**Configuration Management in Microservices**

The very first question that I have is, inside microservices, how we are going to separate the configurations or properties from our business logic, because without separating the configurations and properties from our microservices business logic, we cannot reuse the same Docker image across multiple environments. If we club all our business logic and configurations together, then for each environment we 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?

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 can't 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 we will be having only 1 or 2 applications, and it is very easy to maintain all our configurations wherever we want, whereas with hundreds of microservices it is going to be super complex to maintain all the properties of all the microservices manually.

That's 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. We 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 to handle this challenge.

Below are the solutions. Let’s try to identify which one suite for microservices.

1. Configuring Spring boot with properties and profiles
2. Applying external configuration with spring boot
3. Implementing a configuration server with spring cloud config server.

**How the configuration management happens inside a traditional application and microservices application.**

Back in the days when we were trying to build traditional applications or monolithic applications, all our source code along with the configuration’s files, they will be bundled together. And with that, whenever there is a different configuration is needed for a different environment. We need to rebuild the application code again with the required configurations.

So, with this there is no guarantee that the application would behave consistently because the main business logic may differ from one environment to other environment when we are trying to build the application. This approach might have worked for monolithic application because there is only one application, and they can do multiple builds based upon the environment. Whereas with microservice, since there will be hundreds of microservices doing multiple builds again and again for each environment with the required configuration data is going to be complex.

That's why as per the 15-factor methodology. All the configurations which are going to change between deployments such as credentials, service URLs or resource handles. So, all such configuration data we need to provide outside our build component. So, all such configurations that are likely to change, we need to maintain outside our business logic. So that the application artifacts will be immutable across all environments. Once we do a build and generate a Docker image for our microservice, the same image we can use across environments and the configurations must be injected from the external location.

This way the application build will remains unchanged across all the environments. Like we can see here, first there will be an microservice or cloud native application codebase inside the GitHub repo. We will do compilation and package as a build, which is going to be common for all environments. So inside this builds we may generate a Docker image for our microservice and to this build a component we are going to inject configurations based upon the target environment.

If we are trying to do a deployment into a development environment, then we can inject the configurations related to that development environment at runtime during the startup of the microservice. And both our configurations and the source code will be clubbed and deployed into that target environment like development QA and production.

**How Configurations work in Spring Boot?**

Because we are developing microservices with spring boot only. Inside java eco-system develop to develop microservices most of the developers and organizations, they are going to use spring boot framework. That's why we are also using the spring boot inside this course since we are using spring boot framework.

First let's try to understand what the options are that Spring boot framework is providing to handle the challenge of configuration management in microservices. Once we discuss these basic approaches, then we'll go to the advanced approaches that we have with the help of spring cloud config server.

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's see if the Spring Boot is going to help us in externalizing our application properties.

Spring boots externalize our configuration so that we can work with the same application code within different environments. We can use a variety of external configuration sources, include java properties files, YAML files, environment variables, and command-line arguments.

By default, spring boot looks for the configurations or properties inside application.properties/yaml present in the class path location. But we can have another property files as well and make spring boot to read from them.

Spring boot uses a very particular order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones):

* Properties present inside files like application.properties
* OS environmental variables
* Java system properties (System.getProperties())
* JNDI attributes from java:com/env
* ServelteContext init parameters.
* Command line arguments.

1. **application.properties** - The very first one is we can mention our properties inside the files like application.properties or application.yml. So, whatever we mentioned inside these files, they will have the lowest priority or the lowest preference.
2. **OS environmental variables** – 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 overrided and whatever value we have mentioned inside the environmental variables will be considered by the spring boot framework.

And very similarly, we can also mention the configurations or properties with the approaches like **Java System Properties**, **Jndi attributes** **ServelteContext 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 are going to have highest priority. Whereas the properties that we have mentioned inside the application.properties is going to have the lowest priority.

How to read those properties inside our business logic. So, let's try to understand the same here again in Spring Boot. To read the properties, there are multiple approaches.

To read the properties, there are multiple approaches. present three commonly used approaches inside this course.

1. The very first approach is with the help of @Value annotation. So, whenever we want to read a particular property, we can define a Java field inside our business logic.

On top of that Java field, we can mention an annotation **@Value** along with what is a property key name. Once we mention what is a property key name during the startup of our 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, we can use that anywhere inside our methods or inside our business logic.

1. 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 our 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 our application is running. So, wherever our application is running inside that server, if there are any environment properties define those values we can access with the help of these environment interface.

So, wherever we want to read these environment properties inside your code, first we need to **autowire** environment interface to our class. Post that inside any method where we want to read the property we can use getProperty() method available inside this interface and to this method we need to pass.

1. The next approach that we have is, with the help of @**ConfigurationProperties**. So, if we see the very first and second approach, there are two drawbacks. The very first one is we are going to hardcode our property key name inside our Value annotation or inside our Environment.getProperty() method and these methods there are only useful to read a single property at a time.

If we have multiple properties, then we need to define multiple Java fields with the help @Value. And similarly we 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 **we have many properties configured for our application, then using these third approach is the most recommended approaches**.

This way, using this approach, we 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 we need to define all our properties inside our property file with a prefix value. So, once we define all our properties with the same prefix value we need to use @Configuration properties annotation on top of a Java class along with what is a prefix value.

Once we mention this annotation on top of a class inside the class, we 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 we have configured inside our application.properties file or in any other location.

So, with this approach, by the time the server started successfully, all our property values will be binded to the fields presence inside the class my config and to access all these fields we just need to invoke the get method available for all these fields.

**Approach – 1 On how to read the properties with the help of @value annotation**

So v1 indicates version 1 – Inside this version 1 microservices inside our section 6 we are going to follow the basic approaches provided by the spring boot framework. Once we discuss all the basic approaches provided by the spring boot framework then will look for the advanced approaches inside the next version which is v2.

So, inside this v1-springboot folder we are copying all the microservices that we have in the section4. In the section6 we are going to follow a google jib approach.

Step 1 – inside the application.yml file we can define any properties along with their values which we can read inside our java code

build:  
 version: "1.0.0"

using this property read the value inside our java code and also build a small rest API which can our client applications can invoke and understand what the build version is right now our microservice is using.

Step 2 – Go to the controller class **AccountsController** -> Here, first we read the property define inside our application.yml file -> for that we create a new java field inside our controller class.

@Value("${build.version}")  
private String buildVersion;

This will have null value right now to inject the property value during the start-up of the spring boot application into this java filed we need to mention the annotation is **@value** to this annotation we need to pass the property key name by following the spring expression language @value(“${build.version:1.0.0}”) so with this changes the build version java field will have the value that we define inside the yml file.

Going to build a small REST API service to send the build version information to our client for that -> we have to go to the end of controller class.

