Spring Cloud Config

Table of Contents

Estimated Time: 60 minutes

Requirements

Provided by e-mail last week

What You Will Learn

- How to set up a git repository to hold configuration data
- How to set up a config server (`config-server`) with a git backend
- How to set up a client (`greeting-config`) to pull configuration from the `config-server`
- How to change log levels for a running application (`greeting-config`)
- How to use `@ConfigurationProperties` to capture configuration changes (`greeting-config`)

- How to use `@RefreshScope` to capture configuration changes (`greeting-config`)
- How to override configuration values by profile (`greeting-config`)
- How to use Cloud Bus to notify applications (`greeting-config`) to refresh configuration at scale

---

Exercises

Set up the `app-config` Repo

To start, review your BitBucket repository, which will hold your configuration.

1) Log into http://bitbucket.app.syfbank.com:7990/. Using your SSO ID and password "password". If you don't have one by default, create a repo called app-config.

2) Copy the HTTPS clone URL from your repo. It should be something like:

http://bitbucket.app.syfbank.com:7990/<your-SSO-Id>/app-config.git

3) Open a new terminal window and clone the repo you just created (you may want to create a common location for your BitBucket repos, such as ~/repos):

```
$ cd [location of your bitbucket repos, e.g. ~/repos]
$ git clone <Your app-config repo - HTTPS clone URL>
$ cd app-config
```

Notice that this repository is basically empty. This repository will be the source of configuration data.

Create a `config-server`

1) Create a Config Server Service Instance

Using the cf cli, do the following (for help review the docs):

Create a config server:

```
$ cf create-service p-config-server standard config-server -c ‘{ "git": { "uri
": "<your bitbucket url>" } }’
```

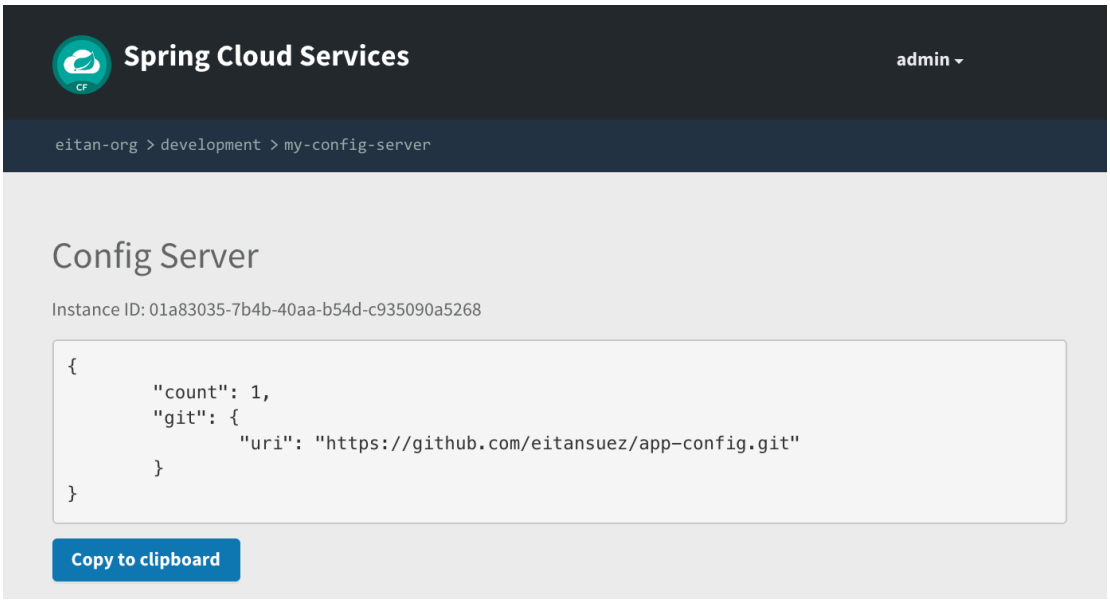
Make sure to substitute your `app-config` repository. Do not use the literal above.

Feel free to name your service anything you like, it doesn’t have to be named `config-server`. The Config Server instance will take a few moments to initialize and then be ready for use.

You can invoke either the `cf services` or `cf service` commands to view the status of the service you just created.

In addition, you can visit your Config Server’s service dashboard in the Apps Manager to view its configuration and status:

In a browser, navigate to the apps manager, and to your space. You should see your config server service displayed there (it may be in a separate tab named `services`). Click on the service, and in the subsequent view, select the `Manage` link.



