**Spring Cloud Config:**

Introduce a better option or the recommended approach that we need to use inside any organization where they are building hundreds of microservices. This option is by using the spring cloud config. Inside the spring ecosystem, there is a project with the name spring cloud config just to handle the configurations inside any cloud native systems like microservices or any cloud applications that we are trying to build.

In this approach, we are going to have a centralized configuration server, which means we need to build a separate application that is going to act as a configuration server with the help of spring cloud config. Using this centralized configuration server, we can overcome all the limitations and drawbacks that we discussed in the previous slide, and whenever we are using this approach, our centralized configuration server is going to provide server and client-side support for externalized configurations in a distributed system like microservices, which means all our individual microservice, they can register as a client with this spring cloud config server and this spring cloud config server can act as a centralized configuration server.

The centralized config server that we are going to build with the help of spring cloud config revolves around two core elements are:

1. The very first one is we are free to store all our configurations or property files inside any location, like we can store inside a GitHub repo or inside a file system or inside a database. We choose a location where we want to store all our configurations and properties securely. Once we store all our properties or configurations, then these configurations are where is going to oversee the configuration data within the data store, facilitating its management and distributing to multiple applications like microservices.

So, once we store all these configuration properties inside a centralized repository. As a next step, we are going to create a configuration server with the help of spring cloud config, and this is going to load all the configurations by connecting to our centralized repository. So, now we are config server holds the properties of all the microservices and environments.

Now as a next step, our individual microservices, they can connect with this config server as a config client, and they can load the configurations during the startup by connecting to this configuration service. So, this is how it is going to work.

Like we can see, there are three microservices like loans, accounts and cards, all of them during the startup They will connect to these centralized configuration servers. This way we are delegating all our properties and configurations to an externalized location and all our microservices they are going to read these properties during the startup based upon the profile that is activated. Apart from supporting all the features of Spring Boot, this config server also overcome all the limitations.

This is the official website https://spring.io/ -> Click on the projects -> Click on the spring cloud -> In spring cloud there are many other sub projects that will help our various challenges while we are building cloud applications or cloud native applications or microservices. -> Right now, we are looking at spring cloud -> Under the spring cloud -> Click Spring cloud config.

What Is Spring cloud?

The Spring Cloud provides various frameworks or projects for developers to quickly build some of the most common patterns of microservices or any other cloud native applications. For example, we already saw spring cloud config is one such framework or project available inside the spring cloud, which is going to help us to overcome the challenge or to implement a common pattern which is maintaining the configurations inside the microservices environment. Very similarly, Spring Cloud also has various other projects, which is going to help us to overcome the challenges and implement common patterns like service registration and service discovery, routing and tracing, load balancing, spring cloud security, distributed tracing and messaging. These are the most used projects inside the spring cloud. But apart from these, there are many other projects available inside Spring Cloud.

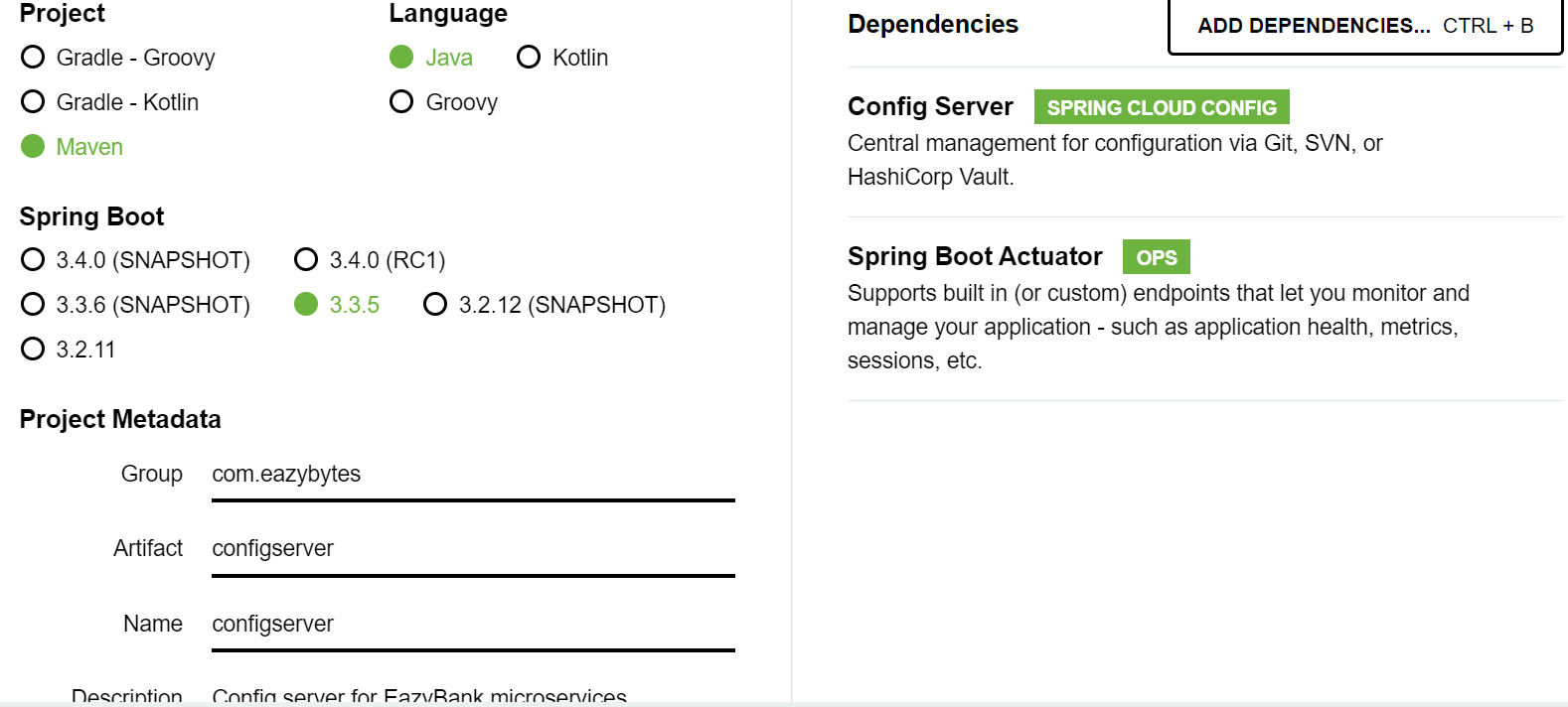
Create a folder -> v2-spring-cloud-config -> Create a project with -> <https://start.spring.io/> website -> Adding dependencies whenever we build a centralized config server we have to select the dependency **config server** ->  Whereas config client we should use whenever we are trying to connect our microservice with the Config Server. So, inside our individual microservices like **accounts, loans and cards**, we need to add this dependency **config client** -> Add the **Actuator dependency** it is going to support to monitor and manage our applications, such as application health, metrics, session etc. -> Generate project.

Why we have completely different version numbers for Spring Boot and Spring Cloud?

The reason is Spring Boot and Spring Cloud are two different projects inside the spring ecosystem. They will have their own version numbers. So, when we try to create a spring boot project from the **start.spring.io**, the automatic mapping of the spring boot version to the spring cloud version will happen automatically inside the website.

And it is also going to add a dependency to add these spring cloud related dependencies along with the version that we have defined inside the top, which is 2023.0.3.

If we need some more information about the version and other things we can go to the official document.  
<https://spring.io/projects/spring-cloud-config>



Step – 1: Open the configserver application -> Main application add the annotation -> **@EnableConfigServer** This is the annotation.

Step – 2: Go to the application.properties but like we are following yml configuration -> Rename to the application.yml -> Once we make this changes we have configserver ready in our microservices network. -> But as of now there Is no place for our config server to read the configurations that’s why we need to move all the required configurations of our microservices to a centralized repository -> We need to configure that centralized repository location inside the configserver. So that our configserver will start reading from the centralized location.

And we have multiple locations right from **classpath** to file system, **GitHub** **repository**, **database** and at the same time we can also store inside the cloud like **AWS S3**. So, there are many options. But inside this course we are going to cover three different approaches which are like very common are:

1. The very first one is we'll store all the configurations of the microservices inside the classpath of the config server.
2. The second approach is we can store inside a file system. When we say file system, we can store anywhere inside our server or inside our local system and we can read the configurations from the folder inside our file system.
3. And the third approach, which is the most used, is with the help of GitHub.

**Approach 1 - Reading configurations from the class path location of Config Server**

Step – 1: First let me try to define a new property inside the application.yml of config server. And this property is spring and under the spring we need to invoke application. After the application we need to invoke name. So, we are trying to set a name for config server. So, whenever we are trying to set a name for our spring boot application, we can use this **spring.application.name property**. Or we can also follow this Yaml configuration inside our spring boot applications. So, the name that I want to give here is config server itself.

spring:  
 application:  
 name: "configserver"

So, from now onwards for all our spring boot applications we are going to give the name for them. Using these names, only the entire spring cloud config server concept is going to work.

Step – 2: Now we'll go to the resources folder and inside this resources folder we are going to create a new folder with the name config because inside this config folder only we going to store all the configurations related to other microservices.

* First, we'll try to get the configurations from the accounts microservice. We know, under this we have three different profiles like **default, prod and QA**. So let me try to copy these three values to the config server location.

So, under this config directory we are trying to paste all these three new files. So here we can see we have three different files.

