DevOps Workshop Lab Guide

Christopher Phillipson

Version 1.0, 2017-10-12

Table of Contents

Labs

Welcome to the lab! Here you will find a collection of exercises and accompanying source-code.

Overview

This workshop contains a number of lab folders meant to be worked through in numerical order - as each exercise builds upon the last. The only exception to this sequence is the **00** folder. This folder references sample applications in **samples**. These applications can be pushed to Cloud Foundry at any time.

Your workspace is the **my_work** folder. If you get stuck implementing any of the labs, **solutions** are available for your perusal.

PCF Environment Access

Account set up

1. If you do not have an account yet, please see the instructor

Target the Environment

1. If you haven't already, download the latest release of the Cloud Foundry CLI from https://github.com/cloudfoundry/cli/releases for your operating system and install it.

2. Set the API target for the CLI (set appropriate end point for your environment) and login:

   PWS

   ```
   $ cf api https://api.run.pivotal.io --skip-ssl-validation
   $ cf login
   ```

   Enter your account username and password, then select an org and space.

Apps Manager UI

1. An alternative to installing the CF CLI is via your PCF Apps Manager interface.

2. Navigate in a web browser to (depending on environment):

   PWS

   ```
   https://console.run.pivotal.io
   ```

3. Login to the interface with your email and password

→ The password will be supplied to you by the instructor

4. Click the 'Tools' link, and download the CLI matching your operating system

Introduction to CF CLI

- Change the working directory to be *devops-workshop-boeing/labs/samples*

  Note the sub-directories present..

  ```
  samples
  ├──── coldfusion-example
  ├──── dotnet-core-sample
  ├──── go-sample
  ├──── nodejs-sample
  ├──── python-sample
  ```

  → If you want to be able to deploy these samples you must have *Go*, *Node*, *.Net Core*, and *Python* installed.

How to target a foundation and login

1. Open a Terminal (e.g., *cmd* or *bash* shell)

2. Target a foundation and login

   PWS

   ```
   $ cf api https://api.run.pivotal.io --skip-ssl-validation
   $ cf login
   ```

   Enter your account username and password, then select an org and space.

How to deploy an application

1. Let's take a look at the CF CLI options

   ```
   cf --help
   ```

2. Let's see what buildpacks are available to us

   ```
   cf buildpacks
   ```

3. Peruse the services you can provision and bind your applications to

   ```
   cf marketplace
   ```

4. Time to deploy an app. How about Node.js?

```
cd nodejs-sample
cf push -c "node server.js"
```

If you are having trouble resolving artifacts, you are likely running in a disconnected environment, so follow these steps and try pushing the app again.

```
yarn config set yarn-offline-mirror ./npm-packages-offline-cache
cp ~/.yarnrc .
rm -rf node_modules/ yarn.lock
yarn install
```

5. Next, let's try deploying a Python app.

```
cd ../python-sample
cf push my_pyapp
```

6. Rinse and repeat for .Net Core and Go apps

```
cd ../dotnet-core-sample
cf push
cd ../go-sample
cf push awesome-go-app -m 64M --random-route
```

7. And for our final trick, how about deploying a Cold Fusion app?

```
cd.. coldfusion-example
```

Grab this file, unpack into src/main/webapp folder with

```
unzip -q cfmagic.zip ./src/main/webapp
```

Then execute

```
gradle war
cf push
```

8. Check what applications have been deployed so far

```
cf apps
```

→ Take some time to visit each of the applications you've just deployed.

9. Let's stop an app, then check that it has indeed been stopped

```
cf stop cf-nodejs
cf apps
```

How to cleanup after yourself

1. Finally, let's delete an app

```
cf delete cf-nodejs
```

→ Repeat `cf delete` for each app you deployed.

Where to go for more help

→ Getting Started with the CF CLI

→ Cloud Foundry Cheat Sheet

Building a Spring Boot Application

In this lab we'll build and deploy a simple Spring Boot application to Cloud Foundry whose sole purpose is to reply with a standard greeting.

Getting started

While we could visit https://start.spring.io to create a new Spring Boot project, we will start with a skeleton

1. Open a Terminal (e.g., *cmd* or *bash* shell)

2. Change the working directory to be *devops-workshop-boeing/labs/my_work/cloud-native-spring*

    ```
    cd /devops-workshop-boeing/labs/my_work/cloud-native-spring
    ```

3. Open this project in your editor/IDE of choice.

    STS Import Help

    Select *File > Import....* In the susequent dialog choose *Gradle > Existing Gradle Project* then click the *Next* button. In the *Import Gradle Project* dialog browse to the *cloud-native-spring* directory (e.g. *devops-workshop-boeing/labs/my_work/cloud-native-spring*) then click the *Open* button, then click the *Finish* button.

Add an Endpoint

Within your editor/IDE complete the following steps:

1. Create a new package *io.pivotal.controller* underneath *src/main/java*.

2. Create a new class named *GreetingController* in the aforementioned package.

3. Add an *@RestController* annotation to the class *io.pivotal.controller.GreetingController* (i.e., */cloud-native-spring/src/main/java/io/pivotal/controller/GreetingController.java*).

    ```
    package io.pivotal.controller;

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

    @RestController
    public class GreetingController {

    }
    ```

4. Add the following request handler to the class *io.pivotal.controller.GreetingController* (i.e., */cloud-native-spring/src/main/java/io/pivotal/controller/GreetingController.java*).

```
@RequestMapping("/")
public String hello() {
    return "Hello World!";
}
```

Completed:

```
package io.pivotal.controller;

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

@RestController
public class GreetingController {

    @RequestMapping("/hello")
    public String hello() {
        return "Hello World!";
    }

}
```

Build the *cloud-native-spring* application

Return to the Terminal session you opened previously and make sure your working directory is set to be *devops-workshop-boeing/labs/my_work/cloud-native-spring*

1. Find out what tasks are available to you with

   ```
   gradle tasks
   ```

2. First we'll run tests

   ```
   gradle test
   ```

3. Next we'll package the application as a libary artifact (it cannot be run on its own)

   ```
   gradle jar
   ```

4. Next we'll package the application as an executable artifact (that can be run on its own because it will include all transitive dependencies along with embedding a servlet container)

```
gradle bootRepackage
```

Run the *cloud-native-spring* application

Now we're ready to run the application

1. Run the application with

```
gradle bootRun
```

2. You should see the application start up an embedded Apache Tomcat server on port 8080 (review terminal output):

```
2015-10-02 13:26:59.264  INFO 44749 --- [lication.main()]
s.b.c.e.t.TomcatEmbeddedServletContainer: Tomcat started on port(s): 8080 (http)
2015-10-02 13:26:59.267  INFO 44749 --- [lication.main()]
io.pivotal.CloudNativeSpringApplication: Started CloudNativeSpringApplication in
2.541 seconds (JVM running for 9.141)
```

3. Browse to http://localhost:8080/hello

4. Stop the *cloud-native-spring* application. In the terminal window type **Ctrl** + **C**

Deploy *cloud-native-spring* to Pivotal Cloud Foundry

We've built and run the application locally. Now we'll deploy it to Cloud Foundry.

1. Create an application manifest in the root folder *devops-workshop-boeing/labs/my_work/cloud-native-spring*

```
touch manifest.yml
```

2. Add application metadata, using a text editor (of choice)

```
---
applications:
- name: cloud-native-spring
  random-route: true
  instances: 1
  path: ./build/libs/cloud-native-spring-1.0-SNAPSHOT-exec.jar
  buildpack: java_buildpack
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
```

3. Push application into Cloud Foundry

```
cf push
```

→ To specify an alternate buildpack, you could update the above to be e.g.,

```
cf push -f manifest.yml -b java_buildpack_offline
```

Assuming the offline buildpack was installed and available for use with your targeted foundation. You can check for which buildpacks are available by executing

```
cf buildpacks
```

4. Find the URL created for your app in the health status report. Browse to your app's /hello endpoint.

5. Check the log output

```
cf logs cloud-native-spring --recent
```

Congratulations! You've just completed your first Spring Boot application.

Adding Persistence to Boot Application

In this lab we'll utilize Spring Boot, Spring Data, and Spring Data REST to create a fully-functional hypermedia-driven RESTful web service. We'll then deploy it to Pivotal Cloud Foundry.

Create a Hypermedia-Driven RESTful Web Service with Spring Data REST (using JPA)

This application will allow us to create, read update and delete records in an in-memory relational repository. We'll continue building upon the Spring Boot application we built out in Lab 1. The first stereotype we will need is the domain model itself, which is `City`.

