Spring Cloud config with the Mule ESB

Mitch Dresdner

Table of Contents

A guide for integrating the Mule Spring Cloud connector with Spring Cloud Config configuration repository.

Summary

As we start pushing out our Mule runtime instances as container based solutions into the Cloud, we look for more creative ways to bind our MicroService based solutions to potentially changing endpoint locations.

Spring Cloud configuration offers us a simple secure solution for deriving our endpoint properties at startup, using a Git repository.

When an endpoint change is needed, the change can applied to the Git repository which Mule MicroServices will read at startup, obviating the need to redeploy the service.

Architecture

Abstracted parameters drive endpoint configuration and Mule supports several ways for working with properties, the most common approach being name/value pairs in YAML or Property files.

Other approaches can make use of setting properties in System environment variables and passing through the -D parameter into the JVM.

```
EXPORT resource-uri=https://resource.hots.io:8082
  or
... -Dresource-uri=https://resource.hots.io:8082
```

While all these approaches are all suitable for configuring Mule properties, in a dynamic MicroServices environment we prefer an approach with less friction and minimal reconfiguration. Spring Cloud Config provides just that, a server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

When Spring Cloud Config is used together with the Mule Spring Cloud Connector, it allows us to abstract our properties to a Git repository where our Mule application can read and apply them at startup.

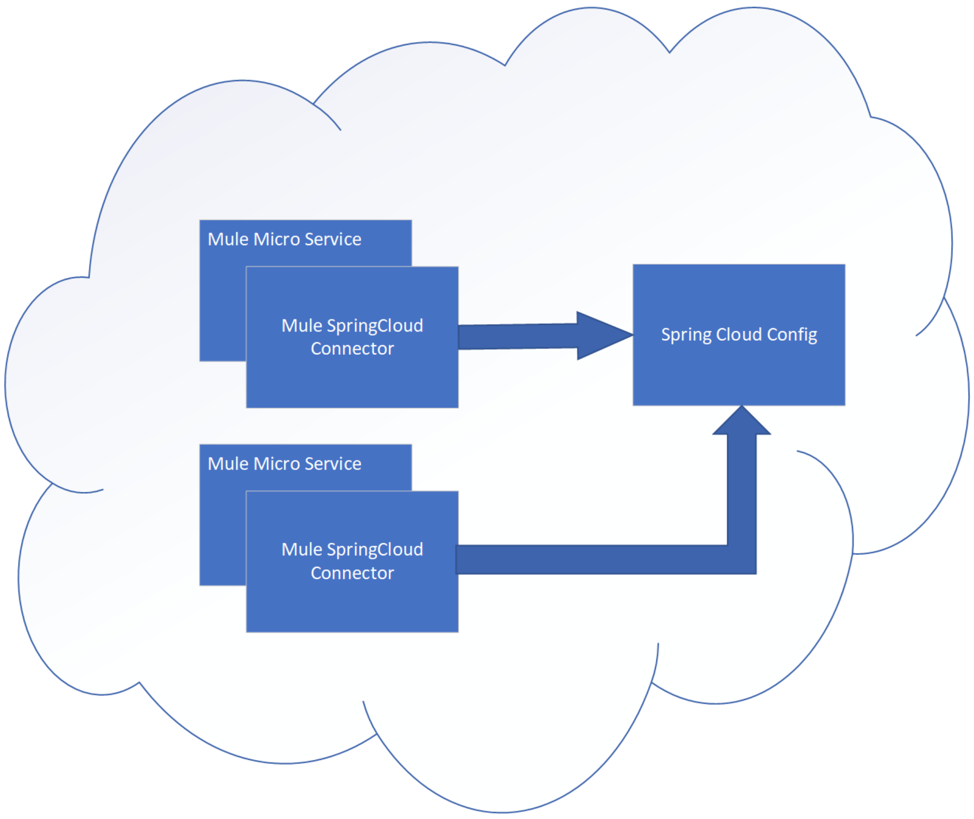


Figure 1. Mule interface with Spring Cloud Config

Properties imported into the Mule runtime environment from Spring Cloud config will be available when transports are initialized.

