Pivotal Cloud Native Tour

Lab 1 - Getting Started with Spring Boot

Bootstrap the Project

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

```
Group
io.pivotal.spring

Artifact
hello-spring-boot
```

3. In the Dependencies section, search for web, and choose **Web** from the autocomplete menu.
4. Click the Generate Project button. Your browser will download a zip file. Unpack that zip file at **$COURSE_HOME/labs/initial.**
5. Import the project's pom.xml into your editor/IDE of choice.
6. Add a `@RestController` annotation to the class `io.pivotal.spring.HelloSpringBootApplication`.
7. Add the following request handler to the class `io.pivotal.spring.HelloSpringBootApplication`:

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

Build and Run with Embedded Apache Tomcat

Spring Boot will embed Apache Tomcat by default.

1. Build the application:

```
$ ./mvnw package
```

2. Run the application:

```
$ java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

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

```
2015-12-06 17:17:30.507  INFO 60277 --- [           main] s.b.c.e.t.To
mcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2015-12-06 17:17:30.511  INFO 60277 --- [           main] io.pivotal.s
pring.HelloSpringBootApplication    : Started HelloSpringBootApplicati
on in 3.201 seconds (JVM running for 3.846)
```

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

```
Hello World!
```

Refactor to Externalize the Config

1. Rename src/main/resources/application.properties to src/main/resources/application.yml. Into that file, paste the following:

```
greeting: Hello
```

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

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

3. Also io.pivotal.spring.HelloSpringBootApplication, change the return statement

of hello() to the following:

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

4. Build the application:

```
$ ./mvnw package
```

5. Run the application:

```
$ java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

6. Visit the application in the browser (http://localhost:8080), and verify that the output is still the following:

```
Hello World!
```

7. Stop the application.

Using Environment Variables for Config

1. Run the application again, this time setting the GREETING environment variable:

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

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

```
Ohai World!
```

3. Stop the application.

Using Spring Profiles for Config

1. Add a spanish profile to application.yml. Your finished configuration should reflect the following:

```
greeting: Hello

---

spring:
  profiles: spanish

greeting: Hola
```

2. Build the application:

```
$ ./mvnw package
```

3. Run the application, this time setting the SPRING_PROFILES_ACTIVE environment variable:

```
$ SPRING_PROFILES_ACTIVE=spanish java -jar target/hello-spring-boot-0.
0.1-SNAPSHOT.jar
```

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

```
Hola World!
```

5. Stop the application.

Resolving Conflicts

1. Run the application, this time setting both the SPRING_PROFILES_ACTIVE and GREETING environment variables:

```
$ SPRING_PROFILES_ACTIVE=spanish GREETING=Ohai java -jar target/hello-
spring-boot-0.0.1-SNAPSHOT.jar
```

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

```
Ohai World!
```

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.

Lab 2 - Introspection, Monitoring, and Metrics using Spring Boot Actuator

Set up the Actuator

1. Continue with the project you completed in Lab 1.
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>
```

Introspection Endpoints

1. Build the application:

```
$ ./mvnw package
```

2. Run the application:

```
$ java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

3. 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/configprops

- Displays a collated list of all @ConfigurationProperties.

http://localhost:8080/env

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

http://localhost:8080/mappings

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

http://localhost:8080/dump

- Performs a thread dump.

http://localhost:8080/trace

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

Just read it - Build and Version Control Info

Spring Boot provides an endpoint (http://localhost:8080/info) that allows the exposure of arbitrary metadata.

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

1. Add the following plugin to your Maven build. It will add Git branch and commit coordinates to the /info endpoint:

```
<plugin>
    <groupId>pl.project13.maven</groupId>
    <artifactId>git-commit-id-plugin</artifactId>
    <configuration>
        <dotGitDirectory>../../../.git</dotGitDirectory>
    </configuration>
</plugin>
```

NOTE
The path …/…/…/.git refers to the .git directory at the root of the course materials.

2. Add the following properties to src/main/resources/application.yml:

```
info:
  build:
        artifact: @project.artifactId@
    name: @project.name@
    description: @project.description@
    version: @project.version@
```

These will add the project’s Maven coordinates to the /info endpoint. The Spring Boot Maven plugin will cause them to automatically be replaced in the assembled JAR.

3. Build the application:

```
$ ./mvnw package
```

4. Run the application:

```
$ java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

5. Visit the application in the browser (http://localhost:8080/info), and verify that the output is similar to the following:

```
{
   "git" : {
      "commit" : {
         "time" : "2015-12-06T17:25:32-0500",
         "id" : "78b0e1b"
      },
      "branch" : "master"
   },
   "build" : {
      "version" : "0.0.1-SNAPSHOT",
      "name" : "hello-spring-boot",
      "artifact" : "hello-spring-boot",
      "description" : "Demo project for Spring Boot"
   }
}
```

Health Indicators

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

1. Normally, when Spring Security is not enabled, the /health endpoint will only expose an UP or DOWN value. To simplify working with the endpoint for this lab, we will turn off its sensitivity. Add the following to src/main/resources/application.yml:

```
endpoints:
  health:
    sensitive: false
```

2. Create the class io.pivotal.spring.FlappingHealthIndicator 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").with
Detail("random", result).build();
        } else {
            return Health.up().withDetail("flapper", "ok").withDetail(
"random", result).build();
        }
    }
}
```

This demo health indicator will randomize the health check.

3. Build the application:

```
$ ./mvnw package
```

4. Run the application:

```
$ java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

5. 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
  }
}
```

Metrics

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

1. Create the class io.pivotal.spring.GreetingService and into it paste the following code:

```
@Component
public class GreetingService {

    @Autowired
    CounterService counterService;

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

    public String getGreeting() {
        counterService.increment("counter.services.greeting.invoked");
        return greeting;
    }
}
```

This class is using the @Autowired CounterService to count the number of times that the getGreeting() method has been invoked.

2. Refactor the contents of the class `io.pivotal.spring.HelloSpringBootApplication`:

```
@Autowired
private GreetingService greetingService;

@RequestMapping("/")
public String hello() {
    return String.format("%s World!", greetingService.getGreeting());
}

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

hello() is now delegating the source of the greeting to our newly created GreetingService.

3. Build the application:

```
$ ./mvnw package
```

4. Run the application:

```
$ java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

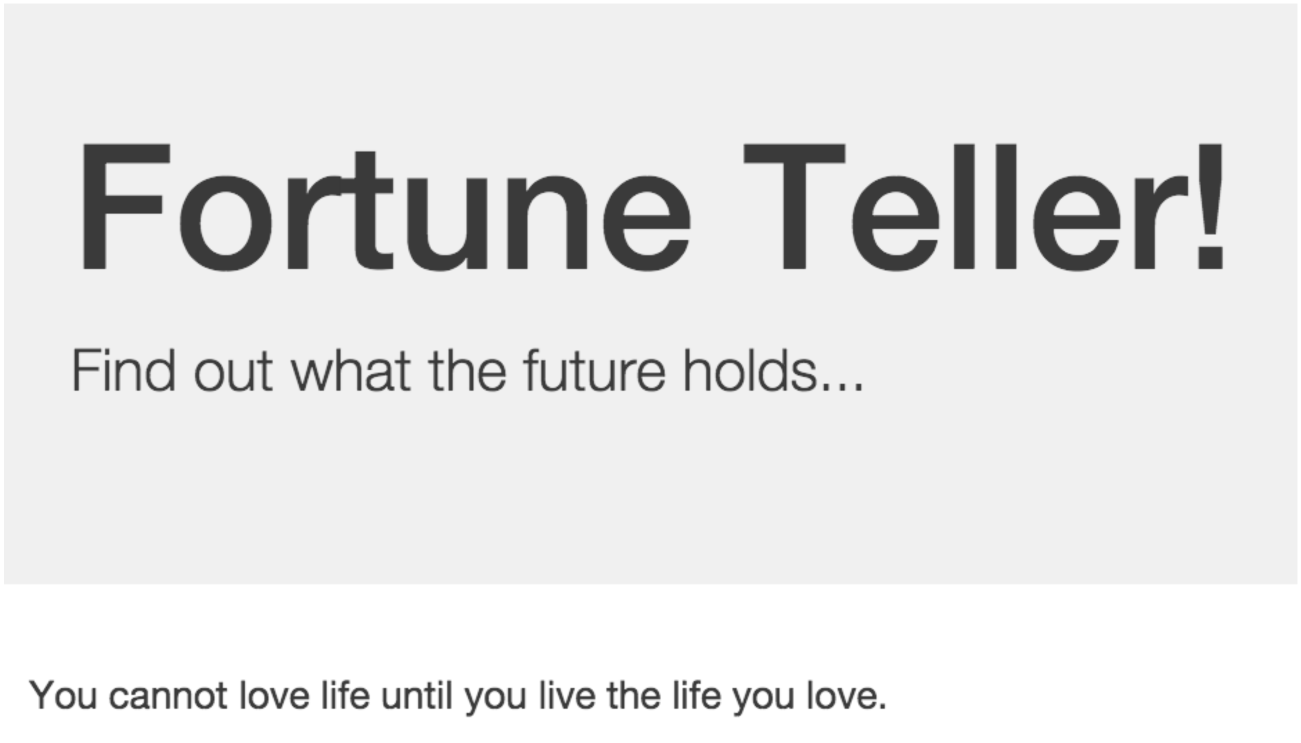
5. Visit the application in the browser (http://localhost:8080) and refresh the page several times.
6. Now visit the /metrics endpoint (http://localhost:8080/metrics). Among the autogenerated metrics you should see a counter for the GreetingService invocations:

```
counter.services.greeting.invoked: 16,
```

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

Lab 3 - Composing Cloud Native Apps with Spring Cloud and Netflix OSS

In this lab we’re going to build a Fortune Teller application from two microservices, one that serves up random Chinese fortune cookie fortunes, and one that presents a user interface. The resulting application looks like the following screenshot:



Fortune Teller!

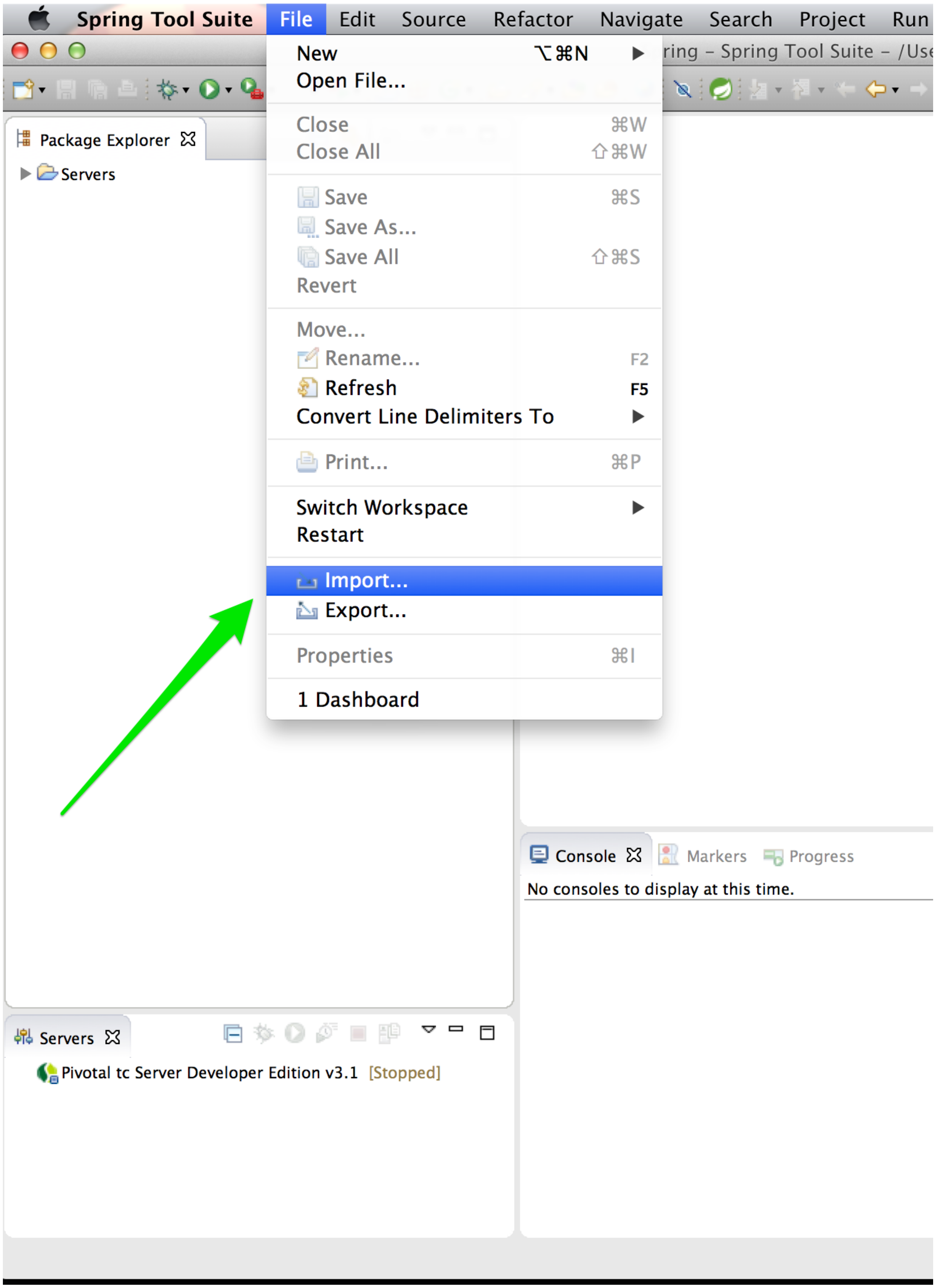
Find out what the future holds...