Add the domain object - City

1. Create the package `io.pivotal.domain` and in that package create the class `City`. Into that file you can paste the following source code, which represents cities based on postal codes, global coordinates, etc:

```
package io.pivotal.domain;

@Data
@Entity
@Table(name="city")
public class City implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String county;

    @Column(nullable = false)
    private String stateCode;

    @Column(nullable = false)
    private String postalCode;

    @Column
    private String latitude;

    @Column
    private String longitude;

}
```

Notice that we're using JPA annotations on the class and its fields. We're also employing Lombok, so we don't have to write a bunch of boilerplate code (e.g., getter and setter methods). You'll need to use your IDE's features to add the appropriate import statements.

→ Hint: imports should start with `javax.persistence` and `lombok`

2. Create the package `io.pivotal.repositories` and in that package create the interface `CityRepository`. Paste the following code and add appropriate imports:

```
package io.pivotal.repositories;

@RepositoryRestResource(collectionResourceRel = "cities", path = "cities")
public interface CityRepository extends PagingAndSortingRepository<City, Long> {
}
```

You'll need to use your IDE's features to add the appropriate import statements.

→ Hint: imports should start with `org.springframework.data.rest.core.annotation` and `org.springframework.data.repository`

Run the *cloud-native-spring* Application

1. Return to the Terminal session you opened previously

2. Run the application

```
gradle clean bootRun
```

3. Access the application using `curl` or your web browser using the newly added REST repository endpoint at http://localhost:8080/cities. You'll see that the primary endpoint automatically exposes the ability to page, size, and sort the response JSON.

```
curl -i http://localhost:8080/cities

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/hal+json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 28 Apr 2016 14:44:06 GMT

{
  "_embedded" : {
    "cities" : [ ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/cities"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile/cities"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 0,
    "totalPages" : 0,
    "number" : 0
  }
}
```

4. To exit the application, type **Ctrl-C**.

So what have you done? Created four small classes and one build file, resulting in a fully-functional REST microservice. The application's `DataSource` is created automatically by Spring Boot using the in-memory database because no other `DataSource` was detected in the project.

Next we'll import some data.

Importing Data

1. Add this import.sql file found in **devops-workshop-boeing/labs/** to `src/main/resources`. This is a rather large dataset containing all of the postal codes in the United States and its territories. This file will automatically be picked up by Hibernate and imported into the in-memory database.

2. Restart the application.

```
gradle clean bootRun
```

3. Access the application again. Notice the appropriate hypermedia is included for `next`, `previous`, and `self`. You can also select pages and page size by utilizing `?size=n&page=n` on the URL string. Finally, you can sort the data utilizing `?sort=fieldName` (replace fieldName with a cities attribute).

```
curl -i localhost:8080/cities

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Application-Context: application
Content-Type: application/hal+json
Transfer-Encoding: chunked
Date: Tue, 27 May 2014 19:59:58 GMT

{
  "_links" : {
    "next" : {
      "href" : "http://localhost:8080/cities?page=1&size=20"
    },
    "self" : {
      "href" : "http://localhost:8080/cities{?page,size,sort}",
      "templated" : true
    }
  },
  "_embedded" : {
    "cities" : [ {
      "name" : "HOLTSVILLE",
      "county" : "SUFFOLK",
      "stateCode" : "NY",
      "postalCode" : "00501",
      "latitude" : "+40.922326",
      "longitude" : "-072.637078",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/cities/1"
        }
      }
```

```
    },

    // ...

    {
      "name" : "CASTANER",
      "county" : "LARES",
      "stateCode" : "PR",
      "postalCode" : "00631",
      "latitude" : "+18.269187",
      "longitude" : "-066.864993",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/cities/20"
        }
      }
    } ]
  },
  "page" : {
    "size" : 20,
    "totalElements" : 42741,
    "totalPages" : 2138,
    "number" : 0
  }
}
```

4. Try the following URL Paths with `curl` to see how the application behaves:

   http://localhost:8080/cities?size=5

   http://localhost:8080/cities?size=5&page=3

   http://localhost:8080/cities?sort=postalCode,desc

Next we'll add searching capabilities.

Adding Search

1. Let's add some additional finder methods to `CityRepository`:

```
@RestResource(path = "name", rel = "name")
Page<City> findByNameIgnoreCase(@Param("q") String name, Pageable pageable);

@RestResource(path = "nameContains", rel = "nameContains")
Page<City> findByNameContainsIgnoreCase(@Param("q") String name, Pageable
pageable);

@RestResource(path = "state", rel = "state")
Page<City> findByStateCodeIgnoreCase(@Param("q") String stateCode, Pageable
pageable);

@RestResource(path = "postalCode", rel = "postalCode")
Page<City> findByPostalCode(@Param("q") String postalCode, Pageable pageable);

@Query(value ="select c from City c where c.stateCode = :stateCode")
Page<City> findByStateCode(@Param("stateCode") String stateCode, Pageable
pageable);
```

→ Hint: imports should start with `org.springframework.data.domain`, `org.springframework.data.rest.core.annotation`, `org.springframework.data.repository.query`, and `org.springframework.data.jpa.repository`

2. Run the application

```
gradle clean bootRun
```

3. Access the application again. Notice that hypermedia for a new `search` endpoint has appeared.

```
curl -i "localhost:8080/cities"

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Application-Context: application
Content-Type: application/hal+json
Transfer-Encoding: chunked
Date: Tue, 27 May 2014 20:33:52 GMT

// prior omitted
    },
    "_links": {
        "first": {
            "href": "http://localhost:8080/cities?page=0&size=20"
        },
        "self": {
            "href": "http://localhost:8080/cities{?page,size,sort}",
            "templated": true
        },
        "next": {
            "href": "http://localhost:8080/cities?page=1&size=20"
        },
        "last": {
            "href": "http://localhost:8080/cities?page=2137&size=20"
        },
        "profile": {
            "href": "http://localhost:8080/profile/cities"
        },
        "search": {
            "href": "http://localhost:8080/cities/search"
        }
    },
    "page": {
        "size": 20,
        "totalElements": 42741,
        "totalPages": 2138,
        "number": 0
    }
}
```

4. Access the new `search` endpoint:

   http://localhost:8080/cities/search

```
curl -i "localhost:8080/cities/search"

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Application-Context: application
Content-Type: application/hal+json
Transfer-Encoding: chunked
Date: Tue, 27 May 2014 20:38:32 GMT

{
    "_links": {
        "postalCode": {
            "href":
"http://localhost:8080/cities/search/postalCode{?q,page,size,sort}",
            "templated": true
        },
        "state": {
            "href": "http://localhost:8080/cities/search/state{?q,page,size,sort}",
            "templated": true
        },
        "nameContains": {
            "href":
"http://localhost:8080/cities/search/nameContains{?q,page,size,sort}",
            "templated": true
        },
        "name": {
            "href": "http://localhost:8080/cities/search/name{?q,page,size,sort}",
            "templated": true
        },
        "findByStateCode": {
            "href":
"http://localhost:8080/cities/search/findByStateCode{?stateCode,page,size,sort}",
            "templated": true
        },
        "self": {
            "href": "http://localhost:8080/cities/search"
        }
    }
}
```

Note that we now have new search endpoints for each of the finders that we added.

5. Try a few of these endpoints in Postman. Feel free to substitute your own values for the parameters.

http://localhost:8080/cities/search/postalCode?q=75202

http://localhost:8080/cities/search/name?q=Boston

http://localhost:8080/cities/search/nameContains?q=Fort&size=1

→ For further details on what's possible with Spring Data JPA, consult the reference documentation

Pushing to Cloud Foundry

1. Build the application

```
gradle bootRepackage
```

2. You should already have an application manifest, `manifest.yml`, created in Lab 1; this can be reused. You'll want to add a timeout param so that our service has enough time to initialize with its data loading:

```
---
applications:
- name: cloud-native-spring
  random-route: true
  memory: 1024M
  instances: 1
  path: ./build/libs/cloud-native-spring-1.0-SNAPSHOT-exec.jar
  buildpack: java_buildpack
  timeout: 180 # to give time for the data to import
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
```

3. Push to Cloud Foundry:

```
cf push

...

Showing health and status for app cloud-native-spring in org zoo-labs / space
development as cphillipson@pivotal.io...
OK

requested state: started
instances: 1/1
usage: 1G x 1 instances
urls: cloud-native-spring-apodemal-hyperboloid.cfapps.io
last uploaded: Thu Sep 28 23:29:21 UTC 2017
stack: cflinuxfs2
buildpack: java_buildpack

     state     since                    cpu      memory         disk
details
#0   running   2017-09-28 04:30:22 PM   163.7%   395.7M of 1G   159M of 1G
```

4. Access the application at the random route provided by CF:

```
curl -k https://cloud-native-spring-{random-word}.{domain}.com/cities
```

Binding to a MySQL database in Cloud Foundry

1. You may have noticed that when using the H2 (in-memory) database that the schema is automatically generated and updated on application start. That's not the case with other SQL databases, such as MySQL, Oracle or PostgreSQL. You will need to instruct the application to automatically generate SQL schema by adding to src/main/resources/application.yml file, e.g.

```
spring:
  jpa:
    hibernate:
      ddl-auto: update
```