@Operation(  
 summary = "Get Build Information",  
 description = "Get Build Information that is deployed into accounts microservice"  
 )  
 @ApiResponses({  
 @ApiResponse(  
 responseCode = "200",  
 description = "Http Status OK"  
 ),  
 @ApiResponse(  
 responseCode = "500",  
 description = "Http Status Internal Server Error",  
 content = @Content(  
 schema = @Schema(implementation = ErrorResponseDto.class)  
 )  
 )  
 }  
 )  
 @GetMapping("/build-info")  
 public ResponseEntity<String> getBuildInfo() {  
 return ResponseEntity  
 .*status*(HttpStatus.*OK*)  
 .body(buildVersion);  
 }  
}

Step 3 – We can remove the @AllArgsConstructor instead we have to create a manual constructor

private final IAccountsService iAccountsService;

public AccountsController(IAccountsService iAccountsService){  
 this.iAccountsService=iAccountsService;  
}// run the accounts microservice with the debug mode.

Step 4 – using this Api endpoint <http://localhost:8080/api/build-info> Get method to get the version response from the postman.

A screenshot of a computer

Description automatically generated

This is the version mention in the application.yml file. So, this way we can read the properties inside the application.properties file or application.yml file or inside any other file with the help of @value annotation and this allows to read all our properties from our java code itself.

This approach is sufficient for microservice? This is not going to work if we have 100s of microservices and if they have 100 different properties, we can’t crate 100s different fields inside our microservices? This is not a good solution.

Under the same time in this approach in the AccountsController while injecting java filed, we are hardcoding the property name like build.version so, creating a java filed and mentioning the hardcoded property value for every microservice is not going to be feasible option. That’s why this approach is only recommended we have only one or two properties, we have many properties this approach will not going to work.

**Approach – 2 Reading configurations using Environment interface**

Here we may have a question like why do we need to define some properties as environment variables? Why can't we directly define them inside the application Yaml file? That's a very valid question.  
Like before, some sensitive information like passwords or any other information. We can't define them inside the application.yml file or inside any other place where it will expose my sensitive information.

With that reason, it is always advisable to define the sensitive configuration details as the environment variables only, so that no one can see those values because they will not have access to our production server, only the server admins they will have access.

This way we're trying to secure our application whenever they are dealing with the sensitive configurations. Like we discussed before, this approach involves an interface with the name environment.

Step – 1: Autowire the environment interface in the accounts controller

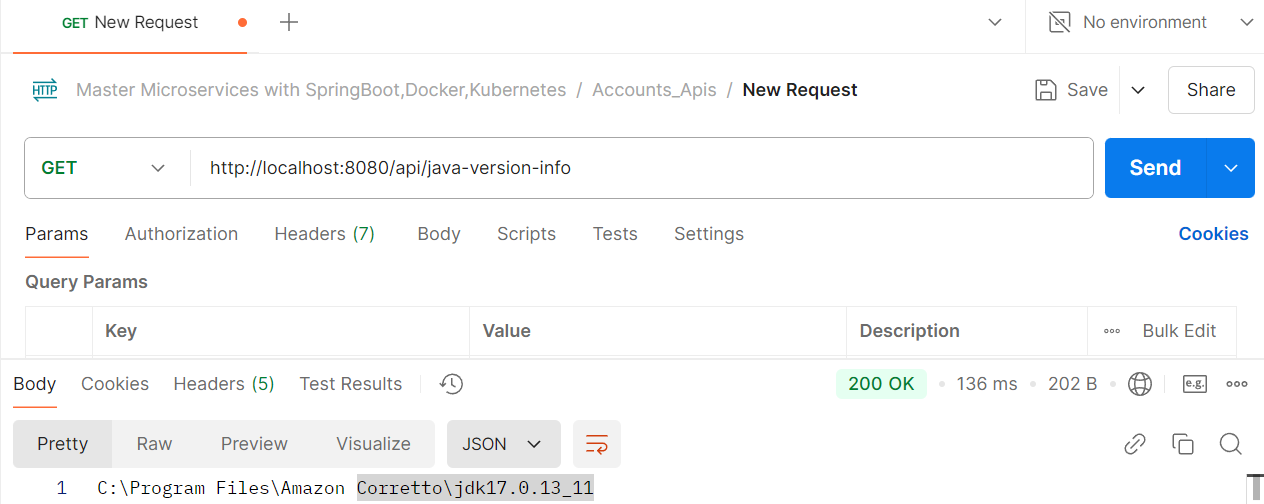
Import the Environment in the spring core framework

@Autowired  
private Environment environment;

With this now all the environment details that we have define inside our local system can be accessed with the help of this environment variable.

Step – 2: Now we are going to create a new REST API that will return what is the Java version that my microservice right now, is using. So, inside our local system or in any Java developer system will always install the Java and will set an environment variable with the name JAVA\_HOME. With the help of these environment interface, we'll try to read that environment property related to Java version and the same we'll return to the client applications whoever is trying to invoke our REST API.

@Operation(  
 summary = "Get java version",  
 description = "Get java version details that is installed in accounts microservice"  
)  
@ApiResponses({  
 @ApiResponse(  
 responseCode = "200",  
 description = "Http Status OK"  
 ),  
 @ApiResponse(  
 responseCode = "500",  
 description = "Http Status Internal Server Error",  
 content = @Content(  
 schema = @Schema(implementation = ErrorResponseDto.class)  
 )  
 )  
}  
)  
@GetMapping("/java-version-info")  
public ResponseEntity<String> getBuildInfo() {  
 return ResponseEntity  
 .*status*(HttpStatus.*OK*)  
 .body(environment.getProperty("JAVA\_HOME"));  
}



Here inside the Postman, we are going to click on this java-version-info API. Http get method. As soon as we click on the send button, I'm getting a response where my Java home is installed. So, inside the response, since we have used the SDK command to install the Java, it is simply giving the Java home folder location, but ideally inside the production servers we will directly install the JDK by downloading from the official website. And in such scenarios, it is going to give us the complete folder location along with the Java version name.

We are also having Maven installed, so whenever we have the Maven install, we are going to have the environment property set with the name Maven\_home.

@GetMapping("/java-version-info")  
public ResponseEntity<String> getBuildInfo() {  
 return ResponseEntity  
 .*status*(HttpStatus.*OK*).body(environment.getProperty("Maven\_HOME"));  
}

We can see it is giving me a location where my Apache maven is installed. So, from this folder location we can easily understand the version right now we have installed. This way we can read any type of environment property inside our Java code with the help of environment interface.

Please note that when we are using this approach, we can only read the environment properties but not the properties that we have defined inside our application.yml.

So, this approach also has disadvantages like we can only read one property at a time and at the same time we need to hardcode the property key name inside our Java code. And this approach is also going to work only if we have 1 or 2 properties. If we have higher number of environment properties that we want to read, then definitely this approach is not recommended. We'll see some advanced options that we have.

**Approach – 3: Reading configurations using @ConfigurationProperties**

Using this third approach, we can read multiple properties at a time with a single pojo class. All the limitations that we have with the previous two approaches will be handled with this third approach. Limitations are like where we are hardcoding the property key names inside the Java code. And at the same time, we can only read one property at a time with the two approaches that we have discussed. So, to demo, the third approach, we going to create a set of properties that are required for our Accounts Microservice. Whenever we are trying to use @ConfigurationProperties approach, first, we need to make sure all our properties have a common prefix name.

