Introduction to Configuration Management challenges inside microservices

So, what is this new challenge that we may face while building microservices?

This challenge is about configuration management inside the microservices.

So, to understand more about this challenge, let me try to pose some questions to you.

The very first question that I have for you is, inside microservices, how we are going to separate the

configurations or properties from your business logic, because without separating the configurations

and properties from your microservices business logic, you cannot reuse the same Docker image across multiple environments.

If you club all your business logic and configurations together, then for each environment you need

to create a separate Docker image along with the relevant properties or configurations.

Since it is not a recommended approach like generating the Docker image again and again for each environment, we need to make sure we are using the same Docker image for all type of environments, including the production.

With such requirements, how are we going to separate the configurations from the business logic?

So, this is the very first question that I have, in the similar lines how we are going to inject the configurations or properties at runtime that are needed by our microservices during the startup of the service, some of the sensitive properties like credentials, we cannot mention them inside any configurations or inside any business logic, such sensitive properties, or configurations.

We need to make sure we are injecting them into microservices during the startup of the service.

In the very similar lines, whenever we are dealing with the configurations or properties, we need

to make sure we are maintaining all these configurations or properties in a centralized repository along with the versioning of them.

Because inside monolithic applications you will be having only 1 or 2 applications and it is very easy

to maintain all your configurations wherever you want, whereas with hundreds of microservices it is

going to be super complex to maintain all the properties of all the microservices manually.

That is why when we are trying to build microservices, we need to make sure we are maintaining all the properties in a centralized repository along with the versioning of them.

So how we are going to achieve that?

So, we have all these challenges in terms of configuration management in microservices.

I will reveal the options that we have to overcome these challenges, for the configuration management in microservices, we have multiple solutions available inside the spring boot ecosystem, and these solutions are like right from very basic first approach, which is configuring Spring Boot itself with the relevant properties and the required profiles.

And the second approach is applying external configuration with the spring boot.

And the third approach is implementing a separate configuration server with the spring cloud config

server project.

So, these are the options that we have.

How Configurations work in Spring Boot

First let us try to understand what are the options that Spring boot framework is providing to handle

the challenge of configuration management in microservices.

The main problem that right now we have is, we want to externalize the properties for our microservices.

So that the same immutable code artifact can be used across different, different environments.

So, to handle these kinds of scenarios, we are trying to learn how to separate the configurations and

codebase and how to externalize the configurations from the codebase.

So, let us see if the Spring Boot is going to help us in externalizing our application properties.

Spring Boot lets you externalize your configuration so that you can work with the same application code in different, different environments.

You do not have to rebuild your application again and again.

There are different approaches that Spring Boot lets you follow to externalize the configuration sources, and these approaches include with the help of property files and Yaml files where you can define all the required configurations and properties that your application or microservice need during the startup of the service.

Apart from these property files and Yaml files, we can also inject the configurations or properties

during the startup of the service with the help of environment variables and command line arguments.

So, there are many approaches, so let us try to understand in detail about them.

By default, your spring boot application is going to look for the configurations or the properties

that you have mentioned inside the application.properties or inside the application.yaml file.

We are already using these inside our accounts, loans, and cards microservice where we have defined various properties inside the application.yml file and we place that inside the classpath location.

Apart from these default name, we can also have other property file names and we can make our spring boot to read from them during the startup of the application.

But if you try to mention the properties or configurations inside your property file, you are clubbing

everything along with the source code.

So, this approach may not work to a great extent.

That is why Spring Boot also supports other ways to override the default values that you have defined inside the application.properties file.

Maybe for development environment, you might have mentioned that default database credentials, but at runtime inside your QA deployment or production deployment, we want to override the default values like username, password, url details of database during the startup of the application.

That is why to overcome this challenge we can always override the properties with the other approaches like command line arguments, environment variables.

So here you may have a question like if I mention the same property in multiple locations, will there

be any preference or the priority that Spring Boot follows?

You can see here these are all the most used approaches to provide the configurations inside

your spring boot applications.

The very first one is you can mention your properties inside the files like application.properties or

application.yml.

So, whatever you mentioned inside these files, they will have the lowest priority or the lowest preference if the same property is mentioned with the help of operating system environmental variables, then the previous value which is mentioned inside the application.properties will be override and whatever value you have mentioned inside the environmental variables will be considered by the spring boot framework.

And very similarly, you can also mention the configurations or properties with the approaches like

Java System Properties, JNDI attributes servlet config, init parameters and command line arguments.

The way the priority works is, the lower items will be overriding the earlier ones, which means command line arguments is going to have highest priority.

Whereas the properties that you have mentioned inside the application.properties is going to have the lowest priority.

So please remember this priority and the order followed by the spring boot framework and this is going to be helpful for you inside your real projects.

Now we understand Spring Boot allows us to externalize our configurations by following all these approaches right from basic approach with the help of application.properties or Yaml file to command line arguments.

The next question that you may have here is, the properties are defined in some location or with some approach how to read those properties inside my business logic.

So, let us try to understand the same here again in Spring Boot.

To read the properties, there are multiple approaches.

I will try to present three commonly used approaches inside this course.

The very first approach is with the help of @Value annotation.

So, whenever you want to read a particular property, you can define a Java field inside your business

logic.

On top of that Java field, you can mention an annotation @Value along with the what is a

property key name.

Once you mention what is a property key name with the format like you can see on the screen, during the startup of your application, Spring Boot is going to look for this property inside all the places like application.properties environmental variables and command line arguments.

So, if the same property is defined in multiple places, then definitely it will follow the order of

priority and accordingly it will populate the property value inside this field.

Once we have the property value populated inside this field, you can use that anywhere inside your

methods or inside your business logic.

The next approach that we have is with the help of environment interface.

Many times, in real projects, many properties like sensitive information or sensitive credentials,

they will be configured with the help of environmental variables.

So, the server admins, they will create these operating system environmental variables during the creation of the server very first time.

Or they might have created some scripts which will create the environmental variables during the creation of the server.

So, the same scripts they will follow for all the servers.

This way any server that is coming inside your microservice network or the cloud native applications,

they will make sure all of them they have the same set of environmental variables along with the required values.

So, to read such environmental variables inside spring boots, there is an interface which is environment interface.

So, inside this environment interface it provides many methods to access properties from the applications environment, where your application is running.

So, wherever your application is running inside that server, if there are any environment properties

define those values you can access with the help of these environment interface.

So, wherever you want to read these environment properties inside your code, first you need to

autowire this environment interface to your class.

Post that inside any method where you want to read the property you can use getProperty() method available inside this interface and to this method you need to pass, what is the environment variable property name.

The next approach that we have is, with the help of @Configuration properties.

So, if you see the very first and second approach, there are two drawbacks.

The very first one is you are going to hardcode your property key name inside your Value annotation

or inside your Environment.getProperty() method and these methods there are only useful to read a

single property at a time.

If you have multiple properties, then you need to define multiple Java fields with the help @Value.

And similarly you need to invoke this getProperty() methods inside environment interface multiple times.

So, these one and two approaches we should use only if we have very few properties that we want to use.

But if you have many properties configured for your application, then using these third approach is

the most recommended approaches.

This way, using this approach, you can avoid hardcoding the property keys.

So, this approach is, with the help of the annotation at @Configuration properties.

How this approach is going to work is you need to define all your properties inside your property file

with a prefix value.

So, once we define all your properties with the same prefix value you need to use @Configuration

properties annotation on top of a Java class along with the what is a prefix value.

Once we mention this annotation on top of a class inside the class, you can create any number of fields along with the getters and setters, but please note that the field names and their return type has to be matched with what you have configured inside your application.property file or in any other location.