Set up `greeting-config`

1) Review the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/pom.xml` By adding `spring-cloud-services-starter-config-client` to the classpath, this application will consume configuration from the `config-server`. `greeting-config` is a config client.

```
<dependency>
        <groupId>io.pivotal.spring.cloud</groupId>
        <artifactId>spring-cloud-services-starter-config-client</artifactId>
</dependency>
```

2) Review the `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/resources/bootstrap.yml`

```
spring:
  application:
    name: greeting-config
```

`spring.application.name` defines the name of the application. This value is used in several places within Spring Cloud: locating configuration files by name, service discovery/registration by name, etc. In this lab, it will be used to locate config files for the `greeting-config` application.

3) Package the `greeting-config` application. Execute the following from the `greeting-config` directory:

```
$ mvn clean package
```

4) Deploy the `greeting-config` application to PCF, without starting the application:

```
$ cf push greeting-config -p target/greeting-config-0.0.1-SNAPSHOT.jar -m 512M
 --random-route --no-start
```

5) Bind the `config-server` service to the `greeting-config` app. This will enable the `greeting-config` app to read configuration values from the `config-server`.

```
$ cf bind-service greeting-config config-server
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. Our app doesn't need to be restaged at this time because it isn't currently running.

6) Our PCF instance is using self-signed SSL certificates. Set the `TRUST_CERTS` environment variable to the API endpoint of your Elastic Runtime instance. You can quickly retrieve the API endpoint by running the command `cf api`.

```
cf set-env greeting-config TRUST_CERTS <your api endpoint>
```

Be sure to specify the api endpoint as a hostname and not a url, i.e. without the leading `https://` scheme. You can safely ignore the *TIP: Use 'cf restage' to*

ensure your env variable changes take effect message from the CLI. Our app doesn't need to be restaged at this time.

NOTE:

All communication between Spring Cloud Services components are made through HTTPS. If you are on an environment that uses self-signed certs, the Java SSL trust store will not have those certificates. By adding the `TRUST_CERTS` environment variable a trusted domain is added to the Java trust store. For more information see the this portion of the SCS documentation.

7) Start the `greeting-config` app.

```
$ cf start greeting-config
```

8) Confirm the `greeting-config` app is up. Browse to your `greeting-config` application. You should be prompted to authenticate. Why? `spring-cloud-services-starter-config-client` has a dependency on Spring Security. Unless the given application has other security configuration, this will cause all application and actuator endpoints to be protected by HTTP Basic authentication.

9) If no explicit username or password has been set then Spring Security will generate one for you. This applies for the `greeting-config` application. Use the following to login:

username: `user`

password: You can find this in the log output in Apps Manager or at the command line:

```
$ cf logs greeting-config –recent
```

Look for a log message similar to the following: `Using default security password: 90a3ef2a-4e98-4491-a528-a47a7162dd2a`. Use this password to login.

Note: Username and password can be explicitly set through the `security.user.name` and `security.user.password` configuration parameters.

6) After logging in you should see the message “Greetings!!!”.

What Just Happened?

At this point, you connected the `greeting-config` application with the `config-server`. This can be confirmed by reviewing the logs of the `greeting-config` application.

`greeting-config` log output:

```
2015-09-18 13:48:50.147  INFO 15706 --- [lication.main()] b.c.PropertySourceBo
otstrapConfiguration : Located property source: CompositePropertySource [name=
'configService', propertySources=[]]
```

There is still no configuration in the git repo for the `greeting-config` application, but at this point we have everything wired (`greeting-config` → `config-server` → `app-config` repo) so we can add configuration parameters/values and see the effects in out client application `greeting-config`.

Configuration parameters/values will be added as we move through the lab.

Unsecure the Endpoints

For these labs we don’t need Spring Security’s default behavior of securing every endpoint. This will be our first example of using the `config-server` to provide configuration for the `greeting-config` application.

1) Create a file called `greeting-config.yml` in your local `app-config` repo directory. Note that the file name matches the name of the greeting-config application as declared in its bootstrap.yml file. Add the content below to the file.

```
security:
   basic:
```

```
    enabled: false # turn off securing our application endpoints
management:
  security:
    enabled: false # turn off securing the actuator endpoints
```

2) Push your changes back to BitBucket

```
$ git add greeting-config.yml
$ git commit -m “New file”
$ git push -u origin master
$ git status  → To check the status of your files
```

3) Restart the `greeting-config` application:

```
$ cf restart greeting-config
```

4) Review the logs for the `greeting-config` application. You can see that configuration is being sourced from the `greeting-config.yml` file.

```
2015-11-02 08:57:32.962  INFO 58597 --- [lication.main()] b.c.PropertySourceBo
otstrapConfiguration : Located property source: CompositePropertySource [name=
'configService', propertySources=[MapPropertySource [name='https://bitbucket..
./app-config.git/greeting-config.yml']]]
```

5) Browse to your `greeting-config` application. You should no longer be prompted to authenticate.

Changing Logging Levels

Next you will change the logging level of the `greeting-config` application.

1) View the `getGreeting()` method of the `GreetingController` class (`$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/greeting/GreetingController.java`).