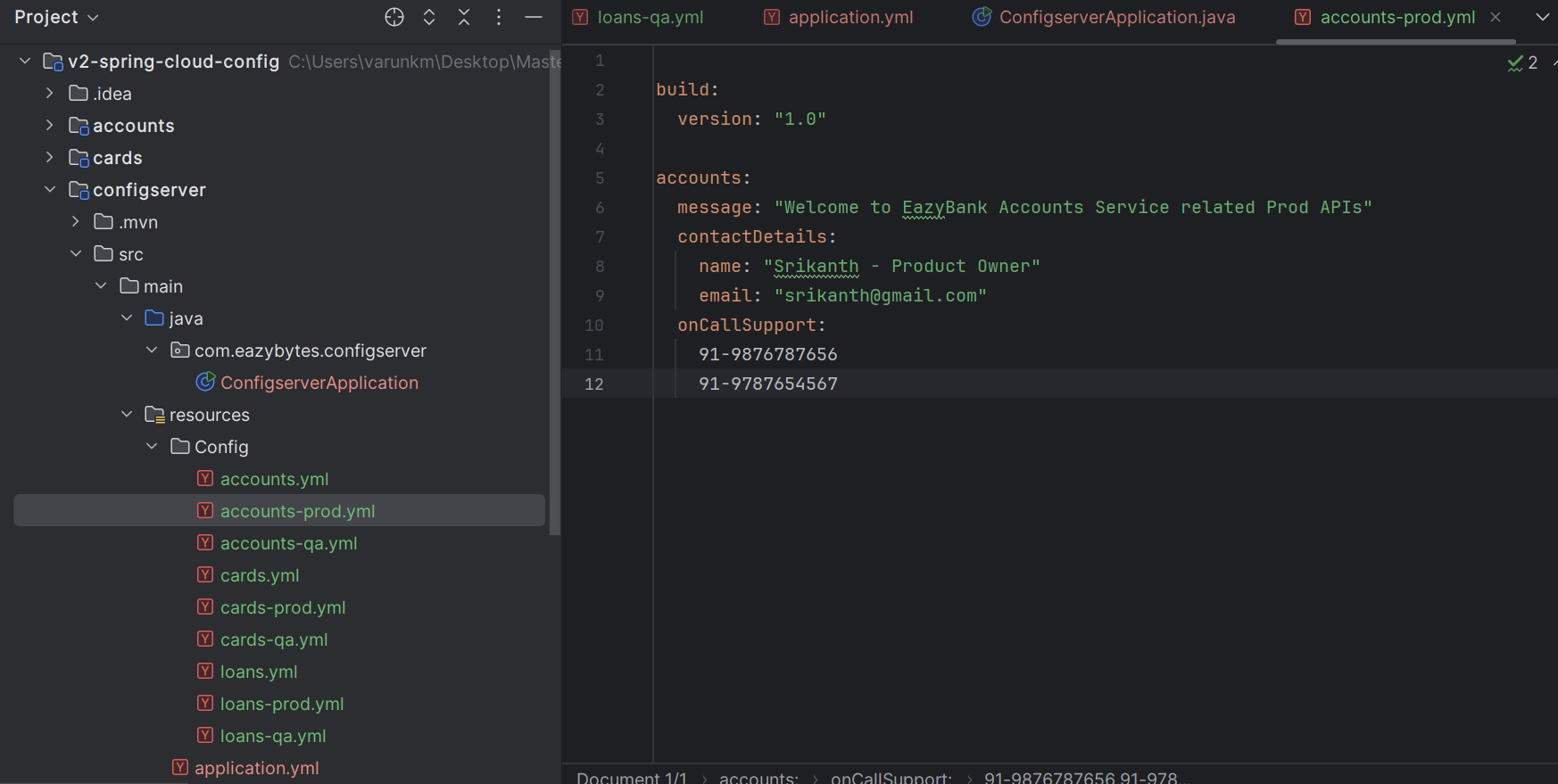
But here there is a challenge for our config server. Like for accounts microservice we can mention these three files.

If we try to bring the files of cards microservice and loans microservice or any other microservice, they are all going to have the same name.

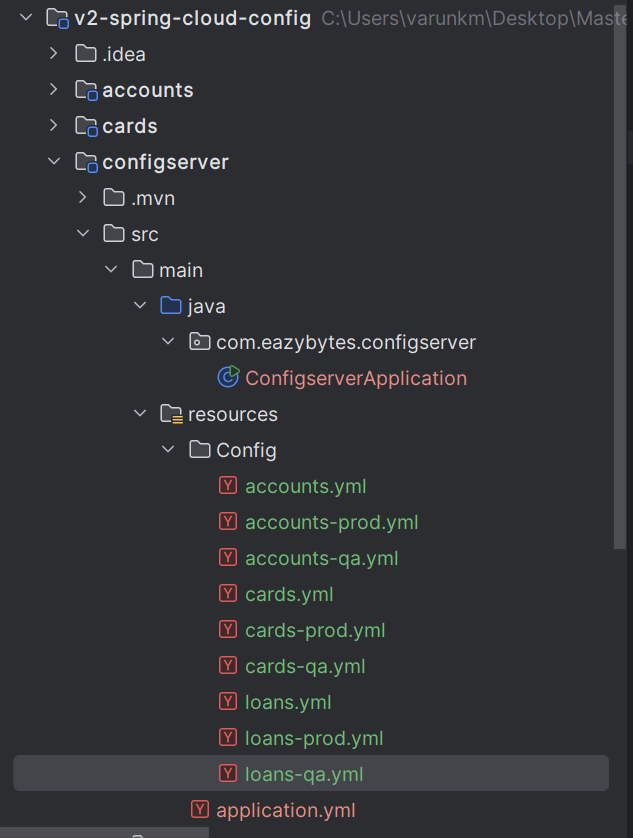
So that will create a lot of confusion to config server like which files it has to load, which properties

it must load during the startup. To overcome this challenge, all our property files, we should name them with our spring boot application name or with the microservice name itself. For accounts, **we are going to create Config folder under the resource folder -> paste all our accounts yml files** spring.application.name as accounts in few minutes. That's why we can change the name of this application.yaml to accounts.yml. Very similarly, we will do the same for others file as well.

We need remove all the properties except only properties related to the build version and accounts details like above for all the three files like accounts.yml, accounts\_prod.yml -> accounts\_qa.yml.

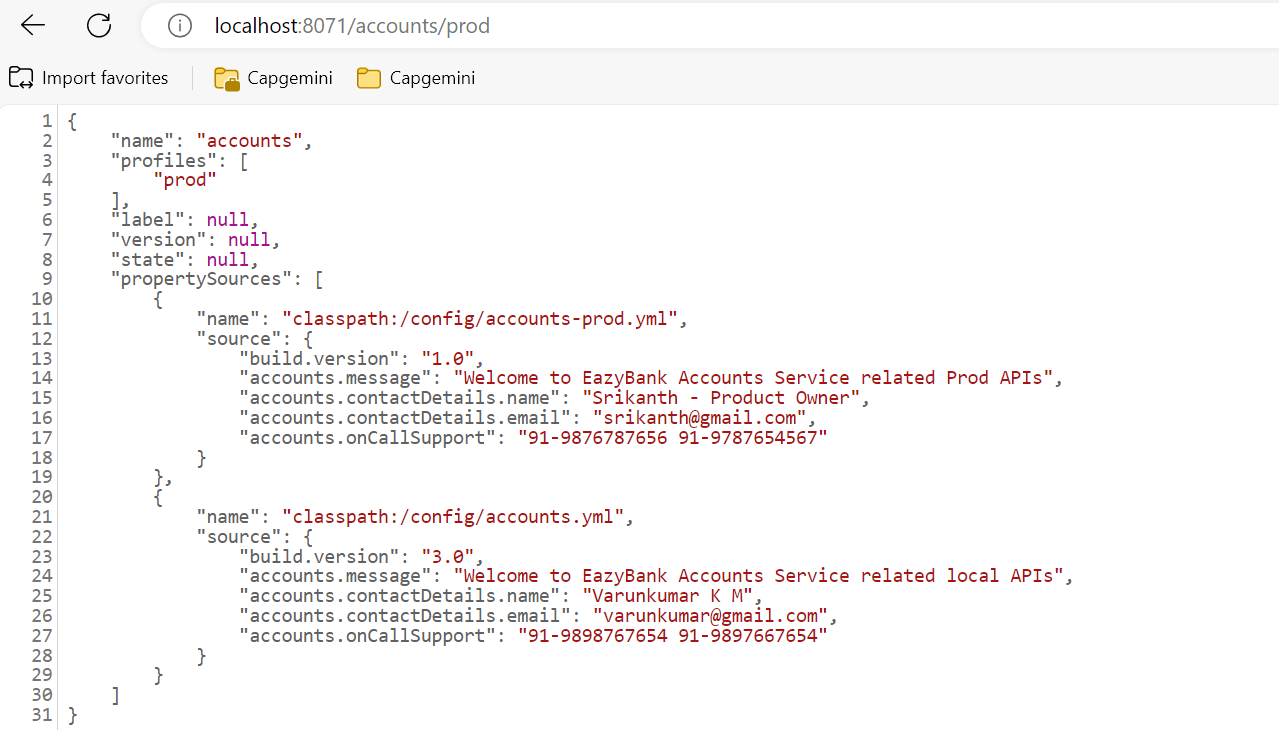


* Do the same thing to the cards and loans microservices with in the same config folder we must paste those files. Instead of underscore we need to use hyphen for the **accounts-qa.yml** like this

We must communicate to my spring cloud config server that we have stored all these files in so and so, location. For the same, we need to open the application.yml of spring cloud configserver. Here we need to create few configurations or properties.

spring:  
 application:  
 name: "configserver"  
 profiles:  
 active: native  
 cloud:  
 server:  
 native:  
 search-locations: "classpath:/config"  
server:  
 port: 8071

Now, we start our configserver application -> Application started with the port 8071 -> check the apis in web



We are getting the properties from prod profile and default profile. That's the expected behaviour because by default all the properties inside our default profile will be loaded.

Apart from that, since we are trying to access the prod profile, it is going to load the properties related to the prod as well. So, these are the prod profile properties, and these are the default profile properties. And during the startup of our microservice, they are going to ignore the default profile values and they will simply follow the prod profile if we try to activate the same prod profile.