Adding the Spring Cloud Connector to Mule

With the overview out of the way, lets begin with the configuration of the Mule Spring Cloud connector and create a demo application to show how all of the pieces fit together. The connector will plug into Mule AnypointStudio and deploy with the runtime solution. It will provide our interface to the Spring Cloud Config Server.

First we'll clone the git repository containing the Mule Spring Cloud connector, build the connector and install into AnypointStudio.

Building a standalone Spring Cloud connector instance

Start by cloning the Mule Spring Cloud connector instance

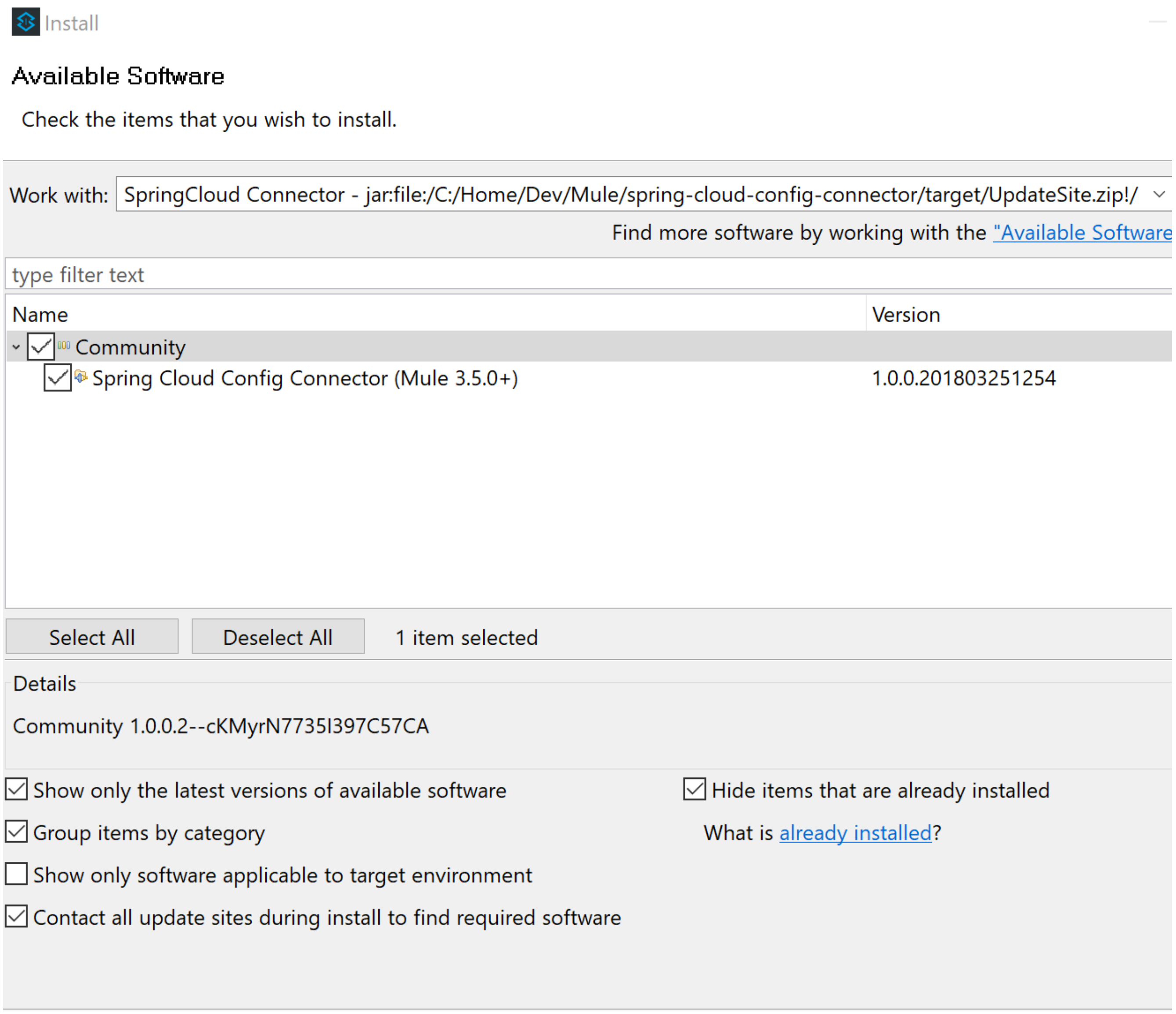
- Change into the source folder
- Build and install the connector

```
git clone https://github.com/mulesoft-labs/spring-cloud-config-connector.git
cd spring-cloud-config-connector
mvn clean install -Ddevkit.studio.package.skip=false
```