2. Repackage the application again

```
gradle bootRepackage
```

3. Let's create a MySQL database instance. Hopefully, you will have p-mysql service available in CF Marketplace.

```
cf marketplace -s p-mysql
```

Expected output:

```
Getting service plan information for service p-mysql as cphillipson@pivotal.io...
OK

service plan    description     free or paid
100mb           100MB default   free
```

4. Let's create an instance of p-mysql with 100mb plan, e.g.

```
cf create-service p-mysql 100mb mysql-database
```

Expected output:

```
Creating service instance mysql-database1 in org zoo-labs / space development as
cphillipson@pivotal.io...
OK
```

5. Let's bind the service to the application, e.g.

```
cf bind-service cloud-native-spring mysql-database
```

Expected output:

```
Binding service mysql-database to app cloud-native-spring in org zoo-labs / space
development as cphillipson@pivotal.io...
OK
```

→ Tip: Use `cf restage cloud-native-spring` to ensure your env variable changes take effect

6. Let's push the application, since we did not push the updated application yet, e.g.

```
cf push
```

7. You may wish to observe the logs and notice that the bound MySQL database is picked up by the application, e.g.

```
cf logs cloud-native-spring --recent
```

Sample output:

```
...
INFO 20 --- [           main] org.hibernate.Version                    : HHH000412:
Hibernate Core {5.0.12.Final}
INFO 20 --- [           main] org.hibernate.cfg.Environment            : HHH000206:
hibernate.properties not found
INFO 20 --- [           main] org.hibernate.cfg.Environment            : HHH000021:
Bytecode provider name : javassist
INFO 20 --- [           main] o.hibernate.annotations.common.Version   :
HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
INFO 20 --- [           main] org.hibernate.dialect.Dialect            : HHH000400:
Using dialect: org.hibernate.dialect.MySQLDialect
INFO 20 --- [           main] org.hibernate.tool.hbm2ddl.SchemaUpdate  : HHH000228:
Running hbm2ddl schema update
...
```

8. You could also bind to the database directly from the `manifest.yml` file, e.g.

```
applications:
- name: cloud-native-spring
  random-route: true
  memory: 1024M
  instances: 1
  path: ./build/libs/cloud-native-spring-1.0-SNAPSHOT-exec.jar
  buildpack: java_buildpack
  timeout: 180 # to give time for the data to import
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
  services:
    - mysql-database
```

9. Attempt to push the app again after making this update

```
cf push
```

10. Also, if you connect to the database, you will notice that the application has updated the schema on application start. You should see `city` table in the MySQL database.

```
cf mysql mysql-database
```

Sample console:

```
mysql: [Warning] Using a password on the command line interface can be insecure.
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 50
Server version: 5.5.5-10.1.24-MariaDB Source distribution

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show tables;
+------------------------------------------------+
| Tables_in_cf_2c143464_fbdd_43df_8379_2bef727f1c99 |
+------------------------------------------------+
| city                                           |
+------------------------------------------------+
1 row in set (0.03 sec)

mysql> select * from city;
Empty set (0.03 sec)
```

In case you don't have CF `mysql` plugin, you can install the plugin with

```
cf install-plugin -r "CF-Community" mysql-plugin
```

→ More details here:

- https://plugins.cloudfoundry.org/

- https://github.com/andreasf/cf-mysql-plugin

11. Notice that the database is not initialized same way it was with H2 (in-memory) database. We need to update `spring.jpa.hibernate.ddl-auto` property to `create` or `create-drop`, since `update` will only update the schema. See more details here https://docs.spring.io/spring-boot/docs/current/reference/html/howto-database-initialization.html

12. Value tables and data are typically imported and managed by schema evolution tools like Flyway or Liquibase, but a review of those tools is out of scope for this workshop.

Enhancing Boot Application with Metrics

Set up the Actuator

Spring Boot includes a number of additional features to help you monitor and manage your application when it's pushed to production. These features are added by adding *spring-boot-starter-actuator* to the classpath. Our initial project setup already included it as a dependency.

1. Verify the Spring Boot Actuator dependency the in following file: **/cloud-native-spring/build.gradle** You should see the following dependency in the list:

```
dependencies {
    compile('org.springframework.boot:spring-boot-starter-actuator')
    // other dependencies omitted
}
```

   By default Spring Boot will use Spring Security to protect these management endpoints (which is a good thing!). Though you wouldn't want to disable this in production, we'll do so in this sample app to make demonstration a bit easier and simpler.

2. Add the following properties to **cloud-native-spring/src/main/resources/application.yml**.

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

3. Run the updated application

```
gradle clean bootRun
```

   Try out the following endpoints with [Postman](). The output is omitted here because it can be quite large:

   http://localhost:8080/health

   → Displays Application and Datasource health information. This can be customized based on application functionality, which we'll do later.

   http://localhost:8080/beans

   → Dumps all of the beans in the Spring context.

   http://localhost:8080/autoconfig

   → Dumps all of the auto-configuration performed as part of application bootstrapping.

http://localhost:8080/configprops

→ Displays a collated list of all @ConfigurationProperties.

http://localhost:8080/env

→ Dumps the application's shell environment as well as all Java system properties.

http://localhost:8080/mappings

→ Dumps all URI request mappings and the controller methods to which they are mapped.

http://localhost:8080/dump

→ Performs a thread dump.

http://localhost:8080/trace

→ Displays trace information (by default the last few HTTP requests).

4. Stop the *cloud-native-spring* application.

Include Version Control Info

Spring Boot provides an endpoint (http://localhost:8080/info) that allows the exposure of arbitrary metadata. By default, it is empty.

One thing that *actuator* does well is expose information about the specific build and version control coordinates for a given deployment.

1. Edit the following file: **/cloud-native-spring/build.gradle** Add the gradle-git-properties plugin to your Gradle build.

   First, you'll need to be able to resolve the plugin so add the following to the *repositories{}* section of the *buildscript{}* block.

   ```
   maven {
       url "https://plugins.gradle.org/m2/"
   }
   ```

   Then, you must edit the file and add the plugin dependency within the *dependencies{}* section of the *buildscript{}* block.

   ```
   classpath("gradle.plugin.com.gorylenko.gradle-git-properties:gradle-git-
   properties:1.4.17")
   ```

   You'll also activate the plugin and add a *gitProperties{}* block just underneath the last *apply plugin:* line.

```
apply plugin: 'com.gorylenko.gradle-git-properties'

gitProperties {
    dateFormat = "yyyy-MM-dd'T'HH:mmZ"
    dateFormatTimeZone = "UTC"
    gitRepositoryRoot = new File("${project.rootDir}/../../..")
}
```

→ Note too that we are updating the path to the *.git* directory.

The effect of all this configuration is that the *gradle-git-properties* plugin adds Git branch and commit coordinates to the **/info** endpoint.

2. Run the *cloud-native-spring* application:

```
gradle clean bootRun
```

3. Browse to http://localhost:8080/info. Git commit information is now included

```
{
    "git": {
        "commit": {
            "time": "2017-09-07T13:52+0000",
            "id": "3393f74"
        },
        "branch": "master"
    }
}
```

4. Stop the *cloud-native-spring* application

   What Just Happened?

   By including the *gradle-git-properties* plugin, details about git commit information will be included in the **/info** endpoint. Git information is captured in a *git.properties* file that is generated with the build. Review the following file: **/cloud-native-spring/build/resources/main/git.properties**

Include Build Info

1. Add the following properties to **cloud-native-spring/src/main/resources/application.yml**.

```
info: # add this section
  build:
    artifact: @project.artifactId@
    name: @project.name@
    description: @project.description@
    version: @project.version@
```

Note we're defining token delimited value-placeholders for each property. In order to have these properties replaced, we'll need to add some further instructions to the *build.gradle* file.

→ if STS reports a problem with the application.yml due to @ character, the problem can safely be ignored.

2. Add the following directly underneath the *gitProperties{}* block within **cloud-native-spring/build.gradle**

```
import org.apache.tools.ant.filters.*

processResources {
    filter ReplaceTokens, tokens: [
        "application.name": project.property("application.name"),
        "application.description": project.property("application.description"),
        "application.version": project.property("version")
    ]
}
```

3. Build and run the *cloud-native-spring* application:

```
gradle clean bootRun
```

4. Browse to http://localhost:8080/info. Build information is now included.

```json
{
    "build": {
        "name": "Cloud Native Spring (Back-end)",
        "description": "Simple Spring Boot application employing an in-memory
relational data-store and which exposes a set of REST APIs",
        "version": "1.0-SNAPSHOT"
    },
    "git": {
        "commit": {
            "time": "2017-09-07T13:52+0000",
            "id": "3393f74"
        },
        "branch": "master"
    }
}
```

5. Stop the cloud-native-spring application.

   What Just Happened?

