Microservices with Spring

Lab Instructions

Building Cloud Native Applications using Spring and Pivotal Cloud Foundry

Version 1.0

spring Pivotal

Copyright Notice

This Page Intentionally Left Blank

This Page Intentionally Left Blank

Table of Contents

Chapter 1. Course Setup

The following software is required to complete this course:

- Java JDK 8
- An IDE or Editor of your choice
- Maven utility: `mvn`
- Git command line: `git`
- Cloud Foundry CLI `cf`
- Curl (Windows users only)
- Optional: Browser plugin to view JSON Data (Chrome or Firefox only)

Ideally this setup should be completed *before* you attend the course.

1.1. Note on Lab Usage

Much of this course involves using command line tools rather than GUI applications. This may be unfamiliar to some of you.

In particular, Windows users will need to open and use Command windows. To do this press `WK` + `R` and type `cmd` (where WK is the Windows Key).

To keep the labs simple, although we encourage you to use an IDE, we only use it for creating and editing files. In most cases we build using `mvn` directly. We accept that this is not typical but it makes the lab instructions simpler.

1.2. Java

A Java 8 JDK needs to be installed. Go to Oracle at http://www.oracle.com/technetwork/java/javase/downloads/index.html and click the Java download button on the right. You should be on the page to download the "Java SE Development Kit".

Click the radio-button to say you "Accept License Agreement" and then pick the right JDK for your platform. Make sure you are downloading the JDK *not* the JRE.

Once downloaded, run the installer. It takes a while to run.

1.3. IDE

We recommend installing the integrated development environment of your choice, such as Eclipse, Spring Tool Suite (STS) or IntelliJ. If you have no preference we recommend STS as it has additional support for the Spring facilities used during the course.

- STS: http://spring.io/tools The download is a zip file, so unpack it somewhere convenient. Windows users are recommended to unpack it directly into `C:\` otherwise you will have problems exceeding the maximum path length (yes even on Windows 8 and 10).
  Once unpacked you should find a directory similar to `sts_bundle/sts-X.X.X.RELEASE` or `sts_bundle/sts-X.X.X.RELEASE-eX.X.X` (the versions will differ). In here is the executable for STS (`STS.exe`, `STS.app` or `STS`). For convenience, you might like to create a shortcut on your Desktop.
- Eclipse: https://eclipse.org/downloads/ Either use the Installer or choose "Eclipse IDE for Java Developers ".
- IntelliJ: https://www.jetbrains.com/idea/download/ The Community Edition is free and will be sufficient for this course.

An IDE is not required to do this course, but if you prefer not to use an IDE, you will need to use your favorite editor instead. Windows users are recommended to download the free Notepad++s editor from https://notepad-plus-plus.org/download

1.4. Maven

Download Maven from https://maven.apache.org/download.cgi - select the binary zip archive (Windows) or the binary tar.gz archive (MacOS, Linux).

Once downloaded, unpack into a convenient directory. You need to add the `bin` directory to your execution path.

- Windows - Right click `My Computer`/`This PC` # `Properties` # `Advanced system settings` # `Environment variables` …`PATH` should already exist, click `Edit` and add to the end. Don't forget to add a semicolon first. For more details on how to do this see http://www.computerhope.com/issues/ch000549.htm.
- MacOS, Linux - Assuming you are using a BASH shell, edit `~/.bashrc` and add:

```
PATH=$PATH:/path/to/maven/bin
```


You will need to close any existing Terminal/Cmd windows and open a new one for the change to take effect.

1.5. Git Command-Line Utility

Go to https://git-scm.com/downloads, click the image corresponding to your platform, download the installer and run it.

Warning: Ignore the large computer monitor labelled "Latest source Release".

- Windows users: When running the installer, select "*Use Git from the Windows Command Prompt*" (this is not the default). Accept all other defaults.

1.6. Cloud Foundry Command-Line Utility

Go to https://github.com/cloudfoundry/cli/releases, download and run the right installer for your platform.

1.7. Curl for Windows

Linux and MacOS users already have the `curl` utility. Windows users can download it from here: https://github.com/S2EDU/PCFImmersionStudentFiles/raw/master/curl.zip.

Once downloaded, unpack it somewhere convenient and add to your path - the same procedure as for Maven.

1.8. Optional Add-on - View JSON Data

Several labs use `curl` to send requests that return JSON data. If you prefer to do this in your browser …

1.8.1. Firefox or Chrome

1. Do an Internet search for the "JSONView" add-on.
2. Install as directed
3. Once installed, try this JSON test page: http://jsonplaceholder.typicode.com/posts/1

1.8.2. Internet Explorer

1. Create a file on your desktop called `json.reg`.
2. Right click and select edit.
3. Copy and paste the following:

```
Windows Registry Editor Version 5.00;
; Tell IE 7,8,9,10,11 to open JSON documents in the browser on Windows XP and later.
; 25336920-03F9-11cf-8FD0-00AA00686F13 is the CLSID for the "Browse in place" .
;
[HKEY_CLASSES_ROOT\MIME\Database\Content Type\application/json]
"CLSID"="{25336920-03F9-11cf-8FD0-00AA00686F13}"
"Encoding"=hex:08,00,00,00
```

4. Save and close.
5. Double click on the `json.reg` to run it. You will get a warning, click `Yes` to continue.
6. Now try this JSON test page: http://jsonplaceholder.typicode.com/posts/1
7. Delete `json.reg`

1.9. Testing Your Setup

If everything has installed correctly, *open a brand new* Terminal/Cmd window. You should be able to run the following:

```
java -version
mvn -version
git --version
cf --version
curl --version
```


Chapter 2. Lab 1a - Getting Started with Spring Boot

Estimated time to complete: 20 minutes

2.1. Generate a project from start.spring.io

1. In your browser, visit https://start.spring.io.
2. Fill out the **Project metadata** fields as follows:

   Group

   ```
   io.spring.pivotal
   ```

   Artifact

   ```
   hello-spring-boot
   ```

   Search For Dependencies

   ```
   Web
   ```

   Spring Boot version

   ```
   1.3.5
   ```

3. Click the **Generate Project** button. Your browser will download a zip file. Unpack that zip file at `$COURSE_HOME/lab_01/lab_01a/initial`.

2.2. Run it locally

1. Import the project's `pom.xml` into your editor/IDE of choice.
2. Add a `@RestController` annotation to the class `io.pivotal.spring.hello.HelloSpringBootApplication` to indicate that we wish to return REST (data) responses not HTML.
   It will be marked in red, so you need to add the import for this annotation.

Note

You IDE can do this for you using the "[big red]*O*rganise imports" option. In Eclipse/STS

press CTRL-SHIFT-O (MacOS: COMMAND-SHIFT-O). In IntelliJ use CTRL+ALT+o (MacOS: COMMAND-ALT-o)

1. Add the following request handler method to the class `io.pivotal.spring.hello.HelloSpringBootApplication` after the static `main` method:

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

   Again you will need to use the IDE to import the `@RequestMapping` annotation.

2. To run the application:

   - Eclipse/STS: Right click in the editor window and select `Run As` # `Spring Boot App`.
   - IntelliJ: In the editor window press CTRL-SHIFT-R (MacOS: COMMAND-SHIFT-R)
     Wait for the application to start up.

3. Open http://localhost:8080 in your browser and it should return `Hello World!`.

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

Chapter 3. Lab 1b - Deploying a Web Application with an Embedded Container

Estimated time to complete: 15 minutes

3.1. Build and Run with Embedded Apache Tomcat

Spring Boot embeds `Apache Tomcat` by default.

Note

For this lab, you should keep working with the project that you have created in Lab 01a

1. Run the application by launching the `HelloSpringBootApplication` class from your IDE
   You should see the application start up an embedded Apache Tomcat server on port 8080:

```
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
i.p.s.hello.HelloSpringBootApplication   : Started HelloSpringBootApplication in 3.023 seconds (JVM running for 3.432)
```

2. Visit the application in the browser (http://localhost:8080), and you should see the following:

```
Hello World!
```

3.2. Hot reload with Spring Boot Devtools

One of the coolest features of Spring Boot is that most of your code can hot redeploy (you do not have to restart your application for changes to be taken into account).

1. Open your `pom.xml` file and add the following dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

2. Restart your application once

3. Inside `HelloSpringBootApplication`, replace the String "Hello World" with "Hello Wooooorld" and save the file. You will see that the application redeploys the changes automatically. You just have to go to your browser and refresh so you can see the new value.

Note

IntelliJ users, the editor auto saves your changes, however devtools only monitors the compiled `.class` files, so please use `Build` # `Make Project`

Note

You still need to wait a few seconds for Spring Boot to refresh your application but it takes 20-30% of the time that it would take to restart your application completely.

3.3. Build and Run with Embedded Eclipse Jetty

Spring Boot also supports embedding a Jetty server.

1. Open `pom.xml` and replace the following:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

   with:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

2. Build and run the application

   You can run it within your Development environment or from the command line (same as earlier in this lab).

   You should see the application start up an embedded Jetty server on port 8080:

```
s.b.c.e.j.JettyEmbeddedServletContainer : Jetty started on port(s) 8080 (http/1.1)
i.p.s.hello.HelloSpringBootApplication  : Started HelloSpringBootApplication in 3.671 seconds (JVM running for 4.079)
```

3. Visit the application in the browser (http://localhost:8080), and you should see the following:

```
Hello World!
```


Note

Hot redeploy works with Jetty in the same way as it works for Tomcat

Chapter 4. Lab 1c - Externalizing Configuration with Spring Boot

Estimated time to complete: 20 minutes

4.1. Refactoring to Externalize the Config

1. You should work using the same project as in the previous labs
2. Inside the folder `src/main/resources/`, you will see a file called `application.properties`. Rename that file into `application.yml`

Note

Spring Boot allows to work with properties files or YML files indifferently. In this workshop you will be using YML files because their syntax is more powerful (and still fairly simple!)

1. Inside `application.yml` paste the following:

```
greeting: Hello
```

2. To the class `io.pivotal.spring.hello.HelloSpringBootApplication`, add a `greeting` field and inject its value:

```
@Value("${greeting}")
String greeting = "Bonjour";
```

3. Also `io.pivotal.spring.hello.HelloSpringBootApplication`, change the return statement of `hello()` to the following:

```
return String.format("%s World!", greeting);
```

4. Make sure your application is running and visit http://localhost:8080 in your browser. Which of the values is picked up? Is it the local one ("Bonjour") or the one in `application.yml` ("Hello")?

4.2. Using Environment Variables for Config

1. You should now stop the application inside your IDE because you are going to launch it from the command line

2. You are now going to add an environment variable from the command line. You should compile and run the application as shown below:

Windows.

```
> mvn clean package
> set GREETING=Ohai && java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

Linux, MacOS:

```
$ mvn clean package
$ GREETING=Ohai java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

Note

in case of any issue running your application, you should confirm that your jar file is the same as in the above code (it varies depending on the name of our project)

1. Visit the application in the browser (http://localhost:8080), and verify that the output has changed to the following:

```
Ohai World!
```

2. Stop the application.

3. Visit http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html to learn more about this outcome and the entire priority scheme for conflict resolution.

Chapter 5. Lab 1d - Introspection, Monitoring, and Metrics using Spring Boot Actuator

Estimated time to complete: 20 minutes

5.1. Set up the Actuator

1. You should use the same project as in the previous labs.

2. To `pom.xml` add the following dependency to include the starter for Spring Boot Actuator:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

3. Run the application from your IDE

4. open this URL in your browser: http://localhost:8080/beans (use right-click `Open in New Tab`). You can see runtime information about your `Spring` beans. However it is hard to read.

5.2. Adding proper JSON formatting

1. Add the following configuration *at the end* of the file `HelloSpringBootApplication.java`:
   (you are allowed to have more than one classes in the same file as long as only one of them is `public`)

```
@Configuration
class RestWsApplicationConfig {  // Note that this class is NOT public

    @Bean
    public Jackson2ObjectMapperBuilder jacksonBuilder() {
        Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder();
        builder.indentOutput(true);
        return builder;
    }
}
```

2. Run the application again and connect to http://localhost:8080/beans

5.3. Introspection Endpoints

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

http://localhost:8080/beans

Dumps all of the beans in the Spring context.

http://localhost:8080/autoconfig

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

http://localhost:8080/env

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

http://localhost:8080/mappings

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

http://localhost:8080/dump

Performs a thread dump.

http://localhost:8080/trace

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

5.4. Health Indicators

Spring Boot provides an endpoint (http://localhost:8080/health) that allows for the notion of various health indicators. Try it now.

1. In production it would be normal to enable Spring Security so that the `/health` endpoint will only expose an `UP` or `DOWN` value unless you are an authorized user. However, to simplify working with the endpoint for this lab, we will leave security off (the default behavior).
   You can experiment with this now by modifying the `sensitivity` property which is `false` by default. Add the following to `src/main/resources/application.yml` and refresh the health page:

```
endpoints:
  health:
    sensitive: true
```

   Make sure to set the property back to `false` before continuing.

2. Create a *new* class `io.pivotal.spring.FlappingHealthIndicator` (in its *own* file) and into it paste the following code:

```
@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.

3. Run the application

4. Visit the application in the browser (http://localhost:8080/health), and verify that the output is similar to the following (and changes randomly!):

```
{
  "status": "UP",
  "flapping": {
    "status": "UP",
    "flapper": "ok",
    "random": 69
  },
  "diskSpace": {
    "status": "UP",
    "free": 113632186368,
    "threshold": 10485760
  }
}
```

To learn more about the autogenerated metrics, visit http://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html.

Chapter 6. Lab 2a - From Zero to Pushing Your First Application

6.1. Setup

1. Set the API target for the CLI:

```
$ cf api {CF-API-URL} --skip-ssl-validation
```

2. Your username will be **workshop.user**. Password will be provided in the class.

3. Follow the steps above for login.

4. Create your own space and target that space:

```
$ cf create-space YOURNAME
$ cf target -s YOURNAME
```

6.2. Build and Push!

1. Change to the *Spring Music* sample application directory:

```
$ cd $COURSE_HOME/lab/lab_02/lab_02a/spring-music
```

2. Using the Gradle Wrapper, build and package the application:

```
$ ./gradlew assemble
```

   The Gradle Wrapper will automatically download the appropriate version of Gradle for this project along with all of *Spring Music*'s dependencies. This may take a few moments.

3. Push the application!

```
$ cf push
```

   You should see output similar to the following listing. Take a look at the listing callouts for a play-by-play of what's happening:

```
Using manifest file .../lab_02/lab_02a/spring-music/manifest.yml
❶

Creating app spring-music in org oreilly-class / space instructor as example@pivotal.io...
OK
❷

Creating route spring-music-hippest-shaman.cfapps.io...
OK
```



```
❸

Binding spring-music-hippest-shaman.cfapps.io to spring-music...
OK
❶

Uploading spring-music...
❸
Uploading app files from: .../lab_02/lab_02a/spring-music/build/libs/spring-music.war
Uploading 569.7K, 90 files
Done uploading
OK

Starting app spring-music in org oreilly-class / space instructor as example@pivotal.io...
❹
-----> Downloaded app package (21M)
-----> Java Buildpack Version: v2.6.1 |  https://github.com/cloudfoundry/java-buildpack.git#2d92e70
-----> Downloading Open Jdk JRE 1.8.0_31 from https://download.run.pivotal.io/openjdk/lucid/x86_64/openjdk
    -1.8.0_31.tar.gz (1.3s)
       Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.2s)
❼
-----> Downloading Spring Auto Reconfiguration 1.7.0_RELEASE from https://download.run.pivotal.io/auto-reconfiguration
    /auto-reconfiguration-1.7.0_RELEASE.jar (0.1s)
-----> Downloading Tomcat Instance 8.0.18 from https://download.run.pivotal.io/tomcat/tomcat-8.0.18.tar.gz (0.4s)
       Expanding Tomcat to .java-buildpack/tomcat (0.1s)
❺
-----> Downloading Tomcat Lifecycle Support 2.4.0_RELEASE from https://download.run.pivotal.io/tomcat-lifecycle-support
    /tomcat-lifecycle-support-2.4.0_RELEASE.jar (0.0s)
-----> Downloading Tomcat Logging Support 2.4.0_RELEASE from https://download.run.pivotal.io/tomcat-logging-support
    /tomcat-logging-support-2.4.0_RELEASE.jar (0.0s)
-----> Downloading Tomcat Access Logging Support 2.4.0_RELEASE from https://download.run.pivotal.io/tomcat-access
    -logging-support/tomcat-access-logging-support-2.4.0_RELEASE.jar (0.0s)
-----> Uploading droplet (66M)
❷

0 of 1 instances running, 1 starting
1 of 1 instances running

App started


OK

App spring-music was started using this command `JAVA_HOME=$PWD/.java-buildpack/open_jdk_jre JAVA_OPTS="
    -Djava.io.tmpdir=$TMPDIR
-XX:OnOutOfMemoryError=$PWD/.java-buildpack/open_jdk_jre/bin/killjava.sh -Xmx382293K -Xms382293K -XX:MaxMetaspaceSize=64
    M
-XX:MetaspaceSize=64M -Xss995K -Daccess.logging.enabled=false -Dhttp.port=$PORT" $PWD/.java-buildpack/tomcat/bin
    /catalina.sh run`
❿

Showing health and status for app spring-music in org oreilly-class / space instructor as example@pivotal.io...

OK

requested state: started
instances: 1/1
usage: 512M x 1 instances
urls: spring-music-hippest-shaman.cfapps.io
last uploaded: Fri Feb 13 15:43:08 UTC 2015

     state     since                    cpu    memory           disk
#0   running   2015-02-13 09:43:55 AM   0.0%   394.5M of 512M   131.1M of 1G
```

❶ The CLI is using a manifest to provide necessary configuration details such as application name, memory to be allocated, and path to the application artifact. Take a look at `manifest.yml` to see how.

❷ In most cases, the CLI indicates each Cloud Foundry API call as it happens. In this case, the CLI has created an application record for *Spring Music* in your assigned space.

❸ All HTTP/HTTPS requests to applications will flow through Cloud Foundry's front-end router called (Go)Router. Here the CLI is creating a route with random word tokens inserted (again, see `manifest.yml` for a hint!) to prevent route collisions across the default `cfapps.io` domain.

❹ Now the CLI is *binding* the created route to the application. Routes can actually be bound to multiple

applications to support techniques such as blue-green deployments.

❺ The CLI finally uploads the application bits to PWS. Notice that it's uploading *90 files*! This is because Cloud Foundry actually explodes a ZIP artifact before uploading it for caching purposes.

❻ Now we begin the staging process. The Java Buildpack is responsible for assembling the runtime components necessary to run the application.

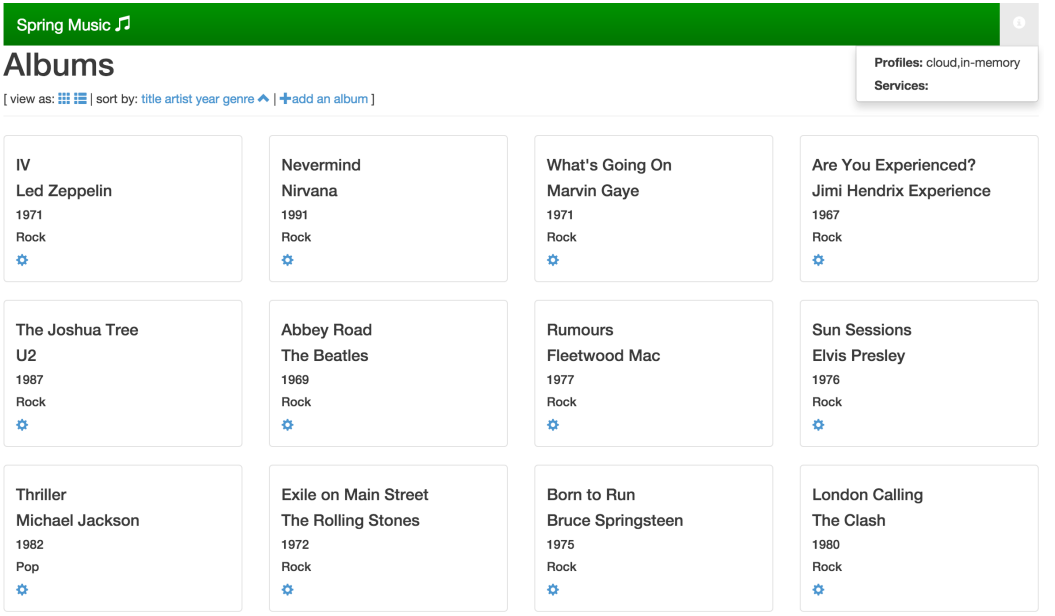
❼ Here we see the version of the JRE that has been chosen and installed.

❽ And here we see the version of Tomcat that has been chosen and installed.

❾ The complete package of your application and all of its necessary runtime components is called a *droplet*. Here the droplet is being uploaded to PWS's internal blobstore so that it can be easily copied to one or more Execution Agents (dedicated VMs that run application instances).

❿ The CLI tells you exactly what command and argument set was used to start your application.
Finally the CLI reports the current status of your application's health. You can get the same output at any time by typing `cf app spring-music`.

4. Visit the application in your browser using the route that was generated by the CLI:



Be sure to click on the (i) *information icon* in the top right-hand corner of the UI. The drop-down gives you important information about the state of the currently running *Spring Music* instance, including what Spring Profiles are turned on and what Cloud Foundry services are bound. It will be important in the next lab!

Chapter 7. Lab 2b - Binding to Cloud Foundry Services

The *Spring Music* application was designed to illustrate the ease with which various types of data services can be bound to and utilized by Spring applications running on Cloud Foundry. In this lab, we'll be binding the application to both MySql and MongoDB databases.

Cloud Foundry services are managed through two primary types of operations:

Create/Delete

: These operations create or delete instances of a service. For a database this could mean creating/deleting a schema in an existing multitenant cluster or creating/deleting a dedicated database cluster.

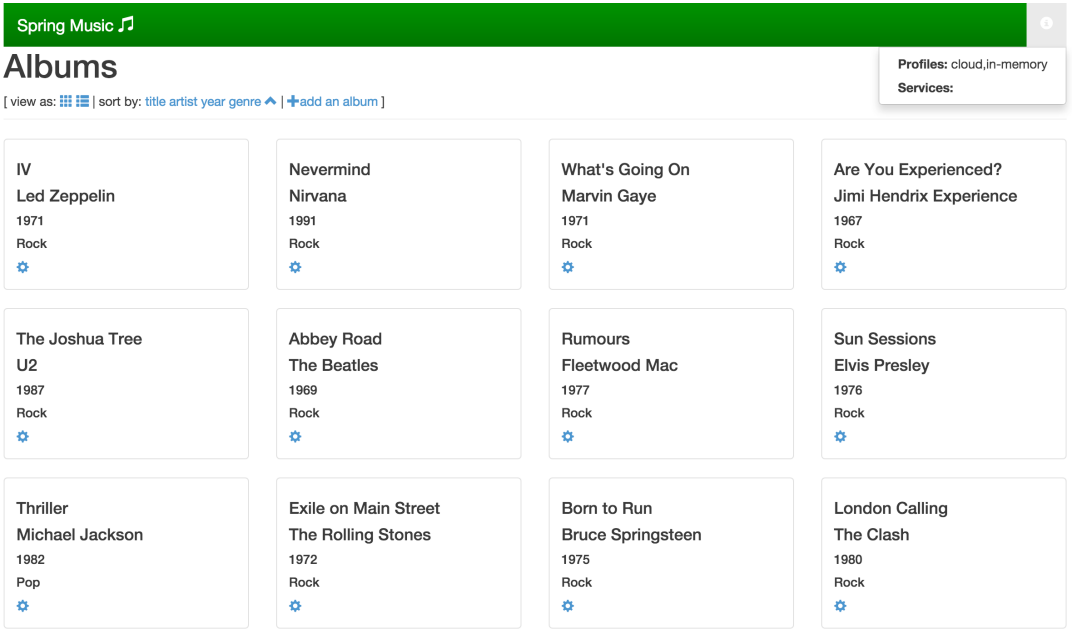
Bind/Unbind

: These operations create or delete unique credential sets for an existing service instance that can then be injected into the environment of an application instance.

7.1. A Bit of Review

Your instance of *Spring Music* should still be running from the end of the previous lab.

Visit the application in your browser by hitting the route that was generated by the CLI:



The information dialog in the top right-hand corner indicates that we're currently running with an in-memory database, and that we're not bound to any services. Let's change that.

7.2. The Services Marketplace

There are two ways to discover what services are available when using Pivotal Cloud Foundry (PCF):

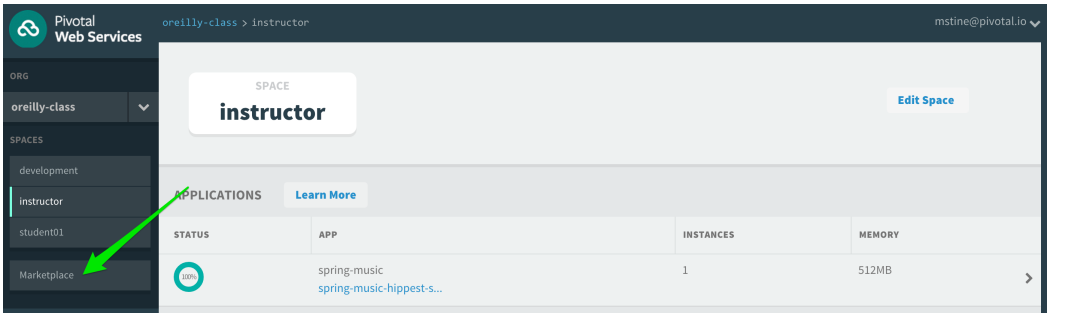
1. The first is available on *any* instance of Cloud Foundry (Open Source or PCF) via the `cf` CLI. Type:

```
$ cf marketplace
```

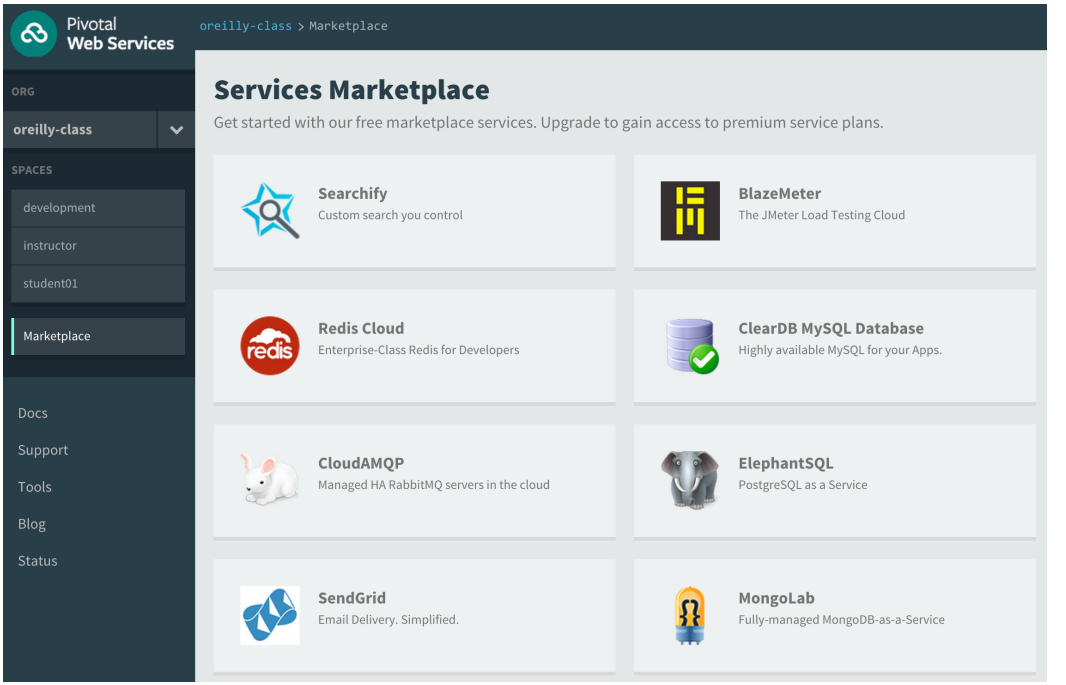
   and you'll get a list of services, their available plans, and descriptions (the services list may vary from what we show here):

```
instructor$ cf marketplace
Getting services from marketplace in org pivotaledu / space development as pchapman@gopivotal.com...
OK

service          plans                                                  description
3scale           free_appdirect, basic_appdirect*, pro_appdirect*       API Management Platform
app-autoscaler   bronze, gold                                           Scales bound applications in response to load
   (beta)
blazemeter       free-tier, basic1kmr*, pro5kmr*                        Performance Testing Platform
cleardb          spark, boost*, amp*, shock*                            Highly available MySQL for your Apps.
...
```


2. The second way is specific to PCF's Application Manager UI. If using Pivotal Web Services (PWS), this is what you see once you login to http://run.pivotal.io. If you haven't already, login now.
   Click on the "Marketplace" link (on the left hand side panel):*The Screenshots below are from PWS (public managed Pivotal CF). If you are using your own installation of PCF, you will see something similar.*



and you'll see the same service/plan/description listing in the browser:



7.3. Creating and Binding to a Service Instance

1. Let's begin by creating a MySql instance provided by Pivotal.
   From the CLI, let's *create* a `p-mysql` service instance:

```
$ cf create-service p-mysql 100mb-dev spring-music-db
Creating service spring-music-db in org CNA / space instructor as example@pivotal.io...
OK
```

   If `p-mysql` is not an available service, use `cleardb` instead (select the free `spark` plan: `cf create-service cleardb spark spring-music-db`).

2. Next we'll *bind* the newly created instance to our `spring-music` application:

```
$ cf bs spring-music spring-music-db
Binding service spring-music-db to app spring-music in org ACME / space instructor as example@pivotal.io...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect
```

3. Notice the admonition to `Use 'cf restage' to ensure your env variable changes take effect`. Let's take a look at the environment variables for our application to see what's been done. We can do this by typing:

```
$ cf env spring-music
```

   The subset of the output we're interested in is located near the very top, titled `System-Provided`:

```
System-Provided:
{
 "VCAP_SERVICES": {
❶
  "p-mysql": [
   {
    "credentials": {
     "hostname": "10.68.50.61",
     "jdbcUrl": "jdbc:mysql://10.68.50.61:3306/cf_ddbe0221_d1fc_478f_b693_69279b2de702?user=mWsYAON34zbaQPAA
    \u0026password=CZImCZ9iSNU7HfP2",
     "name": "cf_ddbe0221_d1fc_478f_b693_69279b2de702",
     "password": "CZImCZ9iSNU7HfP2",
     "port": 3306,
     "uri": "mysql://mWsYAON34zbaQPAA:CZImCZ9iSNU7HfP2@10.68.50.61:3306/cf_ddbe0221_d1fc_478f_b693_69279b2de702?reconnect=true",

❷
     "username": "mWsYAON34zbaQPAA"
    },
    "label": "p-mysql",
    "name": "spring-music-db",
    "plan": "100mb-dev",
    "tags": [
     "mysql",
     "relational"
    ]
   }
  ]
 }
}

{
 "VCAP_APPLICATION": {
  "application_name": "spring-music",
  "application_uris": [
   "spring-music-unbragging-pourparler.pcf2.fe.gopivotal.com"
  ],
  "application_version": "e3fa423d-69ad-41cb-bd17-0e88427dfe4c",
  "limits": {
   "disk": 1024,
   "fds": 16384,
   "mem": 512
```



```
  },
  "name": "spring-music",
  "space_id": "080ba4e6-bafc-482b-9a98-42f612eef736",
  "space_name": "jfullam",
  "uris": [
   "spring-music-unbragging-pourparler.pcf2.fe.gopivotal.com"
  ],
  "users": null,
  "version": "e3fa423d-69ad-41cb-bd17-0e88427dfe4c"
 }
}
```

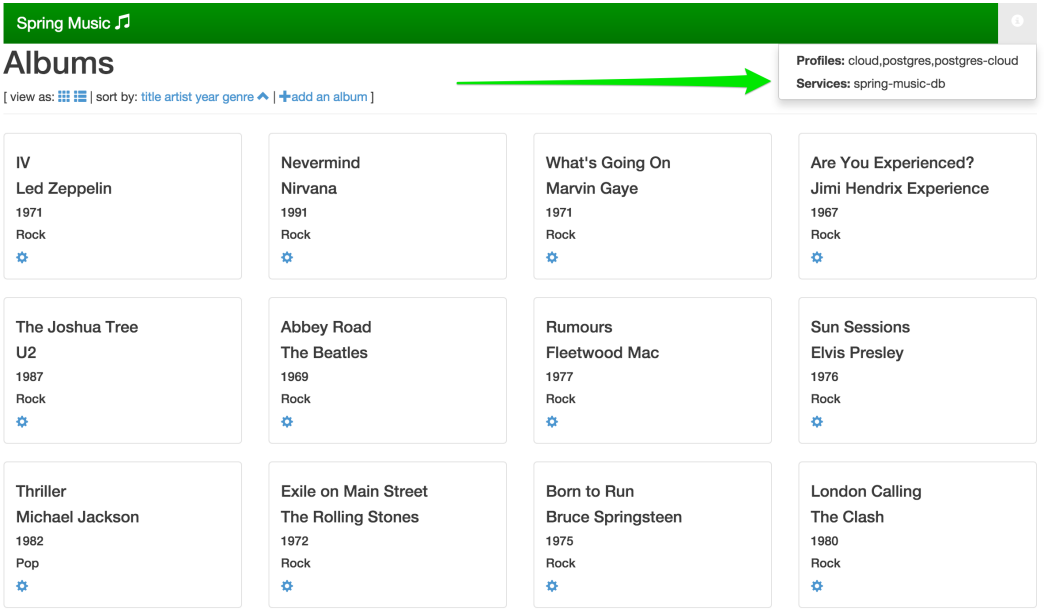
❶ `VCAP_SERVICES` is a special Cloud Foundry environment variable that contains a JSON document containing all of the information for any services bound to an application.

❷ Notice here the unique URI for this instance of MySql that `spring-music` has been bound to.

4. Now let's *restage* the application, which cycles our application back through the staging/buildpack process before redeploying the application.[1]

```
$ cf restage spring-music
```

Once the application is running again, revisit or refresh the browser tab where you have the *Spring Music* application loaded:



[1] In this case, we could accomplish the same goal by only *restarting* the application via `cf restart spring-music`. A *restage* is generally recommended because Cloud Foundry buildpacks also have access to injected environment variables and can install or configure things differently based on their values.

Click on the `(i)` symbol at top-right and it will drop down the information dialog. You should be able to see that the application is now utilizing a MySql database via the `spring-music-db` service.
Note: If the application hasn't picked up the `spring-music-db` service, try doing a complete `cf push` again.

5. **(OPTIONAL STEPS)** If you have a MySql GUI tool handy and you are using a lab environment that has the necessary network access (ask your instructor), you can inspect the music database created. Otherwise, your instructor will demo via a Pivotal VPN connection.

6. In your DB tool, create a new server connection and populate the properties with values from the URI in your `VCAP_SERVICES` environment variable (remember `cf env spring-muisc`!):

7.4. Optional: Swapping from MySql to MongoDB

Only do this next section if you have sufficient time.

1. Now let's bind our *Spring Music* application to MongoDB instead of MySql. First let's create [2] a MongoDB instance from the Pivotal Cloud Foundry marketplace:

```
$ cf cs p-mongodb development spring-music-mongo
Creating service spring-music-mongo in org ACME / space instructor as example@pivotal.io...
OK
```

   If you do not have the `p-mongodb` service available use `mongolab` and choose the `sandbox` plan: `cf cs mongolab sandbox spring-music-mongo`.

2. Next we'll unbind our application from our MySql instance (*Spring Music* does not support being bound to multiple datasources at the same time):

```
$ cf us spring-music spring-music-db
```

   If you visit your application now, you'll see that it still works. If you recall, environment variable changes (such as binding/unbinding of services) don't actually take effect until a *restage* or *restart*.

3. Now let's bind the application to our MongoDB instance:

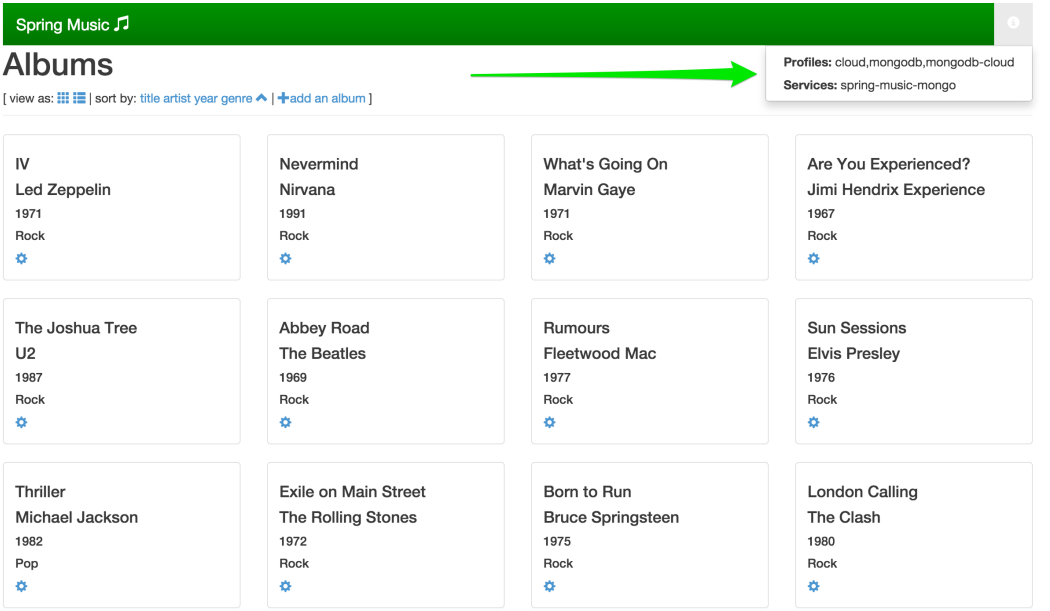
```
$ cf bs spring-music spring-music-mongo
Binding service spring-music-mongo to app spring-music in org oreilly-class / space instructor as example@pivotal.io...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect
```

4. And then do a restage:

```
$ cf restage spring-music
```

   Once the application is running again, revisit or refresh the browser tab where you have the *Spring Music* application loaded:

[2] Notice in this listing that we're typing `cf cs` rather than `cf create-service`. Most CF CLI commands have a shorthand version to save typing time. You can view these shorthand commands via `cf help` or `cf h` (See! More shorthand!).



As you can see from the information dialog, the application is now utilizing a MongoDB database via the `spring-music-mongo` service.

5. **(OPTIONAL)** Similar to the steps to view the relational data in another DB tool, a MongoDB client tool can be used to inspect the MongoDB instance. Your instructor can demo this if you do not have access to the tool and / or the appropriate network access to the MongoDB instance.

7.5. Clean Up

Because of the limited PWS quota we have for this course, let's clean up our application and services to make room for future labs.

1. Delete the `spring-music` application:

```
$ cf d spring-music

Really delete the app spring-music?> y
Deleting app spring-music in org oreilly-class / space instructor as example@pivotal.io...
OK
```

2. If you did the optional section using MongoDB, delete the `spring-music-mongo` service:

```
$ cf ds spring-music-mongo
```



```
Really delete the service spring-music-mongo?> y
Deleting service spring-music-mongo in org oreilly-class / space instructor as example@pivotal.io...
OK
```

3. Delete the `spring-music-db` service:

```
$ cf ds spring-music-db

Really delete the service spring-music-db?> y
Deleting service spring-music-db in org oreilly-class / space instructor as example@pivotal.io...
OK
```


Chapter 8. Lab 2c - Scaling Applications

Cloud Foundry makes the work of horizontally scaling application instances and updating load balancer routing tables easy.

In this lab, we'll use a Spring Boot CLI app designed to illustrate Cloud Foundry operations such as scaling.

8.1. Push the `cf-scale-boot` Application

1. Change to the `cf-scale-boot` sample application directory:

```
$ cd $COURSE_HOME/lab/lab_02/lab_02c/cf-scale-boot
```

2. Spring Boot CLI applications do not require a separate build step, so go ahead and push the application:

```
$ mvn package && cf push
```

3. Once again, this application's manifest is configured to have a random route assigned to the application. So, when the CLI indicates that application is up and running, visit its route in the browser:

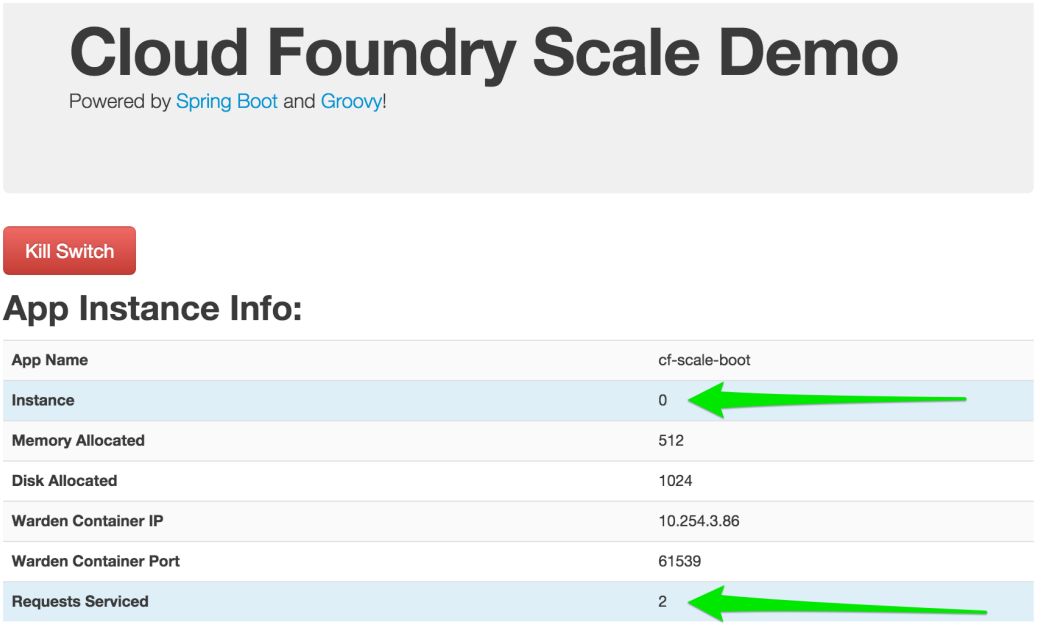


Figure 8.1. The application at initial deployment

You'll see that the application is reporting various bits of information that it has discovered from its environment. Of primary interest is that this application reports its *instance index* - note that this is the instance id *not* the number of instances running. It also keeps track of and reports how many web requests that this instance has serviced.

8.2. Scale the Application Up

1. Now let's increase the number of running application instances to 4 (make sure you have stopped `spring-music` before doing this or you may exceed your quota):

```
$ cf scale -i 4 cf-scale-boot
Scaling app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
OK
```

   In reporting `OK`, the CLI is letting you know that the additional requested instances have been started, but they are not yet necessarily running.

2. We can determine how many instances are actually running like this:

```
$ cf app cf-scale-boot
Showing health and status for app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
OK

requested state: started
instances: 4/4
usage: 512M x 4 instances
urls: cf-scale-boot-stockinged-rust.cfapps.io
last uploaded: Fri Feb 13 18:56:29 UTC 2015

     state      since                  cpu    memory           disk
#0   running    2015-02-13 12:57:10 PM   0.1%   404.8M of 512M   128.9M of 1G
❶
#1   starting   2015-02-13 03:04:33 PM   0.0%   0 of 0           0 of 0
❷
#2   running    2015-02-13 03:04:47 PM   0.0%   398.7M of 512M   128.9M of 1G
#3   starting   2015-02-13 03:04:33 PM   0.0%   0 of 0           0 of 0
```

❶ This application instance has completed the startup process and is actually able to accept requests.
❷ This application instance is still starting and will not have any requests routed to it.

3. Eventually all instances will converge to a running state:

```
$ cf app cf-scale-boot
Showing health and status for app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
OK

requested state: started
instances: 4/4
usage: 512M x 4 instances
urls: cf-scale-boot-stockinged-rust.cfapps.io
last uploaded: Fri Feb 13 18:56:29 UTC 2015

     state     since                  cpu    memory           disk
#0   running   2015-02-13 12:57:10 PM   0.1%   404.8M of 512M   128.9M of 1G
```



```
#1   running   2015-02-13 03:04:51 PM   0.1%   377.5M of 512M   128.9M of 1G
#2   running   2015-02-13 03:04:47 PM   0.1%   397.3M of 512M   128.9M of 1G
#3   running   2015-02-13 03:05:03 PM   0.0%   389.2M of 512M   128.9M of 1G
```

4. Revisit the application route in the browser. Refresh several times. You should observe the instance index and request counters changing as you do so - something like this (you instance and request numbers will differ):

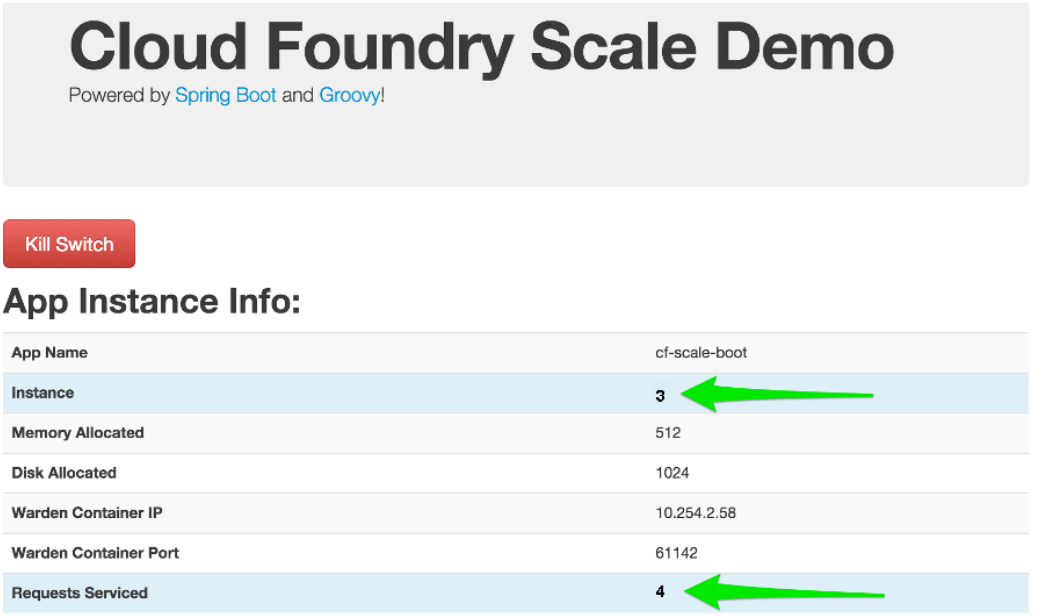


Figure 8.2. The application after scaling up

If scaling has worked correctly, you should see instance ids between 0 and 3.

8.2.1. About the Router

The aforementioned (Go)Router is applying a random routing algorithm to all of the application instances assigned to this route. As an instance reaches the `running` state, its Execution Agent (also known as a Cell) registers that instance in the routing table assigned to its route by sending a message to Cloud Foundry's message bus. All (Go)Router instances are subscribed to this channel and register the routes independently. This makes for very dynamic and rapid reconfiguration!

8.3. Scale the Application Down

1. We can scale the application instances back down as easily as we scaled them up, using the same command structure:

```
$ cf scale -i 1 cf-scale-boot
Scaling app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
OK
```

2. Check the application status again:

```
$ cf app cf-scale-boot
Showing health and status for app cf-scale-boot in org oreilly-class / space instructor
as example@pivotal.io...
OK

requested state: started
instances: 1/1
usage: 512M x 1 instances
urls: cf-scale-boot-stockinged-rust.cfapps.io
last uploaded: Fri Feb 13 18:56:29 UTC 2015

     state     since                  cpu    memory         disk
#0   running   2015-02-13 12:57:10 PM   0.1%   405M of 512M   128.9M of 1G
```

   As you can see, we're back down to only one instance running, and it is in fact the original index 0 that we started with.

3. Confirm that by again revisiting the route in the browser and checking the instance index and request counter. The instance id should now *always* be zero:

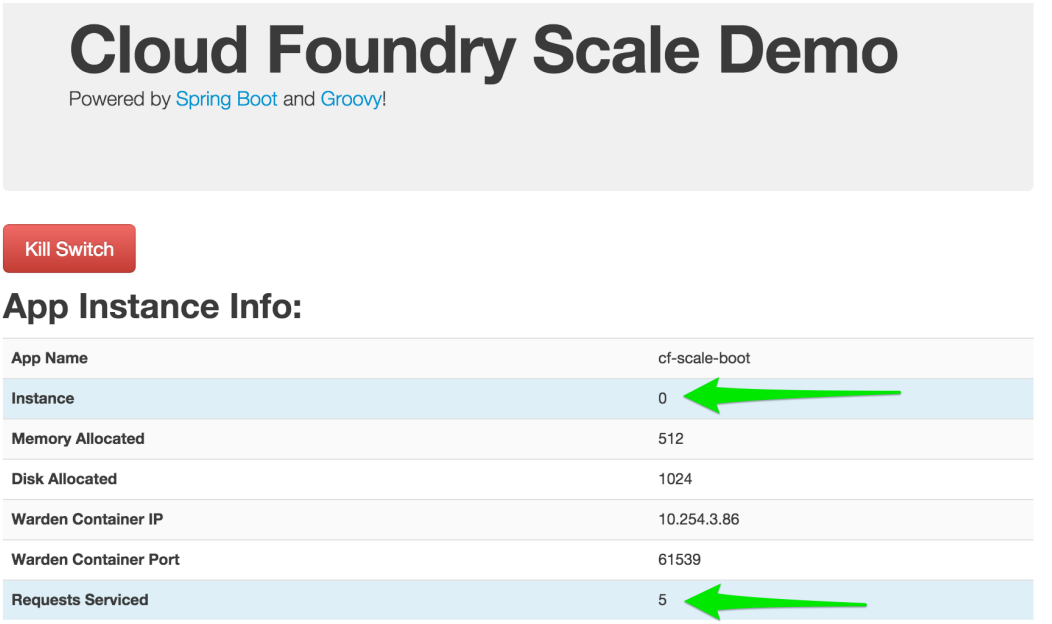


Figure 8.3. The application after scaling down

Chapter 9. Lab 2d - Monitoring Applications

Cloud Foundry provides several built-in mechanisms that allow us to monitor our applications' state changes and behavior. Additionally, Cloud Foundry actively monitors the health of our application processes and will restart them should they crash. In this lab, we'll explore a few of these mechanisms.

9.1. Events

Cloud Foundry only allows application configuration to be modified via its API. This gives application operators confidence that all changes to application configuration are known and auditable. It also reduces the number of causes that must be considered when problems arise.

All application configuration changes are recorded as *events*. These events can be viewed via the Cloud Foundry API, and viewing is facilitated via the CLI.

Take a look at the events that have transpired so far for our deployment of `cf-scale-boot`:

```
$ cf events cf-scale-boot
Getting events for app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...

time                          event                actor              description
2015-02-13T15:18:33.00-0600   audit.app.update     example@pivotal.io   instances: 1  ❶
2015-02-13T15:04:34.00-0600   audit.app.update     example@pivotal.io   instances: 5  ❷
2015-02-13T12:56:35.00-0600   audit.app.update     example@pivotal.io   state: STARTED  ❸
2015-02-13T12:56:26.00-0600   audit.app.update     example@pivotal.io  ❹
2015-02-13T12:56:26.00-0600   audit.app.map-route  example@pivotal.io  ❺
2015-02-13T12:56:24.00-0600   audit.app.create     example@pivotal.io   instances: 1, memory: 512, state: STOPPED,
    environment_json: PRIVATE DATA HIDDEN ❻
```

❻ Events are sorted newest to oldest, so we'll start from the bottom. Here we see the `app.create` event, which created our application's record and stored all of its metadata (e.g. `memory: 512`).

❺ The `app.map-route` event records the incoming request to assign a route to our application.

❹ This `app.update` event records the resulting change to our applications metadata.

❸ This `app.update` event records the change of our application's state to `STARTED`.

❷ Remember scaling the application up? This `app.update` event records the metadata change `instances: 5`.

❶ And here's the `app.update` event recording our scaling of the application back down with `instances: 1`.

1. Let's explicitly ask for the application to be stopped:

```
$ cf stop cf-scale-boot
Stopping app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
OK
```

2. Now, examine the additional `app.update` event:


```
$ cf events cf-scale-boot
Getting events for app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...

time                          event                actor              description
2015-02-13T15:59:10.00-0600   audit.app.update     example@pivotal.io   state: STOPPED
2015-02-13T15:18:33.00-0600   audit.app.update     example@pivotal.io   instances: 1
2015-02-13T15:04:34.00-0600   audit.app.update     example@pivotal.io   instances: 5
2015-02-13T12:56:35.00-0600   audit.app.update     example@pivotal.io   state: STARTED
2015-02-13T12:56:26.00-0600   audit.app.update     example@pivotal.io
2015-02-13T12:56:26.00-0600   audit.app.map-route  example@pivotal.io
2015-02-13T12:56:24.00-0600   audit.app.create     example@pivotal.io   instances: 1, memory: 512, state: STOPPED,
    environment_json: PRIVATE DATA HIDDEN
```

3. Start the application again:

```
$ cf start cf-scale-boot
Starting app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...

0 of 1 instances running, 1 starting
0 of 1 instances running, 1 starting
0 of 1 instances running, 1 starting
1 of 1 instances running


App started


OK

App cf-scale-boot was started using this command `JAVA_HOME=$PWD/.java-buildpack/open_jdk_jre JAVA_OPTS="
    -Djava.io.tmpdir=$TMPDIR -XX:OnOutOfMemoryError=$PWD/.java-buildpack/open_jdk_jre/bin/killjava.sh -Xmx382293K
    -Xms382293K -XX:MaxMetaspaceSize=64M -XX:MetaspaceSize=64M -Xss995K" SERVER_PORT=$PORT $PWD/.java-buildpack
    /spring_boot_cli/bin/spring run app.groovy`

Showing health and status for app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
OK

requested state: started
instances: 1/1
usage: 512M x 1 instances
urls: cf-scale-boot-stockinged-rust.cfapps.io
last uploaded: Fri Feb 13 18:56:29 UTC 2015

     state     since                    cpu    memory           disk
#0   running   2015-02-13 04:01:50 PM   0.0%   389.1M of 512M   128.9M of 1G
```

4. And again, view the additional `app.update` event:

```
$ cf events cf-scale-boot
Getting events for app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...

time                          event                actor              description
2015-02-13T16:01:28.00-0600   audit.app.update     example@pivotal.io   state: STARTED
2015-02-13T15:59:10.00-0600   audit.app.update     example@pivotal.io   state: STOPPED
2015-02-13T15:18:33.00-0600   audit.app.update     example@pivotal.io   instances: 1
2015-02-13T15:04:34.00-0600   audit.app.update     example@pivotal.io   instances: 5
2015-02-13T12:56:35.00-0600   audit.app.update     example@pivotal.io   state: STARTED
2015-02-13T12:56:26.00-0600   audit.app.update     example@pivotal.io
2015-02-13T12:56:26.00-0600   audit.app.map-route  example@pivotal.io
2015-02-13T12:56:24.00-0600   audit.app.create     example@pivotal.io   instances: 1, memory: 512, state: STOPPED,
    environment_json: PRIVATE DATA HIDDEN
```

9.2. Logs

One of the most important enablers of visibility into application behavior is logging. Effective management of logs has historically been very difficult. Cloud Foundry's log aggregation components simplify log management by assuming responsibility for it. Application developers need only log all messages to either `STDOUT` or `STDERR`, and the platform will capture these messages.

9.2.1. For Developers

Application developers can view application logs using the CF CLI.

1. Let's view recent log messages for `cf-scale-boot`:

```
$ cf logs cf-scale-boot --recent
```

Here are two interesting subsets of one output from that command:

Example 9.1. CF Component Logs

```
2015-02-13T14:45:39.40-0600 [RTR/0]      OUT cf-scale-boot-stockinged-rust.cfapps.io - [13/02/2015:20:45:39 +0000] "GET
    /css/bootstrap.min.css HTTP/1.1" 304 0 "http://cf-scale-boot-stockinged-rust.cfapps.io/" "Mozilla/5.0 (Macintosh; Intel
    Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.111 Safari/537.36" 10.10.66.88:50372
    x_forwarded_for:"50.157.39.197" vcap_request_id:84cc1b7a-bb30-4355-7512-5adaf36ff767 response_time:0.013115764
    app_id:7a428901-1691-4cce-b7f6-62d186c5cb55 ❶
2015-02-13T14:45:39.40-0600 [RTR/1]      OUT cf-scale-boot-stockinged-rust.cfapps.io - [13/02/2015:20:45:39 +0000] "GET
    /img/LOGO_CloudFoundry_Large.png HTTP/1.1" 304 0 "http://cf-scale-boot-stockinged-rust.cfapps.io/" "Mozilla/5.0
    (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.111 Safari/537.36"
    10.10.66.88:24323 x_forwarded_for:"50.157.39.197" vcap_request_id:b3e2466b-6a41-4c6d-5b3d-0f70702c0ec1
    response_time:0.010003444 app_id:7a428901-1691-4cce-b7f6-62d186c5cb55
2015-02-13T15:04:33.09-0600 [API/1]      OUT Tried to stop app that never received a start event  ❷
2015-02-13T15:04:33.51-0600 [CELL/12]     OUT Starting app instance (index 2) with guid 7a428901-1691-4cce-b7f6
    -62d186c5cb55 ❸
2015-02-13T15:04:33.71-0600 [CELL/4]      OUT Starting app instance (index 3) with guid 7a428901-1691-4cce-b7f6
    -62d186c5cb55
```

❶ An "Apache-style" access log event from the (Go)Router
❷ An API log event that corresponds to an event as shown in `cf events`
❸ A CELL log event indicating the start of an application instance on that execution agent (also known as a cell).

Example 9.2. Application Logs

```
2015-02-13T16:01:50.28-0600 [APP/0]      OUT 2015-02-13 22:01:50.282  INFO 36 --- [       runner-0] o.s.b.a.e.jmx.Endpoi
    ntMBeanExporter      : Located managed bean 'autoConfigurationAuditEndpoint': registering with JMX server as MBean
    [org.springframework.boot:type=Endpoint,name=autoConfigurationAuditEndpoint]
2015-02-13T16:01:50.28-0600 [APP/0]      OUT 2015-02-13 22:01:50.287  INFO 36 --- [       runner-0] o.s.b.a.e.jmx.Endpoi
    ntMBeanExporter      : Located managed bean 'shutdownEndpoint': registering with JMX server as MBean
    [org.springframework.boot:type=Endpoint,name=shutdownEndpoint]
2015-02-13T16:01:50.29-0600 [APP/0]      OUT 2015-02-13 22:01:50.299  INFO 36 --- [       runner-0] o.s.b.a.e.jmx.Endpoi
    ntMBeanExporter      : Located managed bean 'configurationPropertiesReportEndpoint': registering with JMX server as
    MBean [org.springframework.boot:type=Endpoint,name=configurationPropertiesReportEndpoint]
2015-02-13T16:01:50.36-0600 [APP/0]      OUT 2015-02-13 22:01:50.359  INFO 36 --- [       runner-0] s.b.c.e.t.TomcatEmbe
    ddedServletContainer : Tomcat started on port(s): 61316/http
2015-02-13T16:01:50.36-0600 [APP/0]      OUT Started...
2015-02-13T16:01:50.36-0600 [APP/0]      OUT 2015-02-13 22:01:50.364  INFO 36 --- [       runner-0] o.s.boot.SpringApplication
    : Started application in 6.906 seconds (JVM running for 15.65)
```

As you can see, Cloud Foundry's log aggregation components capture both application logs and CF component logs relevant to your application. These events are properly interleaved based on time, giving you an accurate picture of events as they transpired across the system.

2. To get a running "tail" of the application logs rather than a dump, simply type:


```
$ cf logs cf-scale-boot
```

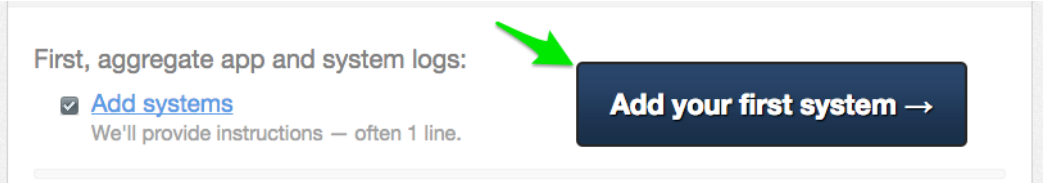
You can try various things like refreshing the browser and triggering stop/start events to see logs being generated.

9.2.2. For Operators (OPTIONAL)

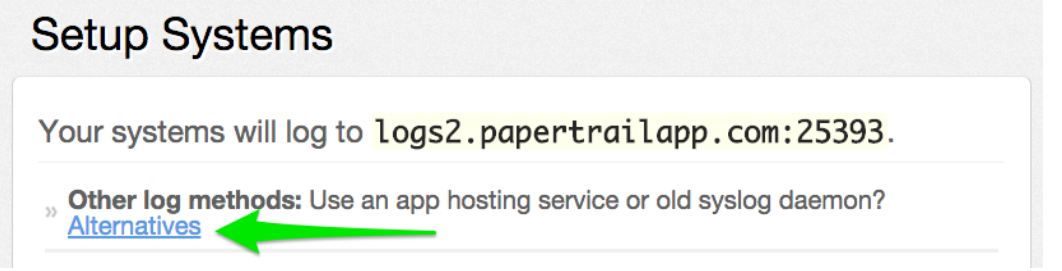
Application operators will also enjoy commands like `cf logs`, but are often interested in long-term retention, indexing, and analysis of logs as well. Cloud Foundry currently only provides short-term retention of logs. To meet these needs, Cloud Foundry provides the ability to drain logs to third-party providers.

In this section, we'll drain logs to a very simple provider called Papertrail.

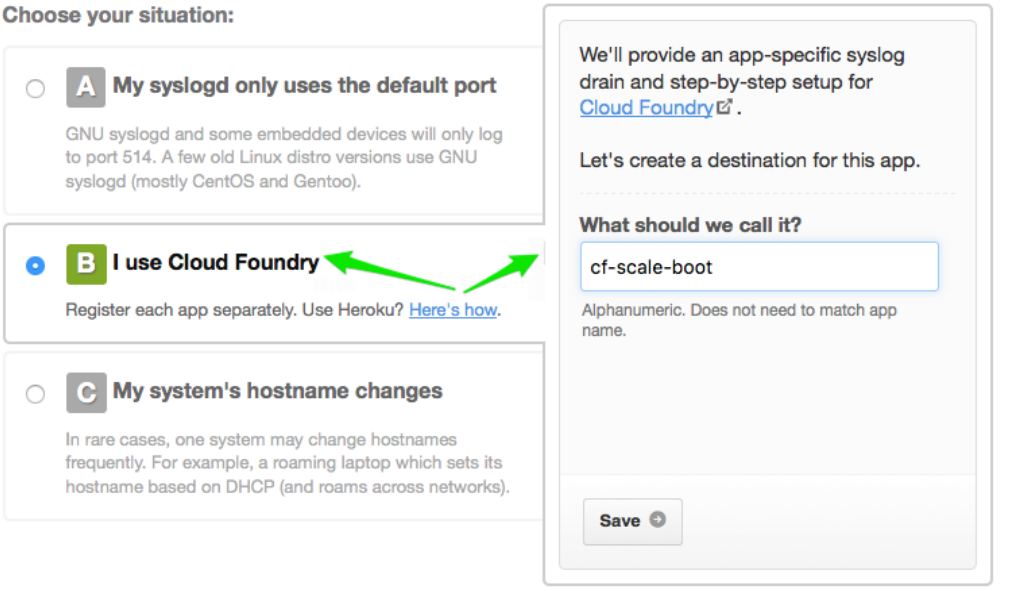
1. Visit https://papertrailapp.com and create a free account.
2. Login to your account and add your first system:



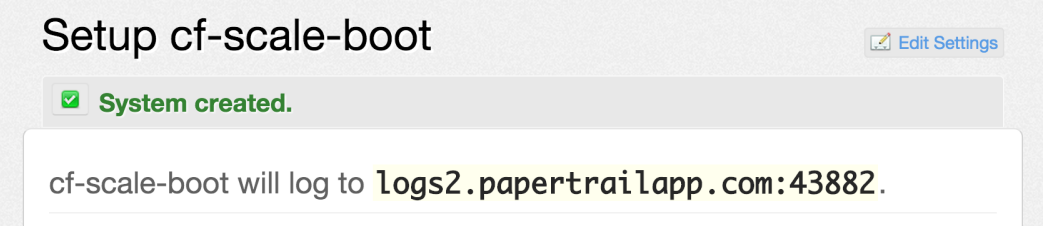
3. Click on "Alternatives":



4. Choose "I use Heroku" and provide a name:



5. Note the URL + Port assigned to your application:



6. We'll use a Cloud Foundry user-provided service instance to create the log drain for our application using the URL + Port provided by Papertrail:

```
$ cf cups cf-scale-boot-logs -l syslog://logs2.papertrailapp.com:43882
Creating user provided service cf-scale-boot-logs in org oreilly-class / space instructor as example@pivotal.io...
OK
```

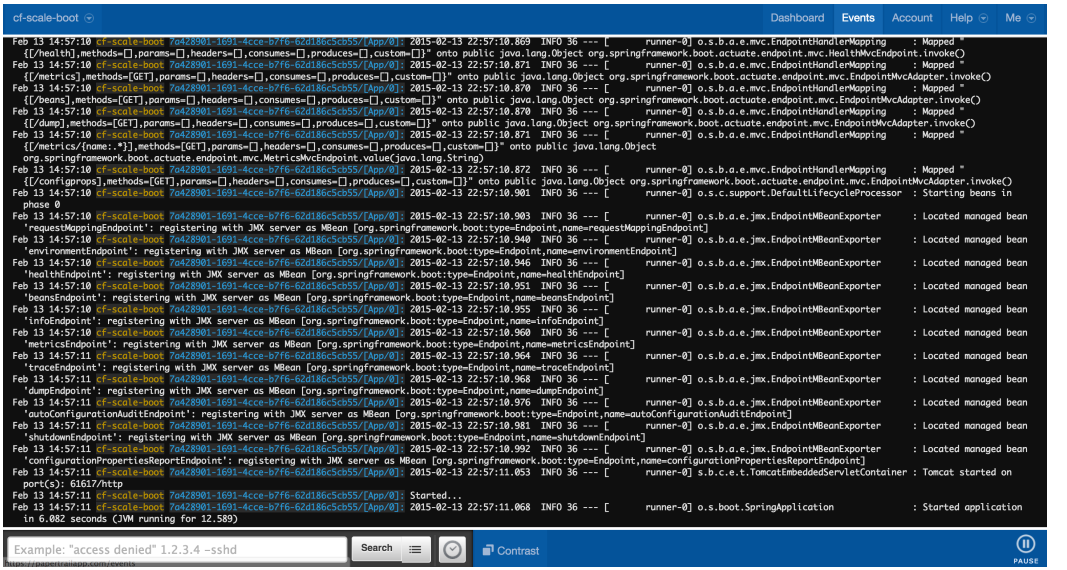
7. We bind that service instance like those we created in Lab 2c:

```
$ cf bs cf-scale-boot cf-scale-boot-logs
Binding service cf-scale-boot-logs to app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect
```


8. We'll use a `cf restart` rather than `cf restage` to make the binding take effect:

```
$ cf restart cf-scale-boot
```

9. Refresh the Papertrail "Events" tab to see log events immediately flowing to the log viewing page:



You can see how to connect to other third-party log management systems in the Cloud Foundry documentation.

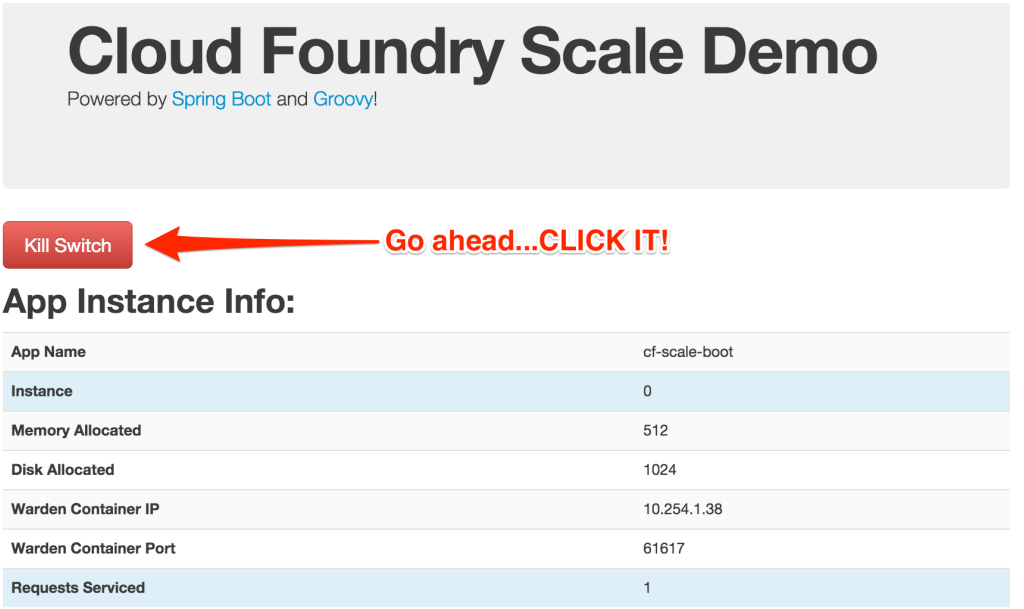
9.3. Health

Cloud Foundry's Heatlh Manager actively monitors the health of our application processes and will restart them should they crash.

1. If you don't have one already running, start a log tail for `cf-scale-boot`:

```
$ cf logs cf-scale-boot
```

2. Visit the application in the browser, and click on the "Kill Switch" button. This button will trigger a JVM exit with an error code (`System.exit(1)`), causing the Health Manager to observe an application instance crash:



3. After clicking the kill switch a couple of interesting things should happen. First, you'll see an error code returned in the browser, as the request you submitted never returns a response:

```
502 Bad Gateway: Registered endpoint failed to handle the request.
```

Also, if you're paying attention to the log tail, you'll see some interesting log messages fly by:

```
2015-02-13T17:17:54.86-0600 [App/0]      OUT 2015-02-13 23:17:54.860 ERROR 36 --- [io-61617-exec-5] WebApplication
    : KILL SWITCH ACTIVATED! ❶
2015-02-13T17:17:54.86-0600 [App/0]      OUT 2015-02-13 23:17:54.869  INFO 36 --- [       Thread-2] ationConfigEmbeddedW
    ebApplicationContext : Closing org.springframework.boot.c$ntext.embedded.AnnotationConfigEmbeddedWebApplicationContext@6a62811d:
    startup date [Fri Feb 13 22:57:05 UTC 2015]; root of context hierarchy
2015-02-13T17:17:54.87-0600 [App/0]      OUT 2015-02-13 23:17:54.870  INFO 36 --- [       Thread-2] o.s.c.support.Defaul
    tLifecycleProcessor  : Stopping beans in phase 0
2015-02-13T17:17:54.87-0600 [App/0]      OUT 2015-02-13 23:17:54.874  INFO 36 --- [       Thread-2] o.s.b.a.e.jmx.Endpoi
    ntMBeanExporter      : Unregistering JMX-exposed beans on shutdown
2015-02-13T17:17:54.87-0600 [App/0]      OUT 2015-02-13 23:17:54.878  INFO 36 --- [       Thread-2] o.s.j.e.a.Annotation
    MBeanExporter        : Unregistering JMX-exposed beans on shutdown
2015-02-13T17:17:57.30-0600 [RTR/1]      OUT cf-scale-boot-stockinged-rust.cfapps.io - [13/02/2015:23:17:54 +0000] "GET
    /killSwitch HTTP/1.1" 502 0 "http://cf-scale-boot-stockinged-rust.cfapps.io/" "Mozilla/5.0 (Macintosh; Intel Mac OS X
    10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.111 Safari/537.36" 10.10.2.122:25194
    x_forwarded_for:"50.157.39.197" vcap_request_id:fc2b93a9-451d-460f-726e-14ada0069ff4 response_time:2.465784807
    app_id:7a428901-1691-4cce-b7f6-62d186c5cb55 ❷
2015-02-13T17:17:57.31-0600 [App/0]      ERR
2015-02-13T17:17:57.38-0600 [API/2]      OUT App instance exited with guid 7a428901-1691-4cce-b7f6-62d186c5cb55 payload:
    {"cc_partition"=>"default", "droplet"=>"7a428901-1691-4cce-b7f6-62d186c5cb55", "version"=>"ebcdb262-2851-4716-83a4
    -c816fa2c68bb", "instance"=>"1eecfb8d3b41492a8e36237b365a4755", "index"=>0, "reason"=>"CRASHED", "exit_status"=>1,
    "exit_description"=>"app instance exited", "crash_timestamp"=>1423869477} ❸
```


❶ Just before issuing the `System.exit(1)` call, the application logs that the kill switch was clicked.
❷ The (Go)Router logs the 502 error.
❸ The API logs that an application instance exited due to a crash.

4. Check the application events to see another indicator of the crash:

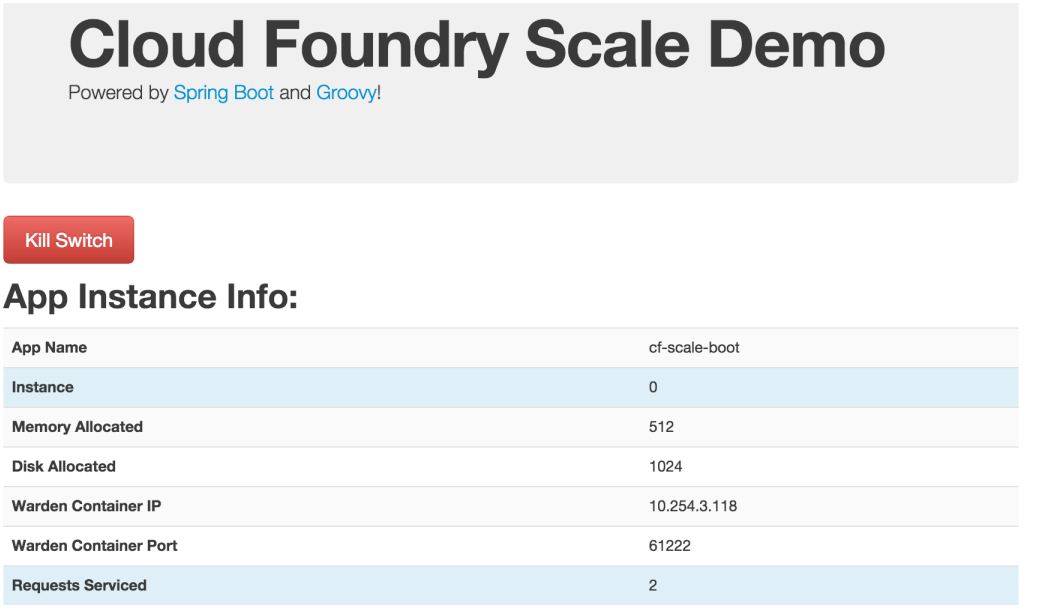
```
$ cf events cf-scale-boot
Getting events for app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...

time                          event                actor              description
2015-02-13T17:17:57.00-0600   app.crash            cf-scale-boot      index: 0, reason: CRASHED, exit_description: app
    instance exited, exit_status: 1
```

5. By this time you should have noticed some additional interesting events in the logs:

```
2015-02-13T17:18:14.67-0600 [CELL/19]     OUT Starting app instance (index 0) with guid 7a428901-1691-4cce-b7f6
    -62d186c5cb55 ❶
2015-02-13T17:18:24.72-0600 [APP/0]      OUT Resolving dependencies....  ❷
2015-02-13T17:18:26.62-0600 [APP/0]      OUT    .   ____          _            __ _ _
2015-02-13T17:18:26.62-0600 [APP/0]      OUT   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
2015-02-13T17:18:26.62-0600 [APP/0]      OUT ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
2015-02-13T17:18:26.62-0600 [APP/0]      OUT  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
2015-02-13T17:18:26.62-0600 [APP/0]      OUT   '  |____| .__|_| |_|_| |_\__, | / / / /
2015-02-13T17:18:26.62-0600 [APP/0]      OUT  =========|_|==============|___/=/_/_/_/
2015-02-13T17:18:26.62-0600 [APP/0]      OUT  :: Spring Boot ::        (v1.1.9.RELEASE)
```

❶ The CELL indicates that it is starting another instance of the application as a result of the health-management system observing a difference between the desired and actual state (i.e. running instances = 1 vs. running instances = 0).
❷ The new application instance starts logging events as it starts up.

6. Revisiting the **HOME PAGE** of the application (don't simply refresh the browser as you're still on the `/killSwitch` endpoint and you'll just kill the application again!) and you should see a fresh instance started:



9.4. Clean Up

Because of the limited PWS quota we have for this course, let's clean up our application and services to make room for future labs.

1. Delete the `cf-scale-boot` application:

```
$ cf d cf-scale-boot

Really delete the app cf-scale-boot?> y
Deleting app cf-scale-boot in org oreilly-class / space instructor as example@pivotal.io...
'OK
```

2. Delete the `cf-scale-boot-logs` service:

```
$ cf ds cf-scale-boot-logs

Really delete the service cf-scale-boot-logs?> y
Deleting service cf-scale-boot-logs in org oreilly-class / space instructor as example@pivotal.io...
OK
```


Chapter 10. Lab 03a - Build a Hypermedia-Driven RESTful Web Service with Spring Data REST

Estimated time to complete: 30 minutes

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.

10.1. Initializing the Application

1. Import the project `$COURSE_HOME/lab_03/lab_03a/initial/companies` into your IDE.

2. Open its `pom.xml` file and add a runtime dependency on the H2 in-memory database:

```
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
```

3. Create the package `io.pivotal.demo.domain` and in that package create the class `Company`. Into that file you can paste the following source code, which represents companies based on exchange and trading symbol:

Note

JPA (Java Persistence API) is an abstraction layer on top of Hibernate. When importing annotations such as `@Entity`, you should use the package `javax.persistence` instead of `org.hibernate`.

```
@Entity
@Table(name = "COMPANIES")
public class Company implements Serializable {
    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    private String symbol;

    private String exchange;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
```



```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSymbol() {
        return symbol;
    }

    public void setSymbol(String symbol) {
        this.symbol = symbol;
    }

    public String getExchange() {
        return exchange;
    }

    public void setExchange(String exchange) {
        this.exchange = exchange;
    }
}
```

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

```
@RepositoryRestResource(collectionResourceRel = "companies", path = "companies")
public interface CompanyRepository extends PagingAndSortingRepository<Company, Long> {
}
```

2. Add JPA and REST Repository support to the `io.pivotal.demo.quotes.CompaniesApplication` class that was generated by Spring Initializr.

```
@SpringBootApplication
@EnableJpaRepositories // <---- Add this
@Import(RepositoryRestMvcConfiguration.class) // <---- And this
public class CompaniesApplication {

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

3. Run the application and open http://localhost:8080/companies in your browser.
   You'll see that the primary endpoint automatically exposes the ability to page, size, and sort the response JSON.
   So what have you done? Created four small classes and one build file, resulting in a fully-functional REST microservice. The application's `DataSource` is created automatically by Spring Boot using the in-memory database because no other `DataSource` was detected in the project.

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



```
    "number" : 0
  }
}
```

Next we'll import some data.

10.2. Importing Data

1. Inside your `'initial` folder, there is a file called `import.sql`. Add it to `src/main/resources`. This is a rather large dataset containing all of the companies and their symbol codes in the NASDAQ and NSYE exchanges. This file will automatically be picked up by Hibernate and imported into the in-memory database.

Note

Here we simply use a Spring Boot convention that a file named `import.sql` in the root of the classpath will be executed at startup time. You can read here if you'd like to know more details about it.

1. Run the application and access it again in your browser

2. 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`.

```
{
  "_embedded" : {
    "companies" : [ {
      "name" : "1347 Capital Corp.",
      "symbol" : "TFSC",
      "exchange" : "NASDAQ",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/companies/1"
        },
        "company" : {
          "href" : "http://localhost:8080/companies/1"
        }
      }

    ....

    }, {
      "name" : "A V Homes, Inc.",
      "symbol" : "AVHI",
      "exchange" : "NASDAQ",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/companies/20"
        },
        "company" : {
          "href" : "http://localhost:8080/companies/20"
        }
      }
    } ]
  },
  "_links" : {
    "first" : {
      "href" : "http://localhost:8080/companies?page=0&size=20"
    },
```



```
    "self" : {
      "href" : "http://localhost:8080/companies"
    },
    "next" : {
      "href" : "http://localhost:8080/companies?page=1&size=20"
    },
    "last" : {
      "href" : "http://localhost:8080/companies?page=320&size=20"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile/companies"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 6419,
    "totalPages" : 321,
    "number" : 0
  }
}
```

3. Try to access the following urls to see how the application behaves:

   - http://localhost:8080/companies?size=5
   - http://localhost:8080/companies?size=5&page=3
   - http://localhost:8080/companies?sort=symbol,desc

   Next we'll add searching capabilities.

10.3. Adding Search

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

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

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

@RestResource(path = "symbol", rel = "symbol")
Page<Company> findBySymbolIgnoreCase(@Param("q") String symbol, Pageable pageable);

@RestResource(path = "exchange", rel = "exchange")
Page<Company> findByExchange(@Param("q") String exchange, Pageable pageable);
```

2. Run the application

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

```
{
...
  "_links" : {
    "next" : {
      "href" : "http://localhost:8080/companies?page=1&size=20"
    },
    "self" : {
      "href" : "http://localhost:8080/companies{?page,size,sort}",
      "templated" : true
    },
    "search" : {
      "href" : "http://localhost:8080/companies/search"
```



```
    }
},
...
```

Note

The `search` endpoint is automatically added by `Spring Data REST` when you have declared at least one `findBy` method

1. Access the new `search` endpoint on http://localhost:8080/companies/search

```
{
  "_links" : {
    "nameContains" : {
      "href" : "http://localhost:8080/companies/search/nameContains{?q,page,size,sort}",
      "templated" : true
    },
    "symbol" : {
      "href" : "http://localhost:8080/companies/search/symbol{?q,page,size,sort}",
      "templated" : true
    },
    "name" : {
      "href" : "http://localhost:8080/companies/search/name{?q,page,size,sort}",
      "templated" : true
    },
    "exchange" : {
      "href" : "http://localhost:8080/companies/search/exchange{?q,page,size,sort}",
      "templated" : true
    },
    "self" : {
      "href" : "http://localhost:8080/companies/search"
    }
  }
}
```

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

2. Try a few of these endpoints. Feel free to substitute your own values for the parameters.
   http://localhost:8080/companies/search/exchange?q=NASDAQ
   http://localhost:8080/companies/search/name?q=Amerco
   http://localhost:8080/companies/search/nameContains?q=Apple&size=5

10.4. Pushing to Cloud Foundry

1. An empty `manifest.yml` file has been placed at the root of your project. Paste the below configuration inside your manifest file:

```
---
applications:
- name: companies
  host: companies-${random-word}
  memory: 512M
  instances: 1
  path: target/lab_03a-companies-0.0.1-SNAPSHOT.jar
  timeout: 180 # to give time for the data to import
```

2. Build the JAR and push it to Cloud Foundry (you need to leave the IDE to do this):


```
mvn package
cf push
```

3. Once it has uploaded and run the buildpack, you should see output like this …

```
1 of 1 instances running

App started

OK

App companies was started using this command `CALCULATED_MEMORY=$($PWD/.java-buildpack/open_jdk_jre/bin/java-buildpack
    -memory-calculator-2.0.0_RELEASE -memorySizes=metaspace:64m.. -memoryWeights=heap:75,metaspace:10,native:10,stack:5
    -memoryInitials=heap:100%,metaspace:100% -totMemory=$MEMORY_LIMIT) && SERVER_PORT=$PORT $PWD/.java-buildpack
    /open_jdk_jre/bin/java -cp $PWD/.:$PWD/.java-buildpack/spring_auto_reconfiguration/spring_auto_reconfiguration
    -1.10.0_RELEASE.jar -Djava.io.tmpdir=$TMPDIR -XX:OnOutOfMemoryError=$PWD/.java-buildpack/open_jdk_jre/bin/killjava.sh
    $CALCULATED_MEMORY org.springframework.boot.loader.JarLauncher`

Showing health and status for app companies in org pivot-cqueiroz / space development as cqueiroz@pivotal.io...
OK

requested state: started
instances: 1/1
usage: 512M x 1 instances
urls: companies-getable-section.cfapps.pez.pivotal.io
last uploaded: Thu Nov 26 11:02:33 UTC 2015
stack: cflinuxfs2
buildpack: java-buildpack=v3.3.1-offline-https://github.com/cloudfoundry/java-buildpack.git#063836b java-main open-jdk
    -like-jre=1.8.0_65 open-jdk-like-memory-calculator=2.0.0_RELEASE spring-auto-reconfiguration=1.10.0_RELEASE

     state     since                    cpu    memory           disk           details
#0   running   2015-11-26 07:03:09 PM   0.0%   442.3M of 512M   151.1M of 1G
```

4. Access the application at the random route provided by CF - your URL will be different, look in the App Manager console to see what it is:

```
http://companies-<your_path>.cfapps.io/companies
```


Chapter 11. Lab 03b - Leveraging Spring Cloud Connectors for Service Binding

In this lab we'll bind our RESTful web service from the previous lab to a MySQL database and leverage Spring Cloud Connectors to easily connect to it.

Note

The completed code for this lab can be found at `$COURSE_HOME/lab_03/lab_03b/solution/companies`.

11.1. Using Spring Cloud Connectors

1. Change to the lab directory (the initial state for this lab is the same as the completed state for theprevious lab, so you can choose to continue with that project if you like):

```
$ cd $COURSE_HOME/lab/lab_03/lab_03b/initial/companies
```

2. At present we're still using the in-memory database. Let's connect to a MySQL database service. From the CLI, let's *create* a MySQL service instance:

```
$ cf cs p-mysql 100mb-dev companies-db
Creating service companies-db...
OK
```

   Again, if `p-mysql` service is not available, please use the free `spark` plan from `cleardb` like this `cf cs cleardb spark companies-db`.

3. Next add the service to your application manifest, which will *bind* the service to our application on the next push. We'll also add an environment variable to switch on the "cloud" profile,

```
---
applications:
- name: companies
  memory: 512M
  instances: 1
  path: target/lab_03b-companies-1.0.0.jar
  timeout: 180
  services:                       # Add
  - companies-db                  # these
  env:                            # four
    SPRING_PROFILES_ACTIVE: cloud  # lines
```

   You can also accomplish the service binding by explicitly binding the service at the command-line:

```
$ cf bind-service companies companies-db
Binding service companies-db to app companies...
OK
```

4. Next we'll add Spring Cloud and MySQL dependencies to our maven pom file. Just after the dependency

declaration for `h2`, add add the following in the `dependencies` section:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-spring-service-connector</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
</dependency>
```

5. Next, let's create the package `io.pivotal.demo.config` and create in that package the class `CloudDataSourceConfig`. Add the following code:

```
@Profile("cloud")
@Configuration
public class CloudDataSourceConfig extends AbstractCloudConfig {
    @Bean
    public DataSource dataSource() {
        return connectionFactory().dataSource();
    }
}
```

   The `@Profile` annotation will cause this class (which becomes Spring configuration when annotated as `@Configuration`) to be added to the configuration set because of the `SPRING_PROFILES_ACTIVE` environment variable we added earlier. You can still run the application locally (with the default profile) using the embedded database.
   With this code, Spring Cloud will detect a bound service that is compatible with `DataSource`, read the credentials, and then create a `DataSource` as appropriate (it will throw an exception otherwise).

6. Add the following to `src/main/resources/application.yml` to cause Hibernate to create the database schema and import data at startup. This is done automatically for embedded databases, but not for a custom `DataSource`. Other Hibernate native properties can be set in a similar fashion:

```
spring.jpa:
  hibernate.ddl-auto: create
```

7. Build the application:

```
$ mvn clean package
```

8. Re-push the application:

```
$ cf push
```

9. In your web browser, have a look at the env variables used by your application (http://companies-<your_path>.cfapps.io/env)

```
...
"vcap" : {
   "vcap.services.companies-db.name" : "companies-db",
   "vcap.application.limits.mem" : "1024",
   "vcap.services.companies-db.label" : "p-mysql",
   ...
   "vcap.services.companies-db.tags" : "mysql,relational",
   ...
```

   The application is now running against a MySQL database.

11.2. Customising the `DataSource`

1. You can customize the database connection that Spring Cloud creates with a few lines of code. Change the `dataSource` method in `CloudDataSourceConfig` to add some pooling and connection configuration:

```
@Bean
public DataSource dataSource() {
    PooledServiceConnectorConfig.PoolConfig poolConfig =
            new PooledServiceConnectorConfig.PoolConfig(5, 200);

    DataSourceConfig.ConnectionConfig connectionConfig =
            new DataSourceConfig.ConnectionConfig("characterEncoding=UTF-8");
    DataSourceConfig serviceConfig = new DataSourceConfig(poolConfig, connectionConfig);

    return connectionFactory().dataSource("companies-db", serviceConfig);
}
```

2. Build the application:

```
$ mvn clean package
```

3. Re-push the application:

```
$ cf push
```


Chapter 12. Lab 04a - Identify the bounded contexts of the Monolith App

In this lab we'll identify the bounded contexts so that they can be easily extracted into microservices.

12.1. Understanding the spring-trader monolith application

1. Change to the lab directory:

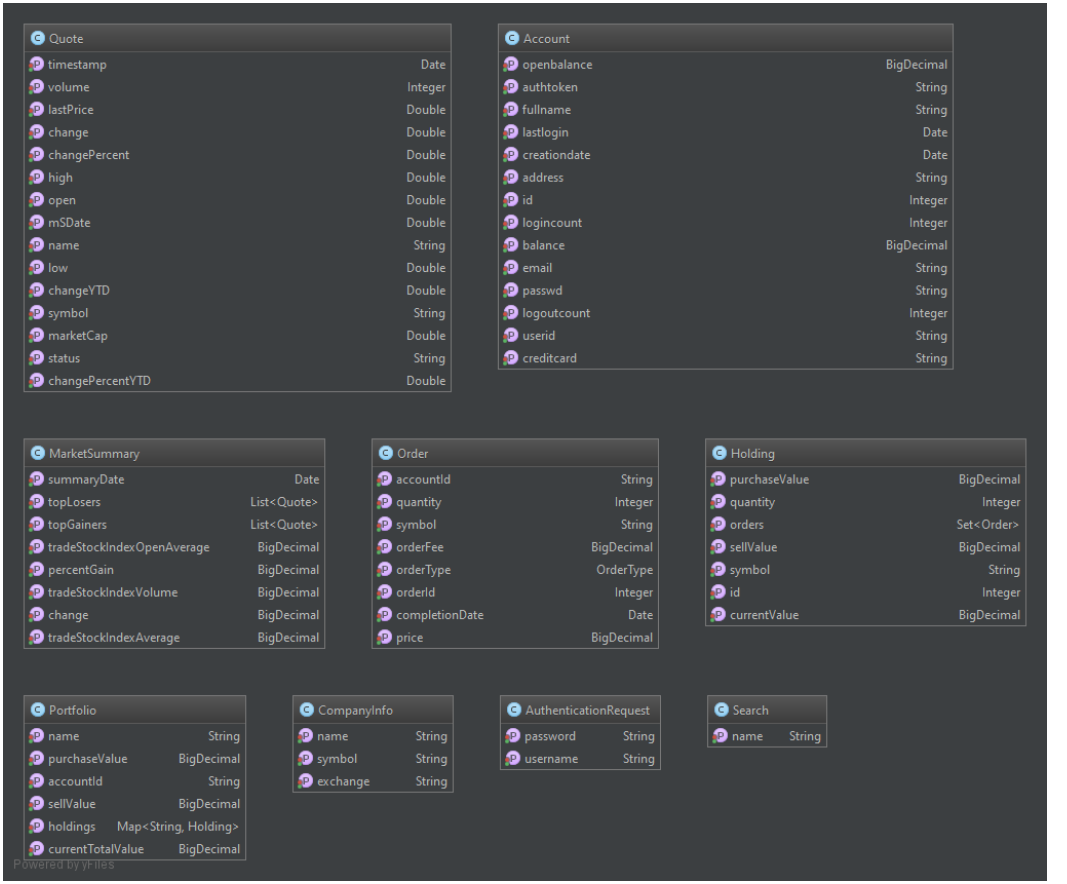
```
$ cd $COURSE_HOME/lab/lab_04/lab_04a/springtrader-monolith
```

   and import the project (via `pom.xml`) into your IDE of choice.

2. Run the application

3. Browse spring-trader on http://localhost:8080
   Create a user account with at least $5000 balance, perform a few trades and check your portfolio.

4. Domain classes
   Inside your IDE, look at the domain classes (package `io.pivotal.springtrader.domain`).
   Try to write down all the relations between those domain classes



12.2. Decomposition into microservices

1. Based on the domain model, identify how the Spring Trader monolith application should be split up.

Chapter 13. Lab 05a - Build a Quotes services using MongoDB

Estimated time to complete: 30 minutes

In this lab we'll begin the first subproject of our complete cloud-native application architecture: *SpringTrader*, which implements the quotes fetching portion of a *'Trading-like*' application.

This microservice will provide the *Quotes Service*, which will allow us to search for companies and symbols in order to make transactions.

We'll be using the MarkitOnDemand API, and we'll model its `Quote and Company as a Stock` concept in this microservice.

The application that we are building is structured like this:

```
                  +---- Quotes
                  |
          Web UI ---+---- Accounts
                  |
                  +---- Portfolio
```

The next four labs look at each component in turn, starting with Quotes.

13.1. Exploring springtrader-quotes

1. Change to the lab directory:

```
$ cd $COURSE_HOME/lab/lab_05/lab_05a/initial/springtrader-quotes
```

   and import the project (via `pom.xml`) into your IDE.

2. Explore the new `quotes` microservice:

   - How many domain classes do you now have?
   - How many controller classes?
   - You can see that `html`, `css` and `javascript` files are not included in `springtrader-quotes`. They will all be contained in the `WebUI` microservice (a dedicated web front-end application) that we will see in a later lab.
   - This is a REST application returning market data in JSON format.

- Stock data is stored in MongoDB via the `StockRepository` - if you examine the code you can see it extends MongoRepository and uses Spring Data to auto-generate the code

3. Run the application from your IDE (if using STS run as a Spring Boot application) It defaults to using an embedded version of MongoDB

4. Access the application at http://localhost:8086/quote/msft
   You should see output like this:

```
{
   "Symbol":"MSFT",
   "Name":"Microsoft Corp",
   "Exchange":"NASDAQ",
   "Status":"SUCCESS",
   "LastPrice":53.95,
   "Change":0.260000000000005,
   "ChangePercent":0.484261501210663,
   "Timestamp":"Fri Nov 27 17:59:00 GMTZ 2015",
   "MSDate":42335.5409722222,
   "MarketCap":4.3094790635E11,
   "Volume":716782,
   "ChangeYTD":46.45,
   "ChangePercentYTD":16.1463939720129,
   "High":54.08,
   "Low":53.81,
   "Open":53.94
}
```

13.2. Preparing for Cloud Foundry

1. We want to bind to a MongoDB data source when running on Cloud Foundry, so let's create one:
   We will use the `cf create-service` command, which can be abbreviated to `cf cs`:

```
$ cf cs p-mongodb development springtrader-quotes-db
Creating service springtrader-quotes-db in org cqueiroz-pivot / space development as cqueiroz@pivotal.io...
OK
```

Note

If you are using Pivotal Web Service, the above service is not available in the command line. You just need to create it using the Web console athttps://console.run.pivotal.io. You can select a MongoDB service and name it `springtrader-quotes-db`.

1. Go into the package `io.pivotal.sprintrader.quotes.config` and observe the differences between `MongoCloudConfig` (`cloud` profile) and `MongoLocalConfig` (`local` profile)
   As you will see, the `local` profile provides you with a MongoDB in-memory database. The `cloud` profile will rely on the service that you have just created on Cloud Foundry.

2. Open `src/main/resources/application.yml` and see that a `cloud` profile has been setup in preparation for

deployment to Cloud Foundry:

```
...
---
spring:
  profiles: cloud
---
...
```

3. Rebuild the JAR:

```
$ mvn clean package
```

13.3. Deploying to Cloud Foundry

1. Open your `manifest.yml` file and paste inside the following configuration:

```
timeout: 180
instances: 1
memory: 512M
env:
    SPRING_PROFILES_ACTIVE: cloud
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
applications:
- name: quotes
  random-route: true
  path: target/lab_05a-quotes-1.0.0-SNAPSHOT.jar
  services: [ springtrader-quotes-db ]
```

2. Push to Cloud Foundry. Assuming it pushes successfully, the health and status output will show you the URL you application has been mapped to (`urls` in the output below):

```
$ cf push

...

Showing health and status for app quotes in org pivot-cqueiroz / space development as cqueiroz@pivotal.io...
OK

requested state: started
instances: 1/1
usage: 512M x 1 instances
urls: quotes-undespising-lenition.cfapps.pez.pivotal.io
last uploaded: Mon Dec 7 22:17:58 UTC 2015
stack: cflinuxfs2
buildpack: java-buildpack=v3.3.1-offline-https://github.com/cloudfoundry/java-buildpack.git#063836b java-main java-opts
    open-jdk-like-jre=1.8.0_65 open-jdk-like-memory-calculator=2.0.0_RELEASE spring-auto-reconfiguration=1.10.0_RELEASE

     state     since                    cpu    memory           disk           details
#0   running   2015-12-07 07:18:48 PM   3.7%   367.2M of 512M   140.2M of 1G
```

3. Using `curl` or your browser, access the application at the random route provided by CF (in this case `http://quotes-xxxx-yyyy.cfapps.io/quote/aapl`). The output should look something like:

```
{
   "Symbol":"AAPL",
   "Name":"Apple Inc.",
   "Exchange":"NASDAQ",
   "Status":"SUCCESS",
   "LastPrice":117.82,
   "Change":-0.210000000000008,
   "ChangePercent":-0.177920867576047,
   "Timestamp":"Fri Nov 27 17:59:00 GMTZ 2015",
   "MSDate":42335.5409722222,
   "MarketCap":6.5688549842E11,
   "Volume":859197,
```



```
    "ChangeYTD":110.38,
    "ChangePercentYTD":6.74035151295524,
    "High":118.41,
    "Low":117.6,
    "Open":118.27
}
```


Chapter 14. Lab 05b - Build Accounts Service with Mysql/H2

Estimated time to complete: 30 minutes

In this lab we'll begin the second subproject of our complete cloud-native application architecture: *SpringTrader*.

This microservice will provide the *Accounts Service*, which will allow us to create and browse user accounts. We'll be using a relational model to represent the user accounts.

Note

The completed code for this lab can be found at `$COURSE_HOME/lab_05/lab_05b/solution/springtrader-accounts`.

14.1. Exploring springtrader-accounts

1. Change to the lab directory:

```
$ cd $COURSE_HOME/lab/lab_05/lab_05b/initial/springtrader-accounts
```

   and import the project (via `pom.xml`) into your IDE of choice.

2. Explore the new `accounts` microservice:

   - How many domain classes do you now have? Are there relationships between those classes?
   - How many controller classes?
     Like `spring-trader-quotes`, you can see that static web files are not included in `springtrader-accounts`. This is also a REST application returning JSON data

3. Review the `application.yml`, it is empty. What do we need?

   - To run multiple apps locally we will need to define a port other than 8080
   - To use JPA to persist account data into our RDBMS, we need to help Spring Boot configure it
   - To set the logging level to INFO - if you have problems later, you can change INFO to DEBUG
     Copy this code …

```
server.port: 8082

security.basic.enabled: false
```



```
spring.jpa:
    hibernate.ddl-auto: create

logging:
  level:
    io.pivotal.springtrader: 'INFO'
```

1. To run the application locally we will use an in-memory H2 database which is sufficient for testing. You need to add the H2 dependency to your `pom.xml` file.

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

2. Open `src/main/resources/import.sql`. As seen previously, Spring Boot automatically loads this file at startup time (based on a naming convention). You will see there are two accounts which will be created by default at application startup time.

3. Run the application by launching `AccountsApplication` and open the url http://localhost:8082/account/1 into your web browser. You should see the following:

```
{
   "id":1,
   "address":"45 Test Dr.",
   "passwd":"123",
   "userid":"johnsmith",
   "email":"john@pivotal.io",
   "creditcard":"999999999",
   "fullname":"John Smith",
   "authtoken":null,
   "creationdate":null,
   "openbalance":1200.50,
   "logoutcount":0,
   "balance":null,
   "lastlogin":null,
   "logincount":0
}
```

14.2. Preparing for Cloud Foundry

1. Create the MySQL service.
   We will use the `cf create-service` command, which can be abbreviated to `cf cs`:

```
$ cf cs p-mysql 100mb-dev springtrader-accounts-db
```

Note

if you are using Pivotal Web Services (PWS), the above service is not available in the command line. You just need to create it using the Web console athttps://console.run.pivotal.io . You can select a MySql database service and name it `springtrader-accounts-db`.

2. Rebuild the JAR:

```
$ mvn clean package
```

14.3. Deploying to Cloud Foundry

1. Paste the following inside `manifest.yml`:

```
---
timeout: 180
instances: 1
memory: 512M
env:
    SPRING_PROFILES_DEFAULT: cloud
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
applications:
- name: accounts
  random-route: true
  path: target/lab_05b-accounts-1.0.0-SNAPSHOT.jar
  services: [ springtrader-accounts-db ]
```

2. Push to Cloud Foundry:

```
$ cf push

...

Showing health and status for app accounts in org pivot-cqueiroz / space development as cqueiroz@pivotal.io...
OK

requested state: started
instances: 1/1
usage: 512M x 1 instances
urls: accounts-unexploited-boneset.cfapps.io
last uploaded: Mon Dec 7 21:01:55 UTC 2015
stack: cflinuxfs2
buildpack: java-buildpack=v3.3.1-offline-https://github.com/cloudfoundry/java-buildpack.git#063836b java-main java-opts
    open-jdk-like-jre=1.8.0_65 open-jdk-like-memory-calculator=2.0.0_RELEASE spring-auto-reconfiguration=1.10.0_RELEASE

     state     since                    cpu    memory           disk           details
#0   running   2015-12-07 06:02:51 PM   0.9%   456.4M of 512M   163.4M of 1G
```

3. Using `curl` or your browser, access the application at the random route provided by CF (in this case `http://accounts-unexploited-boneset.cfapps.io/account/1`). The outpu should be the same as when run locally:

```
{
   "id":1,
   "address":"45 Test Dr.",
   "passwd":"123",
   "userid":"johnsmith",
   "email":"john@pivotal.io",
   "creditcard":"999999999",
   "fullname":"John Smith",
   "authtoken":null,
   "creationdate":null,
   "openbalance":1200.50,
   "logoutcount":0,
   "balance":null,
   "lastlogin":null,
   "logincount":0
}
```


Chapter 15. Lab 05c - Build a Portfolio with MySQL

Estimated time to complete: 30 minutes

In this lab we'll begin the third subproject of our complete cloud-native application architecture: *SpringTrader*, which implements the quotes fetching portion of a "Trading-like" application. This microservice will provide the *Portfolio Service*, which will allow us to see our trading transactions and assets.

Note

The completed code for this lab can be found at `$COURSE_HOME/lab_05/lab_05c/solution/springtrader-portfolio`.

15.1. Exploring springtrader-portfolio

1. Change to the lab directory:

```
$ cd $COURSE_HOME/lab/lab_05/lab_05c/initial/springtrader-portfolio
```

   and import the project (via `pom.xml`) into your IDE of choice.

2. Explore the new `portfolio` microservice:

   - How many domain classes do you now have? Look are the relationship between `Portfolio`, `Holder`, `Order`
   - Open the class `PortfolioService`. See how it relies on the `quotes` and `accounts` microservices? As a consequence, the `portfolio` microservice won't work if the other two microservices are not up and running.

3. Add the following to `application.yml`. As you can see, each of our microservices runs on a different port so they can all run on your local computer.

```
spring:
  profiles.active: local
  application:
     name: portfolio

spring.jpa.hibernate.ddl-auto: create

logging:
  level:
    io.pivotal.springtrader: 'INFO'

---
spring:
  profiles: local

server:
  port: 8081

quotes:
  url: http://localhost:8086
```



```
accounts:
  url: http://localhost:8082

---
spring:
  profiles: cloud

quotes:
  url: http://quotes-nonliteral-nautilus.cfapps.io

accounts:
  url: http://accounts-dormant-dewlap.cfapps.io
```

4. Open `src/main/resources/import.sql`. As seen before, Spring Boot automatically loads this file at startup time. You will therefore have one order created at application startup time.

5. To run the application locally you have to add H2 dependency to your `pom.xml` file.

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

6. Make sure the `accounts` and `quotes` microservices are still running

7. Run the application and open the url http://localhost:8081/portfolio/johnsmith in your web browser. You should see the following:

```
{
  "accountId":"johnsmith",
  "name":"JohnSmith",
  "currentTotalValue":-255.45,
  "purchaseValue":0,
  "sellValue":100.00,
  "holdings":{
    "EMC":{"id":null,
        "symbol":"EMC",
        "quantity":-10,
        "purchaseValue":0,
        "sellValue":100.00,
      "orders":
        [{
          "orderId":1,
          "accountId":"johnsmith",
          "symbol":"EMC",
          "orderFee":1.00,
          "completionDate":1481468400000,
          "orderType":"SELL",
          "price":10.00,
          "quantity":10
          }],
      "currentValue":25.545
      }
  }
}
```

15.2. Preparing for Cloud Foundry

1. We want to bind to a MongoDB data source when running on Cloud Foundry, so let's create one:

```
$ cf cs p-mysql 100mb-dev springtrader-portfolio-db
Creating service instance springtrader-portfolio-db in org pivot-cqueiroz / space development as cqueiroz@pivotal.io...
OK
```


2. Add the appropriate dependencies for the Spring Cloud Connectors:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-spring-service-connector</artifactId>
</dependency>
```

3. Build the JAR:

```
$ mvn clean package
```

15.3. Deploying to Cloud Foundry

1. Create an application manifest in `manifest.yml`:

```
timeout: 180
instances: 1
memory: 512M
env:
    SPRING_PROFILES_ACTIVE: cloud
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
applications:
- name: portfolio
  random-route: true
  path: target/lab_05c-portfolio-1.0.0-SNAPSHOT.jar
  services: [ springtrader-portfolio-db ]
```

2. Before push to Cloud Foundry adjust the application.yml (cloud profile) file with the URLs for the Accounts and Quotes services. Something like:

```
...
...
spring:
  profiles: cloud
  jpa:
     hibernate:
        ddl-auto: create

quotes:
  url: http://quotes-random-words1.cfapps.io

accounts:
  url: http://accounts-random-words2.cfapps.io
```

3. Push to Cloud Foundry:

```
$ cf push

...
```

4. Access the application on http://portfolio<your-random-words>.cfapps.io/portfolio/johnsmith. You should see the following:

```
{
   "accountId":"johnsmith",
   "name":"JohnSmith",
   "currentTotalValue":255.450,
   "purchaseValue":100.00,
   "sellValue":0,
   "holdings":{
```



```
      "EMC":{
         "id":null,
         "symbol":"EMC",
         "quantity":10,
         "purchaseValue":100.00,
         "sellValue":0,
         "orders":[
            {
               "orderId":1,
               "accountId":"johndoe",
               "symbol":"EMC",
               "orderFee":1.00,
               "completionDate":1329759342904,
               "orderType":"BUY",
               "price":10.00,
               "quantity":10
            }
         ],
         "currentValue":25.545
      }
   }
}
```


Chapter 16. Lab 05d - Run the Web User Interface

Estimated time to complete: 30 minutes

In this lab we'll finish our complete cloud-native application architecture: *SpringTrader*, by running the Web User Interface (UI). This application is already complete and will delegate to the three microservices you have worked on in the rest of this session .

Note

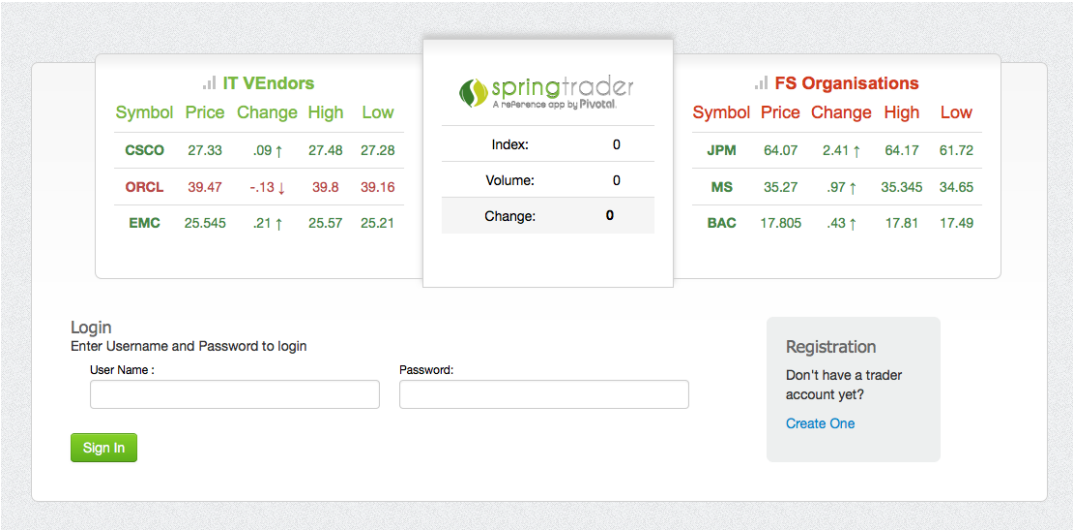
The code for this lab can be found at: `$COURSE_HOME/lab_05/lab_05d/initial/spring-trader-web`.

16.1. Exploring springtrader-portfolio

Please make sure all three microservices from the previous three labs - quotes, accounts and portfolios are still running. Test this by opening each of these in your browser:

1. http://localhost:8086/quote/msft
2. http://localhost:8082/account/1
3. http://localhost:8081/portfolio/johnsmith
4. Import the project `spring-trader-web` into your IDE of choice.
5. Run as a Java application by launching the class `WebApplication`
6. Once it is running, open your browser to http://localhost:8080

You should see something like this. If there are no stock quotes showing, the system is not working.



16.2. Running the Application

1. The first thing you need to do is login. Two accounts exist already:

   - `johnsmith` (password: `123`)
   - `pauljones3` (password: `123`)

Or you can create your own account - on the right-hand side see the *Registration* box. Click the `Create One` link. Enter any details you like and most important: give yourself a few million dollars to play with!

Note

Once you have an account you can start trading. If you get an error "No instances available for accounts", the Web application hasn't found the *accounts* microservice.

16.3. Deploying to Cloud Foundry

1. There is already an application manifest

`$COURSE_HOME/lab_05/lab_05d/solution/spring-trader-web/manifest.yml`:

```
timeout: 180
instances: 1
memory: 512M
env:
    SPRING_PROFILES_DEFAULT: cloud
    JAVA_OPTS: -Djava.security.egd=file:///dev/urandom
applications:
- name: webtrader
  random-route: true
  path: target/web-1.0.0-SNAPSHOT.jar
  services: [ service-registry ]
```

However, before we can push to Cloud Foundry we need to tell the application where to find its microservices.

In your IDE or a suitable editor, open `src/main/resources/application.yml`. It looks like this:

```
spring:
  profiles.active: local
  application:
     name: web

---
spring:
  profiles: local

  jpa:
     hibernate:
        ddl-auto: create-drop
server:
  port: 8080

quotes:
  url: http://localhost:8086

accounts:
  url: http://localhost:8082

portfolios:
  url: http://localhost:8081

---
spring:
  profiles: cloud
  jpa:
     hibernate:
        ddl-auto: update

quotes:
  url: http://quotes-xxxx-yyyy.cfapps.io

accounts:
  url: http://accounts-xxxx-yyyy.cfapps.io

portfolios:
  url: http://portfolios-xxxx-yyyy.cfapps.io
```

In the last section (`spring.profiles: cloud`), you need to change the url properties for the URLs of the three microservices you previously pushed to Cloud Foundry.

Make sure these applications are still running in Cloud Foundry.

2. Push to Cloud Foundry:

```
$ cf push

...
```

3. Access the application on `http://webtrader<your-random-words>.cfapps.io`.

Chapter 17. Lab 06a - Deploying and Using Spring Cloud Config Server

Estimated time to complete: 30 minutes

Now we'll begin using the components found in Spring Cloud to implement patterns for distributed systems. We'll be using Spring Cloud Config to centralize configuration for applications. Our configuration will be stored in a local folder, and served to our applications using the Spring Cloud Config Server.

In this lab, we'll do the following:

1. Create a folder to store your configuration
2. Create a Spring Cloud Config Server and test it with a basic sample application

Note

before starting this lab, you can stop all the applications you had started in the previous labs. You will not need them anymore.

This lab uses 4 different `properties` files, as described in this table:

File name	Location	Purpose
`demo.properties`	Config folder	Configuration file for application `demo`
`demo-dev.properties`	Config folder	`dev` profile configuration for application `demo`
`bootstrap.properties`	springtrader-config-client	Information sent to config server before application starts
`application.properties`	all Spring Boot projects	Spring Boot configuration

17.1. Create the config folder to Store Configuration

1. Change to the lab directory:

```
$ cd $COURSE_HOME/lab/lab_06/lab_06a/initial/
```

2. Create a directory for the configuration files

```
mkdir config
cd config
```

3. Create `application.properties`

```
greeting=Hello world from application.properties
val_1=value from application.properties
```

4. Create `demo.properties`

```
greeting=Hello world from demo.properties
val_2=value from demo.properties
```

5. Create `demo-dev.properties`

```
greeting=Hello world from demo-dev.properties
val_3=value from demo-dev.properties
```

17.2. Create a Spring Cloud Config Server

1. Change to the lab directory:

```
$ cd $COURSE_HOME/lab/lab_06/lab_06a/initial/springtrader-config-server
```

   and import the project (via `pom.xml`) into your IDE of choice.

2. Open `pom.xml`. As you can see, your project has been configured for Spring Cloud already. Add a dependency on `spring-cloud-config-server`:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

3. Now open `io.springtrader.configserver.SpringtraderConfigServerApplication` and add the `@EnableConfigServer` annotation:

```
@SpringBootApplication
@EnableConfigServer  // <--- ADD THIS!
public class SpringtraderConfigServerApplication {

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

4. Now open `src/main/resources/application.properties` and paste the following code:

```
server.port=9000
spring.cloud.config.server.native.searchLocations=file:../config/
```



```
spring.profiles.active=native
```

5. Run the application

6. Test the application in your web browser by connecting to http://localhost:9000/application/default You should be able to see your properties displayed similar to this:

```
{
  "name": "application",
  "profiles": [
    "default"
  ],
  "label": "master",
  "propertySources": [{
    "name": "file:../config/application.properties",
    "source": {
      "val_1": "value from application.properties",
      "greeting": "Hello world from application.properties"
    }
  }]
}
```

   As shown above, you should be able to see the `greeting` value that you have entered inside `application.properties`.

7. Now experiment with following links, pay attention to the value of `greeting` key

   - http://localhost:9000/application-default.properties
   - http://localhost:9000/demo-default.properties
   - http://localhost:9000/demo-dev.properties

Note

`http://localhost:9000/<myApp>/default` displays properties for all managed applications.

Leave the Config Server running, the client will need it.

17.3. Create the Sample Test Application

1. Change to the lab directory:

```
$ cd $COURSE_HOME/lab/lab_06/lab_06a/initial/springtrader-config-client
```

   and import the project (via `pom.xml`) into your IDE of choice.

2. Open `pom.xml` and add the following dependency:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```


Note

`spring-cloud-starter-config` comes with `spring-cloud-config-client` as a transitive dependency

1. In the package `io.springtrader.configclient` create the class `GreetingController`, and into that class paste the following source code:

```
@RestController
public class GreetingController {

    @Value("${greeting}")
    String greeting;

    @RequestMapping("/")
    public String greeter() {
        return greeting;
    }

}
```

2. Inside `src/main/resources` open the `bootstrap.properties` file and set the config server's uri:

```
spring.cloud.config.uri=http://localhost:9000
```

3. Make sure the `config server` is still running. Launch the `config-client` application. Test the client has succeeded in fetching its configuration by using your web browser to connect to http://localhost:8080. You should be able to see a "Hello World" message displayed.

```
Hello world from application.properties
```

4. Add the application name to `bootstrap.properties`

```
spring.cloud.config.uri=http://localhost:9000
spring.application.name=demo
```

5. Restart the client and check that the hello world message has been changed

```
Hello world from demo.properties
```

6. Add the `dev` profile to `bootstrap.properties`

```
spring.cloud.config.uri=http://localhost:9000
spring.application.name=demo
spring.profiles.active=dev
```

7. Restart the client and check that the hello world message has been changed again

```
Hello world from demo-dev.properties
```

8. Take a look at the Spring Environment to see how the `greeting` property is being resolved. You can connect to http://localhost:8080/env The information you need is on the first line.

17.4. [Bonus] Use github to store the config files

1. Create a github repository
2. Push the config folder to the github repository (make sure the properties files are at the root of the repository, for example https://github.com/pivotal-education/springtrader-config-repo)
3. Change the config server's `application.properties` to

```
server.port=9000
spring.cloud.config.server.git.uri= https://github.com/<YourUserName>/<RepoName>.git
```

4. Restart the config server
5. Restart the client to see the values are properly loaded
6. Push some changes to github
7. Restart the client again to check the changes

Chapter 18. Lab 06b - Leveraging Eureka for Service Discovery via Spring Cloud Netflix

Estimated time to complete: 30 minutes

Let’s continue learning the components found in Spring Cloud to implement patterns for distributed systems. We’ll use Spring Cloud Netflix to deploy Eureka, which is a component offering service registration and discovery.

In this lab, we’ll do the following:

1. Create a Eureka server
2. Create two applications, a `producer` and a `consumer`, wire them up using Eureka, and test them.

18.1. Introduction

1. Before starting this lab, you can stop all the web applications that you had started in the previous labs. You will not need them anymore.
2. There are four different web applications to be run locally in this lab. To avoid any confusion, we have listed all the port numbers below:

Application	**port number**
Eureka Server	8761
Producer	8080 (default)
Consumer	8091

18.2. Creating a Eureka Server

1. Change to the lab directory:

```
cd $COURSE_HOME/lab/lab_06/lab_06b/initial/
```


and import all 4 projects into your IDE. For the first part of this lab, we will work on `springtrader-eureka`

2. Open `pom.xml`, change the parent POM to the `spring-cloud-starter-parent`:

```
<parent>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-parent</artifactId>
   <version>Brixton.RELEASE</version>
</parent>
```

3. Add a dependency on `spring-cloud-starter-eureka-server`:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

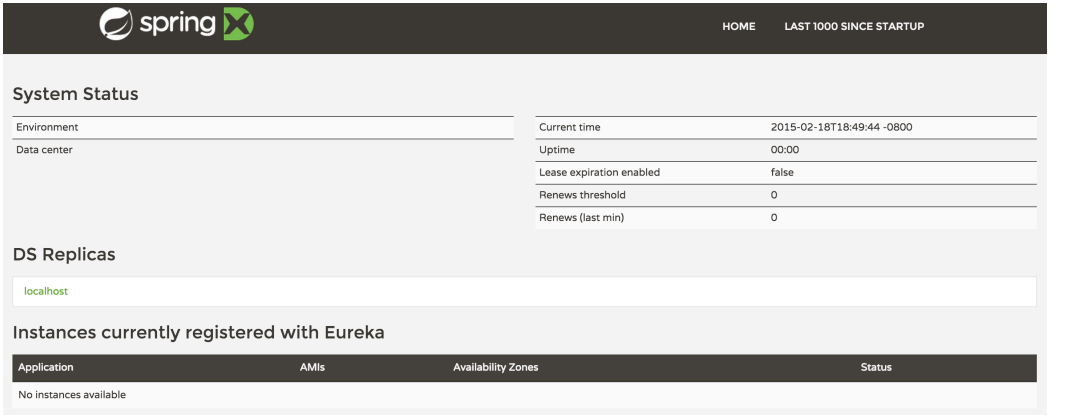
4. In `io.springtrader.eureka.SpringtraderEurekaApplication`, add the `@EnableEurekaServer` annotation:

```
@SpringBootApplication
@EnableEurekaServer    // <-- ADD THIS!
public class SpringtraderEurekaApplication {

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

5. Open `src/main/resources/application.yml` and paste in the following source:

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

6. Run the application and visit http://localhost:8761
   You should see the below console:



18.3. Create and Register the Producer Service

1. In your IDE, open the `lab_06b-springtrader-producer` project

2. Open `pom.xml`, change the parent POM to the `spring-cloud-starter-parent`:

```
<parent>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-parent</artifactId>
   <version>Brixton.RELEASE</version>
</parent>
```

3. Add a dependency on `spring-cloud-starter-eureka`:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

4. In the package `io.springtrader.producer`, create the class `ProducerController`. Into that class paste the following code:

```
@RestController
public class ProducerController {

    private org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(ProducerController.class);
    private AtomicInteger counter = new AtomicInteger(0);

    @RequestMapping("counter-value")
    public Object produce() {
        int value = counter.getAndIncrement();
        log.info("Produced a value: {}", value);

        return Collections.singletonMap("value", value);
    }

}
```


5. Now open `io.springtrader.producer.SpringtraderProducerApplication` and add the `@EnableDiscoveryClient` annotation:

```
@SpringBootApplication
@EnableDiscoveryClient  // <--- ADD THIS!
public class SpringtraderProducerApplication {

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

6. Create the file `src/main/resources/application.yml` and paste in the following source:

```
spring:
  application:
    name: producer

server:
  port: 9009

eureka:
  instance:
    preferIpAddress: true
    leaseRenewalIntervalInSeconds: 10
    metadataMap:
      instanceId: ${server.port}
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

7. Run the producer application

8. Test your producer application http://localhost:9009/counter-value refresh several times to see the counter value moves up

9. Ten seconds after the producer application finishes startup, you should see it log its registration with Eureka:

```
DiscoveryClient_PRODUCER/192.168.1.10:9009 - Re-registering apps/PRODUCER
DiscoveryClient_PRODUCER/192.168.1.10:9009: registering service...
DiscoveryClient_PRODUCER/192.168.1.10:9009 - registration status: 204
```

You should also be able to refresh http://localhost:8761 in the browser and see the producer registered:

18.4. Create and Register the Consumer Service

1. In your IDE, open `lab_06b-springtrader-consumer`

2. Open its `pom.xml` file and change the parent POM to `spring-cloud-starter-parent`:

```
<parent>
   <groupId>org.springframework.cloud</groupId>
```



```
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.RELEASE</version>
</parent>
```

3. Add a dependency on `spring-cloud-starter-eureka`:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

4. In the package `io.springtrader.consumer`, create the class `ProducerResponse`, and into that class paste the following code:

```
public class ProducerResponse {
    private int value;

    public void setValue(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

5. Also in the package `io.springtrader.consumer.controller`, create the class `ConsumerController`, and into that class paste the following code:

```
@RestController
public class ConsumerController {

  @Autowired
  DiscoveryClient discoveryClient;

  RestTemplate restTemplate = new RestTemplate();

  @RequestMapping("/")
  Object consume2() {
      ServiceInstance instance = discoveryClient.getInstances("PRODUCER").get(0);

      ProducerResponse response = restTemplate.getForObject(
              instance.getUri() + "/counter-value", ProducerResponse.class);

      return Collections.singletonMap("value", response.getValue());
  }

}
```

6. Now open `io.springtrader.consumer.SpringtraderConsumerApplication` and add the `@EnableDiscoveryClient` annotation:

```
@SpringBootApplication
@EnableDiscoveryClient  // <--- ADD THIS!
public class SpringtraderConsumerrApplication {

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

7. In `src/main/resources/application.yml`, set the `server.port` property:

```
spring:
  application:
    name: consumer

server:
  port: 8091
```



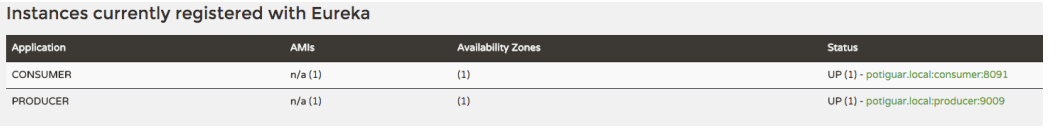
```
eureka:
  instance:
    preferIpAddress: true
    leaseRenewalIntervalInSeconds: 10
    metadataMap:
      instanceId: ${server.port}
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

8. Run the consumer application

9. Ten seconds after the consumer application finishes startup, you should see it log its registration with Eureka:

```
DiscoveryClient_CONSUMER/192.168.1.10:8091 - Re-registering apps/CONSUMER
DiscoveryClient_CONSUMER/192.168.1.10:8091: registering service...
DiscoveryClient_CONSUMER/192.168.1.10:8091 - registration status: 204
```

   You should also be able to refresh http://localhost:8761 in the browser and see the producer registered:



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONSUMER	n/a (1)	(1)	UP (1) - potiguar.local:consumer:8091
PRODUCER	n/a (1)	(1)	UP (1) - potiguar.local:producer:9009

10. Open a browser tab on the consumer application (http://localhost:8091). It should show that it is receiving values from the producer:

```
{
  "value":0
}
```


Chapter 19. Lab 07a - Client-Side Load Balancing with Ribbon

Estimated time to complete: 30 minutes

Let's continue learning the components found in Spring Cloud to implement patterns for distributed systems. We'll again use Spring Cloud Netflix to implement client-side load balancing with Ribbon.

In this lab, we'll do the following:

1. Change the consumer application fromdiscovery lab to:
   a. Use a `LoadBalancerClient`
   b. Use a `RestTemplate` configured to resolve service names from Ribbon
2. Test the new consumer versions against our local pool of producers
3. Try swapping out the Ribbon load balancing algorithm

Note

before starting this lab, you can stop all the applications you had started in the previous labs. You will not need them anymore.

This lab uses 4 running applications. Here is a table that you can refer to during the lab.

Eureka	http://localhost:8761
Producer 1	http://localhost:8080
Producer 2	http://localhost:8082
Consumer	http://localhost:8091

19.1. Setup

1. Four projects have been placed inside `$COURSE_HOME/lab_07/lab_07a/initial/`. Import all of them inside your IDE of choice.

2. Eureka: Open `springtrader-eureka` and launch the class `SpringtraderEurekaApplication`

3. Connect to http://localhost:8761/ to confirm that Eureka has been started successfully.

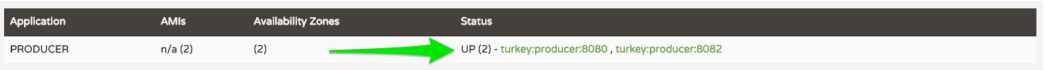
4. A completed `springtrader-producer` project has been placed in `$COURSE_HOME/lab_07/lab_07a/initial/springtrader-producer` for your convenience. Change to that directory and build

```
cd $COURSE_HOME/lab/lab_07/lab_07a/initial/springtrader-producer
mvn clean package
```

5. Now run the producer twice (on different ports), each in a different terminal/command window:

```
Linux, MacOS:
$ SERVER_PORT=8080 java -jar target/lab_07a-springtrader-producer-0.0.1-SNAPSHOT.jar
$ SERVER_PORT=8082 java -Deureka.instance.metadataMap.instanceId=p8082 -jar target/lab_07a-springtrader-producer-0.0.1
    -SNAPSHOT.jar

Windows:
> set SERVER_PORT=8080
> java -jar target/lab_07a-springtrader-producer-0.0.1-SNAPSHOT.jar
> set SERVER_PORT=8082
> java -Deureka.instance.metadataMap.instanceId=p8082 -jar target/lab_07a-springtrader-producer-0.0.1-SNAPSHOT.jar
```

6. Ensure you have two instances of the producer service registered in Eureka on http://localhost:8761 :



19.2. Using the `LoadBalancerClient`

1. Inside your IDE, open the project `springtrader-consumer`

2. Where is Ribbon?
   You might be interested to know how your Ribbon Client has been included inside your project so it can be used by our Consumer. Inside your `pom.xml` file, you will find the following dependency:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

The `spring-cloud-starter-eureka` dependency transitively comes with the `Ribbon` libraries (you will find them in your list of `jar` files).

1. In `io.springtrader.consumer.ConsumerController`, include the `consume()` method as follows:

```
@RequestMapping(value="/", produces="application/json")
public String consume() {
    ProducerResponse response = restTemplate.getForObject("http://producer", ProducerResponse.class);
    return String.format("{\"server port\":%d, \"value\":%d}", response.getServerPort(), response.getValue());
}
```

2. Config `RestTemplate`, add following codes to `SpringtraderConsumerApplication`

```
@LoadBalanced
@Bean
RestTemplate restTemplate() {
    return new RestTemplate();
}
```

3. Run your application by launching `SpringtraderConsumerApplication`.

4. Connect to http://localhost:8091/ and access the consumer application several times. Per the round-robin algorithm, 50% if the requests should be served by producer 8080 (and the other half should go to producer 8082).

```
{
   "server port":8082,
   "value":4
}
```

19.3. Failing one of the producers

You now have a Load Balancer that works across 2 Producers. But what happens when one of the Producers fails?

1. Let's simulate a malfunctioning producer. Open the link http://localhost:8080/setError/true, this will make the http://localhost:8080/ always returns an error.

2. Now try to access the consumer a few times. What can you notice?

3. In the next lab, you will learn how to use a `circuit breaker` so failures can be handled gracefully.

Chapter 20. Lab 07b - Fault-Tolerance with Hystrix

Estimated time to complete: 20 minutes

Let's continue learning the components found in Spring Cloud to implement patterns for distributed systems. We'll again use Spring Cloud Netflix to implement client-side load balancing with Ribbon. However, this time we'll add an implementation of the Circuit Breaker pattern using Netflix Hystrix.

In this lab, we'll do the following:

1. Refactor the consumer application from theLoad Balancing lab to add a `ProducerClient`.
2. Test the new consumer version against our local pool of producers
3. Kill one of the producers and watch the consumer failover.
4. Set the remaining producer into the error state, and watch the circuit breaker fallback behavior.
5. Kill the remaining producer and watch the circuit breaker fallback behavior.
6. Restart a producer and watch the consumer recover.

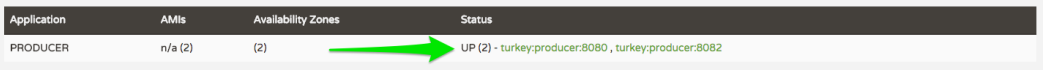
20.1. Setup

1. You can just reuse the Eureka server and the 2 producers from Lab 07a as shown below:

Eureka	http://localhost:8761
Producer 1	http://localhost:8080
Producer 2	http://localhost:8082

In case you did not have time to complete the lab 07a, you can run all 3 application instances from lab 07a's `complete` folder.

1. Ensure you have two instances of the producer service registered in Eureka (http://locahost:8761):



20.2. Using Hystrix

Note

all the failover configuration is done on the `consumer` side (as opposed to the `producer` side. In case the `producer` fails, the `consumer` should know where to fall back to.

1. Inside your IDE, go into the `springtrader-consumer` project.

2. Open its `pom.xml` file and add the following dependency:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

3. You now need to add the relevant configuration for your `Circuit Breaker`. We will rely on Spring Boot's default configuration.
   Inside the class `io.springtrader.consumer.SpringtraderConsumerApplication`, add the annotation `@EnableCircuitBreaker`.

4. Config `RestTemplate`, add following codes to `SpringtraderConsumerApplication`

```
@LoadBalanced
@Bean
RestTemplate restTemplate() {
    return new RestTemplate();
}
```

5. Create the class `io.springtrader.consumer.ProducerClient` and into it paste the following code:

```
@Component
public class ProducerClient {

    @Autowired
    @LoadBalanced
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "getProducerFallback")
    public ProducerResponse getValue() {
        return restTemplate.getForObject("http://producer", ProducerResponse.class);
    }

    public ProducerResponse getProducerFallback() {
        ProducerResponse response = new ProducerResponse();
        response.setValue(-1);
        response.setServerPort(-1);
        return response;
    }
}
```


6. Refactor the class `io.springtrader.consumer.ConsumerController` to autowire the `ProducerClient` instead of `RestTemplate`, and then use it to obtain the `ProducerResponse`:

```
@Autowired
ProducerClient client;

@RequestMapping(value = "/", produces = "application/json")
String consume() {
    ProducerResponse response = client.getValue();
    return String.format("{\"value\":%d}", response.getValue());
}
```

7. Inside `src/main/resources/application.yml` add the following:

```
producer.ribbon.ServerListRefreshInterval: 5000
```

8. Start the application by launching `SpringtraderConsumerApplication`

In your logs, you should see that Hystrix has been loaded:

```
c.n.h.c.m.e.HystrixMetricsPoller         : Starting HystrixMetricsPoller
ration$HystrixMetricsPollerConfiguration : Starting poller
```

1. Test the consumer application on http://localhost:8091 and show that it is still receiving values from the producers. Also, watch the producers and observe that Ribbon is performing round robin load balancing across the two producers

20.3. Failover

1. Shut down one of the two producer processes.

2. Reload url, test the consumer application and show that it is still receiving values from one of the producers. You may get the fallback value of `-1` a few times due to the lag in removing the failed instance from the Ribbon cache. Eventually it will converge to hitting the only remaining healthy instance.

20.4. Fallback

1. Send the remaining producer into the error state by visiting `/setError/true`, now access the consumer, you should **always** see the fallback value `-1`.

2. Shut down the remaining producer process.

3. Reload url, test the consumer application and show that it is only emitting the fallback value of `-1`.

20.5. Recovery

1. Restart one of the producer processes. Wait for it to register with Eureka.
2. Reload url, test the consumer application and show that eventually recovers and starts hitting the new producer process. This can take several seconds as the Eureka and Ribbon caches repopulate.

20.6. Next Steps

Do not shut anything down when you complete this lab. We will add one additional component in next lab

Chapter 21. Lab 07c - Monitoring Circuit Breakers with Hystrix Dashboard

Estimated time to complete: 20 minutes

We will now monitor what's happening in our Circuit Breakers by using Netflix Hystrix Dashboard.

In this lab, we'll do the following:

1. Run everything that we created in the previous lab.
2. Create a Hystrix Dashboard
3. Kill both of the producers and watch the circuit open in the Dashboard.
4. Restart a producer and watch the circuit close in the Dashboard.

21.1. Setup

1. Make sure you've completed the previous lab and have everything still running.

21.2. Building the Hystrix Dashboard

1. The project `springtrader-hystrix-dashboard` has been placed inside `$COURSE_HOME/lab_07/lab_07c/initial/`. Import it inside your IDE of choice.
2. Add the following dependency to `pom.xml`:

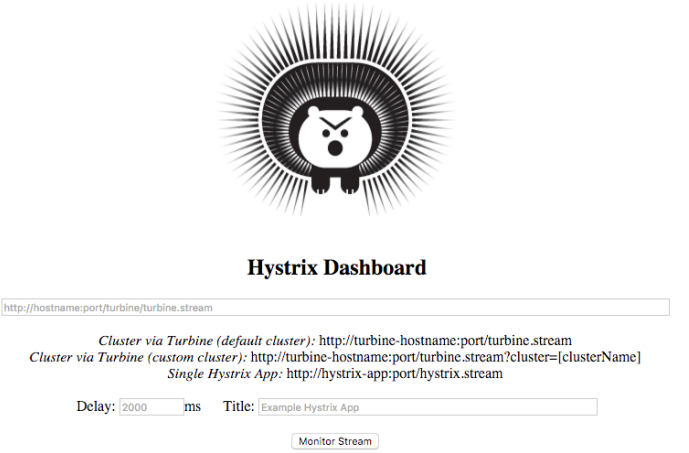
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

3. Add a `@EnableHystrixDashboard` to the class `io.pivotal.springcloud.hystrixdashboard.HystrixDashboardApplication`.
4. Paste the following inside `src/main/resources/application.yml` :

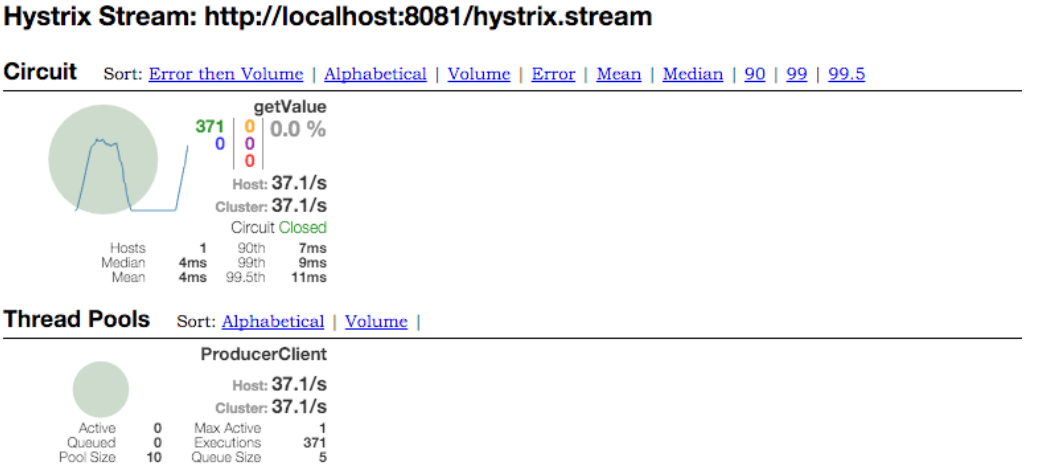
```
server:
  port: 7979
```


5. Run the application by launching `HystrixDashboardApplication`

6. Now, access the Hystrix Dashboard at http://localhost:7979/hystrix. Enter `http://localhost:8091/hystrix.stream` into the text field and click the ``Monitor Stream."

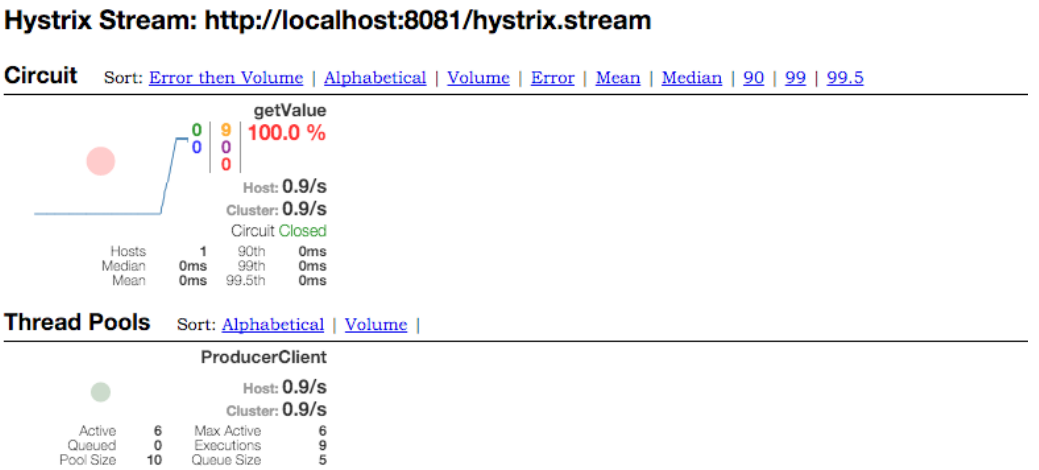


7. Access the consumer application at http://localhost:8091/ several times and check that the circuit breaker is registering successful requests.



21.3. Fallback

1. Shut down both of the producer processes.
2. Test the consumer application and show that it is only emitting the fallback value of `-1`.
3. You should also see the circuit breaker is registering short-circuited requests.



21.4. Recovery

1. Restart one of the producer processes. Wait for it to register with Eureka.
2. Test the consumer application and show that eventually recovers and starts hitting the new producer process. This can take up to one minute as the Eureka and Ribbon caches repopulate.
3. Continue to watch the dashboard. You should also see the circuit breaker dashboard show the recovery.

Chapter 22. Lab 07d - Declarative REST Clients with Feign

Estimated time to complete: 15 minutes

This lab is based on the Load Balancing lab. We will enhance it to use a declarative REST repository based on Feign.

We will do the following:

1. Improve the consumer application from the Load Balancing lab to use a `ProducerClient` enabled by Feign
2. Test the new consumer version against our local pool of producers

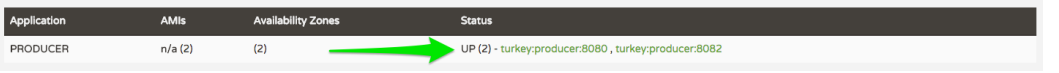
22.1. Setup

1. Stop all the running applications and remove the corresponding projects from your IDE
2. Inside your IDE, import the projects that have been placed for you inside $COURSE_HOME/lab_07/lab_07d/initial/

Note

those projects are identical to the ones that you had completed at the end of the Lab 07a (Load Balancing with Ribbon).

1. Start your Eureka server and 2 producers as we have done in the previous labs. You SHOULD NOT start the consumer as of now. The below applications should be running:
2. Once you are ready to go, ensure you have two instances of the producer service registered in Eureka (http://locahost:8761):



22.2. Using Feign

1. Open your `springtrader-consumer` project inside your IDE. Add the following dependency to `pom.xml`:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

2. Add a `@EnableFeignClients` annotation to the class `io.springtrader.consumer.SpringtraderConsumerApplication`.

3. Create the interface `io.springtrader.consumer.ProducerClient` and into it paste the following code:

```
@FeignClient("producer")
public interface ProducerClient {

    @RequestMapping(method = RequestMethod.GET, value = "/")
    ProducerResponse getValue();
}
```

4. Refactor the class `io.springtrader.consumer.ConsumerController` to autowire the `ProducerClient` instead of `RestTemplate`, and then use it to obtain the `ProducerResponse`:

```
@Autowired
    ProducerClient client;

    @RequestMapping(value="/", produces="application/json")
    public String consume() {


        ProducerResponse response = client.getValue();
        return String.format("{\"server port\":%d, \"value\":%d}", response.getServerPort(), response.getValue());

    }
```

5. Start the `springtrader-consumer` application by launching `SpringtraderConsumerApplication`

6. Test the consumer application on http://localhost:8091/ and show that it is still receiving values from the producers. It should behave exactly like in the lab 07a. The only difference is that your REST binding code is slightly more concise and elegant.

Chapter 23. Lab 08a - Creating an OAuth2 Authorization Server

[abstract]

We will now create an OAuth 2.0 authorization server. Using spring cloud security support we will see how easy it is to create a working authorization server with very little code needed

23.1. Default behaviour

1. The project `lab_08a-springtrader-auth-server` has been placed inside `$COURSE_HOME/lab_08/lab_08a/initial/`. Import it inside your IDE of choice.

2. Run the application by running `SpringtraderAuthServerApplication`

See what happens before you setup your authentication server. Run the following command:

```
curl http://localhost:8000/auth-server/oauth/token \
    -u acme:acmesecret \
    -d grant_type=password \
    -d username=mstine \
    -d password=secret
```

You should see the following output:

```
{
   "timestamp":1463732463298,
   "status":405,
   "error":"Method Not Allowed",
   "exception":"org.springframework.web.HttpRequestMethodNotSupportedException",
   "message":"Request method 'POST' not supported",
   "path":"/auth-server/oauth/token"
}
```

23.2. Configuring the dependencies

1. Add the following dependencies to your pom.xml file

```
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
  </dependency>
```


23.3. Enabling the auth server

1. Update `SpringtraderAuthServerApplication` as shown below:

```
@SpringBootApplication
@EnableAuthorizationServer //// (1)
public class SpringtraderAuthServerApplication {

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

    @Configuration
    protected static class OAuth2Config extends AuthorizationServerConfigurerAdapter {

        @Autowired
        private AuthenticationManager authenticationManager;

        @Override
        public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {
            oauthServer.checkTokenAccess("permitAll()"); //// (4)
        }

        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
            endpoints.authenticationManager(authenticationManager);
        }

        @Override
        public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

            clients.inMemory()
                    .withClient("acme")
                    .secret("acmesecret")
                    .authorizedGrantTypes("authorization_code", "refresh_token",
                            "password").scopes("openid"); //// (2)
        }

    }

    @Configuration
    protected static class AuthenticationManagerConfiguration extends GlobalAuthenticationConfigurerAdapter {

        @Override
        public void init(AuthenticationManagerBuilder auth) throws Exception {
            auth.inMemoryAuthentication()
                    .withUser("mstine").password("secret").roles("USER", "ADMIN").and()
                    .withUser("littleidea").password("secret").roles("USER", "ADMIN").and()
                    .withUser("starbuxman").password("secret").roles("USER", "ADMIN"); //// (3)
        }

    }

}
```

1. This bootstraps the auth server by creating Spring filters to handle the token credentials that are passed over requests
2. Register a single client
3. Register several users, other Authorization managers should be used such as JDBC or LDAP based here
4. enable `/oauth/check_token`

23.4. Testing your Oauth server

1. Restart your application

To test the authentication server execute again a simple POST to the `/auth-server/oauth/token endpoint, notice that 1. `acme:acmesecret` is the client's credential 1. `mstine:secret` is the user's credential (Resource Owner in OAuth2 terms)

```
curl http://localhost:8000/auth-server/oauth/token \
    -u acme:acmesecret \
    -d grant_type=password \
    -d username=mstine \
    -d password=secret
```

You should get back an authentication token

```
{
   "access_token":"661aac97-55ca-49a0-b8b6-a4a1d8cb63de",
   "token_type":"bearer",
   "refresh_token":"9a605803-4013-4818-ae24-22de7b399018",
   "expires_in":43199,
   "scope":"openid"
}
```

Now let's do token verification against the authorization server. You can replace `<your_token_id>` with your value for `access_token`:

```
curl http://localhost:8000/auth-server/oauth/check_token?token=<your_token_id>
```

You should expecting to see the below:

```
{
   "exp":1463675798,
   "user_name":"mstine",
   "authorities":[
      "ROLE_ADMIN",
      "ROLE_USER"
   ],
   "client_id":"acme",
   "scope":[
      "openid"
   ]
}
```

You're all set! In the next lab, you will use the same mechanism to have the `Resource Server` verify your token.

Chapter 24. Lab 08b - Securing a Resource Server with Spring Cloud Security

Now that we have an authorization server from lab 08a, we will create a simple restful webservice and protect it using oauth tokens.

24.1. Running it

1. Import the project `springtrader-resource-server` inside your IDE
2. Run the corresponding application by launching `SpringtraderResourceServerApplication`
3. Access the endpoint on your browser http://localhost:8001/greeting and you should receive back a simple json output

```
{
  "content": "Hello user"
}
```

24.2. Securing it

1. Add the following dependencies to your pom.xml

```
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
  </dependency>
```

2. Modify you `application.yml`, it should look like this:

```
server:
   port: 8001

security:
  oauth2:
    resource:
      token-info-uri: http://localhost:8000/auth-server/oauth/check_token
    client:
      client-id: acme
      client-secret: acmesecret
```


Note

Recall that this is the url we use for the last step of the previous lab

3. Modify the `SpringtraderResourceServerApplication` class to enable OAuth2

```
@SpringBootApplication
@EnableResourceServer  // <-- Add this
public class SpringtraderResourceServerApplication {

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

}
```

   `@EnableResourceServer` will enable the proper Spring Security Filters to look for a token on the headers and validate that against the `token-info-url` to obtain the SecurityPrincipal

4. Restart your Resource Server

5. Access the endpoint again at http://localhost:8001/greeting . You should see the following message showing an unauthorized response:

```
<oauth>
  <error_description>
    Full authentication is required to access this resource
  </error_description>
  <error>unauthorized</error>
</oauth>
```

24.3. Obtain a valid token

Note

the rest of the lab uses `curl` because you will need to customize your request headers. If you have a browser plugin for that, you can choose to do without `curl`.

In order to access the resource now you will need a valid token from your Authorization Server. To do that execute the command:

```
curl http://acme:acmesecret@localhost:8000/auth-server/oauth/token \
    -d grant_type=password \
    -d username=mstine \
    -d password=secret
```

The result should be something like this:

```
{
   "access_token":"5d04666c-11c2-4a4c-9dae-f0def0acc9c1",
   "token_type":"bearer",
   "refresh_token":"361fa725-735f-4853-bb0f-5a8956a01b33",
   "expires_in":43199,
   "scope":"openid"
```



```
}
```

1. Grab the access_token from that result and now invoke the following command to call your service, replace the Authorization header with the proper token from the previous step:

```
curl -X GET -H "Authorization: bearer 5d04666c-11c2-4a4c-9dae-f0def0acc9c1" http://localhost:8001/greeting
```

This is all that is needed to protect a resource to require OAuth2 tokens to be accessed

Chapter 25. Lab 08c - Consuming a Secured Resource from OAuth2-Aware Client

Now that we have secured a resource in lab 08b, we will create a simple restful oauth2-aware client that will securely invoke the resource.

25.1. Create a OAuth2-Aware Client

1. Import the project `lab_08c-springtrader-client` to your IDE

2. Add the following dependency to your pom.xml

```
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
  </dependency>
```

3. Modify you `application.yml`, it should look like this:

```
security:
  oauth2:
    client:
      accessTokenUri: http://localhost:8000/auth-server/oauth/token
      userAuthorizationUri: http://localhost:8000/auth-server/oauth/authorize
      clientId: acme
      clientSecret: acmesecret
      scope: openid
      clientAuthenticationScheme: header
    sso:
      login-path: /login
  basic:
    enabled: false
```

4. Modify the `ClientApplication` class to handle and use the OAuth2 access token:

```
@SpringBootApplication
@EnableOAuth2Client
@RestController //// (1)
public class ClientApplication {

    @Autowired //// (2)
    private OAuth2RestOperations restTemplate;

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

    @RequestMapping("/")
    public String home() {
        return restTemplate.getForObject("http://localhost:8001/greeting", String.class);
    }
}
```

1. This annotation enable configuration for an OAuth2 client in a web application that uses Spring Security and wants to use the Authorization Code Grant from one or more OAuth2 Authorization servers.

2. Rest template that is able to make OAuth2-authenticated REST requests with the credentials of the

provided resource.

25.2. Test your client

1. Run your application by launching `ClientApplication`
2. Access the endpoint using a browser at http://localhost:8080/ and you get back a login screen. Enter 'littleidea' and 'secret'. It will ask you to approve or deny the request. Click approve to view the resource.

Note

`littleidea/secret` is one of the users that you have created in Lab 08a.

Congratulations! You now have secured your resources and used a client to securely invoke them.

Chapter 26. (Optional) Lab 09a - Deploying the StringTrader Microservices version Application

Note

This lab is designed only for full PCF installation

We will now deploy the SpringTrader application you have been developing. We will use the Web microservices (provided) together with the services you have already built and we will deploy them on Pivotal Cloud Foundry.

26.1. Copying the 3 services

1. Copy the 3 services: Account, Quotes and Portfolio into **initial** folder.

```
cd $COURSE_HOME/lab/lab_09/lab_09a/initial/

cp -R ../solution/quotes .
cp -R ../solution/accounts .
cp -R ../solution/portfolio .
```

26.2. Starting Spring Cloud Services on Pivotal Cloud Foundry

1. Create Eureka service (service-registry):

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

2. Initialise Eureka service by clicking on the service Manage link.

3. Create Hystrix Dashboard service (circuit-breaker-dashboard):

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

4. Initialise Hystrix dashboard by clicking on the service Manage link.

5. Make sure all manifest.yml files define the app memory with 1G.

```
memory: 1G
```

26.3. Deploying the Application

1. Create deploy.sh(bat) file.

```
cf push -f quotes/manifest.yml
cf push -f accounts/manifest.yml
cf push -f portfolio/manifest.yml
cf push -f web/manifest.yml
```

   a. If you are using Unix add **#!/bin/sh** at the first file line.

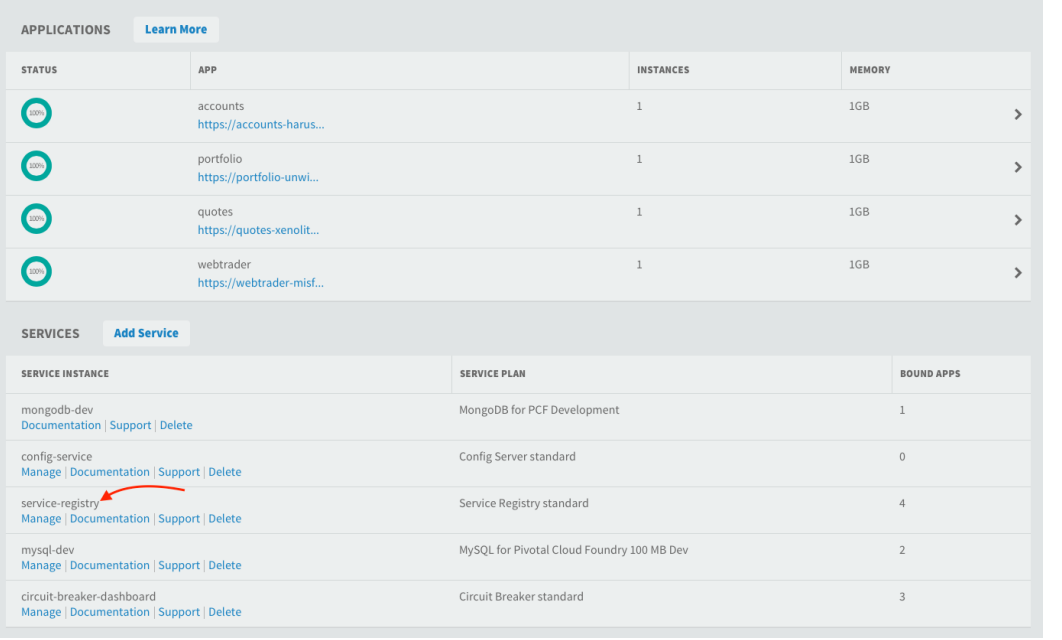
2. Run the deploy command.

```
./deploy.sh (or deploy.bat)
```

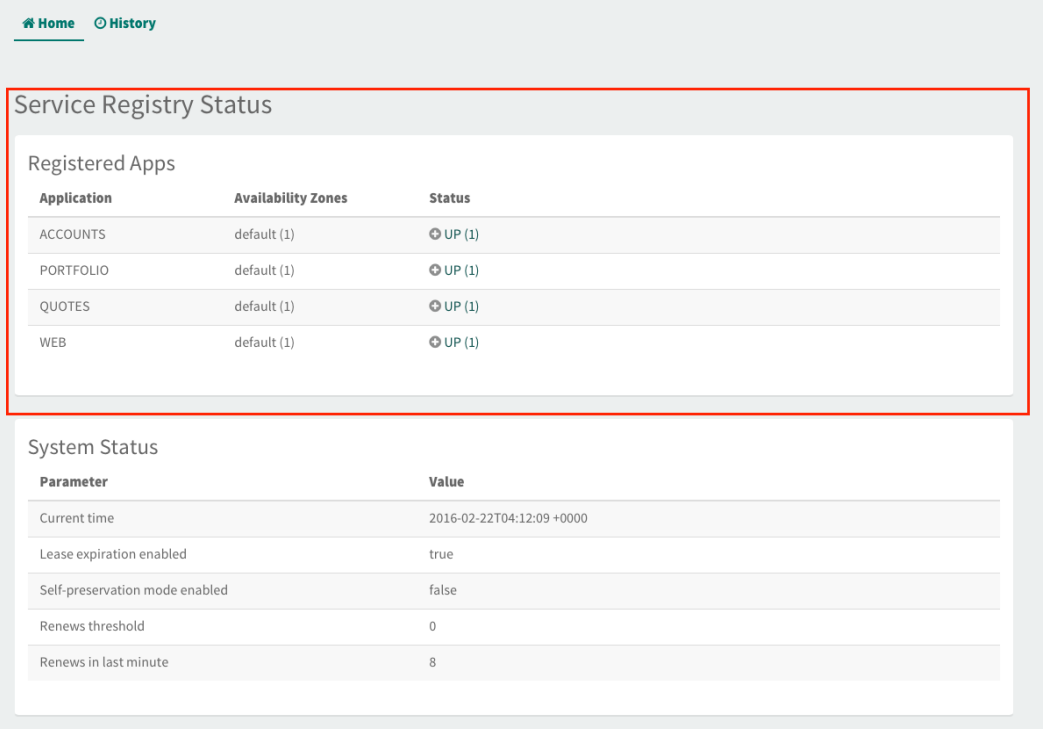
```
App started


OK

App webtrader was started using this command `CALCULATED_MEMORY=$($PWD/.java-buildpack/open_jdk_jre/bin/java-buildpack
    -memory-calculator-2.0.1_RELEASE -memorySizes=metaspace:64m.. -memoryWeights=heap:75,metaspace:10,native:10,stack:5
    -memoryInitials=heap:100%,metaspace:100% -totMemory=$MEMORY_LIMIT) && JAVA_OPTS="-Djava.io.tmpdir=$TMPDIR
    -XX:OnOutOfMemoryError=$PWD/.java-buildpack/open_jdk_jre/bin/killjava.sh $CALCULATED_MEMORY -Djava.security.egd=file://
    /dev/urandom" && SERVER_PORT=$PORT eval exec $PWD/.java-buildpack/open_jdk_jre/bin/java $JAVA_OPTS -cp $PWD/.:$PWD/.java
    -buildpack/spring_auto_reconfiguration/spring_auto_reconfiguration-1.10.0_RELEASE.jar org.springframework.boot.loader.Ja
    rLauncher`

Showing health and status for app webtrader in org instructor / space development as instructor...
OK

requested state: started
instances: 1/1
usage: 1G x 1 instances
urls: webtrader-nonbiological-bluebonnet.pcf12.cloud.fe.pivotal.io
last uploaded: Mon Feb 22 04:33:06 UTC 2016
stack: cflinuxfs2
buildpack: java_buildpack_offline

     state     since                    cpu    memory       disk          details
#0   running   2016-02-22 01:34:38 PM   0.0%   700K of 1G   27.8M of 1G
```

26.4. Testing it out

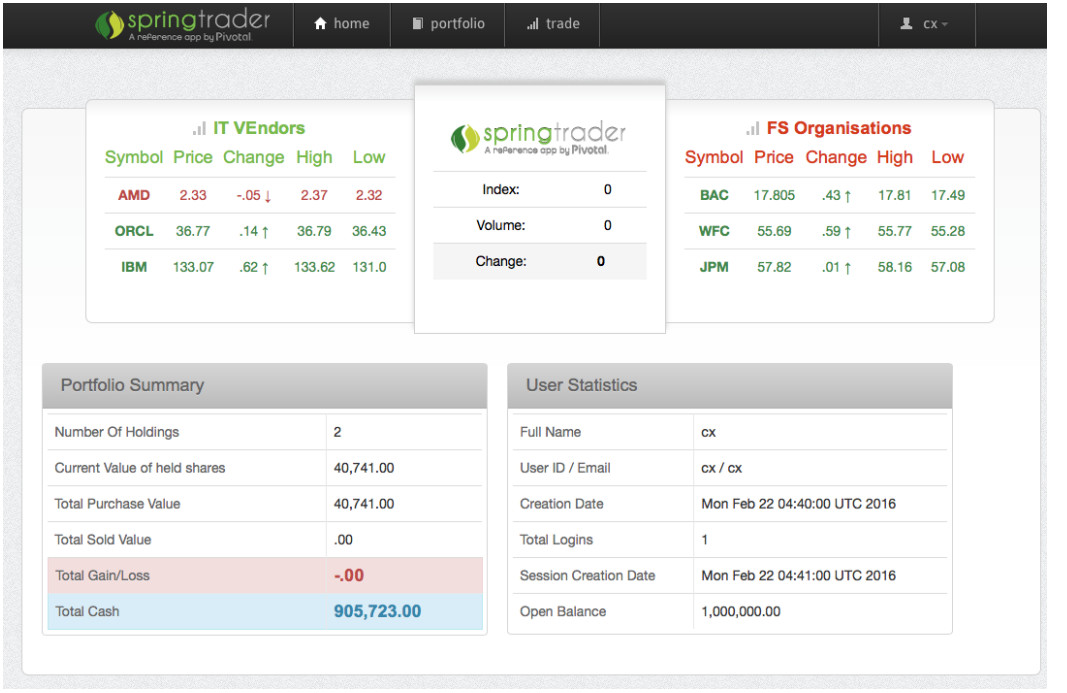
1. Check if all services are registered with Eureka server. Click on Manage service.



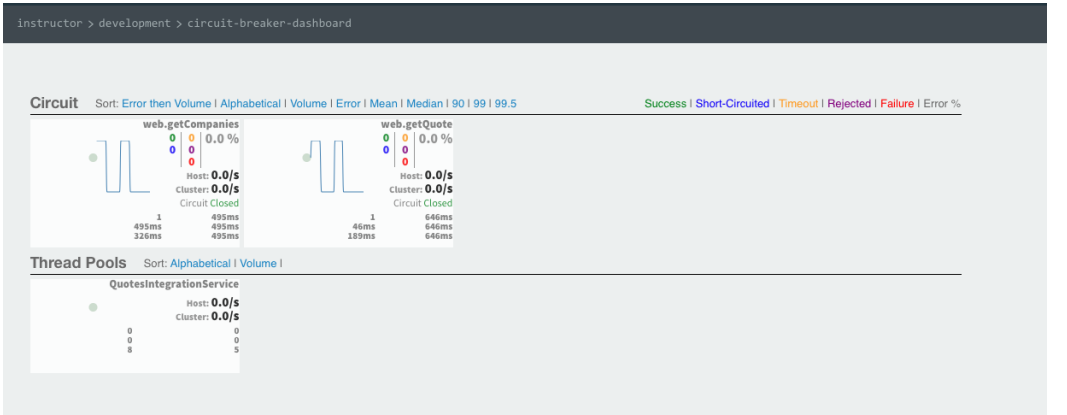
2. See all four services registered with Eureka.



3. In browser, go to the **webtrader** URL shown during cf push



4. Finally, you can check if Hystrix is working. Checking its dashboard.



Appendix A. Eclipse Tips

A.1. Introduction

This section will give you some useful hints for using Eclipse.

A.2. Package Explorer View

Eclipse's Package Explorer view offers two ways of displaying packages. Flat view, used by default, lists each package at the same level, even if it is a subpackage of another. Hierarchical view, however, will display subpackages nested within one another, making it easy to hide an entire package hierarchy. You can switch between hierarchical and flat views by selecting the menu inside the package view (represented as a triangle), selecting either Flat or Hierarchical from the Package Presentation submenu.

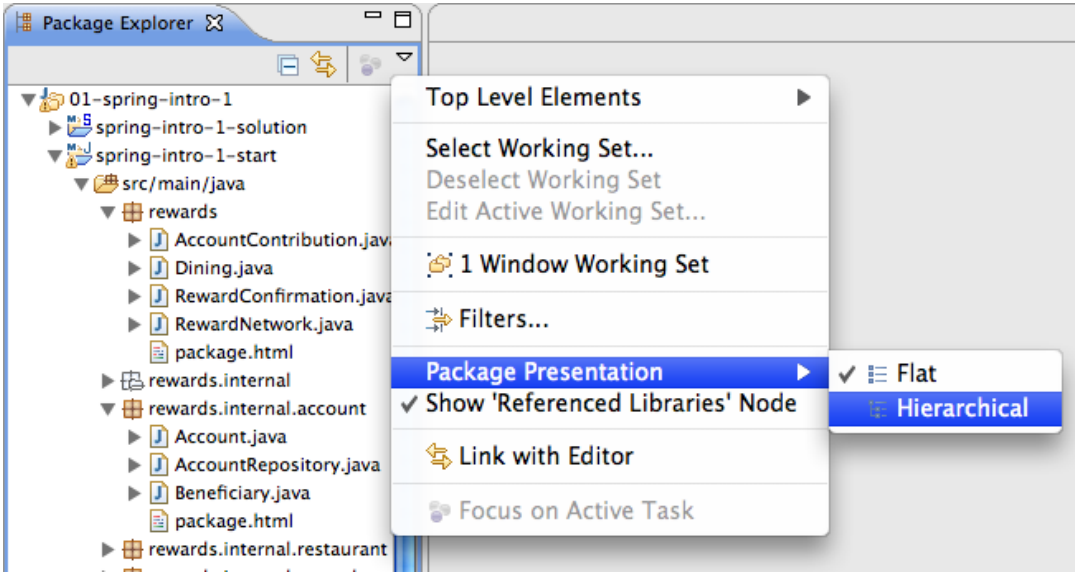


Figure A.1. Swtiching Views

Switch between hierarchical and flat views by selecting the menu inside the package view (represented as a

triangle), selecting either Flat or Hierarchical from the Package Presentation submenu

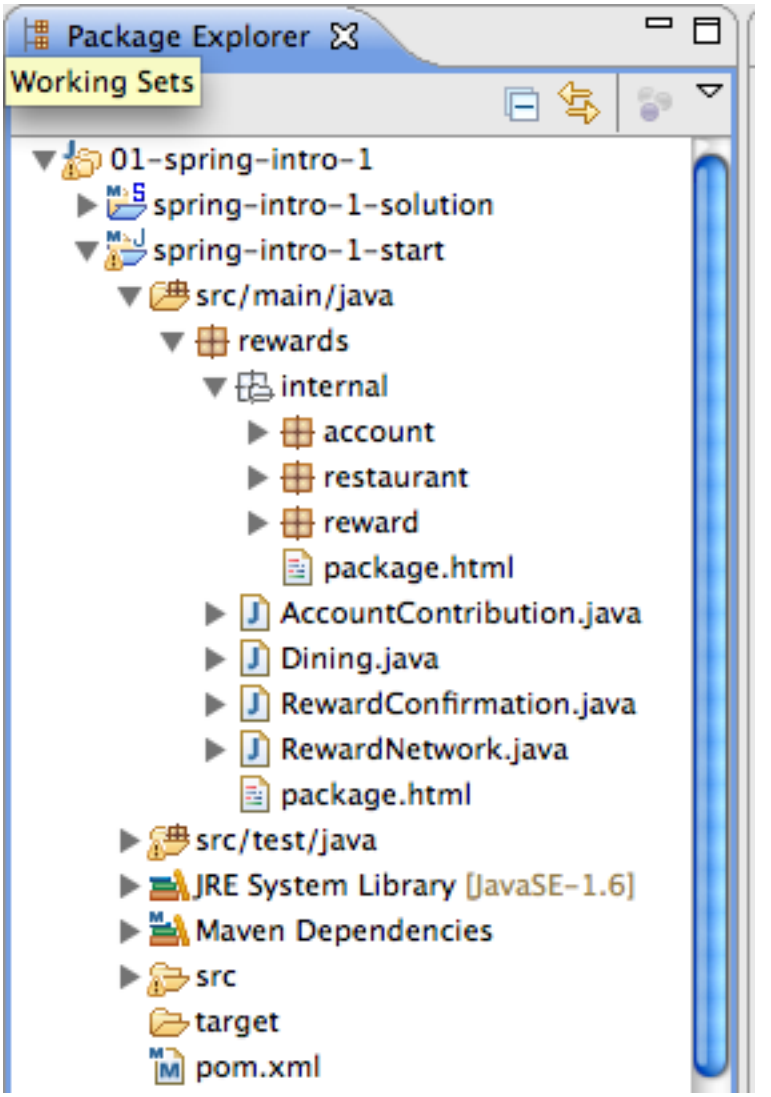


Figure A.2. The hierarchical view shows nested packages in a tree view

A.3. Add Unimplemented Methods

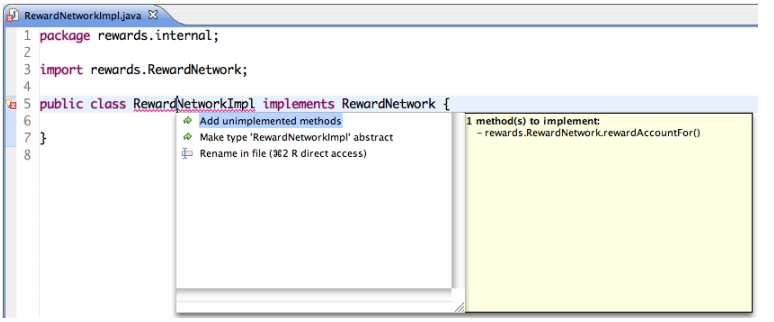


Figure A.3. Add unimplemented methods" quick fix

A.4. Field Auto-Completion

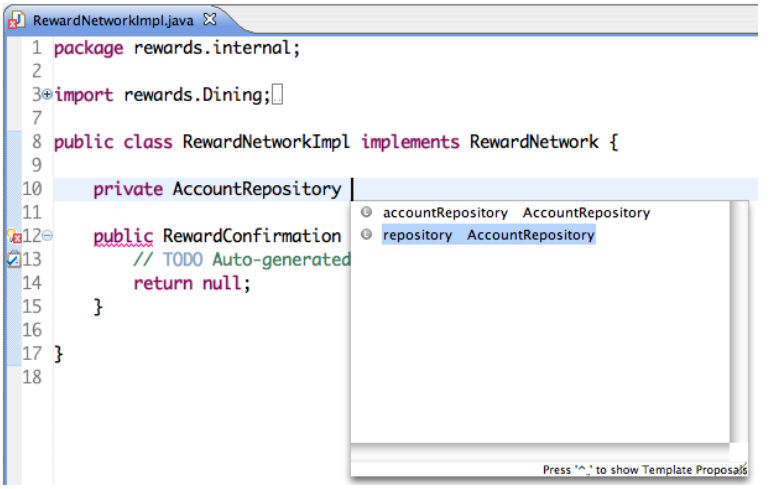


Figure A.4. Field name auto-completion

A.5. Generating Constructors From Fields

You can "Generate a Constructor using Fields" using the Source Menu (ALT + SHIFT + S)

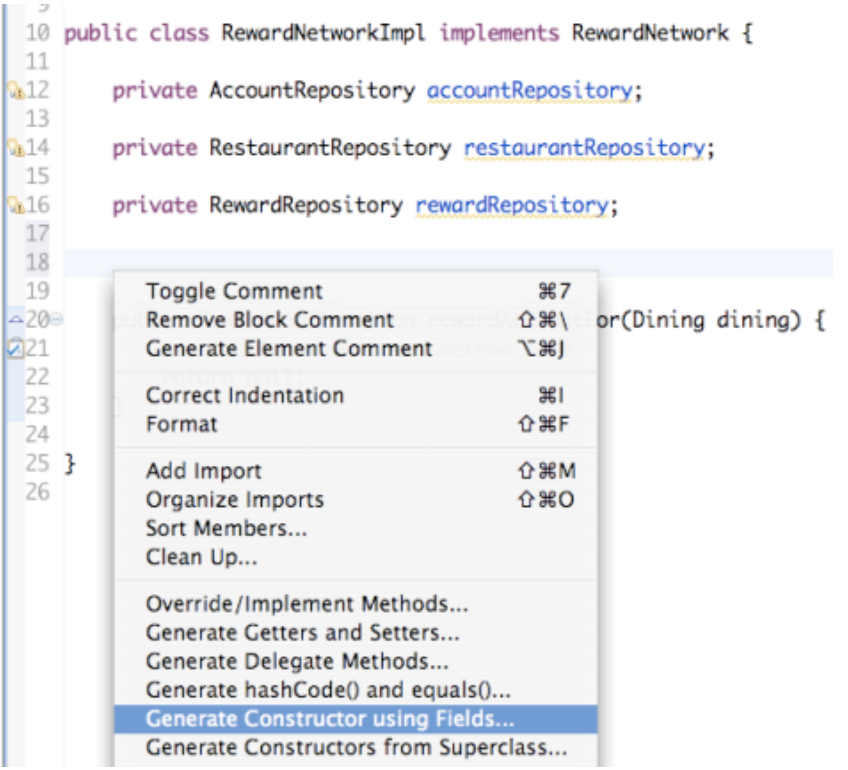


Figure A.5. Generating Constructors

A.6. Field Naming Conventions

A field's name should describe the role it provides callers, and often corresponds to the field's type. It should not describe implementation details. For this reason, a bean's name often corresponds to its service interface. For example, the class JdbcAccountRepository implements the AccountRepository interface. This interface is what callers work with. By convention, then, the bean name should be accountRepository.

A.7. Tasks View

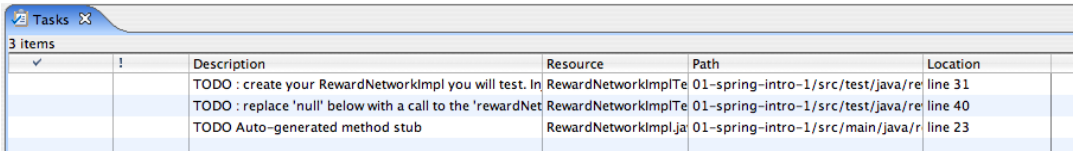


Figure A.6. The tasks view in the bottom right page area

You can configure the Tasks View to only show the tasks relevant to the current project. In order to do this, open the dropdown menu in the upper-right corner of the tasks view (indicated by the little triangle) and select 'Configure Content...'. Now select the TODO configuration and from the Scopes, select 'On any element in same project'. Now if you have multiple project opened, with different TODOs, you will only see those relevant to the current project.

A.8. Rename a File

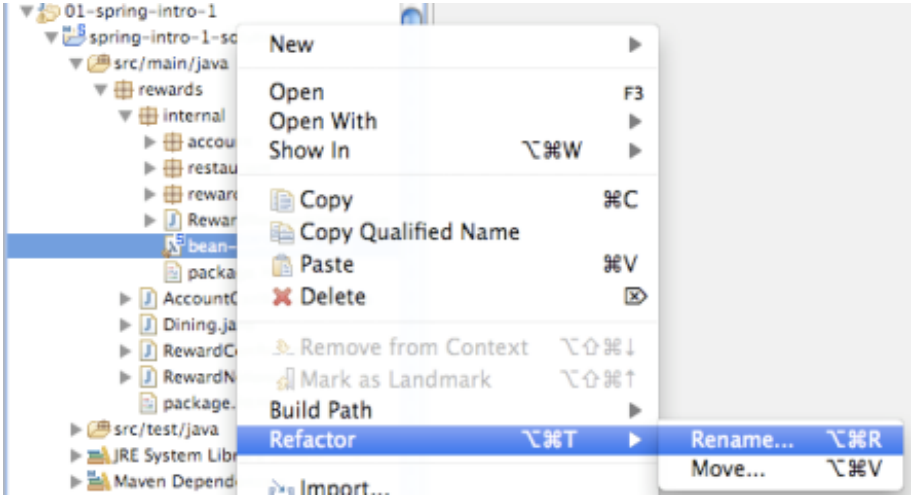


Figure A.7. Renaming a Spring configuration file using the Refactor command