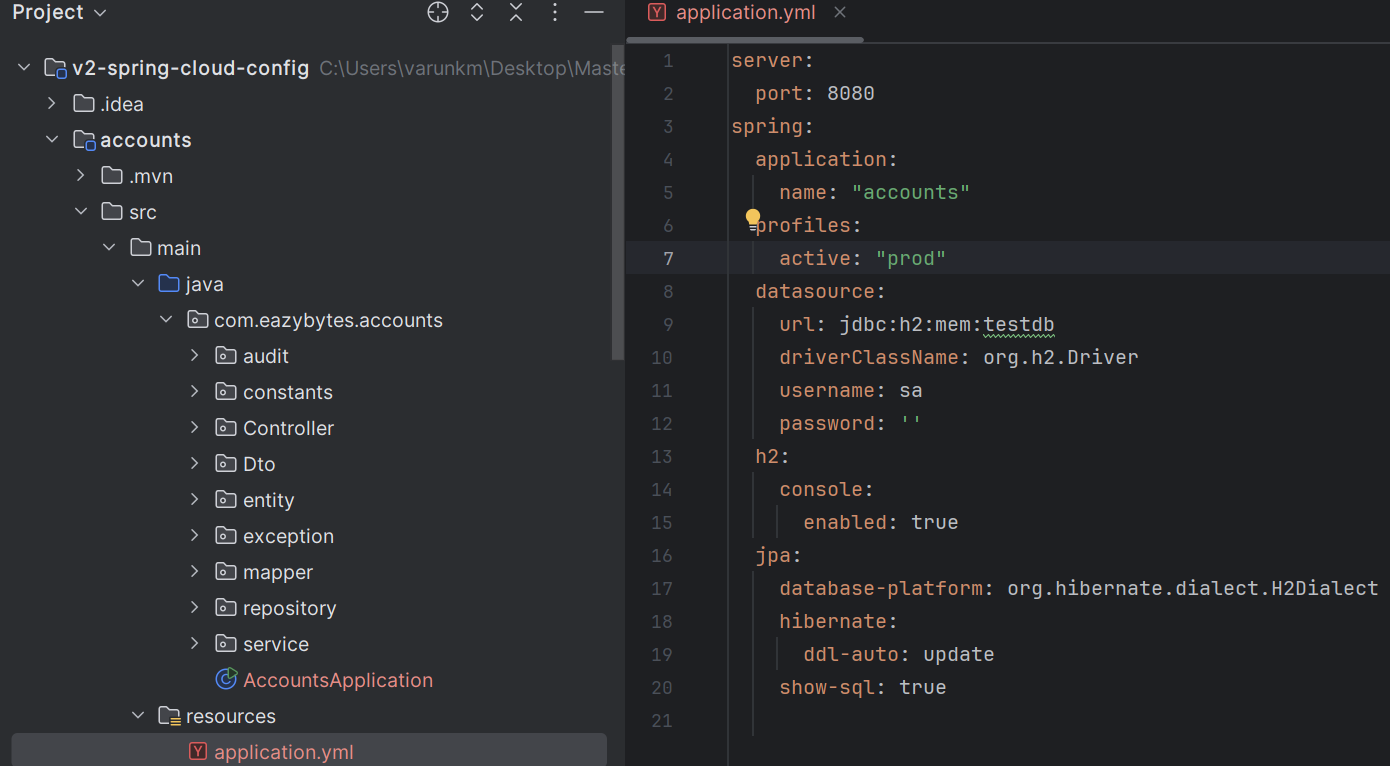
So, with this all the profiles are loaded into my spring cloud config server like whatever configurations we have stored inside the classpath, all of them loaded during the startup of the config server.

As a next step, we need to establish a link between the config server and our individual microservice. So that during the startup of the individual microservice, based upon the profile that we are trying to activate these APIs of the config server will be invoked and the property details present inside them will be considered by our individual microservice.

**Updating Accounts Microservice to read properties from Config Server**

Step – 1: Delete the prod and qa environment properties -> Inside the application.yml file delete all the properties related to the configuration.

So now I have the properties related to the server port. Spring data has to database only and there are no properties related to my spring boot profile files



So, make sure we are giving the same name inside the accounts microservice by following the spring application name. Using these application name only our individual microservice is going to request config server, my name is accounts, please give me the properties related to me based upon the current activated profile. We can also create a property to activate a profile by default. So, for the same we need to mention spring. Under spring we must mention profiles, post profiles active. After this active, we going to mention the profile as prod, which means by default, whenever my accounts microservice is being started, it will activate based upon the prod profile.

If we want to change this behaviour from an external configuration, we can always pass this property which is spring.profiles.active with the help of command line arguments or environment variables or JVM system variables like we discussed in the previous lectures.

Step – 2: Got to the pom.xml file for the accounts microservice -> inside the pom.xml file we need to add a dependency related to the spring cloud config client.

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>

We should also create a property defining what is the version of Spring cloud we want to use. we already have this property, in the spring initializer paste it Java version and post that,

<spring-cloud.version>2023.0.3</spring-cloud.version>

We should also make sure we are copying these dependency management related configurations. The same we need to mention just about the plugin related information. So here mentioned this dependency management

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

So, once we make all these changes, we can click on this load Maven changes which will download all the required dependencies into our accounts Microservice.

Step – 3: As a next step, we need to communicate to accounts microservice about the endpoint details of config server. That's why we need to go to the application.yml of accounts microservice.

And after this config I'm going to mention import, and this import I'm going to pass the URL details of config server and here the URL details that I want to mention is config server colon http localhost 8071. So, this is the port of our config server.

config:  
 import: "optional:configserver:http://localhost:8071/"

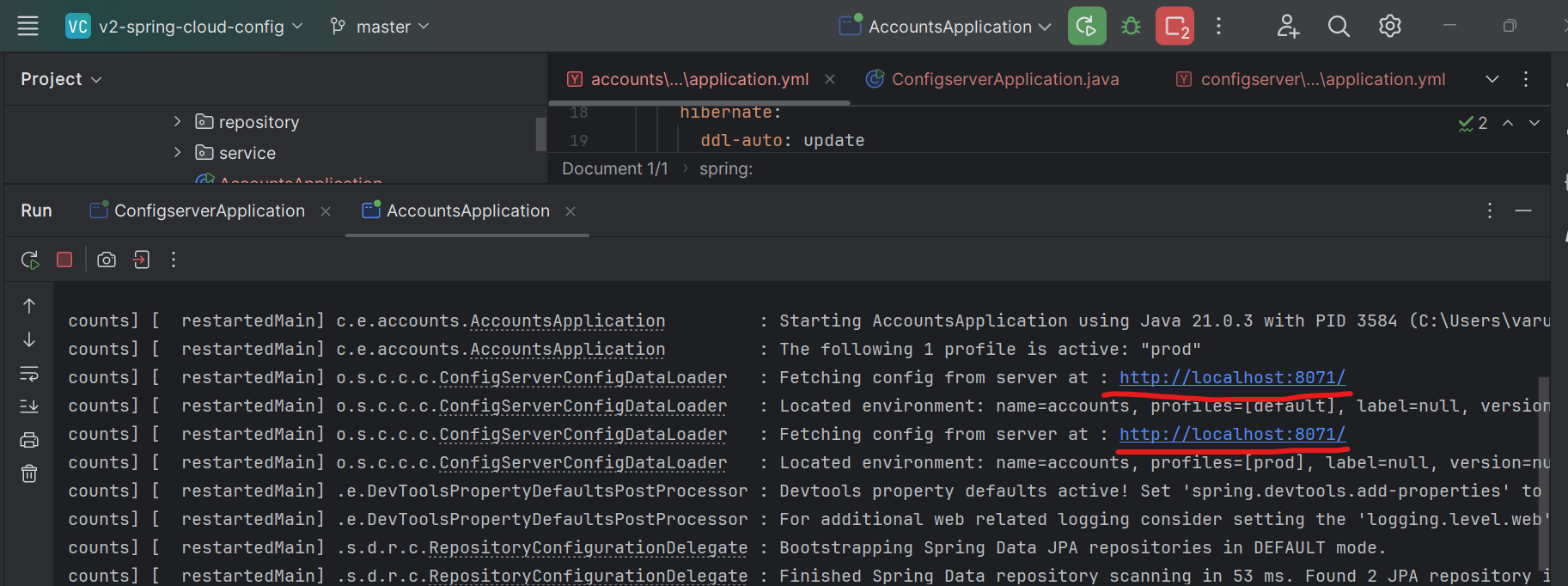
So, this is an indication to accounts microservice or any other spring boot application that, it is going to connect with the config server and don't get confused with the name of the config server

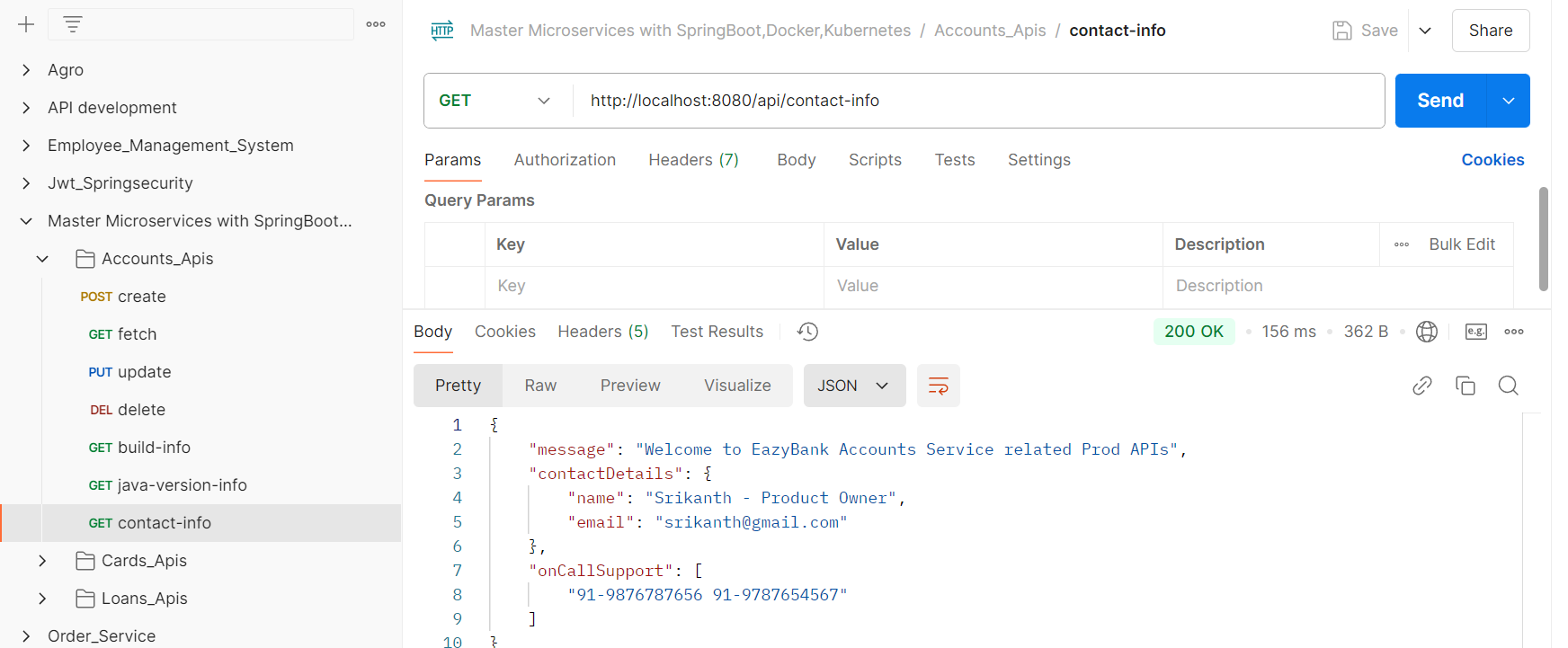
So, with this optional what we are telling to the accounts microservice is, if it is not able to connect with the config server for whatever reasons, it can continue to start the microservice application.