   We have mapped Gradle properties into the /info endpoint.

   Read more about exposing data in the /info endpoint here

Health Indicators

Spring Boot provides an endpoint http://localhost:8080/health that exposes various health indicators that describe the health of the given application.

Normally, when Spring Security is not enabled, the /health endpoint will only expose an UP or DOWN value.

```json
{
  "status": "UP"
}
```

1. Run the cloud-native-spring application:

   ```
   gradle bootRun
   ```

2. Browse to http://localhost:8080/health. Out of the box is a *DiskSpaceHealthIndicator* that monitors health in terms of available disk space. Would your Ops team like to know if the app is close to running out of disk space? DiskSpaceHealthIndicator can be customized via *DiskSpaceHealthIndicatorProperties*. For instance, setting a different threshold for when to report the status as DOWN.

```json
{
    "status": "UP",
    "diskSpace": {
        "status": "UP",
        "free": 42345678945,
        "threshold": 12345678
    }
}
```

3. Stop the cloud-native-spring application.

4. Create the class *io.pivotal.FlappingHealthIndicator* (/cloud-native-spring/src/main/java/io/pivotal/FlappingHealthIndicator.java) and into it paste the following code:

```java
package io.pivotal;

import java.util.Random;

import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class FlappingHealthIndicator implements HealthIndicator {

    private Random random = new Random(System.currentTimeMillis());

    @Override
    public Health health() {
        int result = random.nextInt(100);
        if (result < 50) {
            return Health.down().withDetail("flapper",
"failure").withDetail("random", result).build();
        } else {
            return Health.up().withDetail("flapper", "ok").withDetail("random",
result).build();
        }
    }
}
```

This demo health indicator will randomize the health check.

5. Build and run the *cloud-native-spring* application:

```
$ gradle clean bootRun
```

6. Browse to http://localhost:8080/health and verify that the output is similar to the following (and

28

changes randomly!).

```
{
  "status": "UP",
  "flapping": {
      "status": "UP",
      "flapper": "ok",
      "random": 42
  },
  "diskSpace": {
      "status": "UP",
      "free": 42345678945,
      "threshold": 12345678
  }
}
```

Metrics

Spring Boot provides an endpoint http://localhost:8080/metrics that exposes several automatically collected metrics for your application. It also allows for the creation of custom metrics.

1. Browse to http://localhost:8080/metrics. Review the metrics exposed.

```
{
"mem": 418830,
"mem.free": 239376,
"processors": 8,
"instance.uptime": 59563,
"uptime": 69462,
"systemload.average": 1.5703125,
"heap.committed": 341504,
"heap.init": 262144,
"heap.used": 102127,
"heap": 3728384,
"nonheap.committed": 79696,
"nonheap.init": 2496,
"nonheap.used": 77326,
"nonheap": 0,
"threads.peak": 14,
"threads.daemon": 11,
"threads.totalStarted": 17,
"threads": 13,
"classes": 9825,
"classes.loaded": 9825,
"classes.unloaded": 0,
"gc.ps_scavenge.count": 9,
"gc.ps_scavenge.time": 80,
"gc.ps_marksweep.count": 2,
"gc.ps_marksweep.time": 157,
"httpsessions.max": -1,
"httpsessions.active": 0,
"gauge.response.metrics": 75,
"gauge.response.star-star.favicon.ico": 9,
"counter.status.200.star-star.favicon.ico": 1,
"counter.status.200.metrics": 1
}
```

2. Stop the cloud-native-spring application.

Deploy *cloud-native-spring* to Pivotal Cloud Foundry

1. When running a Spring Boot application on Pivotal Cloud Foundry with the actuator endpoints enabled, you can visualize actuator management information on the Applications Manager app dashboard. To enable this there are a few properties we need to add. Add the following to **/cloud-native-spring/src/main/resources/application.yml**:

```
management:
  security:
    enabled: false
  info:
    git:
      mode: full
  cloudfoundry:
    enabled: true
    skip-ssl-validation: true
```

2. Let's review **/cloud-native-spring/build.gradle**. Note these lines:

```
jar {
    excludes = ['**/application.yml']
}

task execJar (type: Jar, dependsOn: jar) {
    classifier = 'exec'
    from sourceSets.main.output
}

bootRepackage  {
    withJarTask = tasks['execJar']
}
```

→ Note the *bootRepackage* plugin repackages the original artifact and creates a separate classified artifact. We wind up with 2 .jar files.

3. Push application into Cloud Foundry

```
gradle bootRepackage
cf push
```

4. Find the URL created for your app in the health status report. Browse to your app. Also view your application details in the Apps Manager UI:

5. From this UI you can also dynamically change logging levels:

Congratulations! You've just learned how to add health and metrics to any Spring Boot application.

Adding Spring Cloud Config to Boot Application

In this lab we'll utilize Spring Boot and Spring Cloud to configure our application from a configuration dynamically retrieved from a Git repository. We'll then deploy it to Pivotal Cloud Foundry and auto-provision an instance of a configuration server using Pivotal Spring Cloud Services.

Update *Hello* REST service

These features are added by adding *spring-cloud-services-starter-config-client* to the classpath.

1. Delete your exisiting Gradle build file, found here: **/cloud-native-spring/build.gradle**. We're going to make a few changes. Create a new **/cloud-native-spring/build.gradle** then cut-and-paste the content below into it and save.

   Adding a dependency management plugin and other miscellaneous configuration.

```
buildscript {
    ext {
        springBootVersion = '1.5.7.RELEASE'
        springDependencyManagementVersion = '1.0.3.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("gradle.plugin.com.gorylenko.gradle-git-properties:gradle-git-
properties:1.4.17")
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'maven-publish'
apply plugin: 'org.springframework.boot'
apply plugin: 'com.gorylenko.gradle-git-properties'

gitProperties {
    dateFormat = "yyyy-MM-dd'T'HH:mmZ"
    dateFormatTimeZone = "UTC"
    gitRepositoryRoot = new File("${project.rootDir}/../../..")
}

import org.apache.tools.ant.filters.*
```

```
processResources {
    filter ReplaceTokens, tokens: [
        "application.name": project.property("application.name"),
        "application.description": project.property("application.description"),
        "application.version": project.property("version")
    ]
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Dalston.SR4"
        mavenBom "io.pivotal.spring.cloud:spring-cloud-services-
dependencies:1.5.0.RELEASE"
    }
}

dependencies {
    compileOnly('org.projectlombok:lombok:1.16.18')
    compile('org.springframework.boot:spring-boot-starter-actuator')
    compile('org.springframework.boot:spring-boot-starter-data-jpa')
    compile('org.springframework.boot:spring-boot-starter-data-rest')
    compile('org.springframework.boot:spring-boot-starter-hateoas')
    compile('org.springframework.data:spring-data-rest-hal-browser')
    compile('org.springframework.boot:spring-boot-starter-web')
    compile('io.pivotal.spring.cloud:spring-cloud-services-starter-config-client')
    runtime('com.h2database:h2')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

repositories {
    maven {
        url "https://repo.spring.io/plugins-release"
    }
    mavenCentral()
}

jar {
    excludes = ['**/application.yml']
}

task execJar (type: Jar, dependsOn: jar) {
    classifier = 'exec'
    from sourceSets.main.output
}

bootRepackage  {
    withJarTask = tasks['execJar']
}

publishing {
```

```
    publications {
        maven(MavenPublication) {
            groupId 'io.pivotal'
            from components.java
        }
    }
    repositories {
        mavenLocal()
    }
}
```

2. Add an *@Value* annotation, private field, and update the existing *@RequestMapping* annotated method to employ it in *io.pivotal.controller.GreetingController* (/cloud-native-spring/src/main/java/io/pivotal/controller/GreetingController.java):

```
@Value("${greeting:Hola}")
private String greeting;

@RequestMapping("/hello")
public String hello() {
    return String.join(" ", greeting, "World!");
}
```

3. Add a @RefreshScope annotation to the top of the *GreetingController* class declaration

```
@RefreshScope
@RestController
public class GreetingController {
```

Completed:

```
package io.pivotal.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.scope.refresh.RefreshScope;

@RefreshScope
@RestController
public class GreetingController {

    @Value("${greeting:Hola}")
    private String greeting;

    @RequestMapping("/hello")
    public String hello() {
        return String.join(" ", greeting, "World!");
    }

}
```

4. When we introduced the Spring Cloud Services Starter Config Client dependency Spring Security will also be included (Config servers will be protected by OAuth2). However, this will also enable basic authentication to all our service endpoints. Add the following configuration to **/cloud-native-spring/src/main/resources/application.yml**:

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

5. We'll also want to give our Spring Boot App a name so that it can lookup application-specific configuration from the config server later. Add the following configuration to **/cloud-native-spring/src/main/resources/bootstrap.yml**. (You'll need to create this file.)

```
spring:
  application:
    name: cloud-native-spring
```