You cannot love life until you live the life you love.

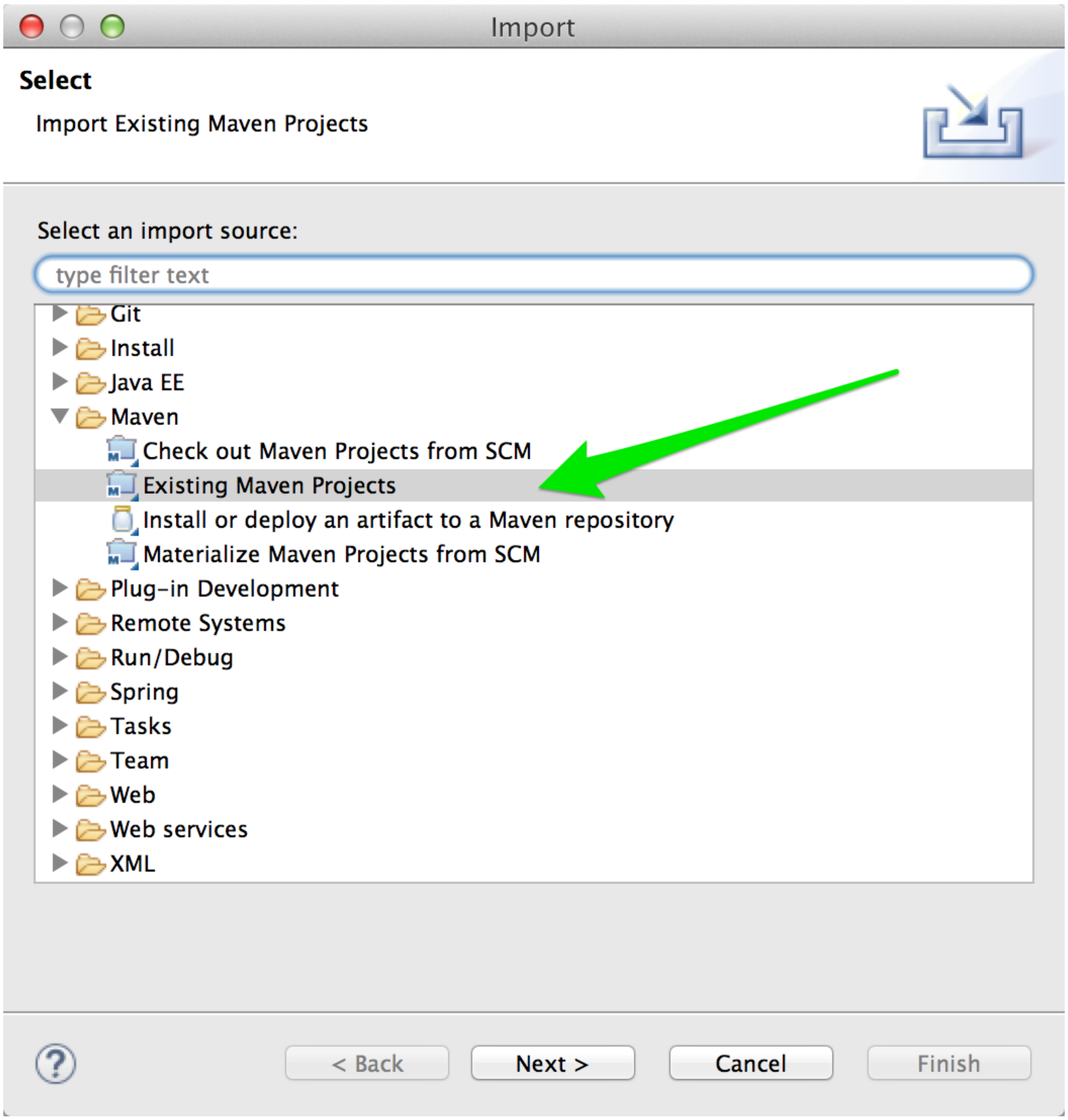
We’ll leverage libraries and services from Spring Cloud and Netflix OSS to help us compose the system.

Bootstrapping

1. Choose File → Import:



2. Choose Maven → Existing Maven Projects:



Import

Select

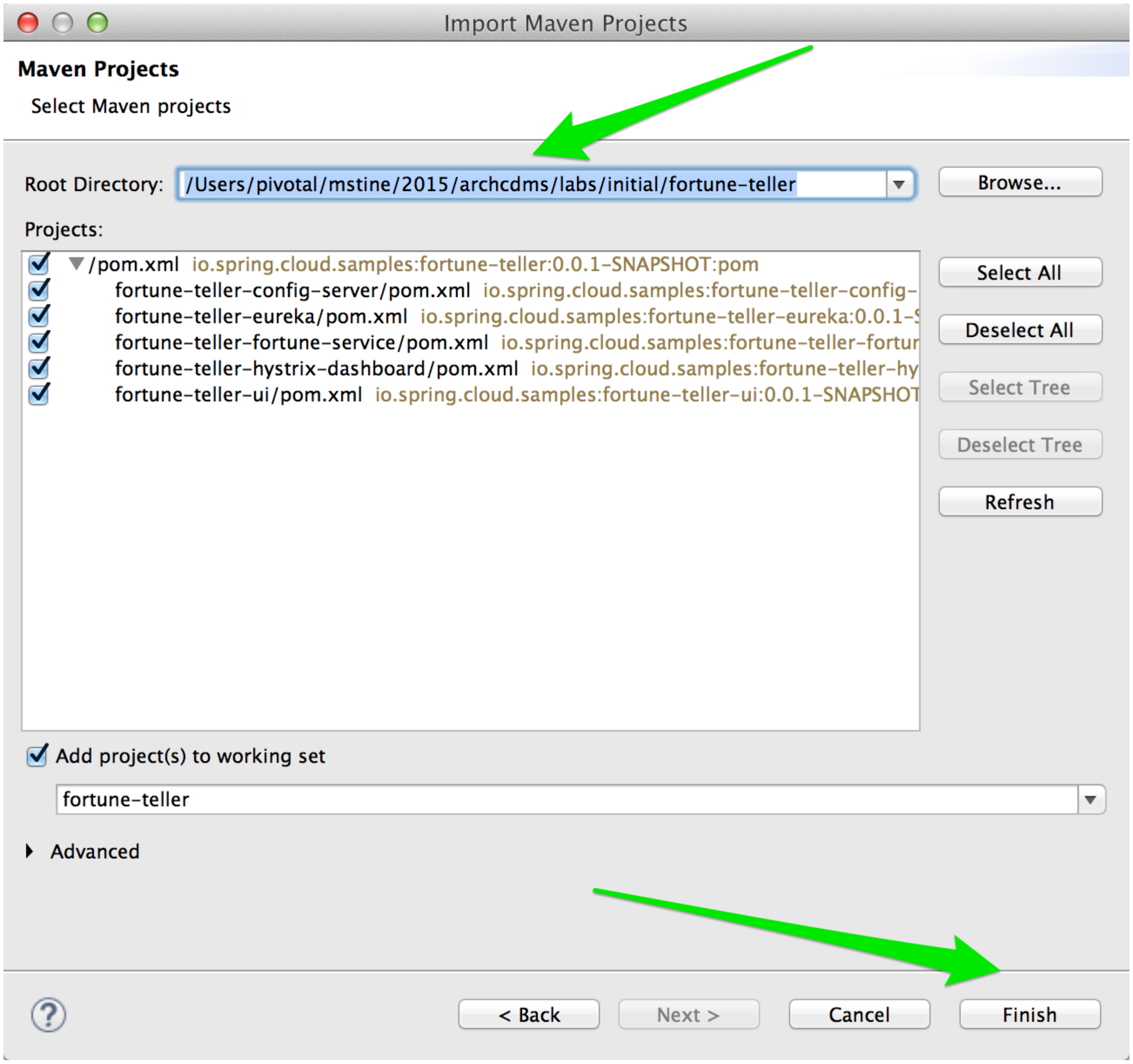
Import Existing Maven Projects

Select an import source:

type filter text

- Git
- Install
- Java EE
- Maven
  - Check out Maven Projects from SCM
  - Existing Maven Projects
  - Install or deploy an artifact to a Maven repository
  - Materialize Maven Projects from SCM
- Plug-in Development
- Remote Systems
- Run/Debug
- Spring
- Tasks
- Team
- Web
- Web services
- XML

3. Choose the project's pom.xml, found at $COURSE_HOME/labs/initial/fortune-teller and click Finish.


Import Maven Projects

Maven Projects

Select Maven projects

Root Directory: /Users/pivotal/mstine/2015/archcdms/labs/initial/fortune-teller

Projects:

```
✓ /pom.xml io.spring.cloud.samples:fortune-teller:0.0.1-SNAPSHOT:pom
✓   fortune-teller-config-server/pom.xml io.spring.cloud.samples:fortune-teller-config-
✓   fortune-teller-eureka/pom.xml io.spring.cloud.samples:fortune-teller-eureka:0.0.1-S
✓   fortune-teller-fortune-service/pom.xml io.spring.cloud.samples:fortune-teller-fortur
✓   fortune-teller-hystrix-dashboard/pom.xml io.spring.cloud.samples:fortune-teller-hy
✓   fortune-teller-ui/pom.xml io.spring.cloud.samples:fortune-teller-ui:0.0.1-SNAPSHOT
```