Reading configurations using @Value annotation

Modify the image version:   
D:\Experiments\Microservices\sb-bank-application\cards\pom.xml  
<plugin>

        <groupId>com.google.cloud.tools</groupId>

    <artifactId>jib-maven-plugin</artifactId>

        <version>3.4.1</version>

        <configuration>

                  <to>

                    <image>nileshzarkar/${project.artifactId}:s6</image>

                  </to>

        </configuration>

</plugin>

Copy the same plugin details and configure it in accounts and loans microservice  
Delete the Dockerfile from accounts microservice – not mandatory

D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application.properties

Add this property  
build.version=1.0

We will now build a REST endpoint to check the version of the API

package com.eazybytes.accounts.controller;

@RestController  
~~@AllArgsConstructor~~

@RequestMapping(path = "/api", produces = "application/json")

@Validated

public class AccountsController {

~~private IAccountsService iAccountsService;~~

    private final IAccountsService iAccountsService;

    public AccountsController(IAccountsService iAccountsService) {

        this.iAccountsService = iAccountsService;

    }

    @Value("${build.version}")

    private String buildVersion;

   …

    @GetMapping("/build-info")

    public ResponseEntity<String> getBuildInfo() {

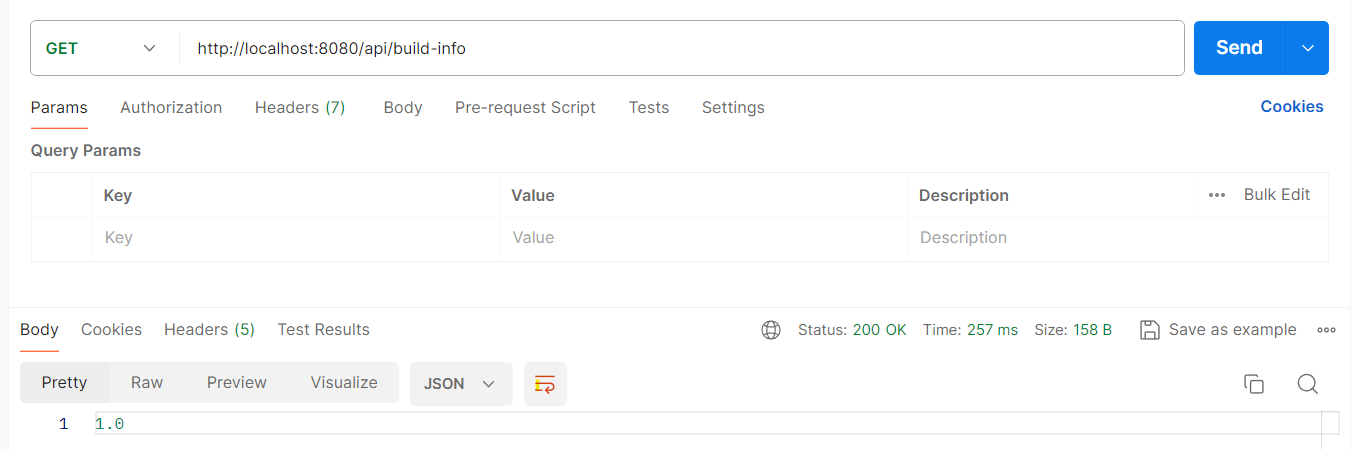
        return ResponseEntity

                .status(HttpStatus.OK)

                .body(buildVersion);

    }

}



Reading configurations using Environment interface

import org.springframework.core.env.Environment;

@RestController

@RequestMapping(path = "/api", produces = "application/json")

@Validated

public class AccountsController {

…

    @Autowired

    private Environment environment;

…

    @GetMapping("/java-version")

    public ResponseEntity<String> getJavaVersion() {

        return ResponseEntity

                .status(HttpStatus.OK)

                .body(environment.getProperty("JAVA\_HOME"));

    }

}

Reading configurations using @ConfigurationProperties

D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application.properties  
# For @ConfigurationProperties

accounts.message=Welcome to EazyBank accounts related loacal APIs

accounts.contactDetails.name=John Joe - Developer

accounts.contactDetails.email=john@eazybank.com

accounts.onCallSupport=-(555)555-1234, -(555)523-1345

package com.eazybytes.accounts.dto;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "accounts")

public record AccountsContactInfoDto(String message, Map<String, String> contactDetails, List<String> onCallSupport) {

}

package com.eazybytes.accounts;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

import com.eazybytes.accounts.dto.AccountsContactInfoDto;

@SpringBootApplication

@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")

@EnableConfigurationProperties(value = AccountsContactInfoDto.class)

public class AccountsApplication {

    public static void main(String[] args) {

        SpringApplication.run(AccountsApplication.class, args);

    }

}

package com.eazybytes.accounts.controller;

@RestController

@RequestMapping(path = "/api", produces = "application/json")

@Validated

public class AccountsController {

    @Autowired

    private AccountsContactInfoDto accountsContactInfoDto;

    @GetMapping("/contact-info")

    public ResponseEntity<AccountsContactInfoDto> getContactInfo() {

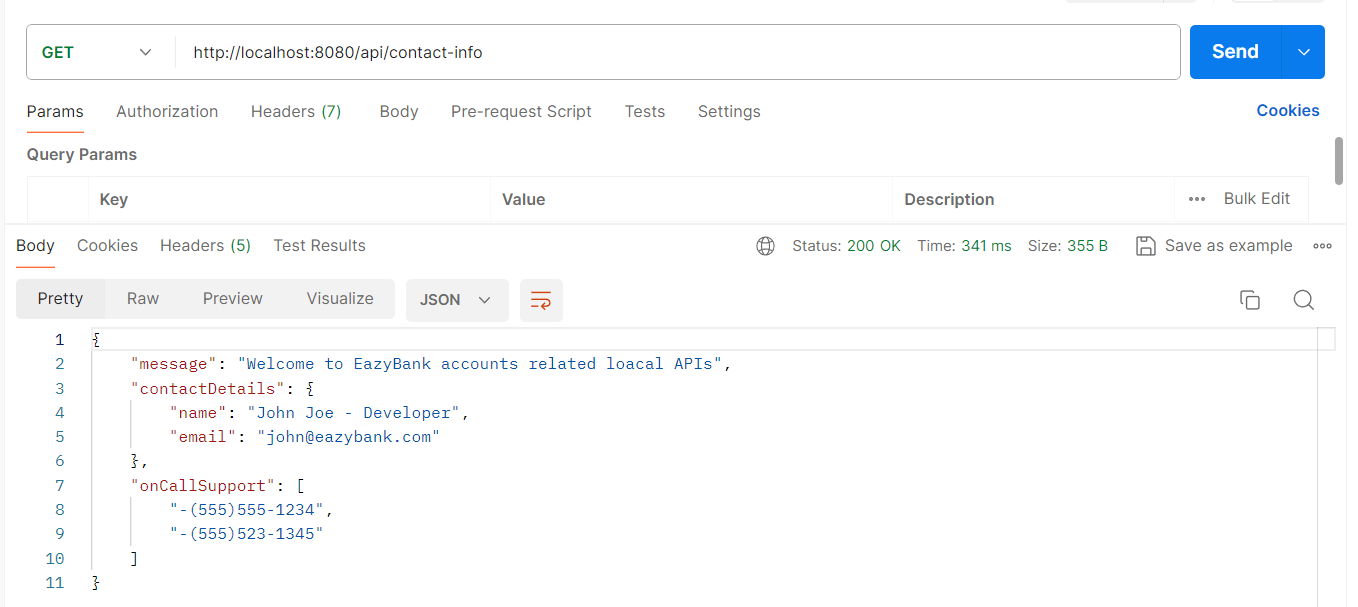
        return ResponseEntity

                .status(HttpStatus.OK)

                .body(accountsContactInfoDto);

    }

}



Introduction to Spring Boot profiles

As of now, we have built these properties inside accounts, microservice.

The challenge that right now we are facing is, if you try to deploy this code into various environments, the set of properties they are going to use will be same.

What if we have an requirement where we should use different property values inside different, different environments?

You will be seeing such requirements very often inside real projects.

For example, take database credentials itself, the properties related to database credentials.

They should not have the same values inside all the environments. Based upon the environment,

they should have different, different properties.

So, let us see how to overcome this challenge and how Spring Boot is going to help in this scenario.

Inside Spring Boot, there is a beautiful concept called profiles.

So, let us try to understand what are these profiles.

Spring Boot provides a great tool for grouping your configurations and properties into so-called profiles, which means you can create different set of files and properties that will get activated based upon the current executing environment.

If you create three different profiles like dev, QA and prod, this will give you a flexibility to you to

have different values for your configurations and the same will be activated based upon the current

active profile or based upon the current executing environment.

Using these profiles, we can perfectly set up our application that will run in different environments

with the same code, but our application is going to use different properties or configurations.

Using the same profiles also, we can control the bean creation process.

We can write logic such a way that my bean must be created only when a particular profile is activated.

This way, this profiles concept inside Spring boot it can influence the application properties that will load and the beans that will get created inside the spring context.

So, let us try to understand more details about these profiles by default inside Spring Boot framework, the default profile is always active.

