Spring Cloud Config Server and Client Tutorials

<https://dzone.com/articles/how-to-setup-the-spring-cloud-configuration-server-with-git>

<https://dzone.com/articles/setting-up-spring-cloud-config-client>

<https://github.com/mag1309/microservices/tree/master>

Companies are slowly adopting microservice over monolithic architecture to scale applications. Compared to monolithic, microservice architecture breaks applications into small, manageable services, i.e., microservices. That means one monolithic application can be converted to many microservices, and these services are developed and deployed independently. These services collaborate with each other to fulfill business objectives. Each microservice manages their own application-specific properties file, i.e., application.properties.

In most situations, multiple microservices and their multiple instances run together to fulfill the business needs.  Any update in the properties file may require the redeployment and restart of services. Consider a situation where property updates need to be done on hundreds of services and their instance. This may require a considerable amount of downtime.

To solve this problem, Spring Boot has provided a Spring configuration server. This will manage all service's properties files from a single point at runtime.

**Advantage of Spring Cloud Config Server**

A centralized application that manages all the application related configuration properties and their versions.

1. If there is any property change, only the centralized repository will be updated, and all the related services will receive the updated property changes without the microservice application redeploy or restart, i.e., application properties can be updated at runtime without redeploying or restarting microservices.
2. Client-Server communication architecture, i.e., Spring Boot provides both client and server-specific components. Server manages all properties from the repository, while microservices act as a client to the server to consume properties.
3. Secure access to the repository with encryption of the sensitive data.
4. Support to properties associated with multiple environments like development, staging, UAT, and production.

This tutorial is helpful to develop a complete working example for a good understanding of concepts like:

1. Repository (Store configuration).

2. Cloud config server (Read configuration from repository).

3. Cloud config client (Read configuration via Cloud Config Server).

4. Update configuration and trigger refresh event.

This tutorial will cover points 1 and 2. Another tutorial will follow to cover points 3 and 4. Let’s follow the tutorials with the basic assumption that Java8, Maven, and Eclipse are installed.

## ****Prerequisites****

* [Java 1.8+](http://www.oracle.com/)
* [Maven](https://maven.apache.org/)
* Eclipse with Spring Tools Suite (STS)
* Git

## Configure Repository

1. Install Git from <https://git-scm.com/downloads> if not available on your machine. Git can be easily checked with the following command.Git checking command

2. Create a folder, e.g., config-properties, where configuration files are to be kept.  Add a property file, e.g., **hello-service.properties** with the following property:

message=Hello world - this message is from config server

3. Instead of application.properties, name of the file kept as the client service name. Here the name of the client service name is **hello-service**. So file name is taken as **hello-service.properties.**

spring.application.name=hello-config-server

server.port=8888

**spring.cloud.config.server.git.uri=file:///d:/config-properties**

4. Now, initialize Git in the configuration folder with the help of the following commands. These commands will help to commit the properties files in the Git repository:

1

$> cd config-properties

2

$> git init

3

$> git add .

4

$> git commit -m 'Initial Release'

$> git init

5

$> git add .

6

$> git commit -m 'Initial Release'

5. Let's add a few more files considering the various other environments like development and UAT. Add and commit these files also to the repository (Refer to above step).