✓ Add project(s) to working set: fortune-teller

Config Server

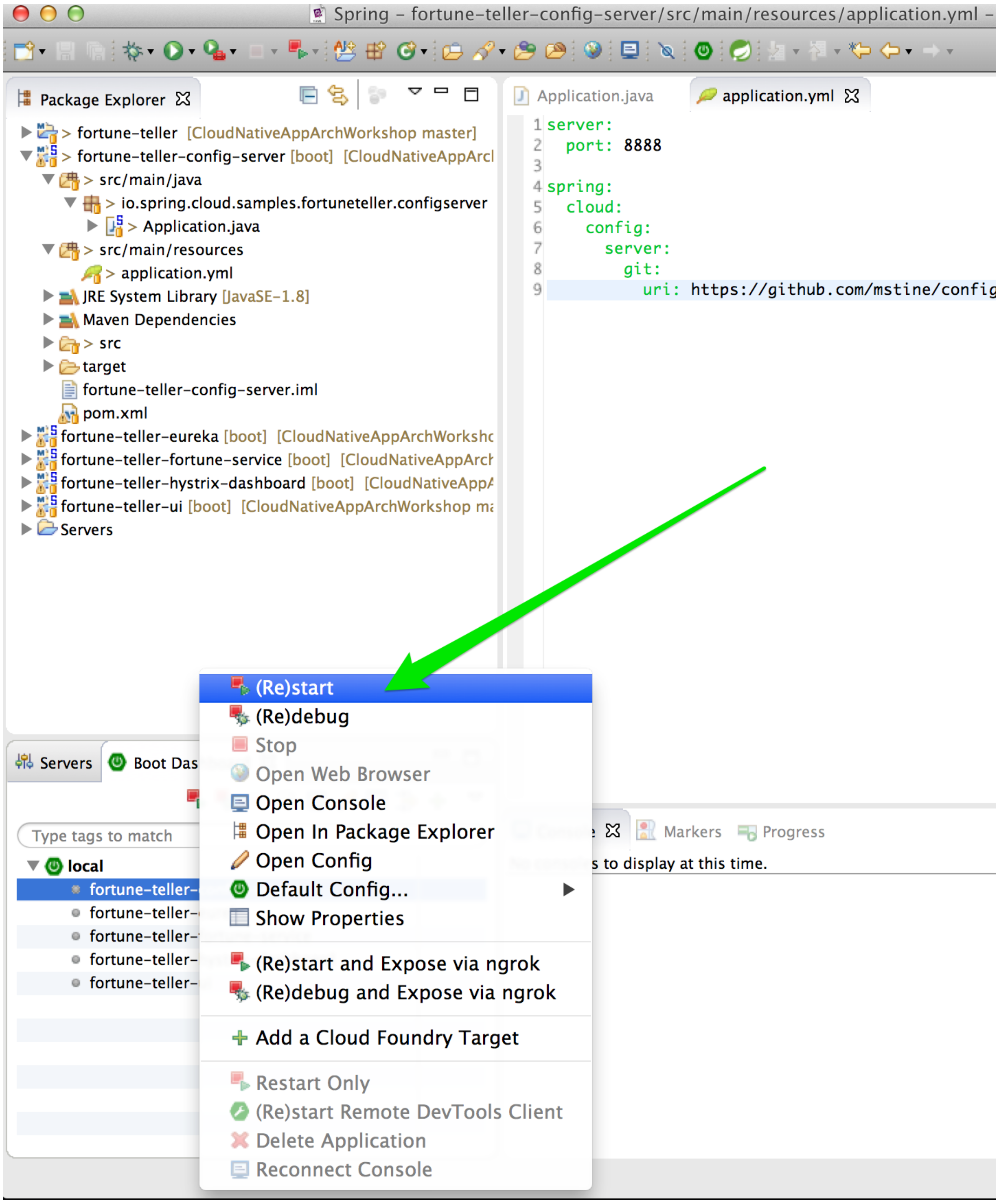
First we'll create a Spring Cloud Config Server to distribute shared configuration across our distributed system. This configuration will form the basis of how the entire system will compose.

1. In the fortune-teller-config-server module, add a `@EnableConfigServer` annotation to the class `io.spring.cloud.samples.fortuneteller.configserver.Application`.
2. Paste the following configuration properties into the file src/main/resources/application.yml:

```
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/mstine/config-repo.git
```

These properties tell the Config Server to listen on port 8888 and to source its configuration from the Git repository found at https://github.com/mstine/config-repo.

3. Browse the file https://github.com/mstine/config-repo/blob/master/application.yml#L1-L18. These lines tell all applications using the cloud Spring profile how to connect to and register themselves with Eureka.
4. In the Boot Dashboard, right click on fortune-teller-config-server and choose (Re)start:


Spring - fortune-teller-config-server/src/main/resources/application.yml -
Package Explorer
> fortune-teller [CloudNativeAppArchWorkshop master]
> fortune-teller-config-server [boot] [CloudNativeAppArcl
> src/main/java
> io.spring.cloud.samples.fortuneteller.configserver
> Application.java
> src/main/resources
> application.yml
JRE System Library [JavaSE-1.8]
Maven Dependencies
> src
target
fortune-teller-config-server.iml
pom.xml
fortune-teller-eureka [boot] [CloudNativeAppArchWorkshc
fortune-teller-fortune-service [boot] [CloudNativeAppArch
fortune-teller-hystrix-dashboard [boot] [CloudNativeAppA
fortune-teller-ui [boot] [CloudNativeAppArchWorkshop ma
Servers
Application.java
application.yml
Servers
Boot Das
Type tags to match
local
fortune-teller-
fortune-teller-
fortune-teller-
fortune-teller-
fortune-teller-
(Re)start
(Re)debug
Stop
Open Web Browser
Open Console
Open In Package Explorer
Open Config
Default Config...
Show Properties
(Re)start and Expose via ngrok
(Re)debug and Expose via ngrok
Add a Cloud Foundry Target
Restart Only
(Re)start Remote DevTools Client
Delete Application
Reconnect Console
Markers
Progress
s to display at this time.

```
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/mstine/config
```

5. Browse to http://localhost:8888/application/default. You should see output similar to the following:

config server output

6. Note that the response JSON defines a Spring PropertySource that contains Eureka configuration consistent with what you saw in the Git repository.

Eureka Server

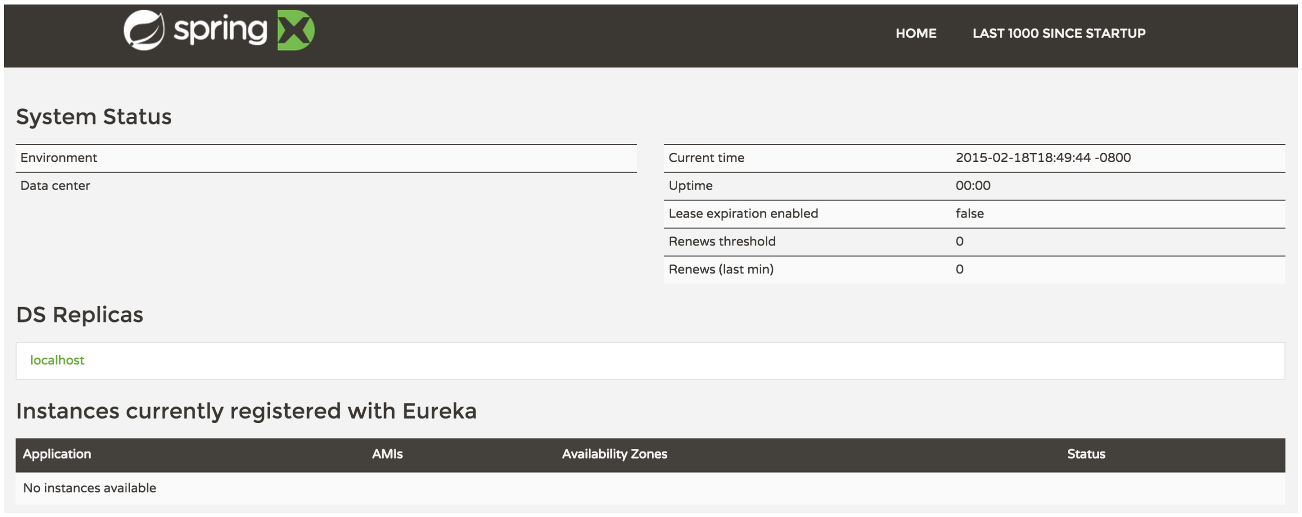
Next we'll create a Eureka service registry using Spring Cloud Netflix. This server will act as the address book our UI application will use to locate the Fortune Service.

1. In the fortune-teller-eureka module, add a `@EnableEurekaServer` annotation to the class `io.spring.cloud.samples.fortuneteller.eureka.Application`.
2. Paste the following configuration properties into the file src/main/resources/application.yml:

```
server:
  port: 8761

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

These properties tell the Eureka Server to listen on port 8761 and to configure itself in what is essentially "standalone" mode.

3. As before, in the Boot Dashboard, right click on fortune-teller-eureka and choose (Re)start.
4. Browse to http://localhost:8761. You should see a UI similar to the following:



System Status

Environment	
Data center	

Current time	2015-02-18T18:49:44 -0800
Uptime	00:00
Lease expiration enabled	false
Renews threshold	0
Renews (last min)	0

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

eureka 1

Fortune Service

Now that our Spring Cloud/Netflix OSS infrastructure is bootstrapped, we'll begin development of our first microservice. The Fortune Service will serve up a list of all fortune cookie fortunes present in its database or a single random fortune.

1. In the fortune-teller-fortune-service module, create the package `io.spring.cloud.samples.fortuneteller.fortuneservice.domain`.
2. Create a JPA domain class, `io.spring.cloud.samples.fortuneteller.fortuneservice.domain.Fortune`. Into that class, paste the following code:

```
        @Entity
        @Table(name = "fortunes")
        public class Fortune {

            @Id
            @GeneratedValue
            private Long id;

            @Column(nullable = false)
            private String text;

            public Long getId() {
                return id;
            }

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

            public String getText() {
                return text;
            }

            public void setText(String text) {
                this.text = text;
            }
        }
```

Add the appropriate imports using the IDE’s quick fix feature.

This class represents the domain model for a fortune, having an unique identifier and the fortune’s text.

3. Next, create the package `io.spring.cloud.samples.fortuneteller.fortuneservice.repositories`.
4. Create a Spring Data JPA repository interface, io.spring.cloud.samples.fortuneteller.fortuneservice.repositories.FortuneRepository. Into that class, paste the following code:

```
public interface FortuneRepository extends PagingAndSortingRepository<
Fortune, Long> {

    @Query("select fortune from Fortune fortune order by RAND()")
    public List<Fortune> randomFortunes(Pageable pageable);
}
```

5. Add the appropriate imports using the IDE's quick fix feature.

This interface will provide all of the basic boilerplate CRUD features we'll want for our Fortune domain class. The @Query annotation defines a SQL statement that will return a random ordering of the fortunes. We'll use this as the basis of returning a random fortune via our REST interface.

6. Next, create the package io.spring.cloud.samples.fortuneteller.fortuneservice.controllers.
7. Create a Spring MVC Controller class, io.spring.cloud.samples.fortuneteller.fortuneservice.controllers.FortuneController. Into that class, paste the following code:

```
@RestController
public class FortuneController {

    @Autowired
    FortuneRepository repository;

    @RequestMapping("/fortunes")
    public Iterable<Fortune> fortunes() {
        return repository.findAll();
    }

    @RequestMapping("/random")
    public Fortune randomFortune() {
        List<Fortune> randomFortunes = repository.randomFortunes(new P
ageRequest(0, 1));
        return randomFortunes.get(0);
    }
}
```

Add the appropriate imports using the IDE's quick fix feature.

This class provides the two REST endpoints of our microservice. The randomFortune method provides a PageRequest to limit the returned results to one.

8. Add @EnableJpaRepositories (to enable the Spring Data JPA repository we created) and @EnableDiscoveryClient (to switch on Eureka registration) annotations to io.spring.cloud.samples.fortuneteller.fortuneservice.Application.

Paste the following configuration properties into the file src/main/resources/bootstrap.yml:

```
spring:
  application:
    name: fortunes
  cloud:
    config:
      uri: ${vcap.services.config-service.credentials.uri:http://localhost:8888}
```

This configuration specifies the name of the application, which will be used to register the service in Eureka. It also tells the service how to locate the config server.

9. Paste the following configuration properties into the file src/main/resources/application.yml:

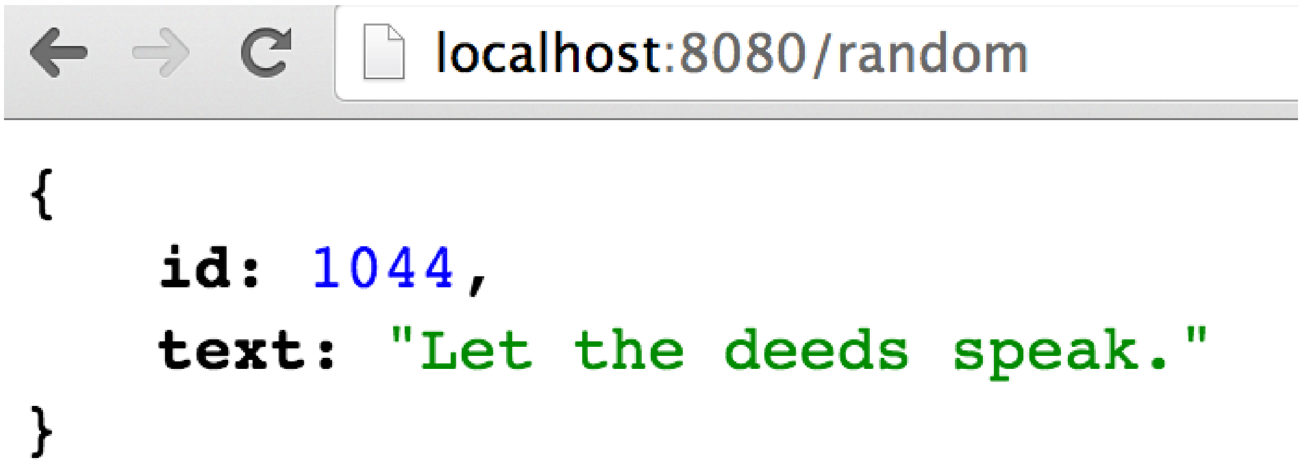
```
spring:
  jpa:
    hibernate:
      ddl-auto: create-drop

---

spring:
  profiles: default
  jpa:
    database: HSQL

---

spring:
  profiles: cloud
  jpa:
    database: POSTGRESQL
```

This configuration specifies the model to DDL setting to be used by Hibernate, as well as the Hibernate dialects to be used for local development and cloud.

10. As before, in the Boot Dashboard, right click on fortune-teller-fortune-service and choose (Re)start.

After the service has been running for ten seconds, refresh your Eureka browser tab. You should see a registration entry for the fortune service.

11. Visit http://localhost:8080/random. You should see output similar to the following:



```
{
    id: 1044,
    text: "Let the deeds speak."
}
```

Fortune UI

Now that our Fortune microservice is running, we'll begin development of our second microservice. The Fortune UI will serve up a AngularJS single page application that consumes the fortune service.

1. In the fortune-teller-ui module, create the package `io.spring.cloud.samples.fortuneteller.ui.services.fortunes`. This package will contain our integration with the fortune service.
2. Create the POJO io.spring.cloud.samples.fortuneteller.ui.services.fortunes.Fortune. Into that class, paste the following code:

```
public class Fortune {
    private Long id;
    private String text;

    public Fortune() {
    }

    public Fortune(Long id, String text) {
        this.id = id;
        this.text = text;
    }

    public Long getId() {
        return id;
    }

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

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

We will use this class to unmarshal the response from the fortune service.