Whatever properties and configurations you have defined inside your application.properties or applications.yml file, they will go into the default profile and these profile will always be activated by

default and based upon your requirements, you can create another profiles by creating property files

or yaml files by following a naming convention like you can see if I want to create two more profiles

for the production environment and the QA environment, I can create the files like application\_prod.properties or application\_prod.properties.

Since in our case we are using Yaml extension, we should create files with the extension.yml.

Once we create these two extra profile files, then inside our application there will be total three

profiles.

One is default profile that you can use inside your local development and the second one will be QA profile and the third one will be production profile.

This way you can create any number of profiles based upon your business requirements.

So now the very next question that you will be having is, how can I activate a specific profile?

We can easily activate a profile with the help of this property available inside the springboard.

So this property is spring.profiles.active and to this property key you need to pass the value

of your profile.

Since I am passing here prod, this means my production related profile will get activated.

Along with that, all the properties and configurations defined inside our application\_prod.properties

or yml file will get considered by the spring boot application.

And if needed you can activate multiple profiles also with the help of comma separated values.

By following these spring boot profiles.

We do not have to rebuild our code and regenerate the Docker image or the software package for every environment because we are going to have the properties and configurations related to all the environments inside the code base itself and we can activate these profiles based upon our requirements.

So, I just wanted to highlight always please remember that once the application is built and packaged, it should not be modified at any cost.

Whenever you are trying to move your application from one environment to other environment, it might be a feasible option inside monolithic application.

But inside microservices, building your application again and again for different environments is not

going to be a feasible option and it is going to be super, super complex and cumbersome process.

With these profiles to some extent, we can configure all the properties related to all the environments inside our application code itself and based upon our requirements, we can activate a specific profile.

But if there is a scenario where you cannot maintain certain sensitive credentials or properties inside

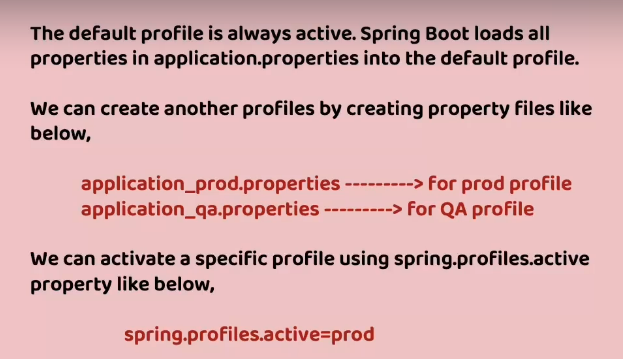
your properties file, then they should be provided externally during the startup of the

application. We will explore what are the various options that we must provide properties externally

during the startup of the application.

But for now, let us try to update our accounts microservice, by creating various profiles for various

environments like QA and production.



Demo of Spring Boot profiles inside accounts microservice

**Create a PROD profile file**D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application\_prod.properties  
spring.config.activate.on-profile=prod

# All demo purpose

# For @Value annotation

build.version=1.0

# For @ConfigurationProperties

accounts.message=Welcome to EazyBank accounts related loacal PROD APIs

accounts.contactDetails.name=Jack Millis - Product Owner

accounts.contactDetails.email=jack@eazybank.com

accounts.onCallSupport=-(555)555-1111, -(555)523-2222

**Create a QA profile file**D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application\_qa.properties  
spring.config.activate.on-profile=qa

# All demo purpose

# For @Value annotation

build.version=2.0

# For @ConfigurationProperties

accounts.message=Welcome to EazyBank accounts related loacal QA APIs

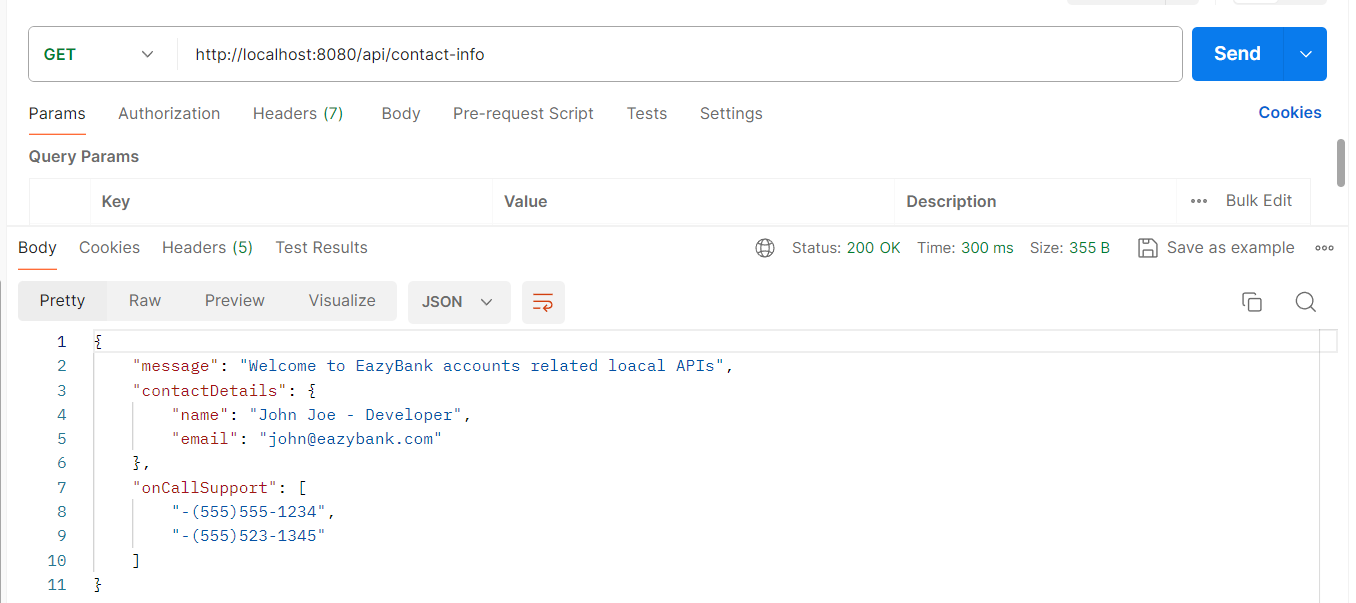
accounts.contactDetails.name=Paolo Vinci - QA Lead

accounts.contactDetails.email=paolo@eazybank.com

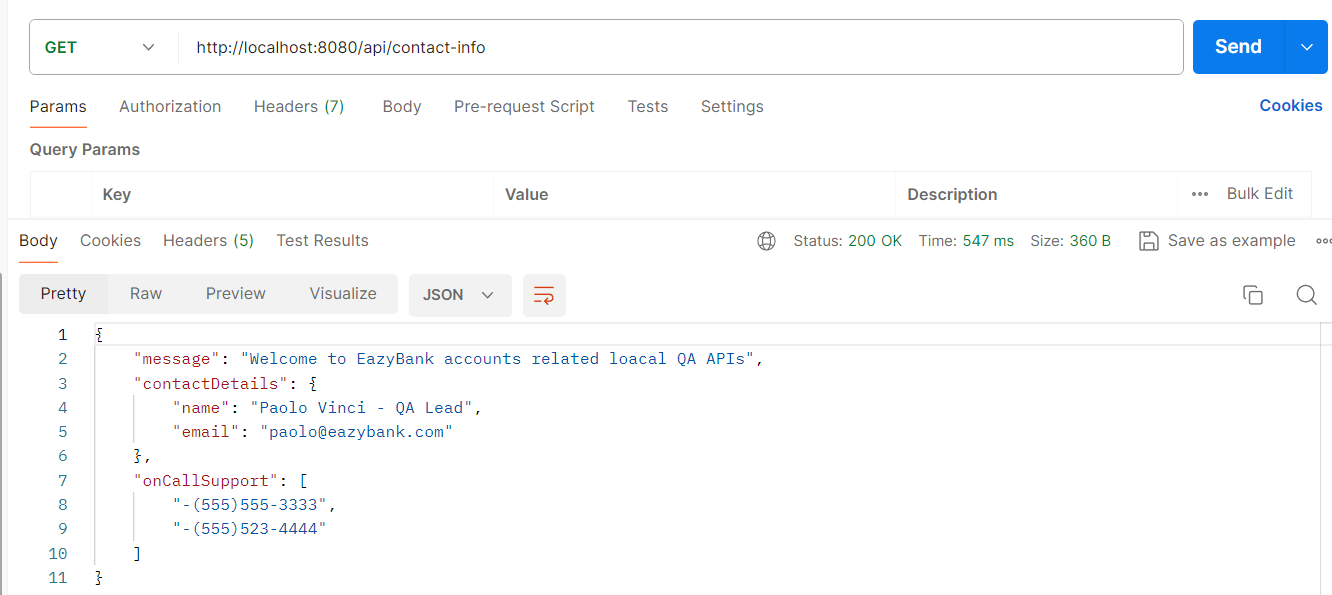
accounts.onCallSupport=-(555)555-3333, -(555)523-4444

