DevOps Workshop Lab Guide

Christopher Phillipson

Version 2.0, 2018-10-30

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

This workshop assumes participants will be interacting with PCF One. Depending on the client and environment, ask the instuctor for an alternate CF API endpoint and/or url for the Apps Manager UI.

Account set up

1. If you do not have an account yet, please ask the instructor for one.

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:

   ```
   $ cf api https://api.run.pcfone.io
   $ 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):

   ```
   https://apps.run.pcfone.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/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 -a
   ```

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/labs/my_work/cloud-native-spring*

   ```
   cd /devops-workshop/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/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/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 build
```

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):

```
2018-08-22 17:40:18.193  INFO 92704 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http)
with context path ''
2018-08-22 17:40:18.199  INFO 92704 --- [           main]
i.p.CloudNativeSpringUiApplication       : Started CloudNativeSpringUiApplication
in 7.014 seconds (JVM running for 7.814)
```

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/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
  buildpacks:
  - java_buildpack_offline
  stack: cflinuxfs3
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
```

The above manifest entries will work with Java Buildpack 4.x series and JDK 8. If you built the app with JDK 11 and want to deploy it you will need to make an additional entry in your manifest, just below JAVA_OPTS, add

```
---
    JBP_CONFIG_OPEN_JDK_JRE: '{ jre: { version: 11.+ } }'
```

3. Push application into Cloud Foundry

```
cf push
```

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

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

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. Along the way we'll take a brief look at Flyway which can help us manage updates to database schema and data.

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`

Use Flyway to manage schema

1. Edit *build.gradle* and add the following dependencies within the *dependencies {}* block

```
compile('org.flywaydb:flyway-core:5.2.0')
compile('com.zaxxer:HikariCP:3.2.0')
```

2. Create a new file named `V1_0__init_database.sql` underneath *devops-workshop/labs/my_work/cloud-native-spring/src/main/resources/db/migration,* add the following lines and save.

```
CREATE TABLE city (
    ID INTEGER PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(100) NOT NULL,
    COUNTY VARCHAR(100) NOT NULL,
    STATE_CODE VARCHAR(10) NOT NULL,
    POSTAL_CODE VARCHAR(10) NOT NULL,
    LATITUDE VARCHAR(15) NOT NULL,
    LONGITUDE VARCHAR(15) NOT NULL
);
```

Spring Boot comes with out-of-the-box integration support for Flyway. When we start the application it will execute a versioned SQL migration that will create a new table in the database.

3. Add the following lines to *devops-workshop/labs/my_work/cloud-native-spring/src/main/resources/application.yml*

```
spring:
  datasource:
    hikari:
      connection-timeout: 60000
      maximum-pool-size: 5
```

+ Hikari is a database connection pool implementation. We are limiting the number of database connections an individual application instance may consume.

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.

```
http :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, modified a build file, added some configuration and SQL migration scripts, 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. Copy the import.sql file found in **devops-workshop/labs/** to *devops-workshop/labs/my_work/cloud-native-spring/src/main/resources/db/migration*. Rename the file to be `V1_1__seed_data.sql`. (This is a small subset of a larger dataset containing all of the postal

codes in the United States and its territories).

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).

```
http :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.

```
http :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

```
http :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=01229

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

http://localhost:8080/cities/search/nameContains?q=West&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 build
```

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
  buildpacks:
  - java_buildpack_offline
  stack: cflinuxfs3
  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 Jul 28 23:29:21 UTC 2018
stack: cflinuxfs2
buildpack: java_buildpack_offline

     state     since                  cpu       memory        disk
details
#0   running   2018-07-28 04:30:22 PM  163.7%    395.7M of 1G   159M of 1G
```

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

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

   {random-word} might be something like loquacious-eagle and {domain} might be cfapps.io if you
   happened to target Pivotal Web Services

5. Let's stop the application momentarily as we prepare to swap out the database provider.

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

Binding to a MySQL database in Cloud Foundry

1. Let's create a MySQL database instance. Hopefully, you will have
   [p.mysql](https://network.pivotal.io/products/pivotal-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
db-small        This plan provides a small dedicated MySQL instance.    free
```

2. Let's create an instance of `p.mysql` with `db-small` plan, e.g.

```
cf create-service p.mysql db-small mysql-database
```

Expected output:

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

So long as the name of the service contains `mysql` the mysql-connector JDBC driver will automatically be added as a runtime dependency.

However, we're going to explicitly define a runtime dependency on the MySQL JDBC driver. Open `build.gradle` for editing and add the following to the `dependencies` section

```
runtime('mysql:mysql-connector-java:8.0.13')
```

And, of course we must rebuild and repackage the application to have the application recognize the new dependency at runtime

```
gradle build
```

3. 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

4. Now let's push the updated application

```
cf push cloud-native-spring
```

5. 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
...
```

6. 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
  buildpacks:
  - java_buildpack_offline
  timeout: 180 # to give time for the data to import
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
  services:
    - mysql-database
```

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

```
cf push
```

8. Let's have a look at how we can interact with the database

Visit Pivotal MySQL*Web then follow these instructions for building the application.

+

```
cd ..
git clone https://github.com/pivotal-cf/PivotalMySQLWeb.git
cd PivotalMySQLWeb
./mvnw -DskipTests=true package
```

+ Then to prepare the application for deployment we'll create a manifest. Open an editor, create and save a file named `manifest.yml` with these contents:

+

```
applications:
- name: pivotal-mysqlweb
  memory: 1024M
  instances: 1
  random-route: true
  path: ./target/PivotalMySQLWeb-1.0.0-SNAPSHOT.jar
  services:
    - mysql-database
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
```

+ Of course, you'll want to deploy the application

+

```
cf push
```

+ And once deployed, you can visit the appliation URL and log in with the default credentials `admin/cfmysqlweb`

+ Take a few moments to explore the features and see that the administrative and diagnostic functions of Pivotal MySQL*Web provide a rather simple way to interact with and keep your database instance up-to-date via an Internet browser.

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 does not expose these management endpoints (which is a good thing!). Though you wouldn't want to expose all of them 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:
     endpoints:
       web:
         exposure:
           include: "*"
   ```

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 :8080/actuator/health

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

   http :8080/actuator/beans

   → Dumps all of the beans in the Spring context.

   http :8080/actuator/autoconfig

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

http :8080/actuator/configprops

→ Displays a collated list of all @ConfigurationProperties.

http :8080/actuator/env

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

http :8080/actuator/mappings

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

http :8080/actuator/threaddump

→ Performs a thread dump.

http :8080/actuator/httptrace

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

http :8080/actuator/flyway

→ Shows any Flyway database migrations that have been applied.

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

Include Version Control Info

Spring Boot provides an endpoint (http://localhost:8080/actuator/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:2.0.0-beta1")
```

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"
    dotGitDirectory = "${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 **/actuator/info** endpoint.

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

```
gradle clean bootRun
```

3. Let's use httpie to verify that Git commit information is now included

```
http :8080/actuator/info
```

```
{
    "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 **/actuator/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. Again we'll use httpie to verify that the Build information is now included

```
http :8080/actuator/info
```

```
{
    "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 /actuator/info endpoint.

   Read more about exposing data in the /actuator/info endpoint [here](#)

Health Indicators

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

Normally, the /actuator/health endpoint will only expose an UP or DOWN value.

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

+ We want to expose more detail about the health and well-being of the application, so we're going to need a bit more configuration to *cloud-native-spring/src/main/resources/application.yml*, underneath the *management* prefix, add

+

```
endpoint:
  health:
    show-details: always
```

1. Run the cloud-native-spring application:

```
gradle bootRun
```

2. Use httpie to verify the output of the health endpoint

```
http :8080/actuator/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.

```
{
    "status": "UP",
    "details": {
        "diskSpace": {
            "status": "UP",
            "details": {
                "total": 499963170816,
                "free": 375287070720,
                "threshold": 10485760
            }
        },
        "db": {
            "status": "UP",
            "details": {
                "database": "H2",
                "hello": 1
            }
        }
    }
}
```

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:

```
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/actuator/health and verify that the output is similar to the following (and changes randomly!).

```
{
    "status": "UP",
    "details": {
        "flapping": {
            "status": "UP",
            "details": {
                "flapper": "ok",
                "random": 63
            }
        },
        "diskSpace": {
            "status": "UP",
            "details": {
                "total": 499963170816,
                "free": 375287070720,
                "threshold": 10485760
            }
        },
        "db": {
            "status": "UP",
            "details": {
                "database": "H2",
                "hello": 1
            }
        }
    }
}
```

Metrics

Spring Boot provides an endpoint http://localhost:8080/actuator/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/actuator/metrics. Review the metrics exposed.

```
{
    "names": [
        "jvm.memory.max",
        "http.server.requests",
        "jdbc.connections.active",
        "process.files.max",
        "jvm.gc.memory.promoted",
        "tomcat.cache.hit",
        "system.load.average.1m",
        "tomcat.cache.access",
        "jvm.memory.used",
        "jvm.gc.max.data.size",
```

```
            "jdbc.connections.max",
            "jdbc.connections.min",
            "jvm.gc.pause",
            "jvm.memory.committed",
            "system.cpu.count",
            "logback.events",
            "tomcat.global.sent",
            "jvm.buffer.memory.used",
            "tomcat.sessions.created",
            "jvm.threads.daemon",
            "system.cpu.usage",
            "jvm.gc.memory.allocated",
            "tomcat.global.request.max",
            "hikaricp.connections.idle",
            "hikaricp.connections.pending",
            "tomcat.global.request",
            "tomcat.sessions.expired",
            "hikaricp.connections",
            "jvm.threads.live",
            "jvm.threads.peak",
            "tomcat.global.received",
            "hikaricp.connections.active",
            "hikaricp.connections.creation",
            "process.uptime",
            "tomcat.sessions.rejected",
            "process.cpu.usage",
            "tomcat.threads.config.max",
            "jvm.classes.loaded",
            "hikaricp.connections.max",
            "hikaricp.connections.min",
            "jvm.classes.unloaded",
            "tomcat.global.error",
            "tomcat.sessions.active.current",
            "tomcat.sessions.alive.max",
            "jvm.gc.live.data.size",
            "tomcat.servlet.request.max",
            "hikaricp.connections.usage",
            "tomcat.threads.current",
            "tomcat.servlet.request",
            "hikaricp.connections.timeout",
            "process.files.open",
            "jvm.buffer.count",
            "jvm.buffer.total.capacity",
            "tomcat.sessions.active.max",
            "hikaricp.connections.acquire",
            "tomcat.threads.busy",
            "process.start.time",
            "tomcat.servlet.error"
    ]
}
```

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**:

```
---
spring:
  profiles: cloud

management:
  cloudfoundry:
    enabled: true
    skip-ssl-validation: true
```

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

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

bootJar {
    enabled = true
    classifier = 'exec'
}
```

→ Note the *bootJar* 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 build
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 = '2.1.1.RELEASE'
       }
       repositories {
           maven { url "https://plugins.gradle.org/m2/" }
           maven { url "https://repo.spring.io/snapshot" }
           maven { url "https://repo.spring.io/milestone" }
           mavenCentral()
       }
       dependencies {
           classpath("gradle.plugin.com.gorylenko.gradle-git-properties:gradle-git-
   properties:2.0.0-beta1")
           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'
   apply plugin: 'io.spring.dependency-management'

   gitProperties {
       dateFormat = "yyyy-MM-dd'T'HH:mmZ"
       dateFormatTimeZone = "UTC"
       dotGitDirectory = "${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:Greenwich.M3"
        mavenBom "io.pivotal.spring.cloud:spring-cloud-services-
dependencies:2.0.3.RELEASE"
    }
}

dependencies {
    compileOnly('org.projectlombok:lombok:1.18.4')
    compile('javax.xml.bind:jaxb-api:2.4.0-b180725.0427')
    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')
    compile('org.flywaydb:flyway-core:5.2.0')
    compile('com.zaxxer:HikariCP:3.2.0')
    runtime('com.h2database:h2')
    runtime('mysql:mysql-connector-java:8.0.13')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

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

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

bootJar {
    enabled = true
```

```
        classifier = 'exec'
    }

    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.config.annotation.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 to conditionally disable security (to ease local workstation deployment) **/cloud-native-spring/src/main/java/io/pivotal/CloudNativeSpringApplication.java** right underneath the `public static void main` method implementation

```
    @Configuration
    static class ApplicationSecurityOverride extends WebSecurityConfigurerAdapter {

        @Override
        public void configure(WebSecurity web) throws Exception {
            web
                .ignoring()
                .antMatchers("/**");
        }
    }
```

Examine this Spring Boot reference for further details.

5. Another thing we'll have to allow is for bean definitions to be overridden. Add this line indented exactly two-spaces underneath `spring:` in **/cloud-native-spring/src/main/resources/application.yml**

```
  main:
    allow-bean-definition-overriding: true
```

6. 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
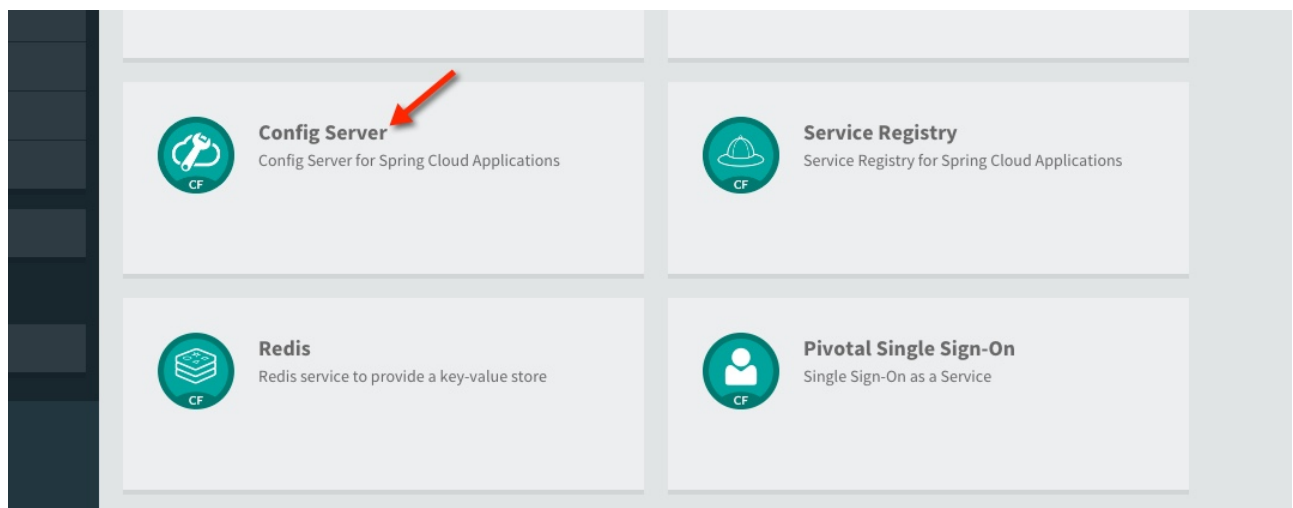
1. Run the application

```
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 *trial*, 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.

   Create a file named `config-server.json` and update its contents to be

```
{
  "git": {
    "uri": "https://github.com/pacphi/config-repo"
  }
}
```

Note: If you choose to replace the value of `"uri"` above with another Git repository that you have commit privileges to, you should make a copy of the `cloud-native-spring.yml` file. Then, as you update configuration in that file, you can test a POST request to the `cloud-native-spring` application's `/refresh` end-point to see the new configuration take effect without restarting the application!

Using the Cloud Foundry CLI execute the following update service command:

```
cf update-service config-server -c config-server.json
```

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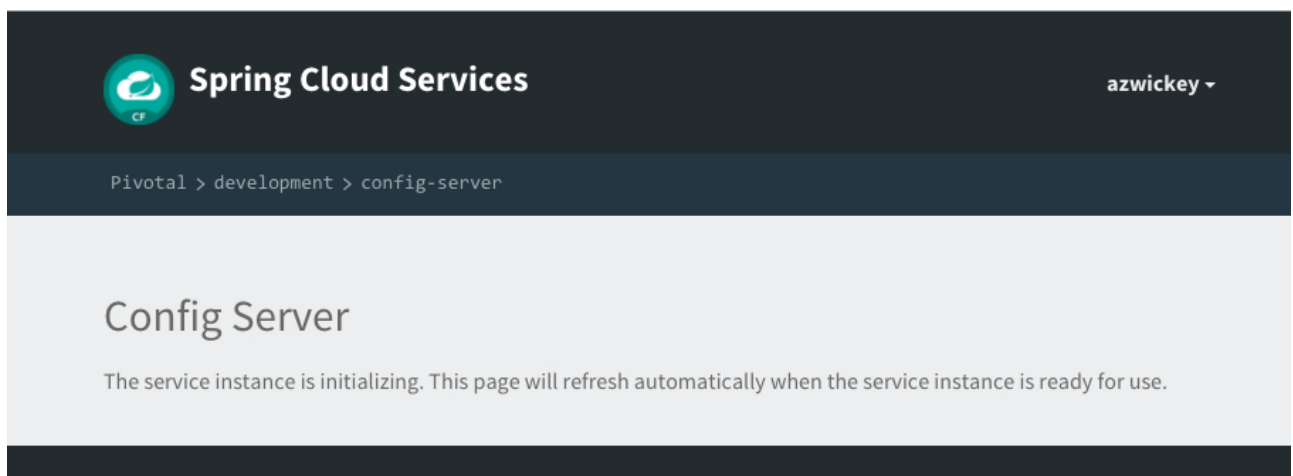
6. We will now bind our application to our config-server within our Cloud Foundry deployment manifest. Add these entries to the bottom of **/cloud-native-spring/manifest.yml**

```
  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
  buildpacks:
  - java_buildpack_offline
  stack: cflinuxfs3
  timeout: 180
  env:
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
  services:
  - config-server
```

Deploy and test application

1. Build the application

   ```
   gradle clean build
   ```

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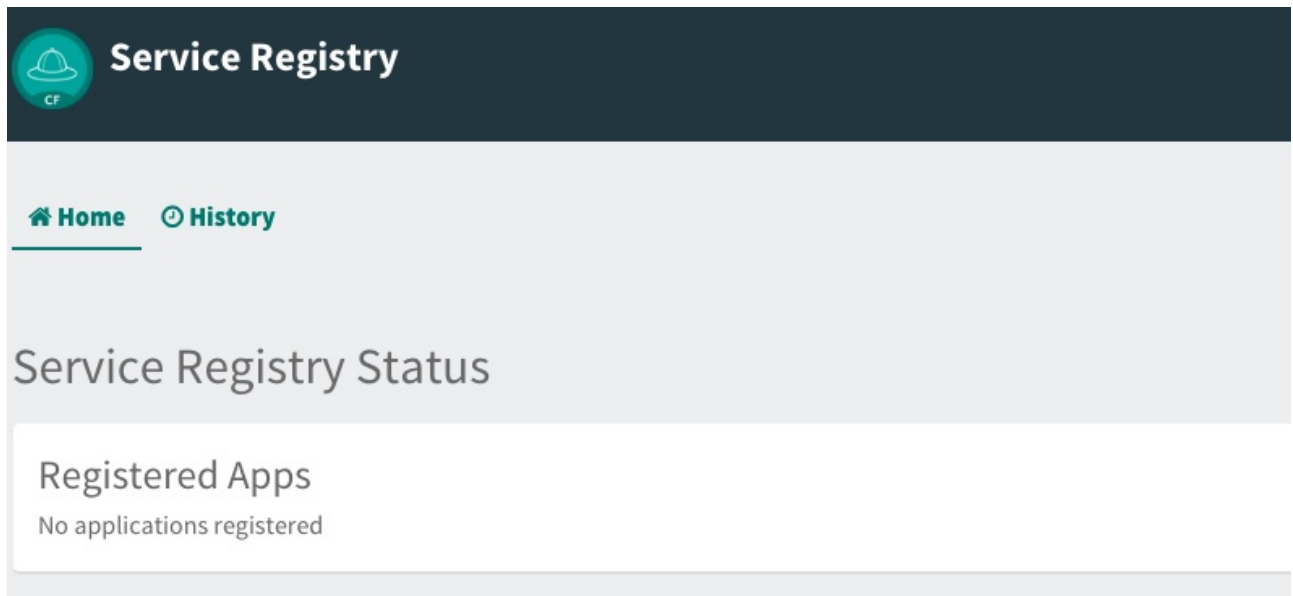
3. We will now bind our application to our service-registry within our Cloud Foundry deployment manifest. Add an additional reference to the service at the bottom of /**cloud-native-spring/manifest.yml** in the services list:

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

Deploy and test application

1. Build the application