Run the *cloud-native-spring* Application and verify dynamic config is working

1. Run the application

```
gradle clean bootRun
```

2. Browse to http://localhost:8080/hello and verify you now see your new greeting.

3. Stop the *cloud-native-spring* application

Create Spring Cloud Config Server instance

1. Now that our application is ready to read its config from a Cloud Config server, we need to deploy one! This can be done through Cloud Foundry using the services Marketplace. Browse to the Marketplace in Pivotal Cloud Foundry Apps Manager, navigate to the Space you have been using to push your app, and select Config Server:
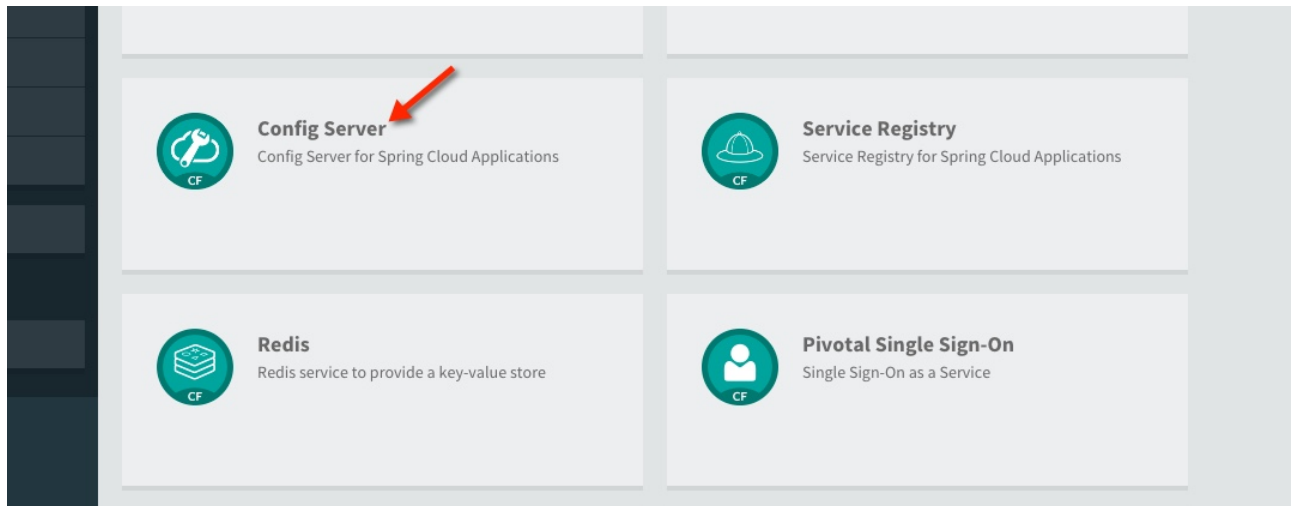
2. In the resulting details page, select the *standard*, single tenant plan. Name the instance **config-server**, select the Space that you've been using to push all your applications. At this time you don't need to select an application to bind to the service:
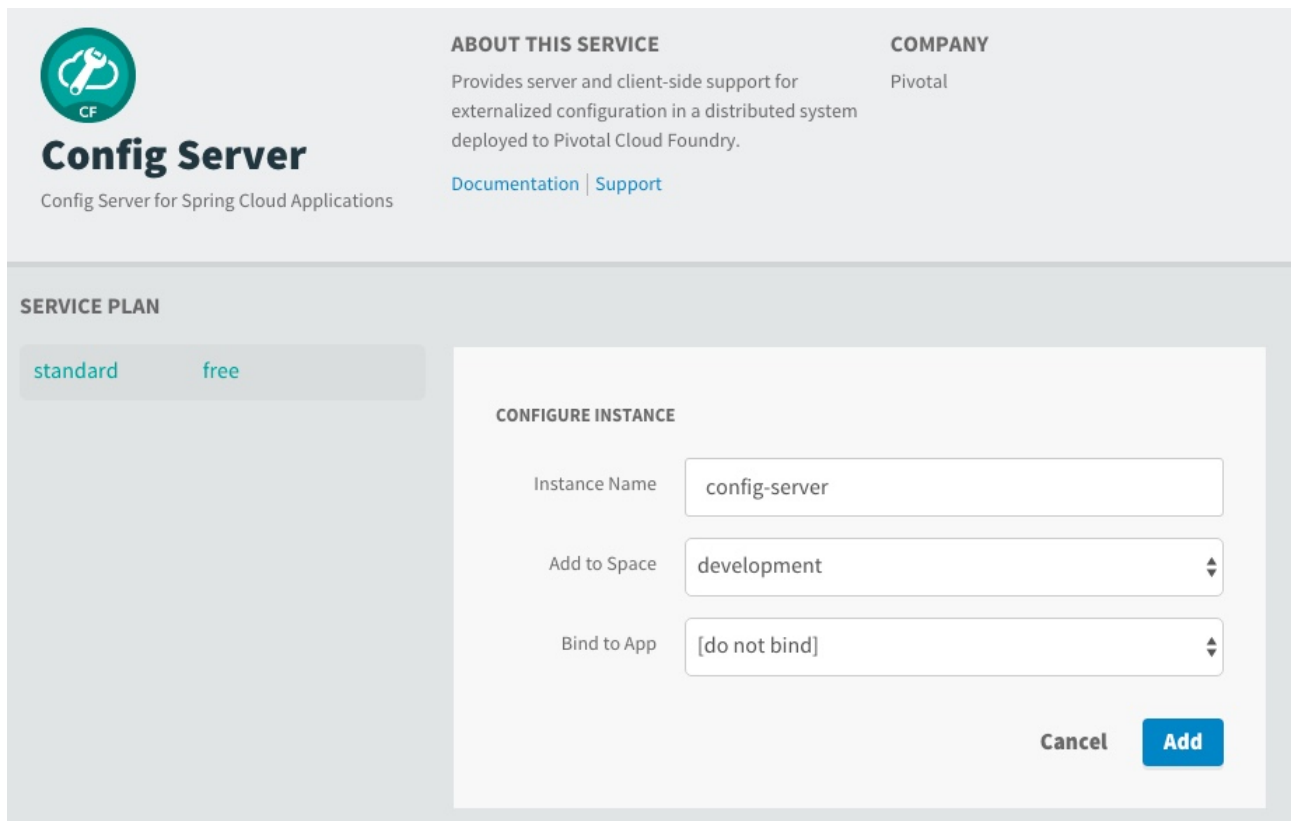
3. After we create the service instance you'll be redirected to your *Space* landing page that lists your apps and services. The config server is deployed on-demand and will take a few moments to deploy. Once the messsage *The Service Instance is Initializing* disappears click on the service you provisioned. Select the Manage link towards the top of the resulting screen to view the instance id and a JSON document with a single element, count, which validates that the instance provisioned correctly:
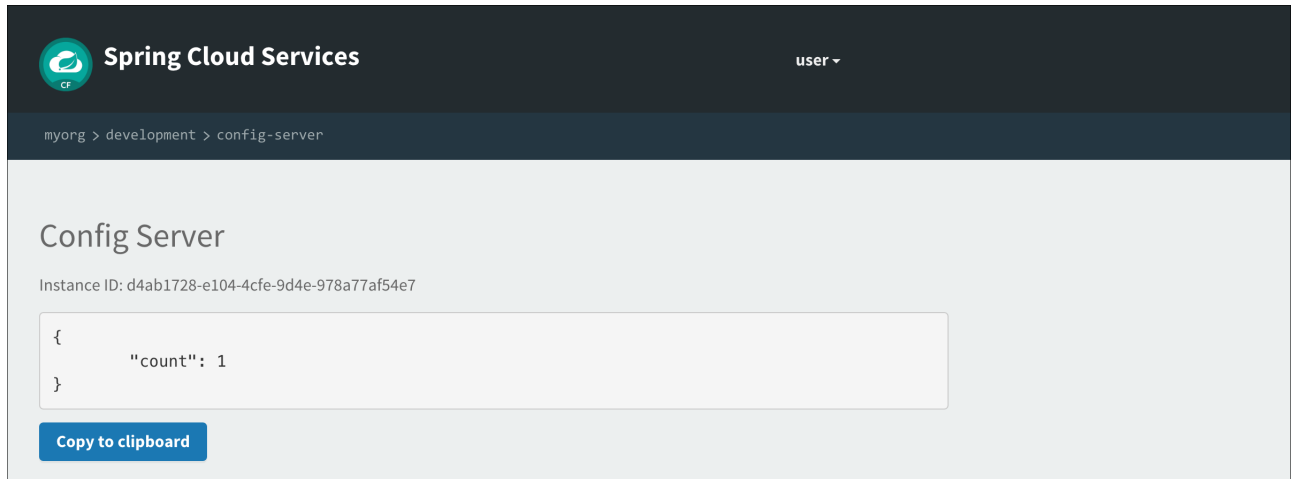
4. We now need to update the service instance with our GIT repository information. Using the Cloud Foundry CLI execute the following update service command:

```
cf update-service config-server -c '{"git": { "uri":
"https://github.com/pacphi/config-repo" } }'
```