Step – 1: Add the configuration details in accounts microservices application.yml file

accounts:  
 message: "Welcome to EazyBank Accounts Service-related local APIs"  
 contactDetails:  
 name: "Varunkumar K M"  
 email: "varunkumar@gmail.com"  
 onCallSupport:  
 91-9898767654  
 91-9897667654

So, we are going to use these properties to send to our client applications whenever they're trying to invoke one new REST API that I'm going to build. So, with the new Rest API that I'm going to build, since I'm sending this information, my client applications, they know to whom they need to contact in case of any issues like what is the developer’s name, what is the email of the developer, what is the on-call support details?

So, we can see under the same accounts prefix property. We have three different properties, message, contact details and on call support and we can see these contact details again has internally other properties like name and email and similarly on call support property has multiple values. Like we are providing two different phone numbers that someone can call in case of any issues when they're trying to invoke our accounts microservice. Now we have different set of properties with the same prefix name accounts.

If we are following the first approach, which is with the help of @Value annotation, if we want to read all these properties, we need to define all these fields along with the hardcoded property key names inside our controller class.

Similarly, if we have any other controller class where we must use the same properties, then in that controller class also we need to create multiple Java fields with the help of @Value annotation along with the hardcoded property key names. So that is going to be super inconvenience.

So, to overcome this challenge we can use the third approach. To use the third approach, first, we need to have a pojo class which represents all the properties that we have defined here. To create a pojo class.

Step – 2: Create a POJO class, for that go to the dto package that we have. Inside this DTO package, we are going to create a new DTO class, and this class name is going to be AccountsContactInfoDto So here we not going to create a Java class. Instead, we go with the Java record. Record is a new type of Java classes introduced in Java 17.

**Let me try to talk few minutes about what this record class is and why Java people introduce this.**

Sometimes we want our Pojo class or DTO classes to simply act as a data carriers, which means firstwe'll create an object of this DTO class and someone can read the data from the object of this DTO class, butthey should not be able to change.So whatever fields data that we pass during the object creation, the same values are going to be final.And anyone can read using the getter methods and there won't be any setter methods.

So, this is the most common scenario and common requirement.So instead of writing all the getter methods and creating a constructor that is going to accept thefinal Java fields, you can simply use this **record class**.

So, to this record class, we can pass all the Java fields inside these brackets. So, to define the Java fields, first I need to know what is the property name that we have defined inside the application.yml.

So, inside my application.yml we can see the very first property name is message, and it is holding a value of type String. And very similarly the second property name is contact details, but here it has a list of properties with the key and value. So inside Java the key and values can be represented with the help of HashMap or Map. And the third property we have here is on call support and this property has list of phone numbers in an alphanumeric format.

We can see **message** is a simple String**. Contact details** supports storing of map data and **on call support** is going to support to store the list of elements. By following the same data types, we need to define the fields inside our pojo class.

So here our pojo class is a record class. That's why we need to define all the fields inside this bracket itself. So, first property is of type String and please make sure the field name is same as the message because we have defined the same inside the application.yml and the second field data type is going to be map and it is going to have key and values of type String.

That's why we mentioning the String, String inside this map and the field name must be same as contact details. Since we have defined the contact details here and inside this, we have key and value which can be stored inside the map with the name, contact details. And very similarly, we need to create a list of string with the name on call support. So let me go to my record and here I'm going to mention the data type as list of String and the field name is going to be on call support.

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

Now behind the scenes, the Java is going to make these fields as final and at the same time it is going to generate a getter method and a constructor behind the scenes.

So, there won't be any setter methods, which means whenever we are using record type classes, we can only initialize the data only once and we cannot change that and whatever we have provided during the object creation, it is going to be final.

Since we also have a requirement to not change these values at runtime again and again, we can use record class here now to map all these properties to Java fields. We need to mention this prefix with the help of @Configuration properties. So, to this annotation we need to mention a parameter which is prefix. And to this prefix we need to pass a prefix which is **accounts**.

@ConfigurationProperties(prefix = "accounts")  
public record AccountsContactInfoDto(String message, Map<String, String> contactDetails, List<String> onCallSupport) {  
}

So please make sure we are mentioning the same prefix both in application.yml and inside this record class.

Step – 3: So, once we make these changes, we need to go to our spring boot main class and on top of this class we need to make sure we are mentioning an annotation which is @EnableConfigurationProperties.

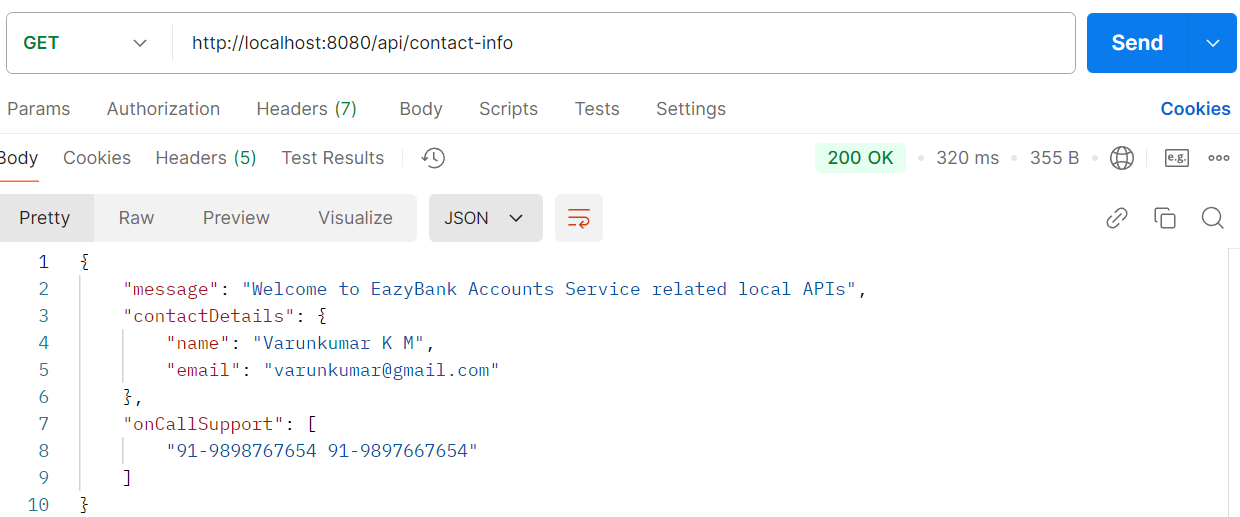
Against to this annotation, we need to invoke a parameter value and inside this value we need to pass the Pojo class details. The class name that we have here is, AccountsContactInfoDto.class So, this is the class name where we try to map all our properties from application.yml to the Pojo class.

Step – 4: Going to create a REST API in the controller class autowired the AccountsContactInfoDto record class.

