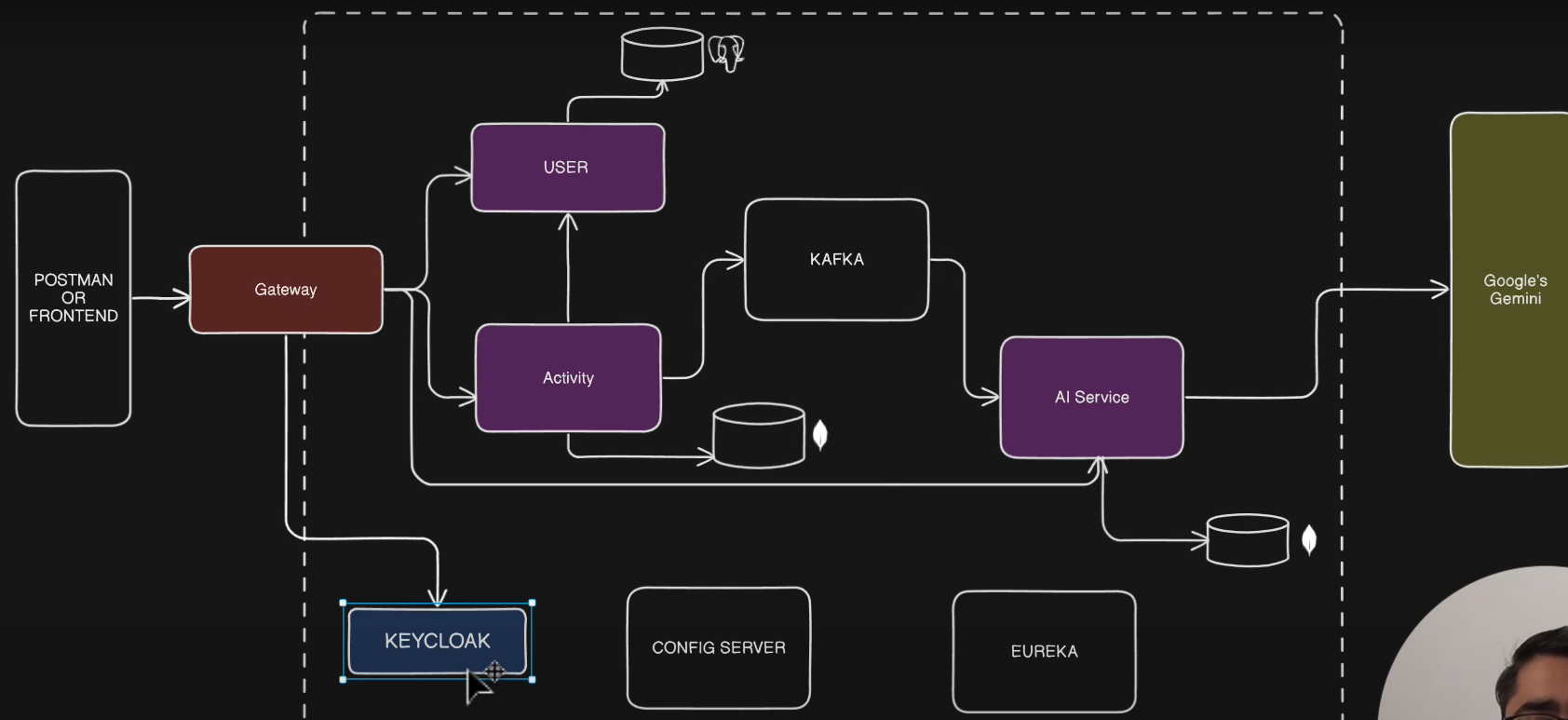
**Spring Boot-React-Fitness App**

**Tech Stack:**

* Spring boot + React Frontend
* Eureka Server (Spring cloud Netflix)
* Spring cloud gateway
* Keycloak
* Apache Kafka
* PostgreSQL/ MySQL / MongoDB
* Google Gemini API
* Spring Cloud config server

**Application Architecture Diagram:**



**Whenever a user makes a request — like sign up, login, view workout plans, track steps, etc. — it first goes through the API Gateway.**

**API Gateway acts as the central entry point for all requests.**

* Routing → Forwards requests to the correct microservice (User, Activity, AI, etc.)
* Authentication → Uses Keycloak for validating tokens.
* Rate limiting & security → Protects backend services**.**

**Example:**

* If a user logs in → Gateway sends the request to Keycloak.
* If a user requests their workout history → Gateway routes the request to the User Service.

**Keycloak (Authentication & Authorization)**

* Handles user authentication and token management.
* When a user logs in/signup:
  + The request goes to Keycloak.
  + Keycloak verifies credentials.
  + Returns an Access Token (JWT) to the client via Gateway.
* For each protected API call:
  + Gateway checks the JWT with Keycloak.
  + If valid → forwards the request.
  + If invalid → rejects it.

**User Service (Microservice)**

* Manages user profiles: name, email, fitness goals, BMI, subscription, etc.
* Stores user data in a PostgreSQL database.
* Sends updates to Kafka when new users sign up or update profiles.

**Example:**

* **When a user sets a fitness goal, the User Service updates the DB and notifies other services via Kafka**

**Activity Service (Microservice)**

* Handles workout tracking, calories burned, step counts, daily logs, etc.
* Stores activity data in MongoDB (for fast, flexible storage).
* Publishes events to Kafka whenever activity data changes.
* Consumes messages from Kafka when AI Service needs updates.

**Kafka (Event Streaming Platform)**

* Acts as a messaging backbone between services.
* Why needed:
  + Decouples services → Activity Service, User Service, and AI Service can communicate asynchronously.
  + Ensures reliability → If AI Service is down, Kafka holds the events until it recovers

**AI Service**

* Uses fitness data + activity history to give personalized recommendations.
* Integrates with Google’s Gemini AI for:
  + Predicting workout schedules.
  + Recommending diets.
  + Providing motivational tips.
* Stores analyzed insights in MongoDB.

**Google’s Gemini AI Integration**

* External AI service.
* The AI Service sends relevant user + activity data.
* Gemini processes and returns:
  + Suggested workouts.
  + Diet charts.
  + Health insights.
  + Personalized predictions.

**Databases**

* PostgreSQL → Used by User Service (structured relational data).
* MongoDB → Used by Activity & AI Service (fast, schema-less fitness tracking data).

**Step-by-step:**

1. User opens app → Requests a personalized workout.
2. Request → Gateway → Verifies JWT with Keycloak.
3. Gateway sends the request to Activity Service.
4. Activity Service fetches user workout history from MongoDB.
5. Sends the data via Kafka → AI Service.
6. AI Service → Calls Gemini API for recommendations.
7. Gemini → Returns insights → AI Service processes them.
8. AI Service saves results → MongoDB.
9. Gateway returns response → Frontend / Postman.

**UserMicroservices:**

1.create a project with few important dependencies (web, postgres, validation)

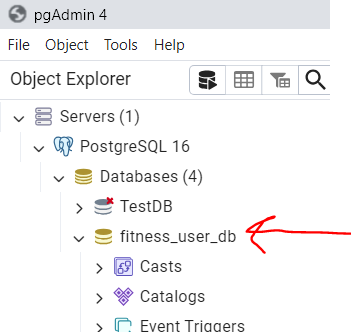
2.download zip file and open it in IDE.

**DB Configuration:**

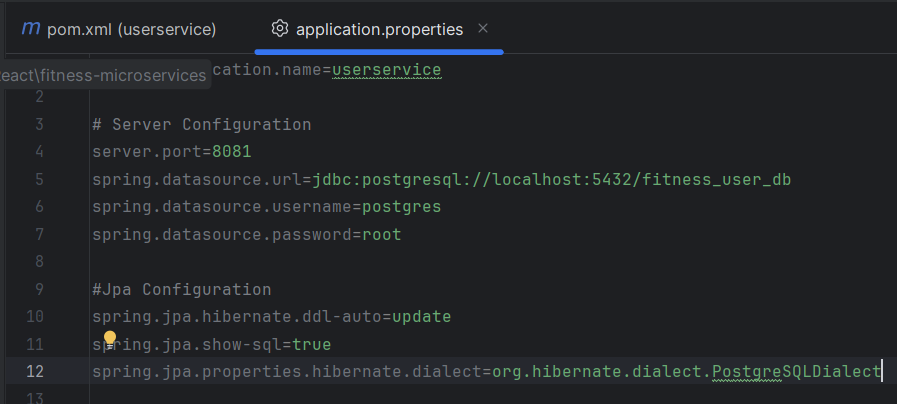
In UserMicroservice, we will be using postgresql database.

Open PgAdmin: (password: root)

Create DB schema “**fitness\_user\_db**” in pgadmin



Configure application.properties file with below details.



**Entity:**

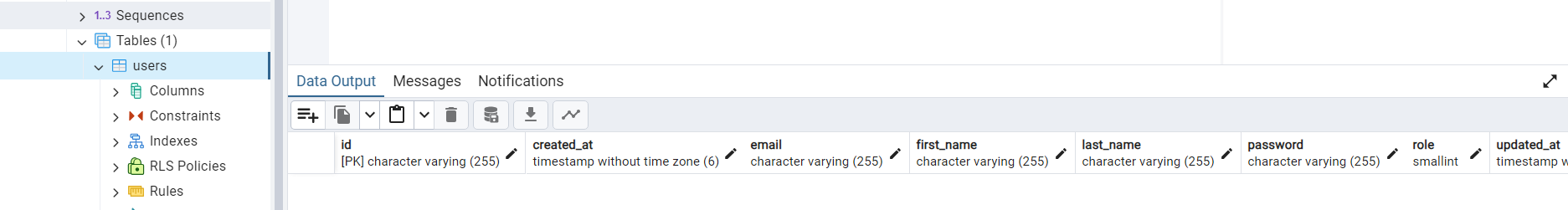
1.create a user entity with few fields

2.provide unique identifier (@id and @generatevalue)

3.run the application and can see that application is up and running.

4.we observed that table got created in pgadmin

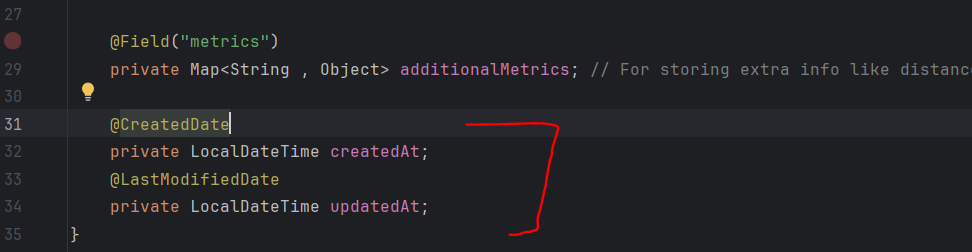
5.right click on table>>view/edit data>>ALL rows

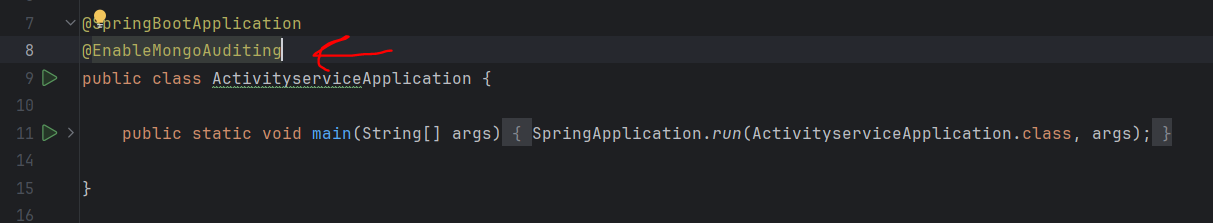


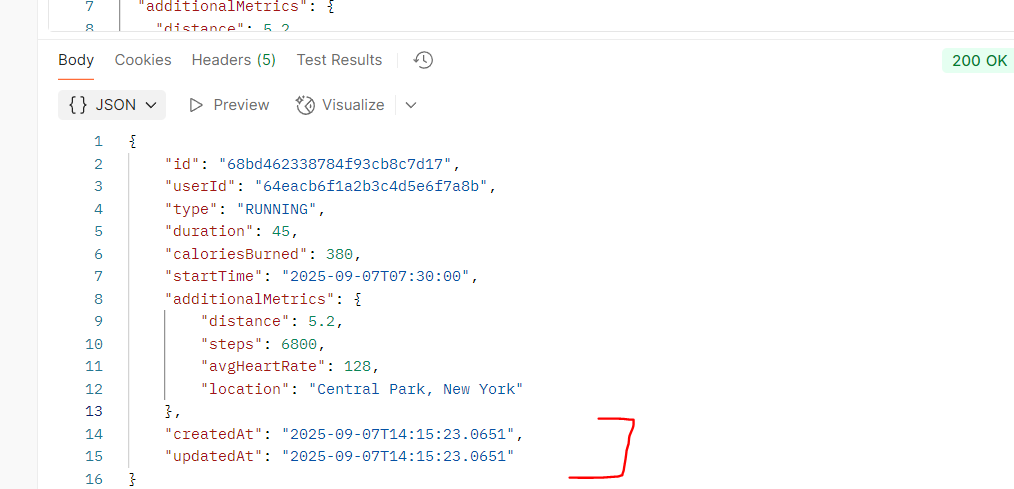
**Activity Microservices:**

To make this work (createdAt and updatedAt) in mongoDb, we need to enable **Mongo auditing** in configuration class or in main application class.

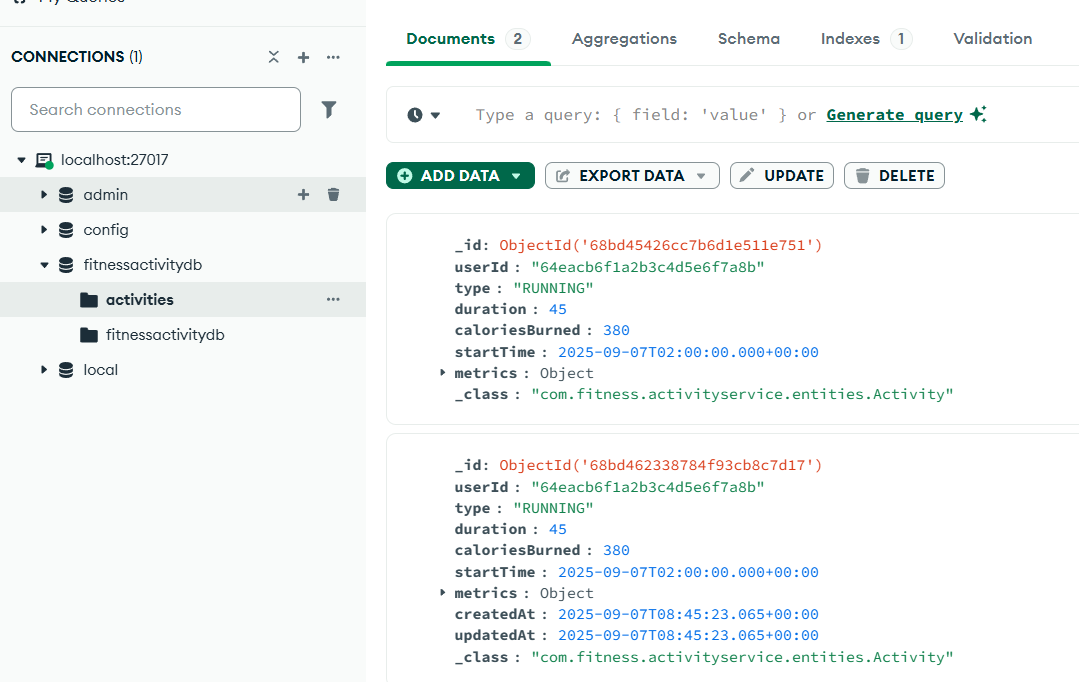
Else, we would be getting (null) in both the fields.







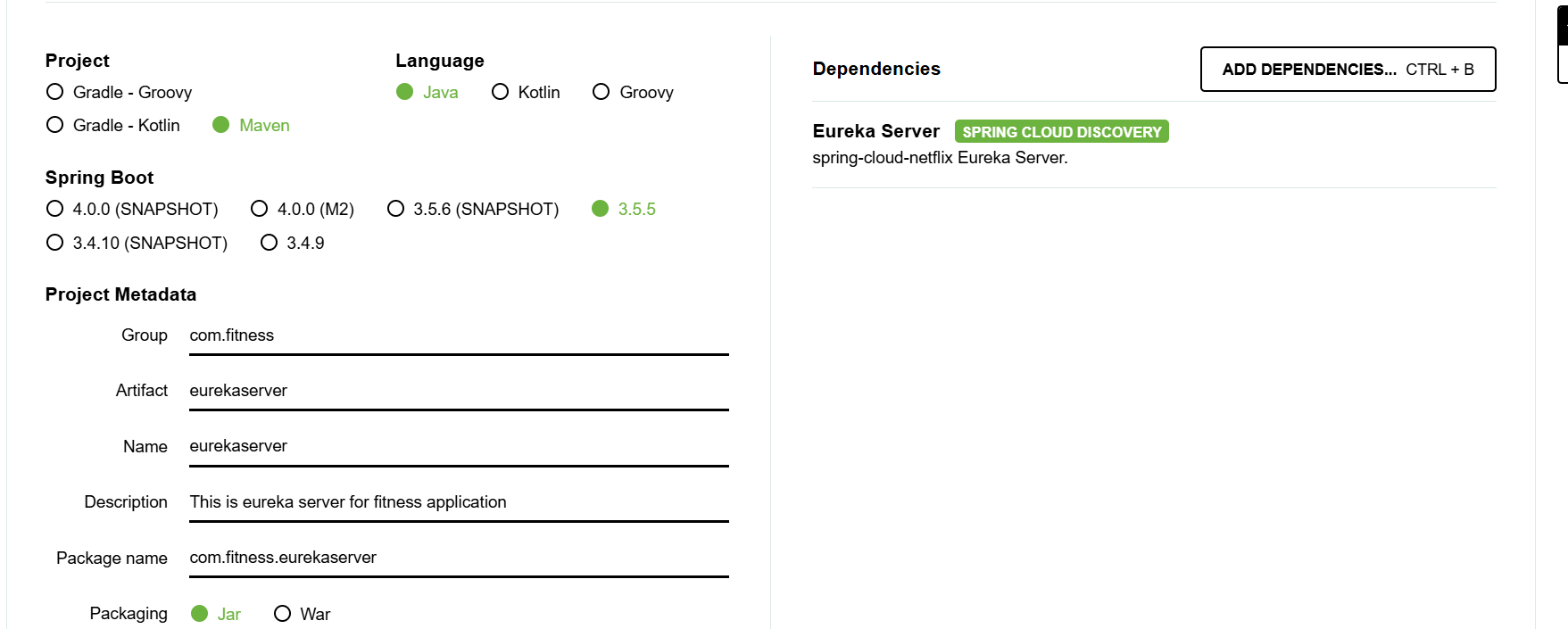
**We can also see the activity details has been saved into Mongo DB successfully (once reloads the page)**



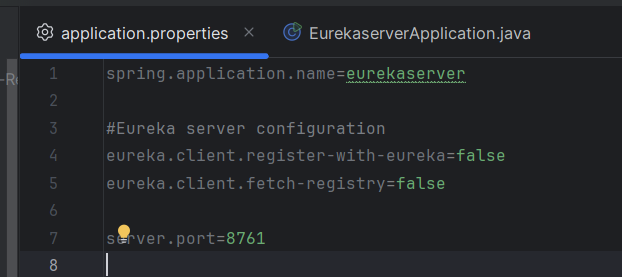
Since we have completed two microservices (User and Activity) and both of them are working fine. We would be moving ahead and working on Eureka server.

**Eureka Server:**

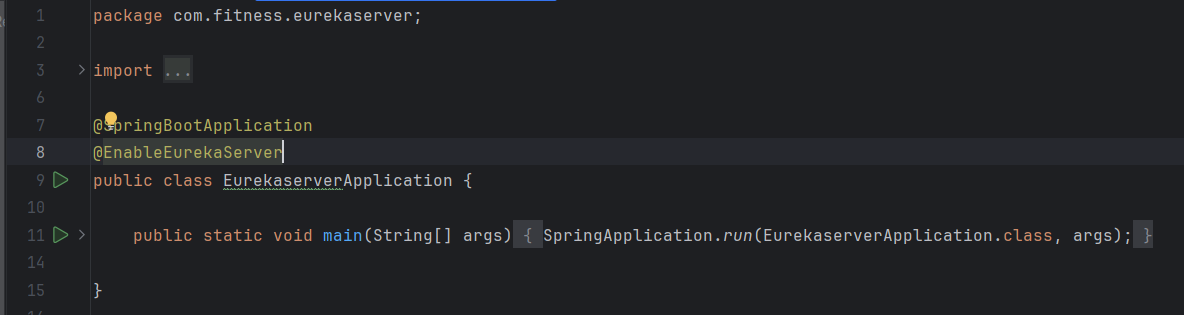
Eureka server is service registry where all other microservices gets registered.



**Application properties file:**

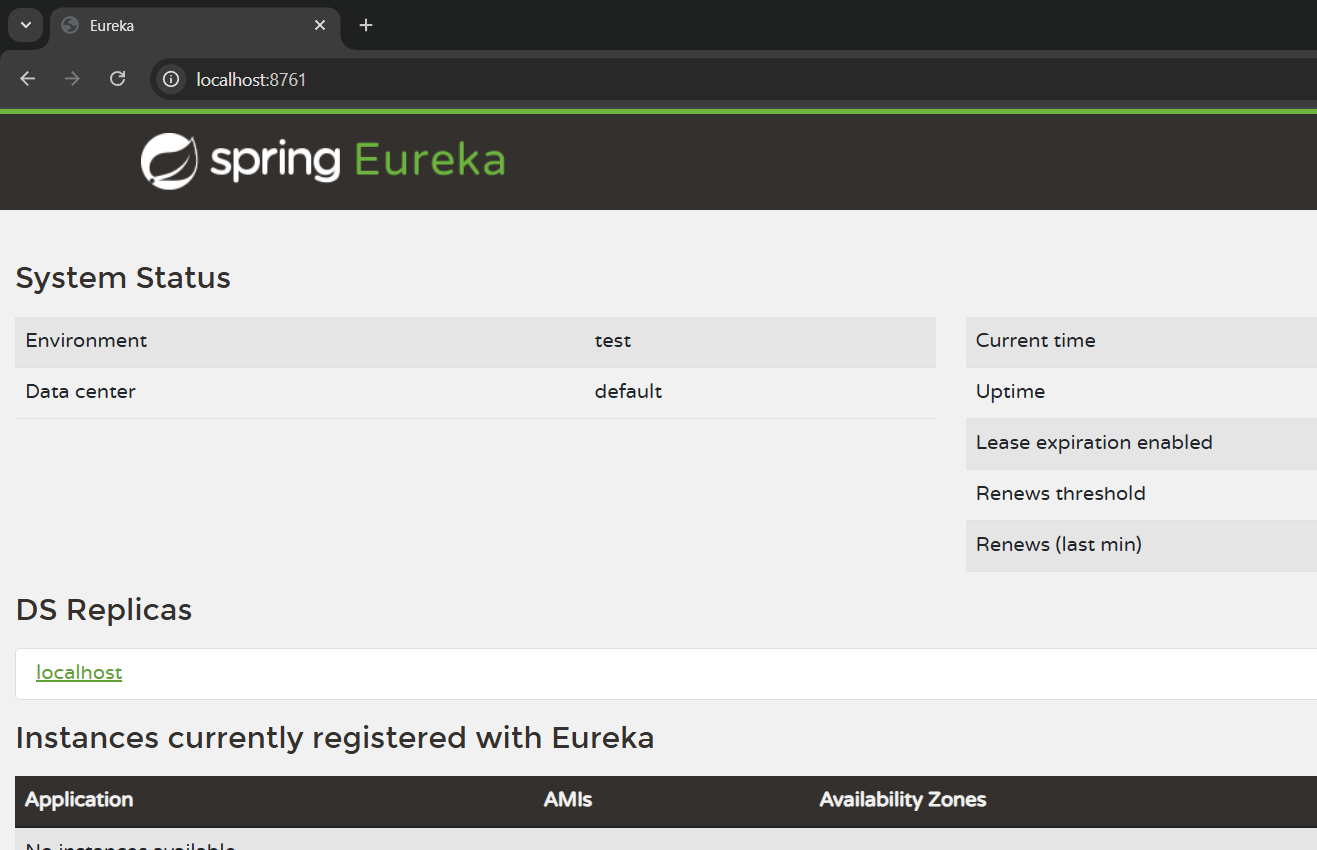


**Enable Eureka Server:**



Now, once we have setup all above, we can start our eureka server application and see that application is up and running on port 8761.

Since we have not registered any microservices to eureka yet, it shows no instances available.

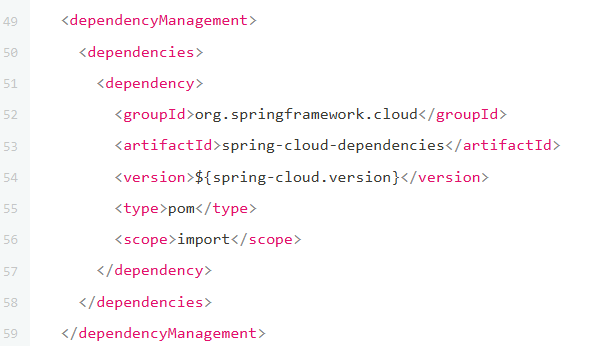


**Register All Microservices to Eureka server:**

1.To register all microservices to eureka, we need to add two dependencies in pom (eureka discovery client) as well as spring cloud (with version).



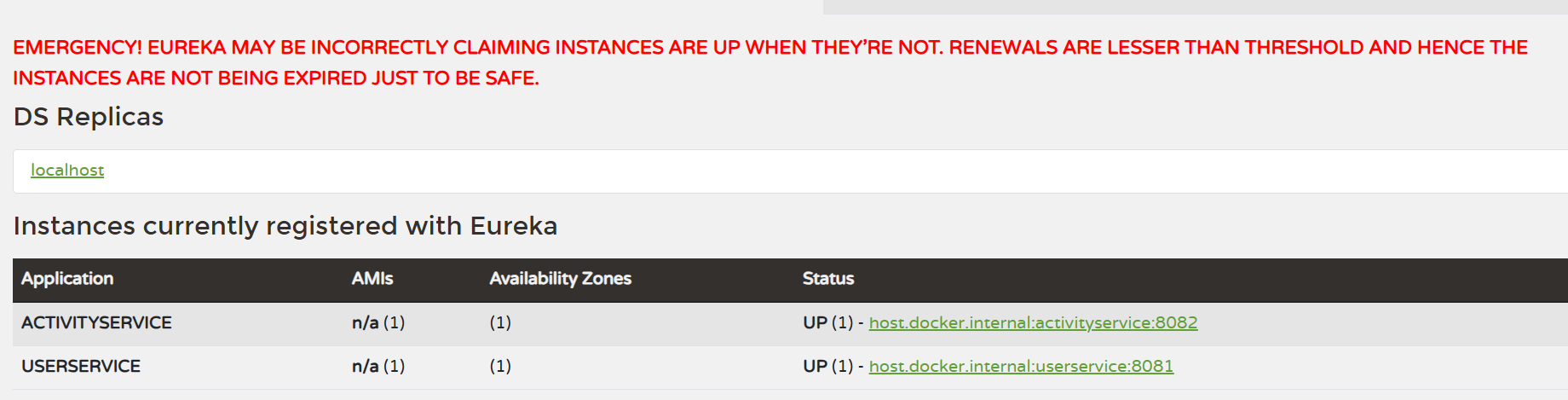




**2.**We need to add one property in both the microservices so that it will register to 8761 (eureka server)



**3.Now, restart all the microservices and can see that all of them have been registered successfully to eureka server.**



**4.Warning: Eureka server uses heartbeat mechanism and it means Eureka Server is not receiving enough heartbeats from your registered microservices. Services keep sending heartbeat to server stating that they are still up. Once heartbeat does not come from services, server deregister them automatically.**

But we can ignore this warning.

**Interservice communication between User and Activity:**

**Problem:**

We observed that when we tried to create activity, we had user which was not in DB and (it was automatically generating)

But it should be like that, when we try to create activity, it should create activity for the user which is already available in user DB.

That means, activity service should communicate with user service and should create activity for that particular user.

**There are several ways to make interservice communication:**

<https://docs.spring.io/spring-framework/reference/integration/rest-clients.html>

**Web Client:**

We will be making an api call from activity service to user service to validate or check if userid is valid or not (means that user id is available in DB or not)

WebClient is part of **Spring WebFlux** and is the **non-blocking, reactive** replacement for **RestTemplate** in Spring Boot.

**Steps to use WebClient for Interservice:**

1.Add dependency (spring webflux) in pom.xml (microservices)

2.Create WebClientConfig (configuration class) in config package.

3.Create Bean which will be load balanced, this bean will give instance of webClientBuilder.

4.We used load balanced because we want service communication with the service name (@LoadBalanced)

5. Create a method userServiceWebClient to expose instance of WebClient



3.Inject WebClient in service

4.Using Webclient for Post requests.

TASK: include end to end workflow steps from chatgpt (to provide code and exact workflow)

**AI Service:**

Till now, we have 2 services (User and Activity).

User service will be managing users and activity service will be managing activity of users (by interservice)

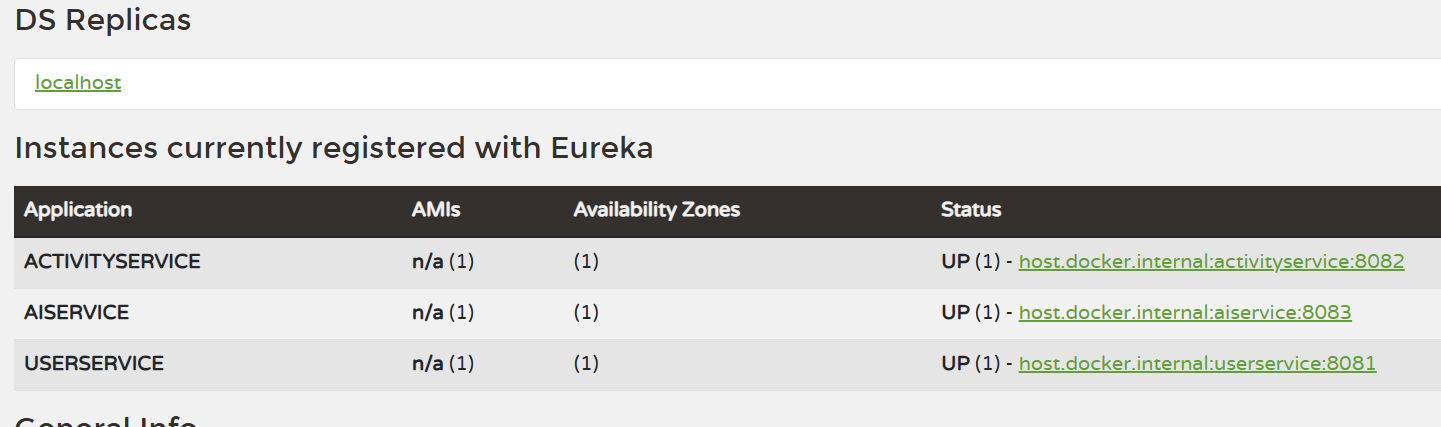
Now, we will be creating AI service which will generate recommendation based on activity given for users.

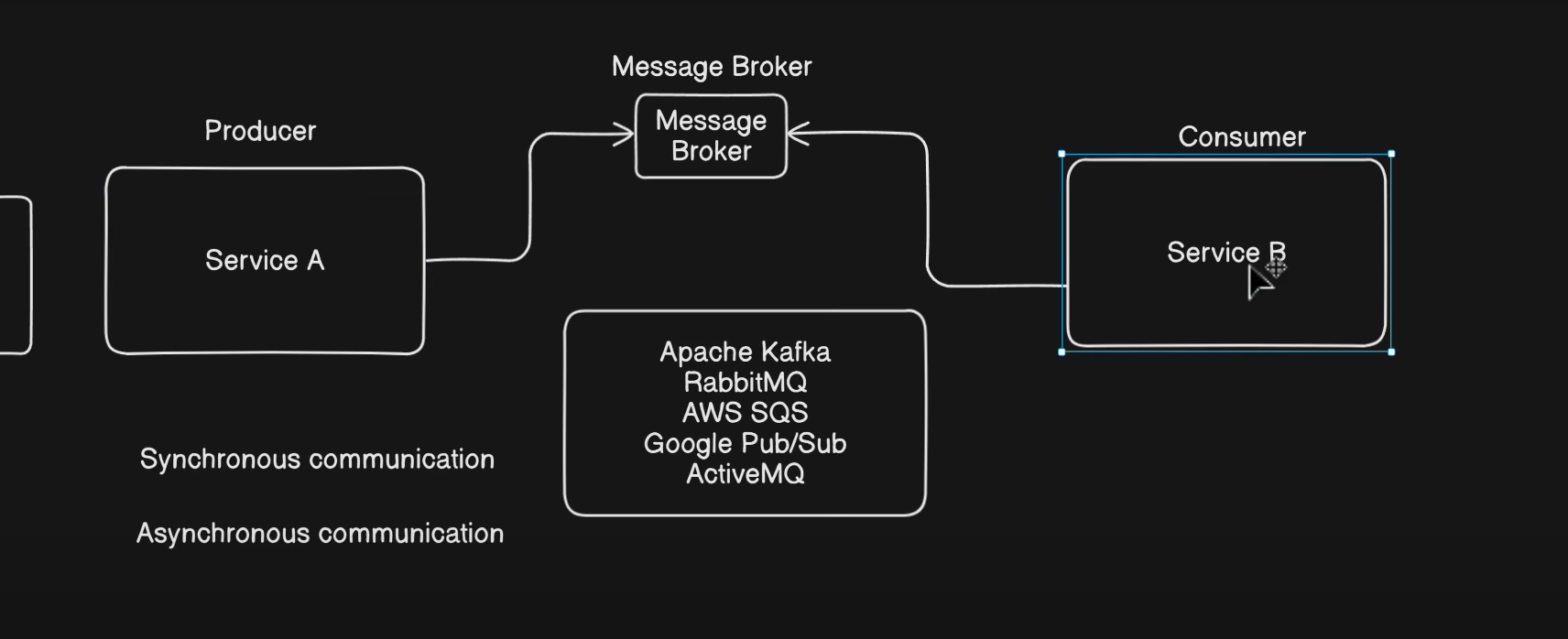
1.create project with dependencies.

2.create db in mongo

3.configure DB in application properties file.

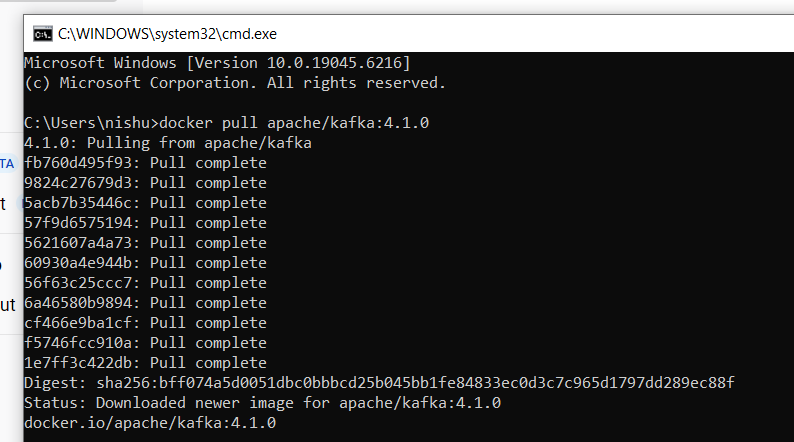
4.Run the application and can see that AI service has been registered successfully with Eureka server.

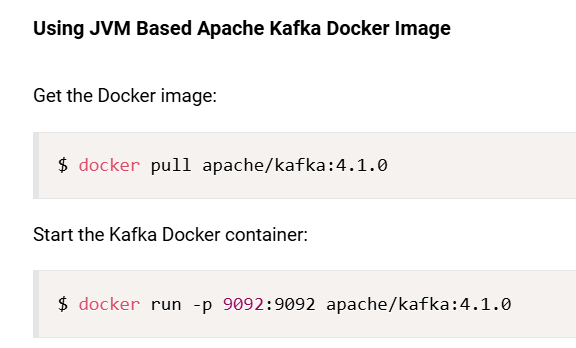




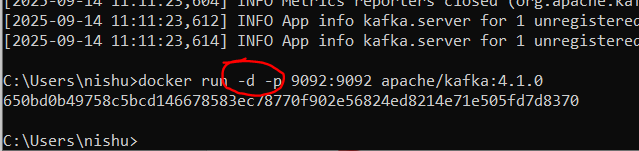
<https://kafka.apache.org/quickstart>

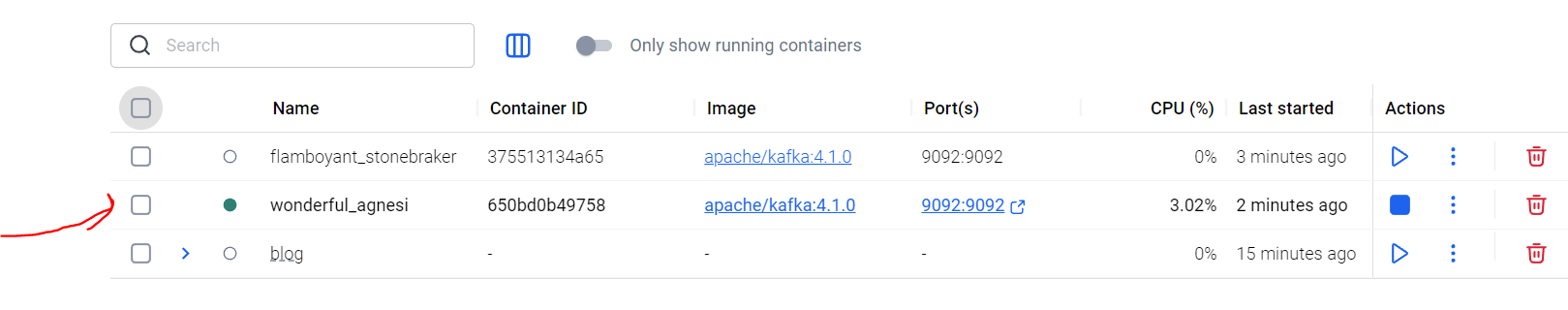
We will use kafka through docker environment so that we don’t have to install/run through manually.





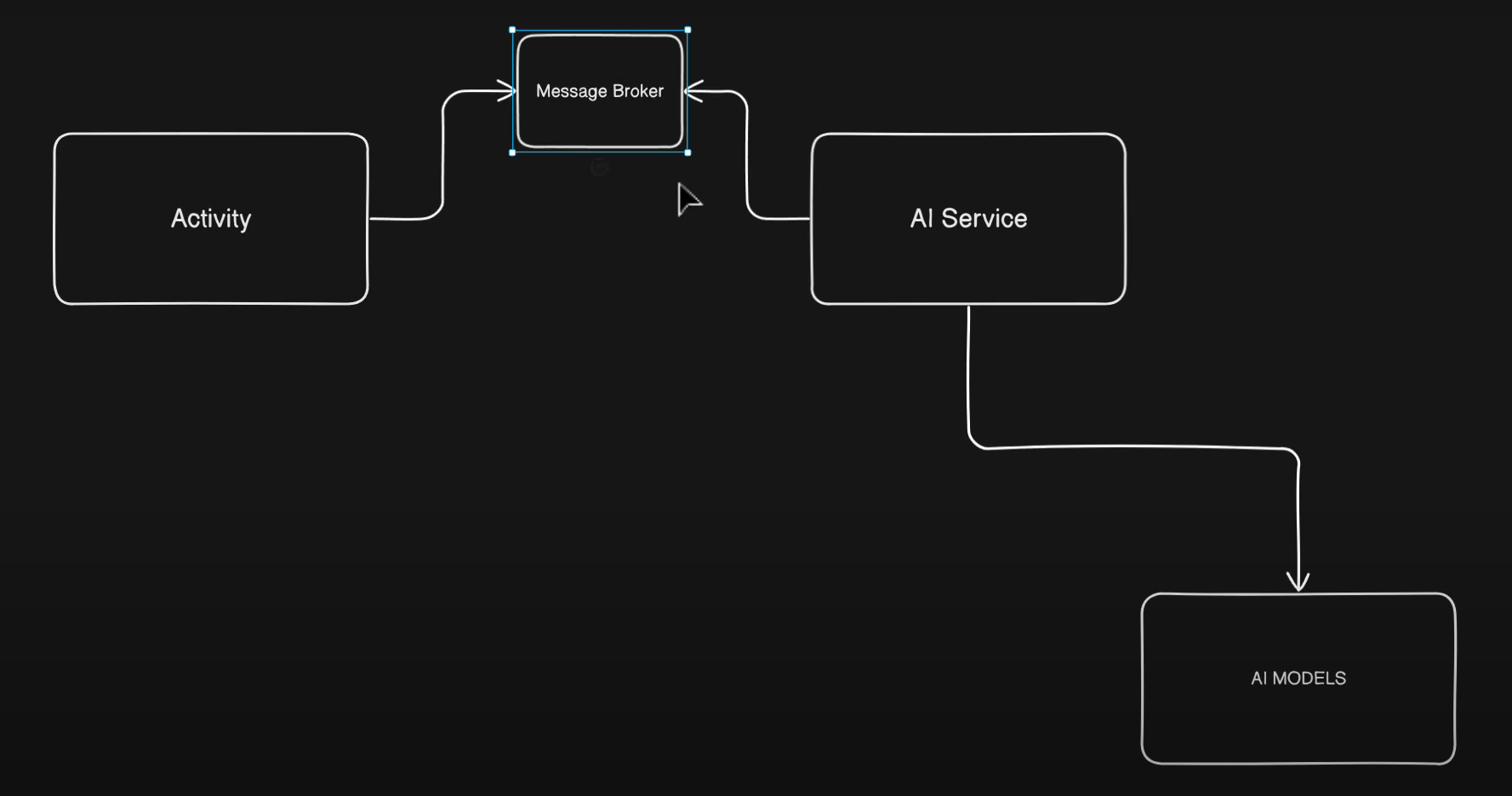
**Apache kafka is running in (detached mode) on Docker:**





Why choose detached mode: so that terminal won’t be blocked, we can run other commands as well.

**Interservice Communication between Activity and AI service using Kafka:**



Unlike Activity service and User service, we did interservice communication using Web client to create activity for specific users.

We will be doing interservice communication between Activity service and AI service using Apache kafka (asynchronously).

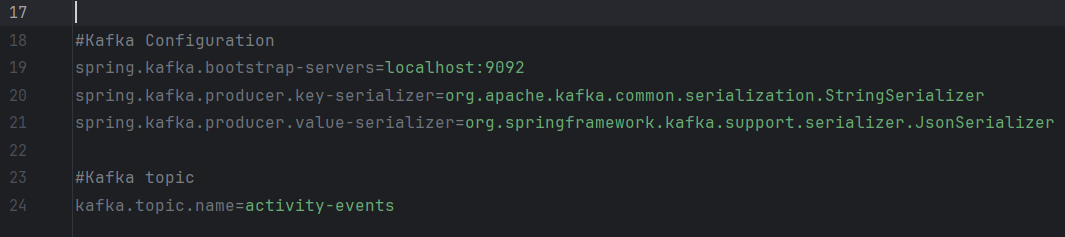
Activity Service will be sending data to AI service using kafka broker

AI Service will be receiving Data from broker sent by Activity Service and will process that.

**Steps:**

**1.Add spring kafka dependency in both (Activity and AI service)**

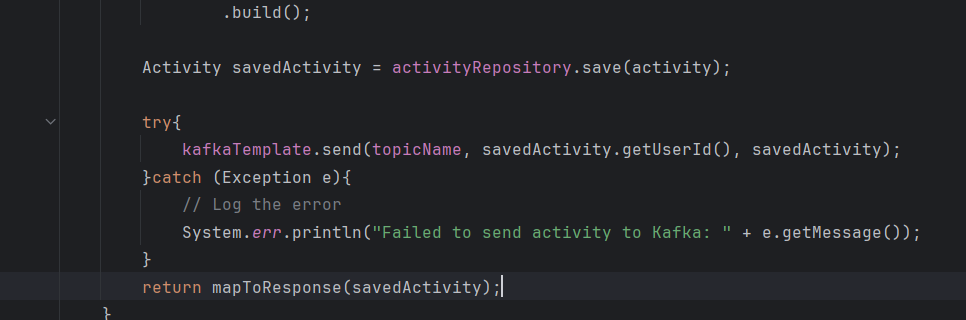
**2.Configure properties file as below**



**3.In Impl class, where we are creating an activity, before saving activity we are sending (producing) data to kafka.**



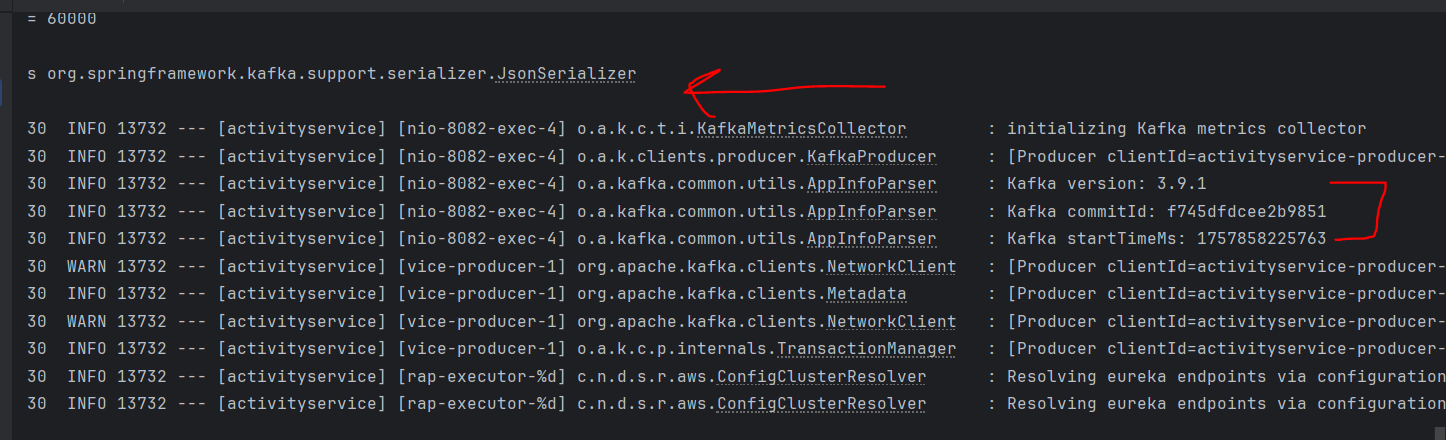
**4.topic name will be injected from properties file by using @Value annotation.**



**5.Run the Activity service and send data in postman (create activity for saved user)**

**6.Console logs:**

**Once connection establishes successfully with kafka, kafka version gets printed on console with commitId, startTime.**



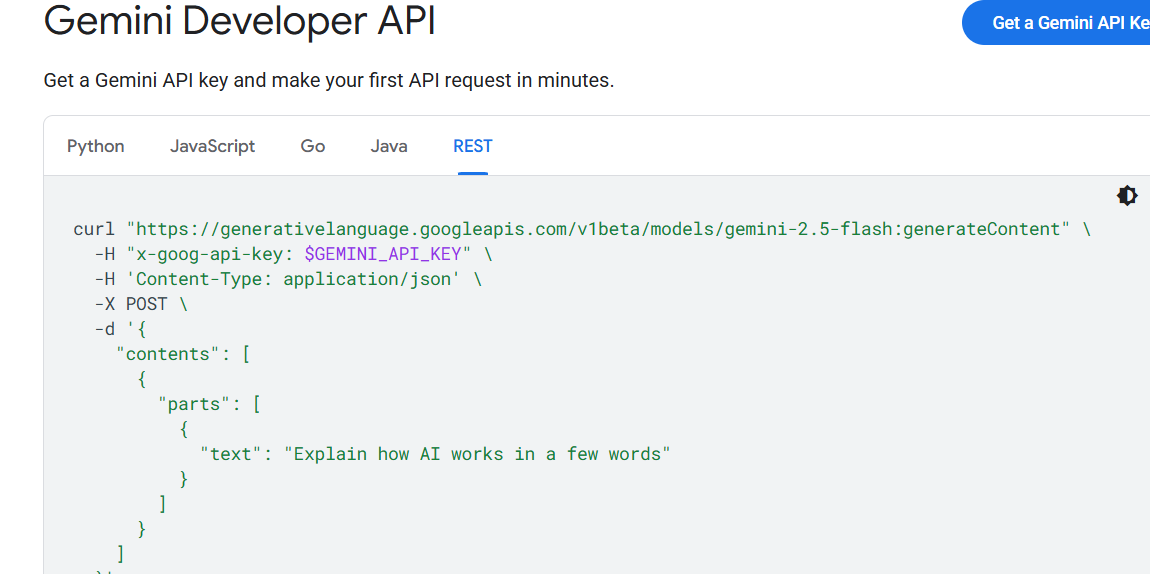
**Configure AI Service:**

**Google Gemini API:**

Go to <https://ai.google/build/>

Build >> Gemini >> Build with Gemini >>Documentation

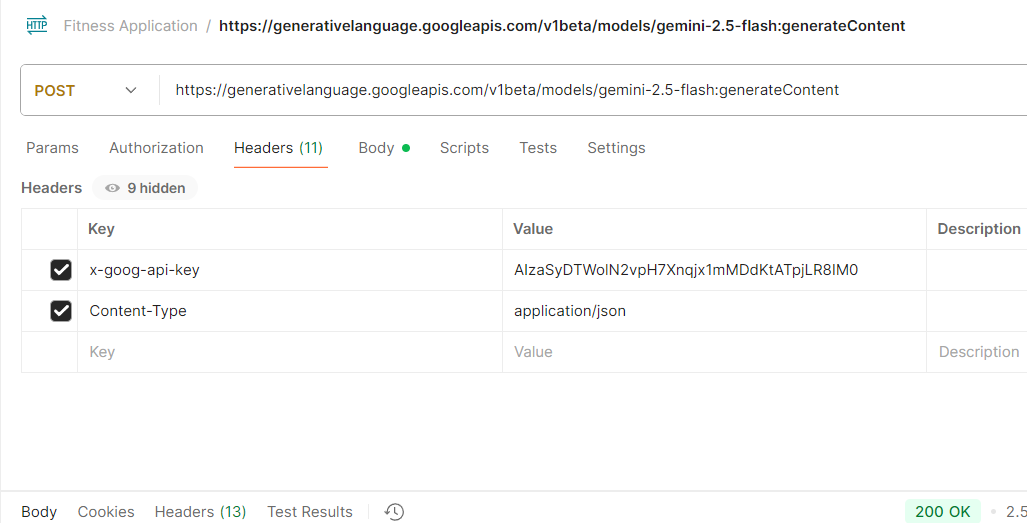
Copy curl command of REST



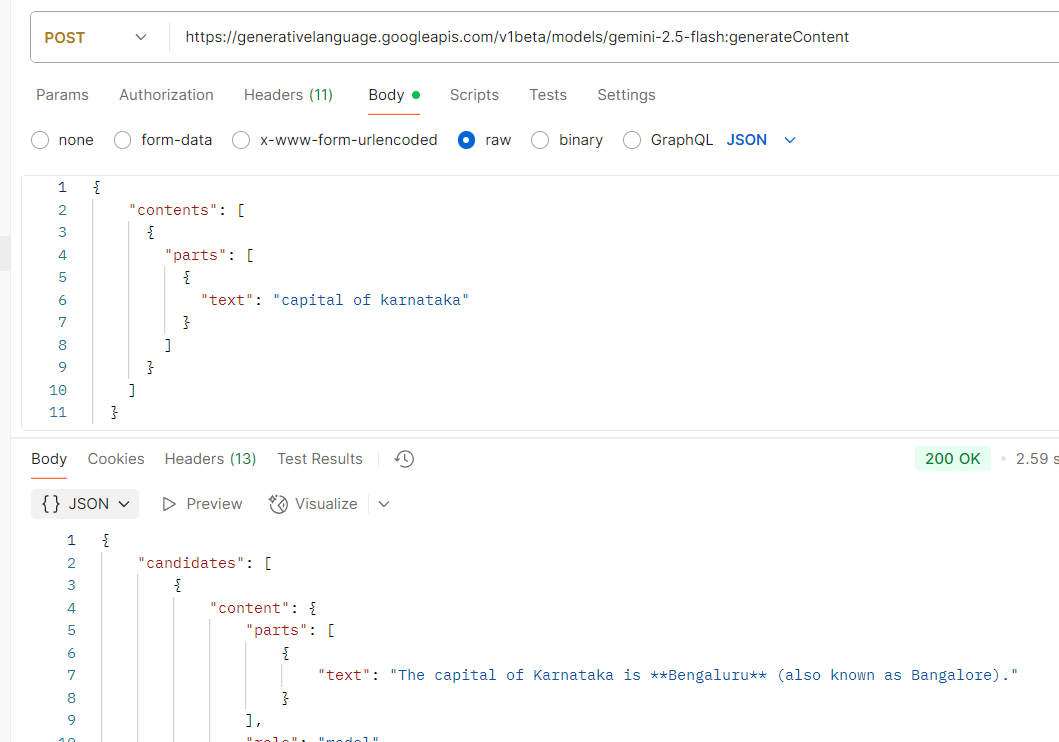
Create (API key) and copy and paste the same on postman requests.

**API Key: AIzaSyDTWolN2vpH7Xnqjx1mMDdKtATpjLR8IM0**

Copy curl command and paste it on postman (on new import), it will automatically adjust with url, headers and body.



Now, we can post our query into AI model and get the answers using google Gemini API.



Now, we will using Google gemini api to get interact with AI microservice

It will generate recommendation based on activity.

We will take activity data and give it to gemini, and with the help of gemini we will get AI generated recommendation based on activity and store it to DB which can be shown against the users.

**AI Service:**

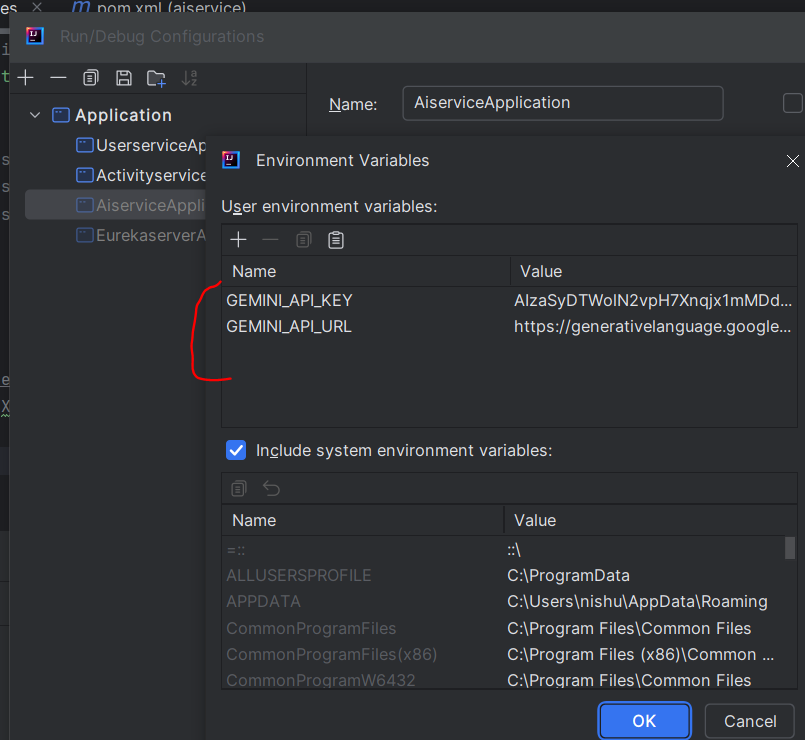
**We will create one service which will be responsible to interact with AI models (gemini api)**

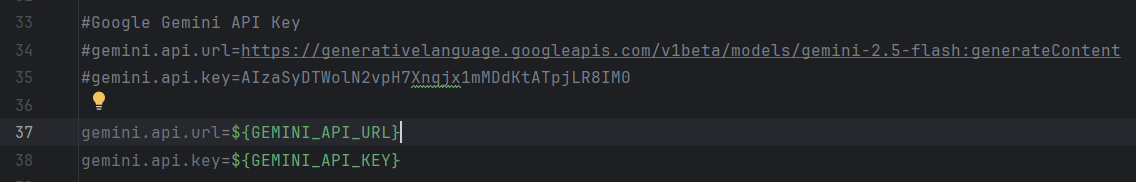
**Class program.**

**Properties file:**

We can inject api key and api url through direct properties file or through environment variable.

So, whenever we run the application api key and url value will be injected from environment variable to properties file (placeholder)





**Add ActivityMessageListener and ActivityAIService program with explanation.**

**Run the application**

**We can see that once we create an activity, AI service gets called and it interacts with Google gemini API and give recommendation based on created activity.**

**Screenshot from console…(to put)**

**Config Server:**

Config server will be responsible to fetch all configuration present in config server.

Currently all the microservices have its own configuration available in properties file and using based on the requirements.

Now, we will put all the configuration of all services at centralized location (config server) and all services will fetch configuration directly from server.

**Example:** Suppose we have AI service which has its own configuration in properties file.

Now, once we start AI service, it will communicate with config server and will fetch their own configuration from config server and apply in the application.

**Steps to create config server:**

1.go to spring initializer

2.create config server project.

3.Add dependency (Spring Cloud Config Server)

4.generate zip and import on IDE.

5.our config server application is ready.

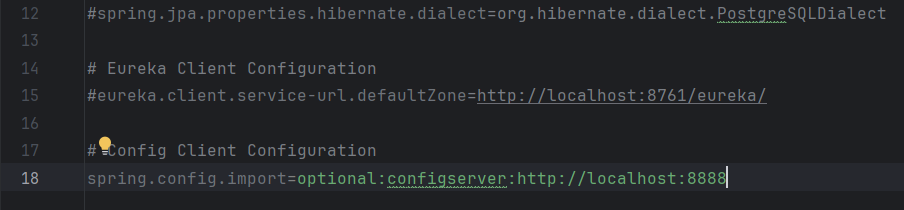
**Communicate between userservice and config server:**

1.Now, we have to make sure user service and all other microservices communicate with config server and fetch the configuration directly.

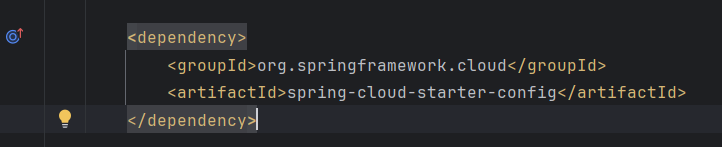
2.Copy all the configuration code from userservice (properties file) and paste it into user-service.properties file available in config folder of config server.

3.delete all the configuration available in user microservice service except application name.

4.We will include config client configuration in user microservice (with optional)



5. Add dependency in user microservice (config client) so that it will communicate directly with config server.



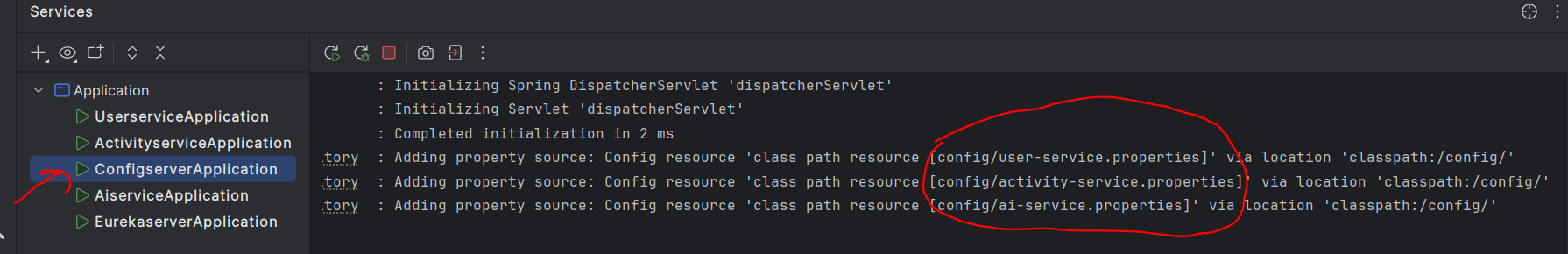
6.Now, restart both the application (config server) and (user microservice) as we made few changes.

7.We changed port for config server to 8888 and added dependency in pom for user microservice.

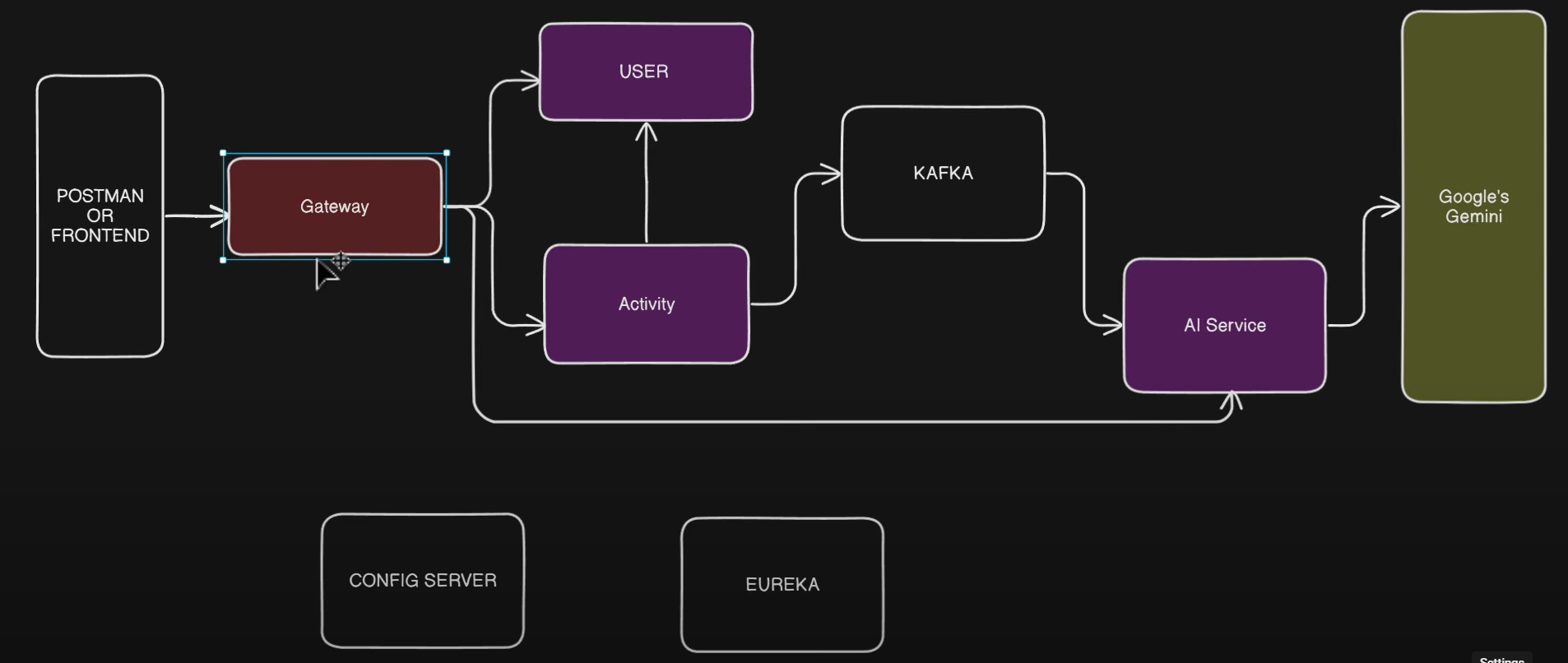
8.once we restart the application, now we can see that configuration will be fetched from config server not from **usermicroservice** properties file.

9.We did the same for **Activity Service** and **AI Service** and started Config server and all other microservices

10.We could see that all microservices are fetching configuration directly from config server.



**API Gateway:**



**Steps to create API Gateway:**

1.create a spring boot project from spring initializer.

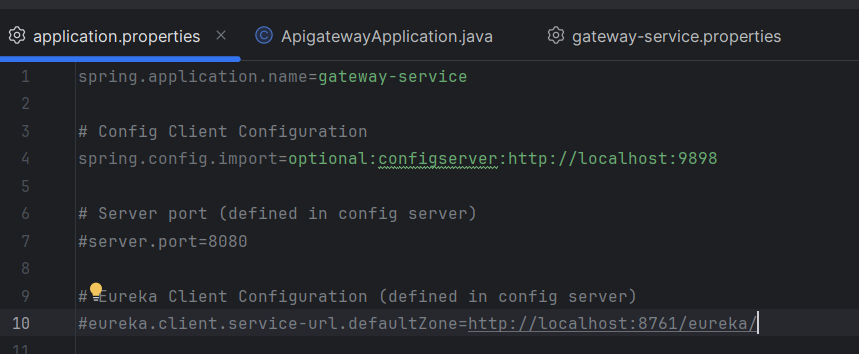
2.Add 3 dependency (gateway, config client, eureka discovery)

3.generate zip file and open in IDE.

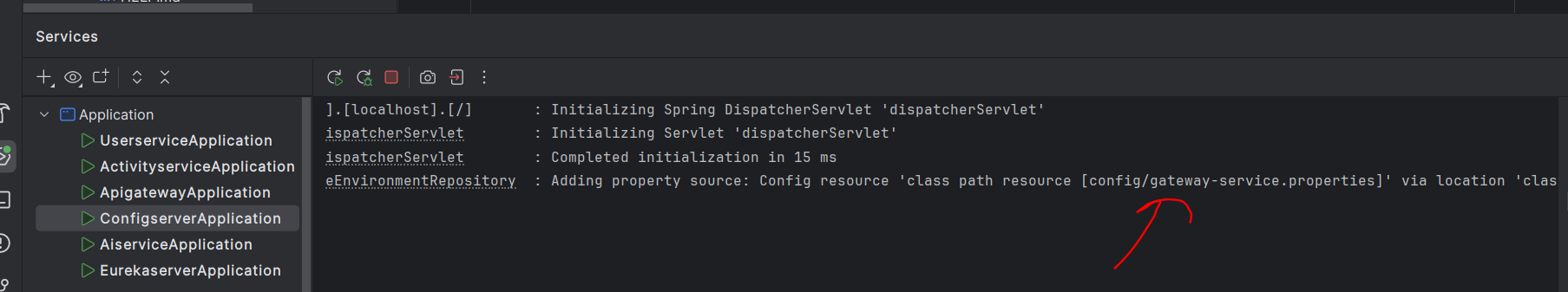
4.configured properties file of API gateway (eureka port and server port) in config folder of config server.

5.create gateway-service in config folder of config server.





6.Now, restart config server and run Gateway service application and we can see that api gateway configuration is fetching from config server.



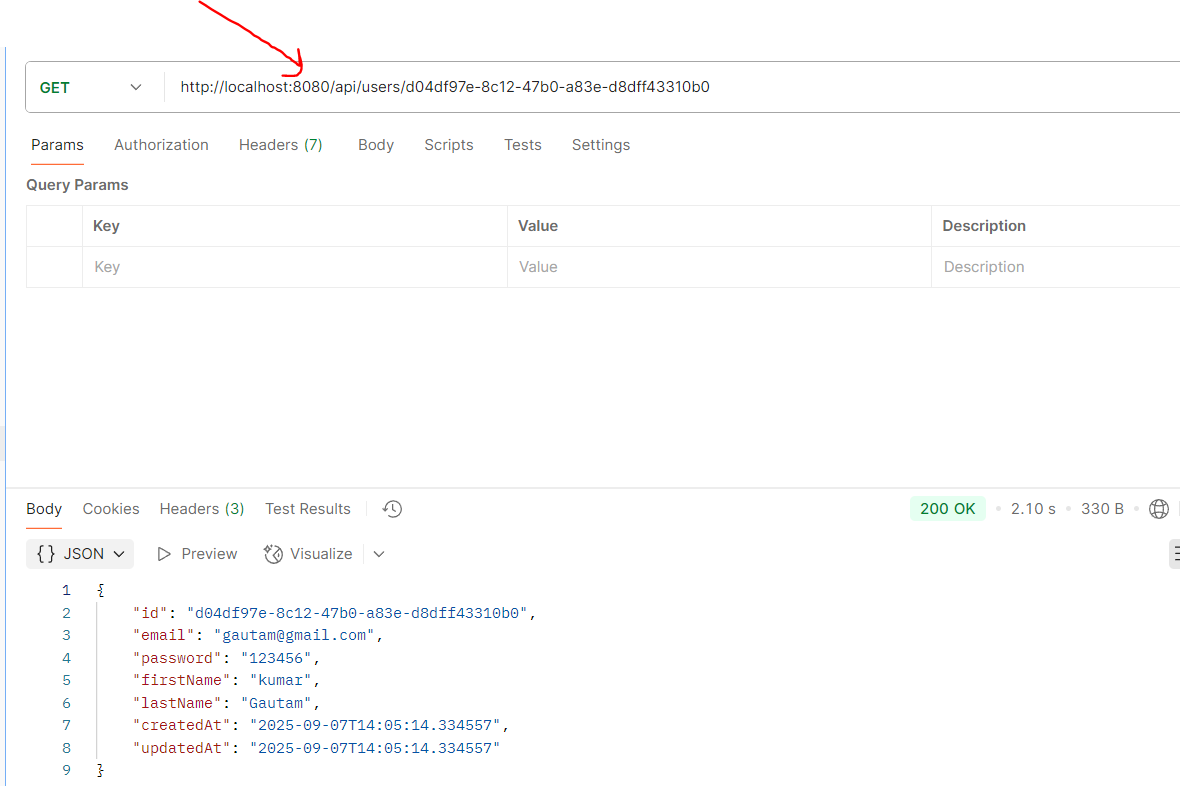
**7.Routing in API Gateway**



**8.Restart config server and run the api gateway application and could see that api gateway is fetching all configuration directly from config server.**

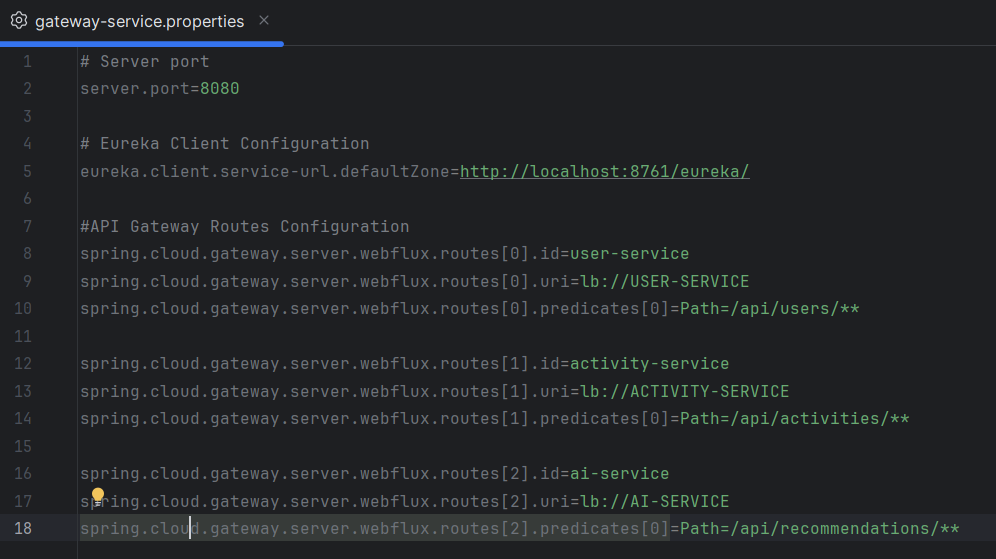
**9.Postman Testing:**

**We are trying to user details through api gateway port (8080) which is routing to User microservice and fetching user details from DB.**



**In this case, Gateway is routing the request to User service and fetching user details from DB.**

**Gateway routing for Activity and AI Service:**



**We can create activity through API gateway which will route from gateway to activity service which will eventually communicate with AI service to give recommendation.**