5. Refresh you Config Server management page and you will see the following message. Wait until the screen refreshes and the service is reintialized:
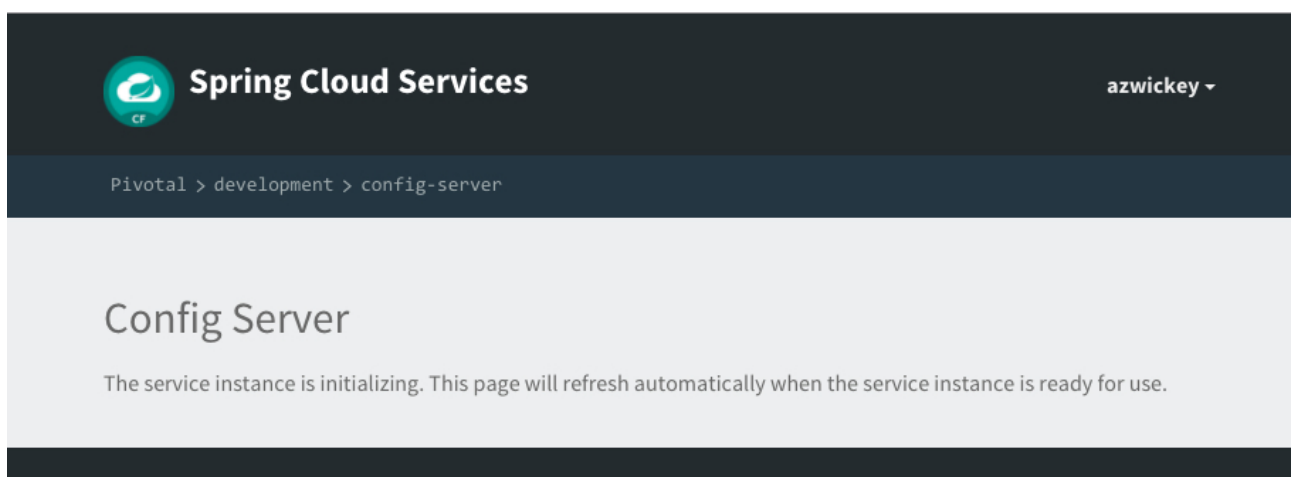
6. We will now bind our application to our config-server within our Cloud Foundry deployment manifest. Add these entries to the bottom of **/cloud-native-spring/manifest.yml**

```
    services:
    - config-server
```

Complete:

```
---
applications:
- name: cloud-native-spring
  host: cloud-native-spring-${random-word}
  memory: 1024M
  instances: 1
  path: ./target/cloud-native-spring-1.0-SNAPSHOT-exec.jar
  buildpack: java_buildpack
  timeout: 180
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
  services:
  - config-server
```

Deploy and test application

1. Build the application

```
gradle clean bootRepackage
```

2. Push application into Cloud Foundry

```
cf push
```

3. Test your application by navigating to the /hello endpoint of the application. You should now see a greeting that is read from the Cloud Config Server!

   Ohai World!

   What just happened??

   → A Spring component within the Spring Cloud Starter Config Client module called a *service connector* automatically detected that there was a Cloud Config service bound into the application. The service connector configured the application automatically to connect to the Cloud Config Server and downloaded the configuration and wired it into the application

4. If you navigate to the Git repo we specified for our configuration, https://github.com/pacphi/config-repo, you'll see a file named *cloud-native-spring.yml*. This filen ame is the same as our *spring.application.name* value for our Boot application. The configuration is read from this file, in our case the following property:

```
greeting: Ohai
```

5. Next we'll learn how to register our service with a Service Registry and load balance requests using Spring Cloud components.

Adding Service Registration and Discovery with Spring Cloud

In this lab we'll utilize Spring Boot and Spring Cloud to configure our application register itself with a Service Registry. To do this we'll also need to provision an instance of a Eureka service registry using Pivotal Cloud Foundry Spring Cloud Services. We'll also add a simple client application that looks up our application from the service registry and makes requests to our Cities service.

Update *Cloud-Native-Spring* Boot Application to Register with Eureka

1. These features are added by adding *spring-cloud-services-starter-service-registry* to the classpath. Open your Gradle build file, found here: **/cloud-native-spring/build.gradle**. Add the following spring cloud services dependency:

    ```
    dependencies {
        // add this dependency
        compile('io.pivotal.spring.cloud:spring-cloud-services-starter-service-
    registry')
    }
    ```

2. Thanks to Spring Cloud instructing your application to register with Eureka is as simple as adding a single annotation to your app! Add an *@EnableDiscoveryClient* annotation to the class *io.pivotal.CloudNativeSpringApplication* (/cloud-native-spring/src/main/java/io/pivotal/CloudNativeApplication.java):

    ```
    @SpringBootApplication
    @EnableDiscoveryClient
    public class CloudNativeSpringApplication {
    ```

Create Spring Cloud Service Registry instance and deploy application

1. Now that our application is ready to registr with an Eureka instance, we need to deploy one! This can be done through Cloud Foundry using the services Marketplace. Previously we did this through the Marketplace UI. This time around we will use the Cloud Foundry CLI:

    ```
    $ cf create-service p-service-registry standard service-registry
    ```

2. After you create the service registry instance navigate to your Cloud Foundry space in the Apps Manager UI and refresh the page. You should now see the newly create Service Registry intance. Select the Manage link to view the registry dashboard. Note that there are not any registered

applications at the moment:
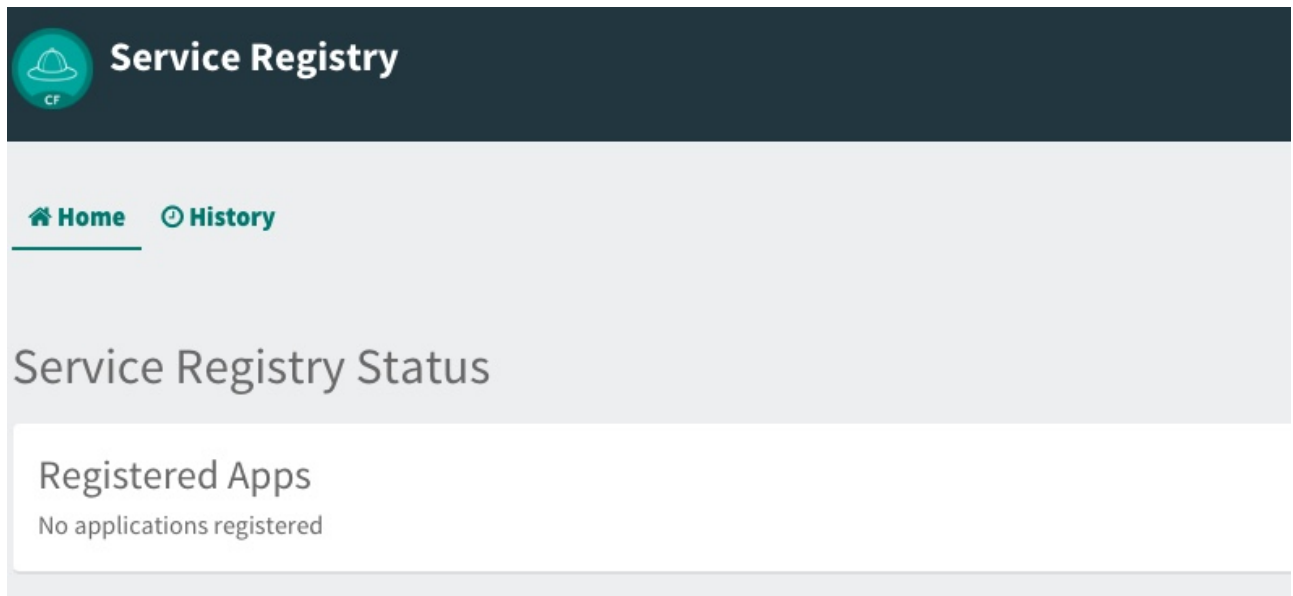
3. We will now bind our application to our service-registry within our Cloud Foundry deployment manifest. Add an additional reference to the service at the bottom of **/cloud-native-spring/manifest.yml** in the services list:

```
services:
- config-server
- service-registry
```

Deploy and test application

1. Build the application

```
gradle bootRepackage
```

2. For the 2nd half of this lab we'll need to have this artifact in our local repository, so install it with the following command:

```
gradle publishToMavenLocal
```