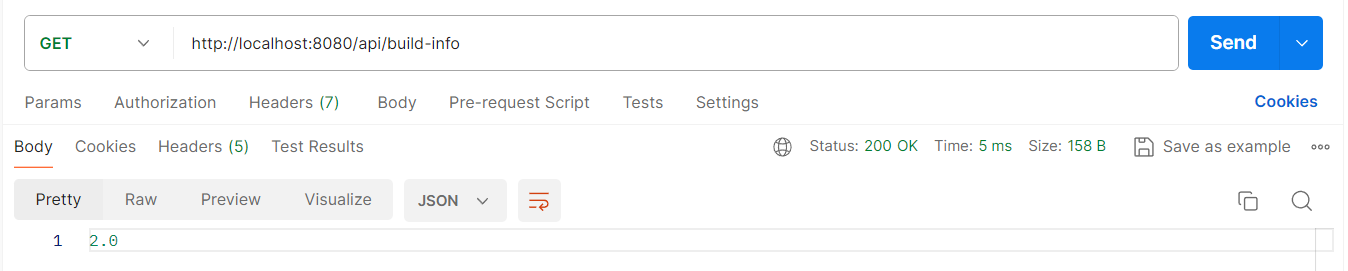
**Modifications in application.properties**….  
spring.config.import=application\_qa.properties, application\_prod.properties

….  
  
Run the application with default profile

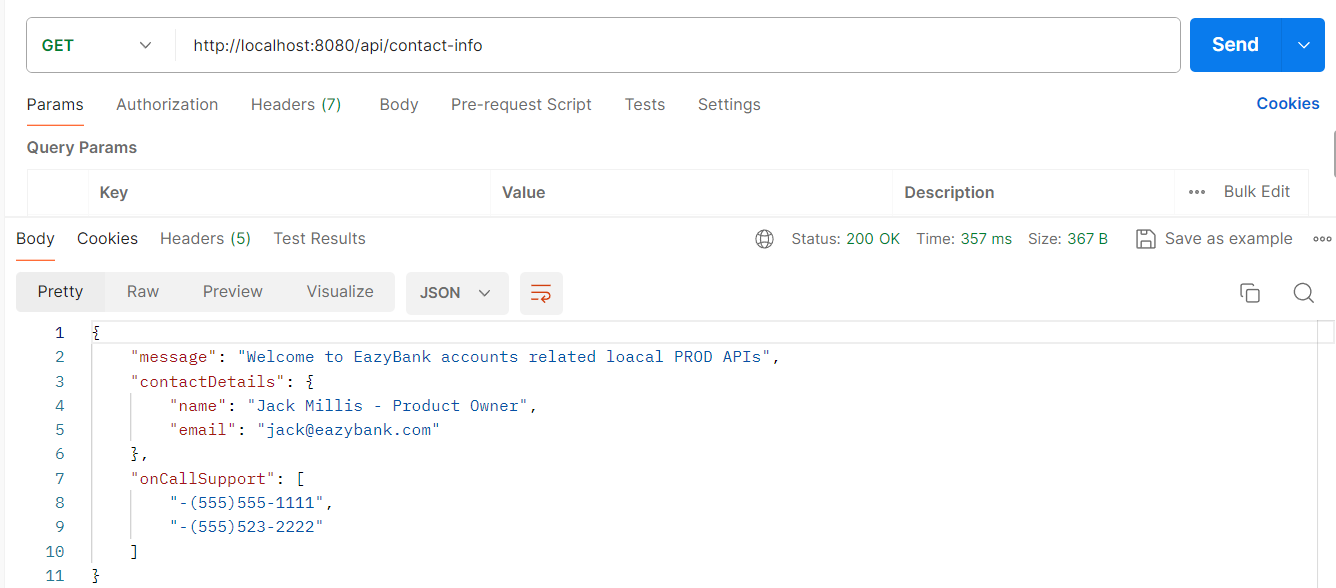


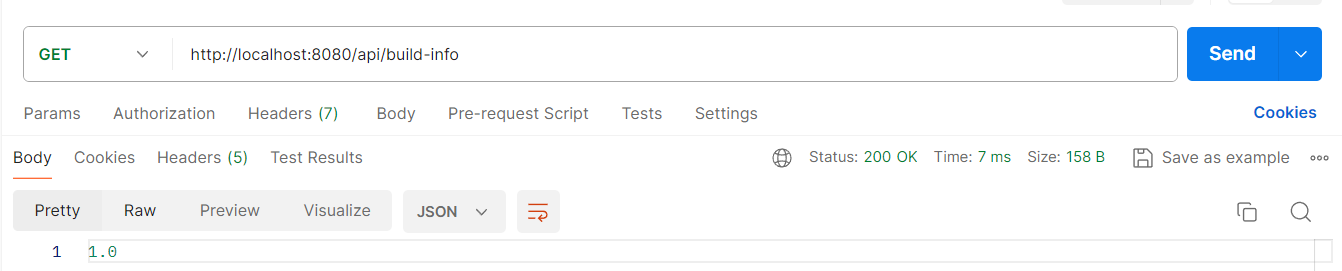
To activate the QA profile  
D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application.properties  
….  
spring.profiles.active=qa

….  
  
Run the application with configured profile  




To activate the PROD profile  
D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application.properties  
….  
spring.profiles.active=prod

….  




Externalizing configurations using command-line JVM & environment options

Activating the spring boot profile through external location or command.  
  
So let us try to understand the same Inside this lecture, Spring boot provides various ways to externalize our configurations and activate them.

And inside these approaches,

the very first mostly used approach is with the help of command line arguments.

Whenever you are trying to provide a given property along with its value, with the help of command

line arguments behind the scenes, Spring boot automatically converts that command line arguments into a key and value pairs and add them to the environment object.

Whenever you use these command line arguments, it is going to have highest precedence compared to other approaches.

Even if you define same property inside your application.yml or any other profile file, all those

values will be overridden because whatever you have defined inside the command line arguments is going to have highest precedence.

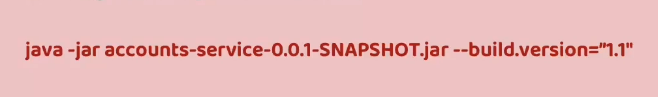
There are multiple ways on how we can pass this command line arguments we saw previously.

We can try to start our spring boot application with a fat jar that got generated inside the target

folder.

For the same, we are going to use the java-jar and what is our jar name?

And this will start our spring boot application or your microservices application. To this command only if you want to provide command line arguments, you need to follow a syntax which is by mentioning the prefix two hyphens followed by what is your property key name and what is its value.



So here the property that I want to override are the property that I want to provide.

Using external configuration like command line arguments is build.version.

That is why I am trying to mention this property along with the required value.

Similarly, you can pass any number of properties to this command by following the same prefix, but

please make sure your command line arguments are separated with a space.

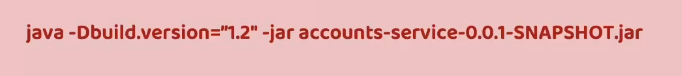
Just like how we can provide externalized configurations with the help of command line arguments, we can also provide by using the JVM system properties.

This JVM system properties has less precedence compared to the command line arguments, but it has more precedence compared to the normal property files like application.yml file.

So, let us try to understand how to provide these JVM properties.

What is the syntax.

To these JVM system properties are very similar to command line arguments.



They can override your spring boot to properties which has a lower priority, but the syntax is going

to be different for JVM system properties.

Using JVM system properties, you need to make sure you are having -d as a prefix before your

property.

You can see here I have provided an example for your reference.

But you may have a question like what if I mention the same property both in command line arguments and JVM system properties?

In such scenarios the precedence are the preference followed by the spring boot will come into picture since the command line arguments is going to have the highest precedence.

