**Spring Cloud Configuration**

-Shubham Ashtaputre

**A] What is Spring Cloud Config?**

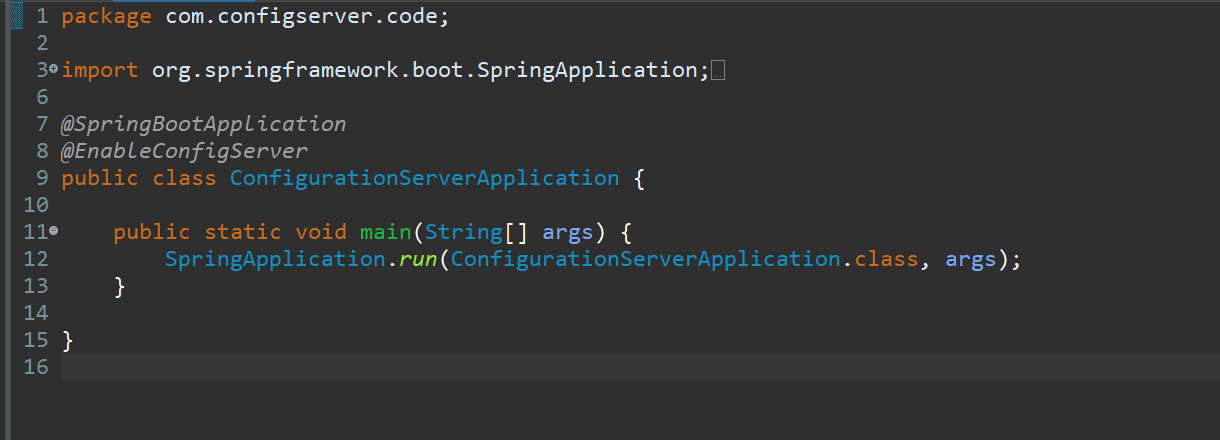
Spring Cloud Config is a framework within the Spring ecosystem that provides support for externalized configuration in distributed systems. It allows you to manage and store configuration data for multiple microservices or applications in a centralized location, making it easier to maintain and modify configurations.

With Spring Cloud Config, you can store configuration files in various backends, such as Git, SVN, and local file system. You can also use different file formats, including YAML, JSON, and properties files. Spring Cloud Config provides a RESTful API for accessing configuration properties and also supports dynamic refresh of configuration changes without restarting the microservices or applications.

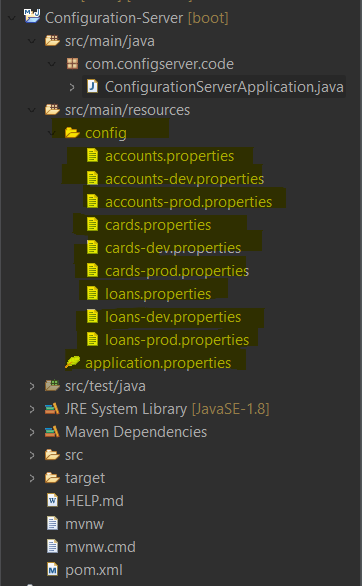
By using Spring Cloud Config, you can avoid hard-coding configuration values in your code and reduce the risk of configuration errors. It also enables you to manage configuration changes in a version control system, track configuration history, and perform audits. Overall, Spring Cloud Config provides a powerful and flexible way to manage configuration data in distributed systems.

**B] How to setup Spring Cloud Config and read properties value from classpath or any location:**

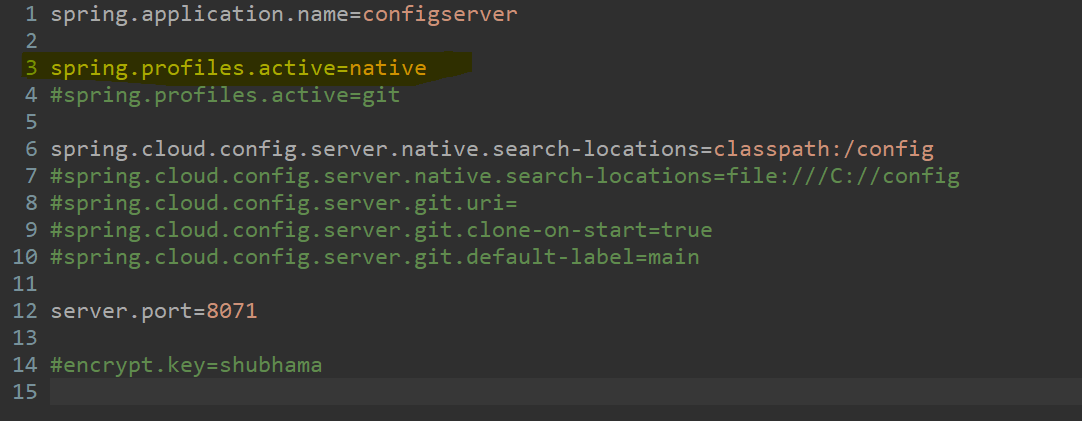
1] add @EnableConfigServer



2] create a config folder in which we will maintain all our properties related to all our microservices



3] inside **application.properties** file:

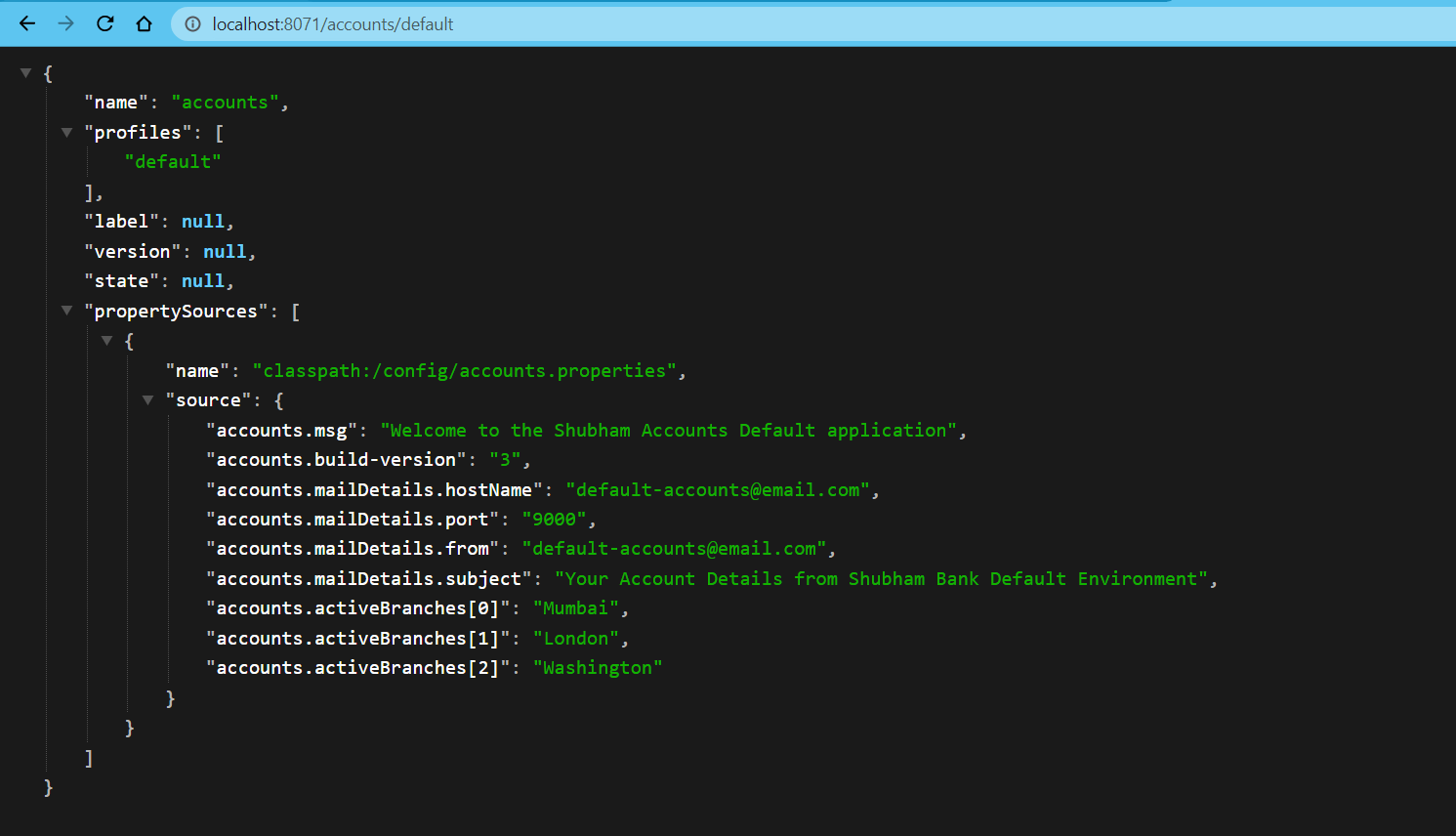


Here, I had set active profile as native i.e. **spring.profiles.active=native.** Now, this **native** means that we are going to use the configuration present either from the classpath or the properties file

And, **spring.cloud.config.server.native.search-locations=classpath:/config** means that my configurations files are present inside location **classpath:/config** and I’am telling spring cloud to search properties from this location

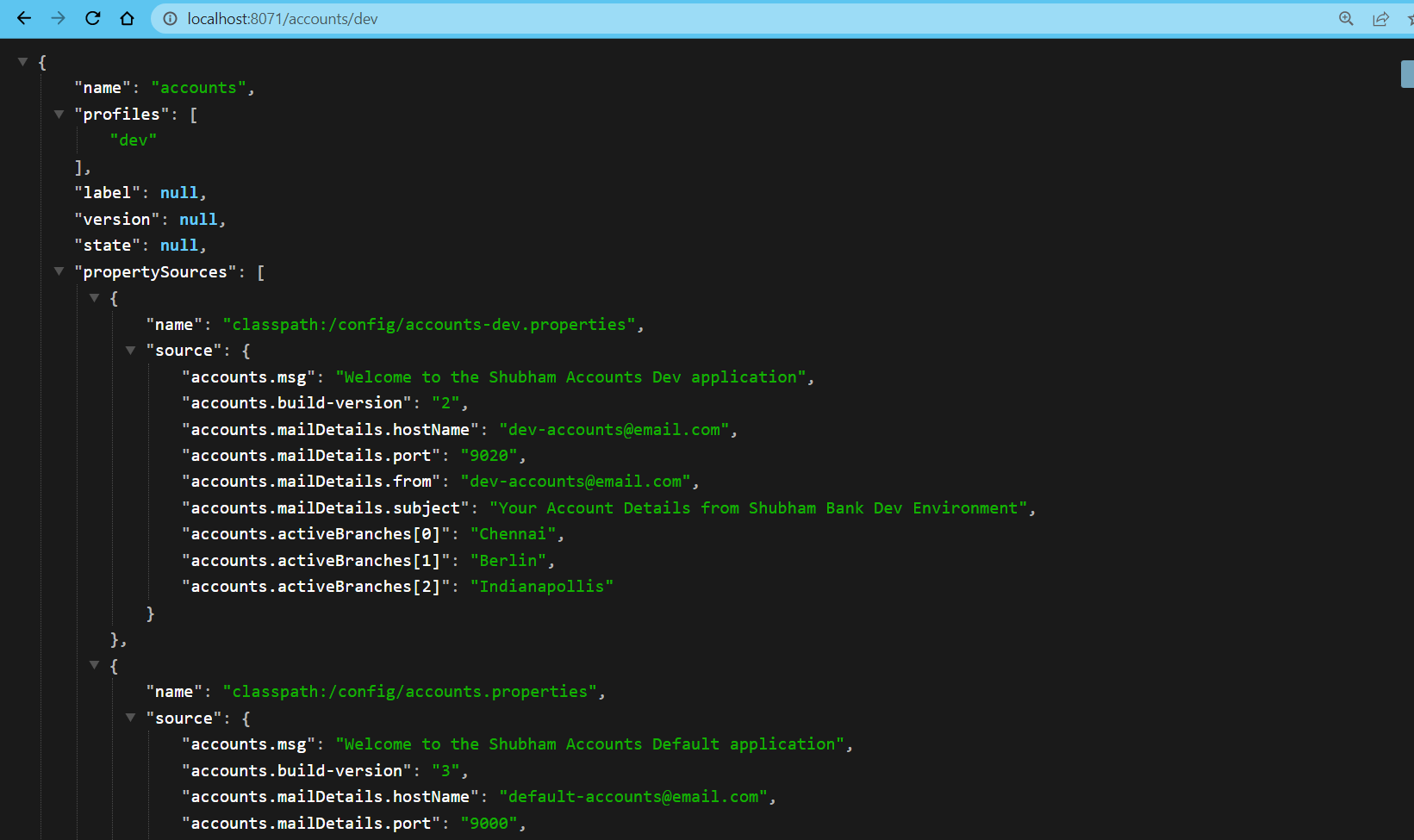
4] now to access this properties values start the application and access the url as below:

[**http://localhost:8071/accounts/default**](http://localhost:8071/accounts/default)

****

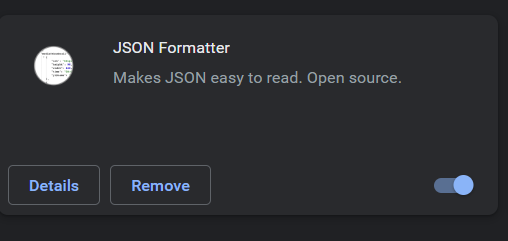
To access dev environment:

**http://localhost:8071/accounts/dev**



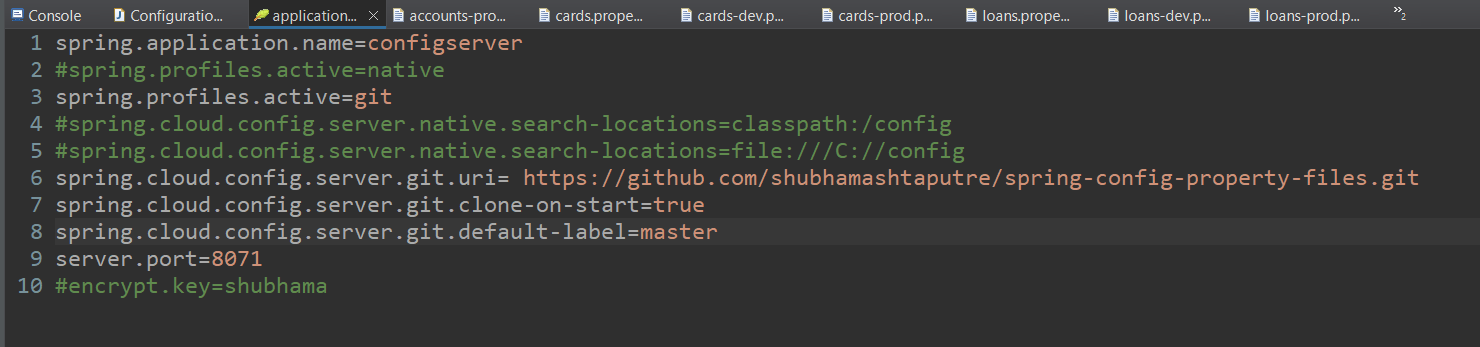
Note:

Here, I’am using the extension JSON formatter extension to view JSON data



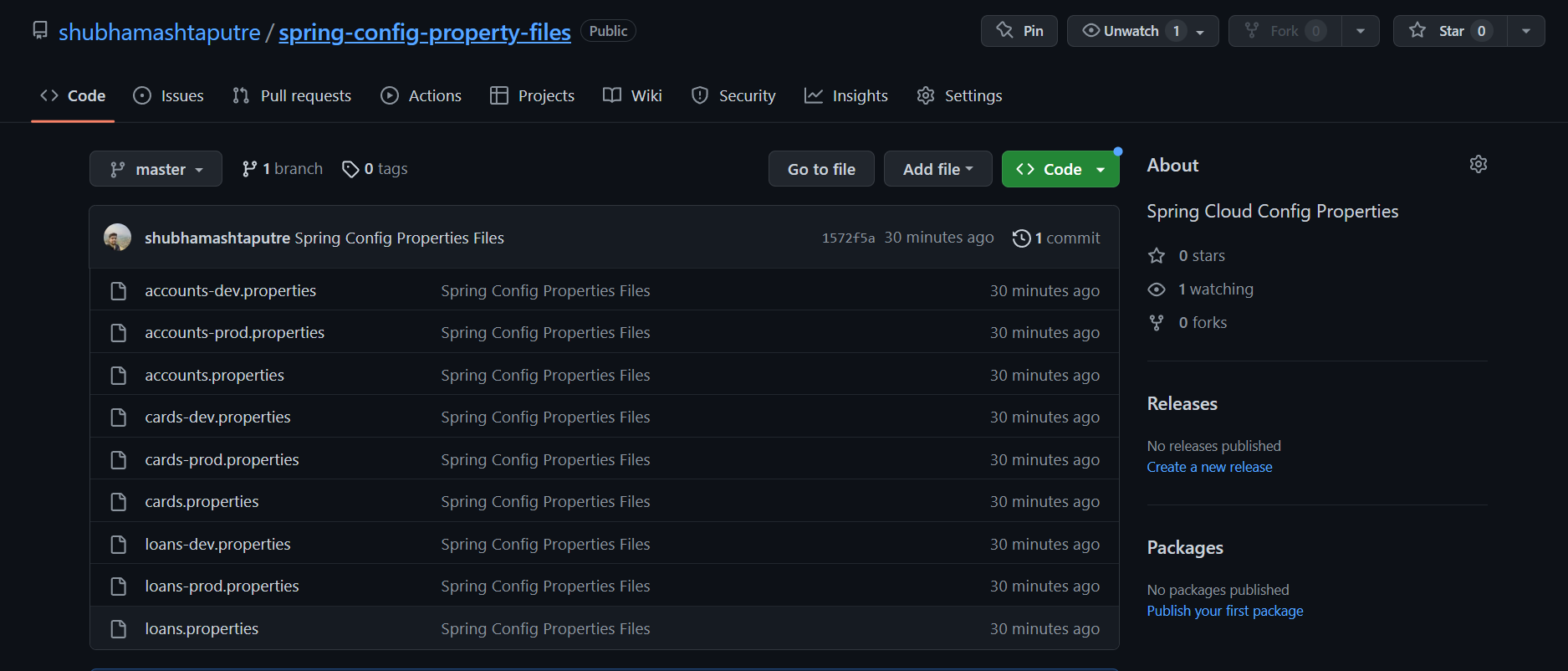
**C] How to read the properties values from the GIT location:**

C1] make changes inside the application.properties file as below:



**spring.cloud.config.server.git.uri** is our github location in which I’am maintaining my all property files, **spring.cloud.config.server.git.default-label** is the branch name in which my property files are present

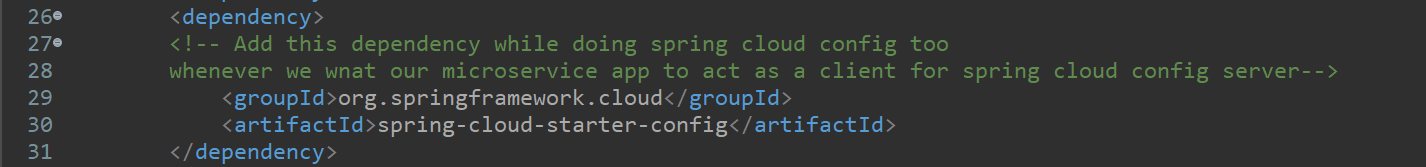
C2] Git repository:

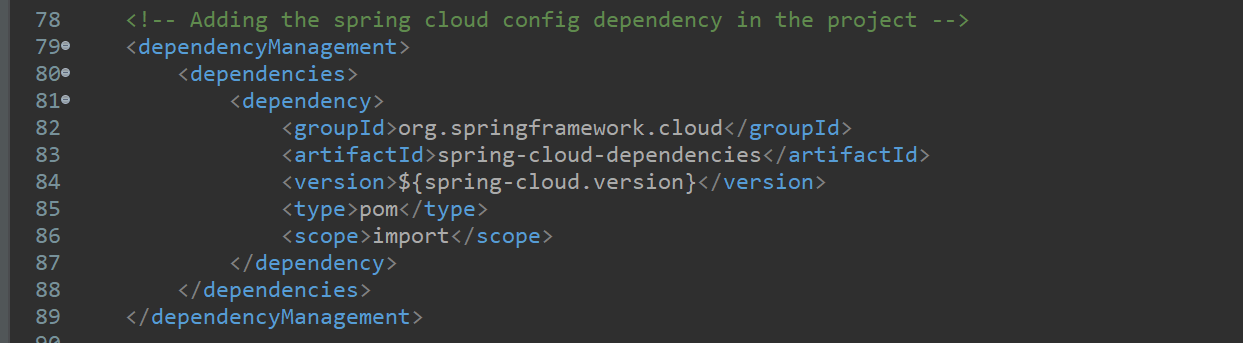


**D] Configuring application to read properties from Config server**

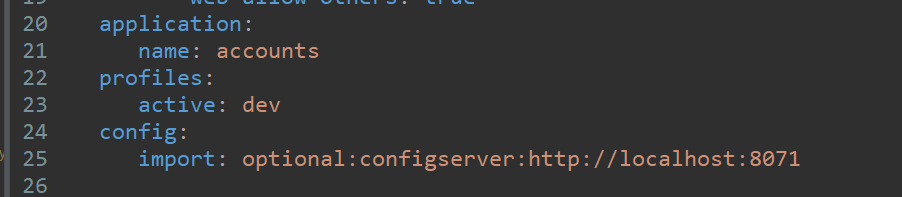
In this case I’am making the Account microservice to read properties values from the config server accounts properties files, below are the steps to achieve it.