message=Hello world - this message is from config server

**hello-service-development.properties**

1

message=Hello world - this message is from development

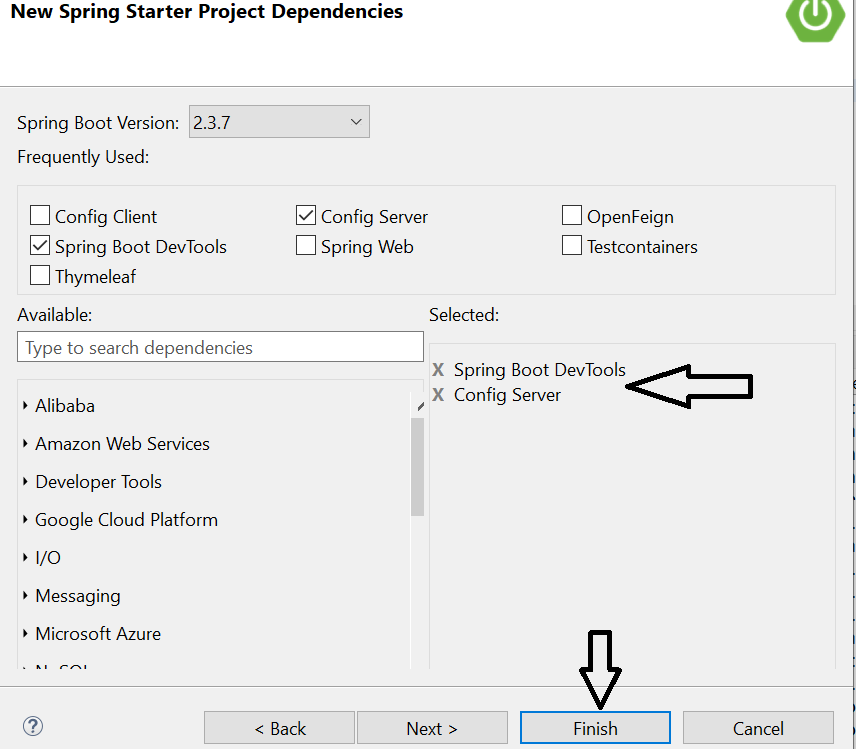
**hello-service-uat.properties**

1

message=Hello world - this message is from UAT

6. The local Git repository is created and used for development/testing purposes. For production deployment, the Repository can be more secure and kept at a remote location on Github, Bitbucket, Gitlab, AWS Code commit, etc.

## Cloud Config Server Setup

1. Create a Spring Starter Project with a name likeHelloConfigServer. Select spring version 2.3.7 (Do not select **2.4.1** as it's having an issue in showing properties files) with dependencies like Config Server and DevTools (used for easy redeployment).

2. Once the project is created, go to the application.properties file available inside the resource folder and add the following in the file:

1

spring.application.name=hello-config-server

2

server.port=8888

3

spring.cloud.config.server.git.uri=file:///d:/config-properties

3. It's standard practice to run a configuration server at port 8888. For a Windows environment, kindly keep an additional "/".  For other environments, "//" will work. spring.cloud.config.server.git.uri=file://d:/config-properties

4. For remote repository, sample GitHub configuration will look like as below:

1

spring.cloud.config.server.git.uri=https://github.com/abcd/springboot

2

spring.cloud.config.server.git.uri.username=test

3

spring.cloud.config.server.git.uri.password=test

5. Now open the main application file, i.e., HelloConfigServerApplication.java, and add **@EnableConfigServer** annotation to make it Config Server. Final java file will look as below

1

package com.hello.example;

import org.springframework.boot.SpringApplication;

4

import org.springframework.boot.autoconfigure.SpringBootApplication;

5

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

6

7

**@EnableConfigServer**

8

@SpringBootApplication

9

public class HelloConfigServerApplication {

10

11

public static void main(String[] args) {

12

SpringApplication.run(HelloConfigServerApplication.class, args);

13

}

14

}

6. All basic configurations are done now. Let's run the application as SpringBootApp. The Tomcat server will start at 8888 as configured. Check the following URL (This will pick the default configuration file from the repository).

* <http://localhost:8888/hello-service/default>

7. If the following response is received, then all configurations are OK, i.e., Config Server is fetching default property from Git repository.

1

{

2

 "name": "hello-service",

3

 "profiles": [

4

   "default"

5

],

6

 "label": null,

7

 "version": "3da88aa3c6103cc7f2b383a3ee5d5dad5e6e870c",

8

 "state": null,

9

 "propertySources": [

10

  {

11

     "name": "file:///d:/config-properties/hello-service.properties",

12

     "source": {

13

       "message": "Hello world - this message is from config server"

14

    }

15

  }

16

]

17

}

# Setting Up a Spring Cloud Config Client

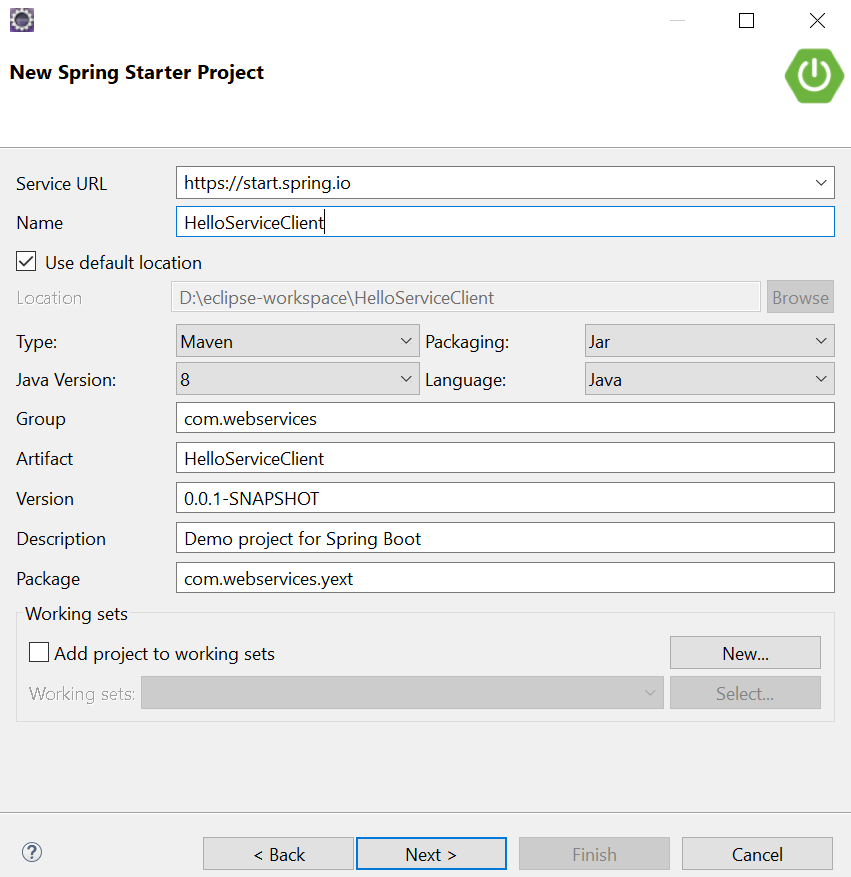
### Learn how to create a configuration client which can communicate with a configuration server to access a git repository and fetch configuration files.

This post is a continuation of another post of mine, ["How to Setup the Spring Cloud Config Server With Git,"](https://dzone.com/articles/how-to-setup-the-spring-cloud-configuration-server-with-git) i.e., setting up of Spring cloud config server with a git repository. In my earlier post, successful communication was established between a cloud config server and a git repository. In this post, the focus will be on the development of a cloud config client and its interaction with a cloud config server.

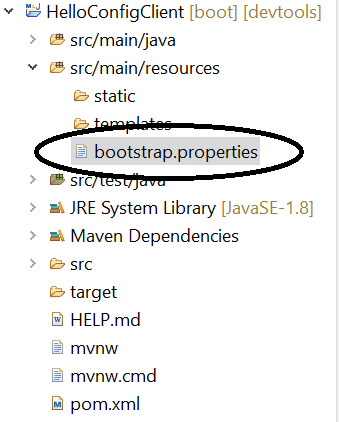
## ****Prerequisites****

* [Java 1.8+](http://www.oracle.com/)
* [Maven](https://maven.apache.org/)
* Eclipse with Spring Tools Suite (STS)
* Git
* Spring Cloud Config Server

**Cloud Config Client Setup**

1. Create a new Spring Starter Project with the name "HelloServiceClient". 

## On the "Next" screen, select Spring version 2.3.7 with dependencies like Spring Web (for REST Controller development), Config Client (to make it a cloud config client), Actuator (for easy monitoring), and DevTools (for easy redeployment). Click "Next" to create a project. If you choose to work in Spring v.2.4+, then "Cloud Bootstrap" will be added as an additional dependency. https://dz2cdn1.dzone.com/storage/temp/14250826-1608547837940.png

