**Summary – Spring configuration**

Levels of configurations:

1. Configuration inside a property file (like ‘my.greeting’ in application.properties file). The downside of this is that every time I want to configure something inside, I need to unzip the application jar, add the changes to the property file, zip to a jar and deliver the jar - next points avoids that.
2. Configuration outside the jar.
   1. I can go to C:\Users\baraky\git\MicroServices\spring-boot-config\target (where the jar is located)
   2. Create another application.properties file in the same location
   3. Add the same ‘my.greeting’ property
   4. Give it another value
   5. Run the jar again. The new property file outside the jar will override the changes made in the property file inside the jar
3. Configuration using command line
   1. I can go to C:\Users\baraky\git\MicroServices\spring-boot-config\target (where the jar is located)
   2. Write the following command: java - jar spring-boot-config-0.0.1-SNAPSHOT.jar --my.greeting="Hello from command line". This will override even the second configuration

From properties file to yml file

We can also change the properties file to a yml file. This gives the opportunity to remove duplicate prefix of properties and show it in indentation. The key principles in using ymls are:

* All the ‘=’ turns to ‘:’
* String property don't need "
* If we have special chars in the property (like: {, }, /, ', \*), we should put " at the beginning and in the end of the String
* You must indent the properties in order to tell yml that it's the same group of properties
* When indenting, don't use tabs only spaces

Env specific configuration

In order to make configuration more env specific we can use Spring profiles. Spring profiles allows us to stuck configuration one on top of the other in a way that all the common configuration can be in the bottom layer and other layers can be env specific. Few principles regarding this:

1. Spring has a default profile that's the applucation.properties file
2. Spring profiles can work with properties files and with yml files
3. We have several ways to define what is the active profile that we want:
   1. Create a properties/ yml file in the same location as the existing one
   2. Name the file application-<chosen name>.properties/ yml
   3. Go to application.properties/ yml file and add the following property: spring.profiles.active: <chosen name>
   4. When running the jar in command line we can add an argument that overrides the above property and make in more env specific
4. We can have different beans instantiated by different configuration in order to trigger different business logic. This is done by annotating the bean with @Profile("<chosen name>"). by default, all the other beans are internally annotated with the @Profile("default") annotation

Externalize configuration

In order to externalize configuration, we can use configuration MS (spring-cloud-config-server). This MS will hold the configuration in git repository files and will fetch them each and every time it is used. Meaning, we don't have to boot the configuration server for the changes to take place. it does that in several ways:

1. First, we must define the MS as configuration client by adding a dependency of spring-cloud-starter-config and dependencyManagement in pom.xml of the MS.
2. We must connect the MS to the configuration MS by adding spring.cloud.config.uri property to the application.yml file
3. We will create an application.yml in the git repository that will be common to all the MS who uses this configuration MS
4. We will create a <specific MS name>.yml in the git repository that will be specific to the MS that's using the configuration MS, we must add property spring.application.name to application.yml file that will specify the name of the client configuration MS

Manage configuration

In order to manage the configuration in real time (without booting the client configuration server each time there's a change in configuration), we must do it in several steps:

1. Add the dependency spring-boot-starter-actuator (if it's not already added) and add the property management.endpoints.web.exposure.include to the application.yml file.
2. Add @RefreshScope to the file that we want to refresh each time there's a change in configuration.
3. We must run <host>:<specific MS port>/actuator/refresh POST method API

Special annotations

* @ConfigurationProperties(<section in yaml file>) (annotates class) – defines a class that its members will be mapped automatically to the properties under the section in the yaml file.
* @RefreshScope (annotates class) - this tells Spring that whenever there's a change in the configuration MS, we will pick the refreshed values for this file only
* @EnableConfigServer (annotates a class) – defines the MS as a configuration server.