Import the connector into Mule

When the build has completed you'll find a file called UpdateSite.zip in the target folder

- In AnypointStudio select Menu options: **Help → Install New Software**
- Press the **Add…** Button
- Choose and name and the location of **UpdateSite.zip**



Work with: SpringCloud Connector - jar:file:/C:/Home/Dev/Mule/spring-cloud-config-connector/target/UpdateSite.zip!/

Find more software by working with the "Available Software

type filter text

Name	Version
☑ Community	
☑ Spring Cloud Config Connector (Mule 3.5.0+)	1.0.0.201803251254

1 item selected

Details

Community 1.0.0.2--cKMyrN7735I397C57CA

☑ Show only the latest versions of available software

☑ Group items by category

☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

☑ Hide items that are already installed

What is already installed?

Figure 2. Install Mule Cloud Config Connector

Complete the installation of the connector.

Creating a sample Mule project

We'll begin our Mule project configuraton by adding the Spring Cloud Config connector to the Global settings.

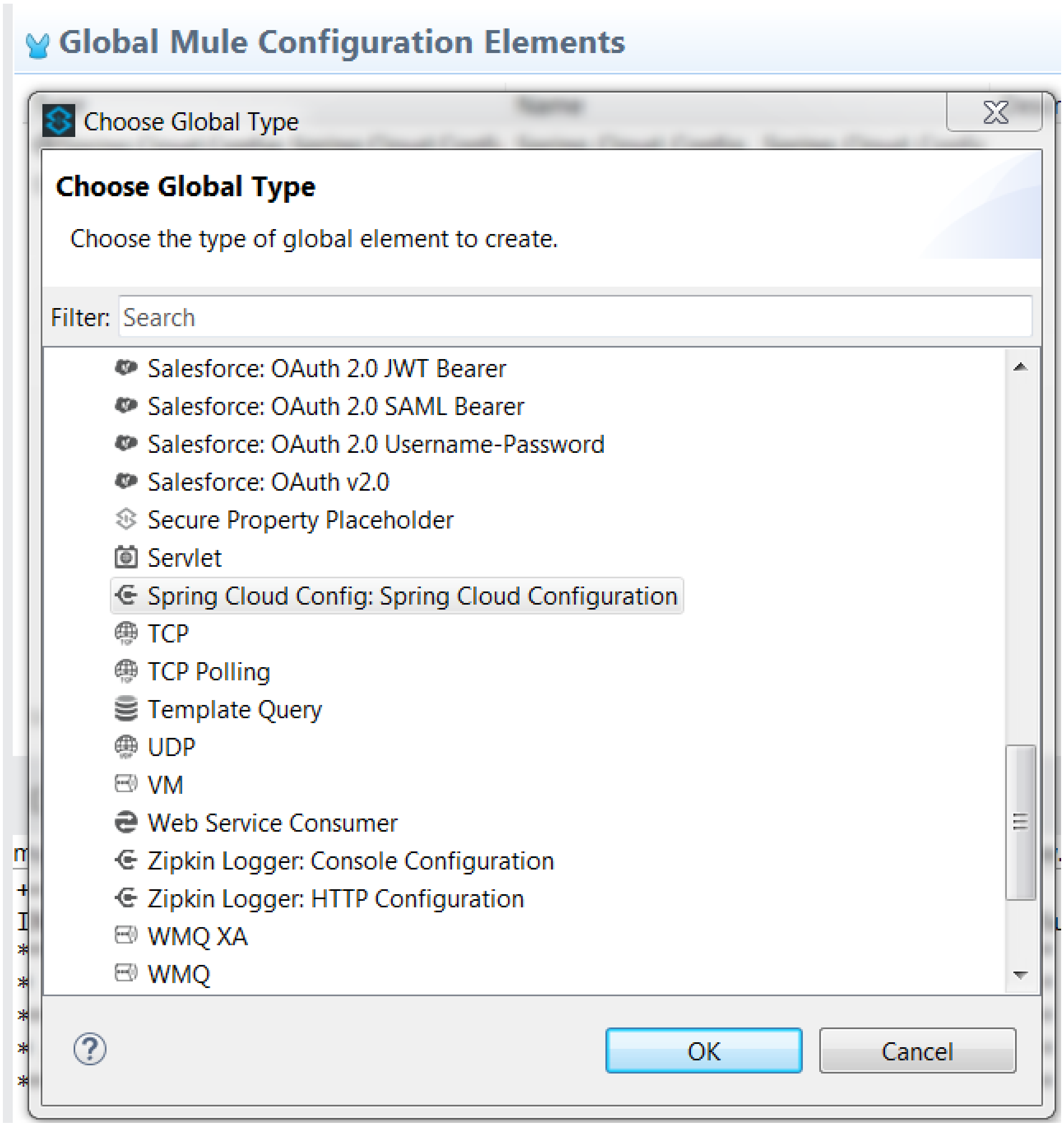
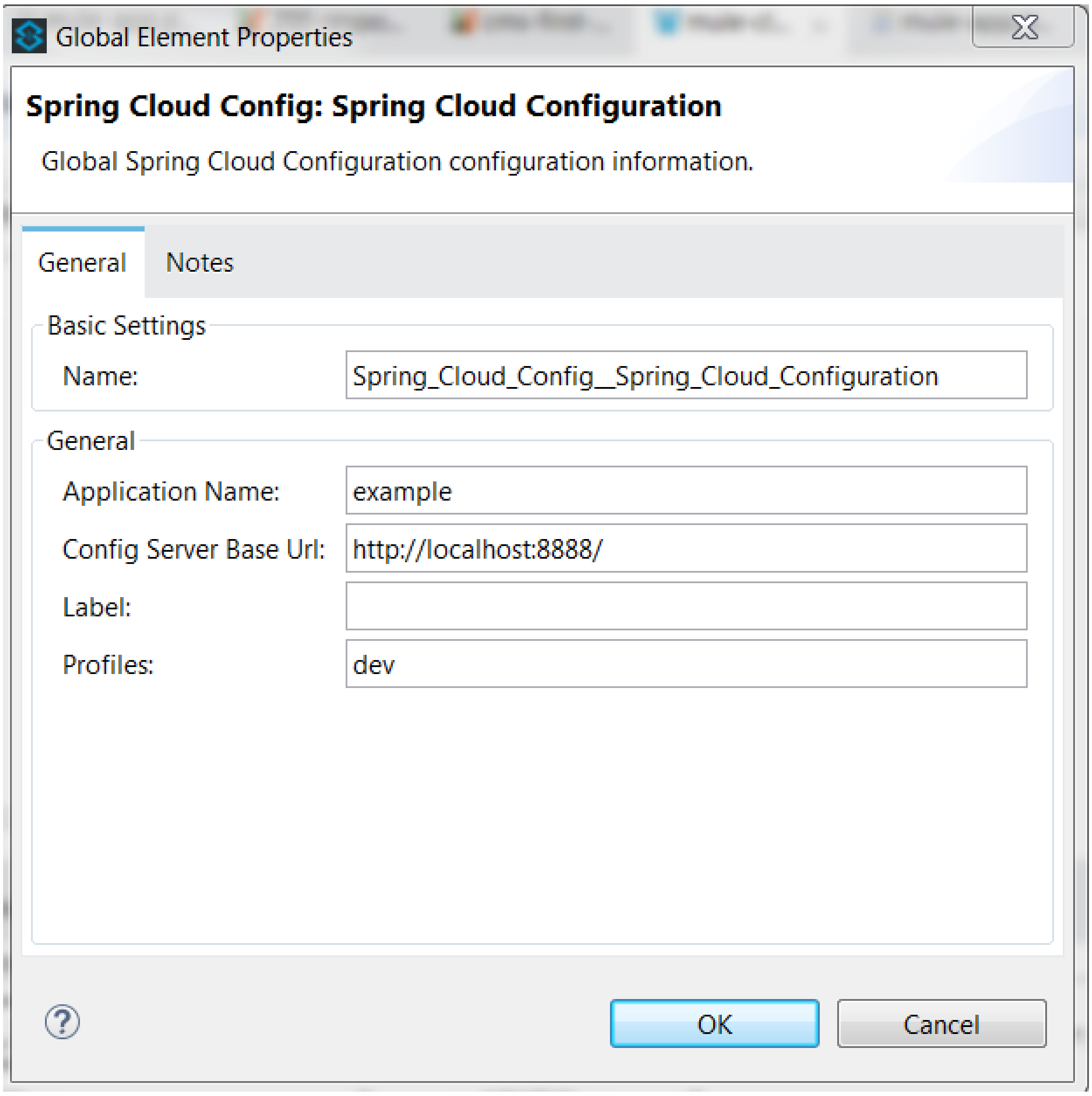