3. Push application into Cloud Foundry

```
cf push
```

4. If we now test our application URLs we will notice no significatnt changes. However, if we view the Service Registry dashboard (accessible from the *Manage* link in Apps Manager) you will see that a service named cloud-native-spring has registered:
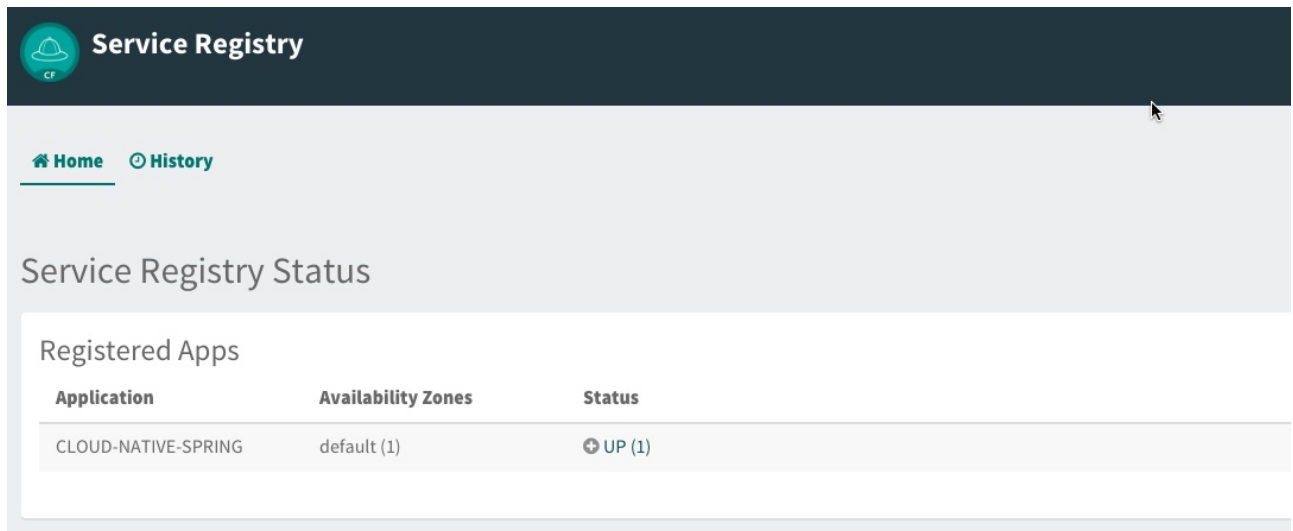
5. Next we'll create a simple UI application that will read from the Service Registry to discover the location of our cities REST service and connect.

Create another Spring Boot Project as a Client UI

As in Lab 1 we will start with a project that has most of what we need to get going.

1. Open a Terminal (e.g., *cmd* or *bash* shell)

2. Change the working directory to be *devops-workshop-boeing/labs/my_work/cloud-native-spring-ui*

   ```
   cd /devops-workshop-boeing/labs/my_work/cloud-native-spring-ui
   ```

3. Open this project in your editor/IDE of choice.

   STS Import Help

   Select *File > Import....* In the susequent dialog choose *Gradle > Existing Gradle Project* then click the *Next* button. In the *Import Gradle Project* dialog browse to the *cloud-native-spring* directory (e.g. *devops-workshop-boeing/labs/my_work/cloud-native-spring-ui*) then click the *Open* button, then click the *Finish* button.

4. As before, we need to add *spring-cloud-services-starter-service-registry* to the classpath. Add this to your POM:

   ```
   dependencies {
       // add this dependency
       compile('io.pivotal.spring.cloud:spring-cloud-services-starter-service-
   registry')
   }
   ```

   We'll also be using the Domain object from our main Boot application. Add that as a dependency too:

```
dependencies {
    // add this dependency
    compile('io.pivotal:cloud-native-spring:1.0-SNAPSHOT')
}
```

5. Since this UI is going to consume REST services it's an awesome opportunity to use Feign. Feign will handle **ALL** the work of invoking our services and marshalling/unmarshalling JSON into domain objects. We'll add a Feign Client interface into our app. Take note of how Feign references the downstream service; it's only the name of the service it will lookup from Eureka Service Registry. Create a new interface that resides in the same package as *CloudNativeSpringUiApplication*:

```
package io.pivotal;

import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.hateoas.Resources;
import io.pivotal.domain.City;


@FeignClient(name = "https://cloud-native-spring")
public interface CityClient {

    @GetMapping(value="/cities", consumes="application/hal+json")
    Resources<City> getCities();
}
```

We'll also need to add a few annotations to our Spring Boot application:

```
@SpringBootApplication
@EnableFeignClients
@EnableDiscoveryClient
public class CloudNativeSpringUiApplication {
```

6. Next we'll create a Vaadin UI for rendering our data. The point of this workshop isn't to go into detail on creating UIs; for now suffice to say that Vaadin is a great tool for quickly creating User Interfaces. Our UI will consume our Feign client we just created. Create the class *io.pivotal.AppUi* (/cloud-native-spring-ui/src/main/java/io/pivotal/AppUi.java) and into it paste the following code:

```java
package io.pivotal;

import com.vaadin.annotations.Theme;

import com.vaadin.server.VaadinRequest;
import com.vaadin.spring.annotation.SpringUI;
import com.vaadin.ui.Grid;
import com.vaadin.ui.UI;
import io.pivotal.domain.City;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.ArrayList;
import java.util.Collection;

@SpringUI
@Theme("valo")
public class AppUi extends UI {

    private final CityClient client;
    private final Grid<City> grid;

    @Autowired
    public AppUi(CityClient client) {
        this.client = client;
        this.grid = new Grid<>(City.class);
    }

    @Override
    protected void init(VaadinRequest request) {
        setContent(grid);
        grid.setWidth(100, Unit.PERCENTAGE);
        grid.setHeight(100, Unit.PERCENTAGE);
        Collection<City> collection = new ArrayList<>();
        client.getCities().forEach(collection::add);
        grid.setItems(collection);
    }
}
```

7. We'll also want to give our UI App a name so that it can register properly with Eureka and potentially use cloud config in the future. Add the following configuration to /**cloud-native-spring-ui/src/main/resources/bootstrap.yml**:

```yaml
spring:
  application:
    name: cloud-native-spring-ui
```

Deploy and test application

1. Build the application. We have to skip the tests otherwise we may fail because of having 2 spring boot apps on the classpath

   ```
   gradle bootRepackage -x test
   ```

   → Note that we're skipping tests here (because we now have a dependency on a running instance of *cloud-native-spring*).

2. Create an application manifest in the root folder /cloud-native-spring-ui

   $ touch manifest.yml

3. Add application metadata

   ```
   ---
   applications:
   - name: cloud-native-spring-ui
     memory: 1024M
     random-route: true
     instances: 1
     path: ./build/libs/cloud-native-spring-ui-1.0-SNAPSHOT-exec.jar
     buildpack: java_buildpack
     timeout: 180 # to give time for the data to import
     env:
       JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
     services:
     - service-registry
   ```

4. Push application into Cloud Foundry

   ```
   cf push
   ```

5. Test your application by navigating to the root URL of the application, which will invoke Vaadin UI. You should now see a table listing the first set of rows returned from the cities microservice:

County	Id	Latitude	Longitude	Name	Postal Code	State Code
SUFFOLK	0	+40.922326	-072.637078	HOLTSVILLE	00501	NY
SUFFOLK	0	+40.922326	-072.637078	HOLTSVILLE	00544	NY
ADJUNTAS	0	+18.165273	-066.722583	ADJUNTAS	00601	PR
AGUADA	0	+18.393103	-067.180953	AGUADA	00602	PR
AGUADILLA	0	+18.455913	-067.145780	AGUADILLA	00603	PR
AGUADILLA	0	+18.493520	-067.135883	AGUADILLA	00604	PR
AGUADILLA	0	+18.465162	-067.141486	AGUADILLA	00605	PR
MARICAO	0	+18.172947	-066.944111	MARICAO	00606	PR
ANASCO	0	+18.288685	-067.139696	ANASCO	00610	PR
UTUADO	0	+18.279531	-066.802170	ANGELES	00611	PR
ARECIBO	0	+18.450674	-066.698262	ARECIBO	00612	PR
ARECIBO	0	+18.458093	-066.732732	ARECIBO	00613	PR
ARECIBO	0	+18.429675	-066.674506	ARECIBO	00614	PR
ARECIBO	0	+18.444792	-066.640678	BAJADERO	00616	PR
BARCELONETA	0	+18.447092	-066.544255	BARCELONETA	00617	PR
CABO ROJO	0	+17.998531	-067.187318	BOQUERON	00622	PR
CABO ROJO	0	+18.062201	-067.149541	CABO ROJO	00623	PR

6. From a commandline stop the cloud-native-spring microservice (the original City service, not the new UI)

```
cf stop cloud-native-spring
```

7. Refresh the UI app.

What happens?

Now you get a nasty error that is not very user friendly!

→ Next we'll learn how to make our UI Application more resilient in the case that our downstream services are unavailable.

