Project Explanation

So, we have created 9 Microservices for the Book My Consultation App.

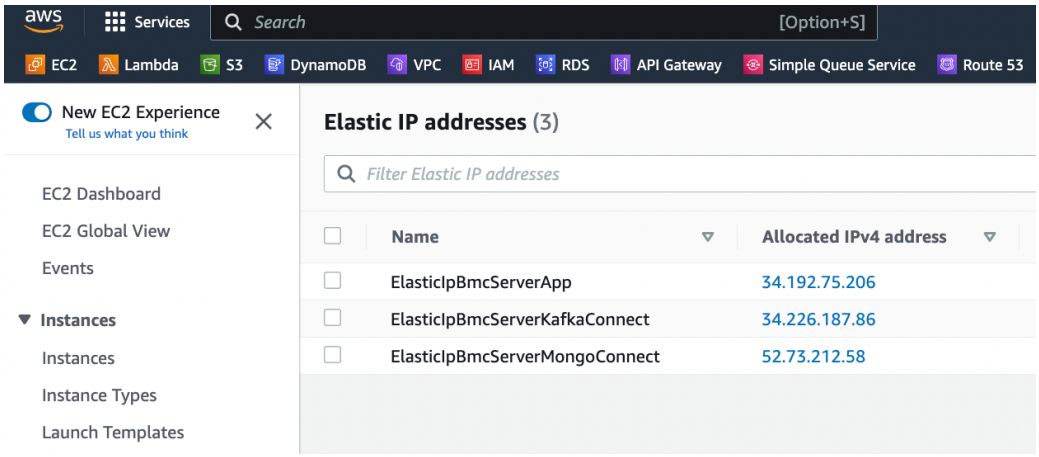
1. Service Registry: service-registry
2. BMC Gateway: bmc-gateway
3. Notification Service: notification-service
4. Rating Service: rating-service
5. User Service: user-service
6. Doctor Service: doctor-service
7. Appointment Service: appointment-service
8. Payment Service: payment-service
9. Security Provider: security-provider

Let us first explain the Deployment Process and then later let us brief you with the API Testing Screenshots, and Code Explanation by Service

Deployment Process

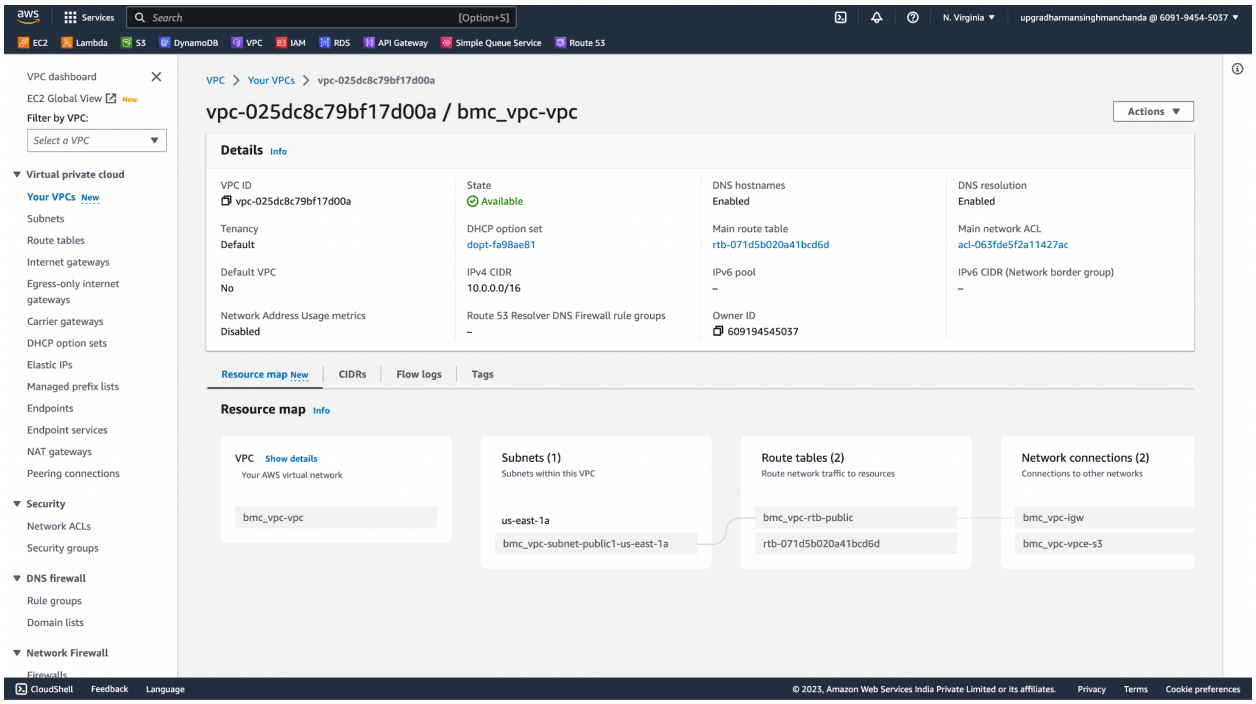
We needed to compose and deploy the whole project. For the same, please find the steps (**with setup screenshots**) that we took for the same:

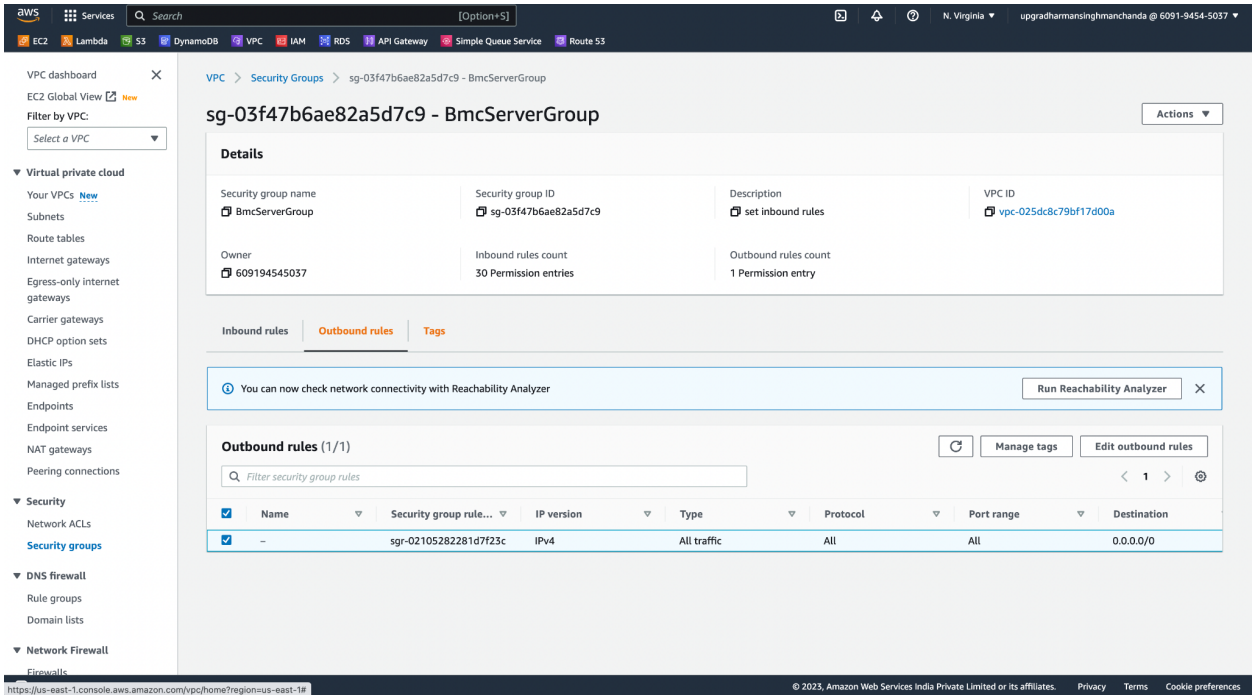
- First we created 3 elastic IP Addresses that we will use later for our 3 servers.
    - First for the Main Server
    - Second for the Kafka Server
    - Third for the Mongo Server

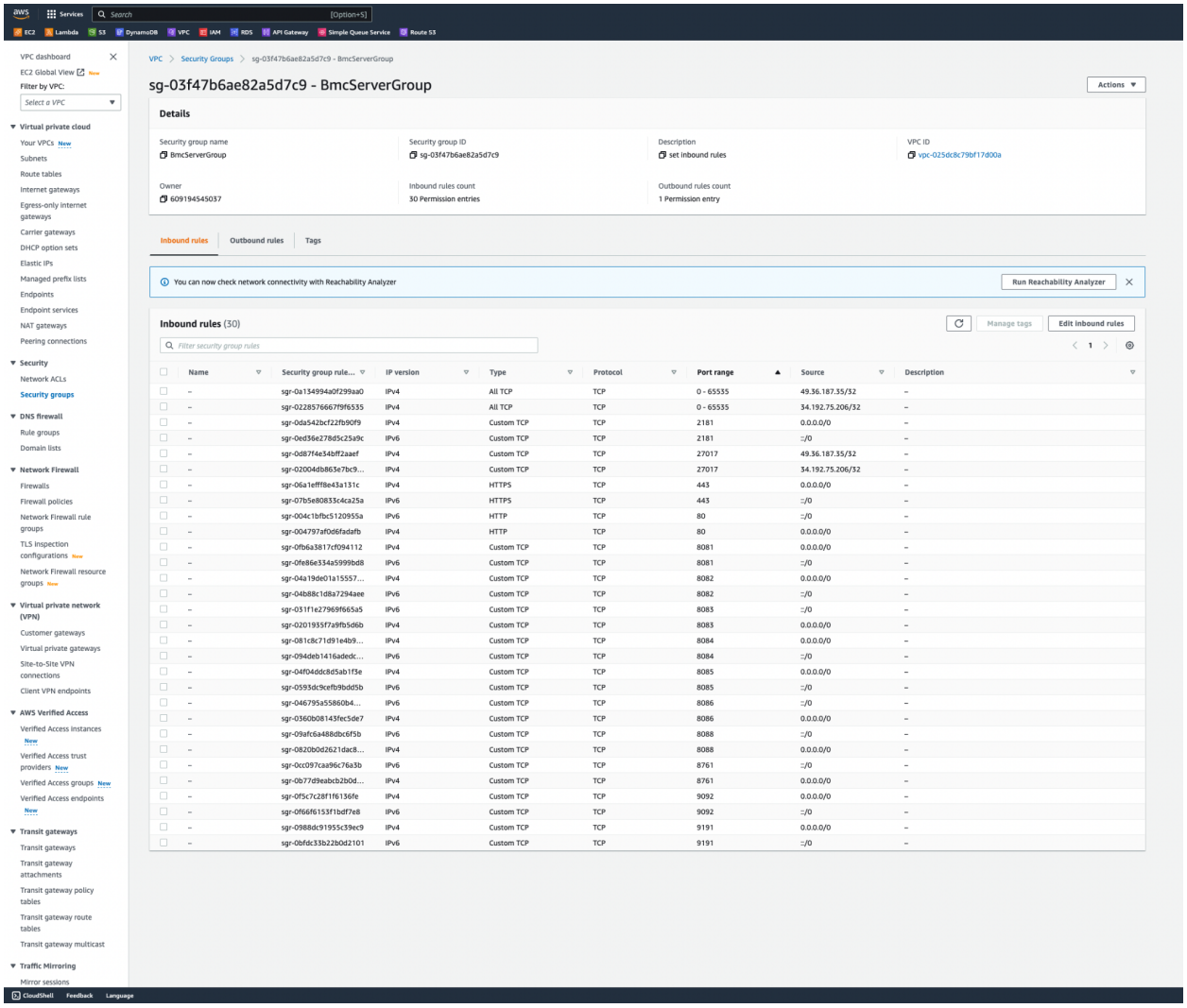


- Now, we created a VPC and Security group respectively referring from this resource. Please find the screenshots attached for the Inbound and Outbound rules that we set in

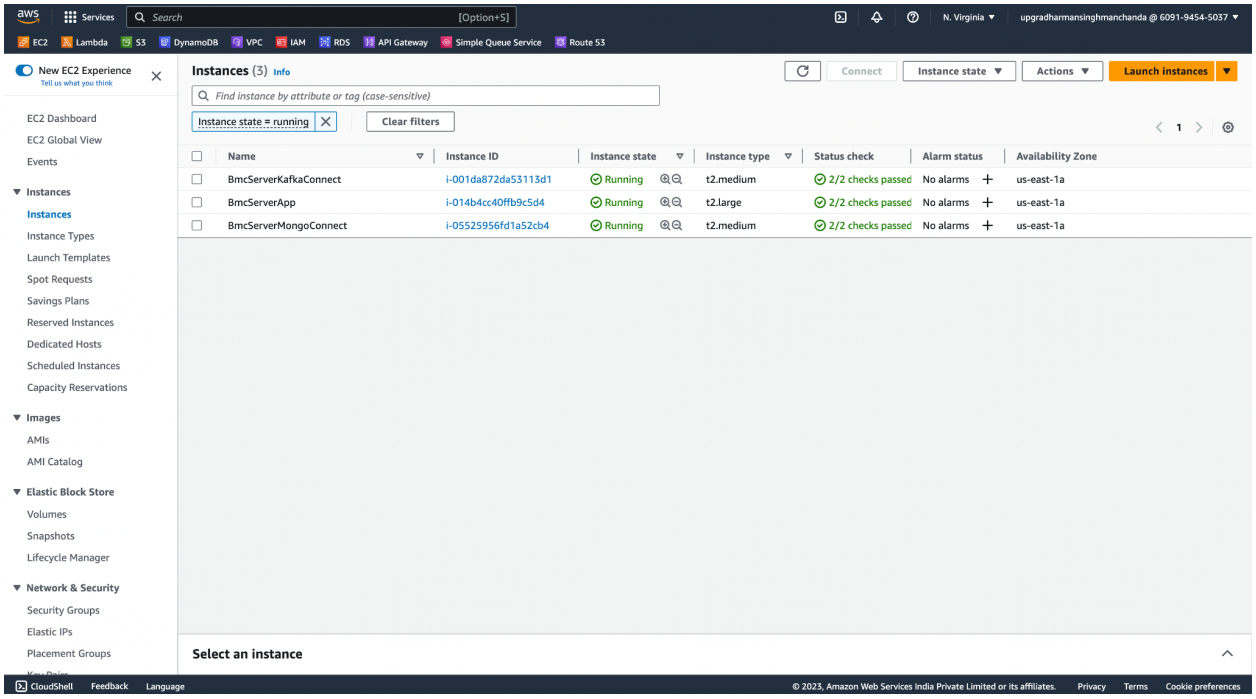
our security groups of our VPC. This also includes My IP Address from our local home desktop/server to connect to both of these instances.



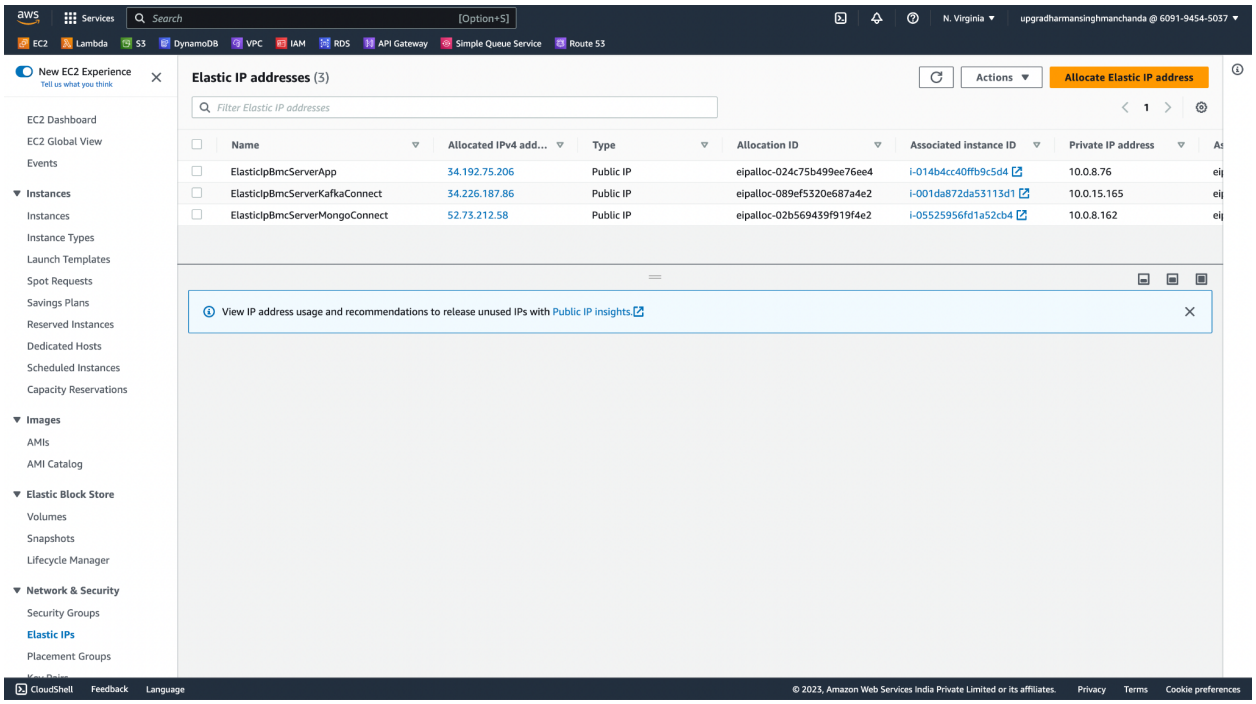




- Then we launched 3 EC2 instances in our VPC referring from this resource. Please find the configuration for the same.
    - First EC2 instance is for running the **Main server** and we choose Ubuntu operating system for the same server. We created a t2.large server for the same with 30 GB Magnetic standard storage.
    - Second EC2 instance is for running the **Kafka server** and we choose the Amazon AMI operating system for the same server. We created a t2.medium server for the same with 8 GB Magnetic standard storage.
    - Third EC2 instance is for running the **Mongo server** and we choose the Amazon AMI operating system for the same server. We created a t2.medium server for the same with 8 GB Magnetic standard storage.



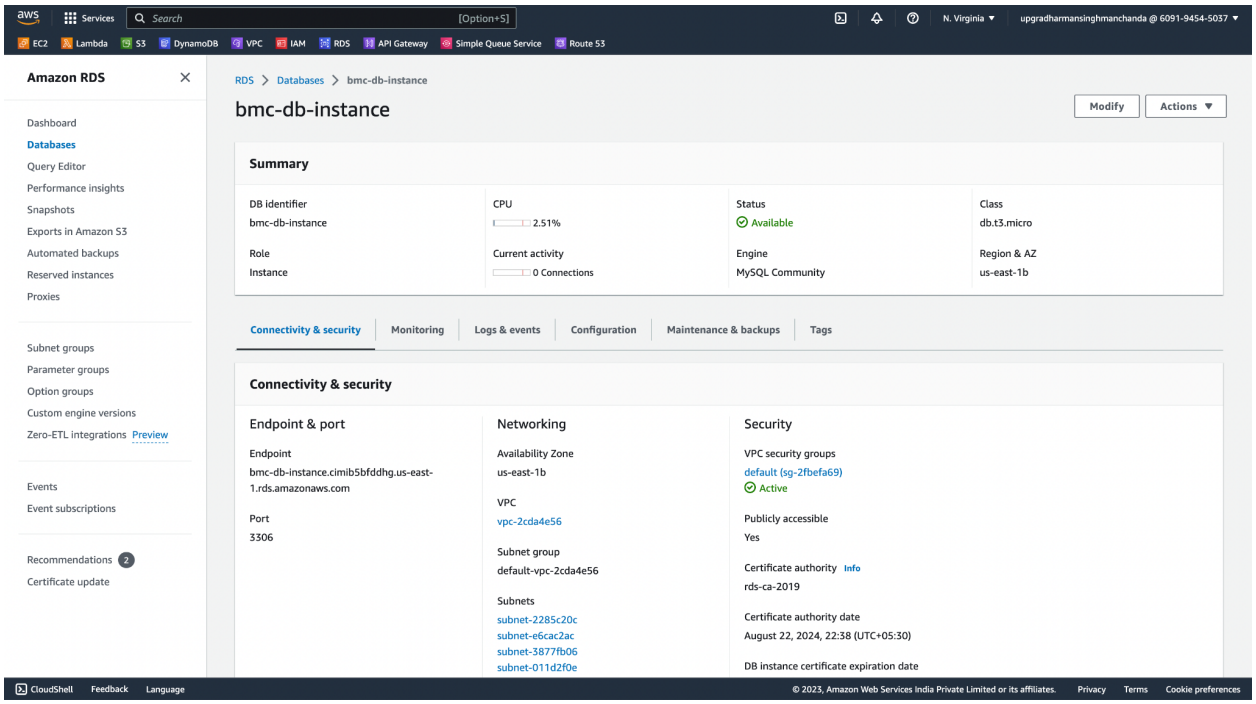
- Now we allocate the Elastic IP Addresses to the EC2 server per server so that we can use the same as pointers for our code.