Global Mule Configuration Elements
Choose Global Type
Choose Global Type
Choose the type of global element to create.
Filter: Search
Salesforce: OAuth 2.0 JWT Bearer
Salesforce: OAuth 2.0 SAML Bearer
Salesforce: OAuth 2.0 Username-Password
Salesforce: OAuth v2.0
Secure Property Placeholder
Servlet
Spring Cloud Config: Spring Cloud Configuration
TCP
TCP Polling
Template Query
UDP
VM
Web Service Consumer
Zipkin Logger: Console Configuration
Zipkin Logger: HTTP Configuration
WMQ XA
WMQ
OK
Cancel

Figure 3. Create the Spring Cloud Config connector settings

Select the Spring Cloud Config connector from undef Connector Configuration and select the Ok button. This adds the settings to the Globals display.

Edit the Spring Cloud Config settings to enter the application name, URL and Profile. These values correspond to the name of the property file you create later for the Git repository configuration. Application Name will match the first part of the property file naming convention, the Profile will match with the second part of the naming convention. Together the two will be matched as: **example-dev.properties**.



Global Element Properties

Spring Cloud Config: Spring Cloud Configuration

Global Spring Cloud Configuration configuration information.

Basic Settings

Name: Spring_Cloud_Config__Spring_Cloud_Configuration

General

Application Name: example

Config Server Base Url: http://localhost:8888/

Label:

Profiles: dev

Figure 4. Global settings for Spring Cloud Config

With the Global configurations complete, we move on to the configuration of the Mule Flow.

The Mule implementation demonstrates a simple flow consisting of the following:

- An HTTP endpoint to trigger the initiation of the flow
- A logger statement to display fetched property

```
<spring-cloud-config:config
  name="Spring_Cloud_Config__Spring_Cloud_Configuration"
  applicationName="example"
  profiles="dev"
  doc:name="Spring Cloud Config: Spring Cloud Configuration"/>

  <http:listener-config name="HTTP_Listener_Configuration"
   host="0.0.0.0" port="${mule.http.port}"
   doc:name="HTTP Listener Configuration"/>


  <flow name="mule-cloud-configFlow">
    <http:listener config-ref="HTTP_Listener_Configuration"
    path="/foo" doc:name="HTTP"/>

    <-- Debug to output properties
        <spring-cloud-config:dump-configuration
         config-ref="Spring_Cloud_Config__Spring_Cloud_Configuration"
         doc:name="Spring Cloud Config"/>
     -->

     <logger message="Property: ActiveMQ URI = ${activemq.url}"
      level="INFO" doc:name="Logger"/>

  </flow>
```

With the Mule flow complete we move on to the creation of the SpringBoot component.

Creating the SpringConfig Server

The SpringConfig Server will be a simple SpringBoot project which will look for property dependencies in a Git repository. Let’s start by creating the Git repository adding a property file and commiting the changes.

Creating the Git Repository for properties

```
cd \home\Dev
mkdir git-localconfig-repo
cd git-localconfig-repo

# Initialize the Git repository
git init
```