Employing a circuit breaker

In this lab we'll utilize Spring Boot and Spring Cloud to make our UI Application more resilient. We'll leverage Spring Cloud Circuit Breaker to configure our application behavior when our downstream dependencies are not available. Finally, we'll use the circuit break dashboard to view metrics of the circuit breaker we implemented, which will be auto-provisioned within Cloud Foundry Pivotal Spring Cloud Services.

Define a Circuit Breaker within the *UI Application*

1. These features are added by adding *spring-cloud-services-starter-circuit-breaker* to the classpath. Open your Gradle build file, found here: **/cloud-native-spring-ui/build.gradle**. Add the following Spring Cloud Services dependency:

```
dependencies {
    // add this dependency
    compile('io.pivotal.spring.cloud:spring-cloud-services-starter-circuit-
breaker')
}
```

2. The first thing we need to add to our application is an *@EnableCircuitBreaker* annotation to the Spring Boot application. Add this annotation below the other ones on the *CloudNativeSpringUiApplication* declaration in the class *io.pivotal.CloudNativeSpringUiApplication* (/cloud-native-spring-ui/src/main/java/io/pivotal/CloudNativeSpringUiApplication.java):

```
@SpringBootApplication
@EnableFeignClients
@EnableDiscoveryClient
@EnableCircuitBreaker
public class CloudNativeSpringUiApplication {
```

3. When we introduced an *@FeignClient* into our application we were only required to provide an interface. We'll provide a dummy class that implements that interface for our fallback. We'll also reference that class as a fallback in our *@FeignClient* annotation. First, create this class in the *io.pivotal* package:

```
@Component
public class CityClientFallback implements CityClient {
    @Override
    public Resources<City> getCities() {
        //We'll just return an empty response
        return new Resources<City>(Collections.emptyList());
    }
}
```

4. Also modify the *@FeignClient* annotation to reference this class as the fallback in case of failure:

```
@FeignClient(name = "https://cloud-native-spring", fallback =
CityClientFallback.class)
public interface CityClient {
```

Create the Circuit Breaker Dashboard

1. When we modified our application to use a Hystrix Circuit Breaker our application automatically begins streaming out metrics about the health of our methods wrapped with a HystrixCommand. We can stream these events through a AMQP message bus into Turbine to view on a Circuit Breaker dashboard. This can be done through cloudfoundry using the services marketplace by executing the following command:

```
cf create-service p-circuit-breaker-dashboard standard circuit-breaker
```

2. If we view the Circuit Breaker Dashboard (accessible from the *Manage* link in Apps Manager) you will see that a dashboard has been deployed but is empty (You may get an *Initializing* message for a few seconds. This should eventually refresh to a dashboard):
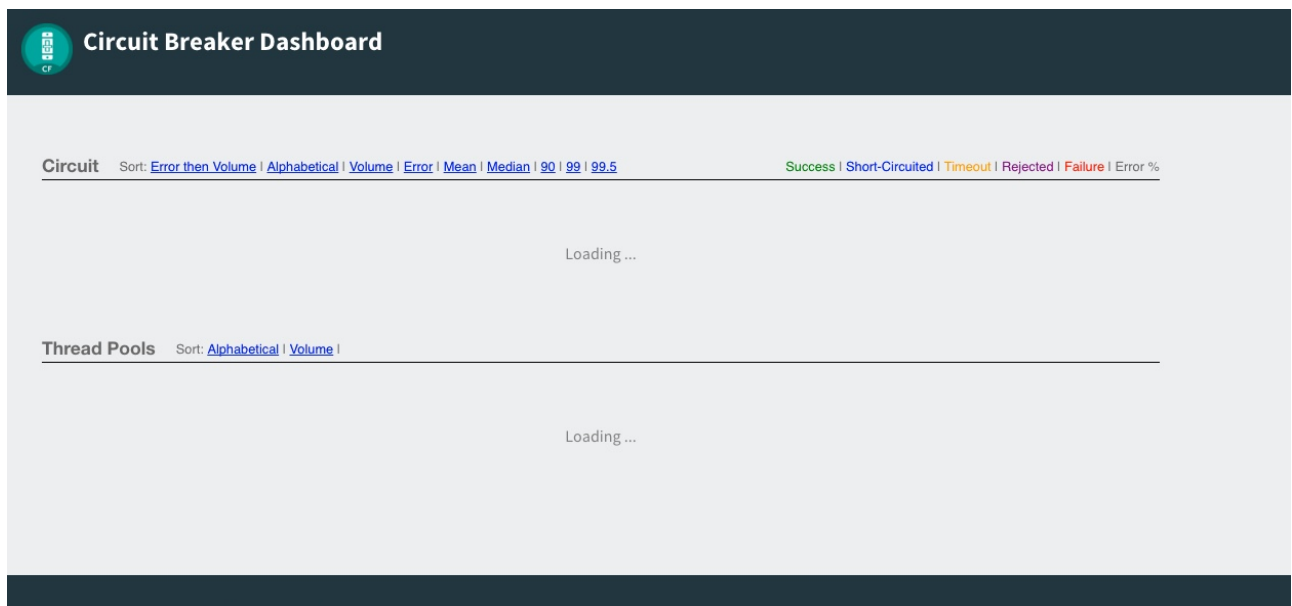
3. We will now bind our application to our *circuit-breaker-dashboard* within our Cloud Foundry deployment manifest. Add this additional reference to a service at the bottom of **/cloud-native-spring-ui/manifest.yml** in the services list:

```
services:
- service-registry
- circuit-breaker
```

Deploy and test application
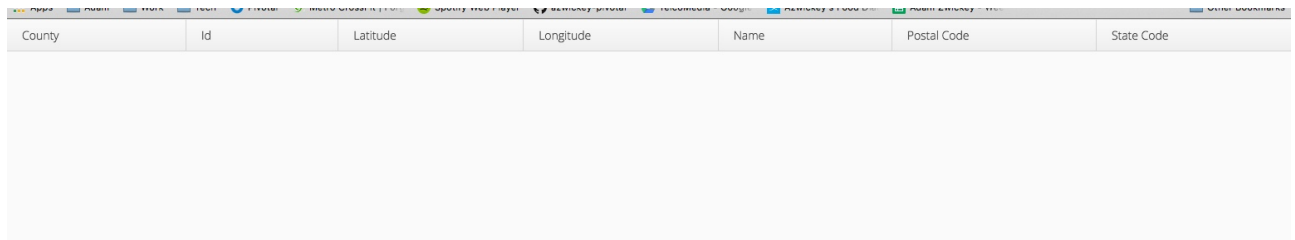
1. Build the application

   ```
   gradle bootRepackage -x test
   ```

2. Push application into Cloud Foundry

   ```
   cf push
   ```

3. Test your application by navigating to the root URL of the application. If the dependent cities REST service is still stopped, you should simply see a blank table. Remember that last time you received a nasty exception in the browser? Now your Circuit Breaker fallback method is automatically called and the fallback behavior is executed.

4. From a commandline start the *cloud-native-spring* microservice (the original City service, not the new UI)

   ```
   cf start cloud-native-spring
   ```

5. Refresh the UI app and you should once again see a table listing the first page of cities.

County	Id	Latitude	Longitude	Name	Postal Code	State Code
SUFFOLK	0	+40.922326	-072.637078	HOLTSVILLE	00501	NY
SUFFOLK	0	+40.922326	-072.637078	HOLTSVILLE	00544	NY
ADJUNTAS	0	+18.165273	-066.722583	ADJUNTAS	00601	PR
AGUADA	0	+18.393103	-067.180953	AGUADA	00602	PR
AGUADILLA	0	+18.455913	-067.145780	AGUADILLA	00603	PR
AGUADILLA	0	+18.493520	-067.135883	AGUADILLA	00604	PR
AGUADILLA	0	+18.465162	-067.141486	AGUADILLA	00605	PR
MARICAO	0	+18.172947	-066.944111	MARICAO	00606	PR
ANASCO	0	+18.288685	-067.139696	ANASCO	00610	PR
UTUADO	0	+18.279531	-066.802170	ANGELES	00611	PR
ARECIBO	0	+18.450674	-066.698262	ARECIBO	00612	PR
ARECIBO	0	+18.458093	-066.732732	ARECIBO	00613	PR
ARECIBO	0	+18.429675	-066.674506	ARECIBO	00614	PR
ARECIBO	0	+18.444792	-066.640678	BAJADERO	00616	PR
BARCELONETA	0	+18.447092	-066.544255	BARCELONETA	00617	PR
CABO ROJO	0	+17.998531	-067.187318	BOQUERON	00622	PR
CABO ROJO	0	+18.062201	-067.149541	CABO ROJO	00623	PR

6. Refresh your UI application a few times to force some traffic though the circuit breaker call path. After doing this you should now see the dashboard populated with metrics about the health of your Hystrix circuit breaker: