Spring Cloud Config

Table of Contents

Estimated Time: 60 minutes

Requirements

Lab Requirements (/spring-cloud-services/requirements)

What You Will Learn

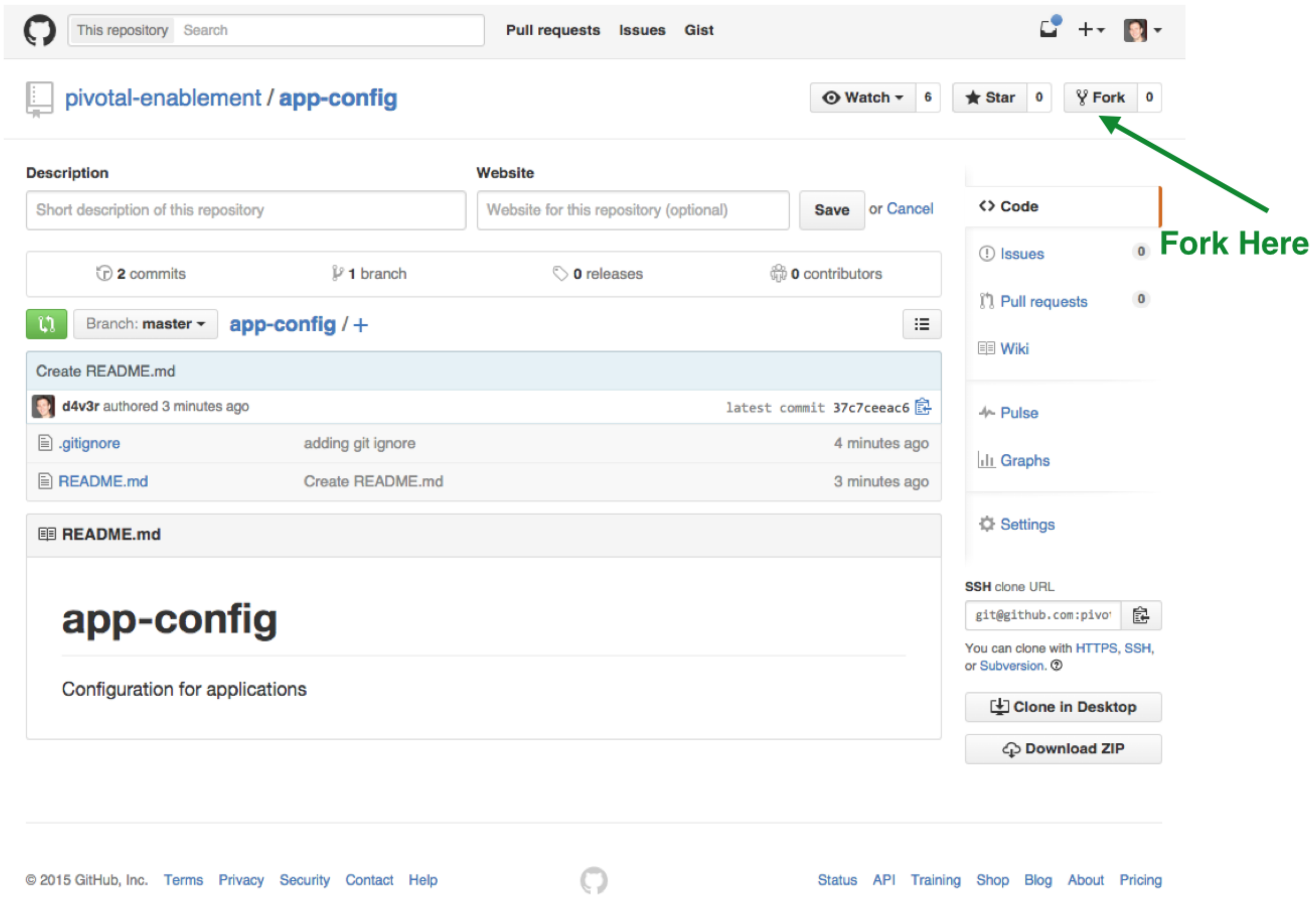
- How to set up a git repository to hold configuration data
- How to set up a config server (`config-server`) with a git backend
- How to set up a client (`greeting-config`) to pull configuration from the `config-server`
- How to change log levels for a running application (`greeting-config`)
- How to use `@ConfigurationProperties` to capture configuration changes (`greeting-config`)
- How to use `@RefreshScope` to capture configuration changes (`greeting-config`)
- How to override configuration values by profile (`greeting-config`)
- How to use Spring Cloud Service to provision and configure a Config Server
- How to use Cloud Bus to notify applications (`greeting-config`) to refresh configuration at scale