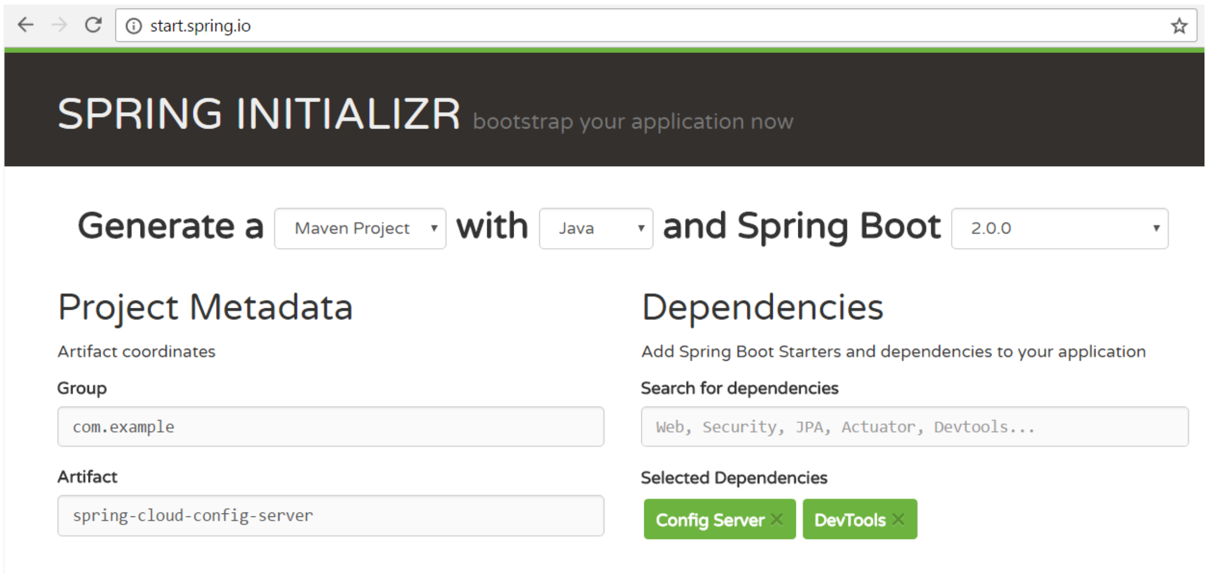
Using your favorite editor create a property file with the following sample properties:

```
# Git Repository location is \home\Dev\git-localconfig-repo

# Use your favorite editor to create the property file below, im going to cheat and
use cat in my git bash shell
cat > example-dev.properties
################################
#  ActimeMQ server properties  #
################################
activemq.url=tcp://localhost:61616

################################
# HTTP Properties              #
################################
mule.http.port=8083
^D
```

The property file name **example-dev.properties** is significant. The first part **example** equates to application name which you added earlier in the Mule global property configuration for SpringConfig, the values after the dash (**dev**) represent the profile names for the properties, which can be a comma separated list of profiles to be read from the repository. Each Profile will match to a corresponding Property file in Git.

Now that we have a property file in a local Git repository we'll commit the changes and move on to creating the SpringConfig Server.

To create our SpringBoot Cloud Config Server project, start at http://start.spring.io/



Generate a Maven Project with Java and Spring Boot 2.0.0

Project Metadata
Artifact coordinates
Group: com.example
Artifact: spring-cloud-config-server

Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies: Web, Security, JPA, Actuator, Devtools...
Selected Dependencies: Config Server, DevTools

Figure 5. Create a SpringBoot Project

With the SpringConfig Server created we'll add the necessary pieces to create the server and bind to our Git properties.

SpringConfig Server settings

- Enable the server with @EnableConfigServer
- Define the server property configuration

```
@EnableConfigServer
@SpringBootApplication
public class Spring CloudConfigServerApplication {

  public static void main(String[] args) {
    SpringApplication.run(Spring CloudConfigServerApplication.class, args);
  }
}
```

Property file configuration

```
# application.properties
spring.application.name=spring-cloud-config-server

# Default port for Spring Cloud Config Server
server.port=8888

# Define the location of our Git repo
spring.cloud.config.server.git.uri=file:///Home/Dev/git-localconfig-repo/
```

Now the the changes are in place for the SpringConfig Server, let's start it up and access the property settings from our Mule application

Start our SpringConfig Server with maven

```
mvn spring-boot:run
```

Next start the Mule flow and trigger the flow to review the results of the SpringConfig Server integration.

Notice that the HTTP Flow starter will derive it's property setting from the Spring Cloud Config Server, initiate the Flow and the Logger statement will print the property value obtained for the ActiveMQ server.

I hope you enjoyed this article as much as I have writting it and look forward to your feedback.

About the Author:

Mitch Dresdner is a Senior Mule Consultant at TerraThink