DevOps Workshop Lab Guide

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. 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

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

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

1. gradlew Project (Most developers will use Maven, but it's not critical)
2. Java
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/main/test*.

Now let's continue with the pre-made 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 (InteliJ is recommended).

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 ./gradlew 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 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. Check the application's metrics at http://localhost:8080/actuator
5. 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.jar
     # can be automatically detected, but it's nice to be clear about it:
     buildpacks:
     - java_buildpack_offline
     env:
       # makes Java a bit faster. Not mandatory.
       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.+ } }'
   ```

   It should be noted that many of these parameters could have been implied. For example, the following works just the same:

```
---
applications:
- name: cloud-native-spring
  random-route: true
  path: ./build/libs/cloud-native-spring-1.0-SNAPSHOT.jar
```

1. 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
```

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
```

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

Create a Dockerfile

Now that we have a running application, we need to containerize it.

We need to create a Dockerfile for our app. . What would be our base operating system? There are many choices. Ubuntu, RedHat, Suse, CentOS. . Who will be in charge of patching the OS? . What version of Java should we use? There are *many* choices. Search Dockerhub 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). . Select a base OS to use. . The jar file containing the applicaiton is under *build/libs/cloud-native-spring-1.0-SNAPSHOT.jar*. . The command to run the java application is

+

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

Write a Dockerfile for this 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
```

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.

create a new namespace for your team:

Creating a namespace

+

```
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:

+

```
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

+

```
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. . You can create an A record pointing to the load balancer, but then you will have to make sure the IP doesn't change. Developers rarely have access to DNS. . You can create an ingress gateway that routes to the deployment, which requires installing an additional solution such as nginx. . You can leverage Istio/KNative

Scaling

Change the deployment so there are 3 pods instead of 1.

Logging

Check the logs of one of the pods. How can you get the logs from all pods in one location? . You can use sidecar containers to manually handle logging to a central solution . You can install Fluentd daemon sets (requires privilege access to *kube_system* namespace) . You can use commercial solutions such as Splunk, Datadog, SumoLogic, Log Insight etc. (at an added cost) . You can use open source solutions such as ELK, Graylog (but it is your responsibility to maintain and upgrade them)

Monitoring

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

+

```
kubectl describe pod <my-pod>
```

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

Adding a database

Our application now needs a database. What will we install? . We can install a database directly as an image. . We can use helm charts. Who takes care of the upgrades? . We can use database solutions as 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 configure the following: . We need to make sure that *only* our application can access the database and no other pods. This requires Kubernetes *taint* commands using labels. . We need to update our application to point to the new database URL, username and password. . We need to do this every time we move to other environments (such as other namespaces or Kubernetes clusters). . We need to store password in a well-encrypted store. The default kubernetes secret management uses base64 encoding **which is not an encryption solution**

Installing a management GUI

It would be nice to install a web UI to manage our kubernetes deployments. . We can use the default Kubernetes dashboard. . We can install desktop solutions such as Lens. . Almost all of these solutions focus on the Kubernetes native components (pods, deployments, services) and not on the application-specific perspective (Apps, logs, metrics).

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

+ image::images/k8s1.jpg[]

+ image::images/k8s2.jpg[]

+ image::images/k8s3.jpg[]

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(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.

```
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 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:8.0.15')
```

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 {
       implementation('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 *plugins{}* section

   ```
   plugins {
       id 'com.gorylenko.gradle-git-properties' version '2.0.0'
   }
   ```

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 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"
       }
   }
   ```

3. 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:
    name: @application.name@
    description: @application.description@
    version: @application.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**:

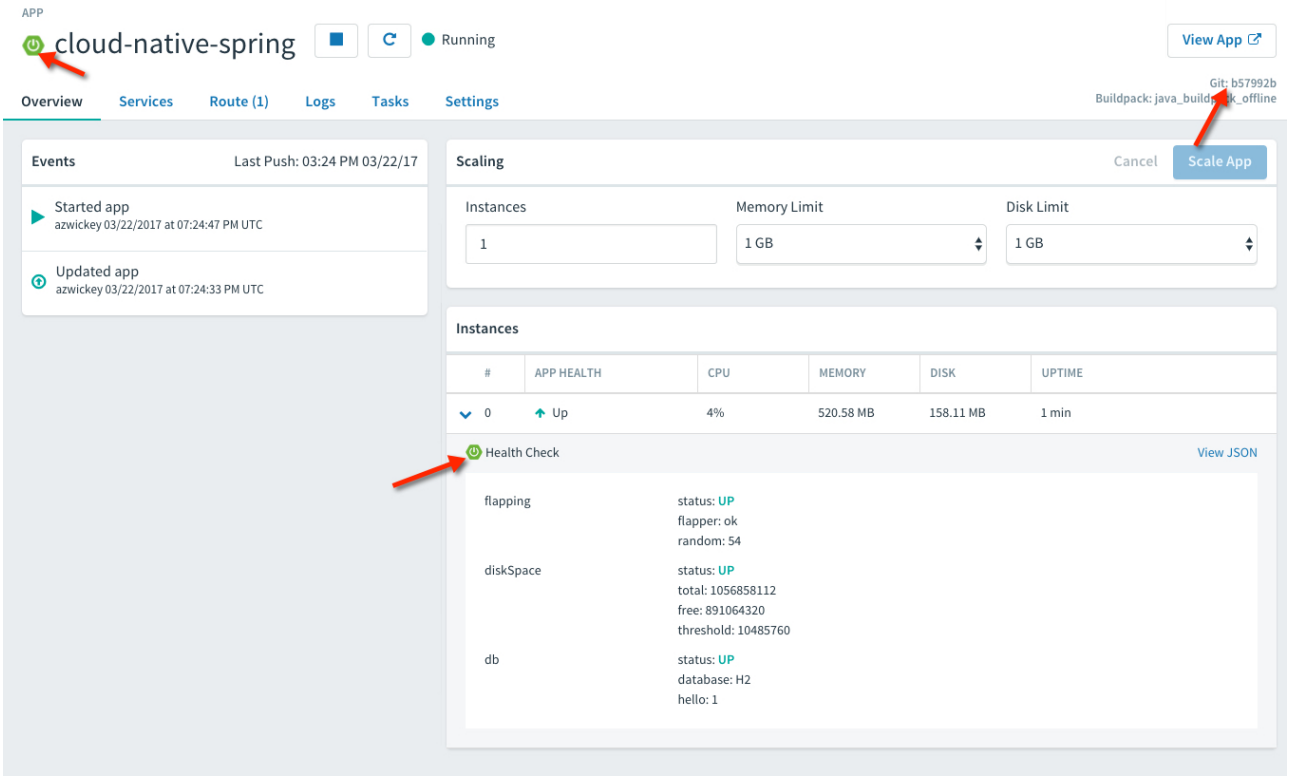
```
---
spring:
  profiles: cloud

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

2. Push application into Cloud Foundry

```
gradle build
cf push
```

3. 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:



4. 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.