**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 boot 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.