Exercises

Set up the `app-config` Repo

To start, we need a repository to hold our configuration.

1) Fork the configuration repo to your account. Browse to: https://github.com/pivotal-enablement/app-config (https://github.com/pivotal-enablement/app-config). Then fork the repo.



2) GitHub displays your new fork. Copy the HTTPS clone URL from your fork.

3) Open a new terminal window and clone the fork you just created (you may want to create a common location for your GitHub repos, such as ~/repos):

```
$ cd [location of your github repos, e.g. ~/repos]
$ git clone <Your fork of the app-config repo - HTTPS clone URL>
$ cd app-config
```

Notice that this repository is basically empty. This repository will be the source of configuration data.

Set up `config-server`

1) Review the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/config-server/pom.xml` By adding `spring-cloud-config-server` to the classpath, this application is eligible to embed a `config-server`.

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

2) Review the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/config-server/src/main/java/io/pivotal/ConfigServerApplication.java`

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

Note the `@EnableConfigServer` annotation. That embeds the `config-server`.

3) Set the GitHub repository for the `config-server`. This will be the source of the configuration data. *Edit the* `$SPRING_CLOUD_SERVICES_LABS_HOME/config-server/src/main/resources/application.yml` *file.*

```
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/d4v3r/app-config.git #<-- CHANGE ME
```

Make sure to substitute your forked `app-config` repository. Do not use the literal above.

4) Open a terminal window and start the `config-server`.

```
$ cd $SPRING_CLOUD_SERVICES_LABS_HOME/config-server
$ mvn clean spring-boot:run
```

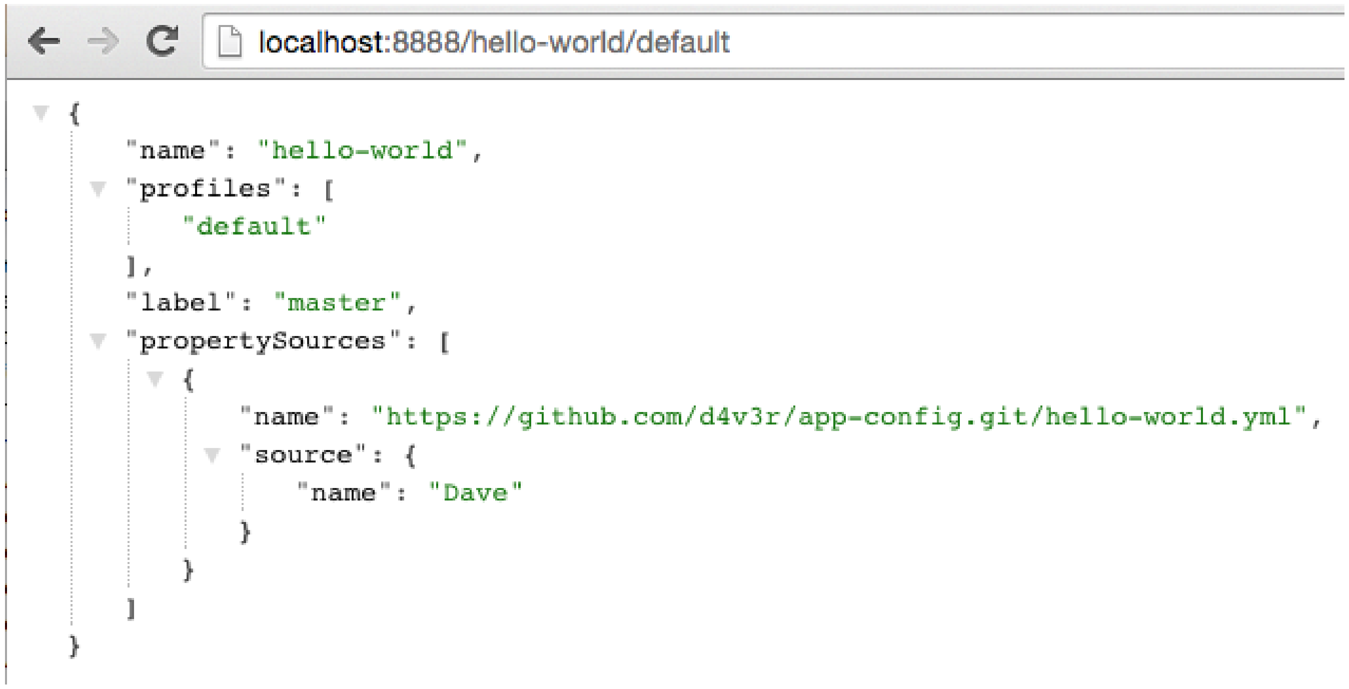
Your `config-server` will be running locally once you see a “Started ConfigServerApplication…” message. You will not be returned to a command prompt and must leave this window open.