Whatever value you have mentioned through the command line arguments will be considered by the spring boot application.

By taking the precedence or the JVM properties.

The next common approach to provide externalized configurations is with the help of environment variables.

This approach has an advantage compared to the other approaches.

The advantage is environment variables are widely used for externalized configurations as they are universally supported.

Regardless of your using Java or Spring Boot, regardless of whatever language or whatever

platform you are using, these environment variables are universally supported, so even Java and Spring Boot also supports these environment variables.

That is why if you have a scenario where you are not using JVM or Spring Boot or Java, but for your

microservice or for your serverless application, if you want to still provide an external configuration,

then this approach is recommended, which is environment variables.

We can also access these environment variables inside Java code with the help of system.getEnvironment() method.

Now let us try to see what is the syntax that we need to follow while we are trying to use this environment variables.

The syntax is first, you need to convert all your property key names to the uppercase values.

They should not be any lowercase values.

All the letters should be in uppercase. And post that,

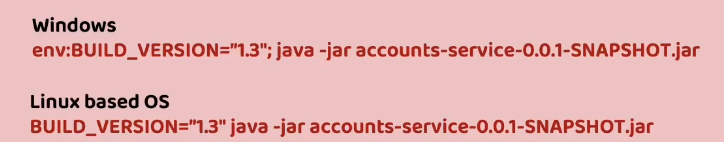
replace all your dots with underscore. For example, if you want to

provide the property of build.version as an environment variable, then you need to make sure you

are following these standards.

With that, you will convert all the letters inside the build.version into capital letters post

that you will replace that dot with underscore.



Let us see an example here.

Inside Windows,

whenever you are trying to set an environment variable through command line, first you need to execute this command which is env colon and what is your environment variable post that mention colon followed by your Java command to start your application and inside Linux based operating system or inside Mac based operating system, you can directly mention what is your environment variable. After that, please give a space.

Activating the profile using command-line, JVM and environment options

Program arguments  
VSCode 🡺 Command Palette 🡺 launch.json  
{

"version": "0.2.0",

"configurations": [

{

"type": "java",

"name": "Current File",

"request": "launch",

"mainClass": "${file}"

},

{

"type": "java",

"name": "AccountsApplication",

"request": "launch",

**"mainClass": "com.eazybytes.accounts.AccountsApplication",**

"projectName": "accounts",

**"args": "--spring.profiles.active=qa"**

},

...

]

}

Run the accounts application as usual using VSCode IDE.

JVM arguments  
VSCode 🡺 Command Palette 🡺 launch.json  
{

"version": "0.2.0",

"configurations": [

{

"type": "java",

"name": "Current File",

"request": "launch",

"mainClass": "${file}"

},

{

"type": "java",

"name": "AccountsApplication",

"request": "launch",

**"mainClass": "com.eazybytes.accounts.AccountsApplication",**

"projectName": "accounts",

**"vmArgs": "-Dspring.profiles.active=qa -Dbuild.version=9.9"**

},

...

]

}

Run the accounts application as usual using VSCode IDE.

Env Variables arguments  
VSCode 🡺 Command Palette 🡺 launch.json  
{

"version": "0.2.0",

"configurations": [

{

"type": "java",

"name": "Current File",

"request": "launch",

"mainClass": "${file}"

},

{

"type": "java",

"name": "AccountsApplication",

"request": "launch",

**"mainClass": "com.eazybytes.accounts.AccountsApplication",**

"projectName": "accounts",

**"env": {**

**"spring.profiles.active": "prod",**

**"build.version": "9.7"**

**}**

},

...

]

}

Assignment to make Spring Boot profile changes inside loans & cards microservices

**LOANS CONFIGURATION READING**Reading Configuration using @Value annotation  
D:\Experiments\Microservices\sb-bank-application\loans\src\main\resources\application.properties  
….  
build.version=5.0

package com.eazybytes.loans.controller;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})  
~~@AllArgsConstructor~~

@Validated

public class LoansController {

~~private ILoansService iLoansService;~~

private final ILoansService iLoansService;

public LoansController(ILoansService iLoansService) {

this.iLoansService = iLoansService;

}

@Value("${build.version}")

private String buildVersion;

@GetMapping("/build-info")

public ResponseEntity<String> getBuildInfo() {

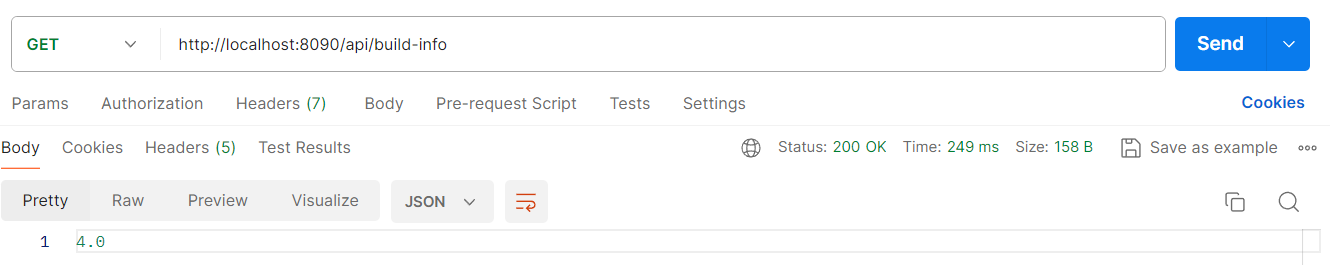
return ResponseEntity

.status(HttpStatus.OK)

.body(buildVersion);

}

}



Reading Configuration using Environment interface  
package com.eazybytes.loans.controller;

import org.springframework.core.env.Environment;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})

@Validated

public class LoansController {

….

    @Autowired

    private Environment environment;

@GetMapping("/java-version")

    public ResponseEntity<String> getJavaVersion() {

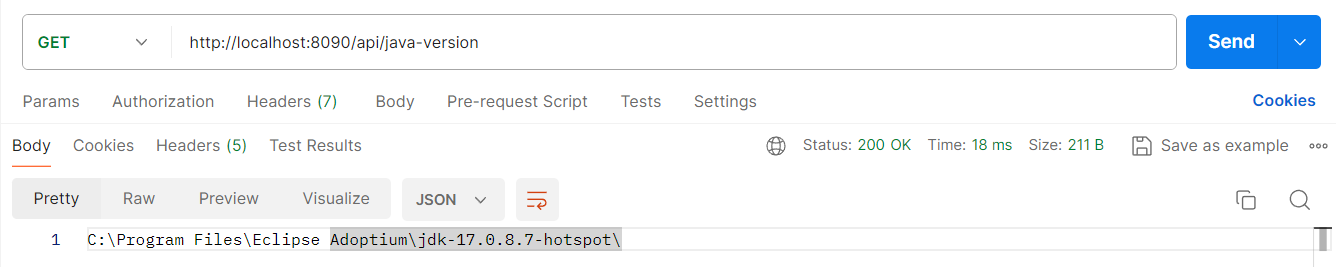
        return ResponseEntity

                .status(HttpStatus.OK)

                .body(environment.getProperty("JAVA\_HOME"));

    }

}



**Modifications in application.properties** D:\Experiments\Microservices\sb-bank-application\loans\src\main\resources\application.properties  
# For @ConfigurationProperties

loans.message=Welcome to EazyBank loans related local APIs

loans.contactDetails.name=Lopez Derick - Developer

loans.contactDetails.email=lopez@eazybank.com

loans.onCallSupport=-(555)555-2222, -(555)523-3333

package com.eazybytes.loans.dto;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "loans")

public record LoansContactInfoDto(String message, Map<String, String> contactDetails, List<String> onCallSupport) {

}

package com.eazybytes.loans;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

import com.eazybytes.loans.dto.LoansContactInfoDto;

@SpringBootApplication

@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")

@EnableConfigurationProperties(value = LoansContactInfoDto.class)

public class LoansApplication {

    public static void main(String[] args) {

        SpringApplication.run(LoansApplication.class, args);

    }

}

package com.eazybytes.loans.controller;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})

@Validated

public class LoansController {

@Autowired

private LoansContactInfoDto loansContactInfoDto;