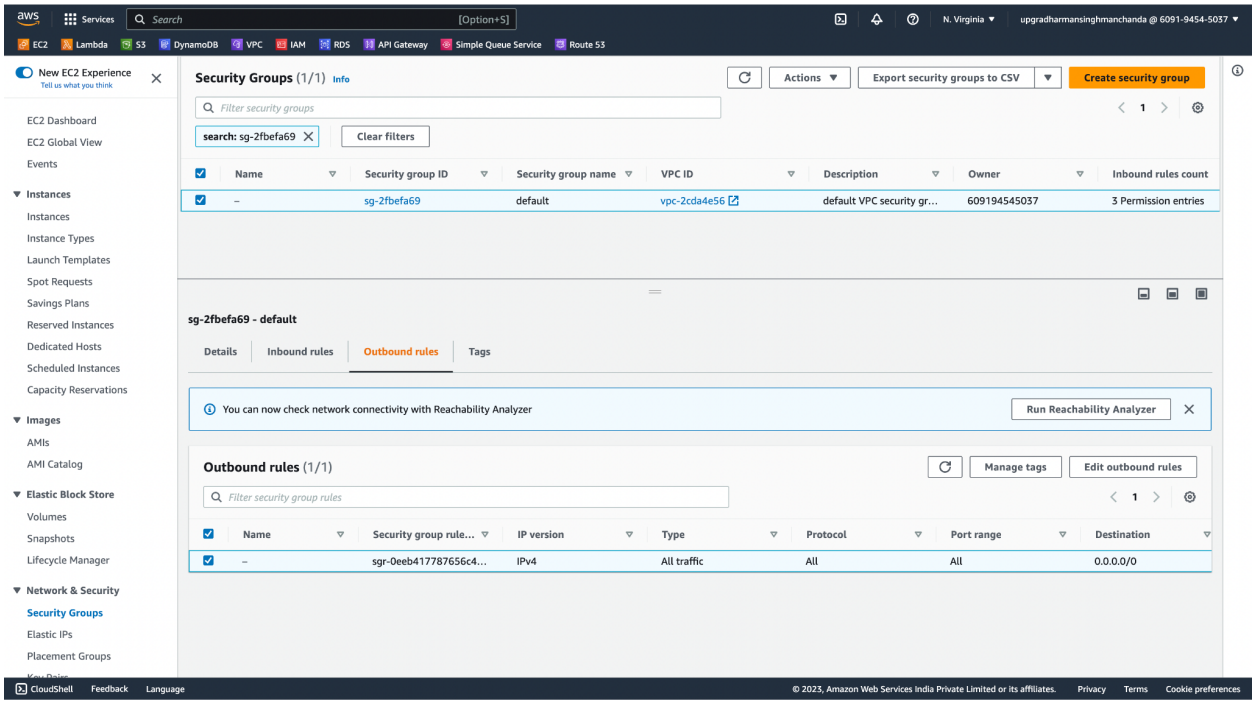
- We store the AWS key-pairs for the instances in one level up directory of the `BookMyConsultation` folder. In our case, it's `book-my-consultation-monorepo`. We store our key pairs in `book-my-consultation-monorepo/aws-keys` (that we won't be

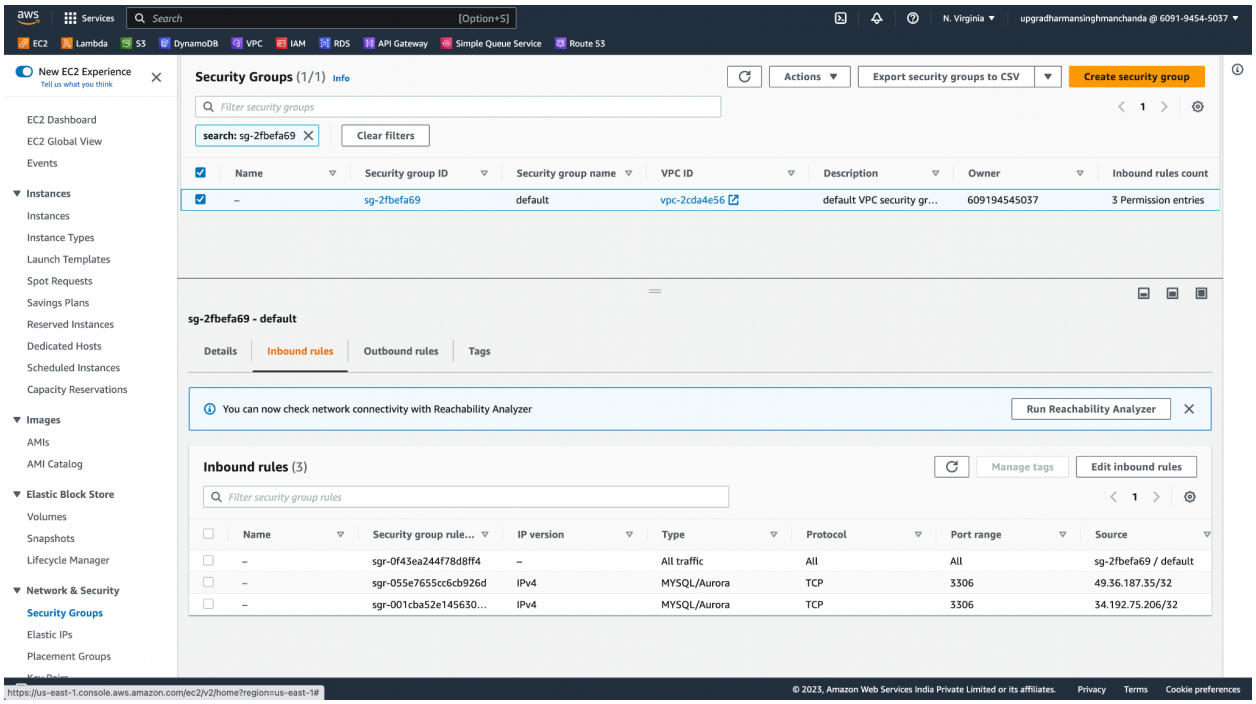
sharing as not needed and concerns privacy), whereas the project lives at `book-my-consultation-monorepo/BookMyConsultation` folder.

- Now, we create a single RDS instance referring from this resource. Please find the screenshot for the running Database server.

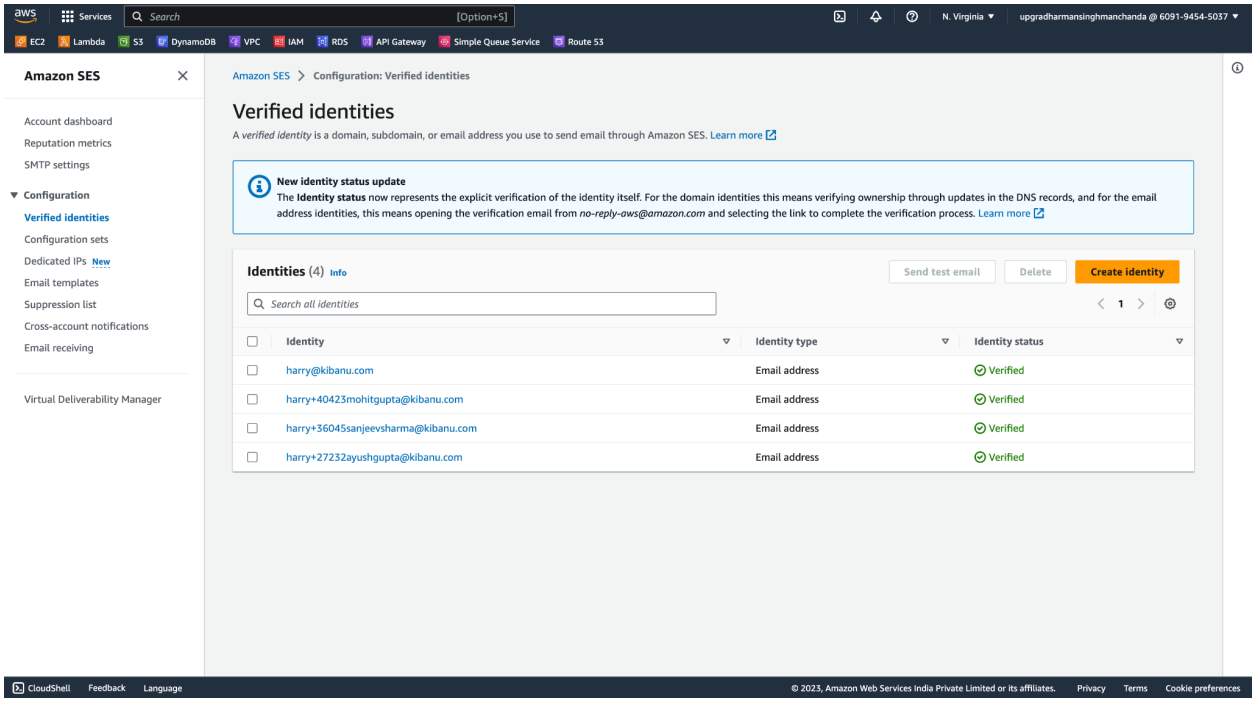


- The RDS instance were set with following specifications
    - User name: admin
    - Password: bmcpassword
- Now for the RDS Instance, we will use the default VPC and Security group that were given by AWS for the region. For this specific security group, please find the screenshots attached for the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

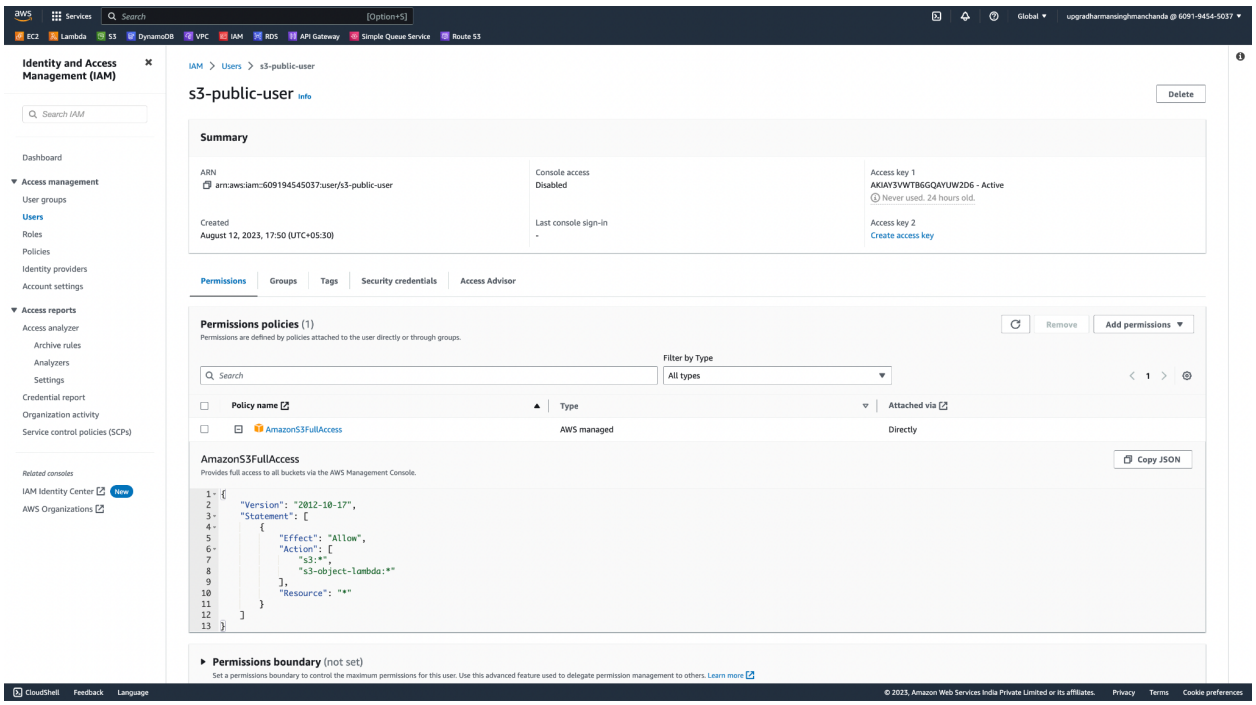




- Now we will login to Amazon Simple Email Service (SES) and generate SMTP credentials (IAM User) for email delivery referring from this resource. We will also create Verified Identities (as seen in the below screenshot) for the same process.



- Now it's time to create an IAM user which will help our app to upload files to S3 bucket.



- Now it is time to log in and gain access to our created instances thus we now log in to our instances, referring from this resource.
- First we connect to the **Mongo Server Instance**, we install MongoDB referring from this resource.

- We now run `systemctl status mongod` to ensure that the **Mongo Server** is running correctly.

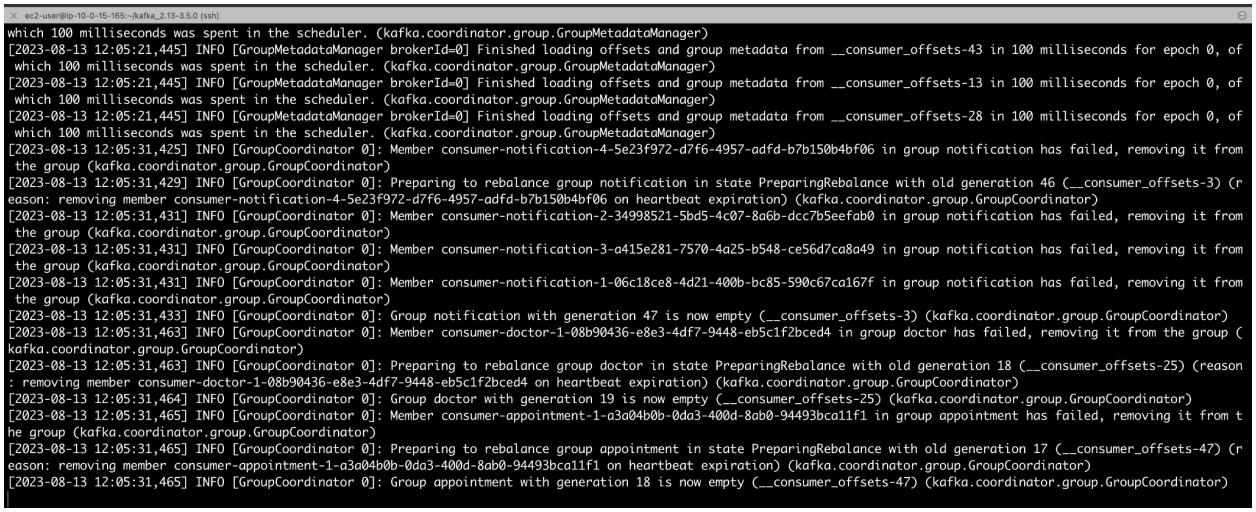
```
ec2-user@ip-10-0-8-162:~ (ssh)
https://aws.amazon.com/amazon-linux-2/
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[ec2-user@ip-10-0-8-162 ~]$
[ec2-user@ip-10-0-8-162 ~]$ systemctl status mongod
● mongod.service - MongoDB Database Server
   Loaded: loaded (/usr/lib/systemd/system/mongod.service; enabled; vendor preset: disabled)
   Active: active (running) since Sun 2023-08-13 09:53:49 UTC; 2h 5min ago
     Docs: https://docs.mongodb.org/manual
 Main PID: 3175 (mongod)
   CGroup: /system.slice/mongod.service
           └─3175 /usr/bin/mongod -f /etc/mongod.conf

Aug 13 09:53:49 ip-10-0-8-162.ec2.internal systemd[1]: Started MongoDB Database Server.
Aug 13 09:53:57 ip-10-0-8-162.ec2.internal mongod[3175]: {"t":{"$date":"2023-08-13T09:53:57.678Z"},"s":"I",  "c":"CONTROL",  "id":7484500, "ctx":"-","msg":"Env...o false"}
Hint: Some lines were ellipsized, use -l to show in full.
[ec2-user@ip-10-0-8-162 ~]$
```

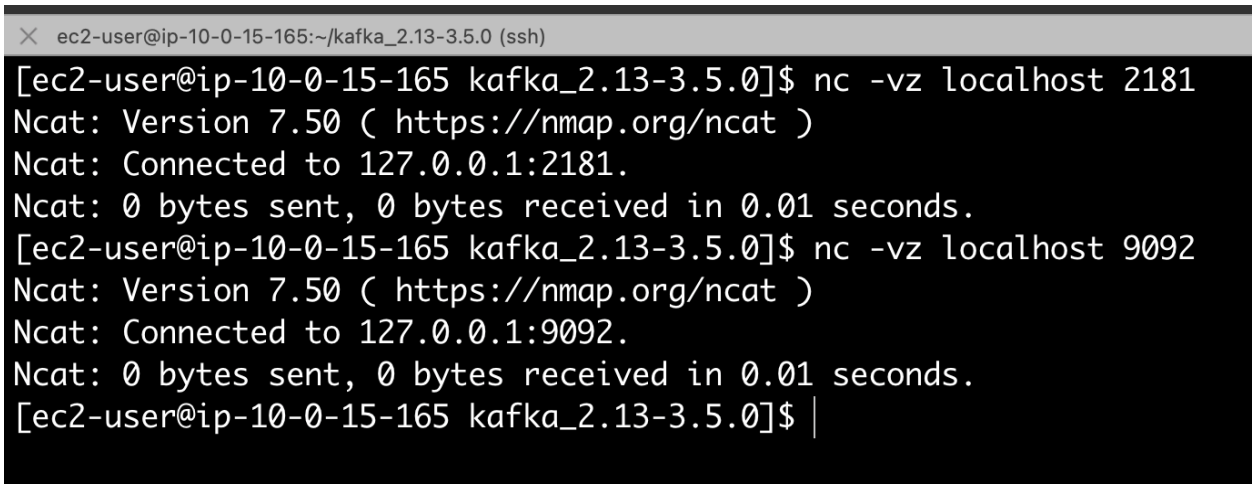
- Now we connect to the **Kafka Server Instance** by opening a new terminal to setup the Notification Process. In the **Kafka server** EC2 instance, we install Kafka referring from this resource. We will `cd` into the kafka directory.
- Now we will run the command: `bin/zookeeper-server-start.sh config/zookeeper.properties` to start the **Zookeeper**

```
ec2-user@ip-10-0-15-165:~/kafka_2.13-3.5.0 (ssh)
[2023-08-13 12:03:08,069] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,074] WARN config/zookeeper.properties is relative. Prepend ./ to indicate that you're sure! (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,081] INFO clientPortAddress is 0.0.0.0:2181 (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,081] INFO secureClientPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,081] INFO observerMasterPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,081] INFO metricsProvider.className is org.apache.zookeeper.metrics.impl.DefaultMetricsProvider (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,083] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.DatadirCleanupManager)
[2023-08-13 12:03:08,084] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.DatadirCleanupManager)
[2023-08-13 12:03:08,084] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCleanupManager)
[2023-08-13 12:03:08,084] WARN Either no config or no quorum defined in config, running in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2023-08-13 12:03:08,085] INFO Log4j 1.2 jmx support not found; jmx disabled. (org.apache.zookeeper.jmx.ManagedUtil)
[2023-08-13 12:03:08,086] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,086] WARN config/zookeeper.properties is relative. Prepend ./ to indicate that you're sure! (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,087] INFO clientPortAddress is 0.0.0.0:2181 (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,087] INFO secureClientPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,087] INFO observerMasterPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,087] INFO metricsProvider.className is org.apache.zookeeper.metrics.impl.DefaultMetricsProvider (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2023-08-13 12:03:08,087] INFO Starting server (org.apache.zookeeper.server.ZooKeeperServerMain)
[2023-08-13 12:03:08,102] INFO ServerMetrics initialized with provider org.apache.zookeeper.metrics.impl.DefaultMetricsProvider@258e2e41 (org.apache.zookeeper.server.ServerMetrics)
[2023-08-13 12:03:08,106] INFO zookeeper.snapshot.trust.empty : false (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
[2023-08-13 12:03:08,118] INFO  (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO   ______                  _                                          (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO  |___  /                 | |                                         (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO     / /    ___     ___   | | __   ___    ___   _ __     ___   _ __   (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO    / /    / _ \   / _ \  | |/ /  / _ \  / _ \ | '_ \   / _ \ | '__|(org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO   / /__  | (_) | | (_) | |   <  |  __/ |  __/ | |_) | |  __/ | |    (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO  /_____|  \___/   \___/  |_|\_\  \___|  \___| | .__/   \___| |_|(org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO                                               | |                     (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,118] INFO                                               |_|                     (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,119] INFO  (org.apache.zookeeper.server.ZooKeeperServer)
[2023-08-13 12:03:08,122] INFO Server environment:zookeeper.version=3.6.4--d65253dcf68e9097c6e95a126463fd5fdeb4521c, built on 12/18/2022 18:10 GMT (org.apache.zookeeper.server.ZooKeeperServer)
```

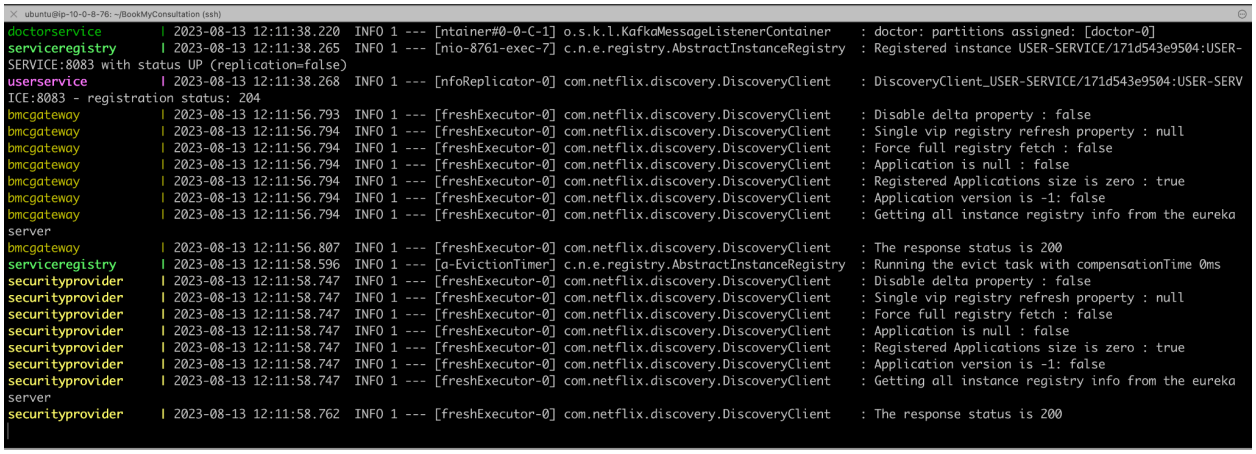
- Now will open a new terminal and `cd` into the kafka directory.
- Now we will run the command: `bin/kafka-server-start.sh config/server.properties` to start the **Kafka Server**.



