DevOps Workshop Lab Guide

Table of Contents

Introduction: Labs Setup . . . 3

Overview . . . 3

PCF Environment Access . . . 3

Apps Manager UI . . . 3

01. Building a Spring Boot Application . . . 5

Getting started . . . 5

Add an Endpoint . . . 6

Build the *cloud-native-spring* application . . . 7

Run the *cloud-native-spring* application . . . 7

02. Enhancing Boot Application with Metrics . . . 9

Set up the Actuator . . . 9

Include Version Control Info . . . 10

Health Indicators . . . 11

Metrics . . . 13

03. Creating a Dockerfile . . . 15

04. Deploying to Kubernetes . . . 16

Creating a namespace . . . 16

Creating a deployment . . . 16

Creating a service . . . 17

Exposing a DNS Record . . . 18

Scaling . . . 18

Testing our results . . . 18

Making changes . . . 19

Logging . . . 19

Monitoring . . . 19

The bottom line: If you got all of the requirements above working well, congradulations - you built your own platform on top of Kubernetes! . . . 19

05. Adding Persistence to our Boot Application . . . 22

Create a Hypermedia-Driven RESTful Web Service with Spring Data REST (using JPA) . . . 22

Add the domain object - City . . . 22

Use Flyway to manage schema . . . 24

Run the *cloud-native-spring* Application . . . 24

Importing Data . . . 25

Adding Search . . . 27

06. Running with persistence in Docker . . . 32

Setting up the database docker container . . . 32

- Setting up the application 32
- Running with persistence in Kubernetes 36
- Source of Database 36

The bottom line 36

Introduction to CF CLI 37

- Open Apps Manager 37
- How to target a foundation and login 37
- How to deploy an application 37
- How to cleanup after yourself 39
- Where to go for more help 39

Deploy *cloud-native-spring* to Pivotal Cloud Foundry 40

- Exploring actuator in Apps Manager 42

Adding Spring Cloud Config to Boot Application 47

- Update *Hello* REST service 47
- Run the *cloud-native-spring* Application and verify dynamic config is working 50
- Create Spring Cloud Config Server instance 50
- Deploy and test application 53

Adding Service Registration and Discovery with Spring Cloud 55

- Update *Cloud-Native-Spring* Boot Application to Register with Eureka 55
- Create Spring Cloud Service Registry instance and deploy application 55
- Deploy and test application 56
- Create another Spring Boot Project as a Client UI 57
- Deploy and test application 62

Introduction: Labs Setup

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. There is also a *samples* directory, containing completed applications that 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

01. Building a Spring Boot Application

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

Getting started

Although we will use a pre-created initial skeleton, it's important you'll learn how to use the Spring Initializr. Head over to the URL and enter the following details:

1. Select a Gradle Project (projects are usually built using gradle or maven)
2. Select Java and the target langauge.
3. Latest stable version
4. **group**: io.pivotal
5. **artifact**: cloud-native-spring
6. Search for the following dependencies:
    a. Web
    b. Hateoas
    c. Rest Repositories
    d. JPA
    e. Actuator
    f. Lombok
7. Click "Generate"
8. Observe the contents of the downloaded ZIP file. This is the structure of a standard Spring Boot application. Code goes into `src/main/java`, properties or static content goes into `src/main/resources`, tests go into `src/test/java`.

Now let's continue with the pre-made skeleton.

1. Open a Terminal (e.g., *cmd* or *bash* shell)
2. Clone the pre-existing git repository:

    ```
    git clone https://github.com/odedia/devops-workshop.git
    ```

3. Change the working directory to be `devops-workshop/labs/my_work/cloud-native-spring`

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

4. Open this project in your editor/IDE of choice (InteliJ is recommended).

```
idea .
```

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`).

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

   Completed:

   ```
   package io.pivotal.controller;

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

   @RestController
   public class GreetingController {

       @GetMapping("/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`

We're going to use Gradle to build and package artifacts. If you don't already have gradle installed, don't worry, we have you covered. You can use the embedded gradlew Wrapper.

1. Find out what tasks are available to you with

   ```
   ./gradlew tasks
   ```

2. First we'll run tests

   ```
   ./gradlew test
   ```

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

   ```
   ./gradlew 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 web server and a servlet container)

   ```
   ./gradlew build
   ```

5. Examine the contents of the `build/libs` directory. You should see the final Spring Boot *jar* file. This jar file is completly portable - it contains everything that app needs, including an embedded Web server. This is why it is so big.

Run the *cloud-native-spring* application

Now we're ready to run the application

1. Run the application with

   ```
   ./gradlew 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**

02. 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 {
       implementation('org.springframework.boot:spring-boot-starter-actuator')
       // other dependencies omitted
   }
   ```

2. Run the application again using `./gradlew bootRun` and then check the application's metrics at http://localhost:8080/actuator.
3. Stop the application by typing **Ctrl+C**.

   By default Spring Boot does not expose all the 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.

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

   ```
   management:
     endpoints:
       web:
         exposure:
           include: "*"
   ```

5. Run the updated application

   ```
   gradle clean bootRun
   ```

   Try out the following endpoints. The output is omitted here because it can be quite large:

   curl http://localhost:8080/actuator/health

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

   curl http://localhost:8080/actuator/beans

   → Displays all of the beans in the Spring context.

curl http://localhost:8080/actuator/configprops

→ Displays a collated list of all @ConfigurationProperties.

curl http://localhost:8080/actuator/env

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

curl http://localhost:8080/actuator/mappings

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

curl http://localhost:8080/actuator/threaddump

→ Displays a thread dump of the currently running application in JSON format.

curl http://localhost:8080/actuator/heapdump

→ Downloads a heap dump that you can import into a JVM profiler such as JProfiler.

curl http://localhost:8080/actuator/httptrace

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

6. 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 *plugins{}* section

   ```
   plugins {
       id 'com.gorylenko.gradle-git-properties' version '2.2.0'
   }
   ```

You'll also configure the plugin by adding a *gitProperties{}* block.

+

```
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.

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

   ```
   gradle clean bootRun
   ```

2. Let’s verify that Git commit information is now included

   ```
   curl http://localhost:8080/actuator/info
   ```

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

3. 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 curl to verify the output of the health endpoint

   ```
   curl http://localhost: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.

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. Browse to http://localhost:8080/actuator/metrics/jvm.memory.used to see how much memory is currently being used.
3. Stop the cloud-native-spring application.

03. Creating a Dockerfile

Now that we have a running application, we want to run it in Kubernetes. But before we can do it, we'll need to create a docker file (or, to be more precise: an OCI-compliant image).

For a simple demo application in this workshop, any choice is probably fine. However, when you go to production things get a big more complicated:

1. What sould be our base operating system? There are many choices. Ubuntu, RedHat, Suse, CentOS.
2. Who will be in charge of patching the OS?
3. What version of Java should we use? There are *many* choices. Search Docker hub for Java and see the various options. Who performs CVE Patching on these images? (Hint: if you don't know the answer, it's probably you).

Write a Dockerfile for this app. Select one of the options from Docker hub for now.

1. The jar file containing the application is under *build/libs/cloud-native-spring-1.0-SNAPSHOT.jar*.
2. The command to run the java application is

```
java -jar <file-name>.jar
```

→ Hint: Use the `ENTRYPOINT` command at the end of your Dockerfile to run the app.

Let's build the docker image:

```
docker build -t cloud-native-spring .
```

Now let's run this container locally to make sure things still work.

```
docker run -p 8080:8080 cloud-native-spring
```

Go to http://localhost:8080/hello and check the results.

Tag the image and push it to docker.io.

```
docker tag cloud-native-spring <your-username>/cloud-native-spring
docker push <your-username>/cloud-native-spring
```

04. Deploying to Kubernetes

In this lab we'll deploy our very simple application to Kubernetes, and try to make it production ready.

Login to PKS cluster shared by the instructor.

Creating a namespace

Create a new namespace for your team:

```
kubectl create namespace <my-team>
```

Creating a deployment

Create a deployment manifest to run the image we just deployed. Here's a skeleton you can use (or write your own):

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    run:
  name:
  namespace: default
spec:
  replicas:
  selector:
    matchLabels:
      run:
  strategy:
    rollingUpdate:
      maxSurge:
      maxUnavailable:
    type:
  template:
    metadata:
      labels:
        run:
    spec:
      containers:
      - image:
        imagePullPolicy:
        name:
        ports:
        - containerPort:
          protocol:
      dnsPolicy:
      restartPolicy:
```

Creating a service

Since we're using PKS, we're lucky - we can use a `LoadBalancer`. If we were to use another solution we might have to revert to a `NodePort` or to implement other solutions. Complete the skeleton yaml below, or write your own:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run:
  name:
  namespace:
spec:
  ports:
  - nodePort:
    port:
    protocol:
    targetPort:
  selector:
    run:
  sessionAffinity: None
  type:
```

Exposing a DNS Record

There are several ways to expose the service as a routable URL:

1. You can create an A record pointing to the load balancer. You will have to make sure the IP doesn't change.

   → Problem: Developers rarely have access to DNS, which means waiting for tickets between development and IT.

2. You can create an ingress gateway that routes to the deployment, which requires installing an additional ingress service to the cluster such as nginx.
3. You can leverage Istio/KNative

For now, let's just use the External IP we got.

Scaling

Edit the deployment yaml so that there are 3 pods instead of 1.

Testing our results

Check your external IP by running:

```
kubectl get svc -n <my-team-namespace>
```

Open http://<my-external-ip>/hello and make sure you got a response.

Making changes

One of the main advantages of going cloud-native is to have a fast feedback loop. What would happen if you were to make a single change in the code right now?

1. Change the greeting message from "Hello World!" to "Hello VMware!".
2. Get your new code to a running state in Kubernetes.

Logging

Check the logs of one of the pods by running `kubectl logs <pod-name> -n <my-team-namespace>`.

Logs from one pod is nice, but your application is being served from multiple pods. How can you get the logs from all pods of your app?

1. You can use sidecar containers to manually handle logging to a central solution
2. You can install Fluentd daemon sets (requires privilege access to `kube_system` namespace)
3. You can use commercial solutions such as Splunk, Datadog, SumoLogic, Log Insight etc. (at an added cost)
4. You can use open source solutions such as ELK, Graylog (but it is now your responsibility to maintain and upgrade this solution)

Monitoring

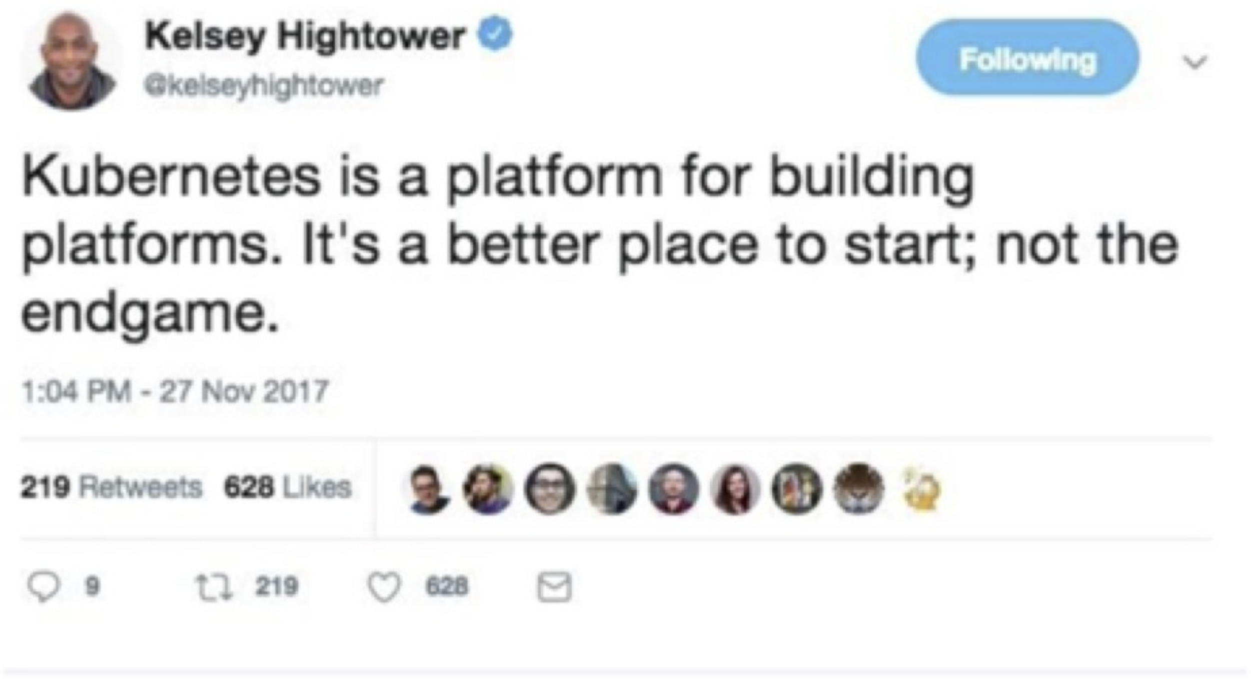
Our container provides basic metric information. We can get some of the data by running:

```
kubectl describe pod <my-pod> -n <my-team-namespace>
```

But this will only give us information on a specific pod. What about connections between pods or deployments? How can we find our own metrics that we expose via actuator? We can query the */actuator* URL but this will only give a response from *one* of the pods.

1. You can use commercial solutions such as SysDig, Dynatrace, NewRelic, Wavefront (at an added cost)
2. You can use open source solutions such as Kibana, Prometheus, Grafana (but it is your responsibility to maintain and upgrade them)

The bottom line: If you got all of the requirements above working well, congradulations - you built your own platform on top of Kubernetes!



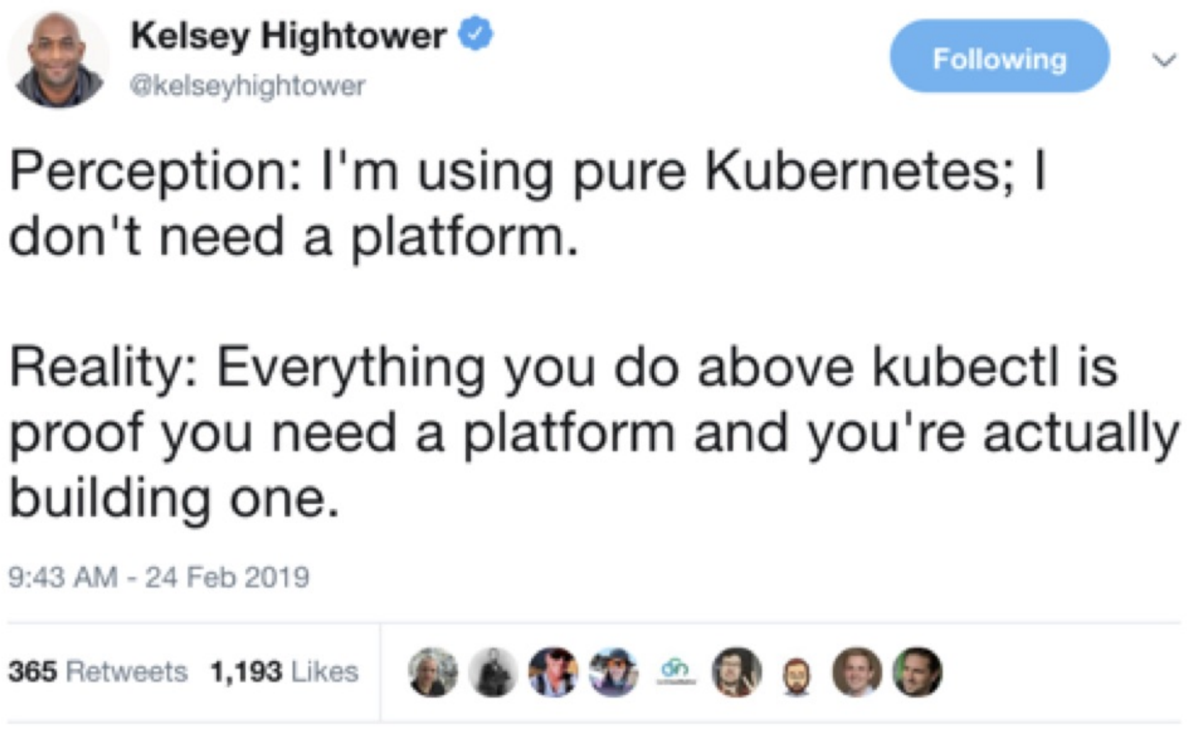
Kelsey Hightower
@kelseyhightower
Following

Kubernetes is a platform for building platforms. It's a better place to start; not the endgame.

1:04 PM - 27 Nov 2017

219 Retweets 628 Likes

9 219 628



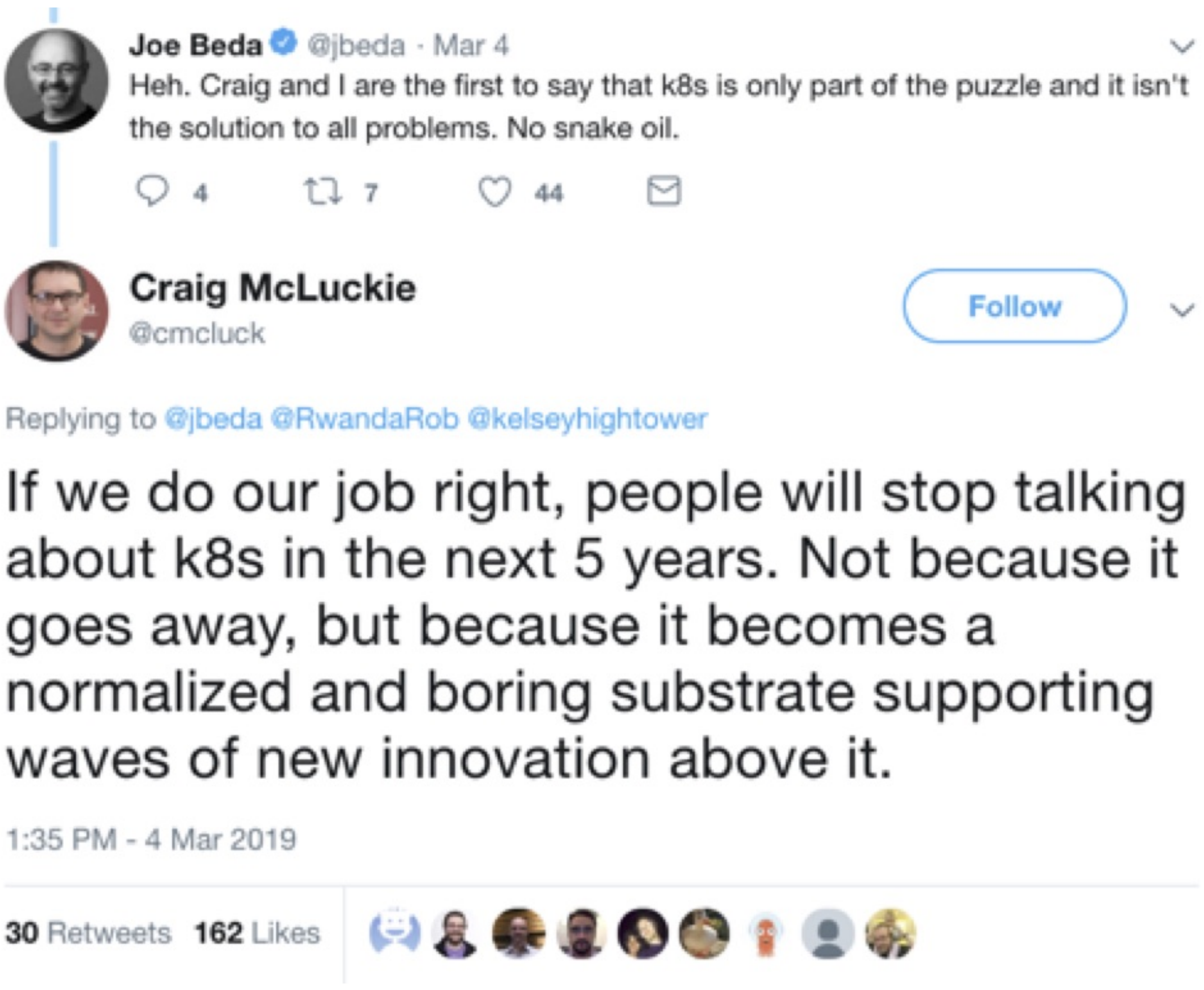
Kelsey Hightower
@kelseyhightower
Following

Perception: I'm using pure Kubernetes; I don't need a platform.

Reality: Everything you do above kubectl is proof you need a platform and you're actually building one.

9:43 AM - 24 Feb 2019

365 Retweets 1,193 Likes



Joe Beda @jbeda · Mar 4
Heh. Craig and I are the first to say that k8s is only part of the puzzle and it isn't the solution to all problems. No snake oil.

4 7 44

Craig McLuckie
@cmcluck

Replying to @jbeda @RwandaRob @kelseyhightower

If we do our job right, people will stop talking about k8s in the next 5 years. Not because it goes away, but because it becomes a normalized and boring substrate supporting waves of new innovation above it.

1:35 PM - 4 Mar 2019

30 Retweets 162 Likes

05. Adding Persistence to our 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. 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(strategy = GenerationType.IDENTITY)
    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

```
implementation('org.flywaydb:flyway-core:5.2.4')
implementation('com.zaxxer:HikariCP:3.3.0')
```

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.

   ```
   curl 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, 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).

```
curl http://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 http://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 http://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=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

06. Running with persistence in Docker

So far we use the in-memory H2 database which is good for unit testing, but it's not feasible for production use - all our data will be gone whenever we restart the application.

Setting up the database docker container

First, we'll need to run a database. We'll use MySQL in our example. Searching Dockerhub, it seems that the default *mysql* image is the best option.

Before running the image, we need to setup a new docker network, so that our app container and our db container can talk to each other:

`docker network create mynet`

Now let's run the database in docker:

`docker run --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=my-secret-pw -d --net mynet mysql:5.7`

The default user for this image is `root`. This is the first sign that the defaults are not production ready, because you'd rarely run your database as root. Also, the image would run the container but does not take into account responsibilities such as upgrading, backups, auditing etc. These are your responsiblity.

Also, the image doesn't create any schemas other than the internal `sys` schema which we cannot use for our application. We'll need to create a new schema first. Let's ssh into our container:

`docker exec -it mysql bash`

Once in the container, we need to run the mysql cli (it would default to *localhost:3306* which is what we want):

`mysql -p`

Enter the password frmo above (`my-secret-pw`)

Now that we are in the cli, we can create our schema:

```
mysql> create schema my_db;
Query OK, 1 row affected (0.00 sec)
```

type `exit` twice to return to the host.

Setting up the application

We now need to add the MySQL JDBC (Java Database Connectivity) driver to our application. Add the following to `build.gradle` under `dependencies`:

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

We now have two drivers in our application: H2 and MySQL. How will Spring know which database I want to use?

Like everything else in Spring, it uses *convention over configuration* and common sense. If you didn’t provide connection parameters to MySQL, it would fallback to H2 since that’s the default, testable database it can use.

Let’s now define the connection parameters for our MySQL database. Update the `spring:database` section in `application.yml` so it would look like this:

```
spring:
  datasource:
    hikari:
      connection-timeout: 60000
      maximum-pool-size: 5
    url: jdbc:mysql://${MYSQL_HOST:localhost}:3306/my_db?useSSL=false
    username: root
    password: my-secret-pw
```

We can immediatly see another issue - our password is written in clear text in our configuration file. We can think of various ways to overcome this:

1. Using a Spring Cloud Config Server
2. Using environment variables when starting the server
3. Managing secrets when runing on Kubernetes (although remember - the default secret plugin is not encypted so it makes no difference!)

Compile the updated application:

`./gradlew build`

We can see another problem: our unit tests now use our "production" database, which is not desirable. Also, if our MySQL database is not running, our tests will fail. run `docker stop mysql` and try builing the application again - the tests would fail.

We’d like to keep using our H2 database for tests. We can do that by adding a different `application.yml` under `src/test/resources`. Anything that we’ll put in this file will *override* the default configuration onyl when tests are running.

Create the file `src/test/resources/application.yml` and populate the following for H2 Database:

```
spring:
  datasource:
    hikari:
      connection-timeout: 60000
      maximum-pool-size: 5
    driver-class-name: org.h2.Driver
    url: jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
    username: sa
    password: sa
```

Build the Docker image for our app again:

`docker build -t <my-id>/cloud-native-spring .`

And run it (notice the use of `--net` to allow us to communicate between two containers):

`docker run -p 8080:8080 --net mynet -e MYSQL_HOST=mysql <my-id>/cloud-native-spring`

→ Interesting point: If you used a base OS called `openjdk`, its going to use Java 11. If your local machine's JDK for compilation is JDK 8, the startup would fail. Make sure you have the correct pipeline of Java versions (or any other language for that matter).

See the log output to confirm you are connected to the new MySQL Database:

```
2020-02-02 12:24:01.154  INFO 7228 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
Sun Feb 02 12:24:01 IST 2020 WARN: Establishing SSL connection without server's
identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and
5.7.6+ requirements SSL connection must be established by default if explicit option
isn't set. For compliance with existing applications not using SSL the
verifyServerCertificate property is set to 'false'. You need either to explicitly
disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for
server certificate verification.
2020-02-02 12:24:01.704  INFO 7228 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2020-02-02 12:24:01.707  INFO 7228 --- [           main]
o.f.c.internal.database.DatabaseFactory  : Database: jdbc:mysql://localhost:3306/my_db
(MySQL 8.0)
2020-02-02 12:24:03.028  INFO 7228 --- [           main]
o.f.core.internal.command.DbValidate     : Successfully validated 2 migrations
(execution time 00:01.269s)
2020-02-02 12:24:03.056  INFO 7228 --- [           main]
o.f.core.internal.command.DbMigrate      : Current version of schema `my_db`: 1.0
2020-02-02 12:24:03.059  INFO 7228 --- [           main]
o.f.core.internal.command.DbMigrate      : Migrating schema `my_db` to version 1.1 -
seed data
2020-02-02 12:25:51.336  INFO 7228 --- [           main]
o.f.core.internal.command.DbMigrate      : Successfully applied 1 migration to schema
`my_db` (execution time 01:48.294s)
```

Verify you still get a response from `http://localhost:8080/cities`.

Running with persistence in Kubernetes

For the sake of time, we'll skip deploying the app again to Kubernetes, but think about the various considerations you'd have to make:

Source of Database

How will we install our database in Kubernetes?

1. We can install a database directly as an image like we just did locally, but that would present issues for upgrades, auditing etc.
2. We can use helm charts. Who would be responsible for upgrading them?
3. We can purchase database solutions based on Kubernetes operators / CRDs from a well-known vendor. The amount of production ready solutions is still low (Confluent, Greenplum, MongoDB).

Regardless of the solution, we need to think of the following:

1. We need to make sure that *only* our application can access the database and no other pods. This requires Kubernetes *taint* commands using labels.
2. We need to update our application to point to the new database URL, username and password.
3. We need to do this every time we move to other environments (such as other namespaces or Kubernetes clusters).
4. We need to store password in a well-encrypted store. The default kubernetes secret management uses base64 encoding **which is not an encryption solution**

The bottom line

If you got all of the requirements above working in a production-ready manner, congradulations - you built your own platform on top of Kubernetes!

+ image::images/k8s1.jpg[]

+ image::images/k8s2.jpg[]

+ image::images/k8s3.jpg[]

Introduction to CF CLI

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

  Note the sub-directories present..

  ```
  samples
   ├──── 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.

Open Apps Manager

Go to the URL https://apps.run.pcfone.io and login with the credentials provided by your instructor.

How to target a foundation and login

1. Open a Terminal (e.g., *cmd* or *bash* shell)
2. Target a foundation and login

```
$ cf login -a https://api.run.pcfone.io
```

+ Enter your account username and password, then select an org and space unless those were chosen for you automatically.

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? Before running *cf push*, always inspect the *manifest.yml* file in each directory.

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

   Notice that PCF also *built* the application for you before creating the container and running the app. You can also build your application locally (if you have `yarn` installed), and just provide the final artifact to PCF:

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

   cf push -c "node server.js"
   ```

5. Next, let's try deploying a Python app. Note that again, PCF builds the application for you:

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

6. Rinse and repeat for .Net Core (again, built during deployment):

   ```
   cd ../dotnet-core-sample
   cf push
   ```

7. Now let's push a Go app. Notice there's no *manifest.yml* in this directory. How did PCF know it's a Go app?

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

8. Check what applications have been deployed so far

   ```
     cf apps
   ```

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

Open Apps Manager and review your applications from the UI:

https://apps.run.pcfone.io

Click the "View App" link on the top right side of each app’s overview screen.

1. Let’s scale an app

   ```
   cf scale cf-nodejs -i 3
   cf apps
   ```

   Refresh the App’s URL and see how *INSTANCE INDEX* changes with each refresh.

2. 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

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. Our application needs to use a MySQL database. How will we deploy one to PCF? Simple, we'll use the marketplace:
2. Run the following command to review the various services availble in the marketplace: `cf marketplace`
3. Review the various plans that are available for the `mysql` service: `cf marketplace -s p.mysql`
4. Let's create a small MySQL Database: `cf create-service p.mysql db-small my-database`
5. Our database is being created. You can check its status by running: `cf service my-database`

→ All of these can also be done directly from the Apps Manager GUI. Go to the Marketplace link on the left-hand side. → Who was responsible for creating this database? It was the IT Department, and they probably added the service from the Pivotal Marketplace. The MySQL service in this case is a supported service from Pivotal. It also handles upgrades, backups, self-healing, monitoring with Healthwatch, auditing, and many other tasks that are not immediatly obvious.

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
  path: ./build/libs/cloud-native-spring-1.0-SNAPSHOT.jar
  buildpacks:
  - java_buildpack_offline   #(This can be automatically detected, but faster if
provided)
  services:
  - my-database
```

+ 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 under `buildpacks`, addand environment variable:

+

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

→ Notice the service in the `manifest.yml`? It's the service we just created. **That's all you need to do**

to make the connection between the application and the service. Remember how much work that required in Kubernetes?

1. Push application into Cloud Foundry

   ```
   cf push
   ```

   → To specify an alternate manifest and buildpack, you could run something like this:

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

2. Find the URL created for your app in the health status report. Browse to your app's `/hello` endpoint.
3. Check the log output

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

4. Check the `/cities` endpoint by going to `https://<your-specific-route>/cities`.

→ Wait a minute! How did this even work?! We didn't update the properties under `src/main/resources/application.yml`, yet somehow we were able to create the `city` table and populate it on startup. More so - we didn't even need to create a schema manually! How is that possible? That's the power of the Open Service Broker API (OSBAPI).

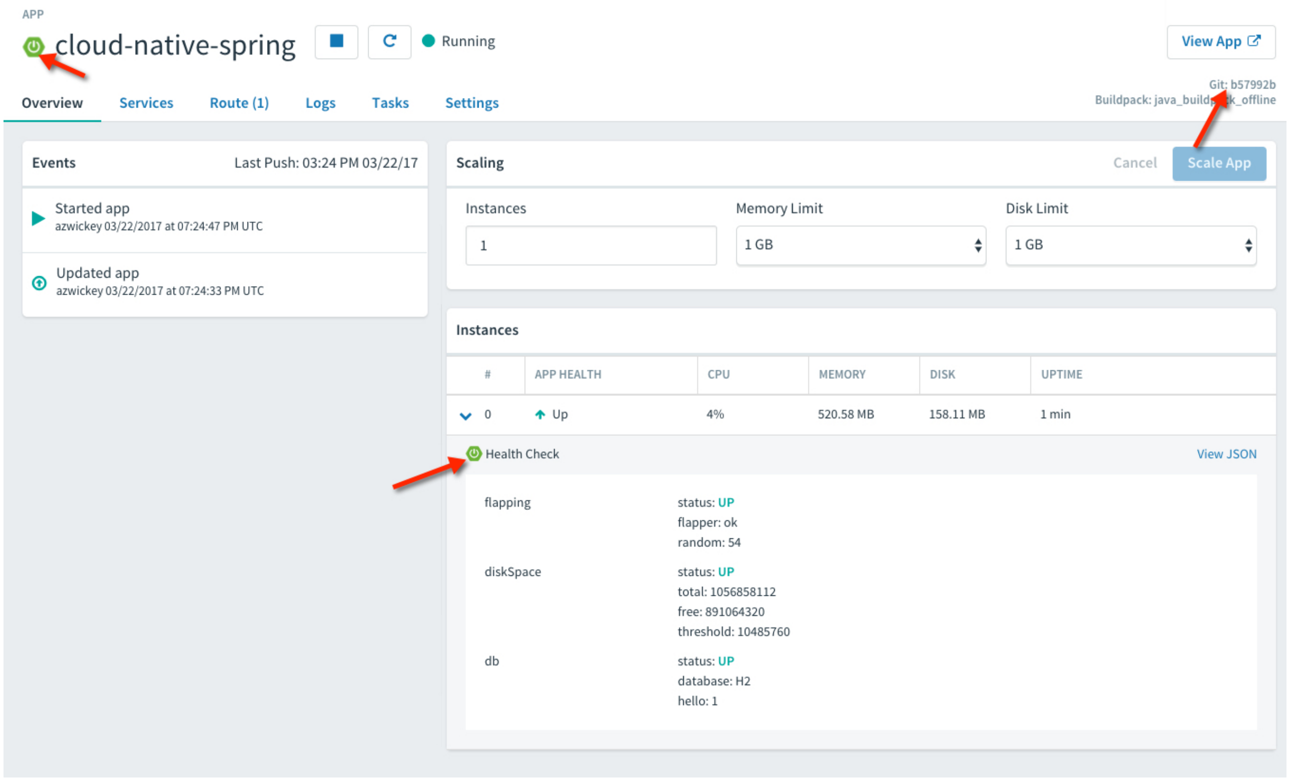
1. To fully understand this magic, run the following command: `cf env cloud-native-spring`
2. Notice the `p.mysql` entry under `VCAP_SERVICES`? That was injected when we told our application that we'd like to `bind` our app to the `my-database` service. It provides all the environment variables needed to connect to the database. Even more impressive - Spring (and .NET with Steeltoe) knows how to autoconfigure itself and bind to this new database auto-magically. The parameters injected by the platform will override the settings in your `application.yml`, so your code will always remain portable. Tomorrow you would move to a production environment and bind to a different database instance called `my-database`, but that's ok - all the parameters would be injected again, no code changes required.
3. Perhaps even more important - you no longer have passwords all over your code and your environment. **only** users that are authoried to access this particular space and app will have the option to view the parameters of the database that includes the password. You can also leverage a PCF service called *Credhub*, which is an encrypted database for properties. If you were to use Credhub, you wouldn't even see the parameters when running `cf env`. **Only** the application's PID process would be able to automatically fetch the parameters on container startup.
4. How fast would it be to change one line in our code and have it running with a routable URL? Let's change `GreetingController.java` to greet us with `Hello VMware on PCF!`. All that's left to do is:

`./gradlew build && cf push`

Check the updated results under `https://<your-specific-route>/hello`

Exploring actuator in Apps Manager

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.
2. Visit the route created for your app and append /actuator/health to see the health status report. See the same details in the Apps Manager UI:



Events — Last Push: 03:24 PM 03/22/17

Started app
azwickey 03/22/2017 at 07:24:47 PM UTC

Updated app
azwickey 03/22/2017 at 07:24:33 PM UTC

Scaling

Instances	Memory Limit	Disk Limit
1	1 GB	1 GB

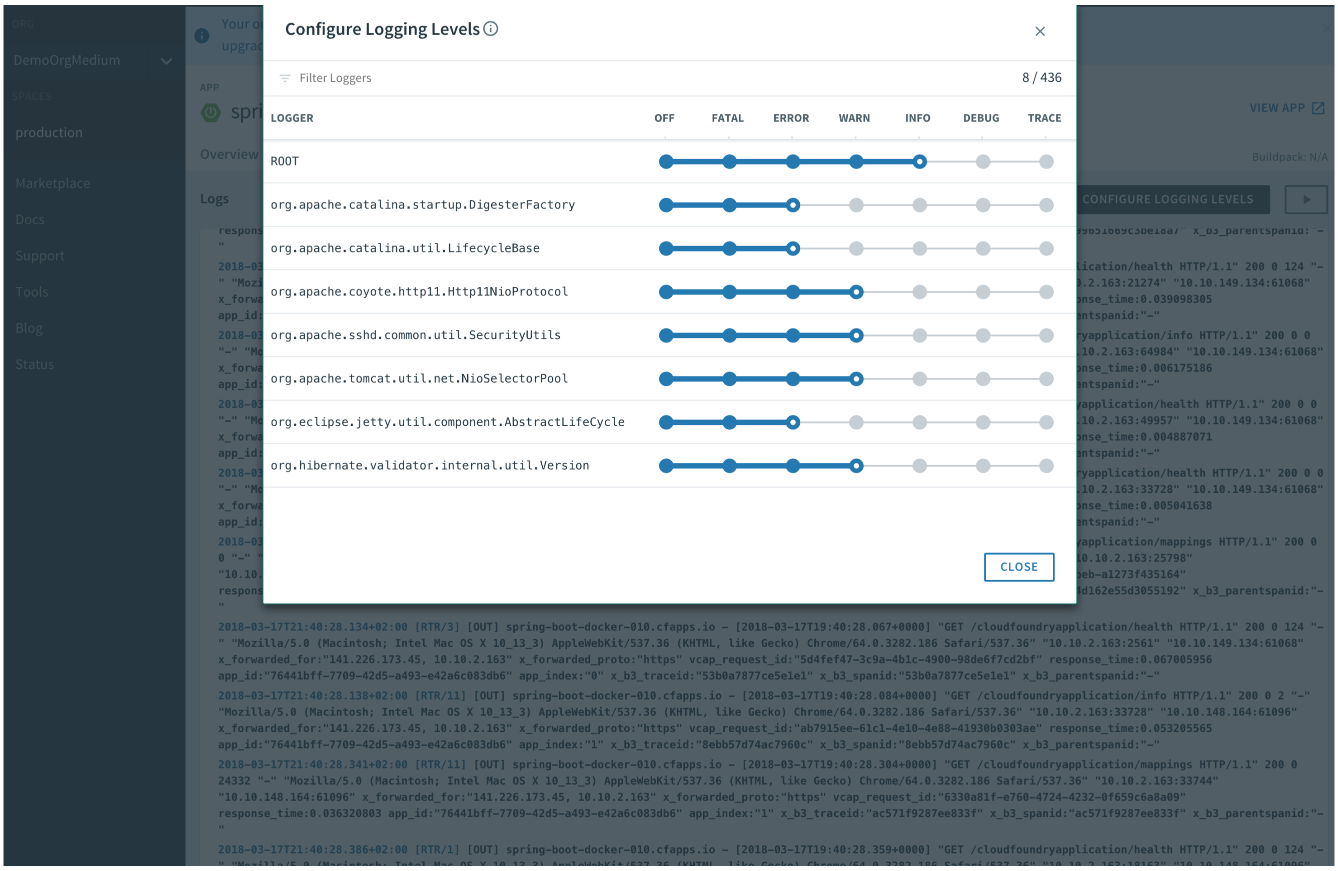
Instances

#	APP HEALTH	CPU	MEMORY	DISK	UPTIME
0	Up	4%	520.58 MB	158.11 MB	1 min

Health Check

flapping	status: UP flapper: ok random: 54
diskSpace	status: UP total: 1056858112 free: 891064320 threshold: 10485760
db	status: UP database: H2 hello: 1

The Logs tab provides an ***aggregated** view of all logs from all instances on our application. You can also change the logging level per class/package at runtime without having to restart:



Configure Logging Levels

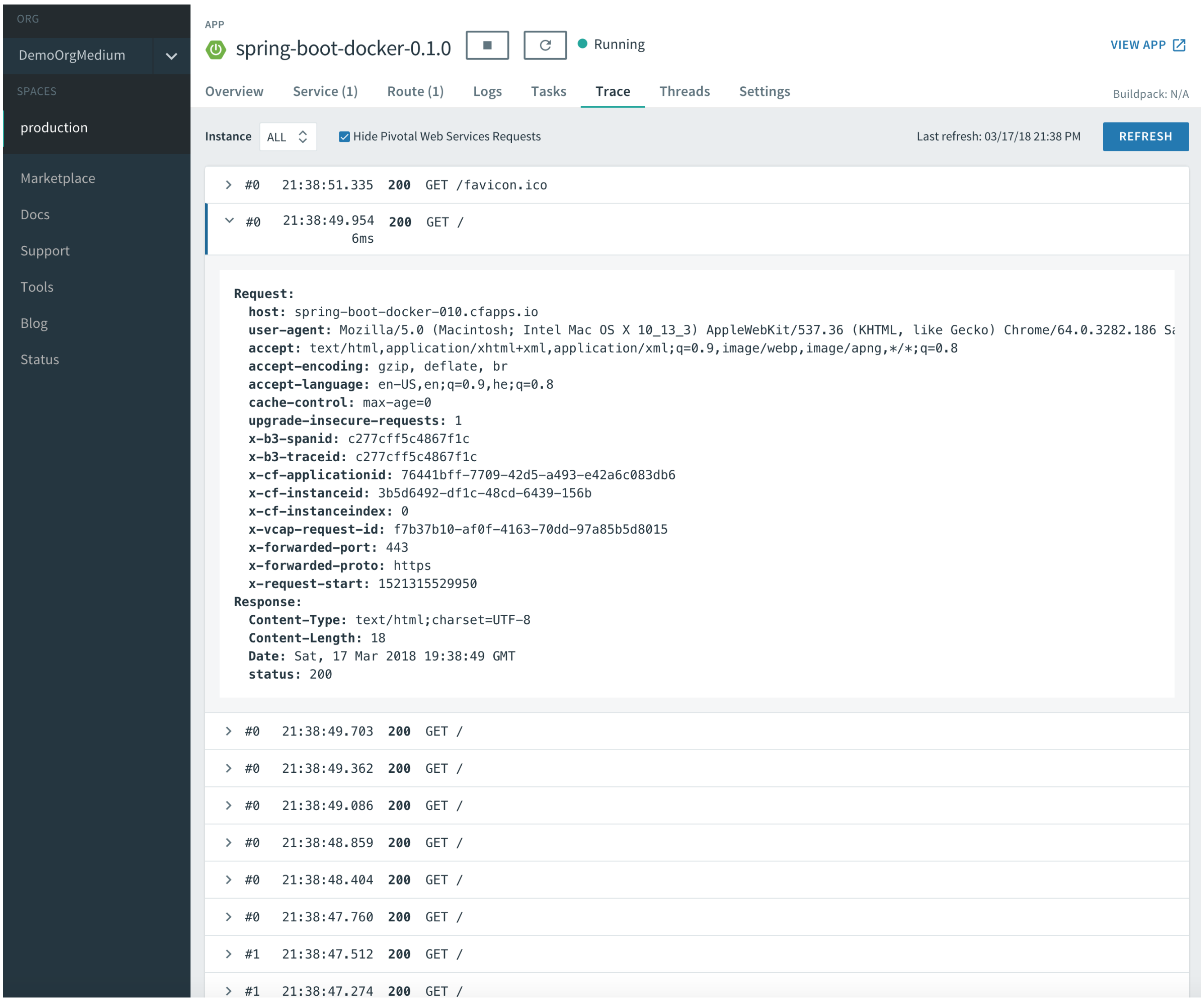
Filter Loggers 8 / 436

LOGGER	OFF	FATAL	ERROR	WARN	INFO	DEBUG	TRACE
ROOT	●	●	●	●	◉	○	○
org.apache.catalina.startup.DigesterFactory	●	●	◉	○	○	○	○
org.apache.catalina.util.LifecycleBase	●	●	◉	○	○	○	○
org.apache.coyote.http11.Http11NioProtocol	●	●	●	◉	○	○	○
org.apache.sshd.common.util.SecurityUtils	●	●	●	◉	○	○	○
org.apache.tomcat.util.net.NioSelectorPool	●	●	●	◉	○	○	○
org.eclipse.jetty.util.component.AbstractLifeCycle	●	●	◉	○	○	○	○
org.hibernate.validator.internal.util.Version	●	●	●	◉	○	○	○

CLOSE

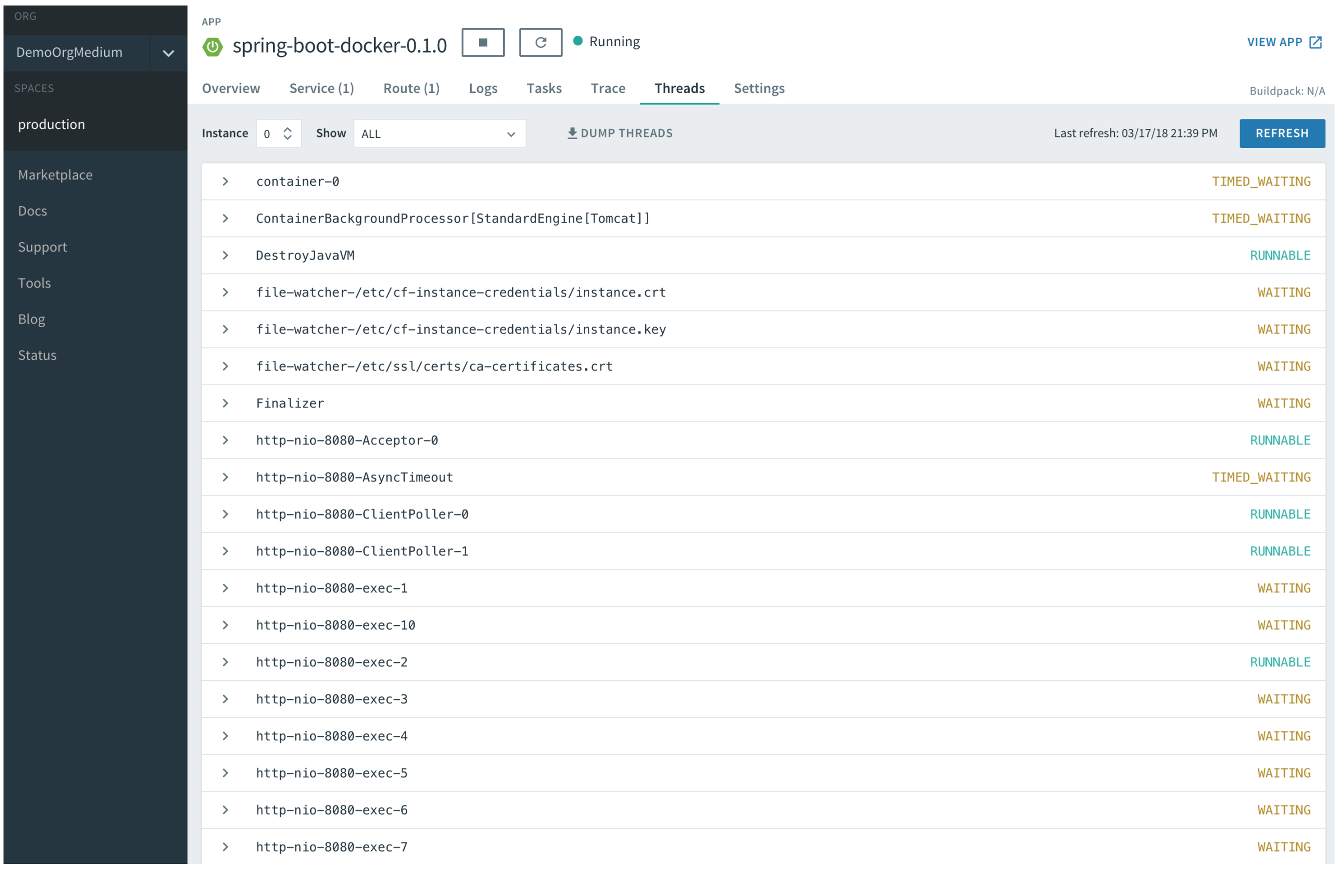
```
2018-03-17T21:40:28.134+02:00 [RTR/3] [OUT] spring-boot-docker-010.cfapps.io - [2018-03-17T19:40:28.067+0000] "GET /cloudfoundryapplication/health HTTP/1.1" 200 0 124 "-
" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36" "10.10.2.163:2561" "10.10.149.134:61068"
x_forwarded_for:"141.226.173.45, 10.10.2.163" x_forwarded_proto:"https" vcap_request_id:"5d4fef47-3c9a-4b1c-4900-98de6f7cd2bf" response_time:0.067005956
app_id:"76441bff-7709-42d5-a493-e42a6c083db6" app_index:"0" x_b3_traceid:"53b0a7877ce5e1e1" x_b3_spanid:"53b0a7877ce5e1e1" x_b3_parentspanid:"-"
2018-03-17T21:40:28.138+02:00 [RTR/11] [OUT] spring-boot-docker-010.cfapps.io - [2018-03-17T19:40:28.084+0000] "GET /cloudfoundryapplication/info HTTP/1.1" 200 0 2 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36" "10.10.2.163:33728" "10.10.148.164:61096"
x_forwarded_for:"141.226.173.45, 10.10.2.163" x_forwarded_proto:"https" vcap_request_id:"ab7915ee-61c1-4e10-4e88-41930b0303ae" response_time:0.053205565
app_id:"76441bff-7709-42d5-a493-e42a6c083db6" app_index:"1" x_b3_traceid:"8ebb57d74ac7960c" x_b3_spanid:"8ebb57d74ac7960c" x_b3_parentspanid:"-"
2018-03-17T21:40:28.341+02:00 [RTR/11] [OUT] spring-boot-docker-010.cfapps.io - [2018-03-17T19:40:28.304+0000] "GET /cloudfoundryapplication/mappings HTTP/1.1" 200 0
24332 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36" "10.10.2.163:33744"
"10.10.148.164:61096" x_forwarded_for:"141.226.173.45, 10.10.2.163" x_forwarded_proto:"https" vcap_request_id:"6330a81f-e760-4724-4232-0f659c6a8a09"
response_time:0.036320803 app_id:"76441bff-7709-42d5-a493-e42a6c083db6" app_index:"1" x_b3_traceid:"ac571f9287ee833f" x_b3_spanid:"ac571f9287ee833f" x_b3_parentspanid:"-
"
2018-03-17T21:40:28.386+02:00 [RTR/1] [OUT] spring-boot-docker-010.cfapps.io - [2018-03-17T19:40:28.359+0000] "GET /cloudfoundryapplication/health HTTP/1.1" 200 0 124 "-
```

The Trace tab shows you the recent REST API calls made to your application:



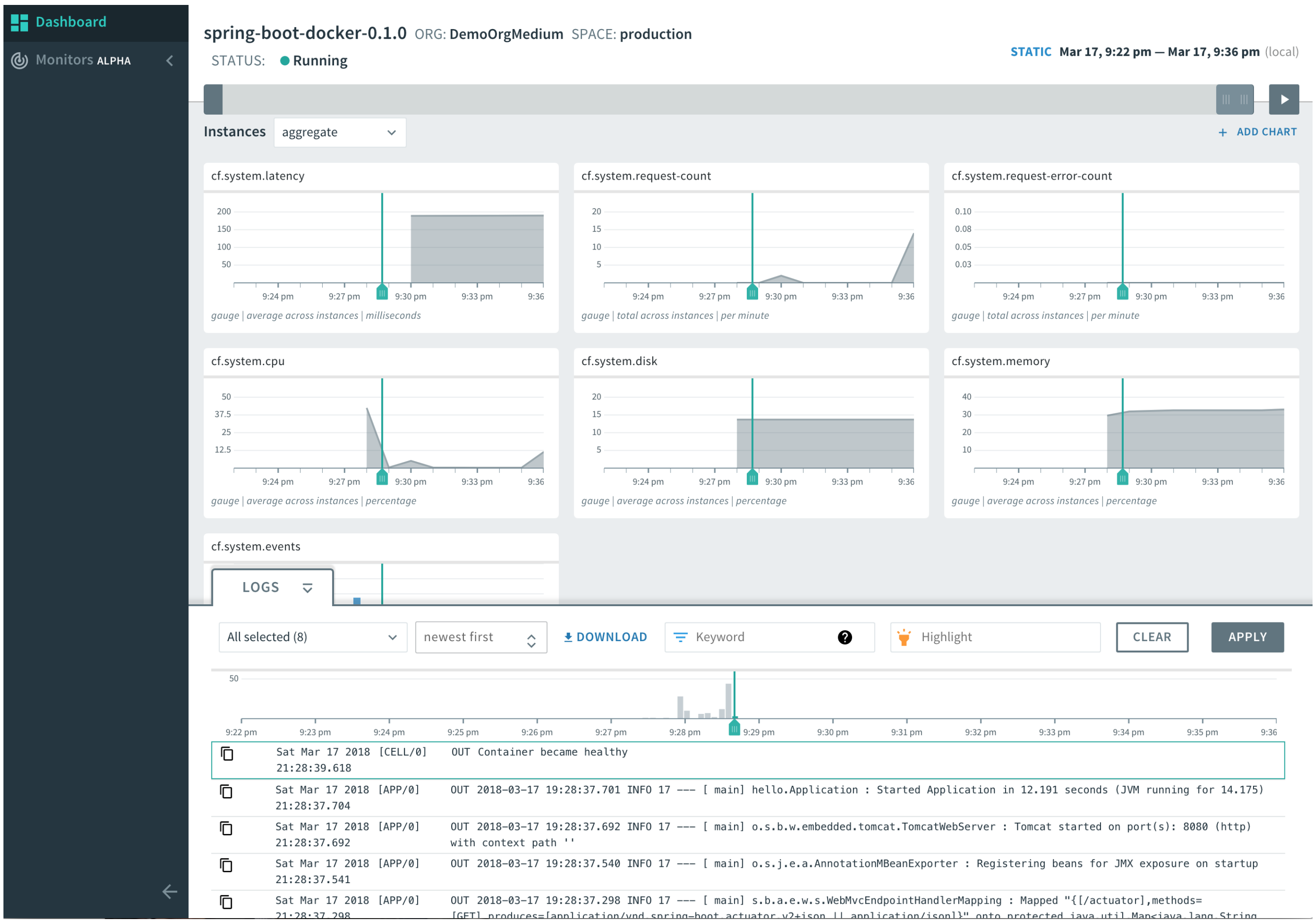
```
> #0   21:38:51.335  200  GET /favicon.ico
v #0   21:38:49.954  200  GET /
              6ms

Request:
  host: spring-boot-docker-010.cfapps.io
  user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Sa
  accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
  accept-encoding: gzip, deflate, br
  accept-language: en-US,en;q=0.9,he;q=0.8
  cache-control: max-age=0
  upgrade-insecure-requests: 1
  x-b3-spanid: c277cff5c4867f1c
  x-b3-traceid: c277cff5c4867f1c
  x-cf-applicationid: 76441bff-7709-42d5-a493-e42a6c083db6
  x-cf-instanceid: 3b5d6492-df1c-48cd-6439-156b
  x-cf-instanceindex: 0
  x-vcap-request-id: f7b37b10-af0f-4163-70dd-97a85b5d8015
  x-forwarded-port: 443
  x-forwarded-proto: https
  x-request-start: 1521315529950
Response:
  Content-Type: text/html;charset=UTF-8
  Content-Length: 18
  Date: Sat, 17 Mar 2018 19:38:49 GMT
  status: 200

> #0   21:38:49.703  200  GET /
> #0   21:38:49.362  200  GET /
> #0   21:38:49.086  200  GET /
> #0   21:38:48.859  200  GET /
> #0   21:38:48.404  200  GET /
> #0   21:38:47.760  200  GET /
> #1   21:38:47.512  200  GET /
> #1   21:38:47.274  200  GET /
```

The Threads tab provides visibility into all threads currently running in your JVM. You can also download a heap dump for offline investigation:



```
container-0                                                   TIMED_WAITING
ContainerBackgroundProcessor[StandardEngine[Tomcat]]          TIMED_WAITING
DestroyJavaVM                                                 RUNNABLE
file-watcher-/etc/cf-instance-credentials/instance.crt        WAITING
file-watcher-/etc/cf-instance-credentials/instance.key        WAITING
file-watcher-/etc/ssl/certs/ca-certificates.crt               WAITING
Finalizer                                                     WAITING
http-nio-8080-Acceptor-0                                      RUNNABLE
http-nio-8080-AsyncTimeout                                    TIMED_WAITING
http-nio-8080-ClientPoller-0                                  RUNNABLE
http-nio-8080-ClientPoller-1                                  RUNNABLE
http-nio-8080-exec-1                                          WAITING
http-nio-8080-exec-10                                         WAITING
http-nio-8080-exec-2                                          RUNNABLE
http-nio-8080-exec-3                                          WAITING
http-nio-8080-exec-4                                          WAITING
http-nio-8080-exec-5                                          WAITING
http-nio-8080-exec-6                                          WAITING
http-nio-8080-exec-7                                          WAITING
```

From the overview screen, clicking on PCF Metrics takes you to a full-blown monitoring dashboard where you can correlate system events along with the logs that happened at that time. Excellent for troubleshooting.


Dashboard
Monitors ALPHA
spring-boot-docker-0.1.0 ORG: DemoOrgMedium SPACE: production
STATUS: Running
STATIC Mar 17, 9:22 pm — Mar 17, 9:36 pm (local)
Instances aggregate
ADD CHART
cf.system.latency
gauge | average across instances | milliseconds
cf.system.request-count
gauge | total across instances | per minute
cf.system.request-error-count
gauge | total across instances | per minute
cf.system.cpu
gauge | average across instances | percentage
cf.system.disk
gauge | average across instances | percentage
cf.system.memory
gauge | average across instances | percentage
cf.system.events
LOGS
All selected (8)
newest first
DOWNLOAD
Keyword
Highlight
CLEAR
APPLY

```
Sat Mar 17 2018  [CELL/0]   OUT Container became healthy
21:28:39.618
Sat Mar 17 2018  [APP/0]    OUT 2018-03-17 19:28:37.701 INFO 17 --- [ main] hello.Application : Started Application in 12.191 seconds (JVM running for 14.175)
21:28:37.704
Sat Mar 17 2018  [APP/0]    OUT 2018-03-17 19:28:37.692 INFO 17 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http)
21:28:37.692                with context path ''
Sat Mar 17 2018  [APP/0]    OUT 2018-03-17 19:28:37.540 INFO 17 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
21:28:37.541
Sat Mar 17 2018  [APP/0]    OUT 2018-03-17 19:28:37.298 INFO 17 --- [ main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator],methods=
21:28:37.298                [GET],produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto protected java.util.Map<java.lang.String
```

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.

   ```
   plugins {
       id 'com.gorylenko.gradle-git-properties' version '2.0.0'
       id 'org.springframework.boot' version '2.0.9.RELEASE'
       id 'io.spring.dependency-management' version '1.0.7.RELEASE'
       id 'java'
   }

   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.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
           mavenBom "org.springframework.cloud:spring-cloud-dependencies:Finchley.SR3"
           mavenBom "io.pivotal.spring.cloud:spring-cloud-services-
   dependencies:2.0.3.RELEASE"
   ```

```
    }
}

dependencies {
    implementation('org.glassfish.jaxb:jaxb-runtime:2.4.0-b180830.0438')
    if (JavaVersion.current() != JavaVersion.VERSION_1_8) {
        implementation('org.javassist:javassist:3.23.0-GA')
    } else {
        implementation('org.javassist:javassist:3.22.0-GA')
    }
    annotationProcessor('org.projectlombok:lombok:1.18.6')
    implementation('org.projectlombok:lombok:1.18.6')
    implementation('org.springframework.boot:spring-boot-starter-actuator')
    implementation('org.springframework.boot:spring-boot-starter-data-jpa')
    implementation('org.springframework.boot:spring-boot-starter-data-rest')
    implementation('org.springframework.boot:spring-boot-starter-hateoas')
    implementation('org.springframework.data:spring-data-rest-hal-browser')
    implementation('org.springframework.boot:spring-boot-starter-web')
    implementation('org.springframework.boot:spring-boot-starter-security')
    implementation('io.pivotal.spring.cloud:spring-cloud-services-starter-config-
client')
    implementation('org.flywaydb:flyway-core:5.2.4')
    implementation('com.zaxxer:HikariCP:3.3.0')
    runtime('com.h2database:h2')
    runtime('mysql:mysql-connector-java:8.0.15')
    testImplementation('org.springframework.boot:spring-boot-starter-test')
}

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

bootRun {
    // support passing -Dsystem.property=value to bootRun task
    systemProperties = System.properties
}

tasks.withType(Test) {
    if (JavaVersion.current() != JavaVersion.VERSION_1_8) {
        jvmArgs += ["--add-opens", "java.base/java.lang=ALL-UNNAMED"]
    }
}
```

2. Add an *@Value* annotation, private field, and update the existing *@GetMapping* 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;

    @GetMapping("/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.GetMapping;
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;

    @GetMapping("/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 at runtime (Config servers will be protected by OAuth2). However, this will also enable basic authentication to all our service endpoints. We will need to add the following to conditionally open security (to ease local workstation deployment).

   In **build.gradle**, we'll need to add an *implementation* dependency

```
implementation('org.springframework.security:spring-security-config')
```

In **/cloud-native-spring/src/main/java/io/pivotal/CloudNativeSpringApplication.java** right underneath the `public static void main` method implementation, add

```
    @Configuration
    static class ApplicationSecurityOverride extends WebSecurityConfigurerAdapter {

        @Override
        public void configure(HttpSecurity http) throws Exception {
            http.csrf().disable();
            http.authorizeRequests().antMatchers("/**").permitAll();
        }
    }
```

Examine this Spring Boot reference for further details. We're going to disable cross-site request forgery. We are also explicitly deactivating security, allowing unauthorized requests to all endpoints.

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

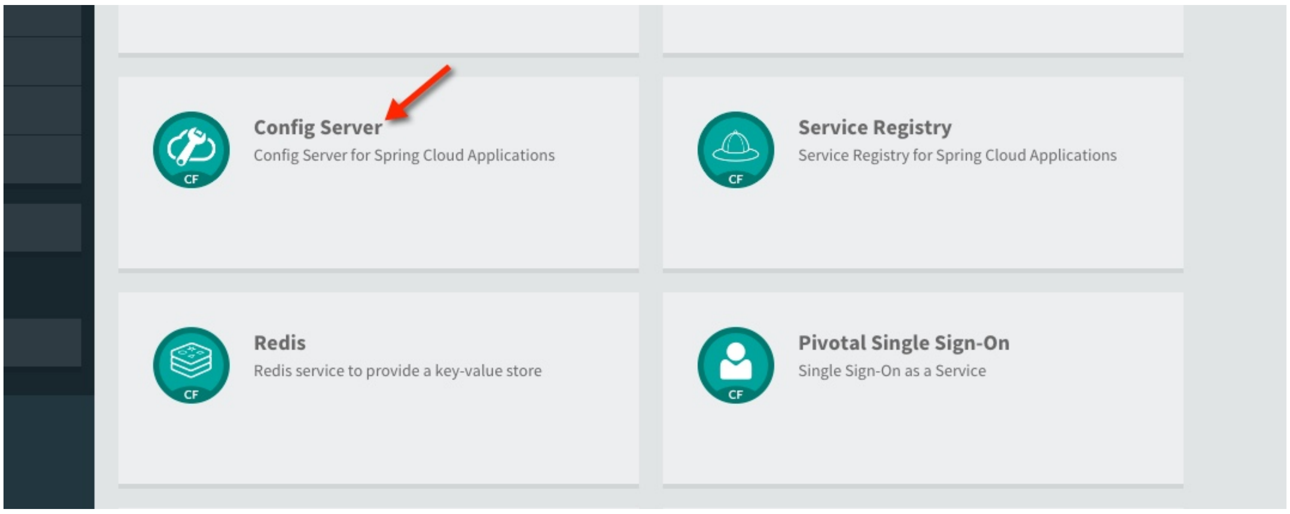
1. Run the application

```
gradle clean bootRun
```

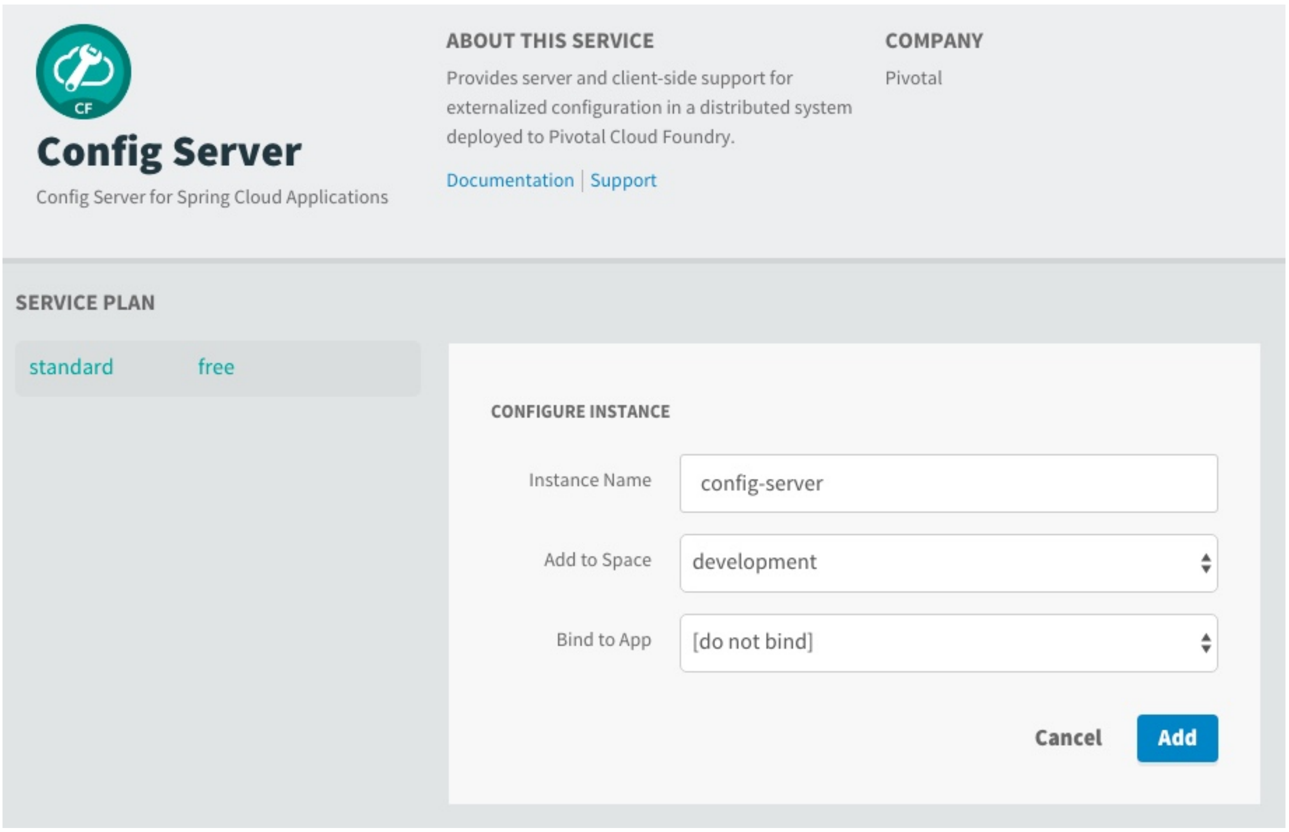
2. Browse to http://localhost:8080/hello and verify you now see your new greeting.
3. Stop the *cloud-native-spring* application

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:


Config Server
Config Server for Spring Cloud Applications
Service Registry
Service Registry for Spring Cloud Applications
Redis
Redis service to provide a key-value store
Pivotal Single Sign-On
Single Sign-On as a Service

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:


Config Server

Config Server for Spring Cloud Applications

ABOUT THIS SERVICE

Provides server and client-side support for externalized configuration in a distributed system deployed to Pivotal Cloud Foundry.

COMPANY

Pivotal

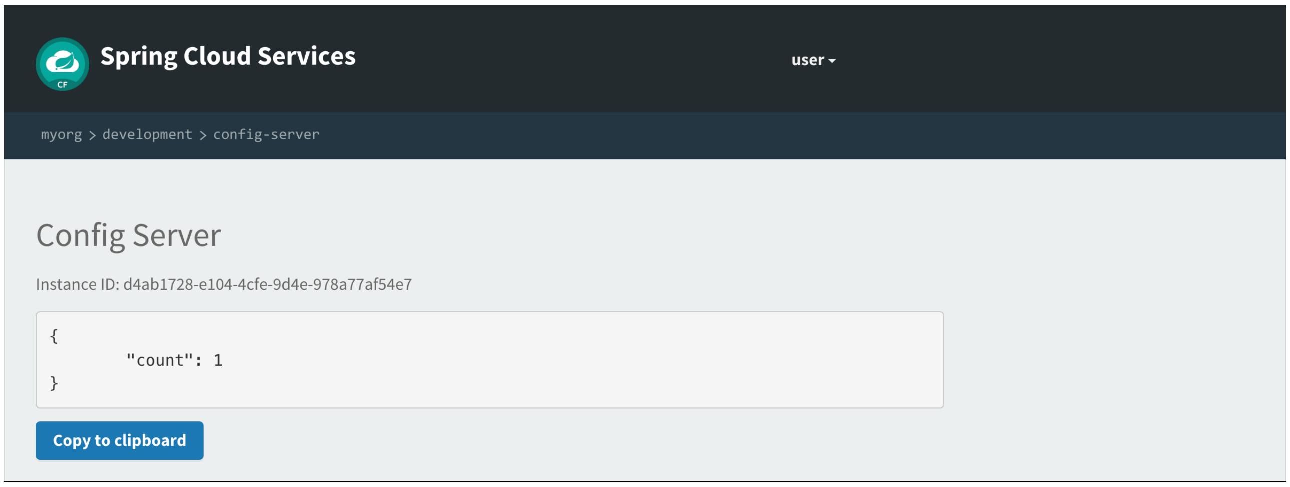
SERVICE PLAN

standard free

CONFIGURE INSTANCE

Instance Name	config-server
Add to Space	development
Bind to App	[do not bind]

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:



Config Server

Instance ID: d4ab1728-e104-4cfe-9d4e-978a77af54e7

```
{
	"count": 1
}
```

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

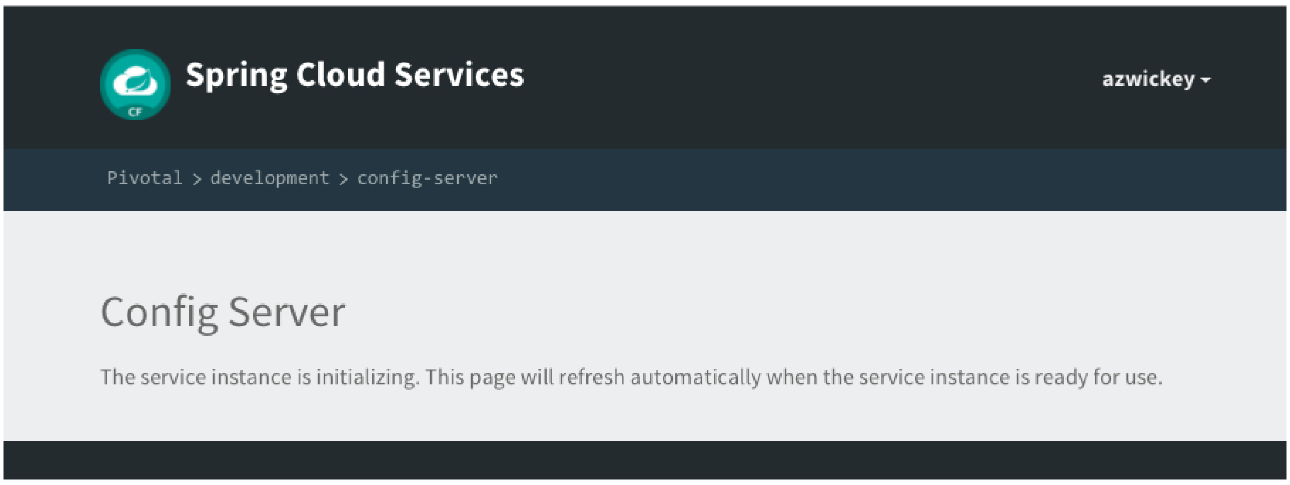
   ```
   {
     "git": {
       "uri": "https://github.com/pacphi/config-repo"
     }
   }
   ```

   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:



Config Server

The service instance is initializing. This page will refresh automatically when the service instance is ready for use.

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.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
       implementation('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 {
   ```

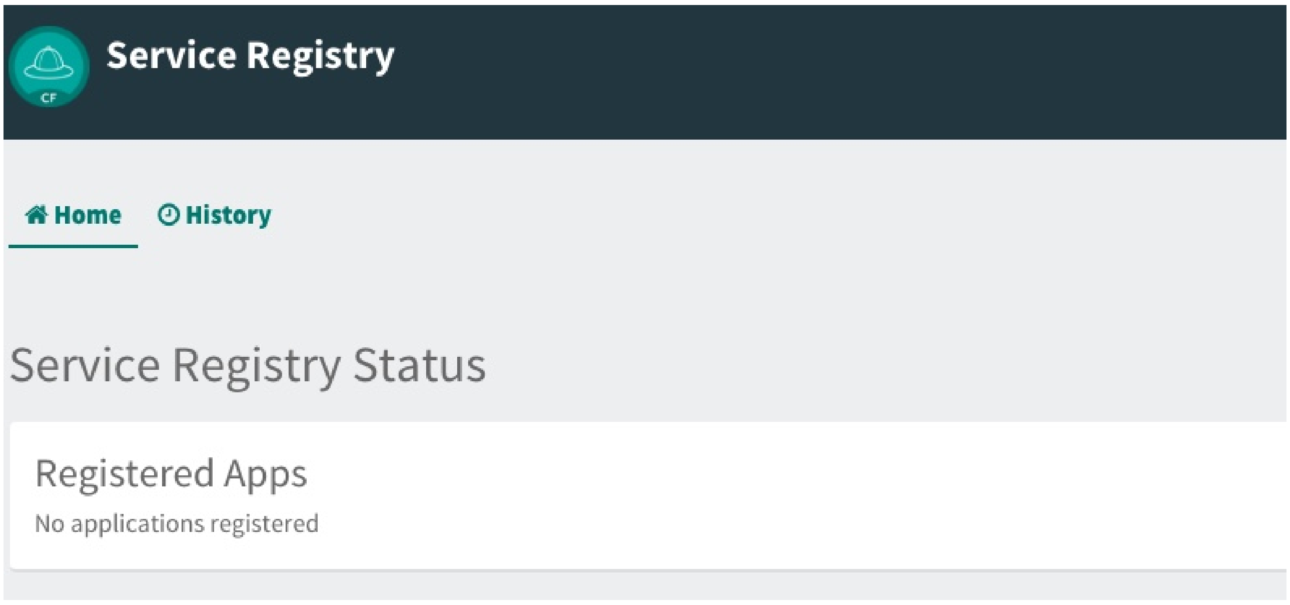
Create Spring Cloud Service Registry instance and deploy application

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:



Service Registry Status

Registered Apps

No applications registered

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

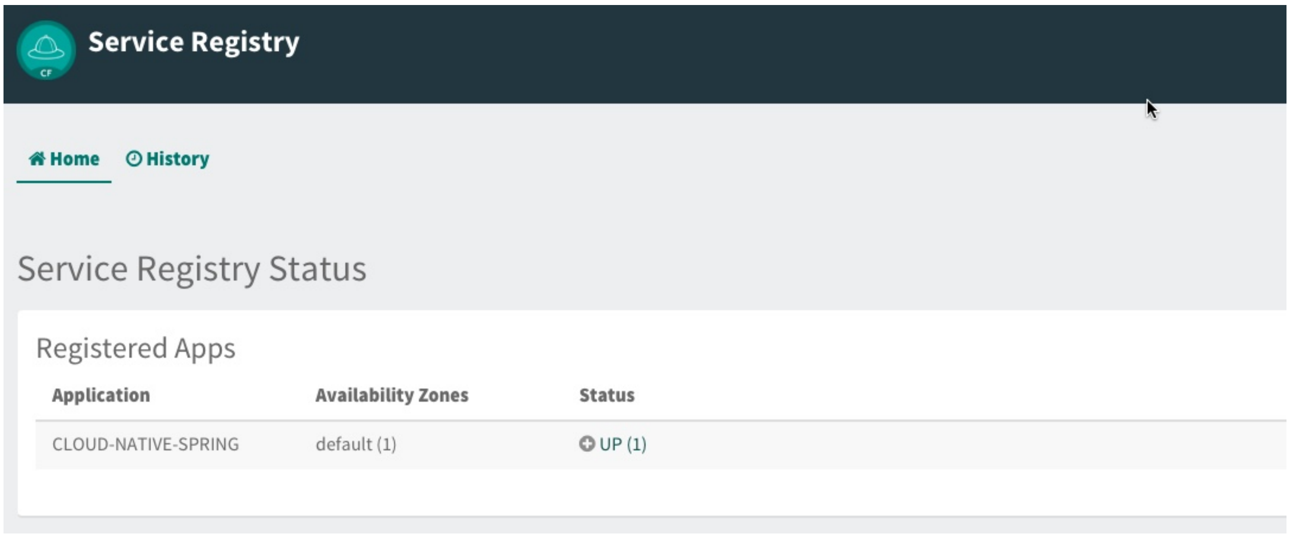
1. Build the application

```
gradle build
```

2. Push application into Cloud Foundry

```
cf push
```

3. 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:


Service Registry Status

Registered Apps

Application	Availability Zones	Status
CLOUD-NATIVE-SPRING	default (1)	UP (1)

4. 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
     implementation('io.pivotal.spring.cloud:spring-cloud-services-starter-service-registry')
   implementation('org.springframework.cloud:spring-cloud-starter-openfeign')
   implementation('org.springframework.cloud:spring-cloud-starter-netflix-ribbon')
   implementation('org.hibernate:hibernate-core:5.4.1.Final')
}
```

Next, we're going to add some annotations to enable service discovery and hypermedia support. In addition, we're going to embed configuration and implementation for handling marshalling/unmarshalling of hal+json. Finally, we'll override security behavior just as we did in `cloud-native-spring`, adding exceptions for static resources provided by Vaadin.

Open **cloud-native-spring-ui/src/main/java/io/pivotal/CloudNativeSpringUiApplication.java** for editing and make sure the contents look like so

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableHypermediaSupport(type = EnableHypermediaSupport.HypermediaType.HAL)
public class CloudNativeSpringUiApplication {

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

    @Configuration
    static class ClientConfig implements WebMvcConfigurer {

        @Autowired
        private HalHttpMessageConverter halHttpMessageConverter;

        @Override
        public void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
            converters.add(halHttpMessageConverter);
        }
    }

    @Configuration
    static class ApplicationSecurityOverride extends WebSecurityConfigurerAdapter {

    @Override
    public void configure(HttpSecurity web) throws Exception {
      web.csrf().disable();
      web.authorizeRequests().antMatchers("/**").permitAll();
    }

        @Override
        public void configure(WebSecurity web) throws Exception {
              web.ignoring().antMatchers(
          // Vaadin Flow static resources
          "/VAADIN/**",

          // the standard favicon URI
          "/favicon.ico",

          // the robots exclusion standard
```

```
            "/robots.txt",

            // web application manifest
            "/manifest.webmanifest",
            "/sw.js",
            "/offline-page.html",

            // (development mode) static resources
            "/frontend/**",

            // (development mode) webjars
            "/webjars/**",

            // (production mode) static resources
            "/frontend-es5/**", "/frontend-es6/**");
          }
      }

  }
```

Don’t forget to adjust the imports!

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.hateoas.Resources;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;

import io.pivotal.domain.City;

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

  @GetMapping(value = "/cities")
  Resources<City> findAll(@RequestParam("page") int page, @RequestParam("size") int
limit);

  @PostMapping(value = "/cities")
  City add(@RequestBody City company);

  @PutMapping(value = "/cities/{id}")
  City update(@PathVariable("id") Long id, @RequestBody City city);

  @DeleteMapping(value = "/cities/{id}")
  void delete(@PathVariable("id") Long id);
}
```

6. Next we'll create a Vaadin Flow 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:

```
package io.pivotal;

import java.util.Collection;
import java.util.Collections;

import javax.annotation.PostConstruct;

import com.vaadin.flow.component.html.H2;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.Route;
import com.vaadin.flow.server.PWA;
import com.vaadin.flow.theme.Theme;
import com.vaadin.flow.theme.material.Material;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.Resources;
import org.vaadin.crudui.crud.impl.GridCrud;

import io.pivotal.domain.City;
import lombok.extern.slf4j.Slf4j;

@Slf4j
@Route(value = "")
@Theme(Material.class)
@PWA(name = "Cities UI, Vaadin Flow with Spring", shortName = "Cities UI")
public class CitiesUI extends VerticalLayout {

    private static final long serialVersionUID = 1L;

    private final CityClient client;
    private final GridCrud<City> crud;

    @Autowired
    public CitiesUI(CityClient client) {
        this.client = client;
        this.crud = new GridCrud<>(City.class);
    }

    @PostConstruct
    protected void init() {
        H2 title = new H2("Cities");
        crud.getGrid().setColumns("id", "name", "county", "stateCode",
"postalCode", "latitude", "longitude");
        crud.getCrudFormFactory().setVisibleProperties("name", "county",
"stateCode", "postalCode", "latitude", "longitude");
        crud.getCrudFormFactory().setUseBeanValidation(true);
        crud.setFindAllOperation(this::getCities);
        crud.setAddOperation(this::addCity);
        crud.setUpdateOperation(this::updateCity);
        crud.setDeleteOperation(this::deleteCity);
        add(title, crud);
        setSizeFull();
    }

    private Collection<City> getCities() {
        Resources<City> resources = client.findAll(0, 500);
        Collection<City> cities = Collections.emptyList();
        if (resources != null) {
            log.trace(resources.toString());
            cities = resources.getContent();
            log.debug("Fetched {} cities.", cities.size());
            if (!cities.isEmpty()) {
                crud.getGrid().setHeightByRows(true);
            }
        }
        return cities;
    }

    private City addCity(City city) {
        log.trace("City to be added is {}", city.toString());
        return client.add(city);
    }

    private City updateCity(City city) {
        log.trace("City to be updated is {}", city.toString());
        return client.update(city.getId(), city);
    }

    private void deleteCity(City city) {
        log.trace("City to be deleted", city.toString());
        client.delete(city.getId());
    }
}
```

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**:

```
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

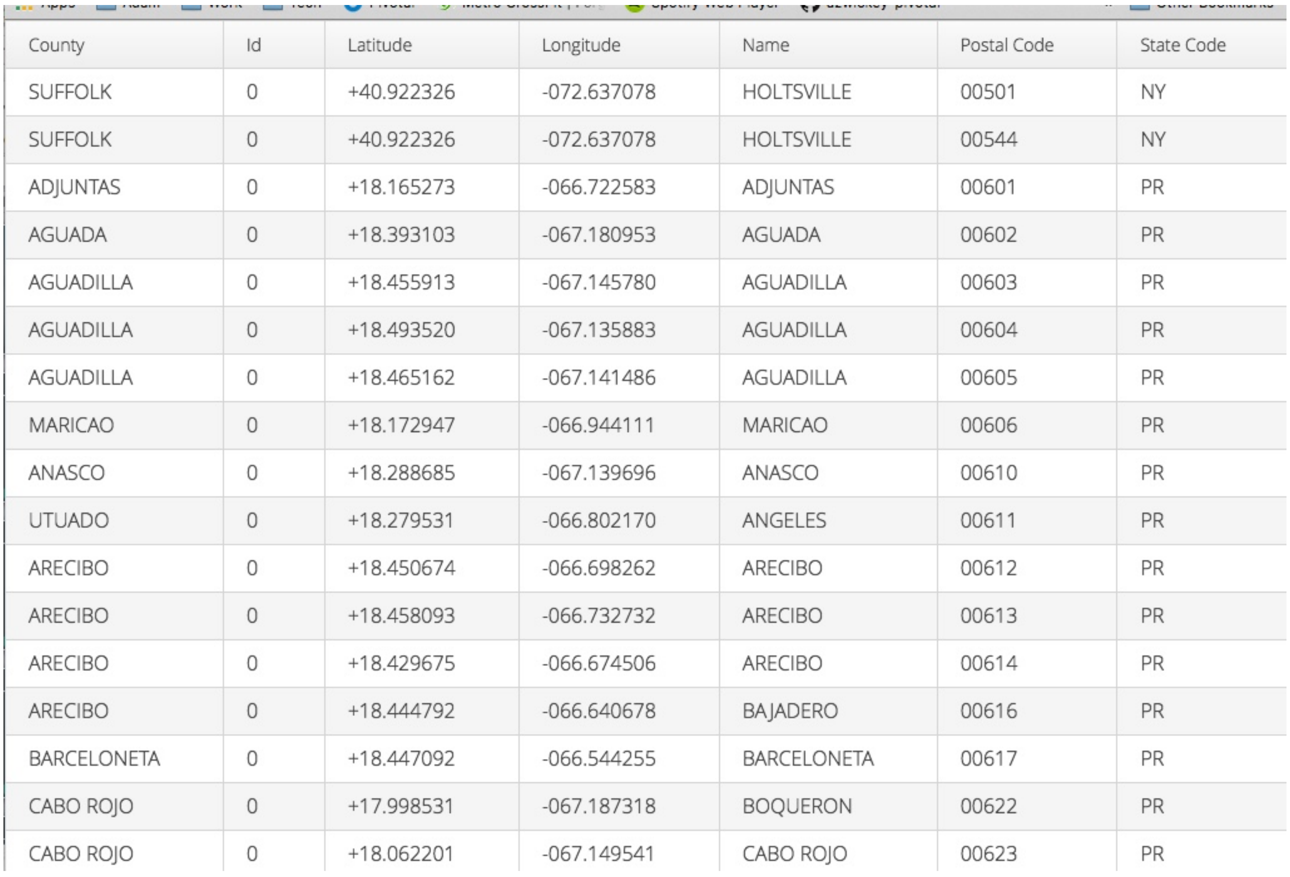
3. Add application metadata

```
---
applications:
- name: cloud-native-spring-ui
  memory: 1024M
  random-route: true
  instances: 1
  path: ./build/libs/cloud-native-spring-ui-1.0-SNAPSHOT.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 `/` endpoint, which will invoke the 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.