@GetMapping("/contact-info")

public ResponseEntity<LoansContactInfoDto> getContactInfo() {

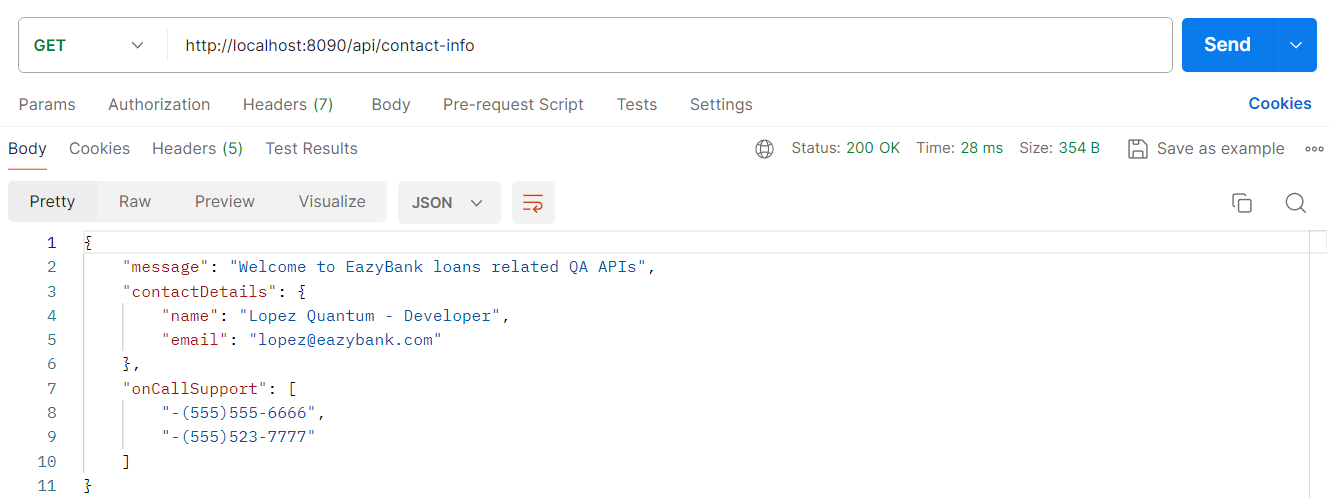
return ResponseEntity

.status(HttpStatus.OK)

.body(loansContactInfoDto);

}

}



**LOANS PROFILE SETTINGS**Create a PROD profile file   
D:\Experiments\Microservices\sb-bank-application\loans\src\main\resources\application\_prod.properties  
# Spring Profile related changes

spring.config.activate.on-profile=prod

# All demo purpose

# For @Value annotation

build.version=3.0

# For @ConfigurationProperties

loans.message=Welcome to EazyBank loans related PROD APIs

loans.contactDetails.name=Lopez Patrik - Developer

loans.contactDetails.email=lopez@eazybank.com

loans.onCallSupport=-(555)555-4444, -(555)523-5555

Create a QA profile file   
D:\Experiments\Microservices\sb-bank-application\loans\src\main\resources\application\_qa.properties  
# Spring Profile related changes

spring.config.activate.on-profile=qa

# All demo purpose

# For @Value annotation

build.version=4.0

# For @ConfigurationProperties

loans.message=Welcome to EazyBank loans related QA APIs

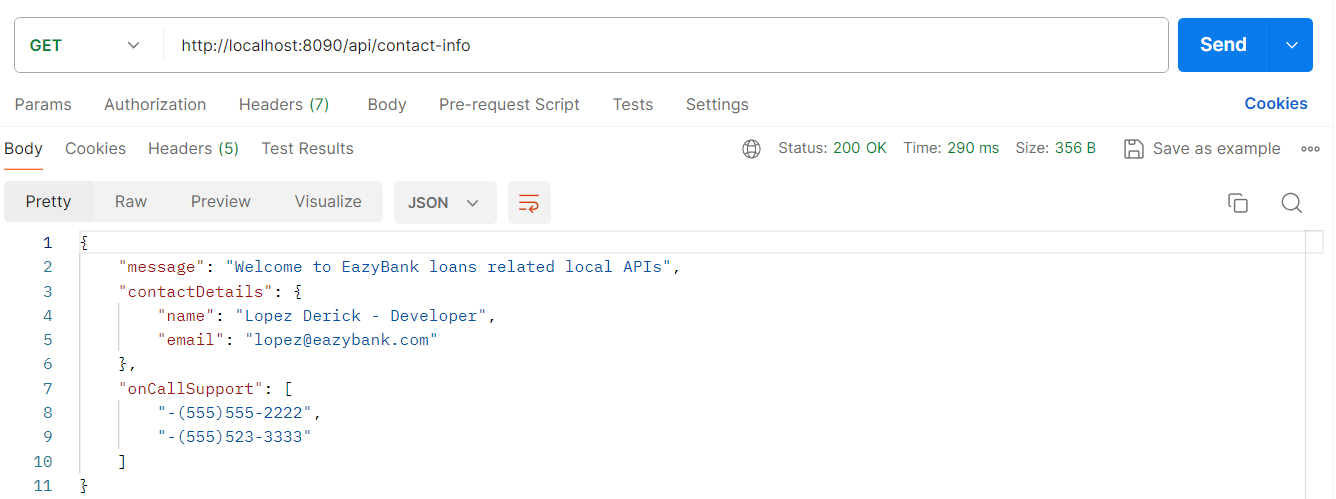
loans.contactDetails.name=Lopez Quantum - Developer

loans.contactDetails.email=lopez@eazybank.com

loans.onCallSupport=-(555)555-6666, -(555)523-7777

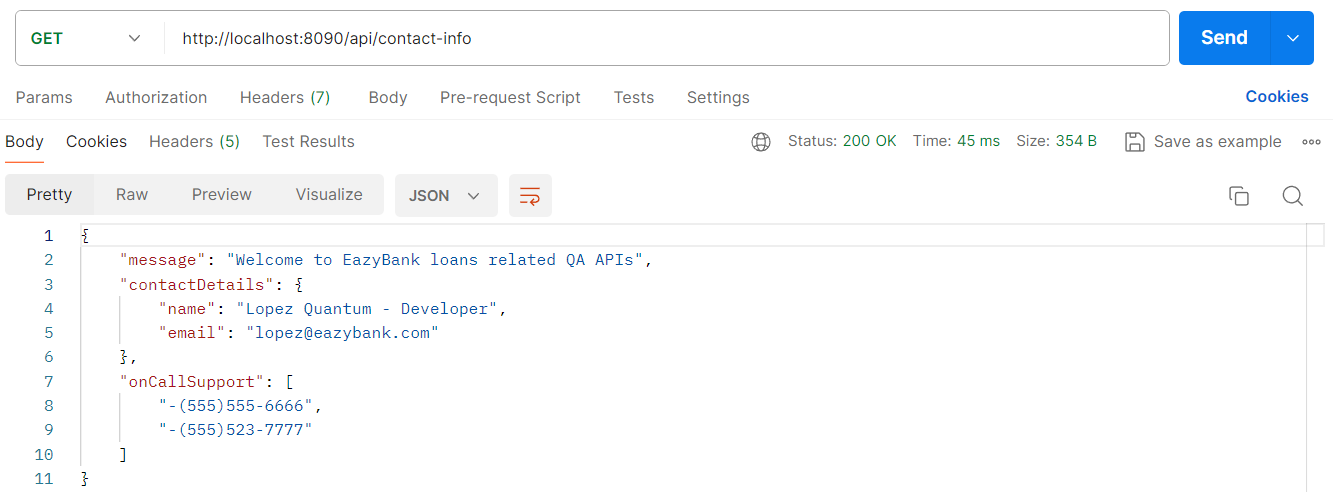
Modifications in application.properties  
# Spring Profile related changes

spring.config.import=application\_qa.properties, application\_prod.properties



To activate the QA profile  
D:\Experiments\Microservices\sb-bank-application\loans\src\main\resources\application.properties

….  
spring.profiles.active=qa

….  