5) Let’s add some configuration. Edit your fork of the `app-config` repo. Create a file called `hello-world.yml`. Add the content below to the file and push the changes back to GitHub. Be sure to substitute your name for `<Your name>`.

```
name: <Your Name>
```

6) Confirm the `config-server` is up and configured with a backing git repository by calling one of its endpoints (http://projects.spring.io/spring-cloud/docs/1.0.3/spring-cloud.html#_quick_start). Because the returned payload is JSON, we recommend using something that will pretty-print the document. A good tool for this is the Chrome JSON Formatter (https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?hl=en) plug-in.

Open a browser window and fetch the following url: http://localhost:8888/hello-world/default (http://localhost:8888/hello-world/default)



```
{
    "name": "hello-world",
    "profiles": [
        "default"
    ],
    "label": "master",
    "propertySources": [
        {
            "name": "https://github.com/d4v3r/app-config.git/hello-world.yml",
            "source": {
                "name": "Dave"
            }
        }
    ]
}
```

What Just Happened?

The `config-server` exposes several endpoints (http://projects.spring.io/spring-cloud/docs/1.0.3/spring-cloud.html#_quick_start) to fetch configuration.

In this case, we are manually calling one of those endpoints (`/{application}/{profile}[/{label}]`) to fetch configuration. We substituted our example client application `hello-world` as the `{application}` and the `default` profile as the `{profile}` . We didn’t specify the label to use so `master` is assumed. In the returned document, we see the configuration file `hello-world.yml` listed as a `propertySource` with the associated key/value pair. This is just an example, as you move through the lab you will add configuration for `greeting-config` (our client application).

Set up `greeting-config`

1) Review the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/pom.xml` By adding `spring-cloud-services-starter-config-client` to the classpath, this application will consume configuration from the `config-server`. `greeting-config` is a config client.

```
<dependency>
        <groupId>io.pivotal.spring.cloud</groupId>
        <artifactId>spring-cloud-services-starter-config-client</artifactId>
</dependency>
```

2) Review the `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/resources/bootstrap.yml`

```
spring:
  application:
    name: greeting-config
```

`spring.application.name` defines the name of the application. This value is used in several places within Spring Cloud: locating configuration files by name, service discovery/registration by name, etc. In this lab, it will be used to locate config files for the `greeting-config` application.