But in our case, since these are not going to be super critical, we have mentioned these optional with this even our config server is not started before the accounts microservice.

My account service is not going to throw any exception or error during the startup. It will just throw a warning, and it will continue to start the microservice application.

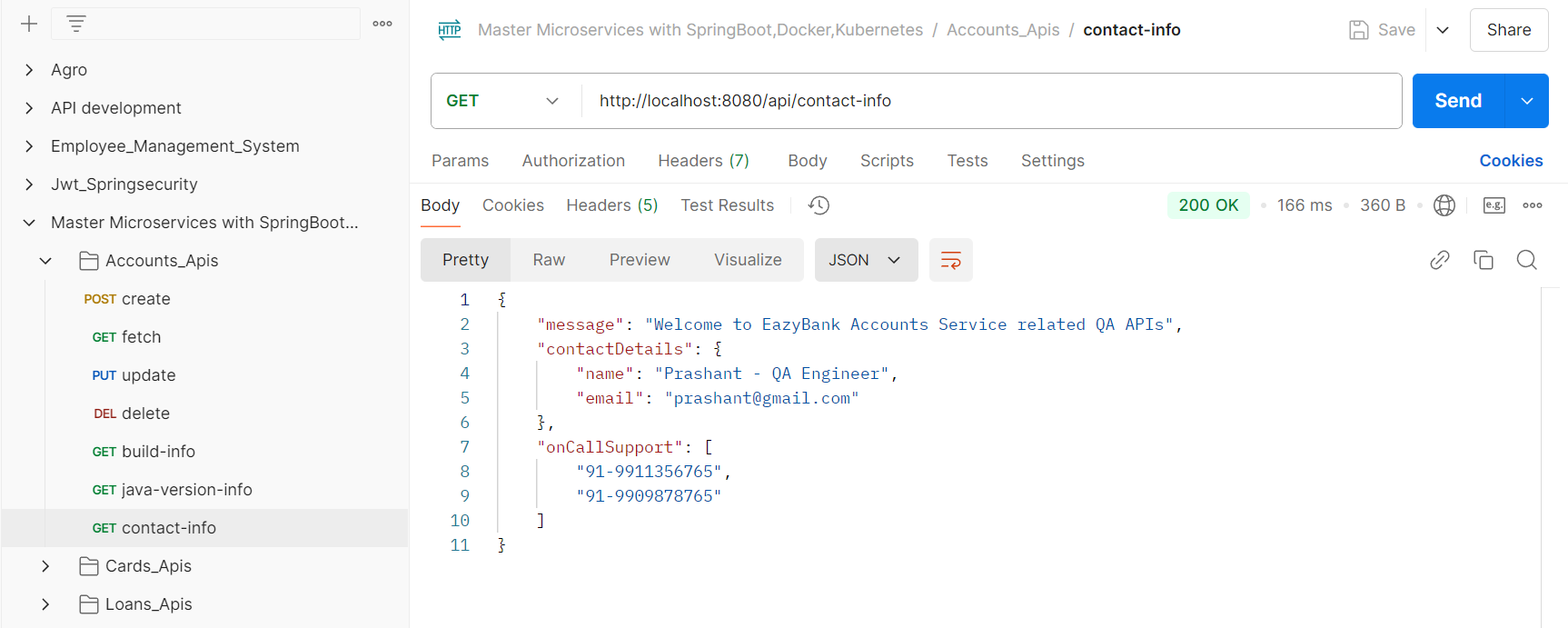
Now if I try to start my accounts microservice, it will go and connect with the config server during the startup. And since we are trying to activate a profile prod with this property here, it is going to fetch only the properties related to the prod profile. And the properties whatever we have defined inside the accounts-prod.yml will be loaded by the accounts microservice.





For changing the environment, we are using the command line argument for that -> Go to AccountsApplication -> Right click -> Modify run configuration -> inside the program arguments -> add this command **--spring.profiles.active=qa** -> Apply, ok

Then start our application has running on the QA environment.



So, this confirms that our accounts microservice is perfectly integrated with the config server and we have moved all the properties of accounts microservice to a different centralized repository, which is config server.

Also, with the same procedure integrate our cards and loans microservice with the config server.

This way we are successfully integrated all our microservices with the spring cloud config server. Right now, all the properties of our microservice are being maintained by my spring-cloud-server. Regardless of how many microservices we have, like 3 or 100 microservices, the process is going to be the same. But as of now we are storing all our microservices inside the classpath of config server. This approach may work for few projects and few other projects they may not like this approach because we are trying to store all our configurations inside the config server itself.

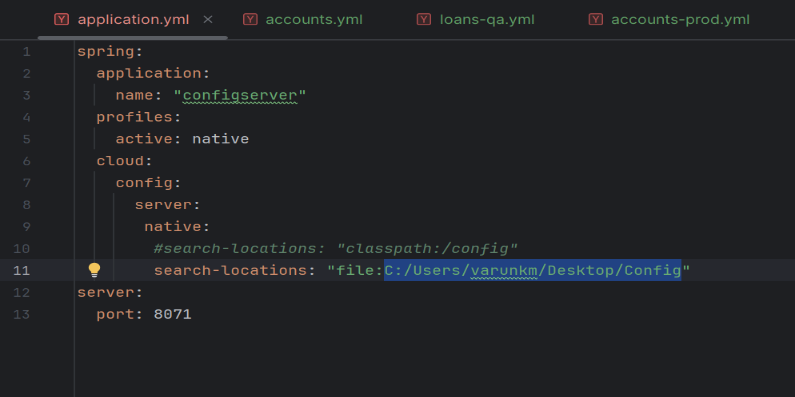
Like if someone has the access to the code of config server. Then definitely they can also see the properties present inside the config server classpath. That's why trying to discuss on how to store all our configurations inside a file system location, so that we can separate the configurations from the spring cloud config server.

**Reading configurations from a file system location:**

And here you may have a question like why some projects they prefer to use file system approach. Using file system approach. We are going to maintain all the required properties inside the server location where our microservice is deployed. And this approach has an advantage like our server admins, they can enforce some security restrictions. Like no one can open that folder and no one can see the content of that folder except our config server application. So that's why few projects they follow this approach as well.

Step – 1: Create one folder ex: config -> Copy all the files in the config folder in local **C:/Users/varunkm/Desktop/Config**

Step – 2: open the config server application.yml file -> instead of classpath of search-locations we paste this location where the config files are available. Example below:

With this now our config server will point to new server search location and this new search location is following the approach of file system.

Step – 3: Start the config server, once its start completely the start the other services like accounts, cards and loans applications.

**Reading configurations from a GitHub repository:**

Using the same GitHub repo, our config server can try to load and read all the properties during the startup. **And this GitHub approach is the most recommended approach** because when we try to store our properties inside GitHub repo, we will get multiple advantageslike we can properly secure our GitHub repo. So that no one can access it.And at the same time, it is also going to support versioning, auditing, in future maybe down the lineafter one two yearsif we want to understand what is the property that we use to have two years back or one year back,we can always see the history inside the GitHub repo, whereas other approaches like **file system** and **classpath** it is nearly impossible to track such versioning changes.So, to get started with the GitHub approach, first we need to move all our properties into a GitHub repo.

Step – 1: Move all our properties into a public GitHub repo.  
<https://github.com/varunkumarkm/eazybytes_repo>

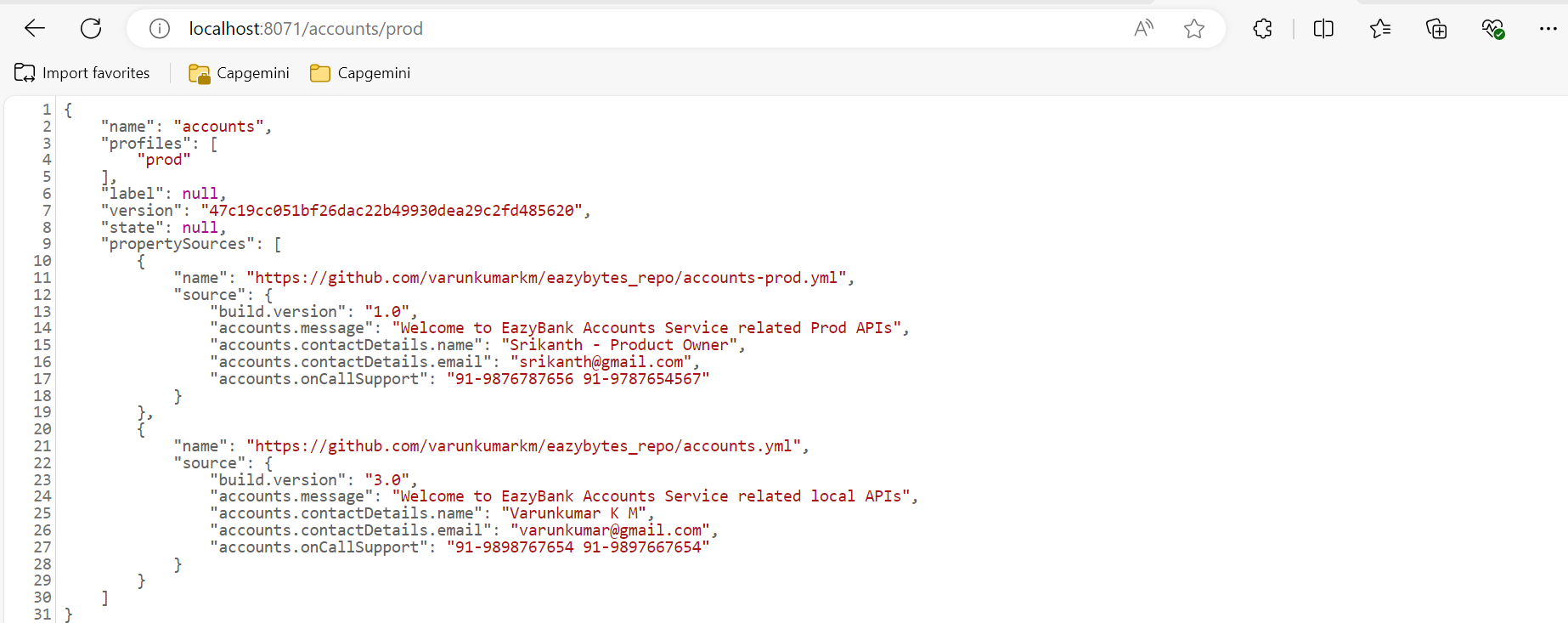
Step -2: Copy this path -> when we are using profiles in the GitHub, we activate the profile is git.

default-label – it means we are using the **master** repository. This default label element is going to helpful if you have multiple branches inside your GitHub repo. To avoid such confusion, we need to always mention what is the default label or what is the default branch.

**timeout: 5** - Under this timeout I'm going to give a value as five. So, this represent that my config server should wait only for maximum five seconds and after the five seconds if it is not able to connect to my GitHub repo for whatever reasons, the config server should immediately throw an exception. So, this will allow my config server to fail immediately and that will give an exception to the operations

**clone-on-start: true** – With this property is telling our config server please try to clone the git hub repo to local system during the start-up itself. If we don’t mention this value has true the cloning of the GitHub repo will happen only when the very first request come to our config server and this may result into some issue our config server gets started properly but when the very first request coming towards our config server like in the form of accounts microservice, cards or loans microservice then it cannot clone and reload in such scenario creates some issues in the accounts microservices.

**force-pull: true** – Sometimes we might have changed some local changes inside the local repo that got cloned inside our config server. By mentioning this we are telling to do all the local changes whenever trying to start or restart our config server. This will make sure our config server reading the properties from the master location which is GitHub repo.

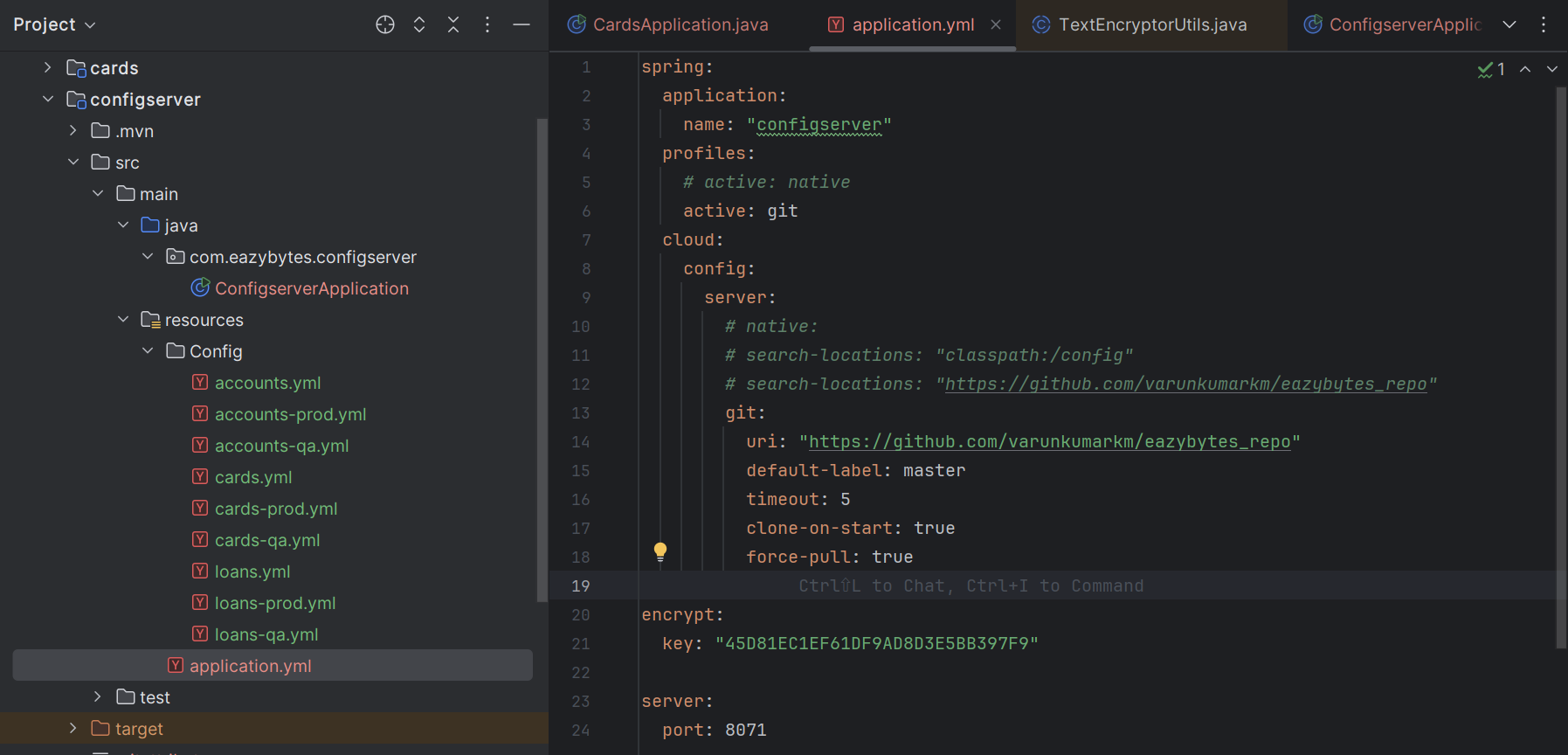


This is the most recommended approach, and these approaches have many advantages -> go to the official website spring.io then explore.

**Encryptions and decryption of properties inside config server:**

sometimes we may want to store our properties in an encrypted format. Currently inside our GitHub repo we stored all the properties in a plain text. What if we have a scenario where we want to store the property value in an encrypted format so that even if someone has access to our GitHub repo or if someone by accidentally open our GitHub repo, they should not be able to see our sensitive property details. That's why it is always advisable to encrypt sensitive properties like passwords or any URL details, folder structures or any other sensitive information.

Step – 1: Go to the config server application. Properties ->



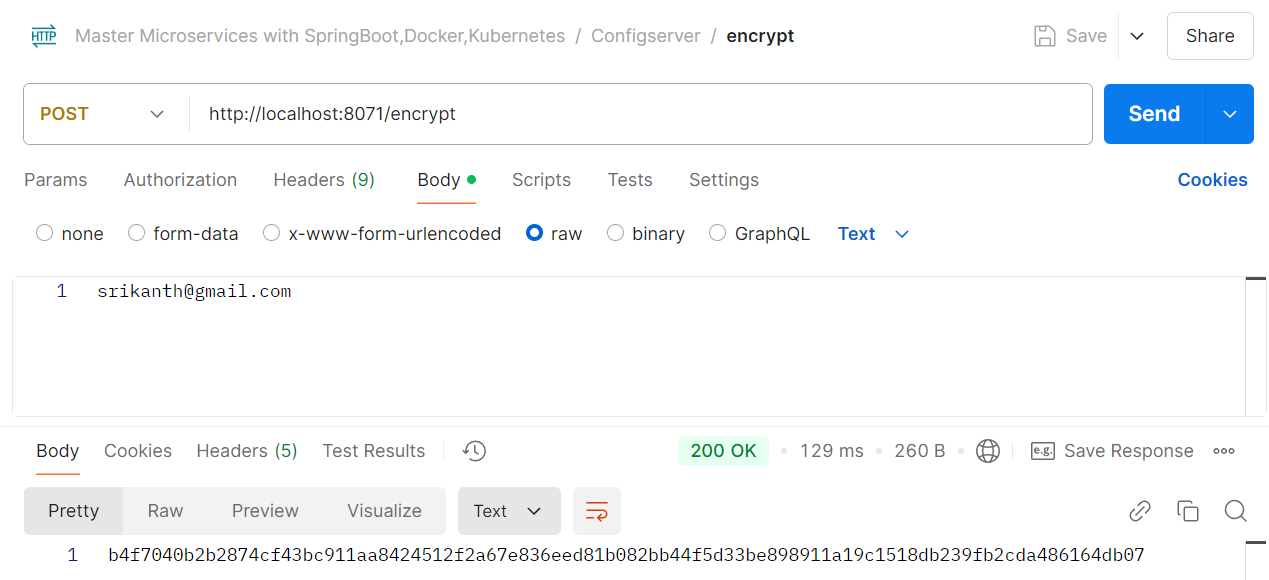
Use to encrypt the properties and to decrypt the properties? Always remember whenever you are trying to provide a secret key, it must be super complex so that it is going to be very tough for the hackers to guess it.

Step – 2: Once we create this property inside our config server, it is going to expose, encrypt and decrypt related APIs. Using which it is going to encrypt or decrypt all our properties.

Executed the config server application -> go to the postman pass this **URL** localhost:8071/encrypt

the URL must be our config server port number which is 8071 and encrypt is the path. To this path we can pass any kind of plain text value, and it is going to give the encrypted value. And this encryption process is going to use the secret key that we have defined inside the application.yml of config server.

So here trying to encrypt a property which is the email inside our accounts prod profile. As of now we can see inside our accounts-prod.yml, under contact details, the email value is mentioned in a plain text value, so anyone who has access to my GitHub repo they can see the email value. Think like I want to encrypt this value inside my GitHub repo. In such scenarios, first we need to understand what the encrypted value of your plaintext value is. That's why I'm trying to take this plain text value from here. Now, inside the body of this post request, I need to select this raw option and please make sure you are selecting this text option because I'm just trying to send a simple text value. So here I have mentioned the email in a plain text. Now if I try to click on the send button, I will get an encrypted value.

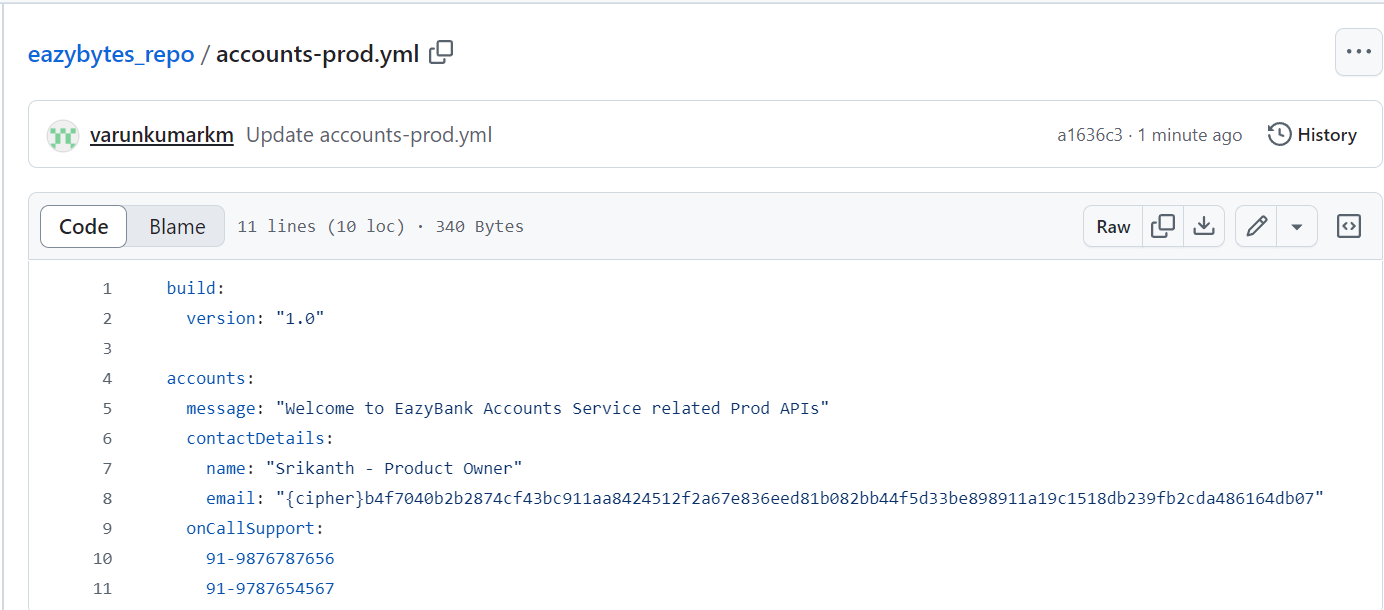


Step – 3: We can copy this, encrypt value and go to the GitHub repo. Inside my GitHub repo like we can see as of now we have the email using the plain text value. Since we have the requirement to encrypt this email value, I can click on this, edit this file, and in the place of plain text value I'm going to mention the encrypted value. Please make sure we also have this double quotes.

But here there is a challenge for my spring cloud config server. How it is going to differentiate between a plain text value and an encrypted value. To help spring cloud config server around this scenario, we need to make sure for the encrypted values mentioning a prefix which is inside the curly braces, we must mention **cipher**.

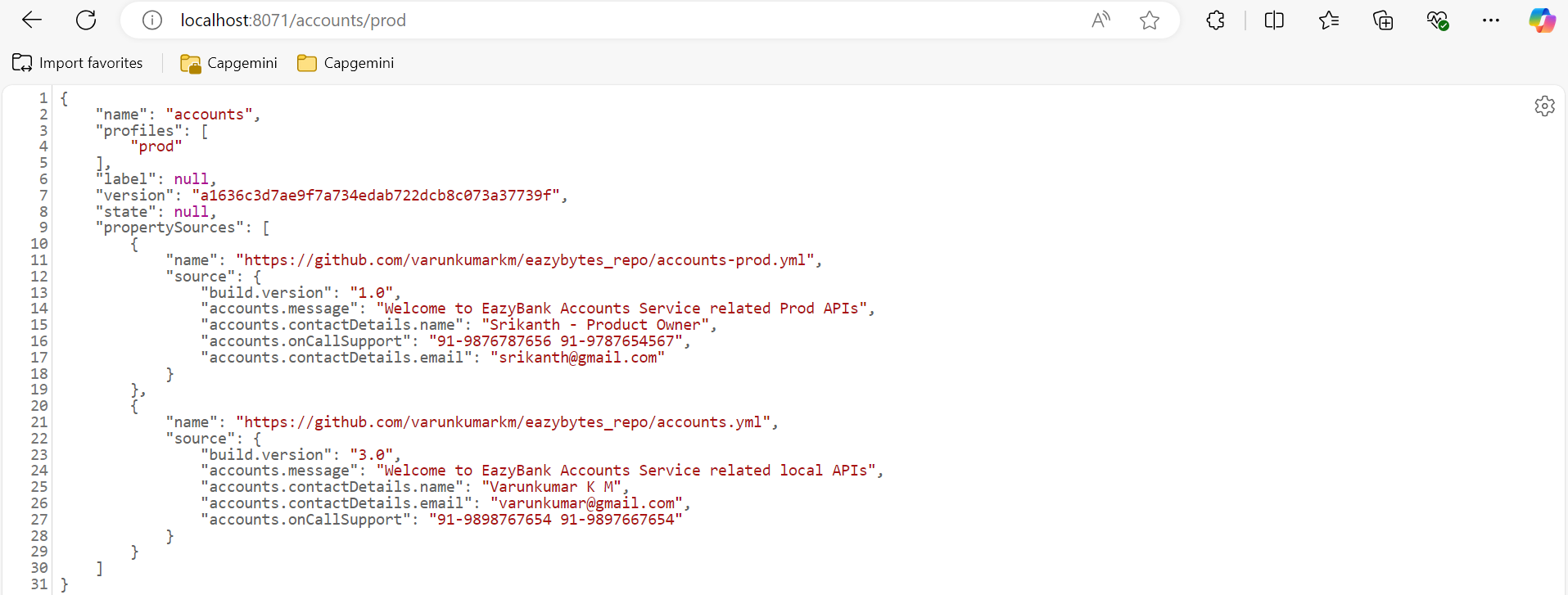
So, whenever my spring cloud config server sees this cipher, which is a prefix to a value, then it assumes whatever value of the cipher is an encrypted value. So, during when it is trying to send these properties to the actual microservice, it is going to decrypt and send the values in a plain text value. This way, even if someone is trying to see the properties inside my GitHub repo, I'm fine because these are encrypted values. They cannot know what the encrypted value is until unless they know the secret key that I have mentioned inside the config server.

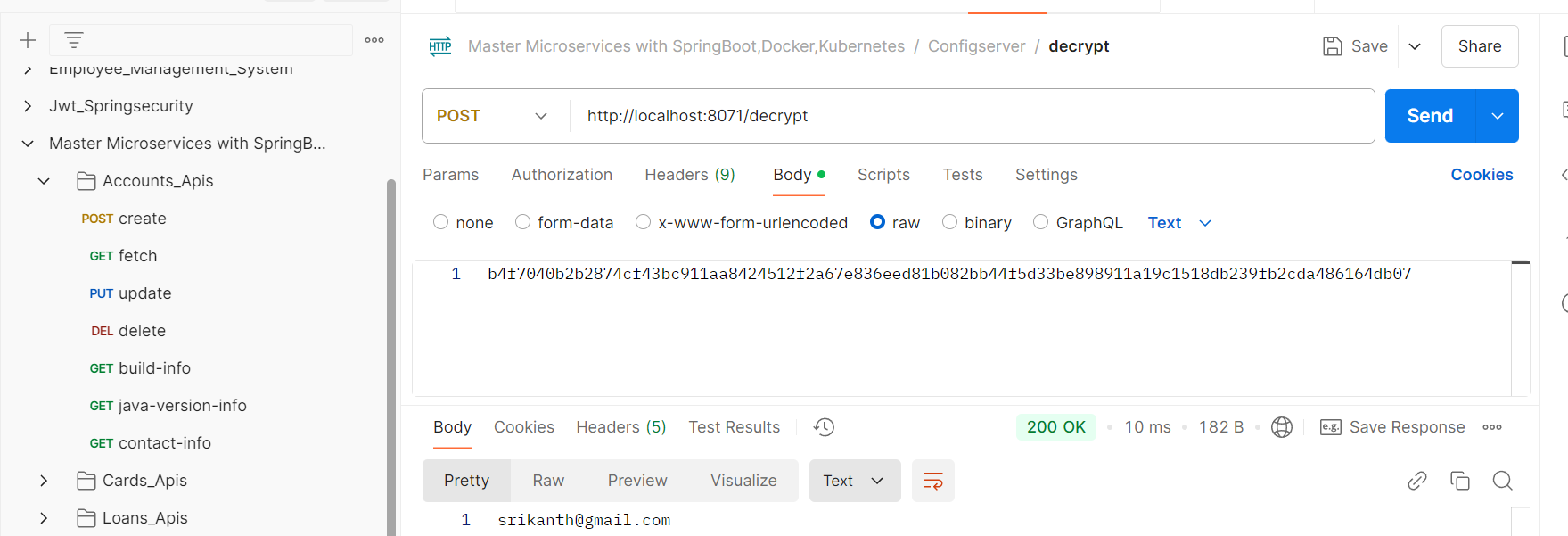
But in real production applications the secret key can be configured with the help of environment variables or CLI arguments or any other approach.



So now commit these changes and commit directly into the master branch. As a next step, let's try to restart our config server so that it reads these latest values. Going to start my config server in a debug mode. Once my config server is started, we can validate if the config server is able to decrypt the value or not by invoking the application slash prod API inside the config server.

So here I'm trying to invoke this API which is account slash prod available inside the config server. getting a response and here the email is decrypted. So, this is the email which we initially encrypted. By the time config server is returning to the clients, it is going to decrypt with the secret key that we have mentioned.





Someone can easily decrypt my encrypted properties then what is the use of encrypting it? Like you said, inside production applications, our config server will work in a very different manner. Here we can invoke any API against config server very easily, but inside production applications, our platform team, they are going to deploy our config server behind the firewalls of our organization. That means no one can invoke the APIs of config server directly. Only the applications which are deployed within the firewall of the organization they can only communicate with our config server. On top of that, if we want to secure our config server, we can secure it with the help of spring security framework, just like any other spring boot application.

**Refresh configurations at runtime using refresh actuator path:**

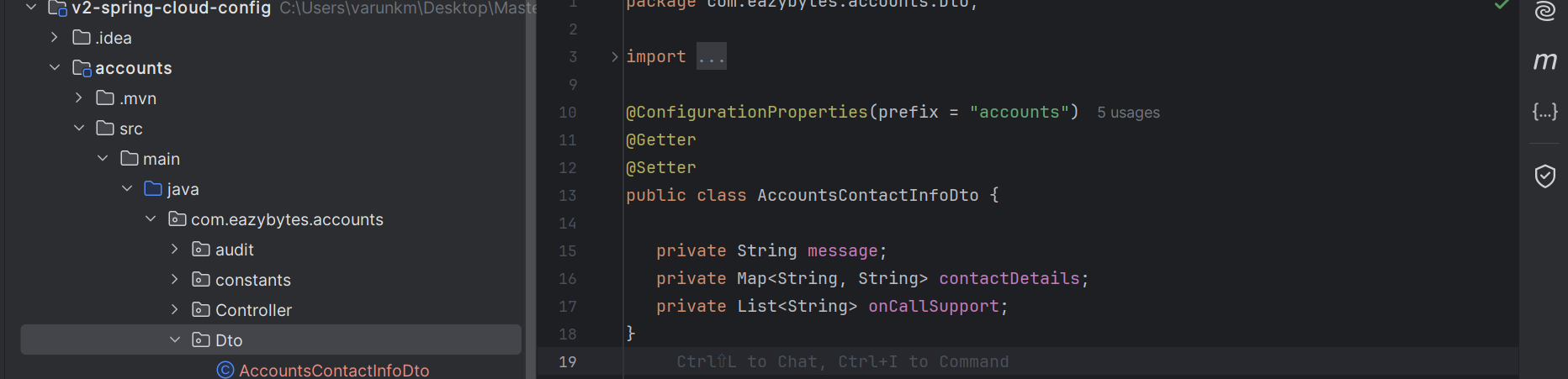
So, everything is working perfectly, and we may think this is the end of the spring cloud config server and we may not face any other challenges in terms of configuration management. But a new problem that we may face inside the microservices environment in terms of configuration management. Think like we have our config server setup and all our microservices started by connecting to the config server they loaded the properties perfectly. Suddenly we want to change a particular property inside our config server and the same we want to be reflected runtime without restarting our microservices.

Here we may have a question, what is the problem that I have if I restart my microservices. Inside microservice It's not one microservice there are hundreds of microservices and there will be multiple instances for each microservice. So, restarting our instances of microservices is again a manual task that someone must take care. Whenever we bring some manual tasks inside microservice, then it is going to make our microservices setup very complex.

That's why we should look for an option on refreshing the properties without restarting the microservices instances. For example, think like you have a feature flag which we have configured inside the config server. So based upon a feature flag like a Boolean flag, we want to control the behaviour of our microservice business logic. When the flag is disabled, we want to execute some other piece of code. These flags we want to change anytime inside the config server, and we want the same to be reflected immediately inside our individual microservices without restart. So, this is the most common scenario that projects will try to achieve inside their microservices network.

How to refresh the configurations or properties inside the microservices without restarting the instances.

Step – 1: Add the spring boot actuator dependency in pom.xml file. -> next step we need to go to our DTO classes **AccountsContactInfoDto** where we are hold all our property details.



Same as change this loans and cards microservices as well.

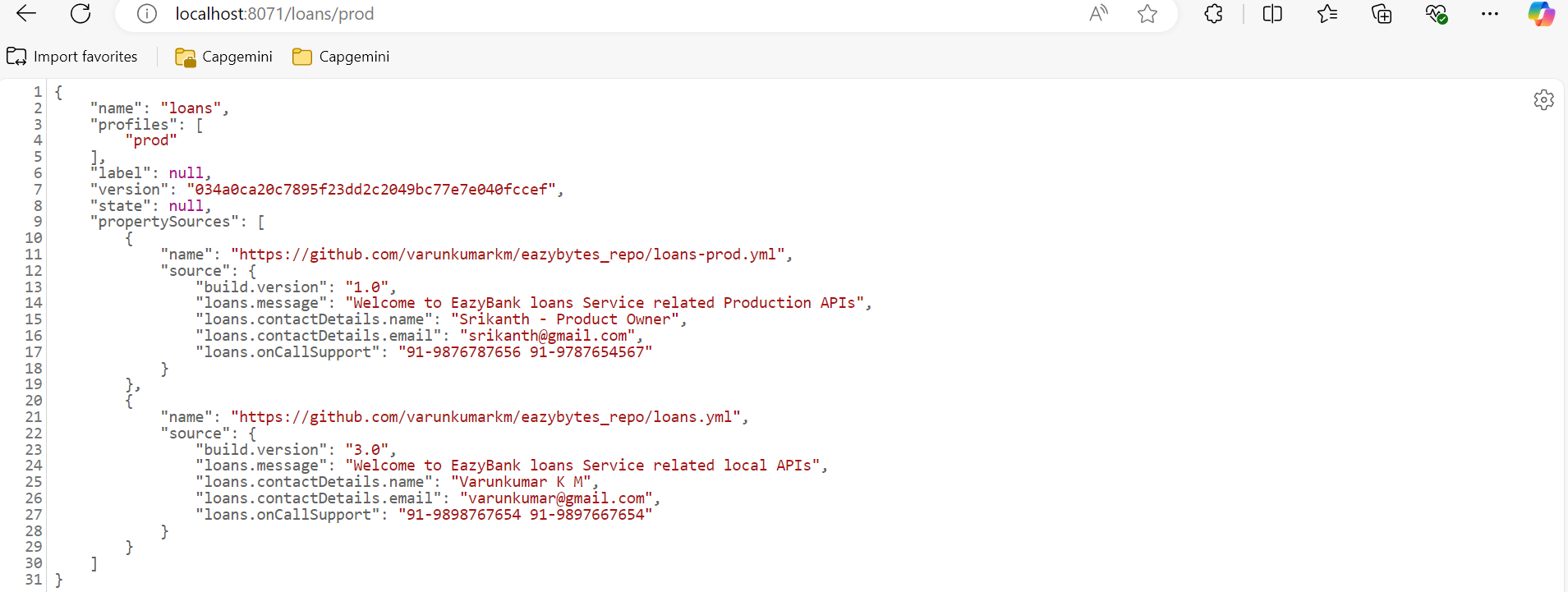
Step – 2: go to accounts microservice application.yml file we need to enable the actuator API paths by default actuator is not going to expose all the management related API paths. That's why we need to specifically enable them by introducing a property here.

management:  
 endpoints:  
 web:  
 exposure:  
 include: "\*"

Same way we should make changes for cards and loans microservices.

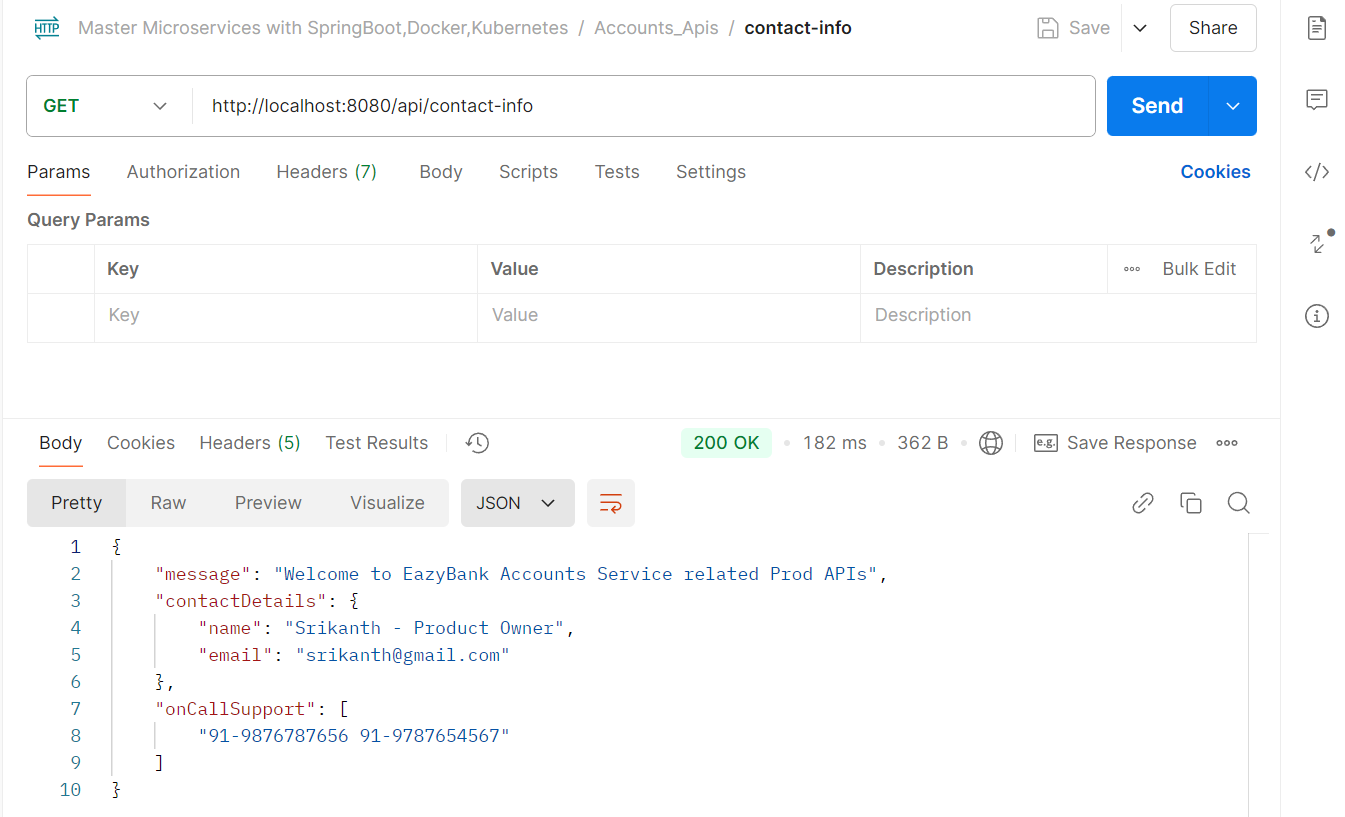
Step – 3:  let's try to start all our microservices post that we can try to change a property runtime inside the GitHub repo and see whether by default, it is reflecting inside our individual microservices.

Our application is running, and I have changed **prod** to **production** my GitHub repos for all the microservices and then I am getting the response form updated GitHub commits.

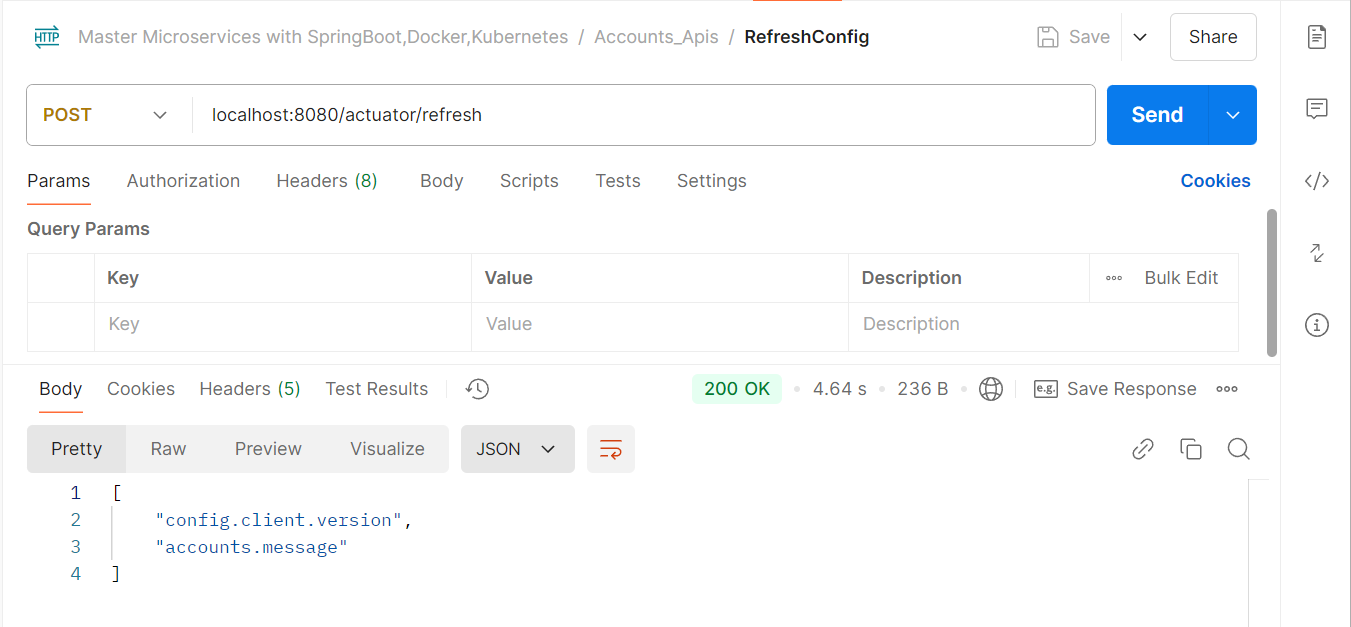


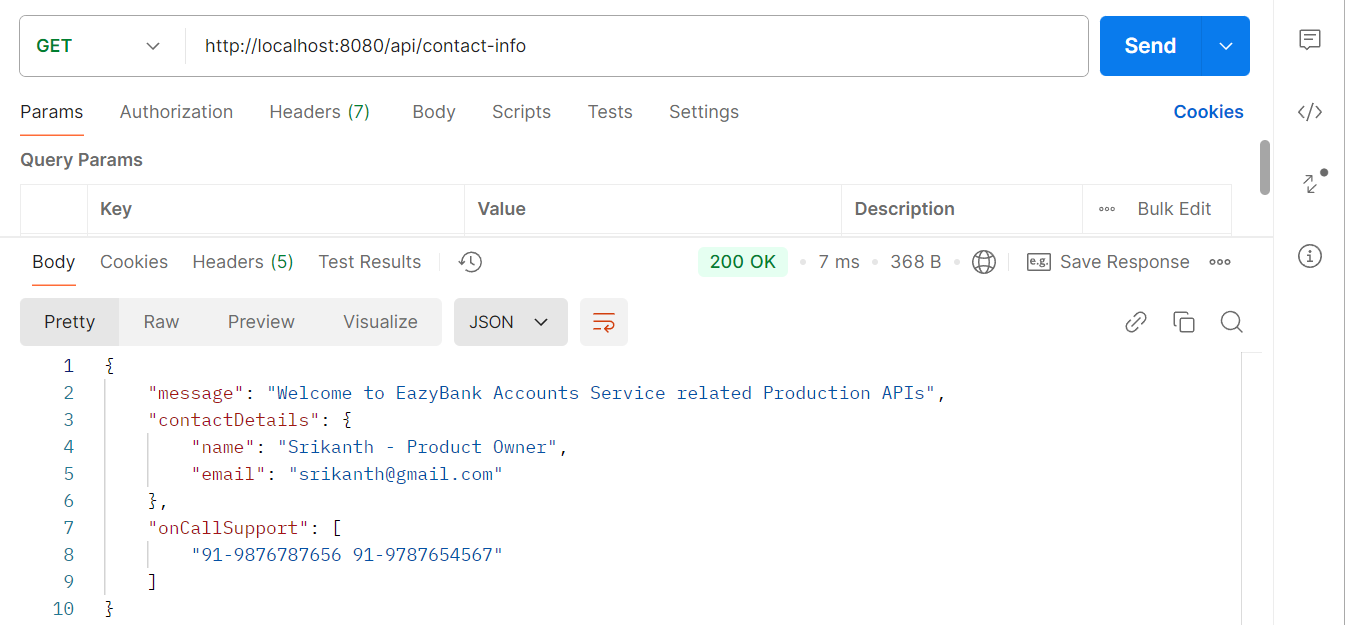
Now the only problem is our microservices should be able to read these latest values and we already know that microservices will only connect with the config server during the startup of the application. Now, to reflect these latest properties, we need to restart the microservices instances and we are trying to avoid that process because we need to avoid restarting our microservices applications as it will impact the traffic and as it involves manual process.

So how to overcome this challenge? actuator is going to expose an API with the name refresh. If we go to the account’s actuator like localhost:8080/actuator against this actuator, we will be having lot many APIs exposed. And here just look for refresh.



It showing only prod here as soon as refresh the actuator in the postman without stopping our microservices we can get the updated property files example below





For the same refresh the cards and loans microservices as well then, we can get the updated property files.

But there is a serious drawback that we have inside this approach. The drawback is thought like we have 100s microservices and each of them has five different instances, which means there will be total 500 microservices instances running inside our production. And for some reason you're trying to change the property in all the microservices. Then we need to invoke the refresh endpoint against all the 500 instances running inside our production. And doing this manually is going to be super, super cumbersome process. Some operations teams are some platform teams, they will try to automate this process by writing some scripts inside the CI/CD pipelines or they will try to write a Jenkins jobs or CI/CD jobs, which will invoke all the microservices instances, refresh and points.

But still, it may not be a convenient solution for many projects. That's why let's explore this further and try to identify is there any better option that we must refresh the properties dynamically without invoking this refresh endpoint for each microservice instances.

**Refresh configurations at runtime using Spring Cloud Bus:**

So, whenever we are using the spring cloud bus behind the scenes this spring cloud bus is going to interlink all our microservices instances with a lightweight message broker like RabbitMQ or Kafka.

With this, the advantage is we need to invoke a bus refresh API path available against our actuator only one time for one of the instances. If there are 500 instances running inside our production, we don't have to invoke the actuator refresh API for all our 500 instances. Instead, we can simply invoke bus refresh API for any of the instance inside this total 500 instances. With that, the spring cloud bus will take care of communicating the changes happened on the spring cloud config server to all other nodes are the instances connected to the same message broker as RabbitMQ.

<https://www.rabbitmq.com/> This is the official website of RabbitMQ -> click on the getting started ->

# latest RabbitMQ 4.0.x - **docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4.0-management**

Run this command RabbitMQ message broker is going to run inside our local as a Docker container. This is a easy way to install and setup RabbitMQ.

Whenever we are trying to set up RabbitMQ behind the scenes, it is going to install two components. One is a management component which is responsible to manage our RabbitMQ and provide the UI to the RabbitMQ.

And the second one is a core component which is going to handle all the message-q related functionality.