```
gradle build
```

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:
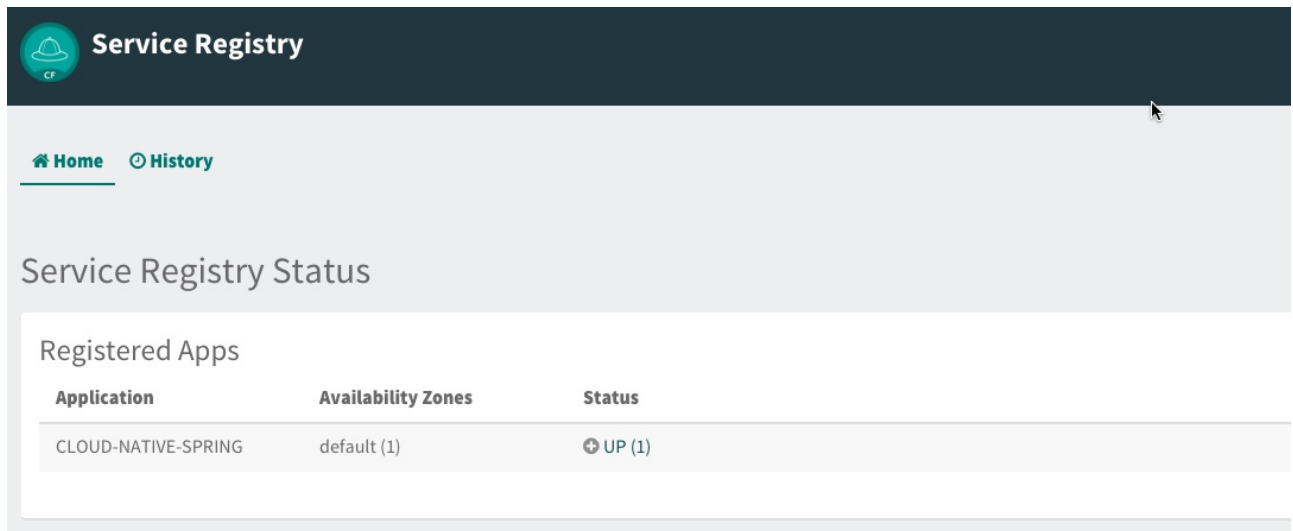
5. Next we'll create a simple UI application that will read from the Service Registry to discover the location of our cities REST service and connect.

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/labs/my_work/cloud-native-spring-ui*

```
cd /devops-workshop/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/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* and some collaborating dependencies to the classpath. Add this to your *build.gradle*:

```
dependencies {
    // add these dependencies
    compile('io.pivotal.spring.cloud:spring-cloud-services-starter-service-
registry')
    compile('org.springframework.cloud:spring-cloud-starter-openfeign')
    compile('org.springframework.cloud:spring-cloud-starter-netflix-ribbon')
    compile('org.hibernate:hibernate-core:5.3.7.Final')
}
```

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')
}
```

If you remember earlier, we disabled security for `cloud-native-spring`, we'll do the same again for `cloud-native-spring-ui`. Open **cloud-native-spring-ui/src/main/java/io/pivotal/CloudNativeSpringUiApplication.java** for editing and make sure the contents look like so

```
package io.pivotal;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigu
rerAdapter;
import org.springframework.security.config.annotation.web.builders.WebSecurity;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;


@EnableDiscoveryClient
@SpringBootApplication
public class CloudNativeSpringApplication {

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

    @Configuration
    static class ApplicationSecurityOverride extends WebSecurityConfigurerAdapter {

        @Override
        public void configure(WebSecurity web) throws Exception {
            web
                .ignoring()
                    .antMatchers("/**");
        }
    }

}
```

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.openfeign.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:

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

Don't forget to add imports!

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 build -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
  buildpacks:
  - java_buildpack_offline
  stack: cflinuxfs3
  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 implement a factory that creates a fallback implementation (that way we get a handle on any exception so we can log out what caused the fallback to occur). We'll also reference that class as a fallback in our *@FeignClient* annotation. First, create this class in the *io.pivotal* package:

```
@Slf4j
@Component
class CityClientFallbackFactory implements FallbackFactory<CityClient> {