1. Once the project is created, go to the resource folder and delete the application.propertiesfile. The application specific properties will be read from the git repository, so this file is no longer required. You will to add a new file, bootstrap.properties, in place of application.properties inside the resource folder. This file will be loaded during the service start process and will be used to access the cloud config server. 
2. Thebootstrap.propertiesfile will have following properties:

spring.application.name=hello-service

server.port=8080

spring.profiles.active=development

**spring.cloud.config.uri**=http://localhost:8888

* + **spring.application.name -** This is the identifier for the microservice application and there will be a property file with the same name in the repository. This will identify which property file to fetch from the repository for a particular microservice. For example, for hello-service.properties, we would use hello-service-development.properties (the -development suffix tells the system to fetch code for development profile).
  + **spring.profile.active**- Identifier used to for environment, i.e. development/production, etc.
  + **spring.cloud.config.uri -**Specifies the URL for the Spring cloud config server.

1. Now create a REST Controller to access the attribute available in the git property file.

1

package com.hello.example.controller;

2

3

import org.springframework.beans.factory.annotation.Autowired;

4

**import org.springframework.cloud.context.config.annotation.RefreshScope;**

5

import org.springframework.core.env.Environment;

6

import org.springframework.http.HttpStatus;

7

import org.springframework.http.ResponseEntity;

8

import org.springframework.ui.Model;

9

import org.springframework.web.bind.annotation.GetMapping;

10

import org.springframework.web.bind.annotation.RestController;

11

12

**@RefreshScope**

13

@RestController

14

public class HelloController {

15

16

**@Autowired**

**private Environment env;**

@GetMapping("/hello")

20

public ResponseEntity<String> getHello(Model model) {

21

return new ResponseEntity<String>( env.getProperty("message"), HttpStatus.OK);

22

}

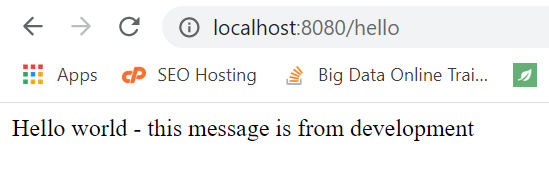
23

}

Here, the "message" attribute is available inside the hello-service-development.properties file which is available in the git repository.

1

message=Hello world - this message is from development

1. The basic setup is now complete. Let's run the application as "Spring Boot App." The application will run at default port, 8080 (make sure, the **HelloConfigServer**application is running at port 8888. If it's not running, then first start the **HelloConfigServer**application. Later. start the **HelloConfigClient** application, as the client application load properties during the start up process).
2. To test the config client, go to the URL *http://localhost:8080/hello*. The config client will look for the configuration file (hello-service-development.properties) in our git repo – the expected response is below:
3. Congratulations on setting up the cloud config client successfully and creating communication with the config server!

## ****Configuration Update****

Now that communication is set up between the cloud config client, the cloud config server, and the git repository, w will happen if any property file is updated after the application's deployment? Can the config client fetch the latest configuration? No, Service/Application needs to be restarted to get the latest update. This is a problem.

To solve this problem, Spring Boot has provided an approach to fetch the latest configuration information from repository at runtime with the help of the @RefreshScope annotation and and the Actuator dependency.

@**RefreshScope**- By default, the configuration values are read at the client’s server startup. This annotation forces the bean to refresh its configuration, i.e. to pull updated configurations from the repository via a config server without the application restarting in real-time.

**Actuator** - Once the /actuator/refresh event for the application is triggered, all the beans that are annotated with @RefreshScope get refreshed. This means updating values retrieved from the config server and updating the client's microservice bean.

We can trigger the refresh event with the help of the URL http://localhost:8080/actuator/refresh. This will result in refreshing all the beans annotated with @RefreshScope. Since the HelloController is annotated with @RefreshScope, its properties will get refreshed.