3. Create the Spring Bean

`io.spring.cloud.samples.fortuneteller.ui.services.fortunes.FortuneService`.

Into that class, paste the following code:

```
@Service
public class FortuneService {

    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "fallbackFortune")
    public Fortune randomFortune() {
        return restTemplate.getForObject("http://fortunes/random", For
tune.class);
    }

    private Fortune fallbackFortune() {
        return new Fortune(42L, "Your future is unclear.");
    }
}
```

Add the appropriate imports using the IDE’s quick fix feature.

This class is our integration point with the fortune service. It uses a special RestTemplate bean that integrates with Ribbon from Netflix OSS. The argument passed to getForObject, http://fortunes/random, will be resolved by Ribbon to the actual address of the fortune service. This method is also protected by a Hystrix circuit breaker using the @HystrixCommand annotation. If the circuit is tripped to open (due to an unhealthy fortune service), the fallbackFortune method will return a dummy response.

4. Next, create the package `io.spring.cloud.samples.fortuneteller.ui.controllers`.
5. Create a Spring MVC Controller class, `io.spring.cloud.samples.fortuneteller.ui.controllers.UiController`.

Into that class, paste the following code:

```
@RestController
public class UiController {

    @Autowired
    FortuneService service;

    @RequestMapping("/random")
    public Fortune randomFortune() {
        return service.randomFortune();
    }
}
```

6. Add the appropriate imports using the IDE’s quick fix feature.

This class provides the REST endpoint that will be consumed by our AngularJS UI.

7. Add @EnableCircuitBreaker (to switch on Hystrix circuit breakers) and @EnableDiscoveryClient (to switch on Eureka registration) annotations to io.spring.cloud.samples.fortuneteller.ui.Application.
8. Paste the following configuration properties into the file src/main/resources/bootstrap.yml:

```
spring:
  application:
    name: ui
  cloud:
    config:
      uri: ${vcap.services.config-service.credentials.uri:http://local
host:8888}
```

This configuration specifies the name of the application, which will be used to register the service in Eureka. It also tells the service how to locate the config server.

9. Paste the following configuration properties into the file src/main/resources/application.yml:

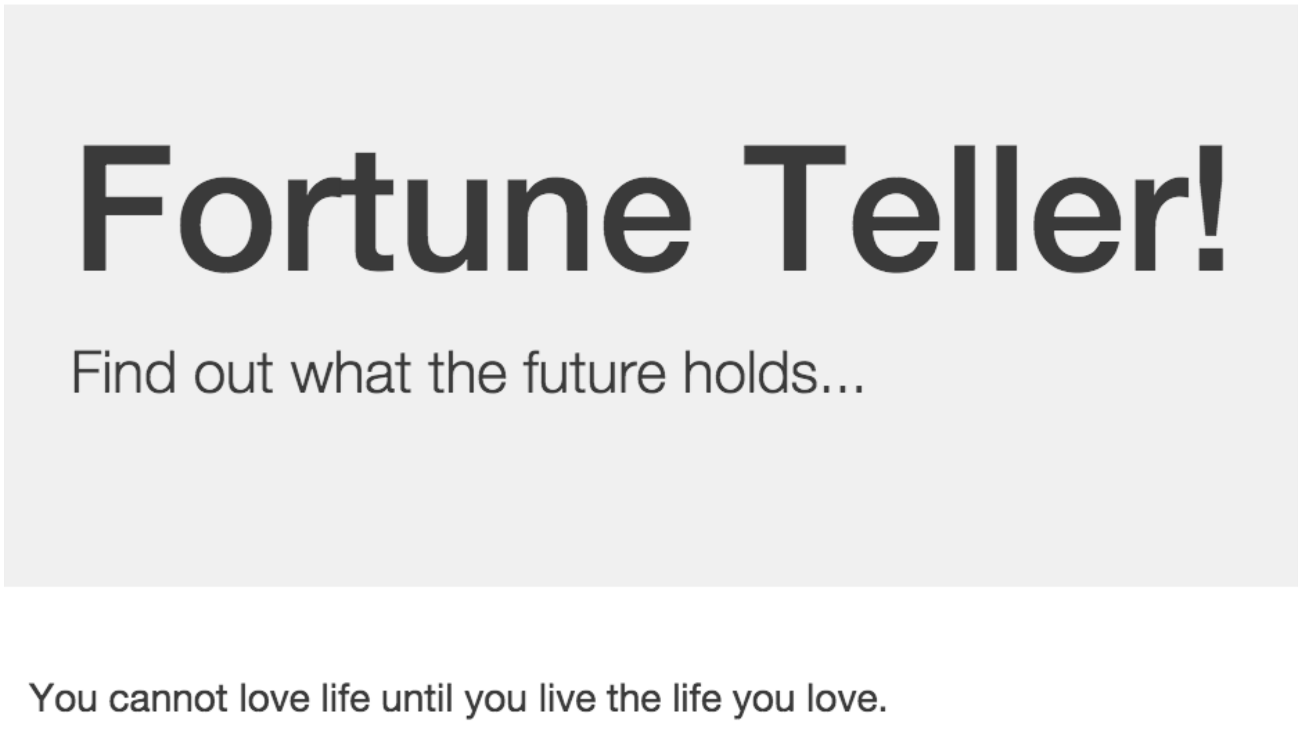
```
server:
  port: 8081

eureka:
  client:
    registryFetchIntervalSeconds: 5

fortunes:
  ribbon:
    ServerListRefreshInterval: 5000
```

This configuration instructs the application to listen on port 8081. It also provides low cache refresh settings for the Eureka client and Ribbon to make the demo more responsive.

10. As before, in the Boot Dashboard, right click on fortune-teller-ui and choose (Re)start.

After the service has been running for ten seconds, refresh your Eureka browser tab. You should see a registration entry for the Fortune UI.

11. Visit http://localhost:8081. You should see output similar to the following:


Fortune Teller!

Find out what the future holds...

You cannot love life until you live the life you love.

Hystrix Dashboard

Finally, we'll add one more piece of infrastructure — a dashboard that allows us to monitor our circuit breakers — using Spring Cloud Netflix.

1. In the fortune-teller-hystrix-dashboard module, add @EnableHystrixDashboard and @Controller annotations to the class io.spring.cloud.samples.fortuneteller.hystrixdashboard.Application.
2. Add the following request handler method to io.spring.cloud.samples.fortuneteller.hystrixdashboard.Application:

```
@RequestMapping("/")
public String home() {
    return "forward:/hystrix";
}
```

3. Add the appropriate imports using the IDE's quick fix feature.

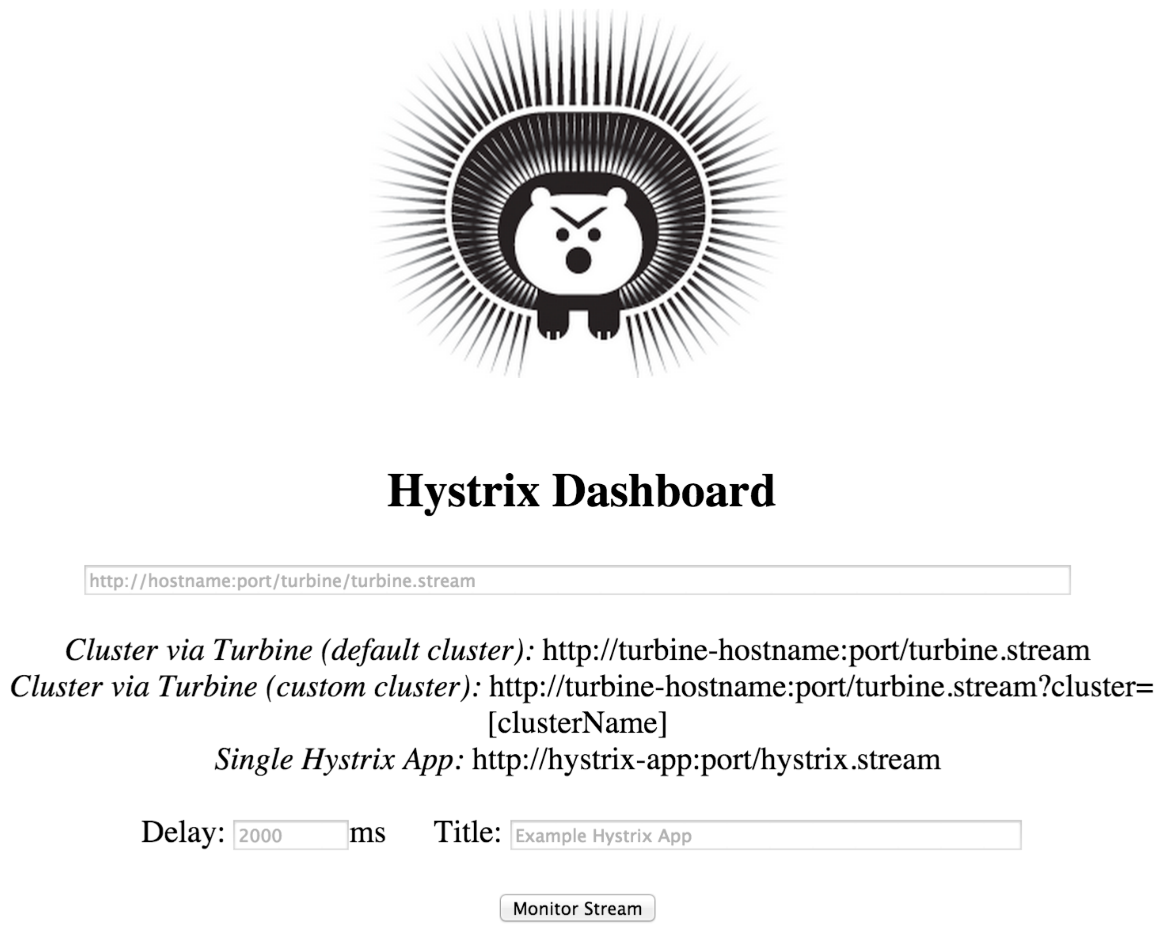
This request handler will allow us to hit the default endpoint our application to interact with the Hystrix Dashboard.

4. Paste the following configuration properties into the file src/main/resources/application.yml:

```
server:
  port: 7979
```

This configuration instructs the application to listen on port 7979.