@Autowired  
private AccountsContactInfoDto accountsContactInfoDto;

@Operation(  
 summary = "Get Contact Info",  
 description = "Contact Info details that can be reached out in case of any issues"  
)  
@ApiResponses({  
 @ApiResponse(  
 responseCode = "200",  
 description = "Http Status OK"  
 ),  
 @ApiResponse(  
 responseCode = "500",  
 description = "Http Status Internal Server Error",  
 content = @Content(  
 schema = @Schema(implementation = ErrorResponseDto.class)  
 )  
 )  
}  
)  
@GetMapping("/contact-info")  
public ResponseEntity<AccountsContactInfoDto> getContactInfo() {  
 return ResponseEntity  
 .*status*(HttpStatus.*OK*)  
 .body(accountsContactInfoDto);  
}

This way we can create hundreds of properties inside our application.yml and for all of them we can define Java fields inside our Dto class. Whether we are following a record class or a normal pojo class, it's up to us. But make sure the field names and the return data types are matching with the property names and the data types that we have inside the application.yml.



Since this is more matured approach, Spring team also recommend using this approach compared to @Value annotation. Off course, we need to use environment interface to read the environment variables that approach we can use.

But in real applications we will not be having multiple environment properties. There will be 2 or 3. Normal properties and configurations there can be multiple properties that we may need to create. That's why using this third approach, which is ConfigurationProperties approach, is going to make more sense.

But here I have a question. we can see as of now we have created an application.yml and we have defined all the properties. But what if I want to have different values for different environment? Maybe I want to follow these values inside my dev environment? How about inside the QA and production environment?

If my requirement is to maintain different values for different environments, then definitely all these three approaches are not going to be helpful. We need something more advanced that is supported by the spring boot framework.

**Profiles:**

If we have a requirement where we should use different property values inside different, differentenvironments?

We 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's 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**.

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

If we create three different profiles like dev, QA and prod, this will give a flexibility to have different values for our 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's try to understand more details about these **profiles** by default inside Spring Boot framework, the default profile is always active. Whatever properties and configurations we have defined inside our 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 our requirements, we can create another profiles by creating property files or yaml files by following a naming convention like we 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** --🡪 For prod profile or **application\_qa.properties** --🡪 For QA profile.

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 we can use inside our local development and the second one will be **QA profile** and the third one will be **production profile**. This way we can create any number of profiles based upon your business requirements.

**So now the very next question that we 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 spring boot. So this property is **spring.profiles.active**=based on which env like prod, QA or dev etc. and to this property key we need to pass the value of our profile.

**spring.profiles.active=prod**

Since we are 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 we can activate multiple profiles also with the help of comma separated values. By following these spring boot profiles.

We don't 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, just wanted to highlight always remember that once the application is built and packaged, it should not be modified at any cost.**

Whenever we are trying to move our application from one environment to other environment, it mightbe a feasible option inside monolithic application.But inside microservices, building our application again and again for different environments is notgoing to be a feasible option and it is going to be complex and difficult 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 we can't maintain certain sensitive credentials or properties inside our 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's try to update our accounts microservice, by creating various profiles for various environments like QA and production etc.

**Demo of the spring boot profiles inside accounts microservices**

Step – 1: So here, just like how we have application.yml file, we are going to create two more different files under the resources folder.

The very first file name that I'm going to create is application\_qa.yml. Since we want to follow the Yml configurations, we need to make sure the extension is .yml.

So, I'm trying to create this file. Similarly, I'm going to create one more file and this time the file name is going to be application\_prod.yml.

Step – 2: So now inside the application.yml file we have many properties, we have port related configurations, we have build version, we have the accounts related configurations. First, we need to identify which properties are going to change from environment to environment inside our microservice.

So if we see the server.port is going to be the same for all environments. We don't want to start our applications in different port numbers in different environments. That's why we don't need to move this property into the other profiles. We can have this as a **default value** which will always be loaded by the spring boot framework.

And very similarly, since we are using **h2 database**, it is going to be the same for all environments. We can simply ignore all this H2 database related configurations.

So, with these we have a requirement that these properties like build version and accounts related properties only will change from environment to environment inside our microservices.

So, what should we do is, we should create a similar set of properties inside our other profiles. But we need to make sure that these properties have different values in different profiles because that's our intention. Otherwise, there is no need of creating different profiles.

So, before we try to copy these properties into different profiles, let me change this build version to **3.0 inside the default profile**, which means inside my local development, the version right now we have deployed is 3.0.

Similarly, inside **QA we'll make it as 2.0** and inside **the production it will be 1.0**. This way we are mentioning different value inside, different, different profiles.

Step – 3: **application\_qa.yml**

We have 2.0 here, whereas inside the default profile it is 3.0. Very similarly, the message is also changed. We can see welcome to easy bank accounts related to QA APIs and the contact details also will be different. Whenever there is an issue inside the QA environment. My clients should reach out to the QA lead.

build:  
 version: "2.0"  
  
accounts:  
 message: "Welcome to EazyBank Accounts Service-related QA APIs"  
 contactDetails:  
 name: "Prashant - QA Engineer”  
 email: "prashant@gmail.com"  
 onCallSupport:  
 91-9911356765  
 91-9909878765

We should also tell the spring boot framework when these profiles are when this file must be activated. For the same, we need to define one more property inside this yaml file is

spring:  
 config:  
 activate:  
 on-profiles: "qa"

Step – 4: **application\_prod.yml**

Now we can try to create similar set of properties inside application\_prod.yml

spring:  
 config:  
 activate:  
 on-profiles: "prod"  
  
build:  
 version: "1.0"  
  
accounts:  
 message: "Welcome to EazyBank Accounts Service Related Prod APIs"  
 contactDetails:  
 name: "Srikanth - Product Owner"  
 email: "srikanth@gmail.com"  
 onCallSupport:  
 91-9876787656  
 91-9787654567

Step – 3: So, once we make these changes, we should be good from these two Yaml files perspective. Now as a next change, we should tell the spring boot framework that we have created two different profiles, and these are the names of the Yaml files. For the same, we need to define a property inside these application Yaml.

The property name is spring.config, so spring is already there in the top and we should make sure that this config is started inside the spring. So, we can see under spring we have created one more child element which is config. Now inside this config we need to mention import and at this import we need to mention the list of Yaml files that we have created since we are trying to mention a list of elements inside yaml.

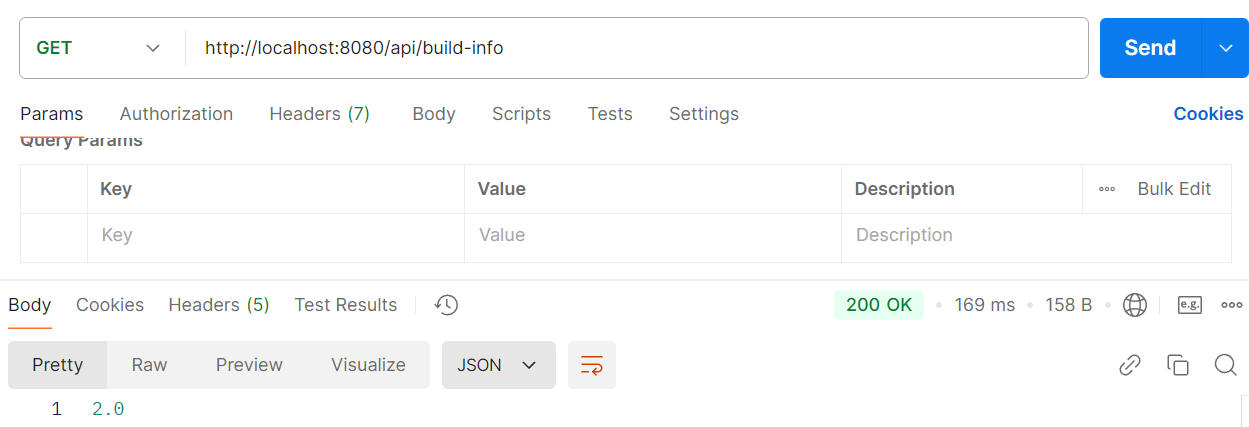
config:  
 import:  
 - "application\_qa.yml"  
 - "application\_prod.yml"