**CARDS  
CARDS CONFIGURATION READING**Reading Configuration using @Value annotation  
D:\Experiments\Microservices\sb-bank-application\cards\src\main\resources\application.properties  
….  
build.version=5.0

package com.eazybytes.cards.controller;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})  
~~@AllArgsConstructor~~

@Validated

public class CardsController {

~~private ICardsService iCardsService;~~

private final ICardsService iCardsService;

public CardsController(ICardsService iCardsService) {

this.iCardsService = iCardsService;

}

@Value("${build.version}")

private String buildVersion;

@GetMapping("/build-info")

public ResponseEntity<String> getBuildInfo() {

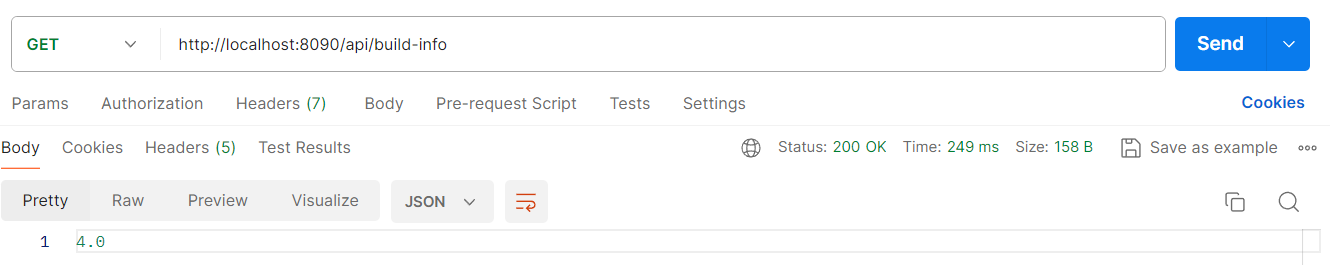
return ResponseEntity

.status(HttpStatus.OK)

.body(buildVersion);

}

}



Reading Configuration using Environment interface  
package com.eazybytes.cards.controller;

import org.springframework.core.env.Environment;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})

@Validated

public class CardsController {

….

    @Autowired

    private Environment environment;

@GetMapping("/java-version")

    public ResponseEntity<String> getJavaVersion() {

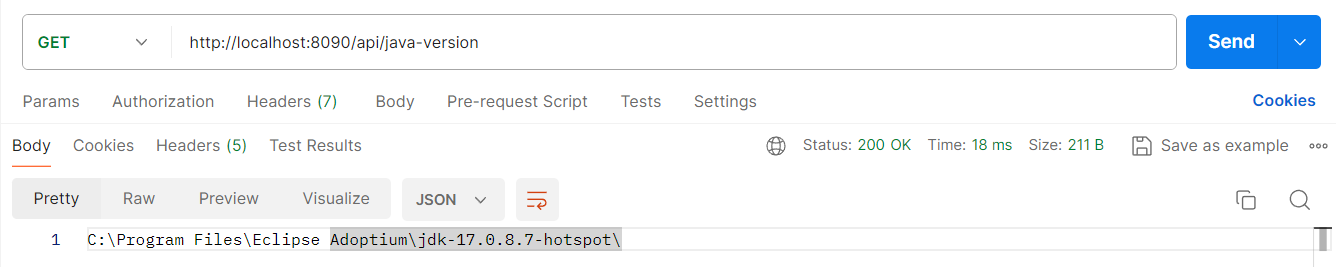
        return ResponseEntity

                .status(HttpStatus.OK)

                .body(environment.getProperty("JAVA\_HOME"));

    }

}



**Modifications in application.properties** D:\Experiments\Microservices\sb-bank-application\cards\src\main\resources\application.properties  
# For @ConfigurationProperties

cards.message=Welcome to EazyBank cards related local APIs

cards.contactDetails.name=Lopez Derick - Developer

cards.contactDetails.email=lopez@eazybank.com

cards.onCallSupport=-(555)555-2222, -(555)523-3333

package com.eazybytes.cards.dto;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "cards")

public record CardsContactInfoDto(String message, Map<String, String> contactDetails, List<String> onCallSupport) {

}

package com.eazybytes.cards;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

import com.eazybytes.cards.dto.CardsContactInfoDto;

@SpringBootApplication

@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")

@EnableConfigurationProperties(value = CardsContactInfoDto.class)

public class CardsApplication {

    public static void main(String[] args) {

        SpringApplication.run(CardsApplication.class, args);

    }

}

package com.eazybytes.cards.controller;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})

@Validated

public class CardsController {

@Autowired

private CardsContactInfoDto cardsContactInfoDto;

@GetMapping("/contact-info")

public ResponseEntity<CardsContactInfoDto> getContactInfo() {

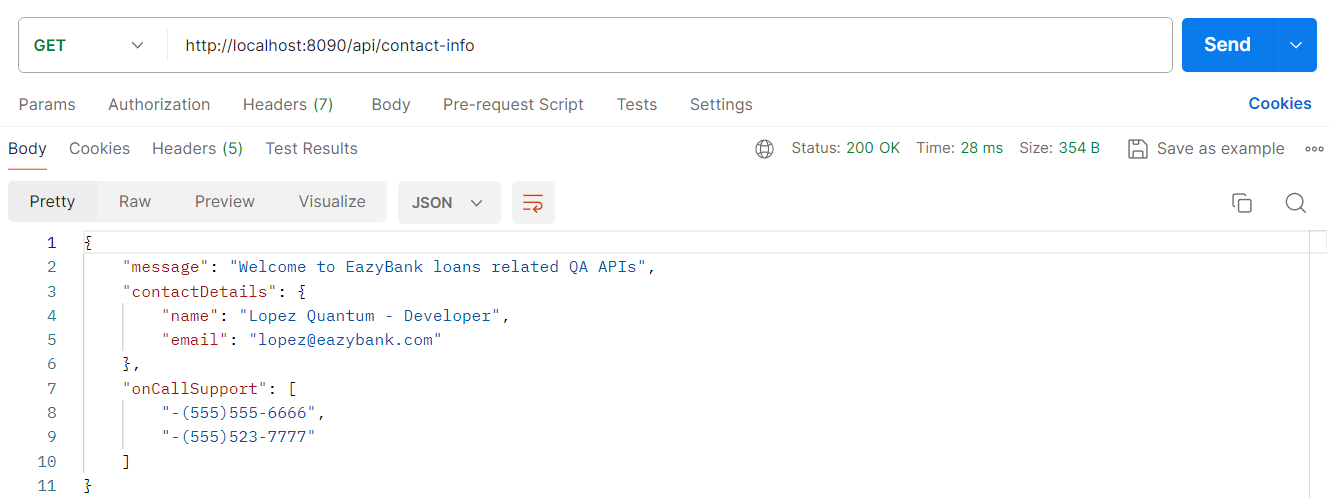
return ResponseEntity

.status(HttpStatus.OK)

.body(cardsContactInfoDto);

}

}



**CARDS PROFILE SETTINGS**Create a PROD profile file   
D:\Experiments\Microservices\sb-bank-application\cards\src\main\resources\application\_prod.properties  
# Spring Profile related changes

spring.config.activate.on-profile=prod

# All demo purpose

# For @Value annotation

build.version=3.0

# For @ConfigurationProperties

cards.message=Welcome to EazyBank cards related PROD APIs

cards.contactDetails.name=Lopez Patrik - Developer

cards.contactDetails.email=lopez@eazybank.com

cards.onCallSupport=-(555)555-4444, -(555)523-5555

Create a QA profile file   
D:\Experiments\Microservices\sb-bank-application\cards\src\main\resources\application\_qa.properties  
# Spring Profile related changes

spring.config.activate.on-profile=qa

# All demo purpose

# For @Value annotation

build.version=4.0

# For @ConfigurationProperties

cards.message=Welcome to EazyBank cards related QA APIs

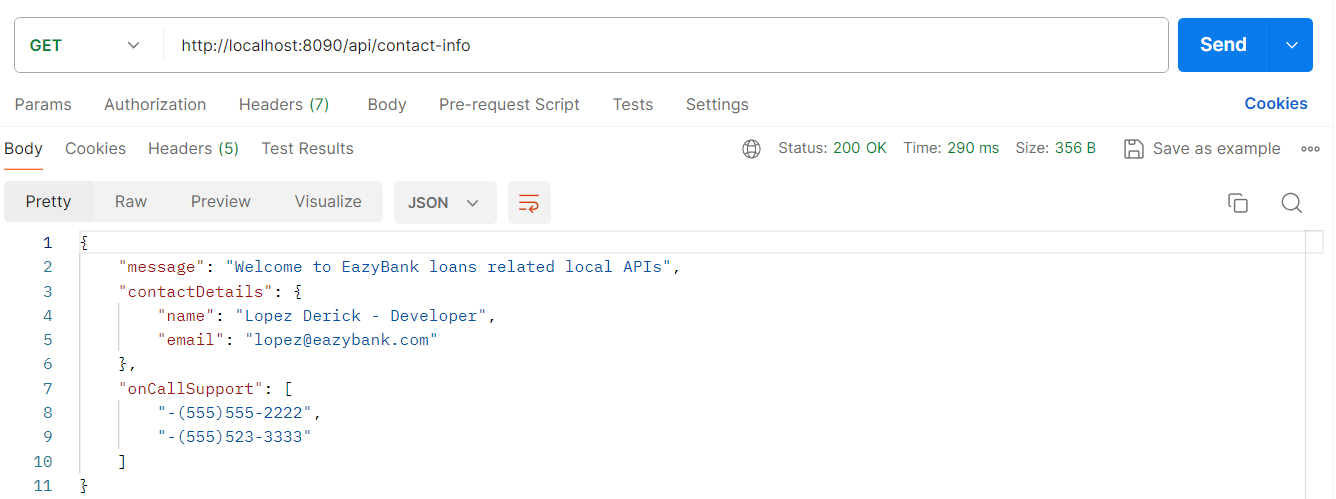
cards.contactDetails.name=Lopez Quantum - Developer