5. As before, in the Boot Dashboard, right click on fortune-teller-hystrix-dashboard and choose (Re)start.
6. Visit http://localhost:7979. You should see output similar to the following:



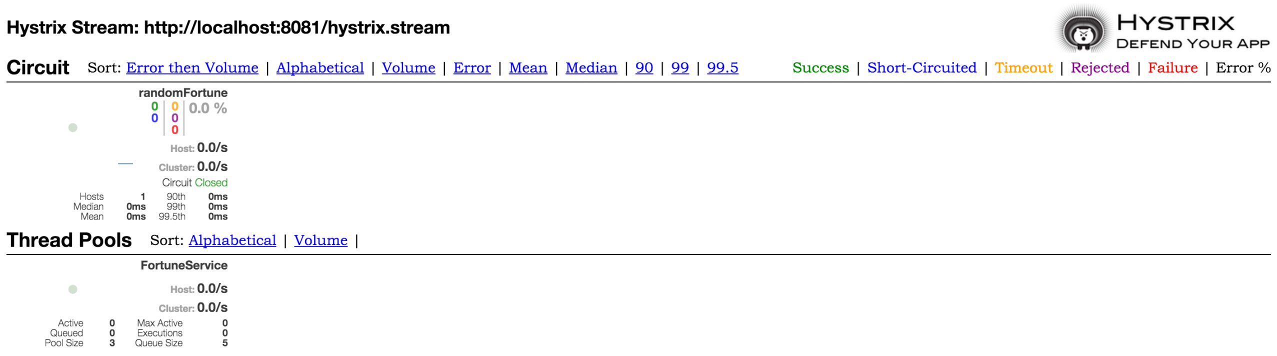
Hystrix Dashboard

http://hostname:port/turbine/turbine.stream

Cluster via Turbine (default cluster): http://turbine-hostname:port/turbine.stream
Cluster via Turbine (custom cluster): http://turbine-hostname:port/turbine.stream?cluster=[clusterName]
Single Hystrix App: http://hystrix-app:port/hystrix.stream

Delay: 2000 ms   Title: Example Hystrix App

7. Paste http://localhost:8081/hystrix.stream into the text field and click Monitor Stream. You should see output similar to the following:



Hystrix Stream: http://localhost:8081/hystrix.stream

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

```
                 randomFortune
                 0 | 0 | 0.0 %
                 0 | 0 |
                   | 0 |
                 Host: 0.0/s
              Cluster: 0.0/s
                Circuit Closed
   Hosts     1    90th    0ms
  Median   0ms    99th    0ms
    Mean   0ms  99.5th    0ms
```

Thread Pools Sort: Alphabetical | Volume |

```
                FortuneService
                 Host: 0.0/s
              Cluster: 0.0/s
    Active   0   Max Active   0
    Queued   0   Executions   0
 Pool Size   3   Queue Size   5
```

Lab 4 - From Zero to Pushing Your First Application to Pivotal Cloud Foundry

Target

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

```
$ cf api api.run.pivotal.io
```

3. Login to Pivotal Web Services:

```
$ cf login
```

Follow the prompts.

Build and Push!

Change to the Spring Music sample application directory:

```
$ cd $COURSE_HOME/labs/complete/spring-music
```

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
```

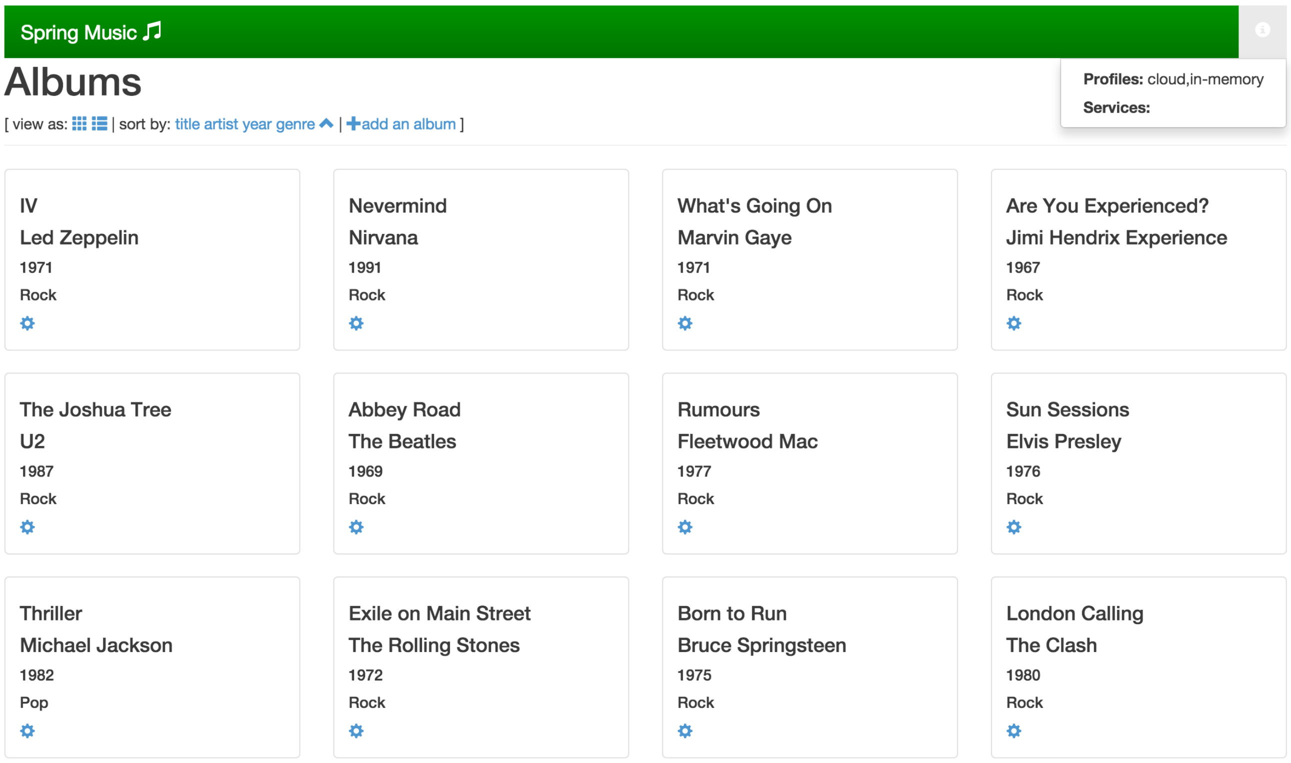
You should see output similar to the following listing. Take a look at the listing callouts for a play-by-play of what’s happening:

```
Using manifest file
/Users/pivotal/mstine/2015/OReilly/CloudNativeArchitectureClass/day_01
/session_02/lab_05/spring-music/manifest.yml (1)

Creating app spring-music in org oreilly-class / space instructor as m
stine@pivotal.io...
OK (2)

Creating route spring-music-hippest-shaman.cfapps.io...
OK (3)

Binding spring-music-hippest-shaman.cfapps.io to spring-music...
OK (4)

Uploading spring-music... (5)
Uploading app files from: /Users/pivotal/mstine/2015/OReilly/CloudNati
veArchitectureClass/day_01/session_02/lab_05/spring-music/build/libs/s
pring-music.war
Uploading 569.7K, 90 files
Done uploading
OK

Starting app spring-music in org oreilly-class / space instructor as m
stine@pivotal.io... (6)
-----> Downloaded app package (21M)
-----> Java Buildpack Version: v2.6.1 |  https://github.com/cloudfound
ry/java-buildpack.git#2d92e70
-----> Downloading Open Jdk JRE 1.8.0_31 from https://download.run.piv
otal.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) (
7)
-----> Downloading Spring Auto Reconfiguration 1.7.0_RELEASE from http
s://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.pi
votal.io/tomcat/tomcat-8.0.18.tar.gz (0.4s)
       Expanding Tomcat to .java-buildpack/tomcat (0.1s) (8)
-----> Downloading Tomcat Lifecycle Support 2.4.0_RELEASE from https:/
/download.run.pivotal.io/tomcat-lifecycle-support/tomcat-lifecycle-sup
port-2.4.0_RELEASE.jar (0.0s)
-----> Downloading Tomcat Logging Support 2.4.0_RELEASE from https://d
ownload.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 ht
tps://download.run.pivotal.io/tomcat-access-logging-support/tomcat-acc
```

```
ess-logging-support-2.4.0_RELEASE.jar (0.0s)
-----> Uploading droplet (66M) (9)

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:OnOutOf
MemoryError=$PWD/.java-buildpack/open_jdk_jre/bin/killjava.sh -Xmx3822
93K -Xms382293K -XX:MaxMetaspaceSize=64M -XX:MetaspaceSize=64M -Xss995
K -Daccess.logging.enabled=false -Dhttp.port=$PORT" $PWD/.java-buildpa
ck/tomcat/bin/catalina.sh run` (10)

Showing health and status for app spring-music in org oreilly-class /
space instructor as mstine@pivotal.io... (11)
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
```

1. 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.
2. 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.
3. 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.

4. 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.
5. 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.
6. Now we begin the staging process. The Java Buildpack is responsible for assembling the runtime components necessary to run the application.
7. Here we see the version of the JRE that has been chosen and installed.
8. And here we see the version of Tomcat that has been chosen and installed.
9. 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 Droplet Execution Agents (DEA's) for execution.
10. The CLI tells you exactly what command and argument set was used to start your application.
11. 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.
12. Visit the application in your browser by hitting the route that was generated by the CLI:


Spring Music ♫
Albums
[view as: | sort by: title artist year genre | add an album]
Profiles: cloud,in-memory
Services:

IV Led Zeppelin 1971 Rock	Nevermind Nirvana 1991 Rock	What's Going On Marvin Gaye 1971 Rock	Are You Experienced? Jimi Hendrix Experience 1967 Rock
The Joshua Tree U2 1987 Rock	Abbey Road The Beatles 1969 Rock	Rumours Fleetwood Mac 1977 Rock	Sun Sessions Elvis Presley 1976 Rock
Thriller Michael Jackson 1982 Pop	Exile on Main Street The Rolling Stones 1972 Rock	Born to Run Bruce Springsteen 1975 Rock	London Calling The Clash 1980 Rock

Be sure to click on the "information icon" in the top right-hand corner of the UI. This 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 become important in the next lab!

Demo - Lab 5 - 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 PostgreSQL and MongoDB databases.

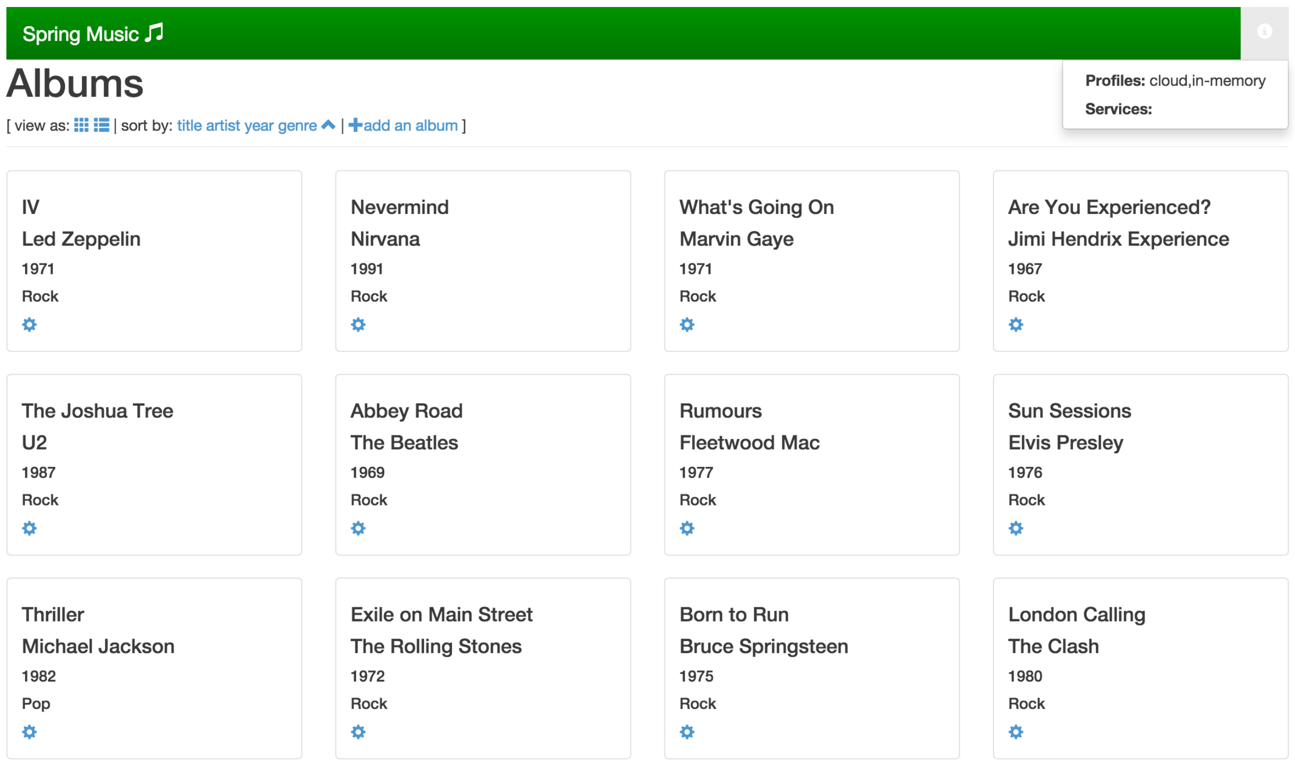
Cloud Foundry services are managed through two primary types of operations:

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.

A Bit of Review

Your instance of Spring Music should still be running from the end of Lab 7. Visit the application in your browser by hitting the route that was generated by the CLI:


Spring Music

Profiles: cloud,in-memory

Services:

Albums

[view as: | sort by: title artist year genre | +add an album]

IV Led Zeppelin 1971 Rock	Nevermind Nirvana 1991 Rock	What's Going On Marvin Gaye 1971 Rock	Are You Experienced? Jimi Hendrix Experience 1967 Rock
The Joshua Tree U2 1987 Rock	Abbey Road The Beatles 1969 Rock	Rumours Fleetwood Mac 1977 Rock	Sun Sessions Elvis Presley 1976 Rock
Thriller Michael Jackson 1982 Pop	Exile on Main Street The Rolling Stones 1972 Rock	Born to Run Bruce Springsteen 1975 Rock	London Calling The Clash 1980 Rock

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.

The Services Marketplace

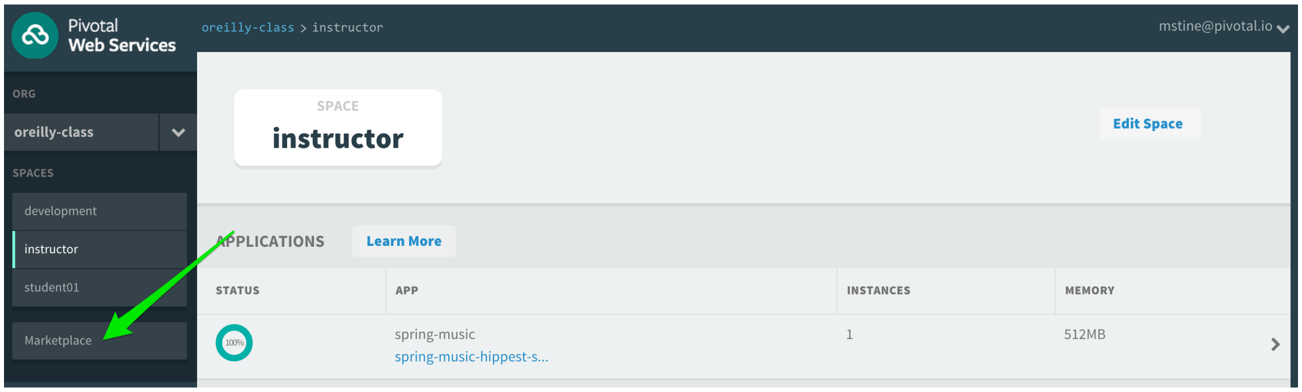
There are two ways to discover what services are available on Pivotal Web Services. The first is available on any instance of Cloud Foundry: the CLI. Just type:

```
$ cf marketplace
```

and you'll get a list of services, their available plans, and descriptions. On Pivotal Web Services, the "free" tier of plans is normally the first one listed (e.g. turtle for elephansql).

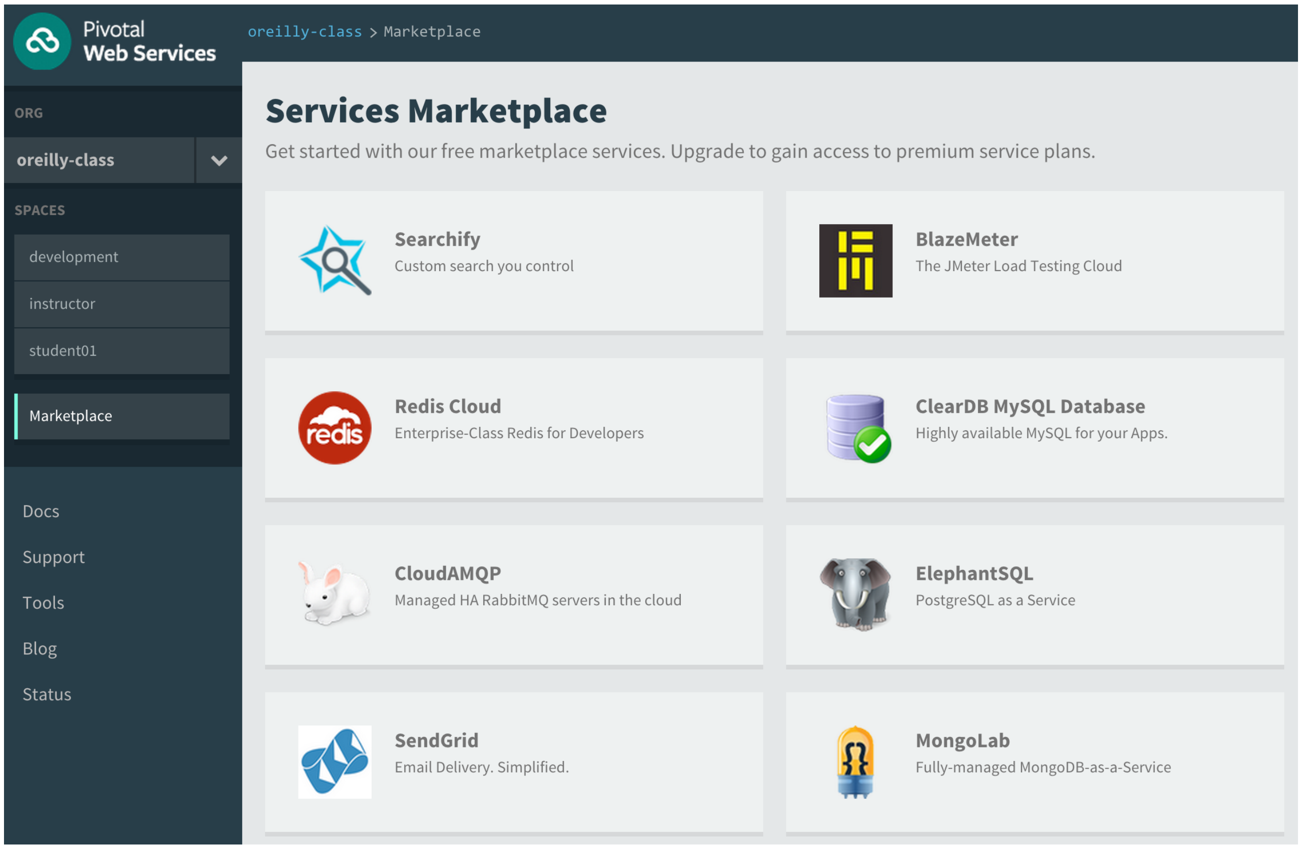
The second way is specific to PWS's Application Manager UI. If you haven't already, login to it by visiting http://console.run.pivotal.io.

Click on the "Marketplace" link:



STATUS	APP	INSTANCES	MEMORY
100%	spring-music spring-music-hippest-s...	1	512MB

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



Services Marketplace

Get started with our free marketplace services. Upgrade to gain access to premium service plans.

Searchify	Custom search you control
BlazeMeter	The JMeter Load Testing Cloud
Redis Cloud	Enterprise-Class Redis for Developers
ClearDB MySQL Database	Highly available MySQL for your Apps.
CloudAMQP	Managed HA RabbitMQ servers in the cloud
ElephantSQL	PostgreSQL as a Service
SendGrid	Email Delivery. Simplified.
MongoLab	Fully-managed MongoDB-as-a-Service

Demo - Creating and Binding to a Service Instance

1. Let's begin by creating a PostgreSQL instance provided by ElephantSQL. From the CLI, let's create a free ElephantSQL service instance:

```
$ cf create-service elephantsql turtle spring-music-db
Creating service spring-music-db in org oreilly-class / space instruct
or as mstine@pivotal.io...
OK
```

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 oreilly-cla
ss / space instructor as mstine@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": { (1)
  "elephantsql": [
   {
    "credentials": {
     "max_conns": "5",
     "uri": "postgres://xnxzdeao:ntLT0M10gz014oXnKkkjeIqLn3Td2EgH@baba
r.elephantsql.com:5432/xnxzdeao" (2)
    },
    "label": "elephantsql",
    "name": "spring-music-db",
    "plan": "turtle",
    "tags": [
     "Data Stores",
     "Cloud Databases",
     "Developer Tools",
     "Data Store",
     "postgresql",
     "relational",
     "New Product"
    ]
   }
  ]
 }
}
```

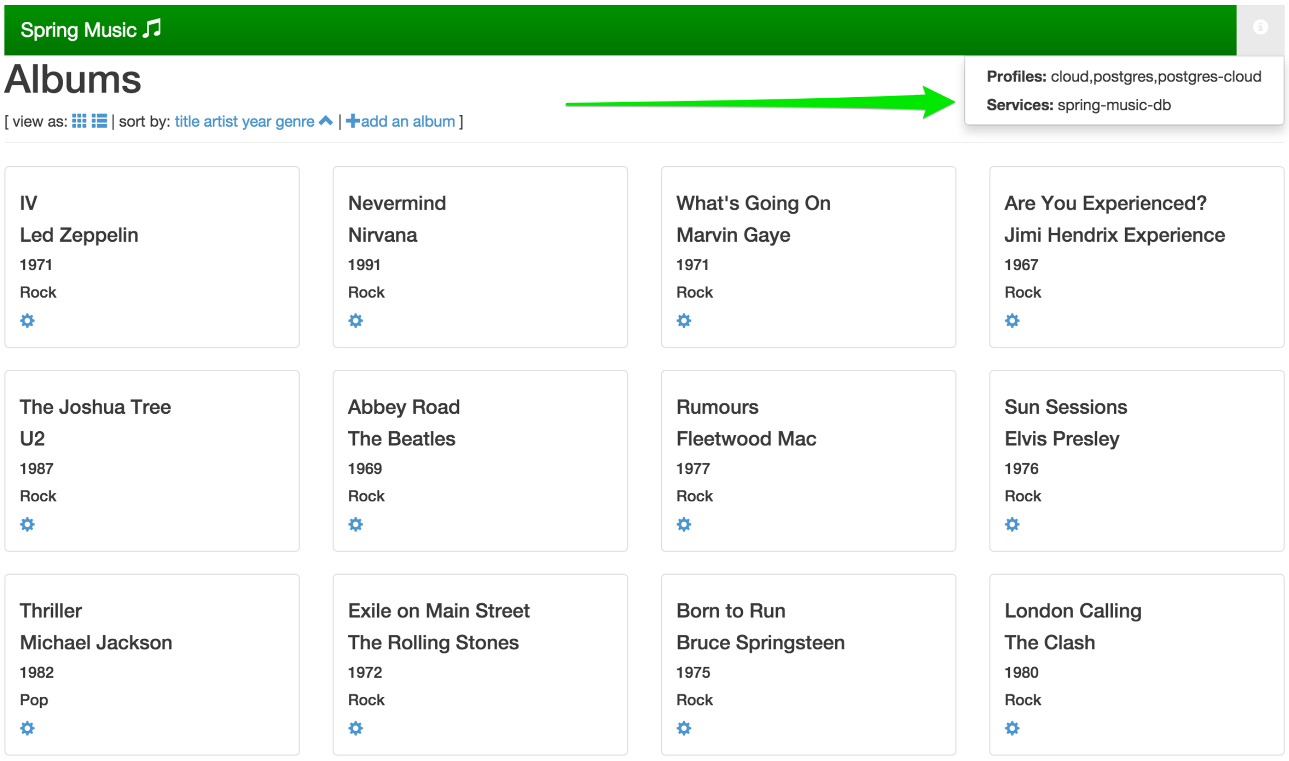
VCAP_SERVICES is a special Cloud Foundry environment variable that contains a JSON document containing all of the information for any services bound to an application.