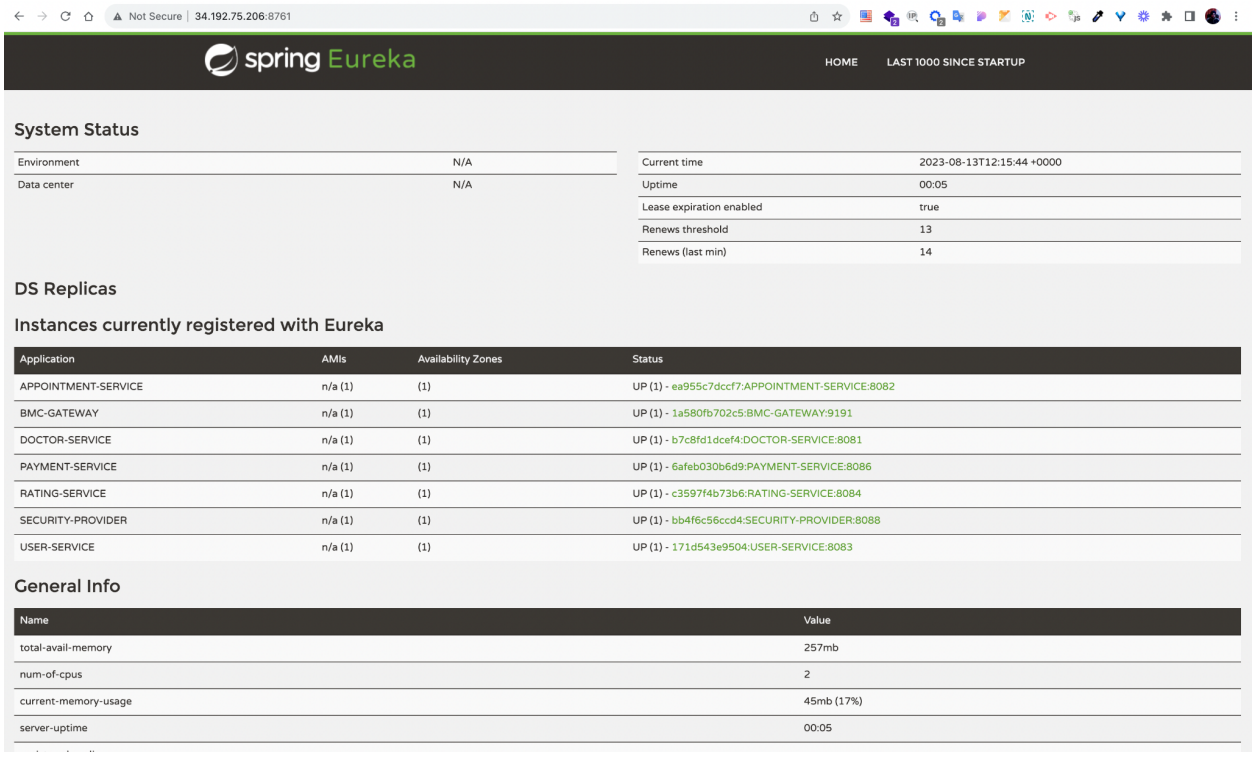
- Now will open a new terminal and `cd` into the kafka directory to ensure whether **Zookeeper** and **Kafka Server** has started or not by running `nc -vz localhost 2181` and `nc -vz localhost 9092` respectively.



- Now let's connect to the **Main Server** by opening a new terminal, we will now run `sudo docker compose up --build` to deploy our Multi container project.

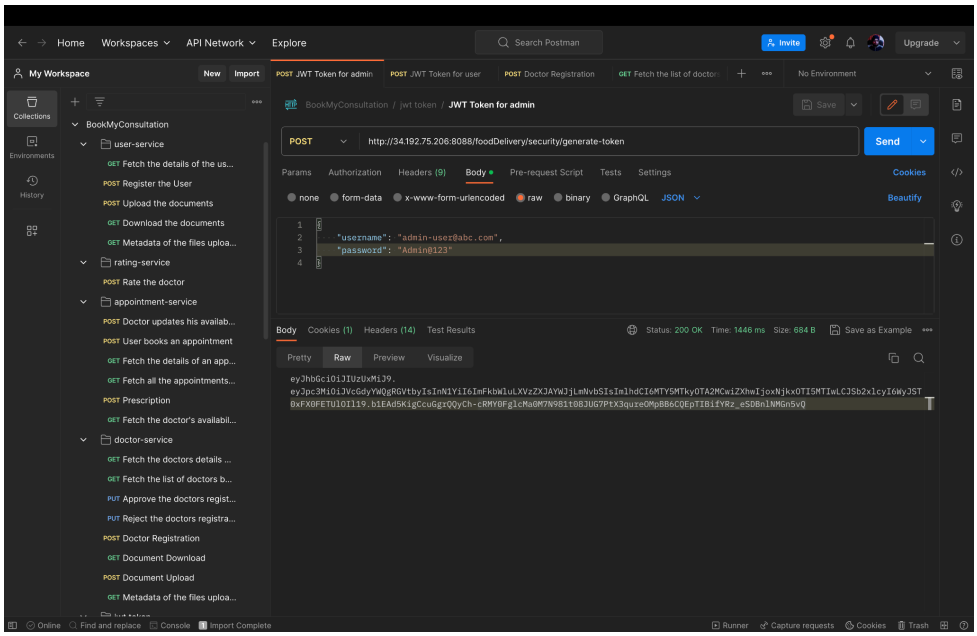


- We can confirm that all servers by opening a new terminal and running the command `sudo docker ps -a` respectively.
- Finally our project has been deployed, the Images for all the services are created, and all the containers are running, we can confirm the same by visiting PORT `8761` from our **Main Server** EC2 instance's Elastic IP (see attached screenshot). In our case it was `34.192.75.206:8761` respectively.

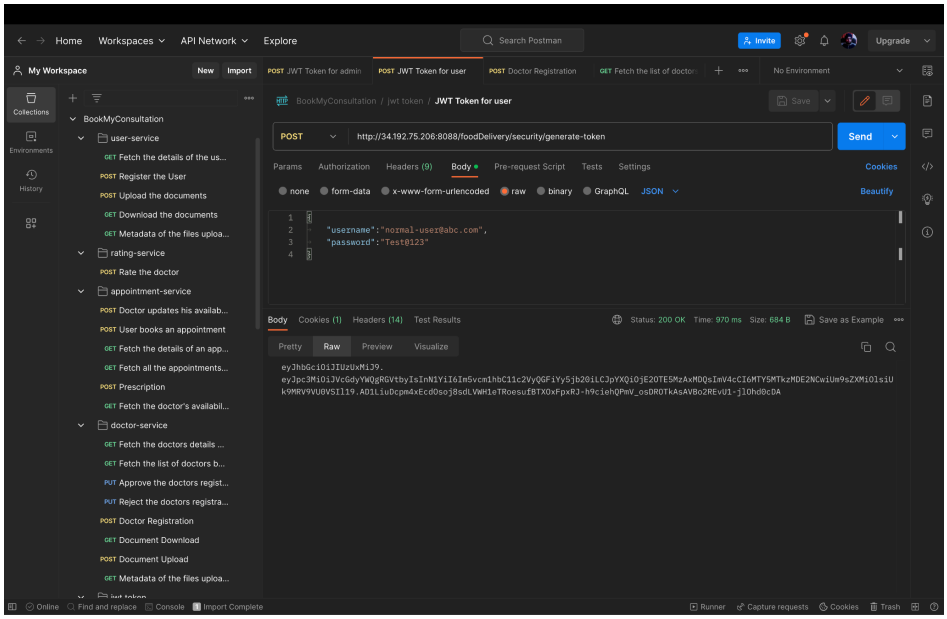


API Testing Screenshots

- Now it’s time to test whether APIs are being hit correctly using postman or not.
- First let us generate a JWT token for the ADMIN respectively.

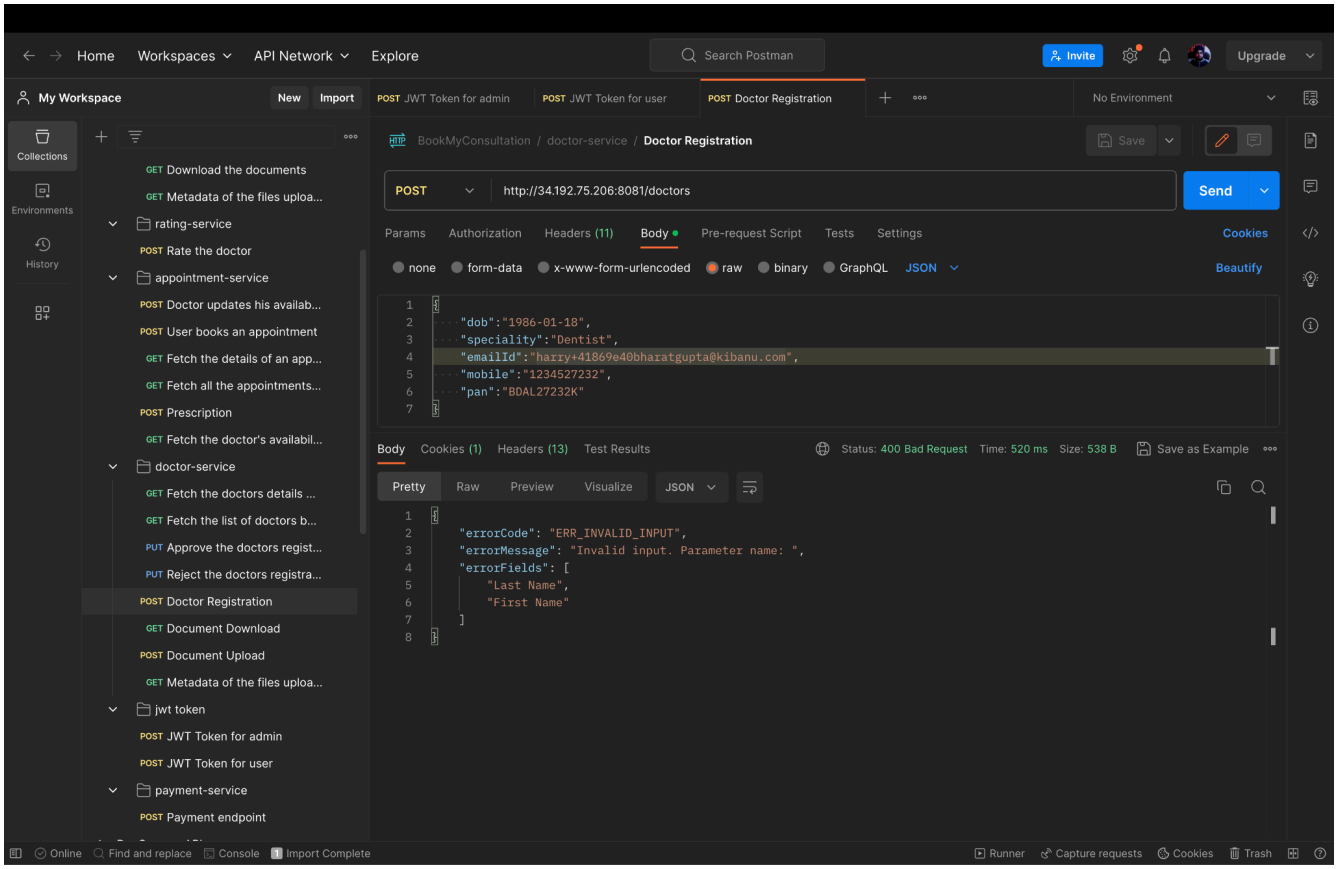


- Now let us generate a JWT token for the USER respectively.

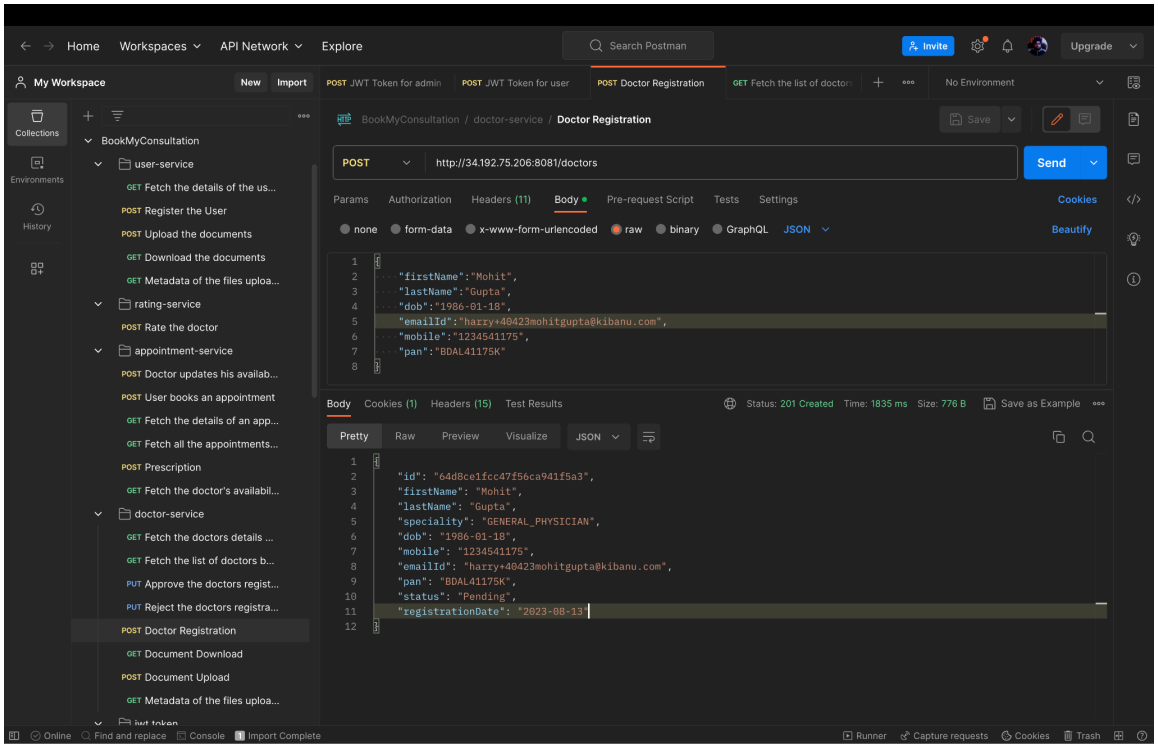


- NOW we will need to send a request to `POST /doctors` respectively. The Request body will contain firstName, lastName, dob, emailID, mobile, and PAN respectively. If the token has the role of either USER or ADMIN, then the endpoint can be accessed.

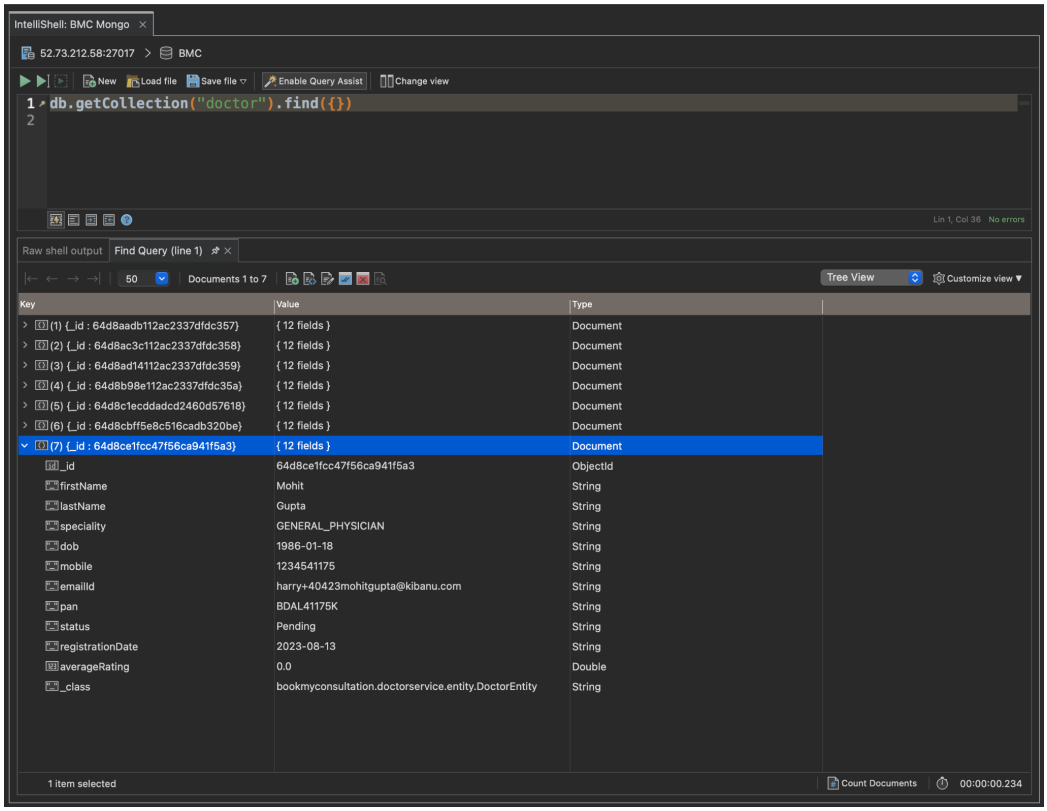
- But before sending the same, let us send an invalid request by deliberately not sending firstName and lastName respectively. We will see what's been seen in the below screenshot, an error will be thrown by the API.



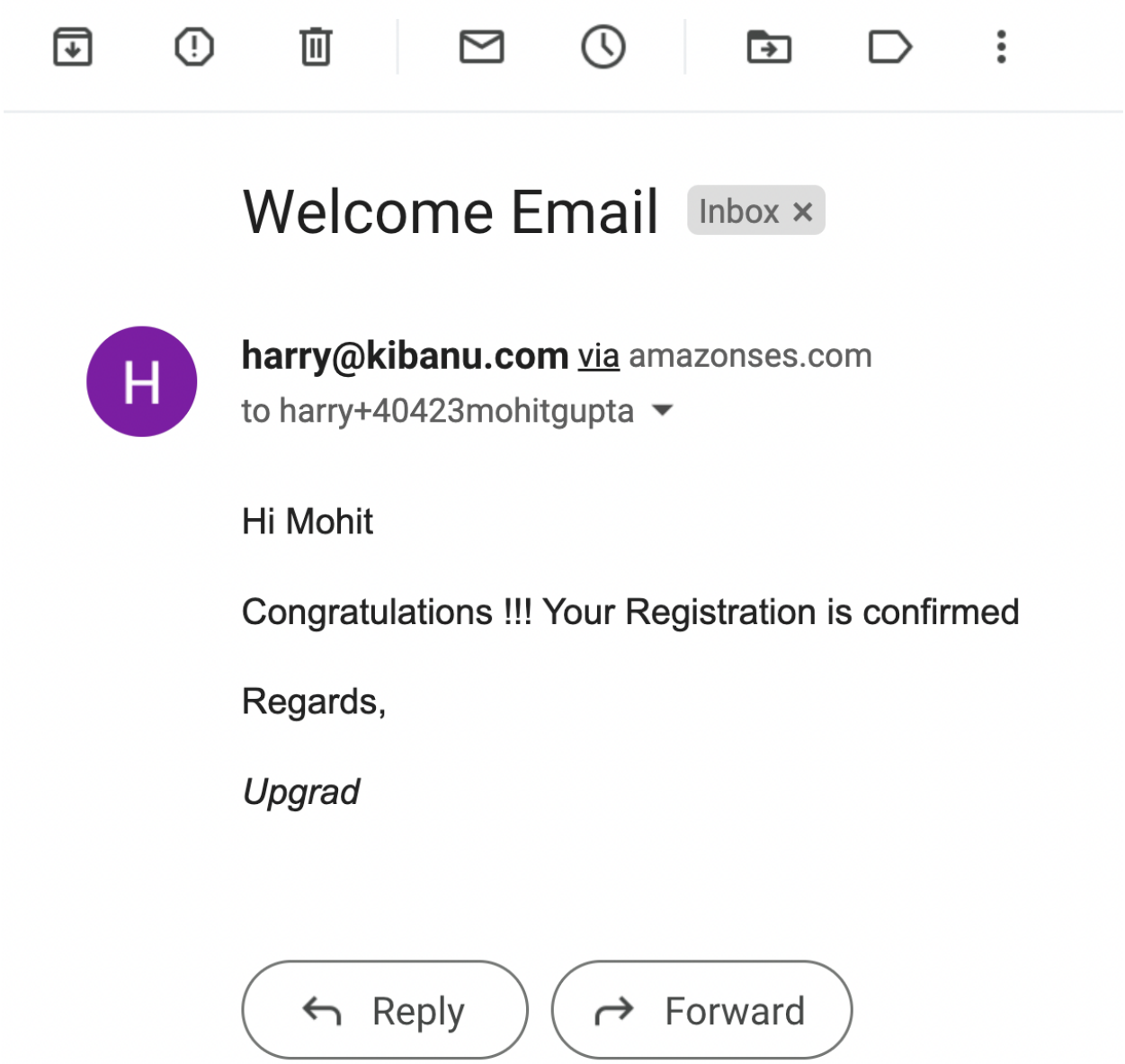
- Now it's time to send a Valid API request to `POST /doctors` respectively.



- We can check the Doctor's database to confirm that the doctor collection has been successfully added.

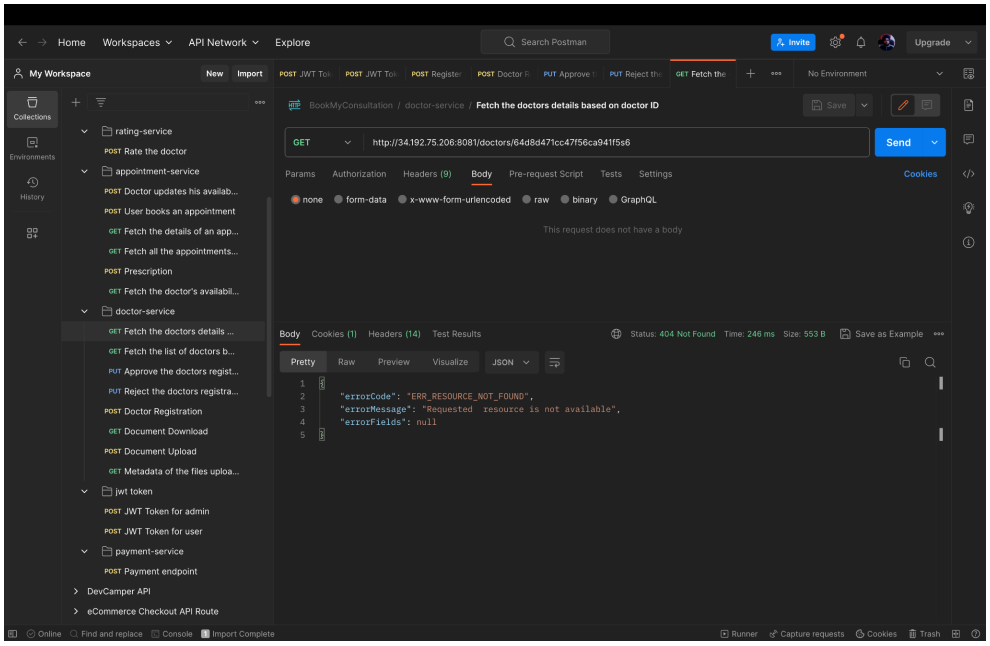


- Now let’s check our email to confirm that the email has been received.

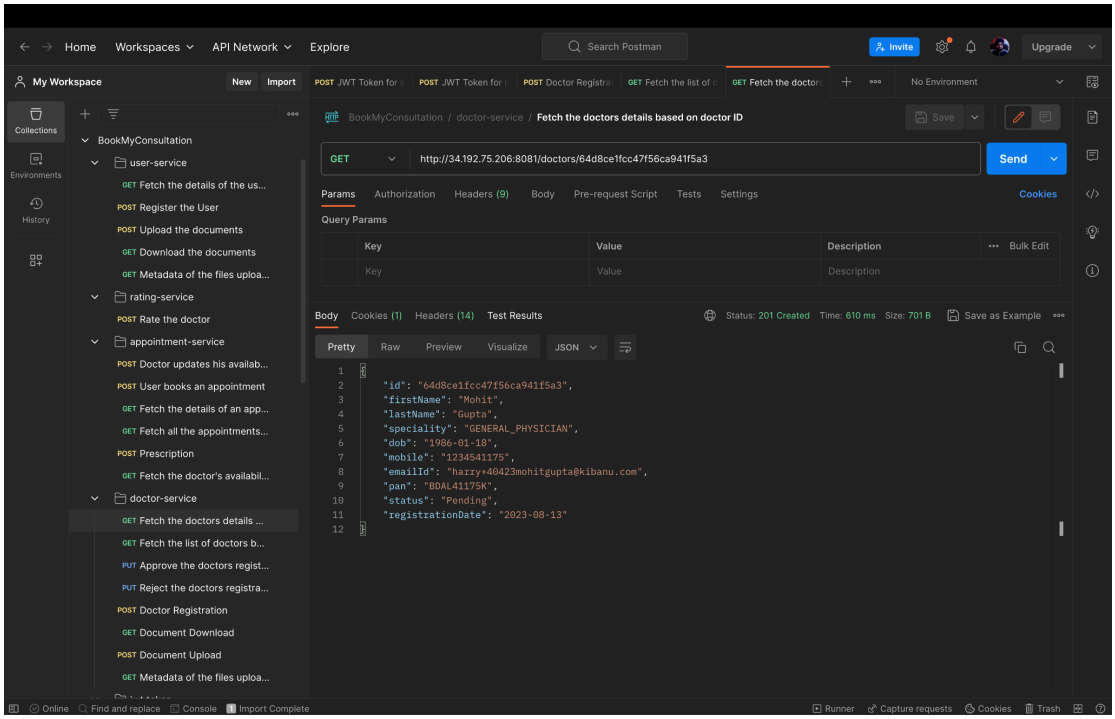


- NOW based on the id returned from the `POST /doctors` response, we will look to send a request to `GET /doctors/{doctorId}` respectively to make sure of the contents. If the token has the role of either USER or ADMIN, then the endpoint can be accessed.

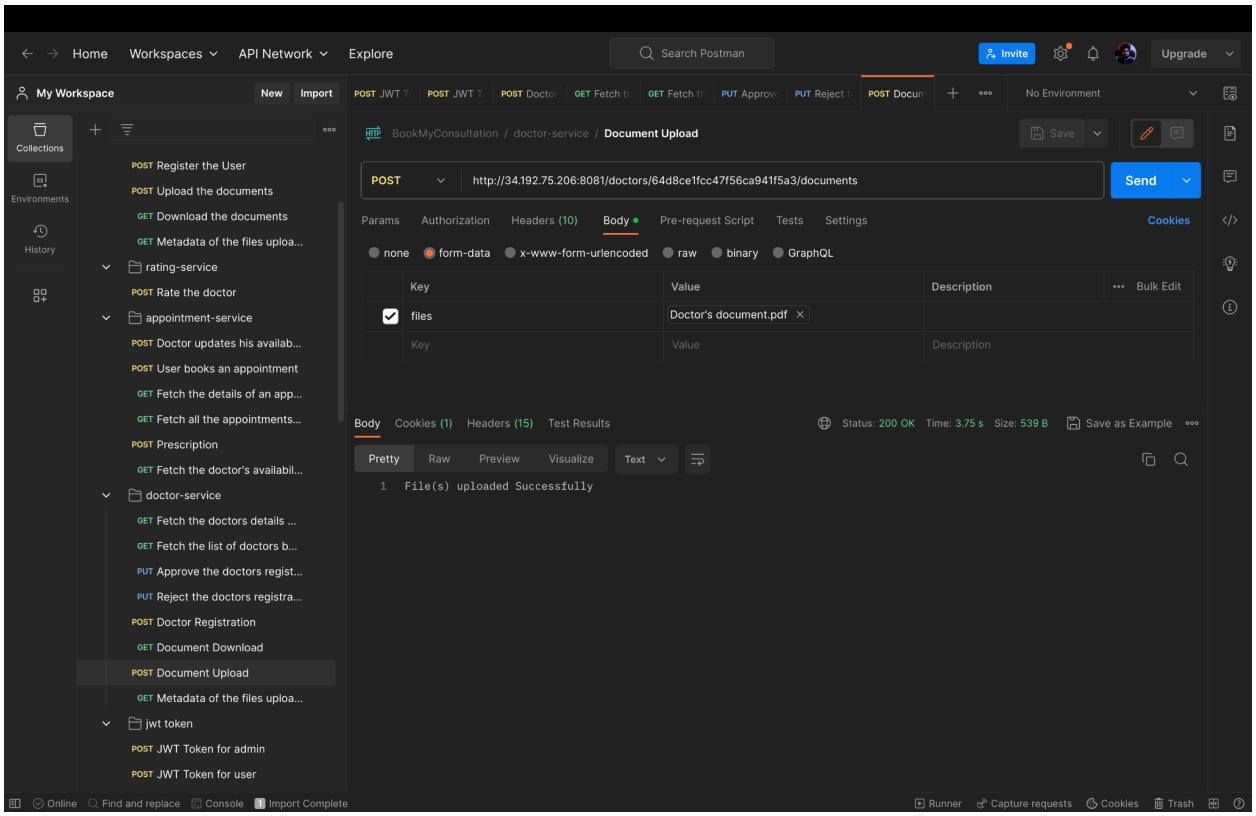
- But before sending the same, let us send an invalid request by deliberately not sending the correct `doctorId`. We will see what's been seen in the below screenshot, an error will be thrown by the API.



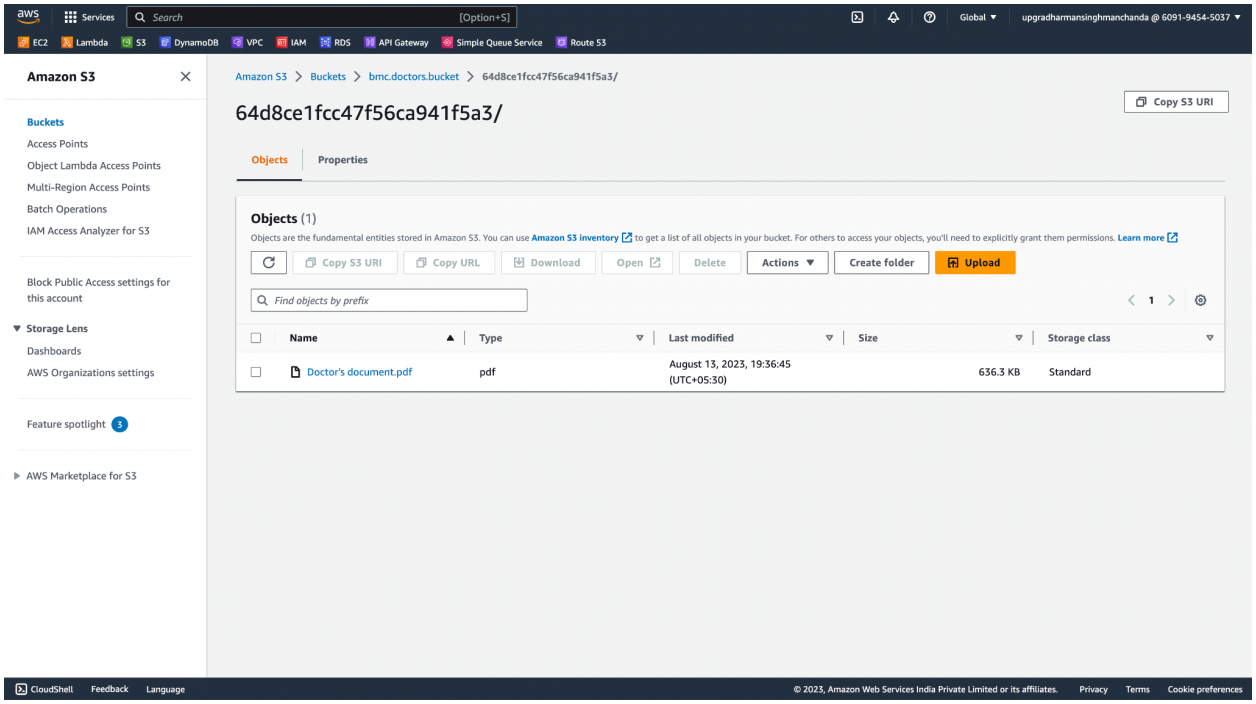
- Now it's time to send a Valid API request to `GET /doctors/{doctorId}` respectively.



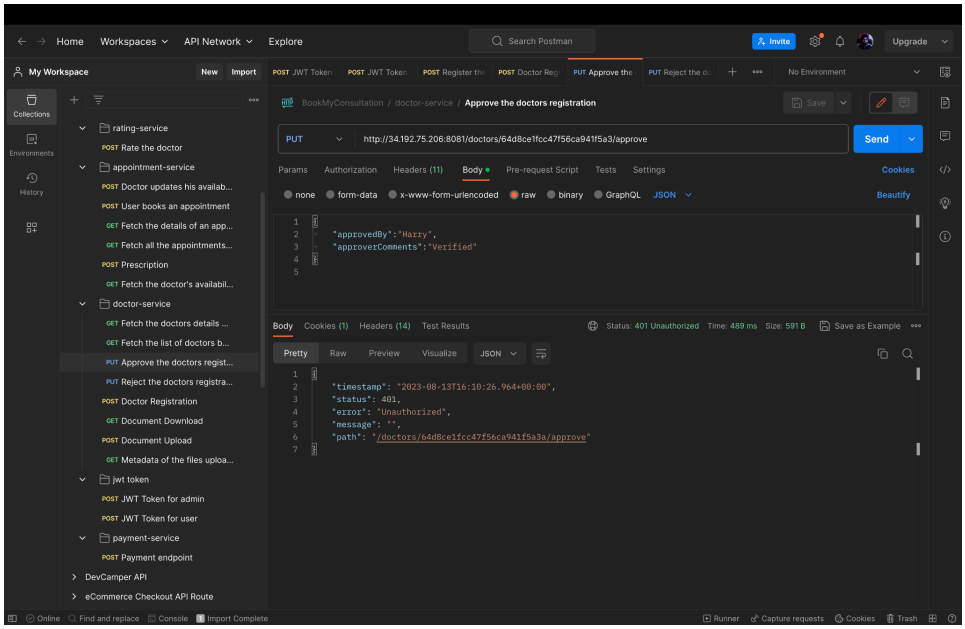
- Now the doctor will be uploading the documents. Behind the scenes, it will be uploaded to an S3 bucket.
- For the same let's send a `POST /doctors/{doctorId}/document` respectively. For the Request body, we will select formData this time, and upload the respective files that need to be uploaded. If the token has the role of either USER or ADMIN, then the endpoint can be accessed.



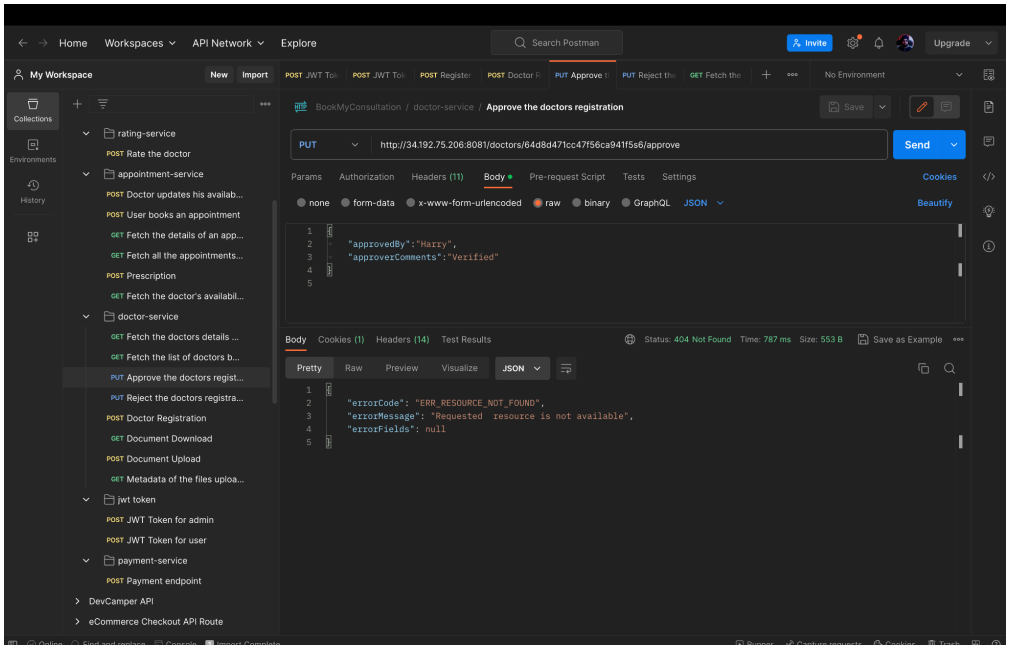
- We will see File(s) uploaded successfully as a response from the API. Now we can check the Amazon S3 console to confirm whether the document has been uploaded or not.



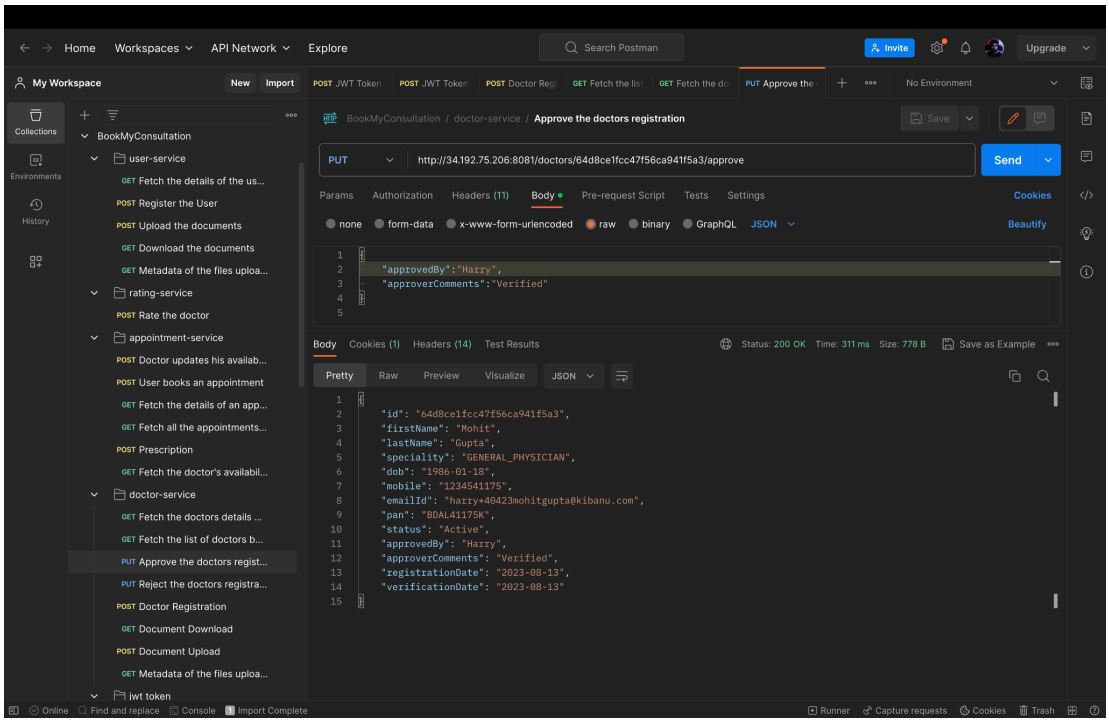
- Now let's test the endpoint that is responsible for approving the doctor's registration request. For the same, we will look to send a `PUT doctors/{doctorId}/approve` respectively. For the Request body, we will send approvedBy, approverComments respectively. If the token has the role of ADMIN, then only the endpoint can be accessed.
- But before sending the same, let us send an invalid request by deliberately not sending the token with the role of USER. We will see what's been seen in the below screenshot, an error will be thrown by the API.



- Still we will be testing invalid requests more by sending requests with wrong id for doctorId at `PUT doctors/{doctorId}/approve` respectively. We will see what's been seen in the below screenshot, an error will be thrown by the API.



- Now it's time to send a valid request and look into whether everything works well.



- Now we can check email to find out whether we received email as a doctor by the notification service for being approved as a doctor.

Verification Email Inbox ×

H

harry@kibanu.com via amazonses.com
to harry+40423mohitgupta

Hi Mohit

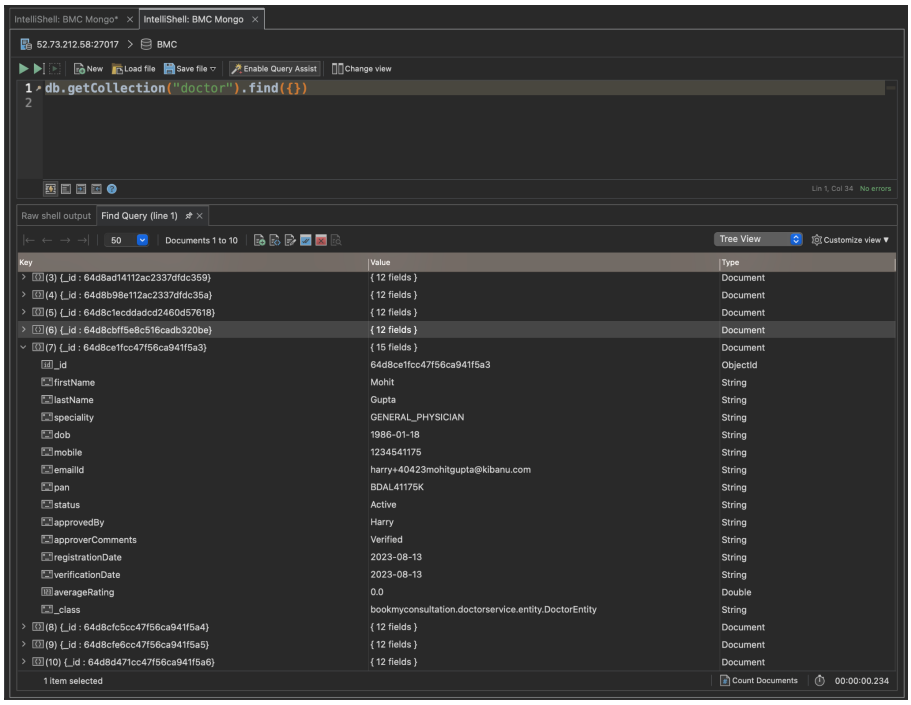
Your Registration is Active by Harry.

Approver Comment : Verified

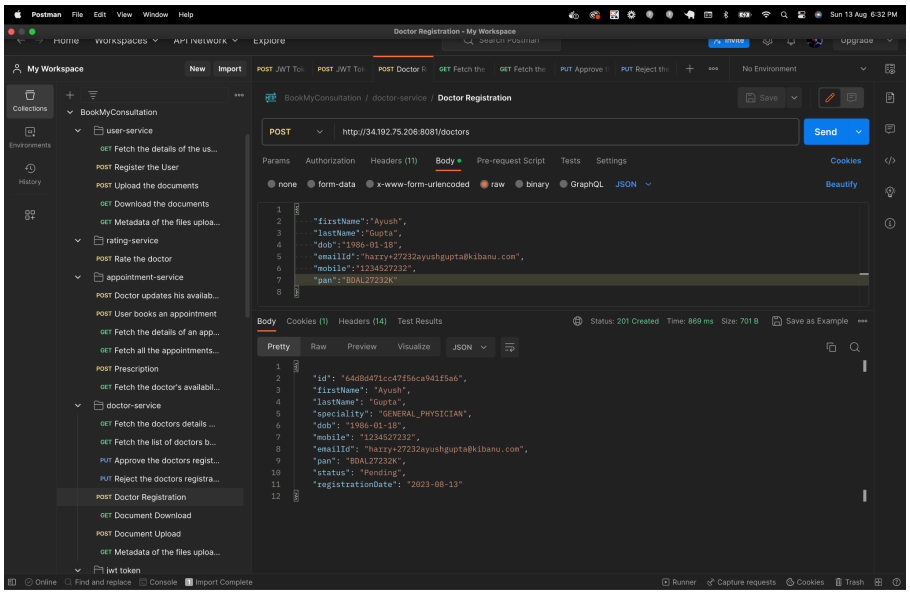
Regards,

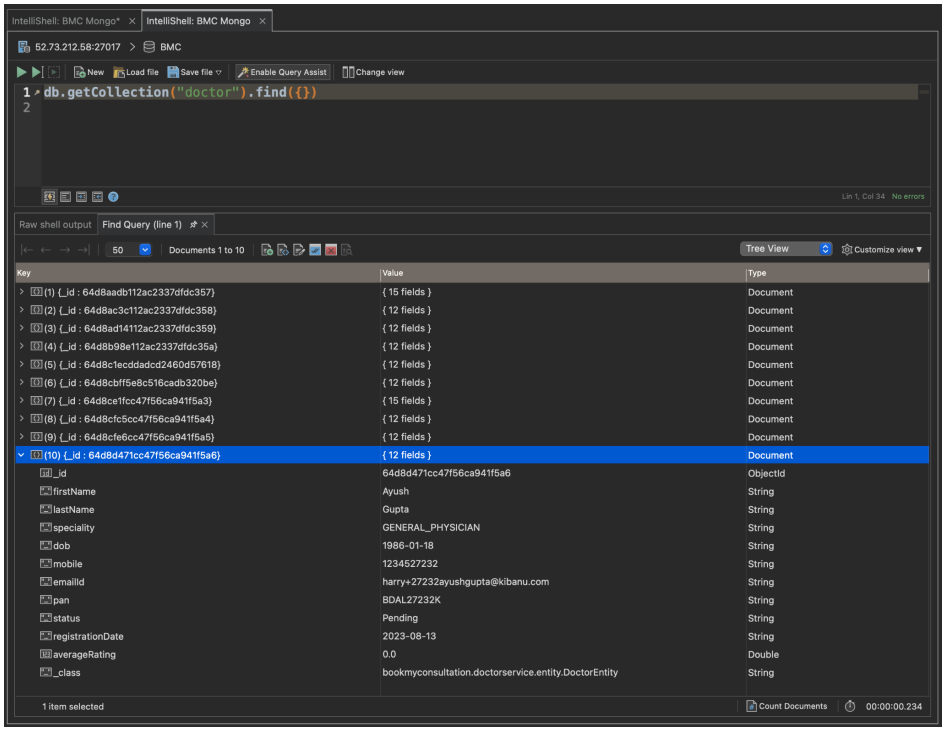
Upgrad

- Now let's test whether the doctor collection has been updated to active status or not.

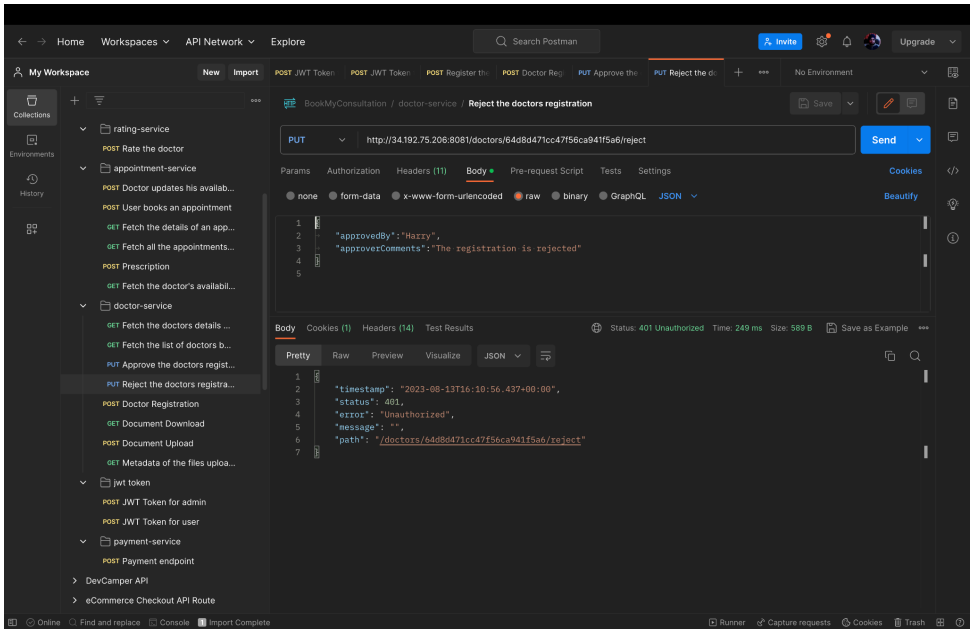


- Now let's test the endpoint that is responsible for rejecting the doctor's registration request. For the same, we will look to send a `PUT doctors/{doctorId}/reject` respectively. For the Request body, we will send approvedBy, approverComments respectively. If the token has the role of ADMIN, then only the endpoint can be accessed.
- For testing the same, let's create a new doctor registration so that we can reject the same as an ADMIN.

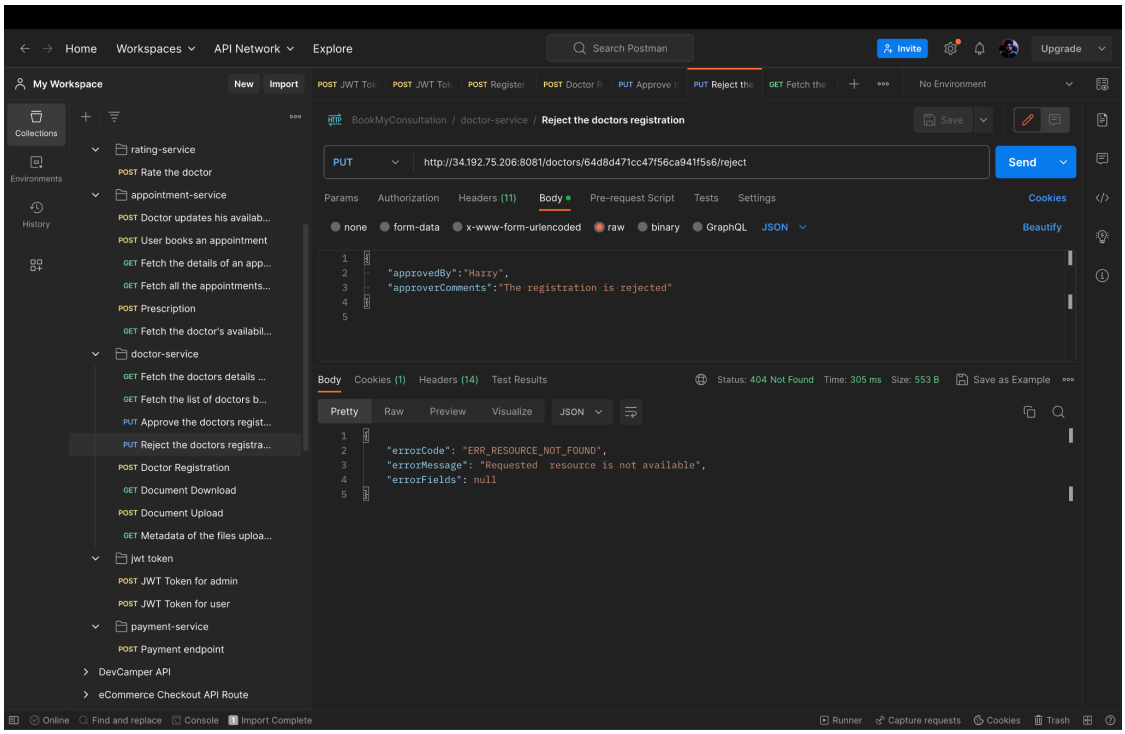




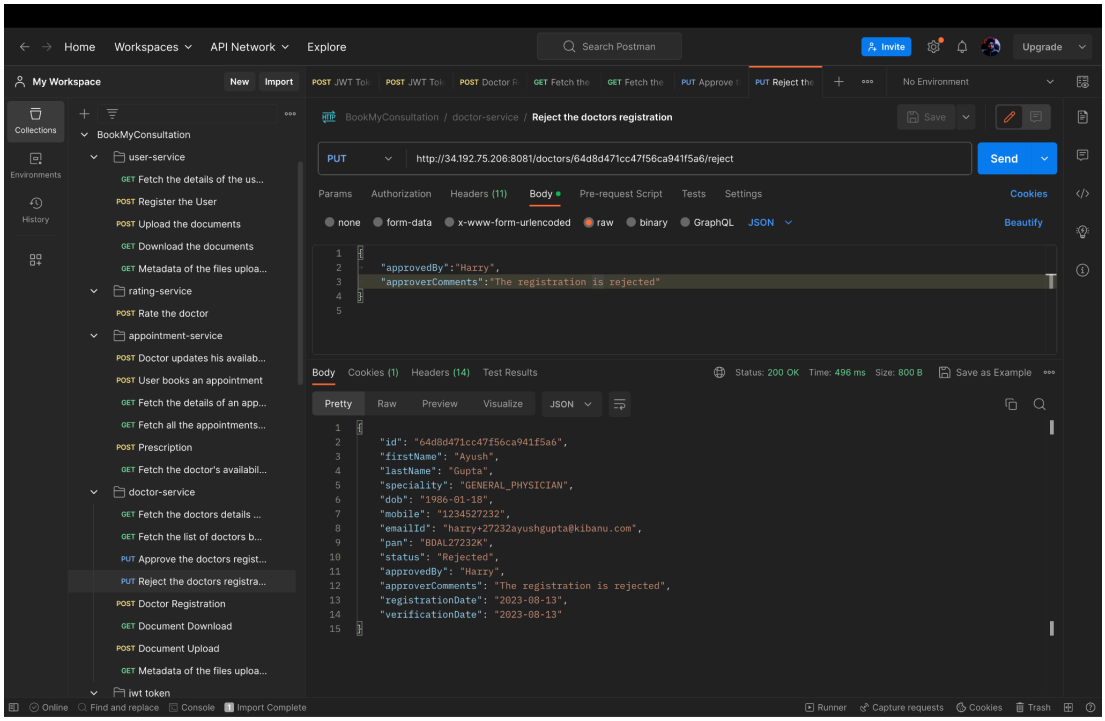
- Now it's time to test our reject endpoint, but before sending a valid request, let us send an invalid request by deliberately not sending the token with the role of USER. We will see what's been seen in the below screenshot, an error will be thrown by the API.



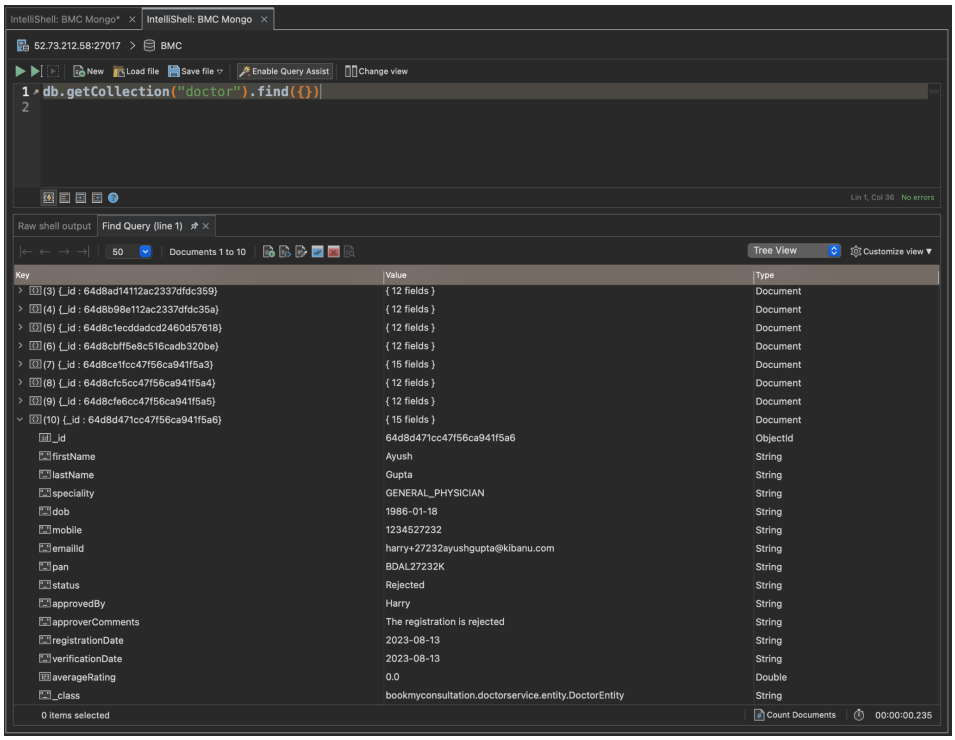
- Still we will be testing invalid requests more by sending requests with wrong id for doctorId at `PUT doctors/{doctorId}/reject` respectively. We will see what's been seen in the below screenshot, an error will be thrown by the API.



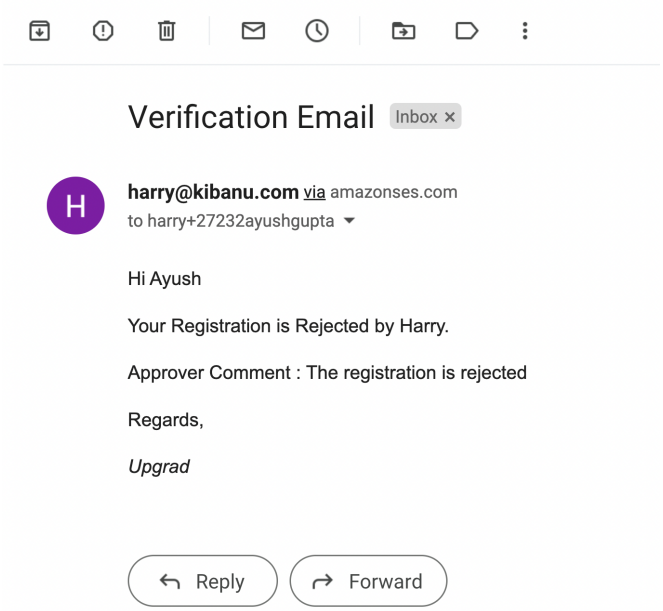
- NOW it’s time to send a valid request and confirm whether as an admin we can reject the doctor or not.



- Now let’s test whether the doctor collection has been updated to rejected status or not.



- Now we can check email to find out whether we received email as a doctor by the notification service for being rejected as a doctor.



Verification Email Inbox ×

harry@kibanu.com via amazonses.com
to harry+27232ayushgupta

Hi Ayush

Your Registration is Rejected by Harry.

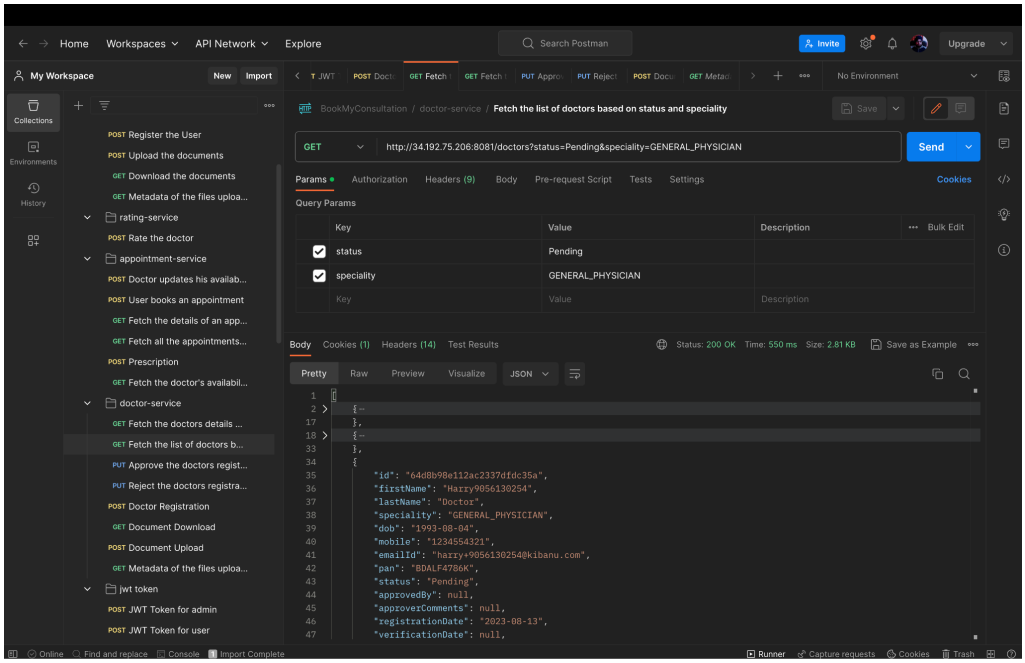
Approver Comment : The registration is rejected

Regards,

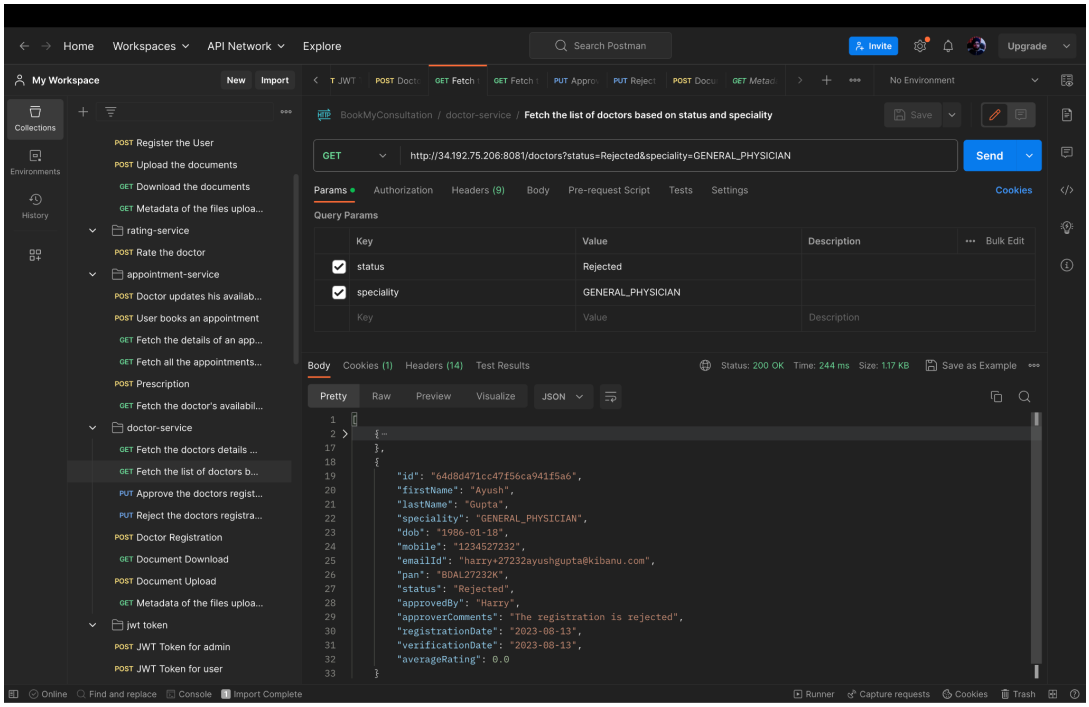
Upgrad

Reply Forward

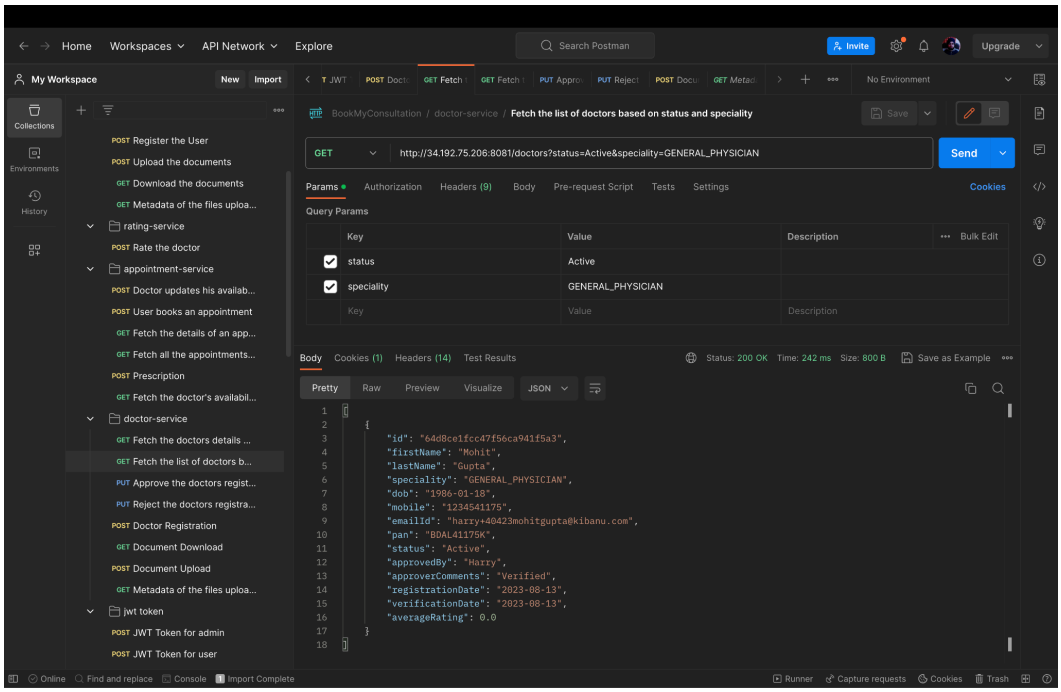
- Now let's test the `GET /doctors` respectively by various statuses to confirm whether the correct list is being returned. Role: USER, ADMIN
- First let's test with pending status by testing `GET /doctors?status=Pending` respectively.