    @Override
    public CityClient create(final Throwable t) {
        return new CityClient() {
            @Override
            public Resources<City> getCities() {
                log.info("Fallback triggered by {} due to {}",
t.getClass().getName(), t.getMessage());
                return new Resources<City>(Collections.emptyList());
            }
        };
    }
}
```

Hint: imports should start with `java.util`, `io.pivotal`, `feign.hystrix`, `lombok.extern.slf4j`, `org.springframework.stereotype`, and `org.springframework.hateoas`.

4. Also modify the *@FeignClient* annotation to reference this factory implementation in case of failure:

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

5. Finally, enable Hystrix instrumentation on the Feign client by adding the following key-value to `application.yml`

```
feign:
  hystrix:
    enabled: true
```

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 trial 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):
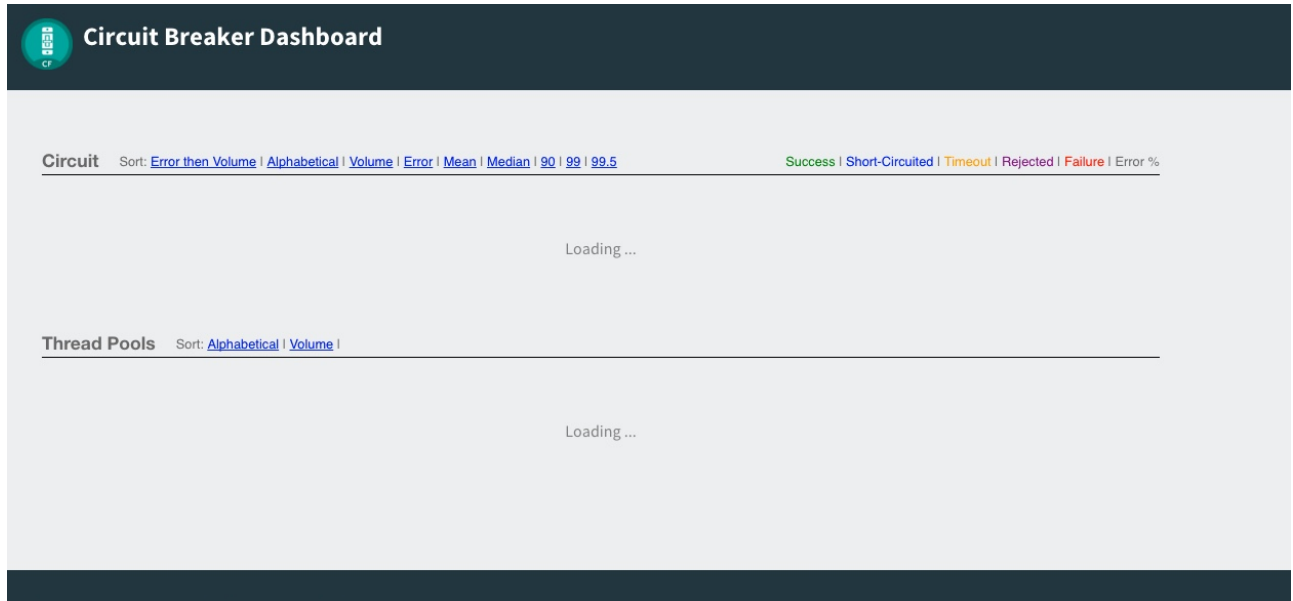
3. We will now bind our application to our *circuit-breaker-dashboard* within our Cloud Foundry deployment manifest. Add this additional reference to a service at the bottom of **/cloud-native-spring-ui/manifest.yml** in the services list:

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

Deploy and test application

1. Build the application

```
gradle build -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)
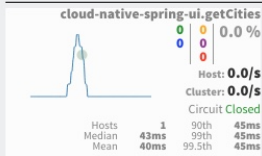
```
cf start cloud-native-spring
```

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:

Circuit Breaker Dashboard

Circuit   Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5   Success | Short-Circuited | Timeout | Rejected | Failure | Error %

cloud-native-spring-ui.getCities

0	0	0.0 %
0	0	
0	0	

Host: **0.0/s**

Cluster: **0.0/s**

Circuit Closed

Hosts	1	90th	45ms
Median	43ms	99th	45ms
Mean	40ms	99.5th	45ms

Thread Pools   Sort: Alphabetical | Volume |

CityService

Host: **0.0/s**

Cluster: **0.0/s**

Active	0	Max Active	0
Queued	0	Executions	0
Pool Size	7	Queue Size	5