**To check: why use optional in config server path**

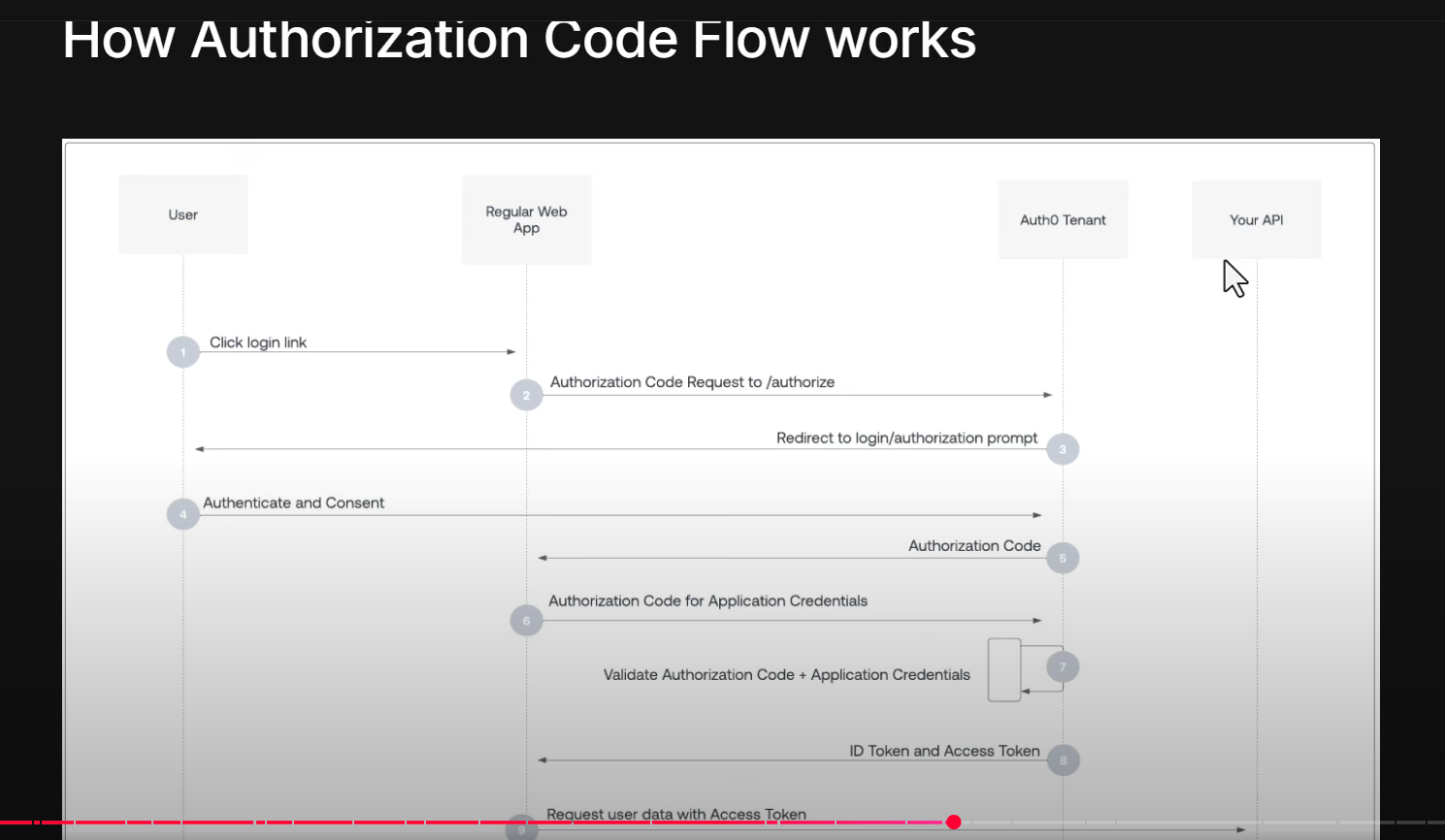
**OAuth2:**

**We will learn why Oauth2 was invented.**

**Problem:**

* You had to share your credentials all the time to the client so that they can access the api, application etc.
* Security risk (misuse of credentials)
* Limited control (once credentials shared, no control over it)
* Inconvenience

**What is OAuth2:** OAuth (Open Authorization) is a standard protocol that allows user to grant access third party application access to their information without sharing their passwords.



**KeyCloak:**

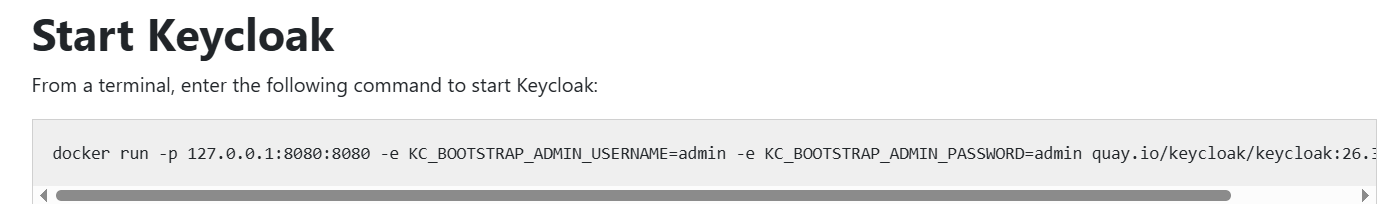
<https://www.keycloak.org/>

<https://www.keycloak.org/getting-started/getting-started-docker>

We will be using keycloak using docker container.

We have docker run command by which we can run keycloak instance on our local machine (we don’t have to download keycloak for this)

**Configure Keycloak in our local machine:**



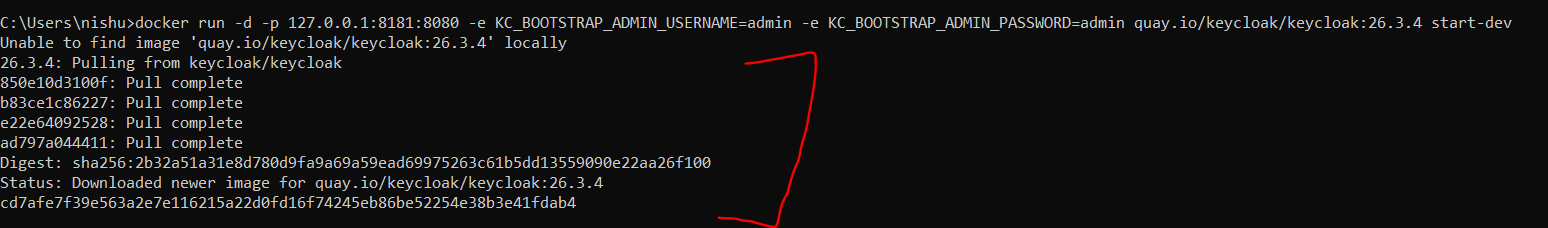
1.Run the above command (change port to 8181) as our API gateway also running on 8080.

2.add (-d) in command which runs in detached mode, so that we can run other command in same window.

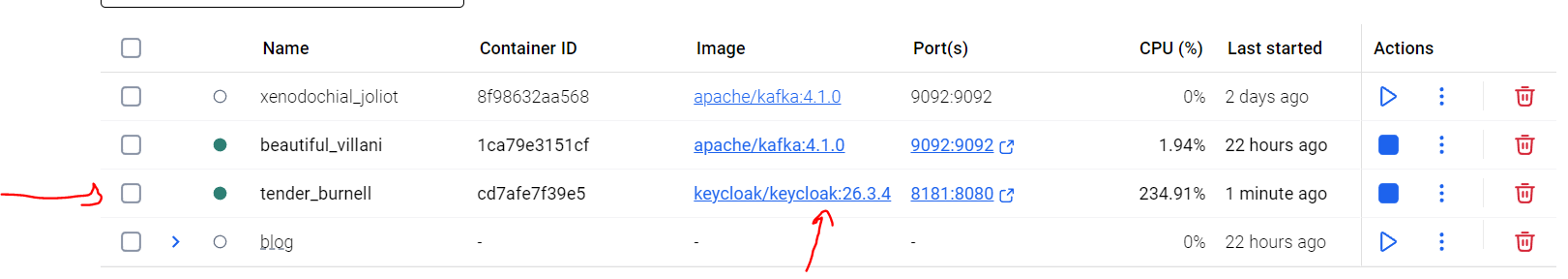
3.**exact command ran on cmd:**

docker run -d -p 127.0.0.1:8181:8080 -e KC\_BOOTSTRAP\_ADMIN\_USERNAME=admin -e KC\_BOOTSTRAP\_ADMIN\_PASSWORD=admin quay.io/keycloak/keycloak:26.3.4 start-dev

4.since we don’t have keycloak image locally, it will pull from docker hub and download.

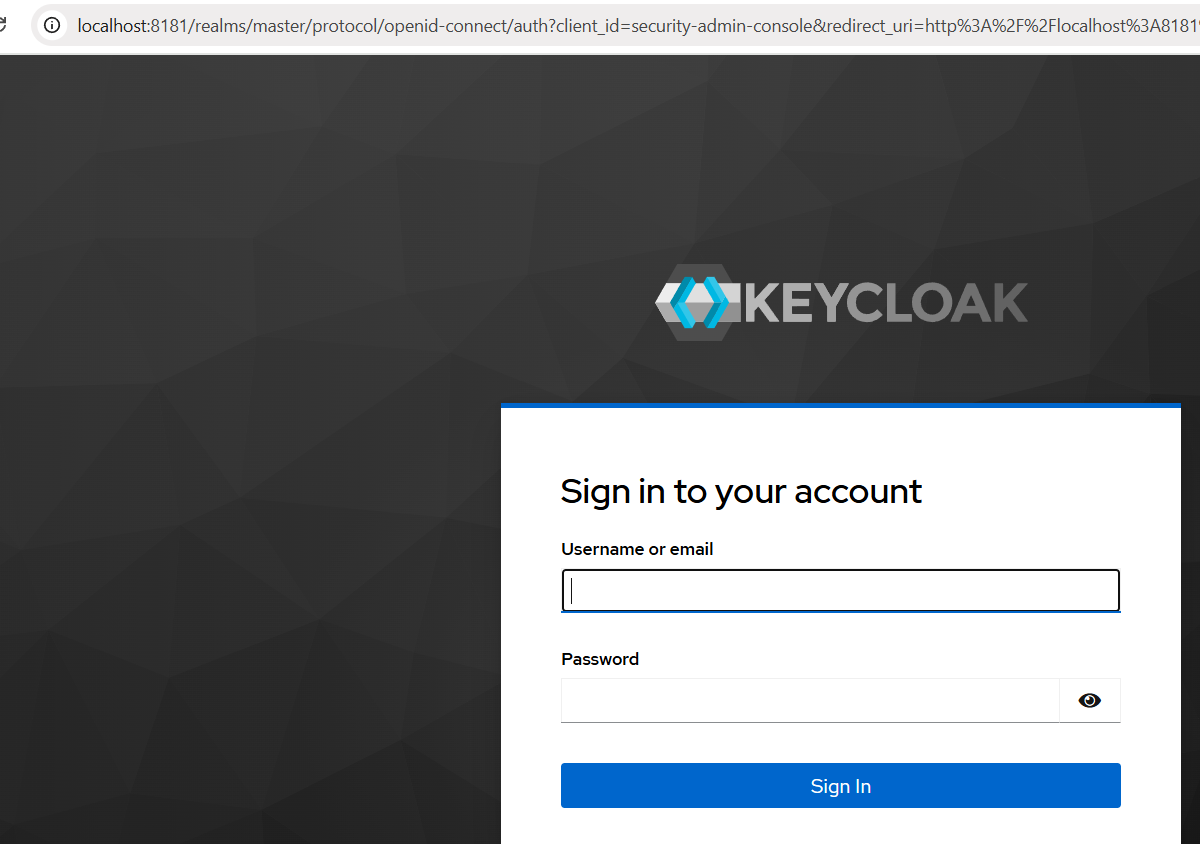


5.Once download and pulled, we can see keycloak image and running container on local docker desktop (with port 8181)



6.Login to Keycloak admin console given in document.

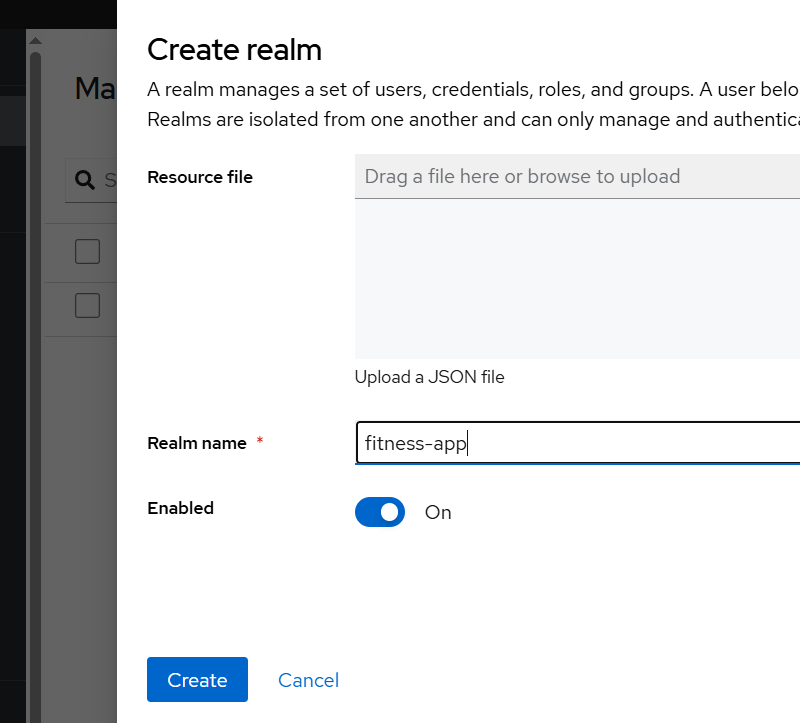
7.login with port 8181 and we will see keycloak login page.



8.Login with username: admin and password: admin mentioned in docs.

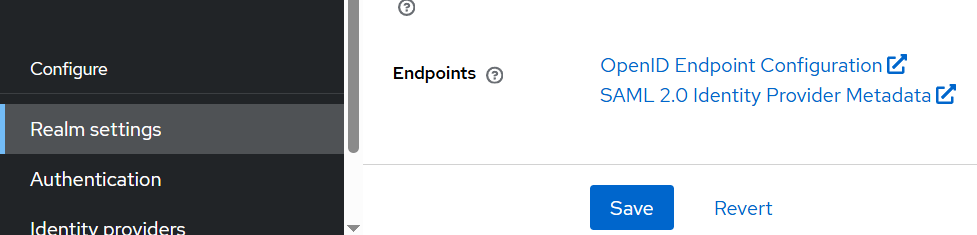
9.Realm (like workspace, we can create as many as realm as per new project)

10. Created a realm for fitness application on keycloak



11.Created client for my project realm (oauth-pkce-fitness-client) and proceed.