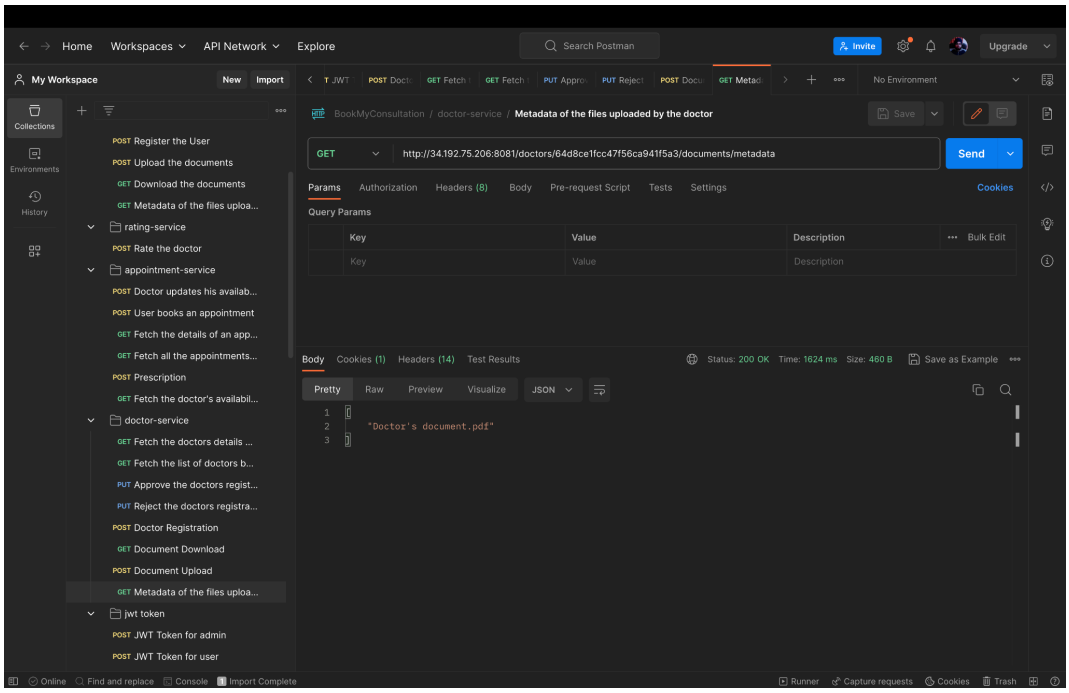
- Now let's test with rejected status by testing `GET /doctors?status=Rejected` respectively.



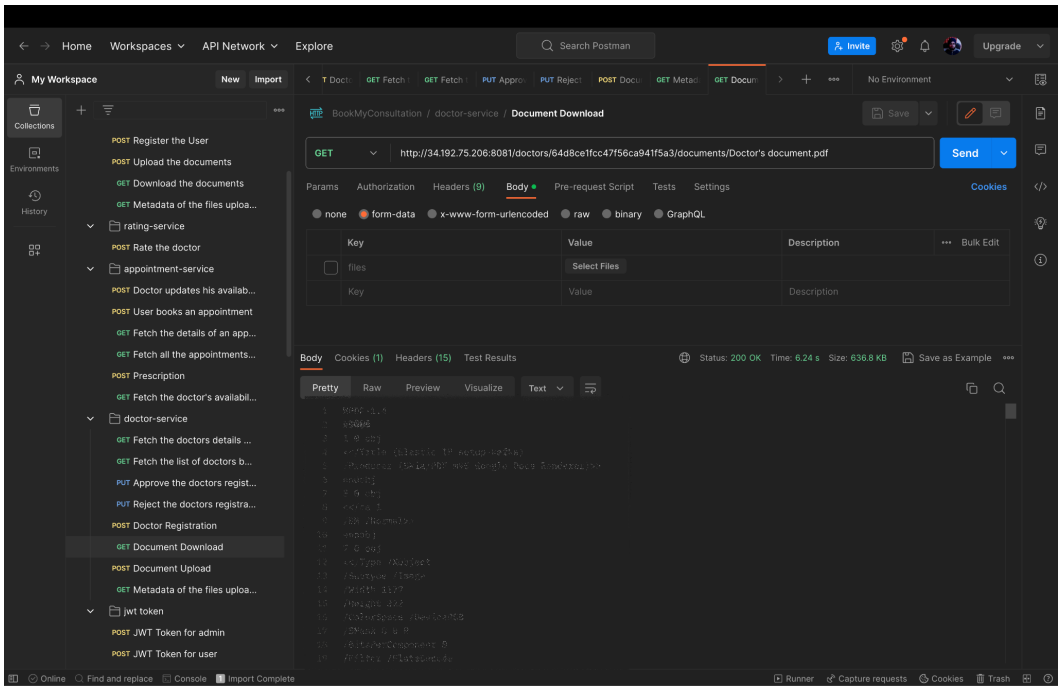
- Now let's test with active status by testing `GET /doctors?status=Active` respectively.



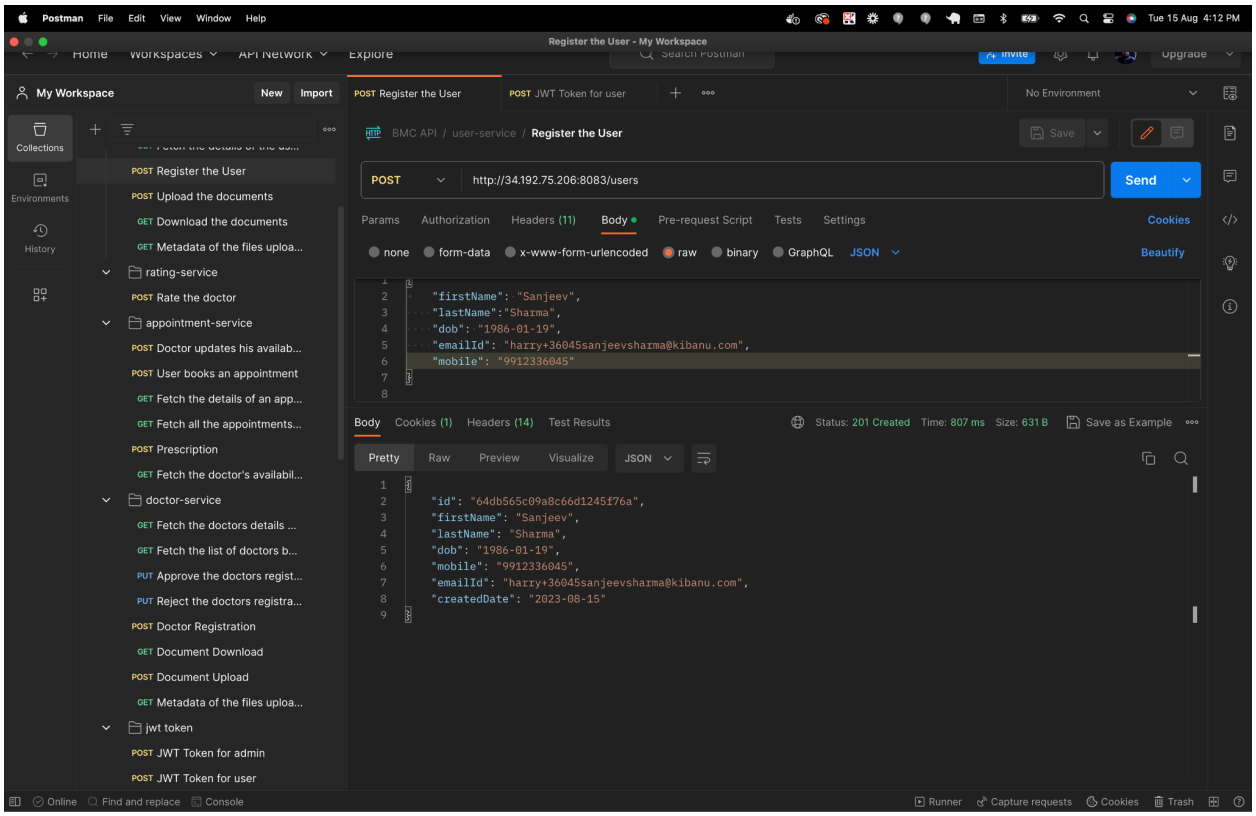
- Now let's test `GET /doctors/{doctorId}/documents/metadata` that is responsible for returning the list of the documents uploaded by the doctor. Role: ADMIN,USER

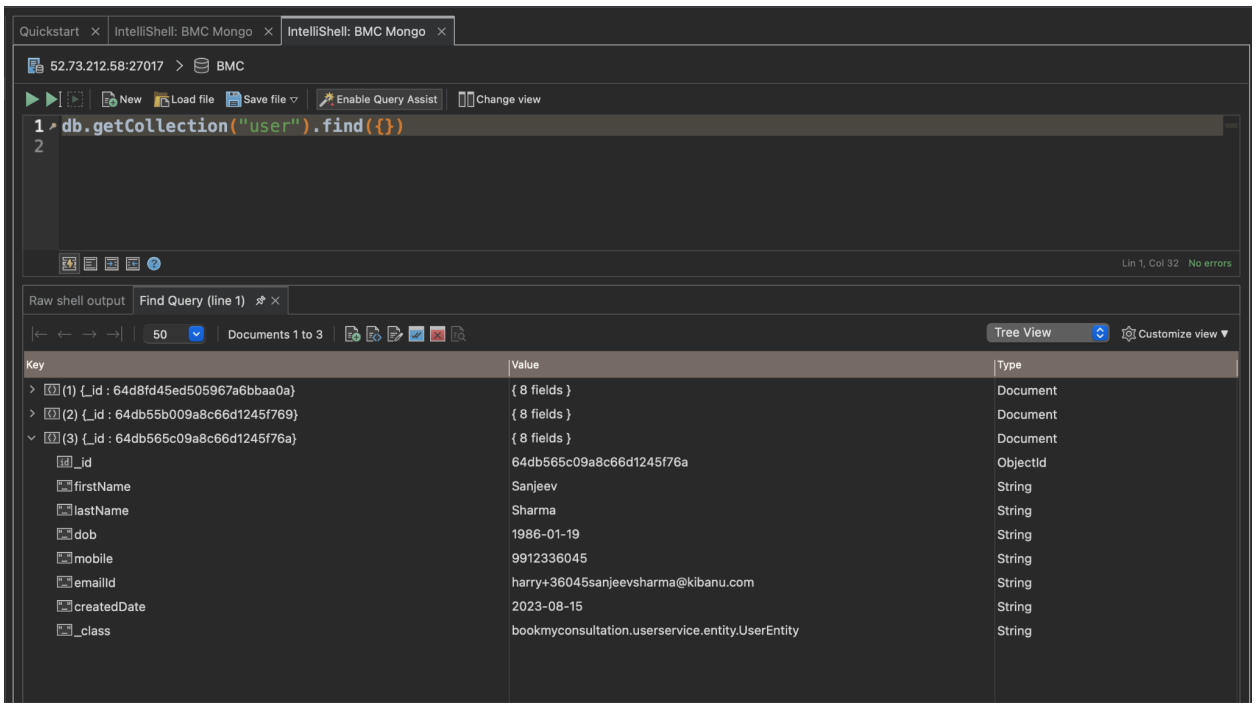


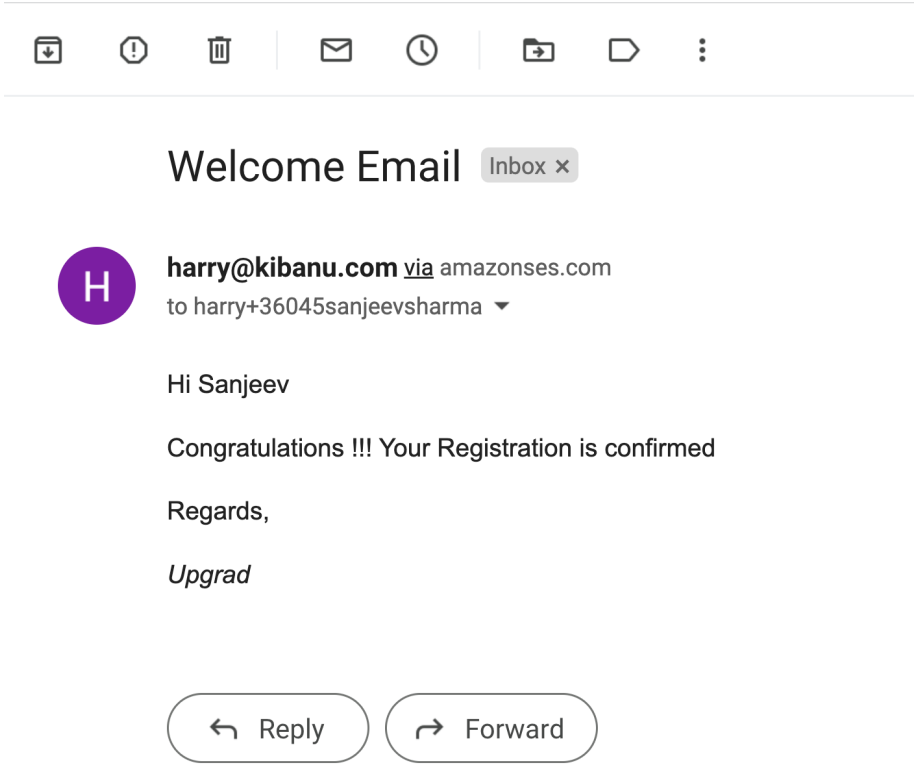
- Now let's test `GET /doctors/{doctorId}/documents/{documentName}` that is responsible for downloading the documents uploaded by the doctor based on the name of the document. Role: ADMIN,USER



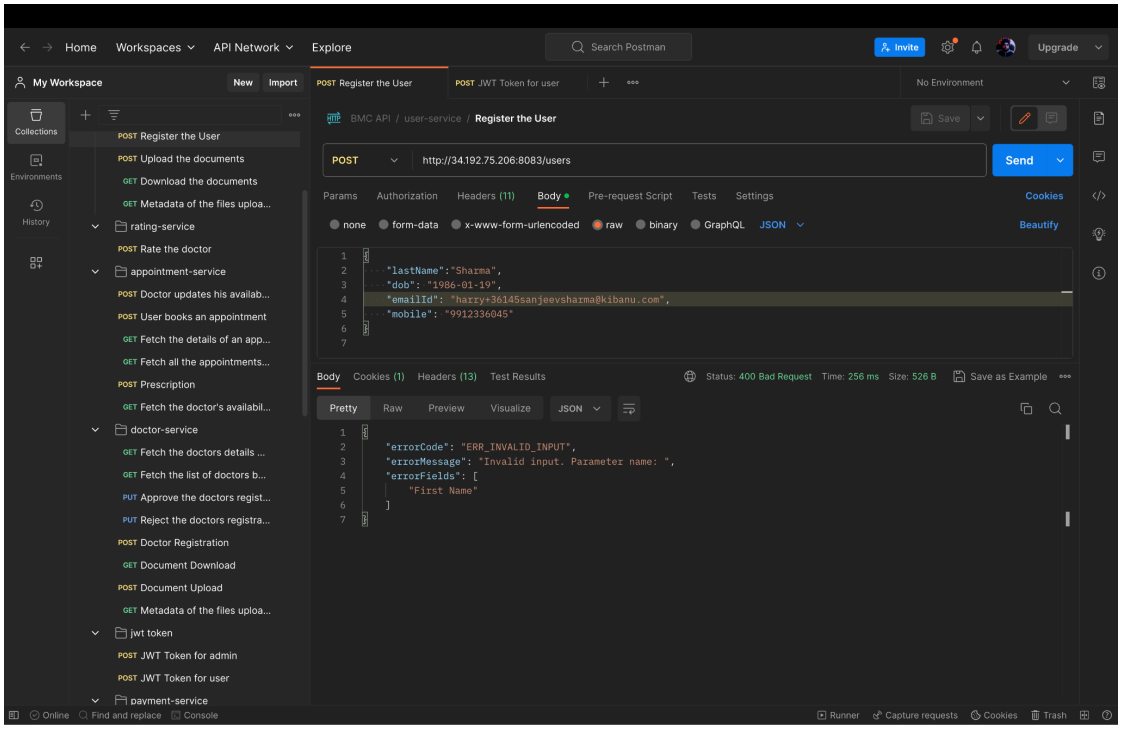
- Now let's test `POST /users` respectively. Request body: firstName, lastName, dob, emailID, mobile. If the token has the role of USER, then the endpoint can be accessed.



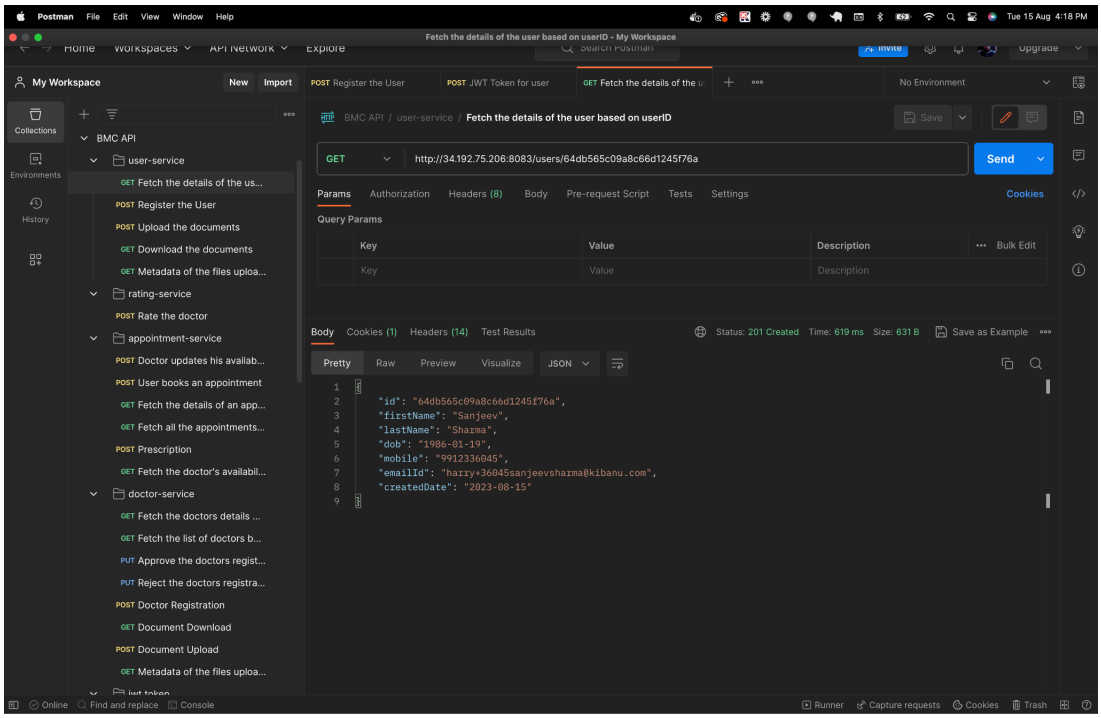




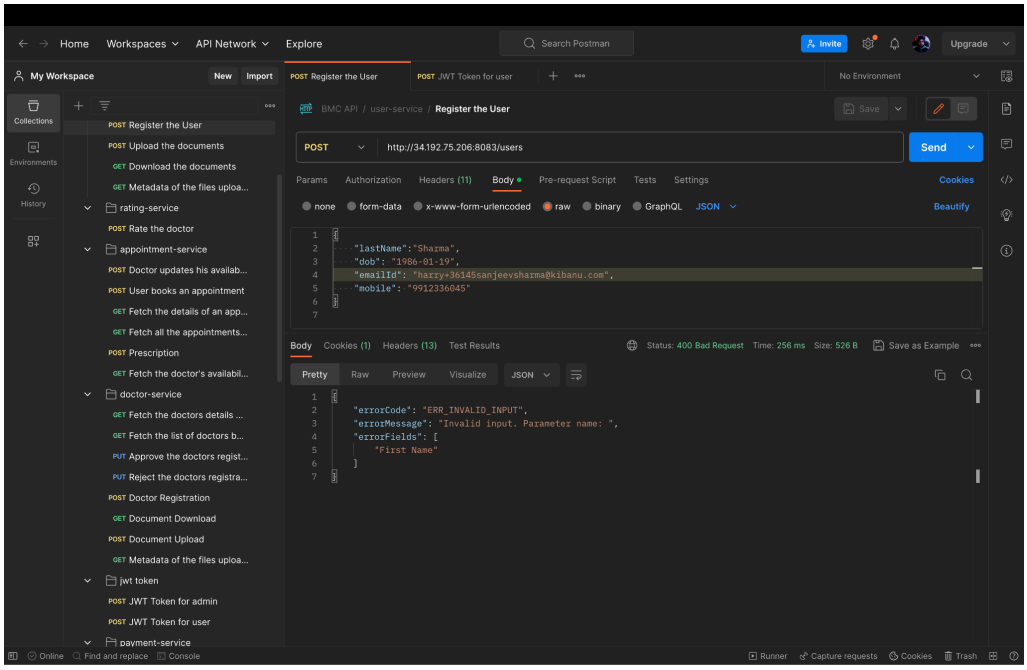
- In same `POST /users` endpoint, let's also test by providing an invalid input, we should expect an error in such case, and it's working (see screenshot)



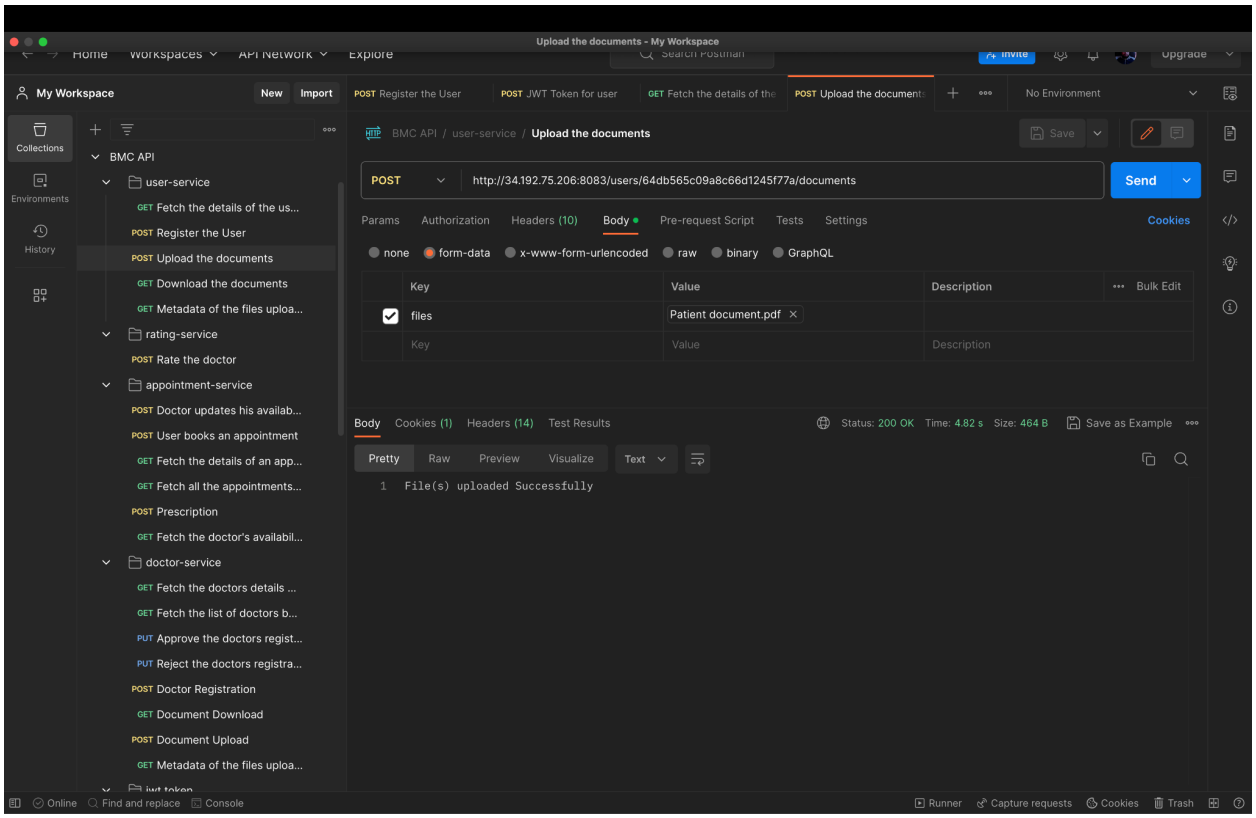
- Now let's test `GET /users/{userID}` respectively. Role: USER, ADMIN



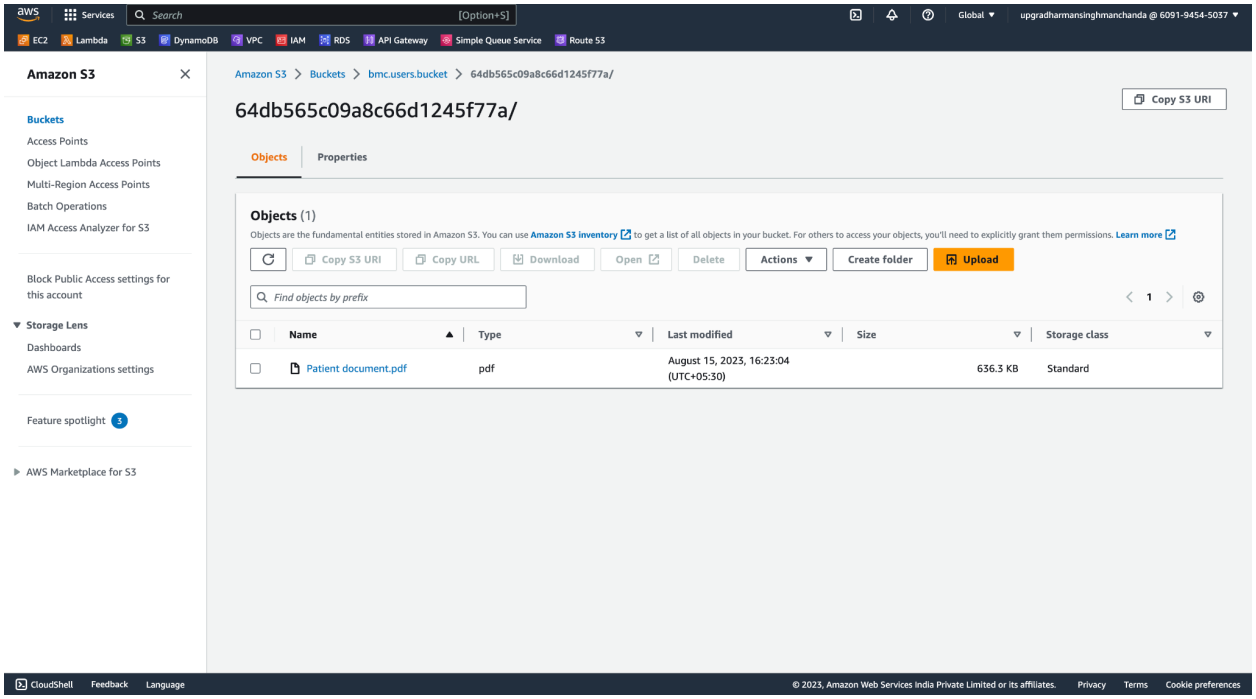
- In same `GET /users/{userID}` endpoint, let's also test by providing an invalid input, we should expect an error in such case, and it's working (see screenshot)



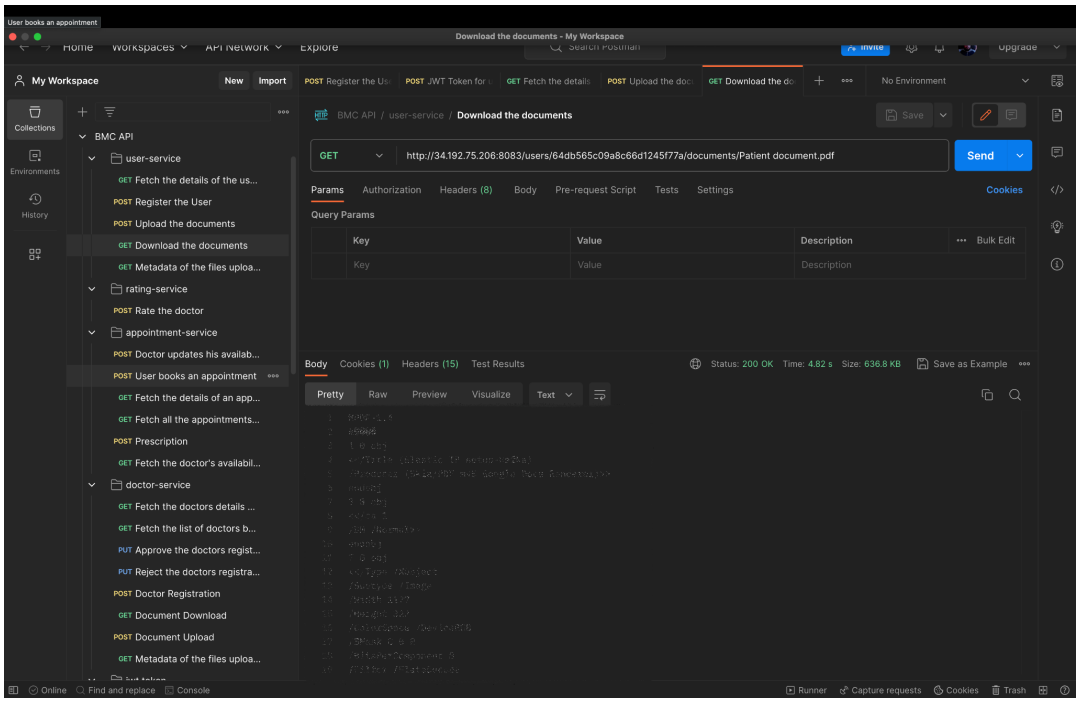
- Now let's test the File upload endpoint: `POST /users/{id}/documents` respectively.



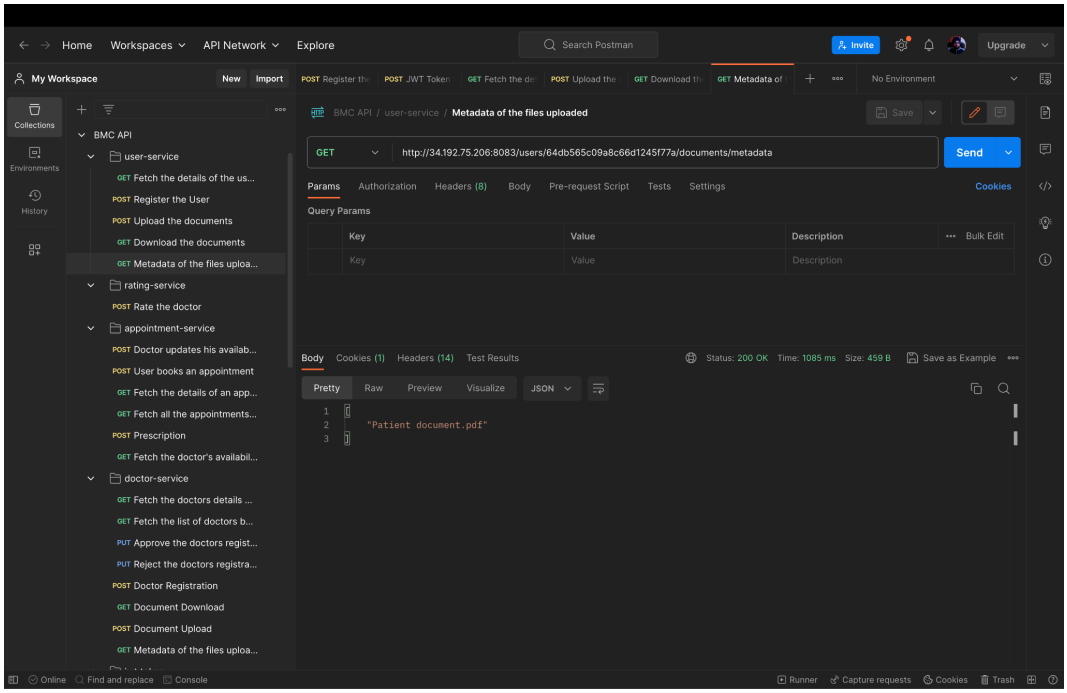
- We can now confirm the file upload by checking the Amazon S3 bucket.



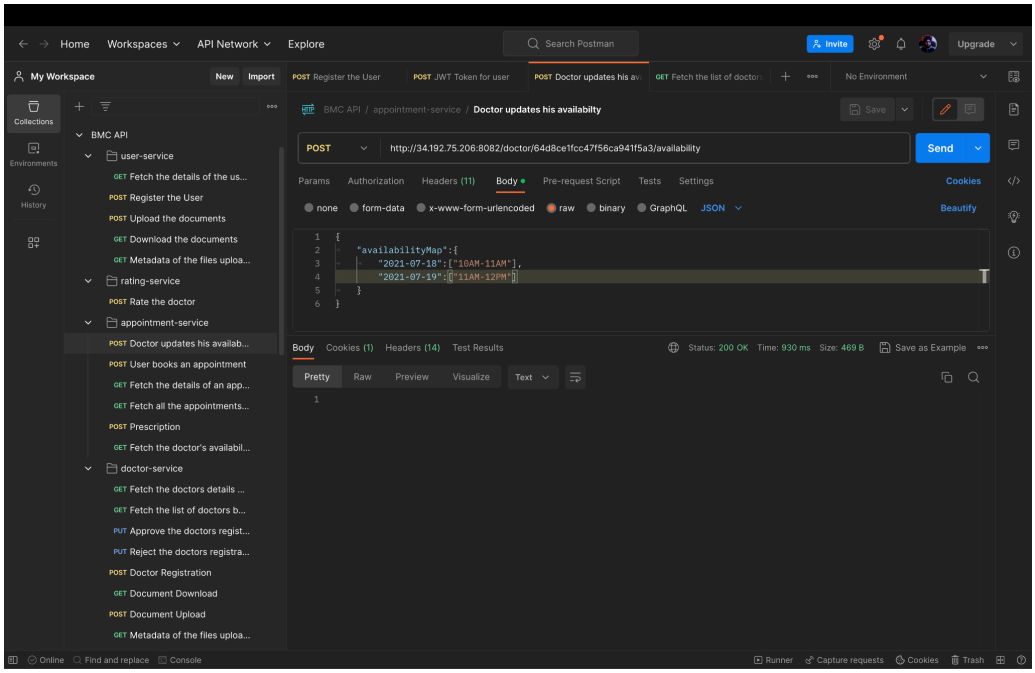
- Now let's test the File download endpoint: `GET /users/{id}/documents/{documentName}` respectively.



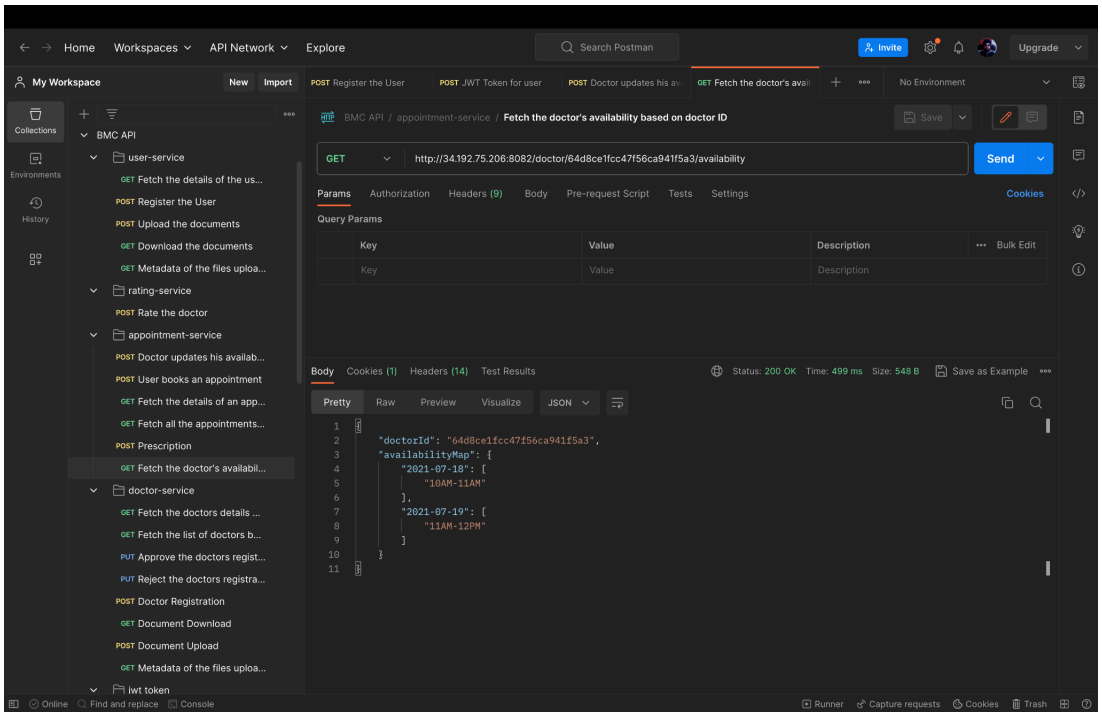
- Now let's test the endpoint that returns the names of the file uploaded by a user: `GET /users/{id}/documents/metadata` respectively.



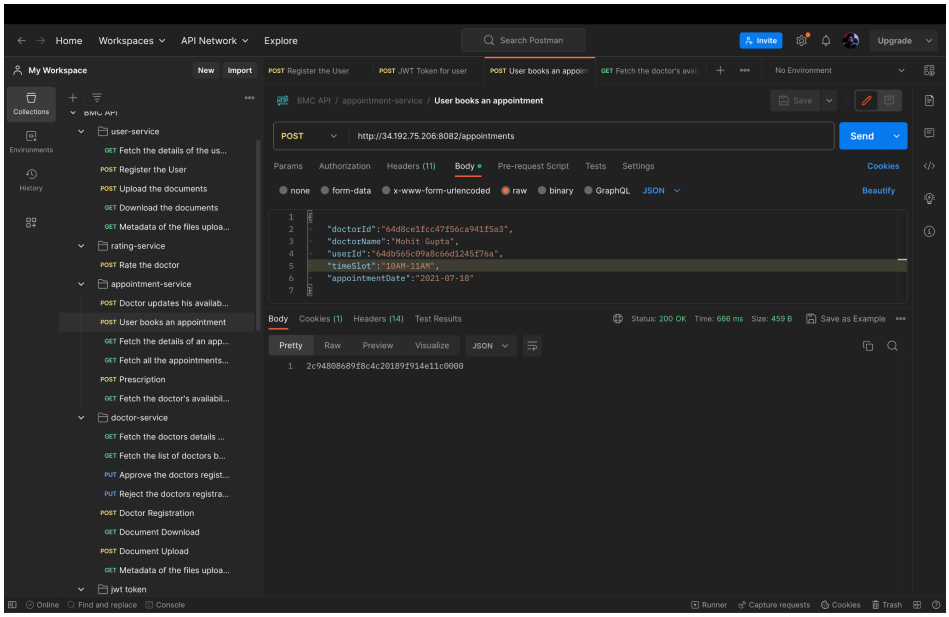
- Now let's test the endpoint responsible for updating the availability of the doctors. `POST /doctor/{doctorId}/availability` respectively. Request body: Availability Map. Role: USER, ADMIN respectively.