1] Add the below dependency in account pom.xml file:





2] Inside the **application.yml** file for Account microservice application add below changes

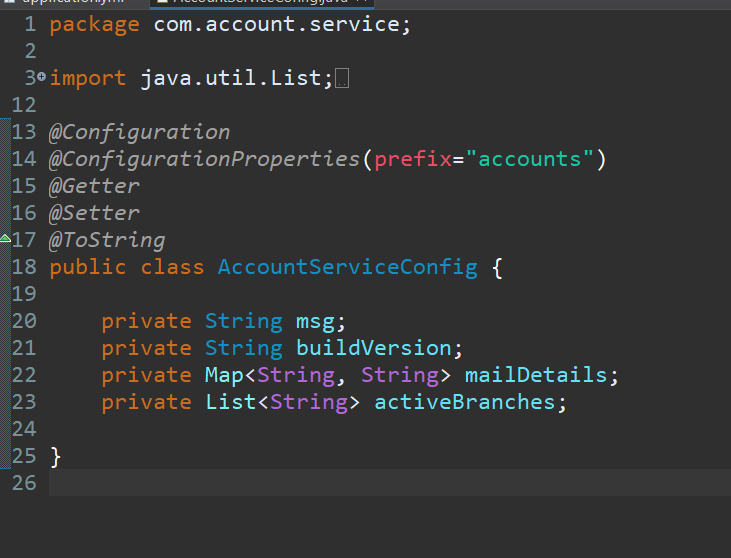


2.1] Here, accounts is the name of the properties files present inside the Configuration-Server application whose properties I want to read

2.2] Here, **active: dev** means I want to read properties of dev environment of this accounts properties file

2.3] Now this inside import is the address from where I will get mr required properties files present at remote location

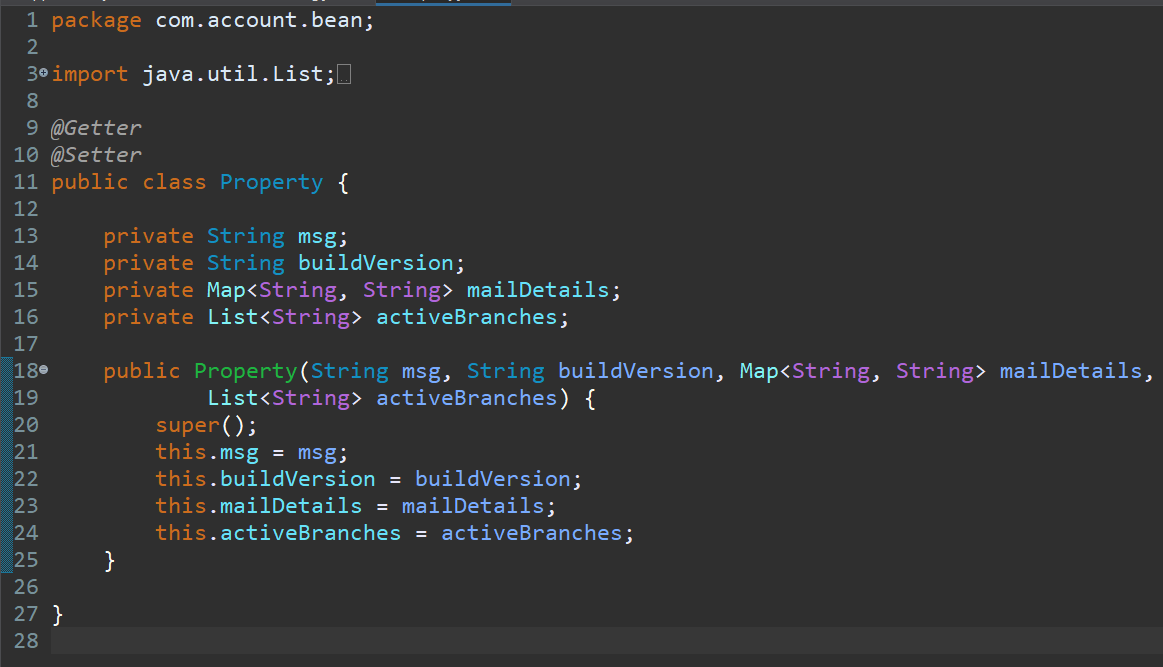
3] Create a ‘**AccountServiceConfig**’ service file:



Here, if you see I had used annotation as **@ConfigurationProperties(prefix="accounts")**

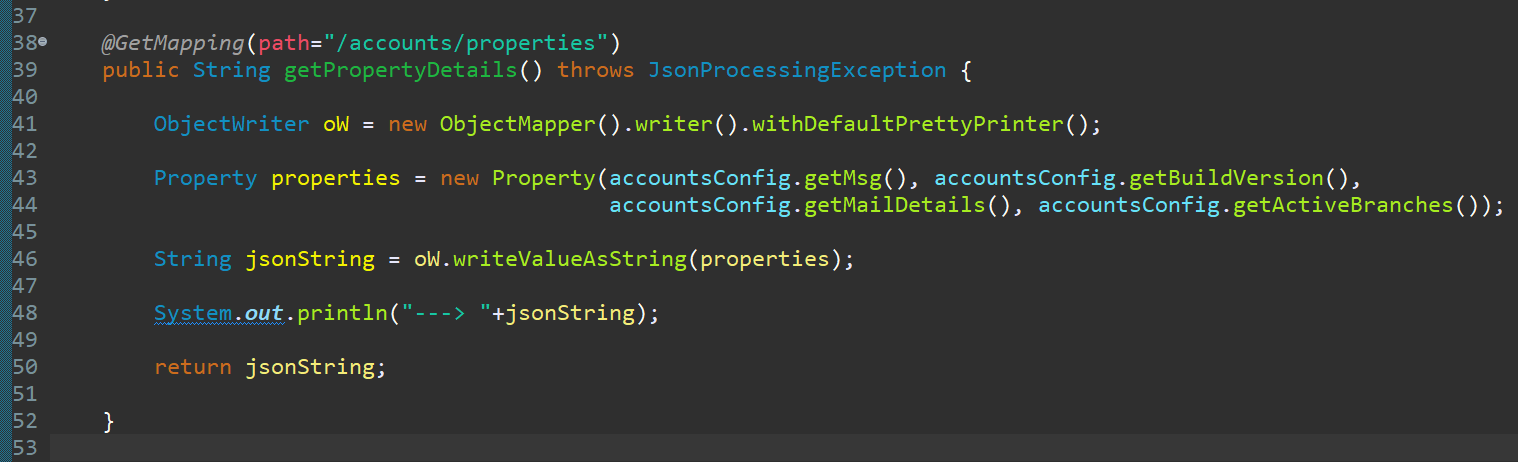
Which describes as, **@ConfigurationProperties** is an annotation in the Spring Framework that is used to bind external configuration properties to a Java bean. It is commonly used in Spring Boot applications to externalize the configuration and separate it from the code.