Absent from the bootstrap.yml is the `spring.cloud.config.uri`, which defines how `greeting-config` reaches the `config-server`. Since there is no `spring.cloud.config.uri` defined in this file, the default value of `http://localhost:8888` is used. Notice that this is the same host and port of the `config-server` application.

3) Open a new terminal window. Start the `greeting-config` application:

```
$ cd $SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config
$ mvn clean spring-boot:run
```

4) Confirm the `greeting-config` app is up. Browse to http://localhost:8080 (http://localhost:8080). You should be prompted to authenticate. Why? `spring-cloud-services-starter-config-client` has a dependency on Spring Security (http://projects.spring.io/spring-security/). Unless the given application has other security configuration, this will cause all application and actuator endpoints to be protected by HTTP Basic authentication.

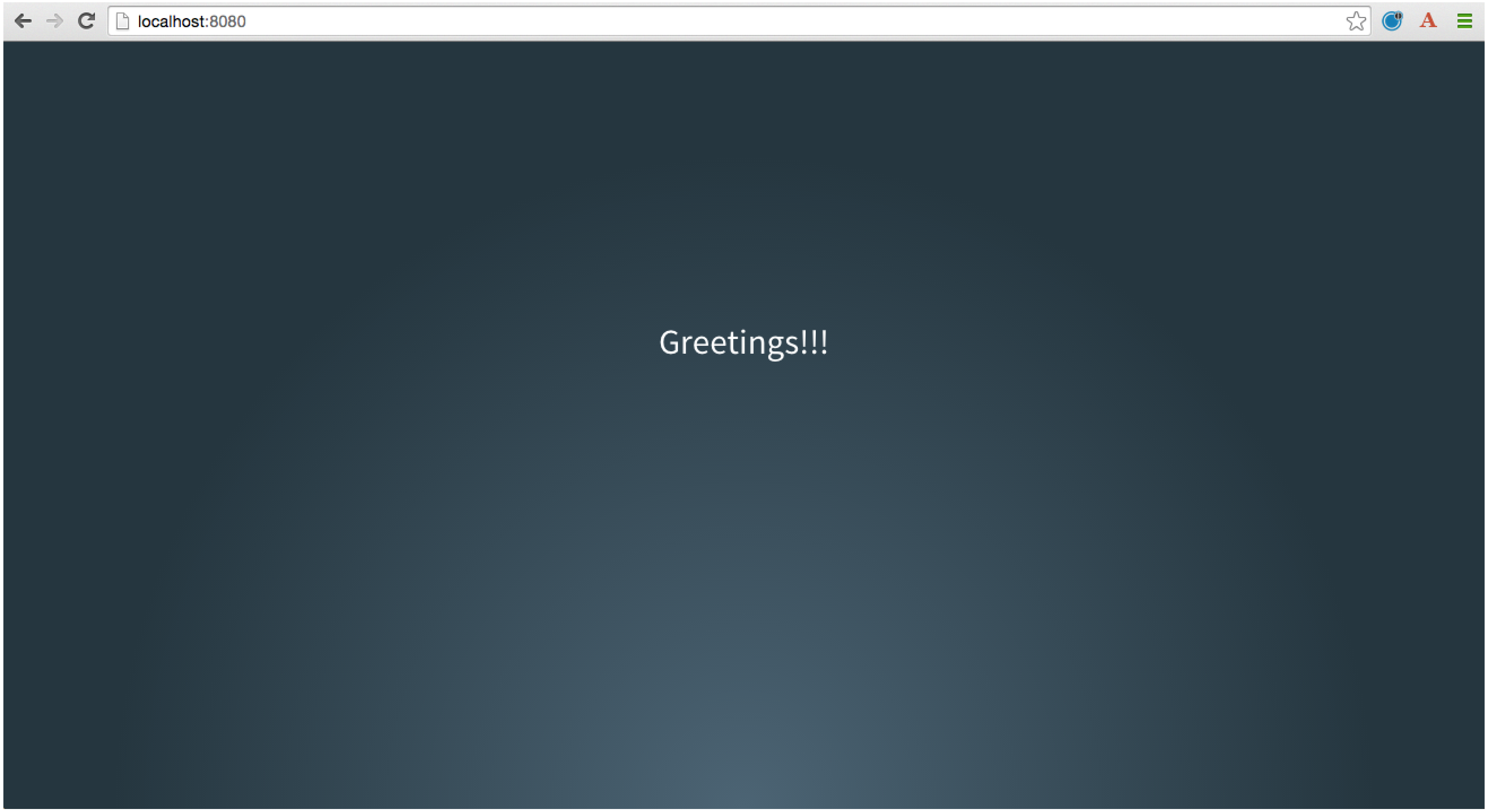
5) If no explicit username or password has been set then Spring Security will generate one for you. This is applies for the `greeting-config` application. Use the following to login:

username: `user`

password: You can find this in the terminal output. Look for a log message similar to the following: `Using default security password: 90a3ef2a-4e98-4491-a528-a47a7162dd2a`. Use this password to login.

Note: Username and password can be explicitly set through the `security.user.name` and `security.user.password` configuration parameters.

6) After logging in you should see the message “Greetings!!!”.



Greetings!!!

What Just Happened?

At this point, you connected the `greeting-config` application with the `config-server`. This can be confirmed by reviewing the logs of the `greeting-config` application.

`greeting-config` log output:

```
2015-09-18 13:48:50.147  INFO 15706 --- [lication.main()] b.c.PropertySourceBootstrapConfigu
ration :
Located property source: CompositePropertySource [name='configService', propertySources=[]]
```

There is still no configuration in the git repo for the `greeting-config` application, but at this point we have everything wired (`greeting-config → config-server → app-config` repo) so we can add configuration parameters/values and see the effects in out client application `greeting-config`.

Configuration parameters/values will be added as we move through the lab.

7) Stop the `greeting-config` application

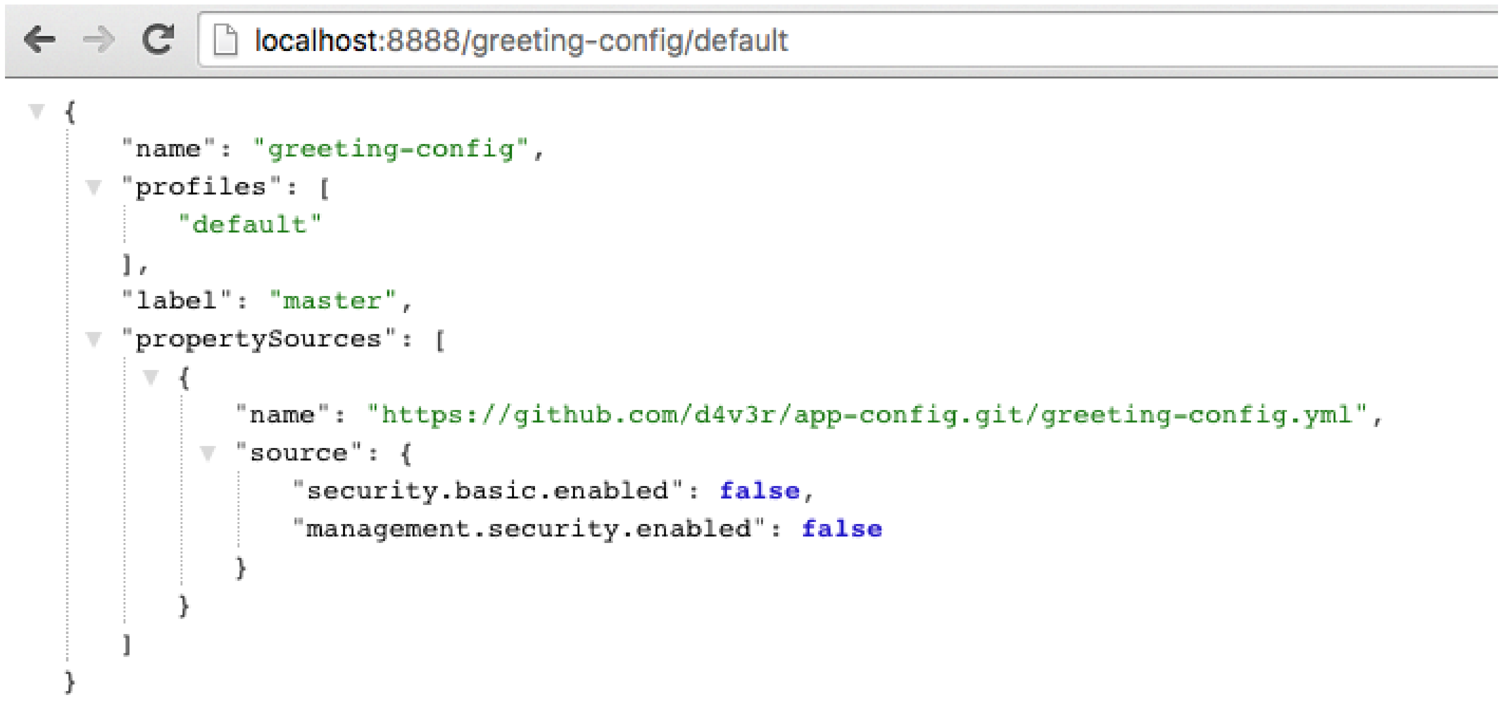
Unsecure the Endpoints

For these labs we don't need Spring Security's default behavior of securing every endpoint. This will be our first example of using the `config-server` to provide configuration for the `greeting-config` application.

1) Edit your fork of the `app-config` repo. Create a file called `greeting-config.yml`. Add the content below to the file and push the changes back to GitHub.

```
security:
  basic:
    enabled: false # turn of securing our application endpoints

management:
  security:
    enabled: false # turn of securing the actuator endpoints
```

2) Browse to http://localhost:8888/greeting-config/default (http://localhost:8888/greeting-config/default) to review the configuration the `config-server` is providing for `greeting-config` application.



