with the following details:

- Method:** POST
- URL:** `http://{{aws_ip}}:8084/payments?appointmentId=2c928085827c982f01827c9896c70000`
- Params:** One query parameter `appointmentId` with value `2c928085827c982f01827c9896c70000`.
- Body:** The response is displayed in JSON format:

```
{  "createdDate": "Tue Aug 09 11:41:07 IST 2022",  "appointmentId": "2c928085827c982f01827c9896c70000"}
```
- Status:** 200 OK, 2.32 s, 443 B

The interface includes tabs for Params, Authorization, Headers (9), Body, Pre-request Script, Tests, Settings, and Cookies. The Body tab is active, showing the response in JSON format. The response is a JSON object with two fields: `createdDate` and `appointmentId`.

Fig 9.1: Request and response for create Payment

RatingService

1. createRating

The screenshot shows a REST client interface for a POST request to `http://{{aws_ip}}:8085/ratings`. The request is configured with the following details:

- Method:** POST
- URL:** `http://{{aws_ip}}:8085/ratings`
- Body:** The request body is displayed in JSON format:

```
{  "doctorId": "62efcd7b8117b7562eda2b04",  "rating": 5}
```
- Status:** 200 OK, 3.06 s, 305 B

The interface includes tabs for Params, Authorization, Headers (10), Body, Pre-request Script, Tests, Settings, and Cookies. The Body tab is active, showing the request in JSON format. The response is a JSON object with two fields: `doctorId` and `rating`.

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.

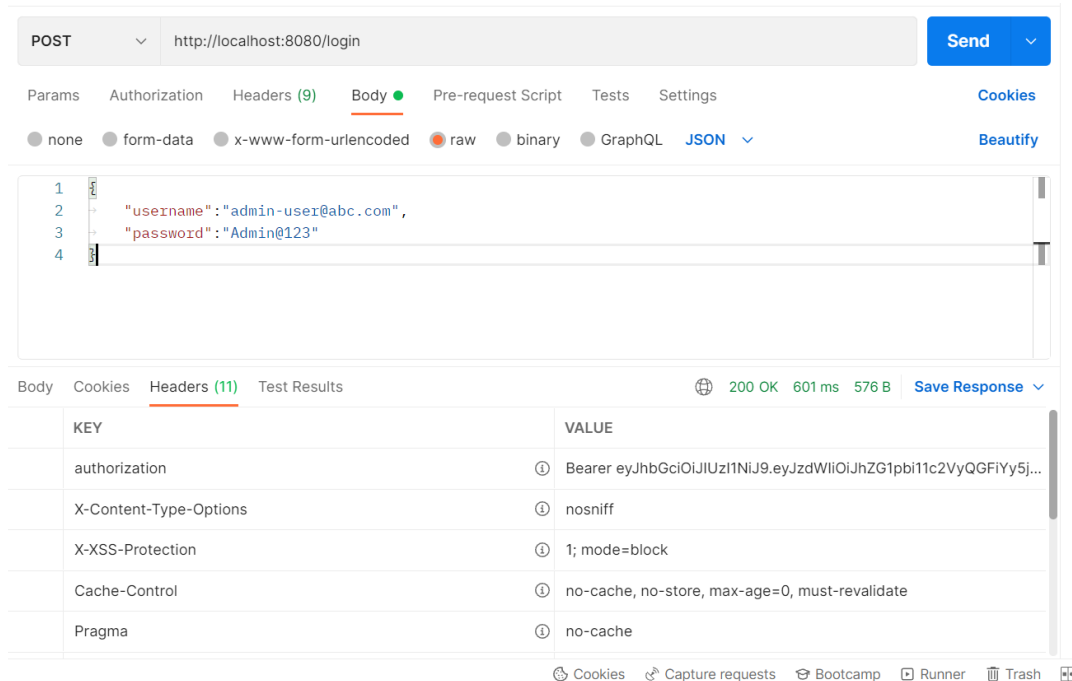


Fig 12.1: Request and response for Admin JWT token generation

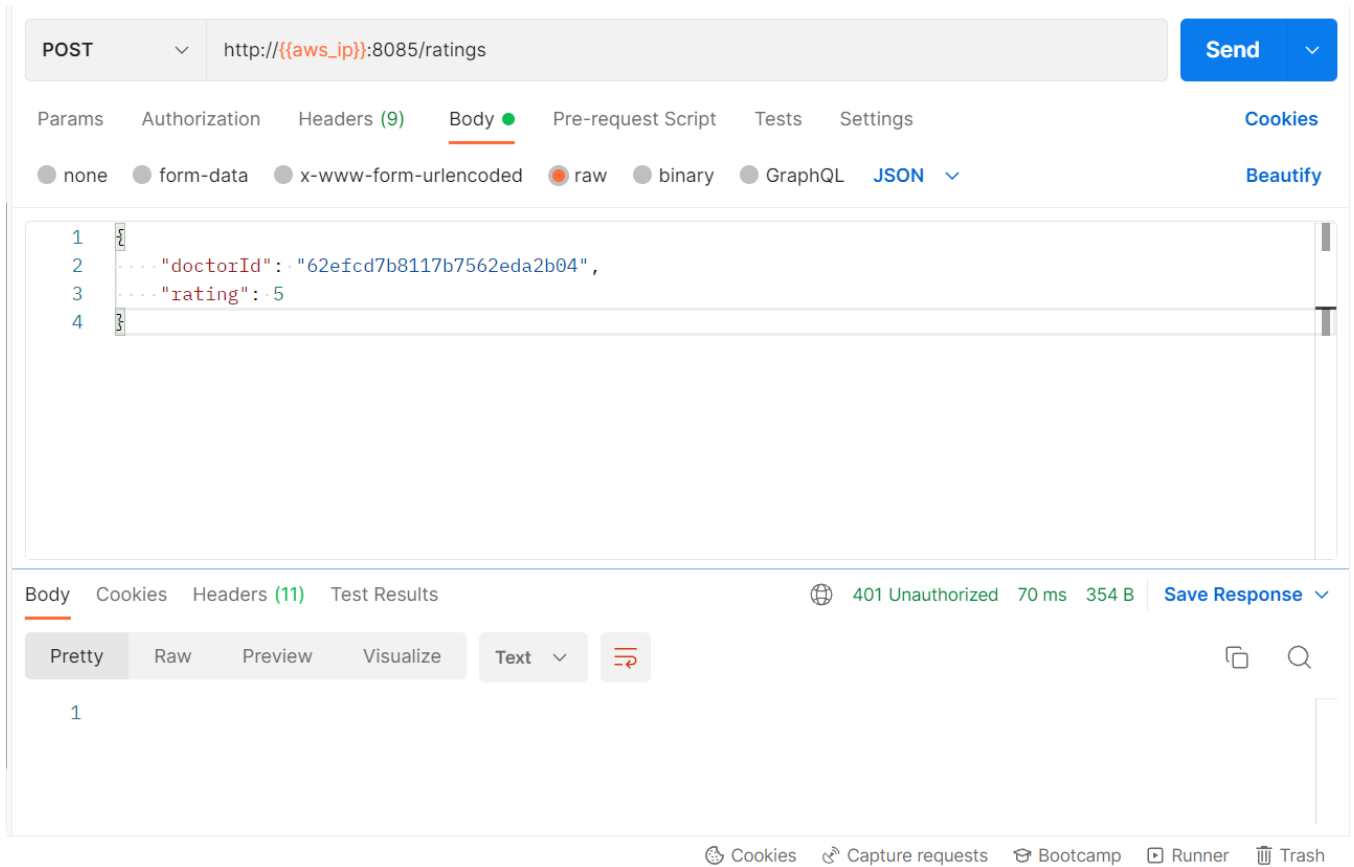


Fig 12.4: Response of an API without authorization

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.