Notice here the unique URI for this instance of PostgreSQL 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:


Spring Music

Albums

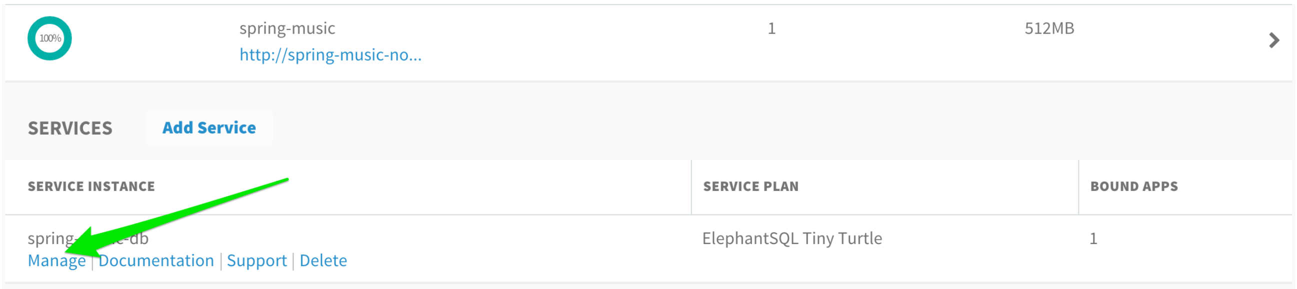
[view as: | sort by: title artist year genre | add an album]

Profiles: cloud,postgres,postgres-cloud
Services: spring-music-db

IV Led Zeppelin 1971 Rock	Nevermind Nirvana 1991 Rock	What's Going On Marvin Gaye 1971 Rock	Are You Experienced? Jimi Hendrix Experience 1967 Rock
The Joshua Tree U2 1987 Rock	Abbey Road The Beatles 1969 Rock	Rumours Fleetwood Mac 1977 Rock	Sun Sessions Elvis Presley 1976 Rock
Thriller Michael Jackson 1982 Pop	Exile on Main Street The Rolling Stones 1972 Rock	Born to Run Bruce Springsteen 1975 Rock	London Calling The Clash 1980 Rock

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

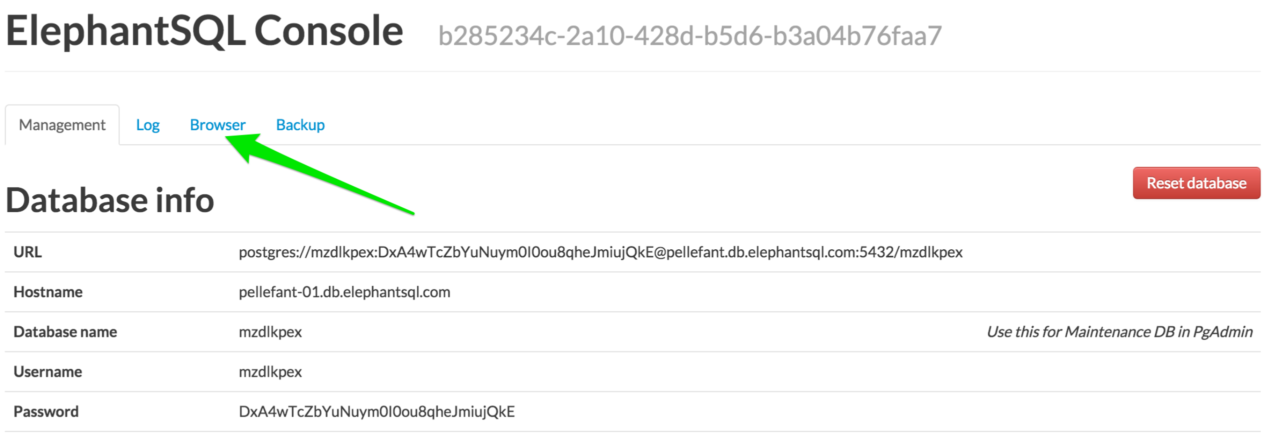
5. Let's take a direct look at the data in PostgreSQL by utilizing the Manage link in PWS App Manager:


100% spring-music http://spring-music-no... 1 512MB

SERVICES Add Service

SERVICE INSTANCE	SERVICE PLAN	BOUND APPS
spring-music-db Manage | Documentation | Support | Delete	ElephantSQL Tiny Turtle	1

6. Click on the Browser link:


ElephantSQL Console b285234c-2a10-428d-b5d6-b3a04b76faa7

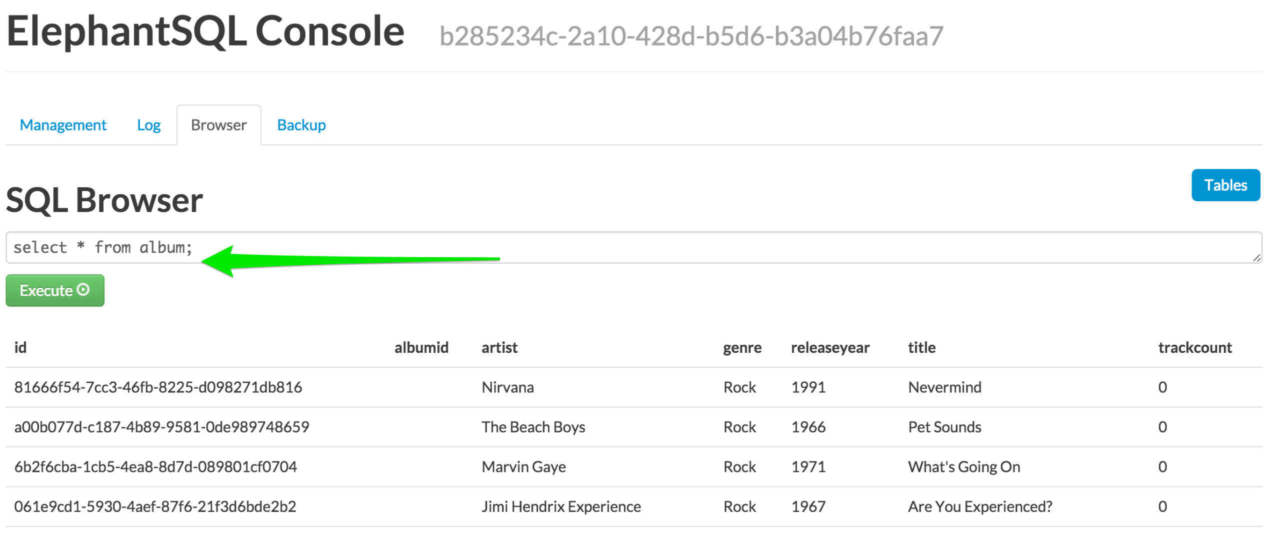
Management Log Browser Backup

Database info

Reset database

URL	postgres://mzdlkpex:DxA4wTcZbYuNuym0l0ou8qheJmiujQkE@pellefant.db.elephantsql.com:5432/mzdlkpex	
Hostname	pellefant-01.db.elephantsql.com	
Database name	mzdlkpex	*Use this for Maintenance DB in PgAdmin*
Username	mzdlkpex	
Password	DxA4wTcZbYuNuym0l0ou8qheJmiujQkE	

7. Paste in the query select * from album; and click Execute:



```
select * from album;
```

id	albumid	artist	genre	releaseyear	title	trackcount
81666f54-7cc3-46fb-8225-d098271db816		Nirvana	Rock	1991	Nevermind	0
a00b077d-c187-4b89-9581-0de989748659		The Beach Boys	Rock	1966	Pet Sounds	0
6b2f6cba-1cb5-4ea8-8d7d-089801cf0704		Marvin Gaye	Rock	1971	What's Going On	0
061e9cd1-5930-4aef-87f6-21f3d6bde2b2		Jimi Hendrix Experience	Rock	1967	Are You Experienced?	0

Demo - Lab 6 - Monitoring Applications with Pivotal Cloud Foundry

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.

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
mstine@pivotal.io…

time event actor description
2015-02-13T15:18:33.00-0600 audit.app.update mstine@pivotal.io instances: 1 (6)
2015-02-13T15:04:34.00-0600 audit.app.update mstine@pivotal.io instances: 5 (5)
2015-02-13T12:56:35.00-0600 audit.app.update mstine@pivotal.io state: STARTED
(4)
2015-02-13T12:56:26.00-0600 audit.app.update mstine@pivotal.io (3)
2015-02-13T12:56:26.00-0600 audit.app.map-route mstine@pivotal.io (2)
2015-02-13T12:56:24.00-0600 audit.app.create mstine@pivotal.io instances: 1,
memory: 512, state: STOPPED, environment_json: PRIVATE DATA HIDDEN (1)
```

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).

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

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

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

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

And here's the app.update event recording our scaling of the application back down with instances: 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
mstine@pivotal.io…
OK
```

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
mstine@pivotal.io…
```

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

Start the application again:

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

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

For Developers
Application developers can view application logs using the CF CLI.

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 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 (1)
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 (2)
2015-02-13T15:04:33.51-0600 [DEA/12] OUT Starting app instance (index 2) with
guid 7a428901-1691-4cce-b7f6-62d186c5cb55 (3)
2015-02-13T15:04:33.71-0600 [DEA/4] OUT Starting app instance (index 3) with
guid 7a428901-1691-4cce-b7f6-62d186c5cb55
```

An “Apache-style” access log event from the (Go)Router

An API log event that corresponds to an event as shown in cf events

A DEA log event indicating the start of an application instance on that DEA.

Example 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.EndpointMBeanExporter : 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.EndpointMBeanExporter : 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.EndpointMBeanExporter : 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.TomcatEmbeddedServletContainer : 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.

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.

Health

1. Demonstrate scale demo application

Lab 7 - Push Cloud Native Apps to Pivotal Cloud Foundry

Build and Push!

1. Return to the Fortune Teller application directory that you finished in Lab 3:

```
$ cd $COURSE_HOME/labs/initial/fortune-teller
```

2. Using Maven, build and package the application:

```
$ mvn package
```

Maven will automatically download all of Fortune Teller's dependencies. This may take a few moments.

3. Push the Spring Cloud services:

```
$ cf push -f manifest-services.yml
```

This will push the Spring Cloud Config Server, a Eureka server, and a Hystrix Dashboard, all with random routes.

4. Edit scripts/create_services.sh to add the random routes that were generated for you:

```
cf cups config-service -p '{"uri":"http://config-server-fluxional-sutt
ee.cfapps.io"}'
cf cups service-registry -p '{"uri":"http://eureka-unprevalent-toper.c
fapps.io"}'
cf cs elephantsql turtle fortunes-db
```

5. Run scripts/create-services.sh to create the services that you need:

```
$ scripts/create_services.sh
```

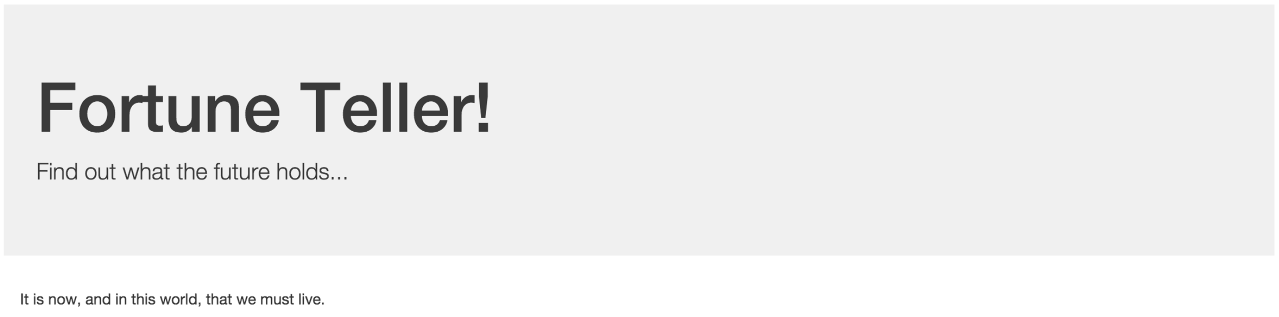
```
Creating user provided service config-service in org platform-eng / sp
ace nfjs-workshop as mstine@pivotal.io...
OK
Creating user provided service service-registry in org platform-eng /
space nfjs-workshop as mstine@pivotal.io...
OK
Creating service fortunes-db in org platform-eng / space nfjs-workshop
 as mstine@pivotal.io...
OK
```

6. Push the microservices:

```
$ cf push -f manifest-apps.yml
```

This will push the fortunes service and the ui application.