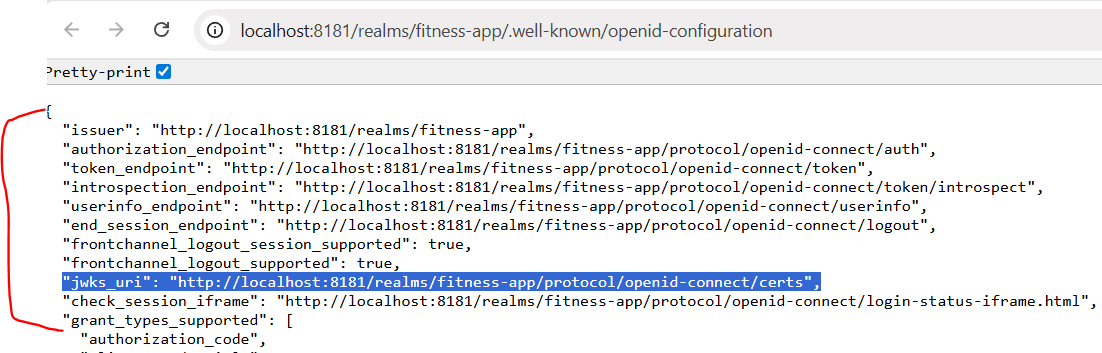
12.Go to Realm settings and click on OpenID endpoint configuration link.



13. We will get lot of be default given endpoints which can be accessible.

**How to implement Security with Keycloak:**

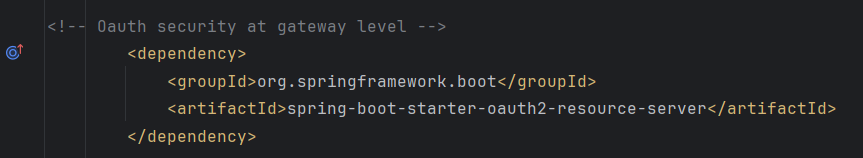
* We will implement security at API gateway level with help of keycloak which will check if token is available and valid or not
* Once token is valid, then only api gateway will process the request further to other microservices internally.
* "jwks\_uri": <http://localhost:8181/realms/fitness-app/protocol/openid-connect/certs>



* We will configure api gateway for this url (jwks uri)
* Keycloak will generate the token for each request and API gateway will validate that token.
* To validate the token, API gateway need access to public key.
* API gateway will fetch public key and validate the token (if tampered or not).

**Steps:**

1.Add dependency in Gateway pom.xml (**OAuth2 Resource Server**)



2.API gateway (resource server) will verify access token which was issued by Authorization server (Keycloak).  
3. We will configure spring security for Oauth in gateway properties file available in config folder of configserver.



4. go to <http://localhost:8181/realms/fitness-app/.well-known/openid-configuration> and copy jwks uri and put in properties file.

5.Define security config in api gateway.



6.Start all the application and test through postman.

**Testing Postman:**

1.Now, once we send the request on postman to create activity, we get unauthorized 401 error (because we didn’t provide token) to access apis.

2.create oauth token on postman with the help of keycloak.

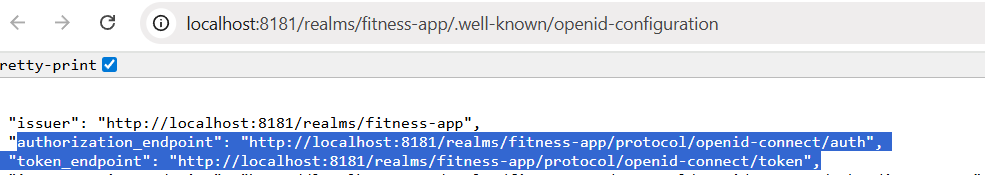
(a) click on Authorization

(b) select auth type “OAuth 2.0”

(c) configure new token

(d) provide token name, grant type (pkce), callback url

(e) provide auth url and access token url



(f) provide code challenge method (sh256)

(g) click on get new access token option

3. You will be prompted to new window (keycloak window)

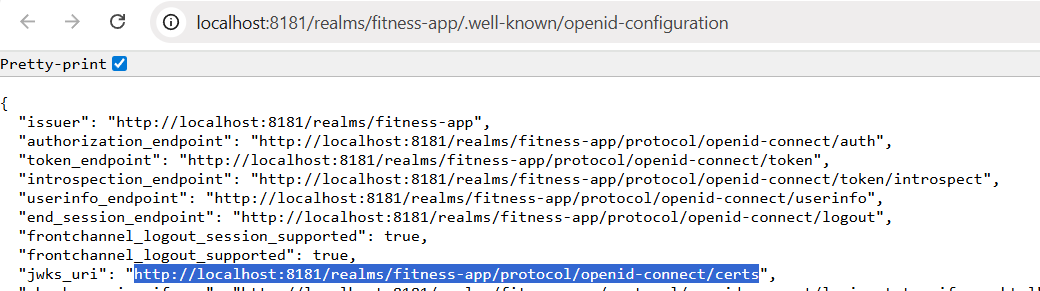
4.provide user and password created on keycloak

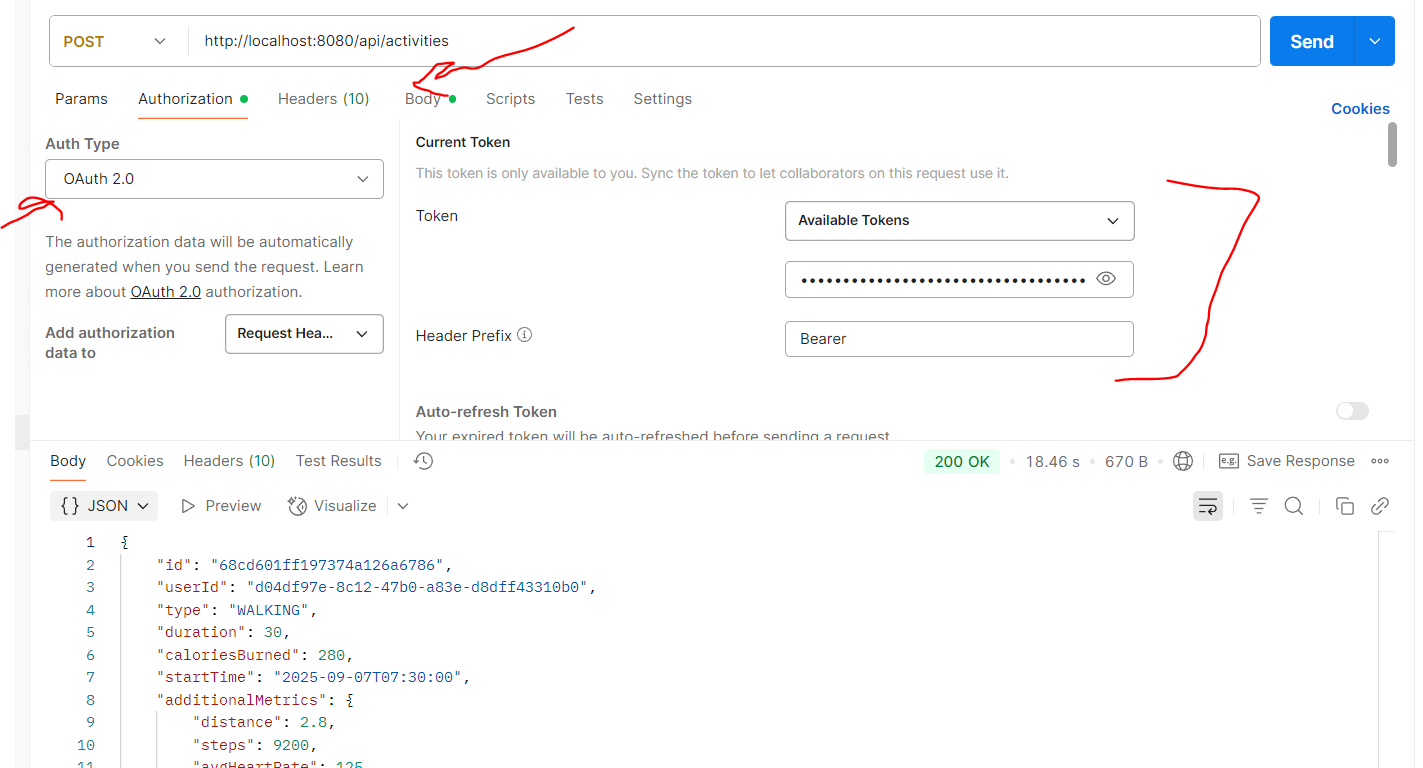
5.You will be authorized and will get new token (current token)

6.Use the same token and hit api on postman. We will be able to access successfully.

7.Hence, keycloak generated new token and API gateway validated the same if valid or tampered or expired

8.API gateway use this jwks url <http://localhost:8181/realms/fitness-app/protocol/openid-connect/certs> which is mentioned in properties file by taking public key to validate the token





**Syncing User Data with Keycloak:**

We will extract the token and decode it.

We will extract all information from the token generated by keycloak

We will send request from Gateway to register the user if user is not existing already in DB (and in user service)

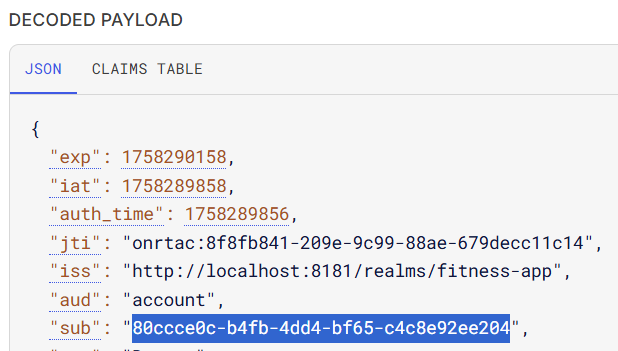
Conclusion:

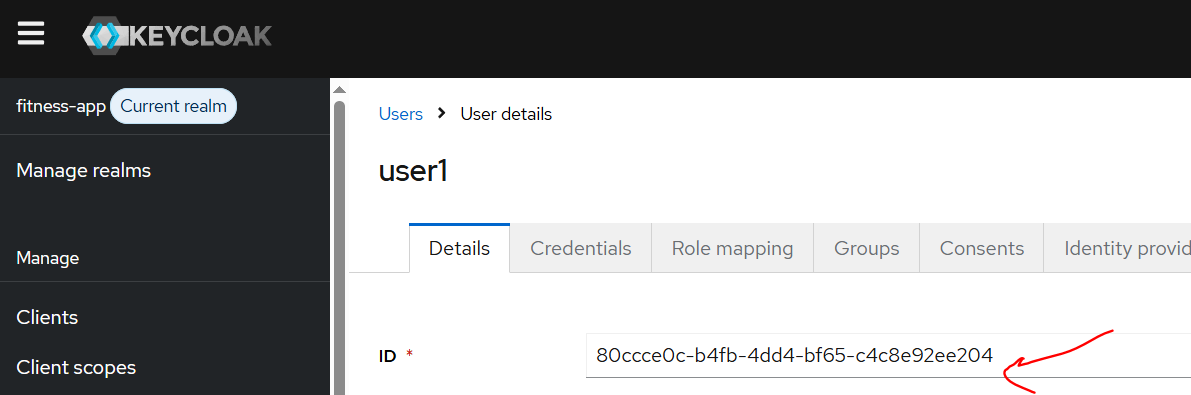
We will write the logic/code in API gateway which will intercept all the incoming requests from keycloak while creating user on keycloak , will take out the token, decode it and will fetch all the information from token and it will validate if that user exists or not in user service.

If user does not exist in user db, create that user parallelly with keycloak and if it exists forward the request.

Once we extract all data from keycloak generated token, we will set “sub” part which is nothing but keycloakID.

Same keycloakId we can see for specific user on keycloak admin console.





We can write the logic in API gateway to intercept all the requests coming from keycloak and will check if creating user is available or not. IF not, it will create the user and forward the request.

We will use the concept of “WebFilter” interface which has filter method and return Mono<Void>.

**Video timing: 7:30 to 8:15, due to less time and more project, we skipped this part of implementation.**