```
@RequestMapping("/")
String getGreeting(Model model){
    logger.debug("Adding greeting");
    model.addAttribute("msg", "Greetings!!!");
    if(greetingProperties.isDisplayFortune()) {
      logger.debug("Adding fortune");
      model.addAttribute("fortune", fortuneService.getFortune());
    }    //resolves to the greeting.vm velocity template
    return "greeting";
 }
```

We want to see these debug messages. By default only log levels of `ERROR`, `WARN` and `INFO` will be logged. You will change the log level to `DEBUG` using configuration. All log output will be directed to `System.out` & `System.error` by default, so logs will be output to the terminal window(s).

2) In your `app-config` repo. Add the content below to the `greeting-config.yml` file and push the changes back to BitBucket.

```
security:
   basic:
     enabled: false
management:
   security:
     enabled: false
logging: # <----New sections below
   level:
     io:
```

```
      pivotal: DEBUG
```

Note that we have set the log level for classes in the `io.pivotal` package to `DEBUG`.

3) While watching the `greeting-config` terminal, refresh your `greeting-config` application url. Notice there are no `DEBUG` logs yet.

4) For the `greeting-config` application to pick up the configuration changes, it must be told to do so.

Review the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/pom.xml`. For the `greeting-config` application to pick up the configuration changes, it must include the `actuator` dependency.
The `actuator` adds several additional endpoints to the application for operational visibility and tasks that need to be carried out. In this case, we have added the actuator so that we can use the `/refresh` endpoint, which allows us to refresh the application config on demand.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Notify `greeting-config` app to pick up the new config by POSTing to the `greeting-config` `/refresh` endpoint. Open a new terminal window and execute the following:

```
$ curl -X POST http://<your greeting-config url>/refresh
```

Refresh your `greeting-config` application url while viewing the `greeting-config` terminal. You should see the debug line “Adding greeting”

(If you don’t see the change, verify that you have pushed the greeting-config.yml to BitBucket and repeat this step.)

Congratulations! You have used the `config-server` and `actuator` to change the logging level of the `greeting-config` application without restarting the `greeting-config` application.

Turning on a Feature with `@ConfigurationProperties`

Use of `@ConfigurationProperties` is a common way to externalize, group, and validate configuration in Spring applications. `@ConfigurationProperties` beans are automatically rebound when application config is refreshed.

1) Review `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/greeting/GreetingProperties.java`. Use of the `@ConfigurationProperties` annotation allows for reading of configuration values. Configuration keys are a combination of the `prefix` and the field names. In this case, there is one field (`displayFortune`).
Therefore `greeting.displayFortune` is used to turn the display of fortunes on/off. Remaining code is typical getter/setters for the fields.

```
@ConfigurationProperties(prefix="greeting")
public class GreetingProperties {
        private boolean displayFortune;
        public boolean isDisplayFortune() {
                return displayFortune;
        }
        public void setDisplayFortune(boolean displayFortune) {
                this.displayFortune = displayFortune;
        }
}
```

2) Review `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/greeting/GreetingController.java`. Note how the `greetingProperties.isDisplayFortune()` is used to turn the display of fortunes on/off. There are times when you want to turn features on/off on demand. In this case, we want the fortune feature “on” with our greeting.

```
@Controller
@EnableConfigurationProperties(GreetingProperties.class)
public class GreetingController {
        Logger logger = LoggerFactory.getLogger(GreetingController.class);
        @Autowired
        GreetingProperties greetingProperties;
        @Autowired
        FortuneService fortuneService;
        @RequestMapping("/")
        String getGreeting(Model model){
                logger.debug("Adding greeting");
                model.addAttribute("msg", "Greetings!!!");
                if(greetingProperties.isDisplayFortune()){
                        logger.debug("Adding fortune");
                        model.addAttribute("fortune", fortuneService.getFortu
ne());
                }
                //resolves to the greeting.vm velocity template
                return "greeting";
        }
}
```

3) Edit your `greeting-config.yml`. Add a `greeting.displayFortune` property as shown below and set it to `true`. Push the changes back to BitBucket.

```
security:
   basic:
     enabled: false
management:
   security:
     enabled: false
```

```
logging:
   level:
     io:
       pivotal: DEBUG
greeting:
   displayFortune: true # <----Change to true
```

4) Notify `greeting-config` app to pick up the new config by POSTing to the `/refresh` endpoint.