7. In a browser, access the fortunes-ui application at the route that was created for you:

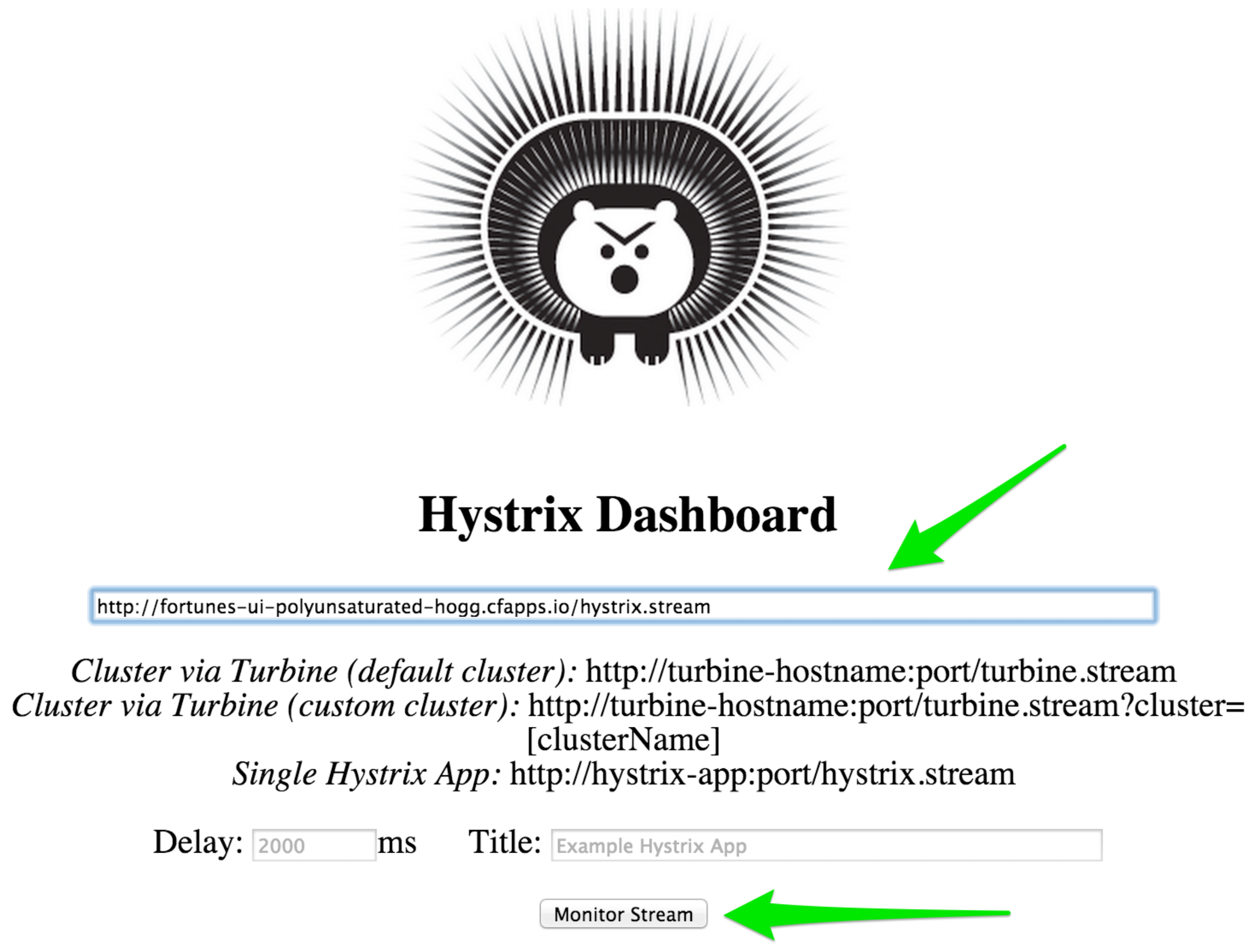


Fortune Teller!

Find out what the future holds...

It is now, and in this world, that we must live.

8. Now, in another browser tab, access the Hystrix Dashboard at the route that was created for you. Enter the route for the UI application and click the “Monitor Stream.”



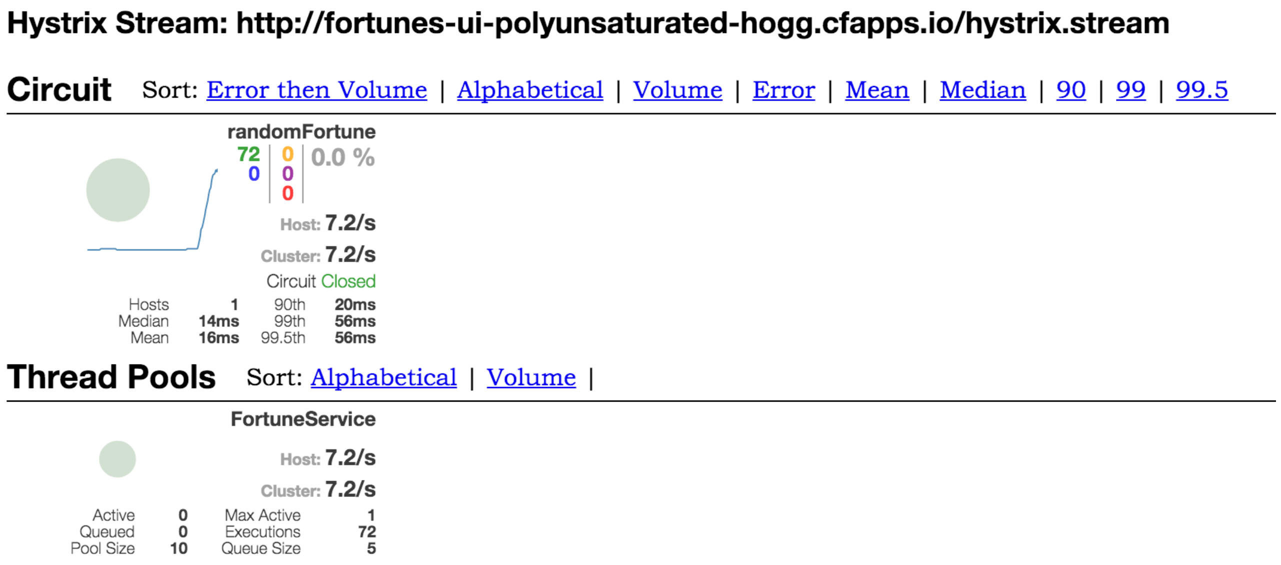
Hystrix Dashboard

http://fortunes-ui-polyunsaturated-hogg.cfapps.io/hystrix.stream

Cluster via Turbine (default cluster): http://turbine-hostname:port/turbine.stream
Cluster via Turbine (custom cluster): http://turbine-hostname:port/turbine.stream?cluster=[clusterName]
Single Hystrix App: http://hystrix-app:port/hystrix.stream

Delay: 2000 ms Title: Example Hystrix App

9. Access the fortunes-ui and show that the circuit breaker is registering successful requests.



Hystrix Stream: http://fortunes-ui-polyunsaturated-hogg.cfapps.io/hystrix.stream

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

```
randomFortune
72 | 0 | 0.0 %
 0 | 0
   | 0
Host: 7.2/s
Cluster: 7.2/s
Circuit Closed
 Hosts      1    90th   20ms
Median   14ms    99th   56ms
  Mean   16ms  99.5th   56ms
```

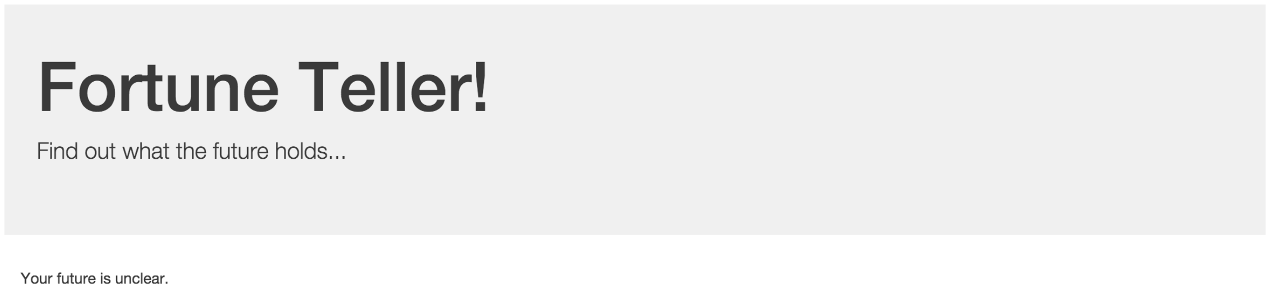
Thread Pools Sort: Alphabetical | Volume |

```
FortuneService
Host: 7.2/s
Cluster: 7.2/s
   Active   0  Max Active   1
   Queued   0  Executions  72
Pool Size  10  Queue Size   5
```

10. Stop the fortunes application:

```
$ cf stop fortunes
```

11. Access the fortunes-ui and see that the “fallback fortune” is being returned.

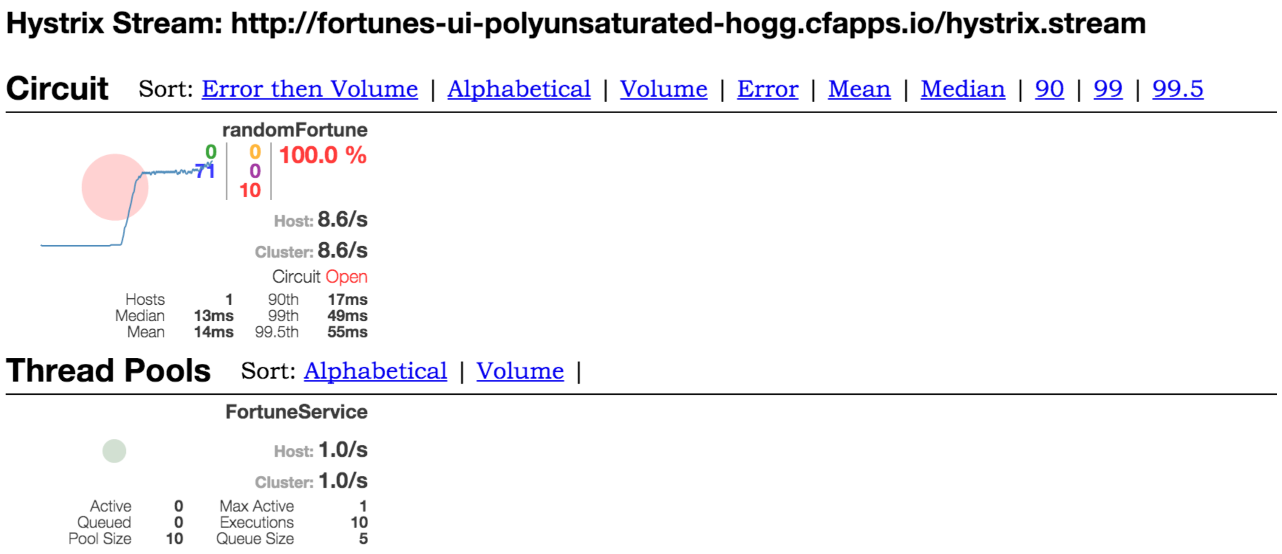


Fortune Teller!

Find out what the future holds...

Your future is unclear.

12. Access the fortunes-ui and show that the circuit breaker is registering short-circuited requests.



Hystrix Stream: http://fortunes-ui-polyunsaturated-hogg.cfapps.io/hystrix.stream

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

randomFortune

```
0   0 | 100.0 %
71  0
   10
        Host: 8.6/s
     Cluster: 8.6/s
     Circuit Open
 Hosts     1    90th   17ms
Median  13ms    99th   49ms
  Mean  14ms  99.5th   55ms
```

Thread Pools Sort: Alphabetical | Volume |

FortuneService

```
        Host: 1.0/s
     Cluster: 1.0/s
   Active   0  Max Active    1
   Queued   0  Executions   10
Pool Size  10  Queue Size    5
```

13. Start the fortunes application:

```
$ cf start fortunes
```

Continue to access the fortunes-ui and watch the dashboard. After the fortunes service has re-registered with Eureka and the fortunes-ui load balancer caches are refreshed, you will see the circuit breaker recover. You should then start getting random fortunes again!