Class 7 – Spring Boot Microservice Enterprise Applications with Java

Contents

Review

- Class6 Homework and Assignment
- Q&A

Break

Microservices Infrastructure components

Discovery

In the cloud, services are often ephemeral and it's important to be able to talk to these services abstractly, without worrying about the host and ports for these services. This could be DNS based or based off a service like the Netflix Eureka project[1]. Eureka also allows more sophisticated load-balancing than DNS + a typical loadbalancer can handle (e.g.: round-robin). We'll use a service registry and Spring Cloud's `DiscoveryClient` abstraction.

Code: eureka-service in the source zip file

In **eureka-service** folder

- Start Spring Boot app
- Dashboard at http://localhost:9999/

[1] http://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html

Configuration

We want externalize all configuration and Spring Boot and Spring Cloud provides a simple solution that integrates with the existing Spring configuration property placeholders. (12-factor[2], Spring[3])

Externalized configuration is a key tenet as properties may change between environments eg: the database host name is different in different environments. There are two levels of this:

1. Properties within an individual Microserivce and for Spring Boot the order of property loading is strictly defined: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html
2. Properties loaded from a server and for this we will use the Spring Cloud Config Server

In both cases Spring also provides "profiles" that can be used to load environment specific properties and it is all based on the Spring Property Placeholder Configuration.

Code: In **config-service** folder

You also need the Git repo with sample properties – this MUST be a git repo. Clone from:

https://github.com/rahulaga/tbtf-sample-props

- Start Spring Boot app for config-service
- Fetch default: http://localhost:8888/application/default (see Postman collection)
- Fetch dev profile: http://localhost:8888/checking-account-service/dev

The HTTP service has resources in the form:

- **/{application}/{profile}[/{label}]**
- /{application}-{profile}.yml
- /{label}/{application}-{profile}.yml
- /{application}-{profile}.properties
- /{label}/{application}-{profile}.properties

Properties follow an hierarchy of inheritance.

- Default: application.properties
- Override with {applicaton-name}.properties
- Override with {application-name}-{profile}.properties

Note: Spring uses relaxed bindings so these are all the same properties: foo.barBaz, foo.bar-baz, FOO_BAR_BAZ

To provide a profile on Spring Boot startup provide -Dspring.profiles.active=foo

[2] https://12factor.net/config
[3] http://cloud.spring.io/spring-cloud-static/Camden.SR5/#_spring_cloud_config

Testing REST services – Automation

Test automation for REST services is easy since your test harness is basically calling your REST APIs which are nicely documented! This serves as great black-box testing. These automated tests are usually the "e2e" or "integration" tests and make REST calls to a server. Test method:

1. Start a HTTP server
2. Execute tests
3. Shutdown HTTP server

We will use Rest Assured[4].

Example code – take a look at the code under tbtf-bank-test.

[4] http://rest-assured.io/