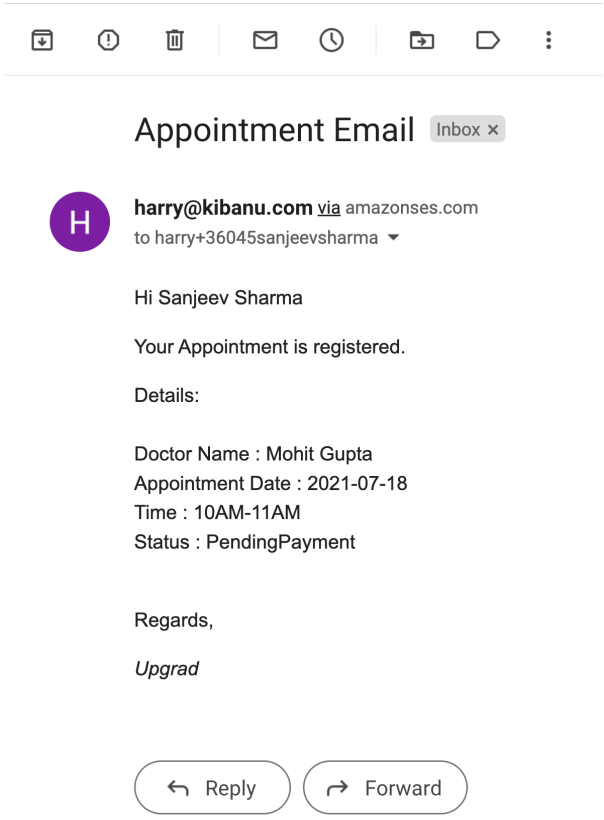


- Now let's test the endpoint responsible for returning the availability of the doctors. `GET /doctor/{doctorId}/availability` respectively. Role: USER, ADMIN

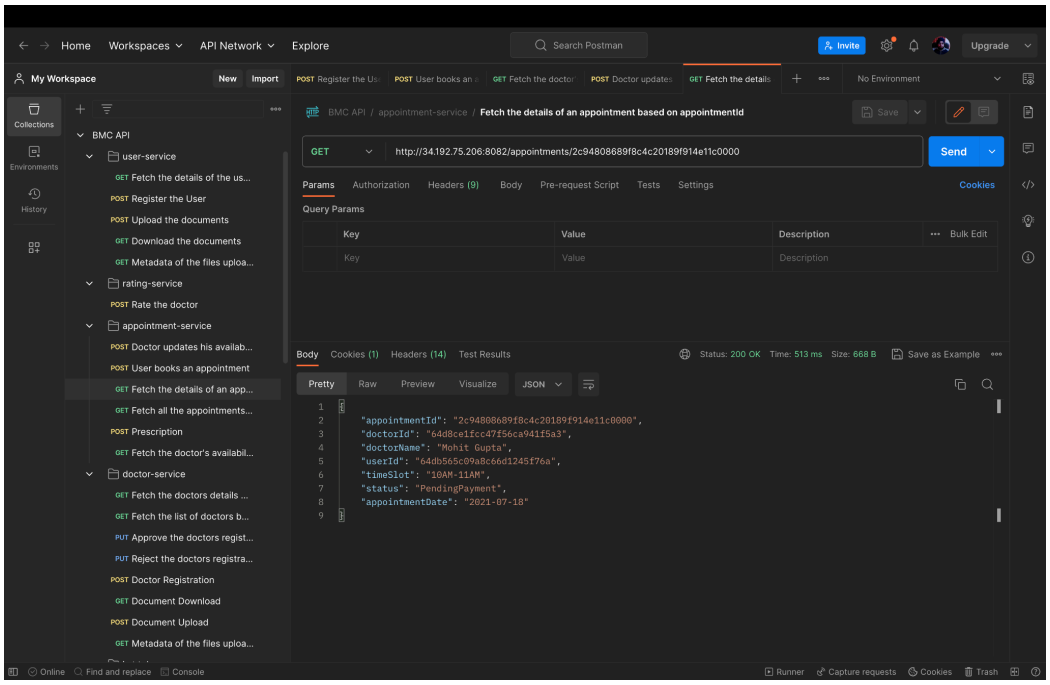


- Now let's test the endpoint responsible for booking an appointment. `POST /appointments` respectively. Request body: doctorId, userId, appointmentDate, timeSlot. Role: USER, ADMIN

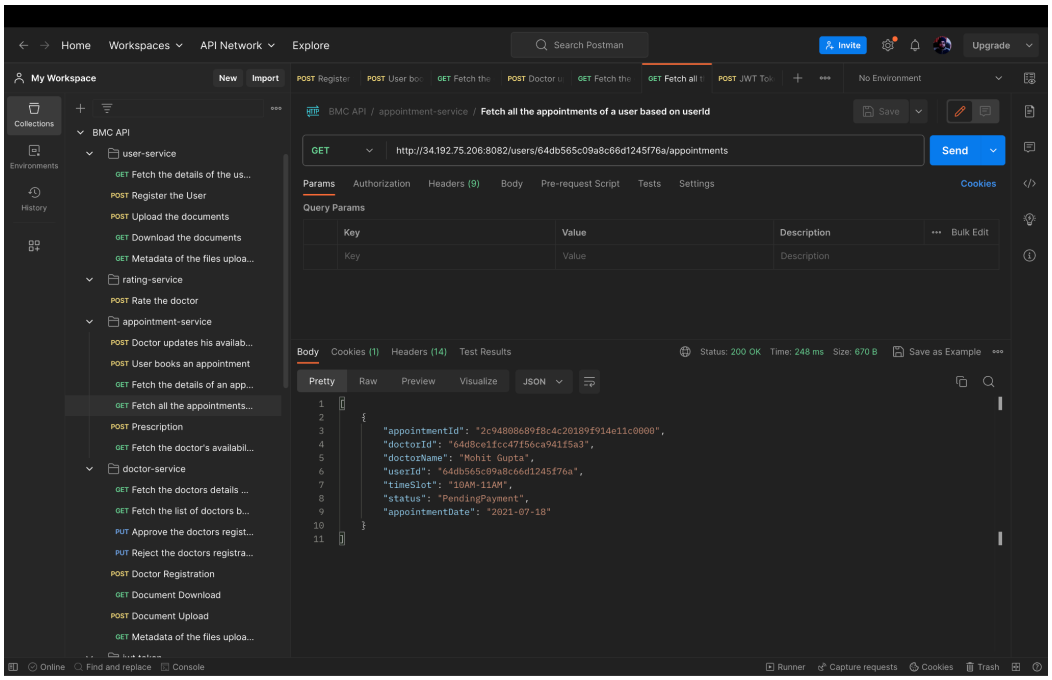




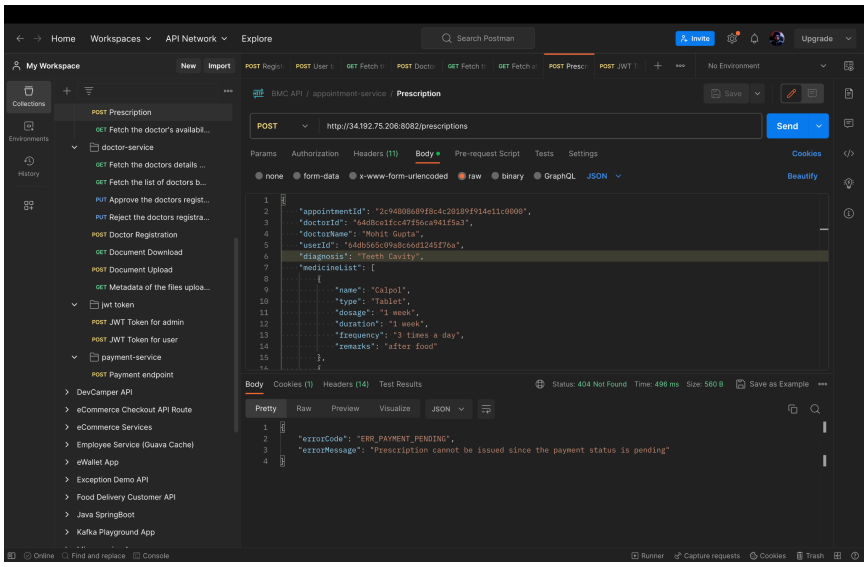
- Now let's test the endpoint responsible for retrieving the details of an appointment. `GET /appointments/{appointmentId}` respectively. Role: USER, ADMIN.



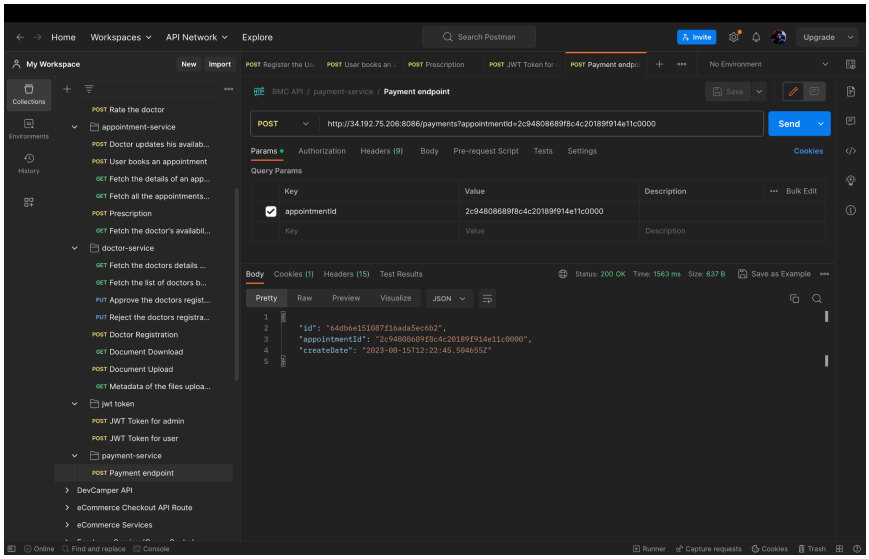
- Now let's test the endpoint responsible for retrieving the details of all the appointments corresponding to a userId. `GET /users/{userId}/appointments` respectively. Role: USER, ADMIN.



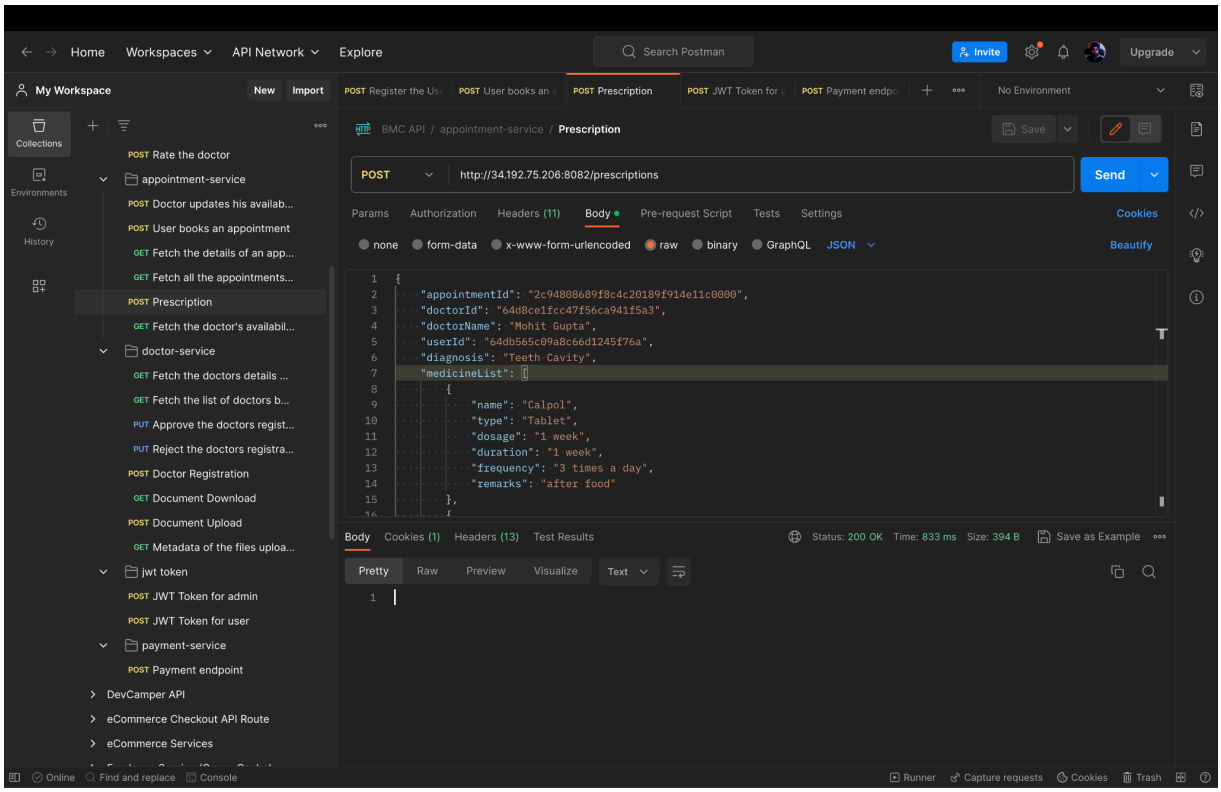
- Now let's test the endpoint responsible for sending the prescriptions for the appointment. As defined, the doctors will send the prescription to the users after the appointment. The prescription can be sent only if the status of the appointment is Confirmed. As we know that when the appointment is made, the status of the appointment is set to 'PendingPayment'. Once the payment is made, the status will change to Confirmed. The API endpoint is `POST /prescriptions` respectively. Request body: appointmentId, doctorId, userId, diagnosis, medicineList. Role: USER

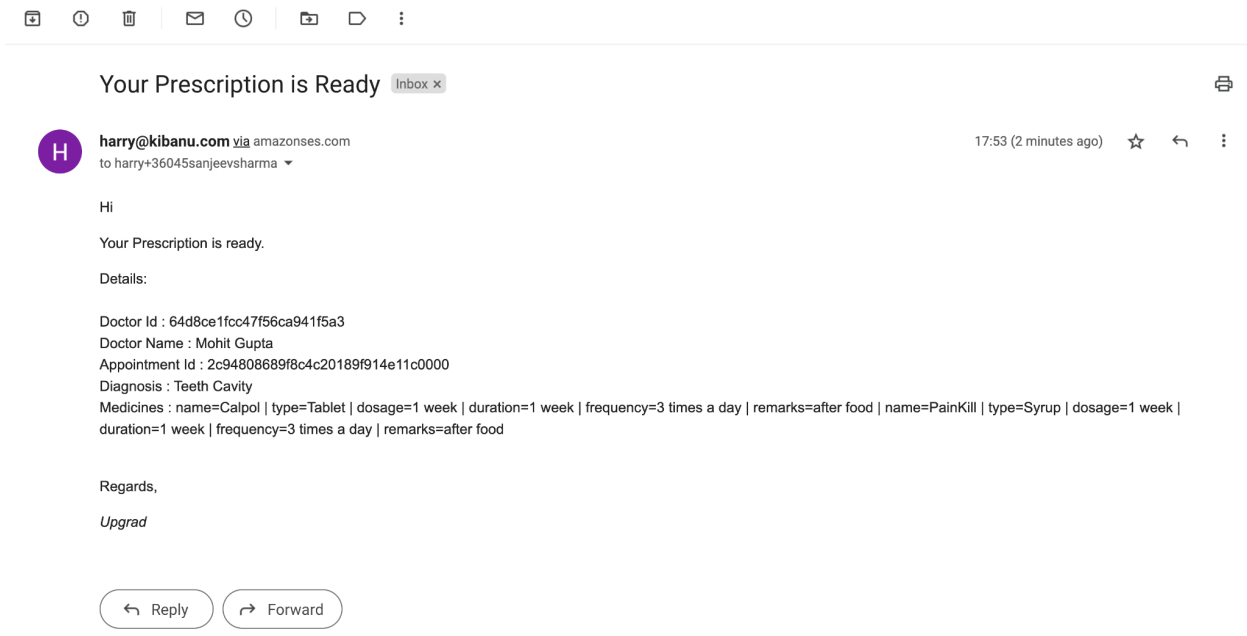


- As we can see there is an error thrown to the `POST /prescriptions` API. This is good because it's expected.
- It's time to test the Payment Endpoint, `POST /payments?appointmentId={appointmentId}` respectively. Role: USER, no request body need to be sent.

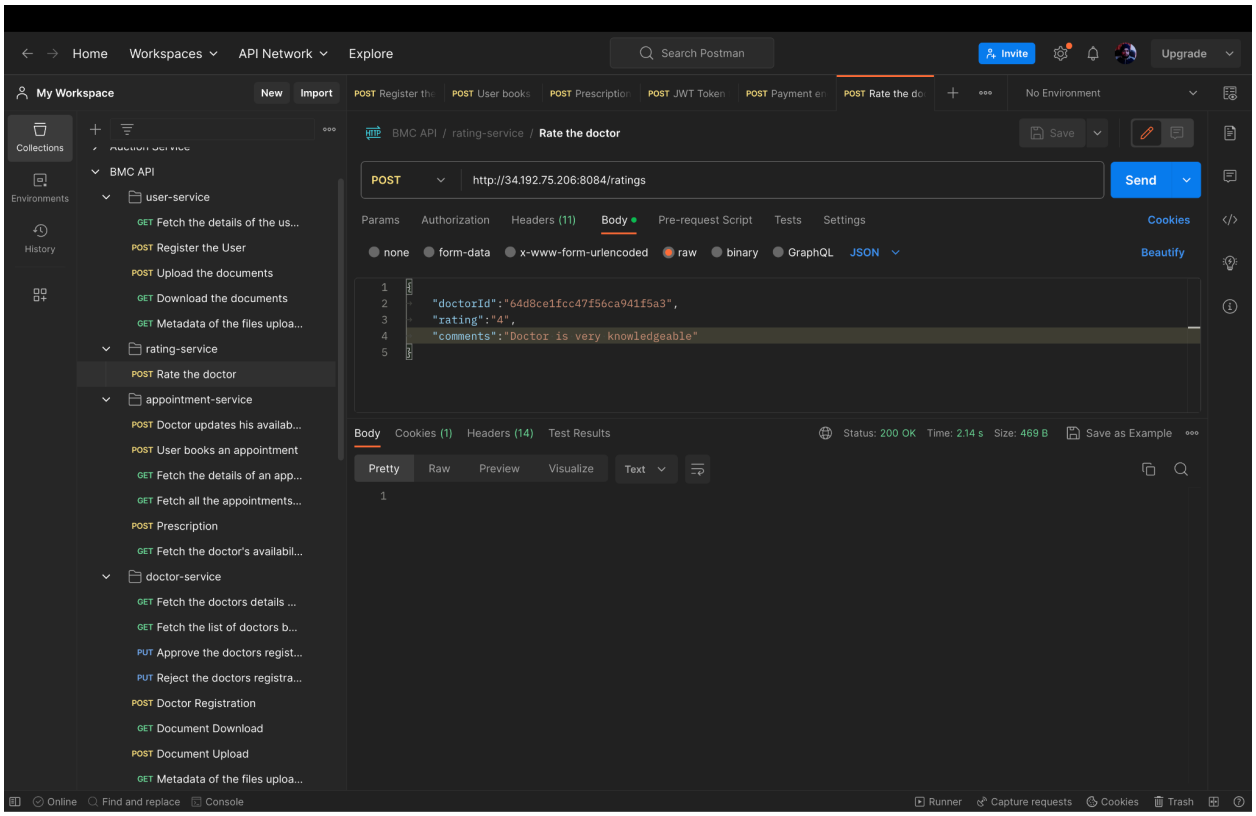


- Now as payment is done, let's test the `POST /prescriptions` again with same details as last time. This time we get `200 OK` respectively and also get the email as expected.





- Now let's test the endpoint that is used by the users to submit the ratings of their experience with the doctor with whom they had an appointment. `POST /ratings` respectively. Request body: doctorId, rating. Role: USER. Here also we get `200 OK` respectively.



Code Explanation by Service

In the development of the BookMyConsultation Project, our team extensively crafted a robust architecture by employing a total of 9 microservices, each contributing uniquely to make the whole project work smoothly. These sets of microservices create a seamless and efficient healthcare consultation platform. Let's take a closer look at the building blocks of the BookMyConsultation Project.

1. Service Registry: At the heart of the microservices-based application, lies the challenge of managing service instances dynamically as they come online or go offline. This dynamicity is efficiently managed by the Service Registry, akin to a sophisticated database that maintains a record of available instances of every microservice within the application. In our architectural design, the Eureka framework is adeptly employed to handle this critical task. With its agility in tracking service instances, Eureka ensures that the system remains responsive and fault-tolerant, effortlessly accommodating new services while elegantly handling service unavailability.

2. BMC Gateway (API Gateway): In the pursuit of a user-friendly interface and a streamlined API management process, the BMC Gateway is an indispensable component in our application. This intelligent API gateway takes charge of intercepting all incoming requests and skillfully routing them to their designated services. With its role as a mediator between the UI and underlying microservices, the BMC Gateway not only enhances security but also optimizes the user experience by offering a consolidated and organized interaction point.

3. Notification Service: The Notification Service serves as a vigilant listener, tuned in to the Kafka topics that serve as conduits for messages exchanged between services. This vigilant service ensures timely notifications, orchestrating the delivery of critical emails triggered by pivotal events such as Doctor's approval, Doctor's rejection, Appointment confirmation, and Prescription. By fulfilling this pivotal role, the Notification Service guarantees users and doctors stay informed and engaged throughout their healthcare journey.

4. Rating Service: The user experience should be the heart of any platform, and the Rating Service is used to enhance that same experience. This service empowers the users to rate their interactions and experiences with healthcare providers.

5. User Service: User onboarding is the foundational step in establishing a seamless connection between individuals and the healthcare ecosystem. The User Service microservice handles this critical aspect, offering a set of endpoints that facilitate user registration. Beyond the basics, users can also upload medical documents, ensuring that their past medical history is securely integrated into the platform.

6. Doctor Service: Empowering doctors to seamlessly integrate into the BookMyConsultation ecosystem makes a strong case for a sophisticated microservice that handles many facets of Doctor’s onboarding journey. This Doctor Service steps into this role, providing designed endpoints for not only doctor registration but also embarking on the verification process overseen by BMC employees, ensuring the accuracy and authenticity of the doctor's details. Upon approval, the doctor's status transitions from Pending to Active, or in the case of rejection, to Rejected, managing the influx of healthcare providers into the platform.

7. Appointment Service: Scheduling appointments seamlessly, a cornerstone of the BookMyConsultation platform, is facilitated by the Appointment Service microservice. With a suite of defined endpoints, users can effortlessly check doctor availability and schedule appointments at their convenience. Similar to the User Service, the Appointment Service also accommodates document uploads, facilitating the exchange of crucial medical information and ensuring the continuity of care..

8. Payment Service: The endpoints defined within the payment service are used to make the payment, and while that happens, the status of the appointment will change from PendingPayment to Confirmed.

9. Security Provider: The Security Provider microservice fortifies the BookMyConsultation ecosystem with a robust security layer. At the heart of this service lies the generation and management of JSON Web Tokens (JWTs). While token validation happens in each microservice, the Security Provider stands as the authoritative source of truth. This centralization ensures consistency, accuracy, and a unified approach to security, offering users and doctors the peace of mind while doing healthcare interactions.

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