4] Create Property POJO file:



In this POJO class we will pass the properties file values from **AccountServiceConfig** using getters and setters

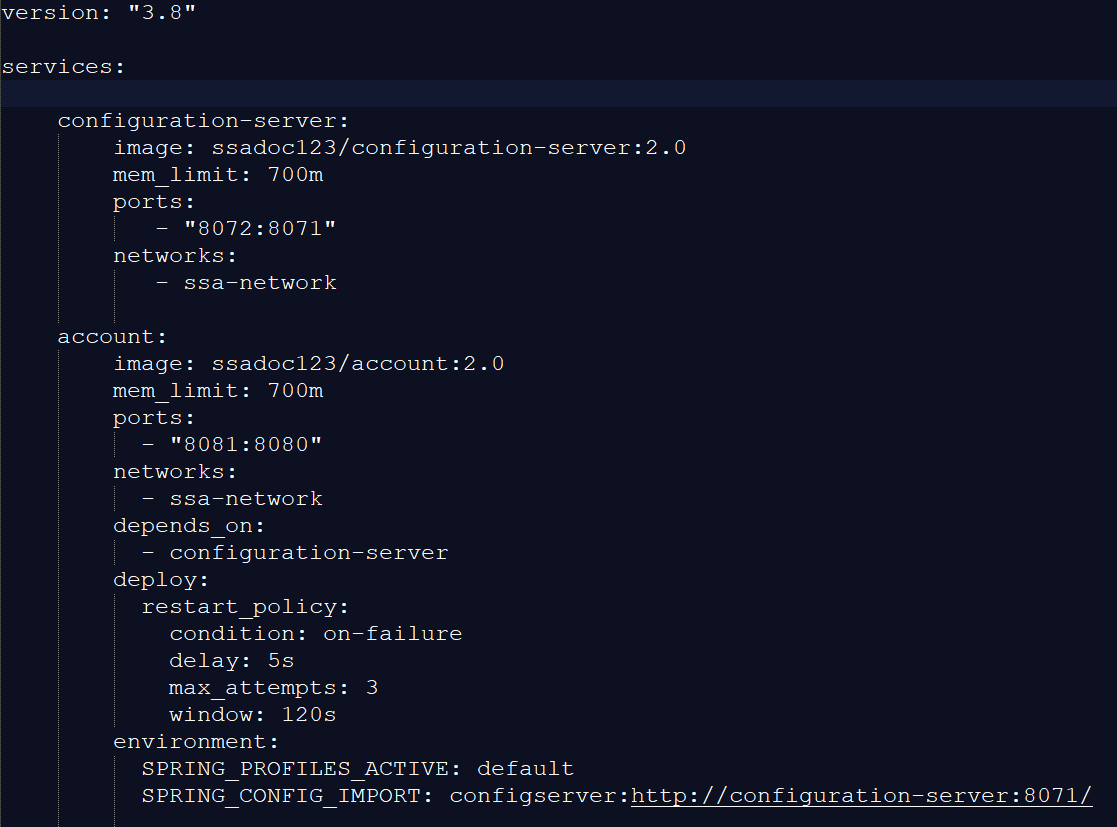
5] In controller class create the getter method as:

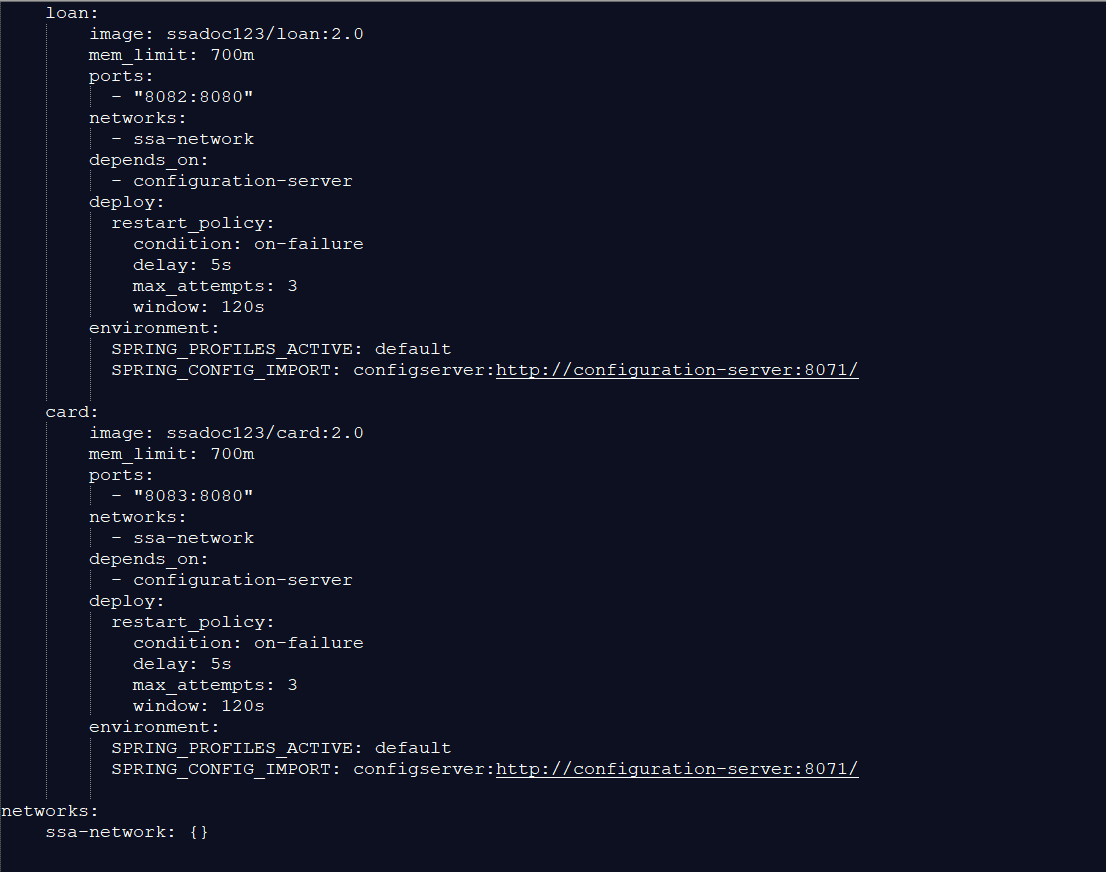


**E] Spring Cloud Configuration set the properties dynamically in the Docker-Compose file**

1] Now, suppose you have deployed your microservice application into docker environment and after sometime you realized you need to change the environment from dev to prod for a articular microservice, at that time you again need to make changes into **application.yml** file of that microservice and again deploy it’s image into docker