cards.contactDetails.email=lopez@eazybank.com

cards.onCallSupport=-(555)555-6666, -(555)523-7777

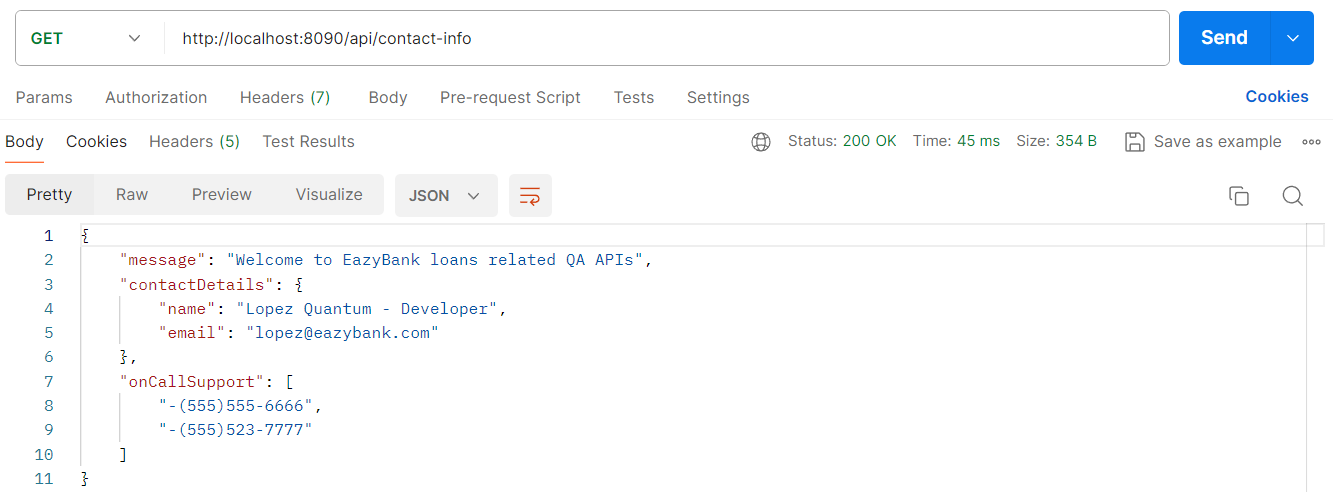
Modifications in application.properties  
# Spring Profile related changes

spring.config.import=application\_qa.properties, application\_prod.properties



To activate the QA profile  
D:\Experiments\Microservices\sb-bank-application\cards\src\main\resources\application.properties

….  
spring.profiles.active=qa

….  


Drawbacks of externalized configurations using Spring Boot alone

Introduction to Spring Cloud Config

We have a lot of limitations and drawbacks which we discussed in the previous lecture.

Now, inside this lecture, let me try to introduce a better option or the recommended approach that

you 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 you 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, your centralized configuration server is going to provide server and client-side support for externalized configurations in a distributed system like microservices,

which means all your 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.

For now, please note that this configuration server which we are going to build with the help of spring cloud config, it is going to act as a central place to manage all your external properties or configurations for all your microservices across all the environments.

The centralized config server that we are going to build with the help of spring cloud config revolves

around two core elements.

The very first one is you are free to store all your configurations or property files inside any location,

like you can store inside a GitHub repo or inside a file system or inside a database.

You choose a location where you want to store all your configurations and properties securely.

Once you store all your properties or configurations, then this configuration server is going

to oversee the configuration data within the data store, facilitating its management and distributing

to multiple applications like microservices.

So, let us try to understand more details about these two core elements.

Like I said, in the very first step, you can save all your configurations and properties inside a

centralized repositories, and this repository can be a database or a GitHub repo or a file system or

inside your class path and there are many other options that spring cloud config server supports to store your configuration properties.

I will be showing all those options by looking at the official documentation of spring cloud config server.

So, once we store all these configuration properties inside a centralized repository. As a next step,

you 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 your centralized repository.

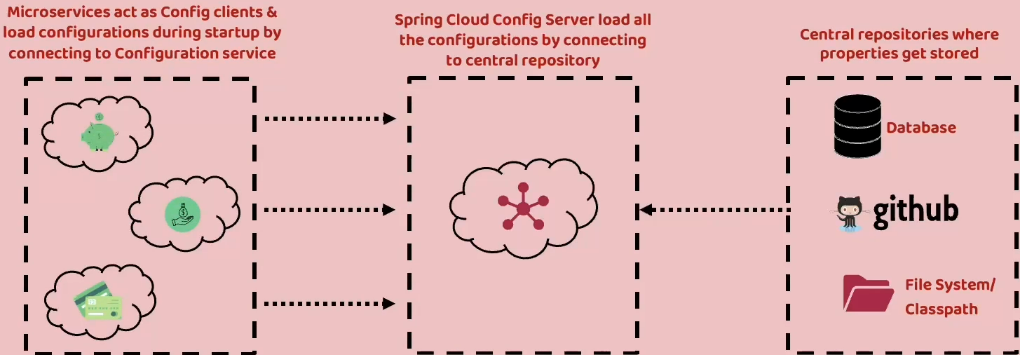
So now you are config server holds the properties of all the microservices and environments.

Now as a next step, our individual microservices, they can connect with this config server as a config

clients 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 you can see, there are three microservices like loans, accounts, and cards, all of them during

the startup

They will connect to this centralized configuration server.

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 that we have discussed in the previous lecture.

Building Config Server using Spring Cloud Config

pom.xml  
…  
<parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

**<version>3.2.7-SNAPSHOT</version>**

        <relativePath/> <!-- lookup parent from repository -->

    </parent>

    <groupId>com.eazybytes</groupId>

    <artifactId>configserver</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>configserver</name>

    <description>Demo project for Spring Boot</description>

…  
<properties>

        <java.version>17</java.version>

**<spring-cloud.version>2023.0.1</spring-cloud.version>**

</properties>

…

<dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-actuator</artifactId>

        </dependency>

    <dependency>

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

            <artifactId>spring-cloud-config-server</artifactId>

        </dependency>

…

</dependencies>  
  
<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>  
….

You can create this pom from spring initializer.  
Check the spring boot and spring cloud compatibility release train   
<https://spring.io/projects/spring-cloud>

package com.eazybytes.configserver;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

@EnableConfigServer

public class ConfigserverApplication {

    public static void main(String[] args) {

        SpringApplication.run(ConfigserverApplication.class, args);

    }

}

D:\Experiments\Microservices\sb-bank-application\configserver\src\main\resources\application.properties

spring.application.name=configserver

# default port of config server

server.port=8071

Reading configurations from the class path location of Config Server

Updating Accounts Microservice to read properties from Config Server

Updating Loans & Cards Microservice to read properties from Config Server

Reading configurations from a file systems location

Reading configurations from a GitHub repository

Encryption and Decryption of properties inside Config server

Refresh configurations at runtime using Spring Cloud Bus

Refresh config at runtime using Spring Cloud Bus & Spring Cloud Config monitor

Updating Docker compose file to adapt Config Server changes – Part 1

Introduction to Liveness and Readiness probes

Updating Docker compose file to adapt Config Server changes – Part 2

Optimizing Docker Compose file

Generating Docker images and pushing them into Docker Hub

Testing Config Server changes end to end using Docker compose & default profile

Preparing Docker Compose files for QA & prod profiles