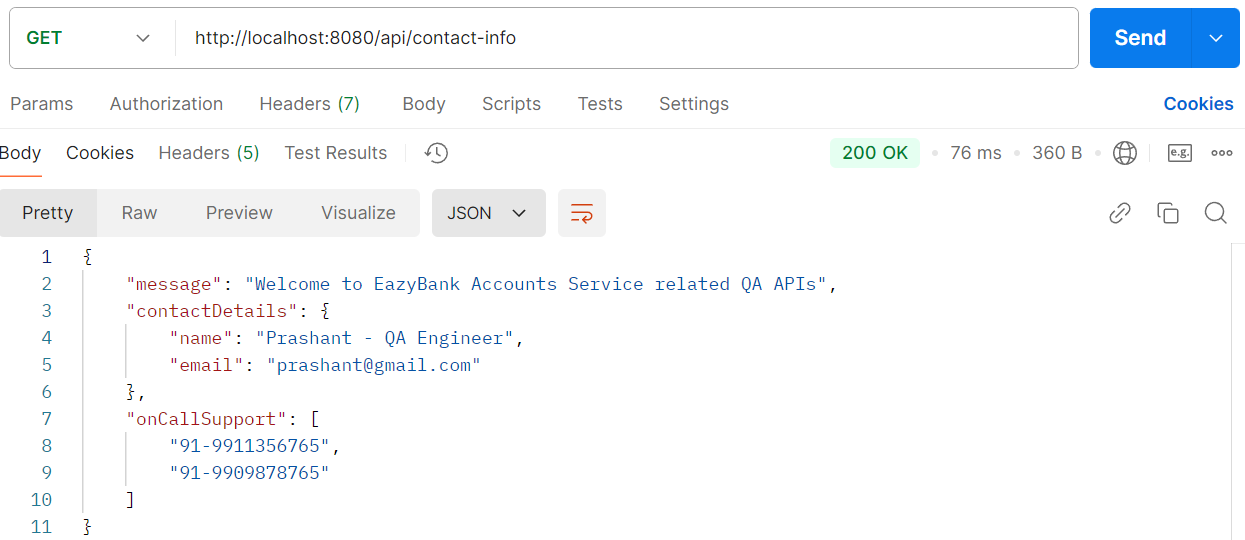
If we do not activate any profile, it defaults profile will activate.

Step – 4: I need to activate a QA or Prod environment for the same I need to go for the application.yml file for the we just under the spring elements profiles.

profiles:  
 active:  
 - "qa"

it is going to do is, it is going to override the default values present inside this Yaml file with the values defined inside the application\_qa.yml file.





Now here we may have a question. Suppose if we want to change this value to production or suppose if we want to activate a production profile inside a production environment, I need to change this value and I need to regenerate my Docker image. With that, we are not making our code base as immutable. That's why let's try to explore what are the various options that we must change the property values dynamically during the startup through an external parameter.

**Externalizing configurations using command-line, JVM & environment options**

Externalizing configurations using command-line

We know how to activate a specific profile by hardcoding the profile values inside application.yml file. With this we have a **disadvantage** every time we want to move our code from one environment to other environment and activate a profile specific to that environment. We need to make sure we are regenerating our Docker image or our web application package, which is against the **15-factor methodology** that we have discussed.

**So, to overcome this challenge, we need to identify is there any way inside Spring boot where we can activate a specific profile from an external location or through an external parameter?**

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 we 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 we use these command line arguments, it is going to have highest precedence compared to other approaches. Even if we define same property inside our application.yml or any other profile file, all those values will be overridden because whatever we 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 our microservices application. To this command only if we want to provide command line arguments, we need to follow a syntax which is by mentioning the prefix two hyphens followed by what is our 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's why trying to mention this property along with the required value. Similarly, we can pass any number of properties to this command by following the same prefix, but make sure our command line arguments are separated with a space.

**Java -jar accounts-service-0.0.1-SNAPSHOT.jar –build.version=”1.1”**

Externalizing configurations using JVM system properties

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's 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 our 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, we need to make sure we are having -d as a prefix before our property.

**Java -Dbuild.version=”1.2” -jar accounts-service-0.0.1-SNAPSHOT.jar**

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

In such scenarios the precedence is 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 we have mentioned through the command line arguments will be considered by the spring boot application.

Externalizing configurations using environment variables

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 our using Java or Spring Boot, regardless of whatever language or whatever platform we are using, these environment variables are universally supported, so even Java and Spring Boot also supports these environment variables. That's why if we have a scenario where we are not using JVM or Spring Boot or Java, but for our microservice or for our serverless application, if we 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.

Windows:  
**env:BUILD\_VERSION=”1.3”; java -jar accounts-service-0.0.1-SNAPSHOT.jar**

Linux based OS  
**BUILD\_VERSION=”1.3” java -jar accounts-service-0.0.1-SNAPSHOT.jar**

The syntax is first, we need to convert all our 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 our dots with underscore. For example, if we want to provide the property of build.version as an environment variable, then we need to make sure we are following these standard. With that, we will convert all the letters inside the build.version into capital letters post that we will replace that dot with underscore.

Let's see an example above. Inside Windows, whenever we are trying to set an environment variable through command line, first we need to execute that command which is env colon and what is your environment variable post that mention colon followed by our Java command to start our application and inside Linux based operating system or inside Mac based operating system, we can directly mention what is our environment variable. After that, please give a space. After the space we can mention our Java command.