2] Now, to solve this issue what you can do is create a docker compose file and set the properties file values for that microservice from there itself and avoid further complications as below:





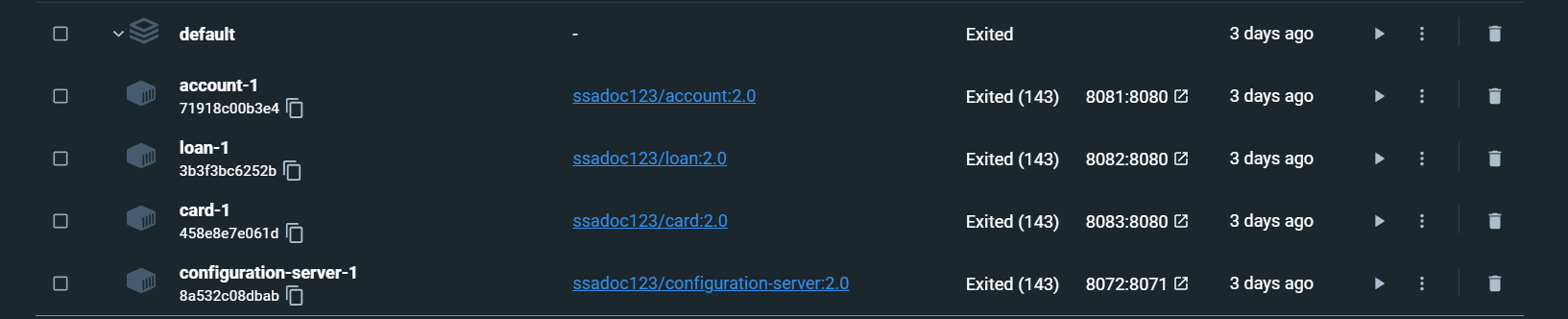
3] Now, if you see in this docker-compose file what I have done is I had deployed 4 microservices including spring cloud config server too from which my rest 3 microservices will read their environment properties

4] Here, if you see in account, loan and card microservice its says that these apps depends on **configuration-server** image so unless this **configuration-server** container is not running the rest of these 3 microservice will not start, and further I had also set the **restart\_policy** which will restart my 3 microservices the number 3 times as defined, if my **configuration-server** has not started in first 2 attempts else even during 3 times these 3 microservices are restarting and still at that time **configuration-server** has not started then my I should get error

5] Further if you see al these 4 microservices comes under same network so that they will start and stop simultaneously, they form a group

6] Further in **environment** I had also set which active environment I want to use for my particular microservice and the **configuration-server** URL too to read the property values

7] After deployment of this docker compose file in docker environment the docker container looks as below:



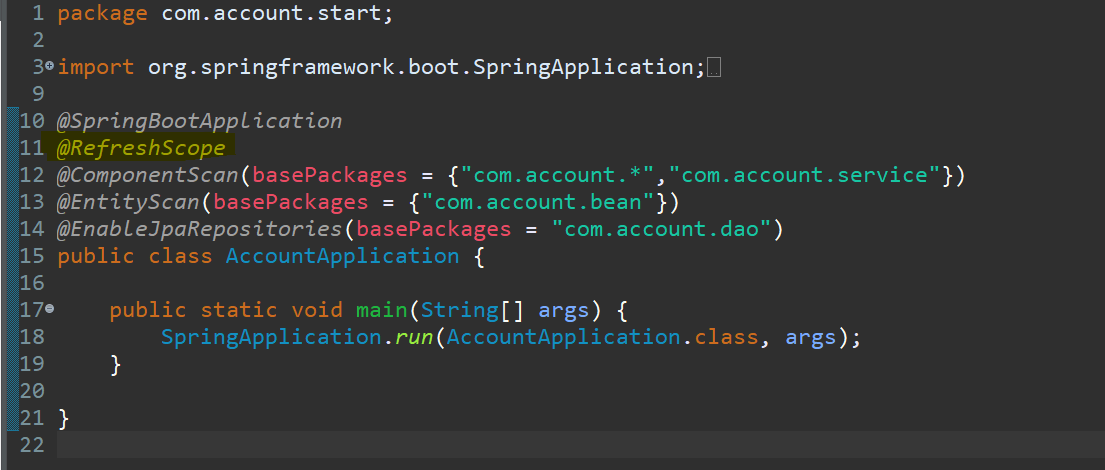
F] Using **@RefreshScope** annotation:

1] Now, suppose my microservices are running into the docker container and suppose I want change property values of some microservices, in that case I have to stop my all microservices and then change the property file values and again restart the microservices,

2] But this thing is not possible in real time as in production environment my live application is hosted and I had to again and again change my property files values and restart the docker server then I would affect the end user

3] So in that case I will simply use the **@RefreshScope** annotation which will refresh the updated property files values in background without need to restart the server

4] We assign this annotation as below in main file:



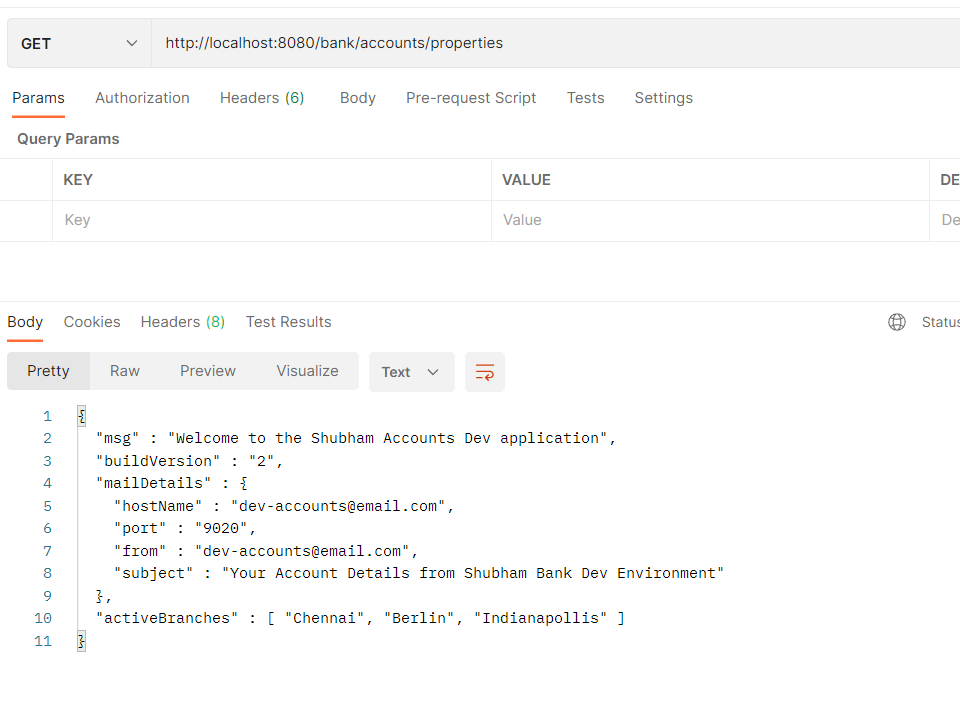
That’s it

5] Follow the below steps to see this annotation usage:

5.1] Start the **ConfigurationServerApplication** application

5.2] For example start the **AccountApplication** application

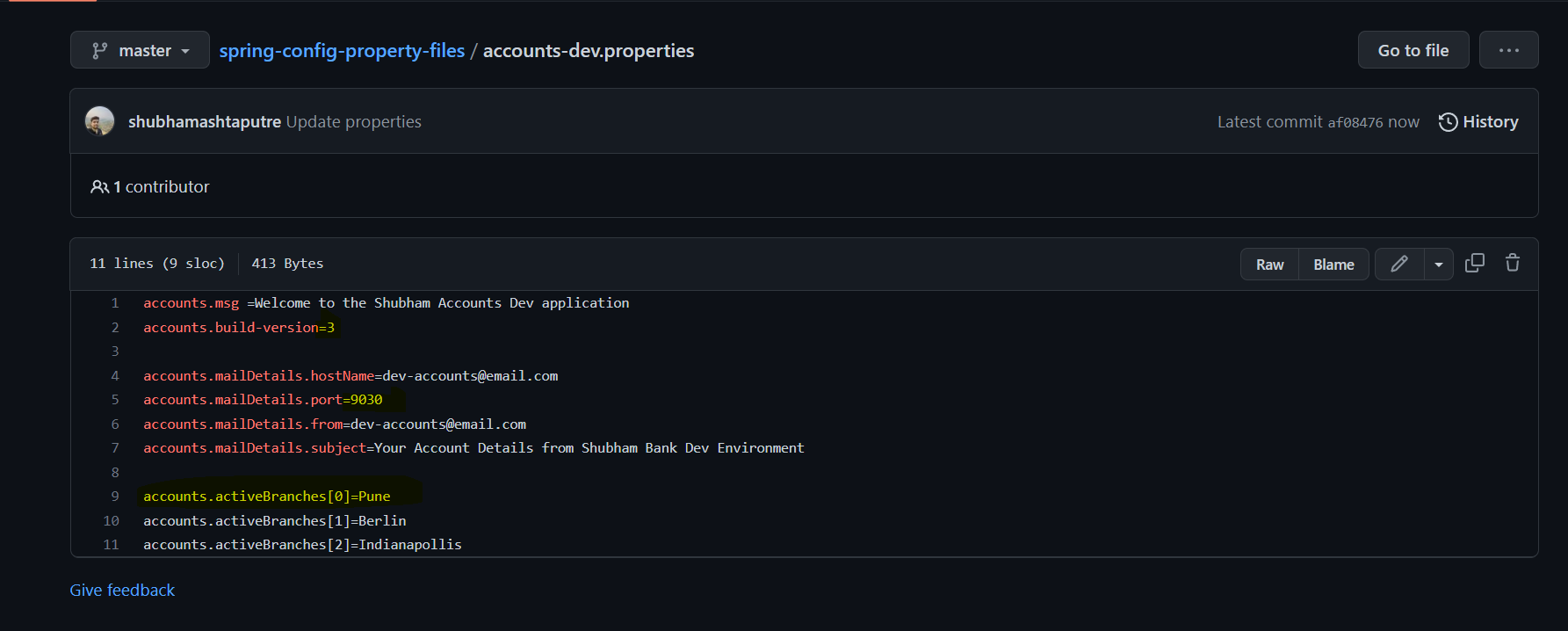
5.3] If you can see my ‘dev’ environment is active in the account microservice as below:



5.4] now, I will change the property file value of this account dev environment from the github location as my config server is read property values from github