```
$ curl -X POST http://<your greeting-config url>/refresh
```

5) Then refresh your `greeting-config` application url and see the fortune included.

Congratulations! You have turned on a feature without restarting using the `config-server`, `actuator` and `@ConfigurationProperties`.

Override Configuration Values By Profile

1) Set the active profile to qa for the `greeting-config` application. In the example below, we use an environment variable to set the active profile.

```
$ cf set-env greeting-config SPRING_PROFILES_ACTIVE qa
```

2) Make sure the profile is set:

```
$ cf env greeting-config
```

3) In your `app-config` repository, create a new file: `greeting-config-qa.yml`. Fill it in with the following content:

```
greeting:
    displayFortune: false # <---- Use the opposite value as greeting-config.yml
```

Make sure to commit and push to BitBucket.

4) Re-start your `greeting-config` application. It will pick up the new environment variable and override values in `greeting-config.yml` with those in `greeting-config-qa.yml`. You should not see a quote displayed anymore.

5) You can play with the values in `greeting-config.yml` and `greeting-config-qa.yml` - remember to call the `/refresh` endpoint to pick up any config value changes. You can also play with the values of the SPRING_PROFILES_ACTIVE environment variable – remember to restart the app when you change environment variables.

What Just Happened?

Configuration from `greeting-config.yml` was overridden by a configuration file that was more specific (`greeting-config-qa.yml`).

Refreshing Application Configuration at Scale with Cloud Bus

Until now you have been notifying your application to pick up new configuration by POSTing to the `/refresh` endpoint.

When running several instances of your application, this poses several problems:

- Refreshing each individual instance is time consuming and too much overhead

- When running on Cloud Foundry you don't have control over which instances you hit when sending the POST request due to load balancing provided by the `router`

Cloud Bus addresses the issues listed above by providing a single endpoint to refresh all application instances via a pub/sub notification.

1) Create a RabbitMQ service instance, bind it to `greeting-config`

```
$ cf create-service p-rabbitmq standard cloud-bus
$ cf bind-service greeting-config cloud-bus
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. Our app doesn't need to be restaged. We will push it again with new functionality in a moment.

2) Include the cloud bus dependency in the `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/pom.xml`. *You will need to paste this in your file.*

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

3) Repackage the `greeting-config` application:

```
$ mvn clean package
```

4) Deploy the application and scale the number of instances.

```
$ cf push greeting-config -p target/greeting-config-0.0.1-SNAPSHOT.jar -i 3
```

5) Observe the logs that are generated by refreshing the `greeting-config /` endpoint several times in your browser and tailing the logs. Allow this process to run through the next few steps.

```
[mac, linux]

 $ cf logs greeting-config | grep GreetingController

[windows]

 $ cf logs greeting-config # then search output for "GreetingController"
```

All app instances are creating debug statements. Notice the `[App/X]`. It denotes which app instance is logging.

```
2015-09-28T20:53:06.07-0500 [App/2]      OUT 2015-09-29 01:53:06.071 DEBUG 34
--- [io-64495-exec-6] io.pivotal.greeting.GreetingController   : Adding fortun
e

2015-09-28T20:53:06.16-0500 [App/1]      OUT 2015-09-29 01:53:06.160 DEBUG 33
--- [io-63186-exec-5] io.pivotal.greeting.GreetingController   : Adding greeti
ng

2015-09-28T20:53:06.16-0500 [App/1]      OUT 2015-09-29 01:53:06.160 DEBUG 33
--- [io-63186-exec-5] io.pivotal.greeting.GreetingController   : Adding fortun
e

2015-09-28T20:53:06.24-0500 [App/1]      OUT 2015-09-29 01:53:06.246 DEBUG 33
--- [io-63186-exec-9] io.pivotal.greeting.GreetingController   : Adding greeti
ng

2015-09-28T20:53:06.24-0500 [App/1]      OUT 2015-09-29 01:53:06.247 DEBUG 33
--- [io-63186-exec-9] io.pivotal.greeting.GreetingController   : Adding fortun
e

2015-09-28T20:53:06.41-0500 [App/0]      OUT 2015-09-29 01:53:06.410 DEBUG 33
--- [io-63566-exec-3] io.pivotal.greeting.GreetingController   : Adding greeti
ng
```

7) Turn logging down. In your `app-config` repo edit the `greeting-config.yml`. Set the log level to `INFO`. Make sure to push back to BitBucket.

```
logging:

   level:

     io:
```

```
      pivotal: INFO
```

8) Notify applications to pickup the change. Open a new terminal window. Send a POST to the `greeting-config` `/bus/refresh` endpoint.

```
$ curl -X POST http://<your greeting-config url>/bus/refresh
```

9) Refresh the `greeting-config` `/` endpoint several times in your browser. No more logs!

10) Stop tailing logs from the `greeting-config` application.

11) Stop greeting-config

```
$ cf stop greeting-config
```

Back to TOP