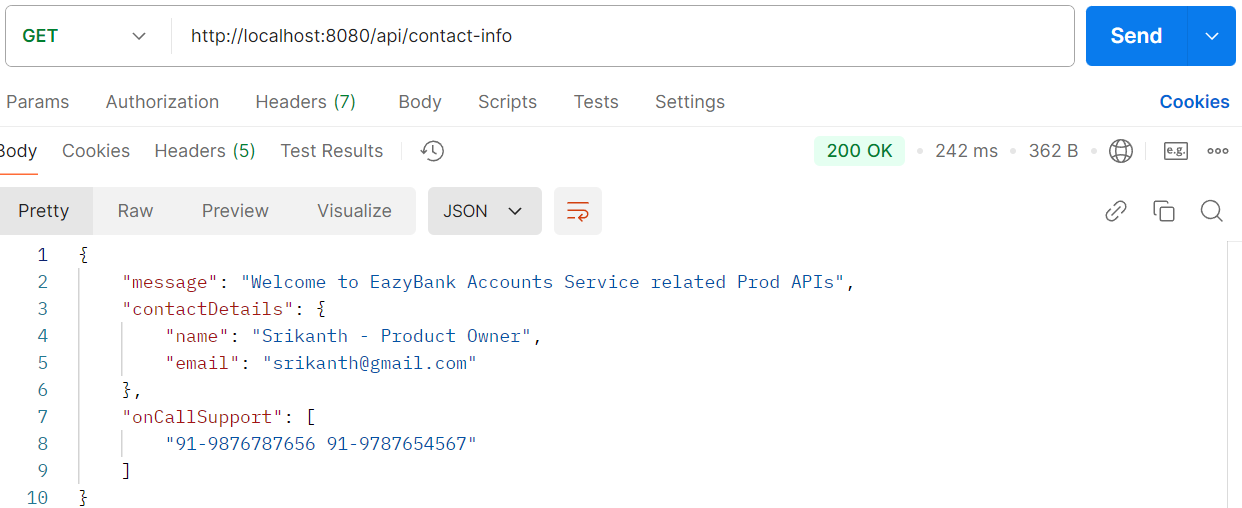
So, these are the different approaches to externalize the configurations.

**The demo of how to activate a specific profile using externalized configuration approaches**

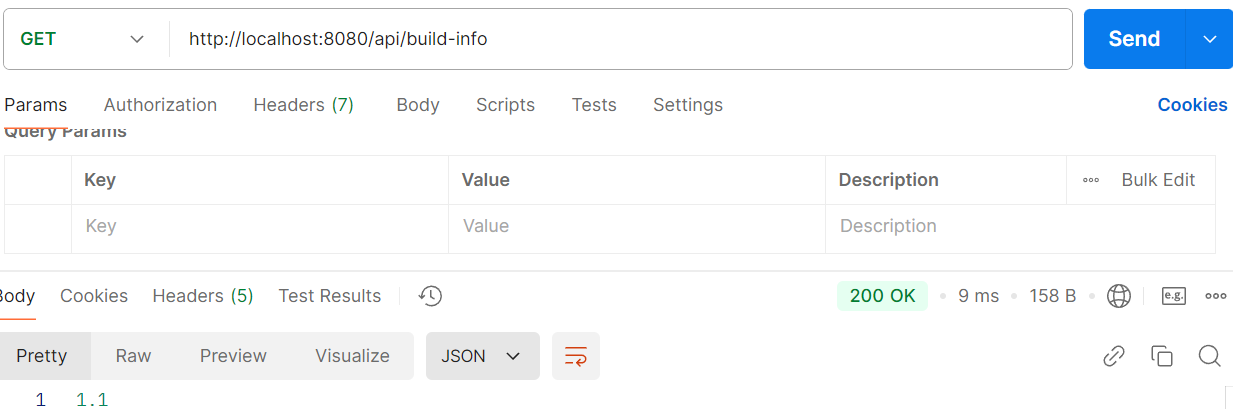
We are developers and we have the access to the IDE, let's try to utilize the same.

1. command line arguments:

Go to the accounts microservices main application -> Right click on the class -> Here we have an option Modify **Run configuration** -> There is a text field **program arguments** -> **--spring.profiles.active=prod --build.version=1.1** -> click on apply –> ok -> start the application.

This confirms our application run production environment

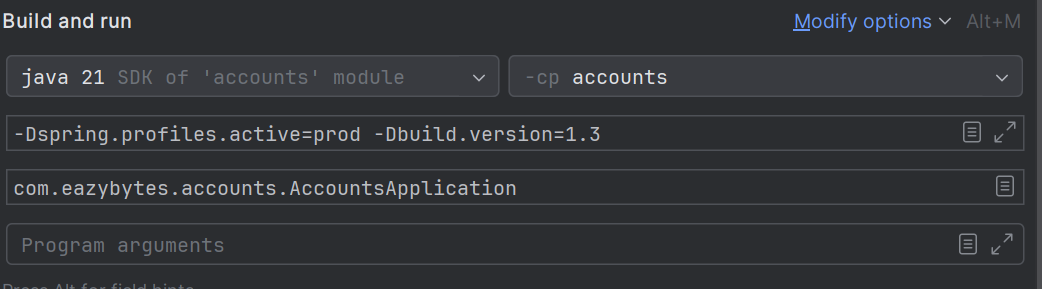
Build info will get the 1.1 version

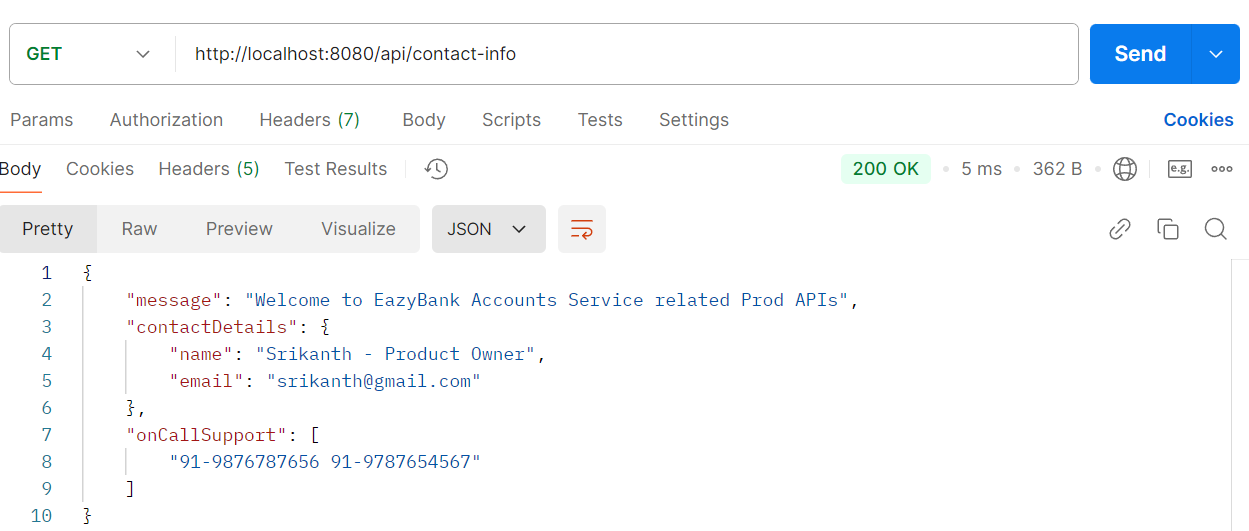


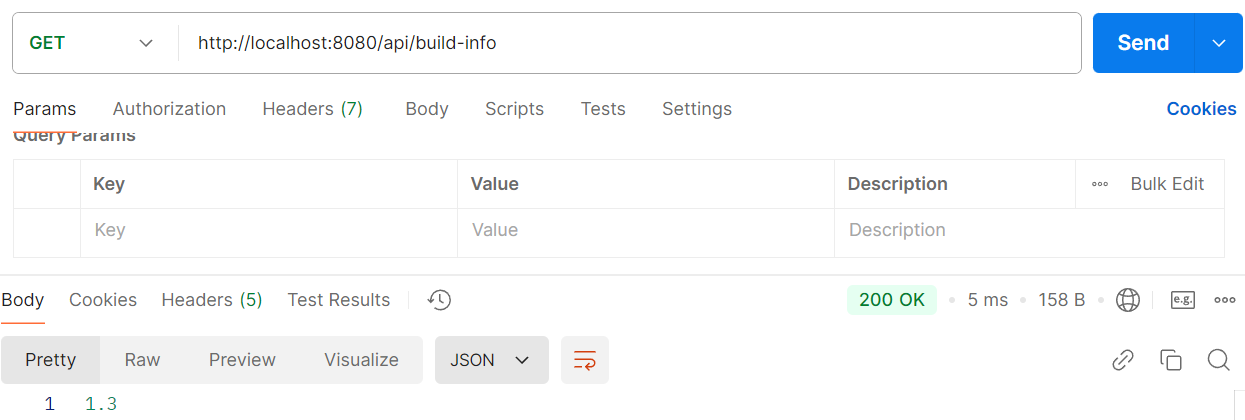
So, this is one approach on how to provide the configurations using an externalized approach.

1. JVM system variables:

Go to the accounts microservices main application -> Right click on the class -> Here we have an option Modify **Run configuration** -> There is a text field **program arguments** -> remove the command arguments -> click on the Modify options -> here there is an option Add VM options -> click on this option -> we got new text filed which is VM options -> **-Dspring.profiles.active=prod -Dbuild.version=1.3** add this -> click apply and ok.



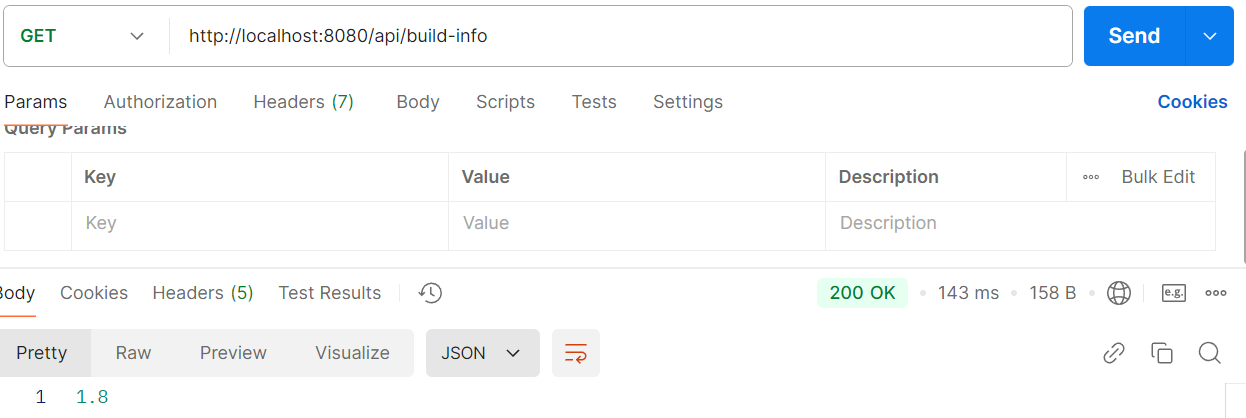


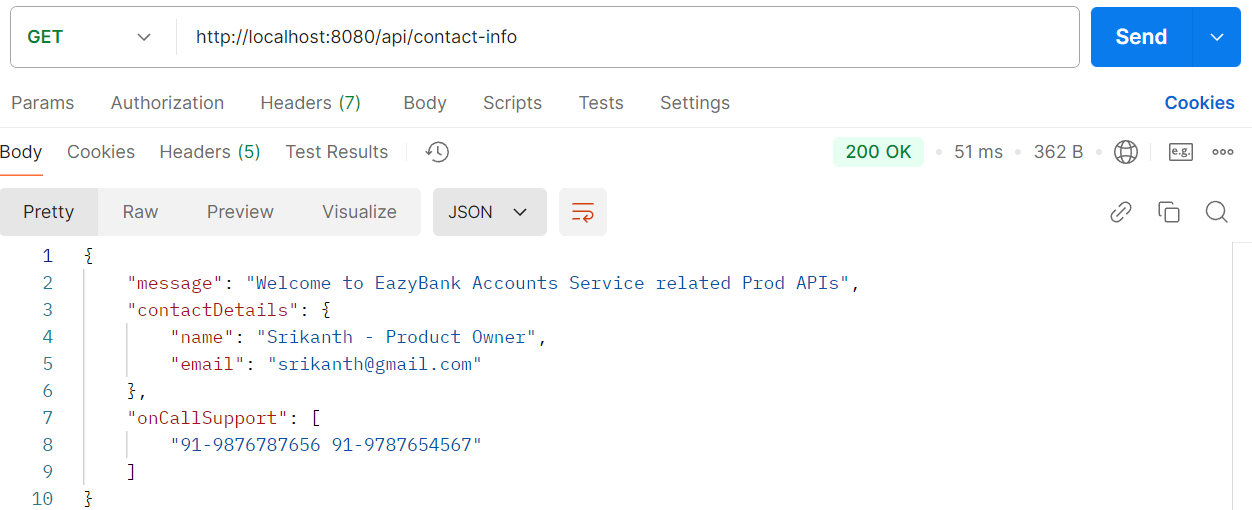


Now we can test the build info. Here we should get the output as 1.3 and this is working perfectly fine.

1. Environment variables:

Go to the accounts microservices main application -> Right click on the class -> Here we have an option Modify **Run configuration** -> delete the VM options -> we have an option Environment variables -> **SPRING\_PROFILES\_ACTIVE=prod;BUILD\_VERSION=1.8** -> Click apply and ok -> stop and restart the application.