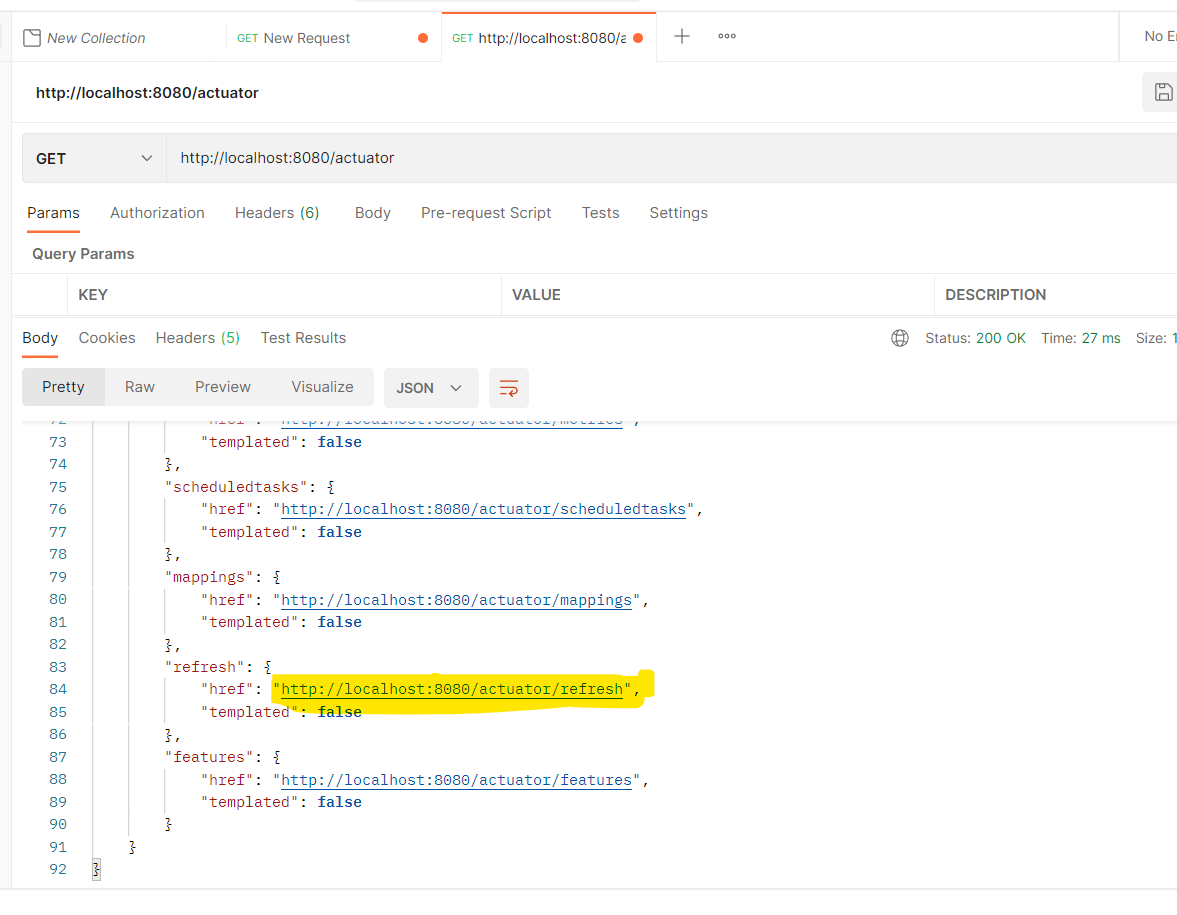


**Original values**



**Updated Values**

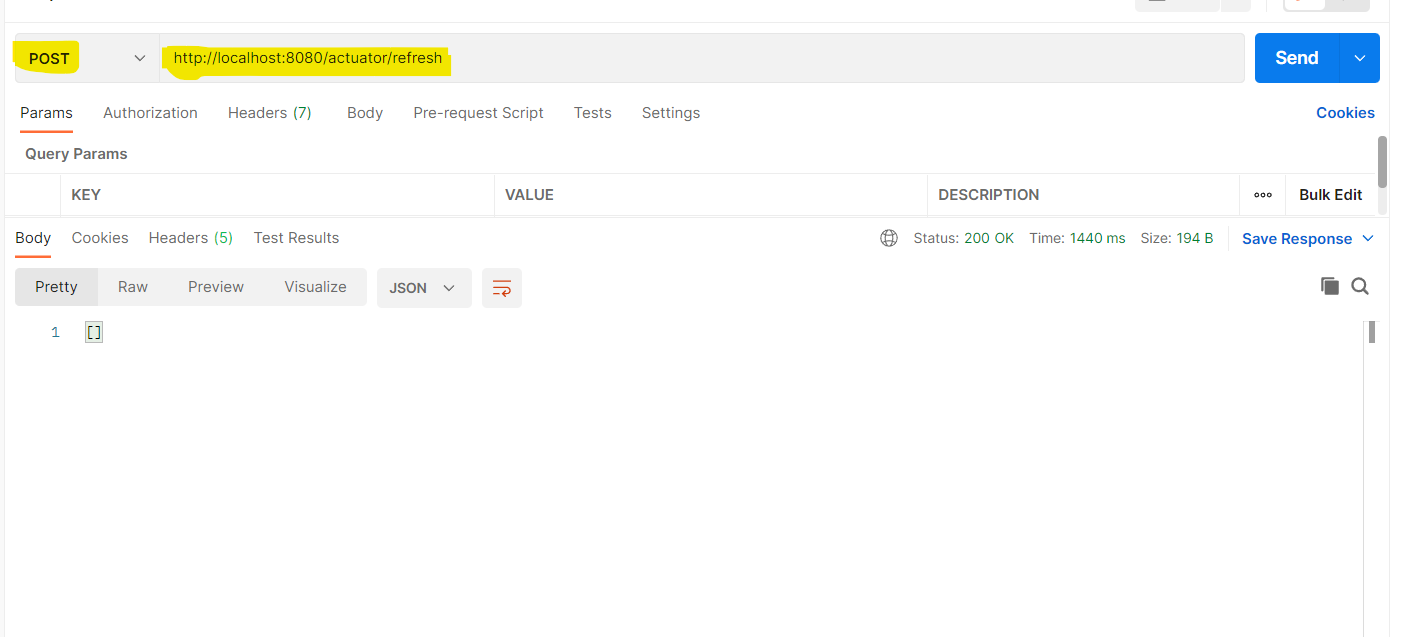
5.5] Check the actuator in the Accounts url as below:



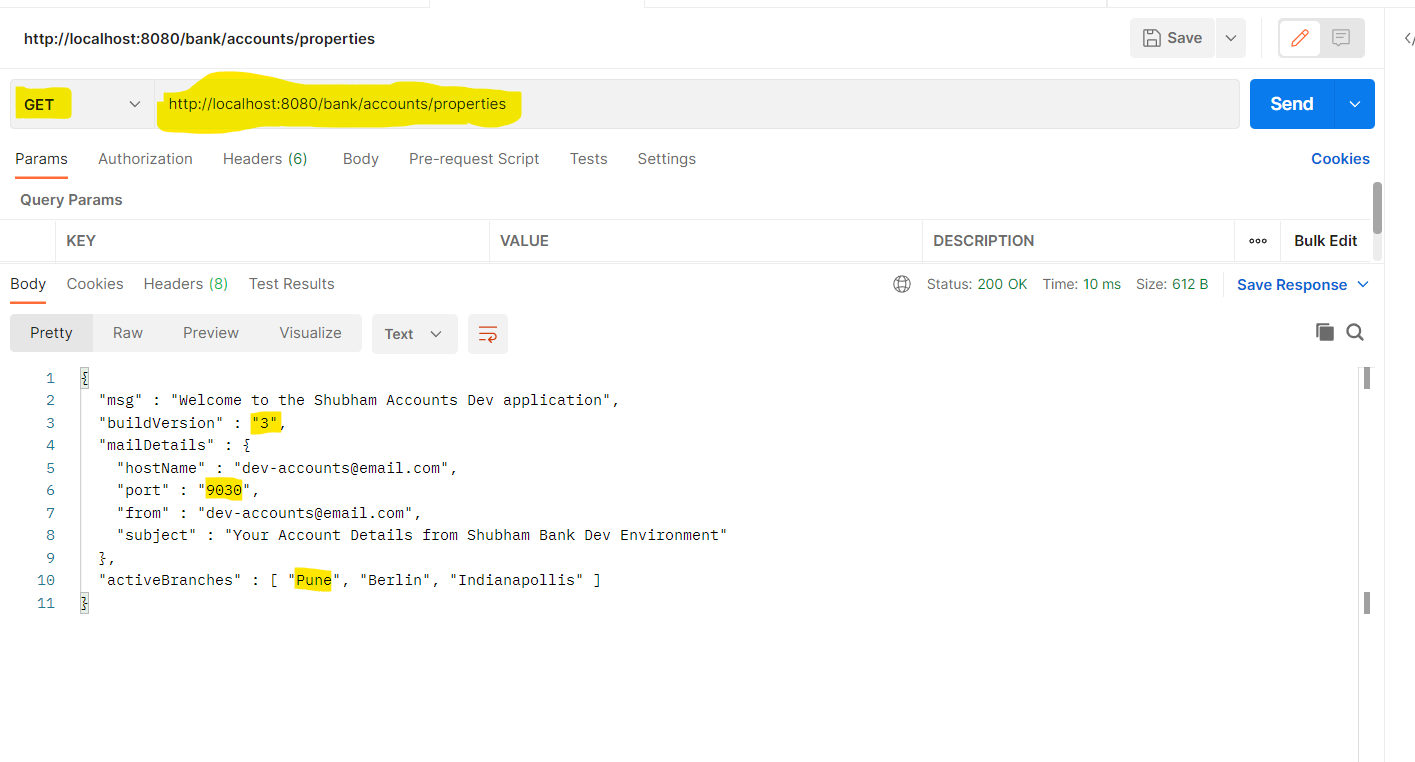
You will see this refresh url as below:

"http://localhost:8080/actuator/refresh"

Hit this refresh URL using POST method as below:



And then call you, account property **URL** you will see updated values:



Without restarting the server these values are updated