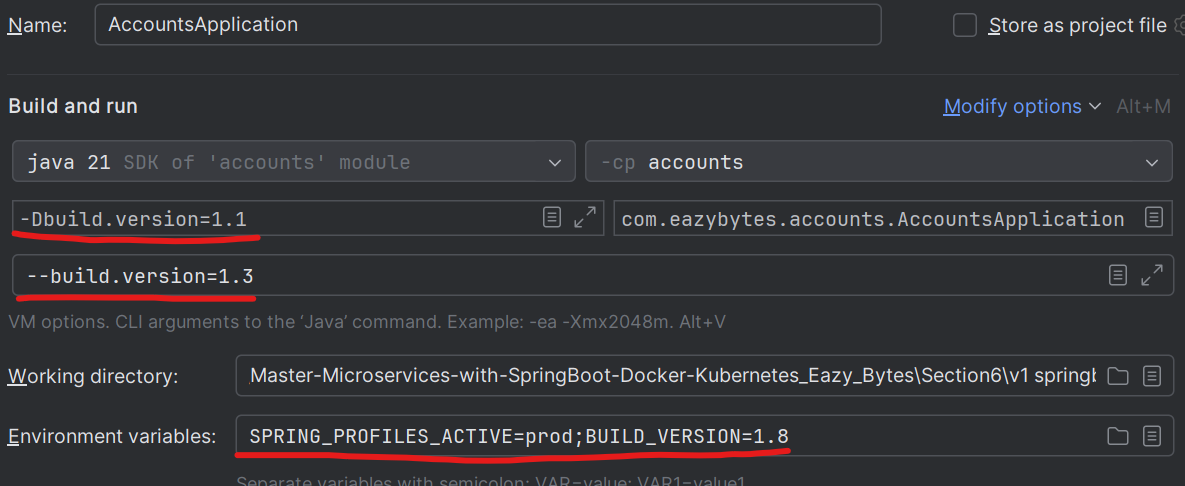


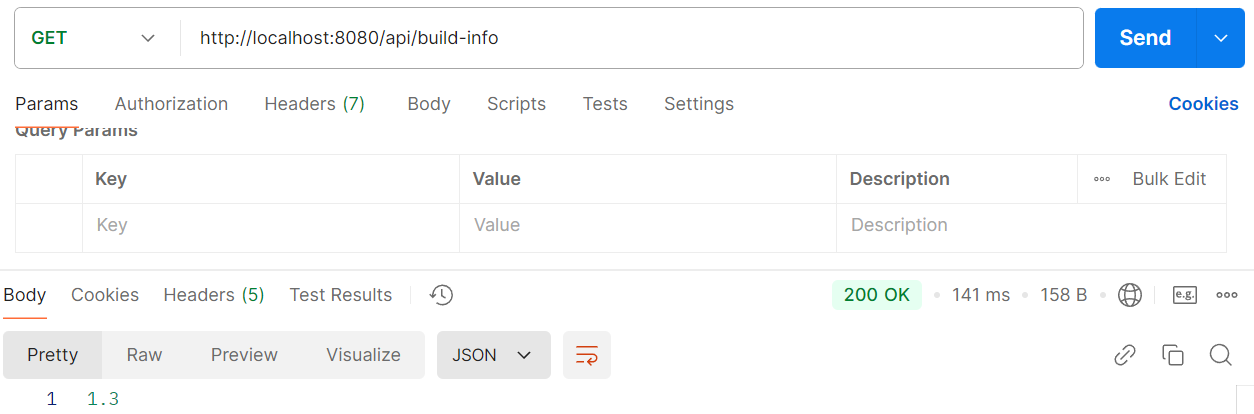


we can do is we can try to provide the build.version property in all the approaches.

We already have environment variable **SPRING\_PROFILES\_ACTIVE=prod;BUILD\_VERSION=1.8** -> add program arguments which means command line arguments **--build.version=1.3** -> and very similarly add the VM options which means JVM system variables **-Dbuild.version=1.1**

So as per our understanding, command line arguments have highest preference. So that's why now the output should be 1.3 where anyway we are activating the spring profiles active is equal to prod inside this environment variables. So let's try to focus on build.version.





We'll go to AccountsApplication again and clicking on this Modify Run Configurations. This time will remove the command line arguments. With that, now the fight will between the JVM system variables and the environment variables. And in these two the highest preference goes to the JVM system variables. That's why we will get the output as 1.1.

So, this way we can make our microservices immutable and the same Docker image we can deploy in multiple environments without the need of regenerating Docker image again and again. So, this solves the problem that we have. But do we think this is a best approach and we can follow this?

Off course this may work to some extent. We will be seeing some projects or some organizations using this Spring Boot profiles concept and externalize the configuration concept to maintain their properties inside the microservices environment. But these may work to some extent, and this is the most basic approach to maintain the configurations inside the microservices or cloud native application. This basic approach will not work. This approach has some good number of disadvantages.

**Drawbacks of externalized configurations using Spring Boot alone**

1. Like we saw whenever we are trying to use these approaches, it often involves executing separate commands with the help of Java commands or manually setting up the application, which means we need to inject these CLI arguments, JVM properties, environment variables manually at some point of time, like maybe inside the CI/CD pipelines.

So, this process again may introduce potential errors during the deployment. And again, we can see we need to dependent on someone like there should be some human setting up all these externalized configurations during the startup of the application. And doing that for all instances for all the microservices is going to be super challenging.

Though we can automate most of these tasks with the help of CI/CD pipelines like GitHub actions or Jenkins. But still there is a chance that someone can mess up this process.

1. The next limitation is if we have hundreds of microservice, then we will also have thousands of configuration properties that will evolve and changes on day-to-day basis. So just like how our application code, we should also follow the strategies to store and maintain the versioning of our configurations and properties based upon our release.

And at the same time, we should also support the auditing functionality like who access to our configuration, which clients access to our configuration data. With the spring boot profiles alone, we are putting all our configurations inside the source code itself. So, anyone who has access to my source code or Docker image, they can easily understand all my configurations and it is not a wise decision to expose all the configurations to anyone.

That's why it is always recommended to store our configuration separately inside a centralized repository where it supports all kind of versioning, tracking revision and auditing the configurations.

1. And whenever we are trying to use environment variables, they lack granular access, like whoever has access to our production server, like all the server admins, they can see our environment variables. Maybe we are trying to configure database credentials as some sensitive properties with the help of environment variables because we don't want to mention them inside the application.yml file or inside the profile files directly. Since that is going to expose our sensitive information to everyone.

But even if we try to follow the environment variables with this approach, also our server admins will be seeing our environment variables. That's why this is considered as a serious security limitation.

1. Inside the microservices, we will deploy multiple instances of a single microservice like we have three microservices like accounts, loans and cards. But if we decided to deploy three instances of each microservice, then there will be total nine Docker containers or total nine instances will be running inside our production in such scenarios, providing all these externalized configurations with the CLI arguments, JVM properties or any other approach is going to be super, super challenging.

And here we just talking about three microservices where each one has three instances. Think like 100 microservice, where each one has at least five instances, then the number of instances is going to run inside, our production is going to be 500 instances. We can imagine, even if we have a single manual task involved for a microservice, the same must be repeated in all the microservice.

That's why this approach is not going to work. When we are trying to deploy multiple instances of our microservices.

1. The next limitation is around the security. Think like we want to provide the database password to our spring boot container or our spring boot microservice, regardless of whatever approach we use like CLI, JVM or environment variables or spring boot profiles, we need to mention the plain text password which can expose our password to everyone.

And these approaches, they are not going to support any encryption or decryption and storing the secrets securely inside our application. So that's why this is also going to be a security limitation.

1. And the last important limitation is, with all these approaches that we have discussed as of now, whenever we want to change the property value at runtime, we need to restart our application.

Suppose if we want to change a property value, once the microservices instances are started, then definitely we need to change this property and restart all my containers or microservices, which is not again a good way to handle this. The microservices should be able to automatically read the latest property value without the restart. With spring boot alone, this is not going to happen. Every time we want to change something, we need to restart the application or microservice. So, with all these limitations, we cannot use spring boot alone to manage the configurations inside the microservices applications.

If we are seeing someone using spring boot alone to handle the configurations inside the microservice, then there are two chances. Either the project is very small in nature where they have, with a very limited number of properties with very low severity applications, or they may not be knowing the advanced options that we have to handle these kinds of scenarios.