```
{
    "name": "greeting-config",
    "profiles": [
        "default"
    ],
    "label": "master",
    "propertySources": [
        {
            "name": "https://github.com/d4v3r/app-config.git/greeting-config.yml",
            "source": {
                "security.basic.enabled": false,
                "management.security.enabled": false
            }
        }
    ]
}
```

3) Start the `greeting-config` application:

```
$ mvn clean spring-boot:run
```

4) Review the logs for the `greeting-config` application. You can see that configuration is being sourced from the `greeting-config.yml` file.

```
2015-11-02 08:57:32.962  INFO 58597 --- [lication.main()] b.c.PropertySourceBootstrapConfigu
ration : Located property source: CompositePropertySource [name='configService', propertySou
rces=[MapPropertySource [name='https://github.com/d4v3r/app-config.git/greeting-config.ym
l']]]
```

5) Browse to http://localhost:8080 (http://localhost:8080). You should no longer be prompted to authenticate.

Changing Logging Levels

Next you will change the logging level of the `greeting-config` application.

1) View the `getGreeting()` method of the `GreetingController` class (`$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/greeting/GreetingController.java`).

```
@RequestMapping("/")
String getGreeting(Model model){

  logger.debug("Adding greeting");
  model.addAttribute("msg", "Greetings!!!");

  if(greetingProperties.isDisplayFortune()){
    logger.debug("Adding fortune");
    model.addAttribute("fortune", fortuneService.getFortune());
  }

  //resolves to the greeting.vm velocity template
  return "greeting";
}
```

We want to see these debug messages. By default only log levels of `ERROR`, `WARN` and `INFO` will be logged. You will change the log level to `DEBUG` using configuration. All log output will be directed to `System.out` & `System.error` by default, so logs will be output to the terminal window(s).

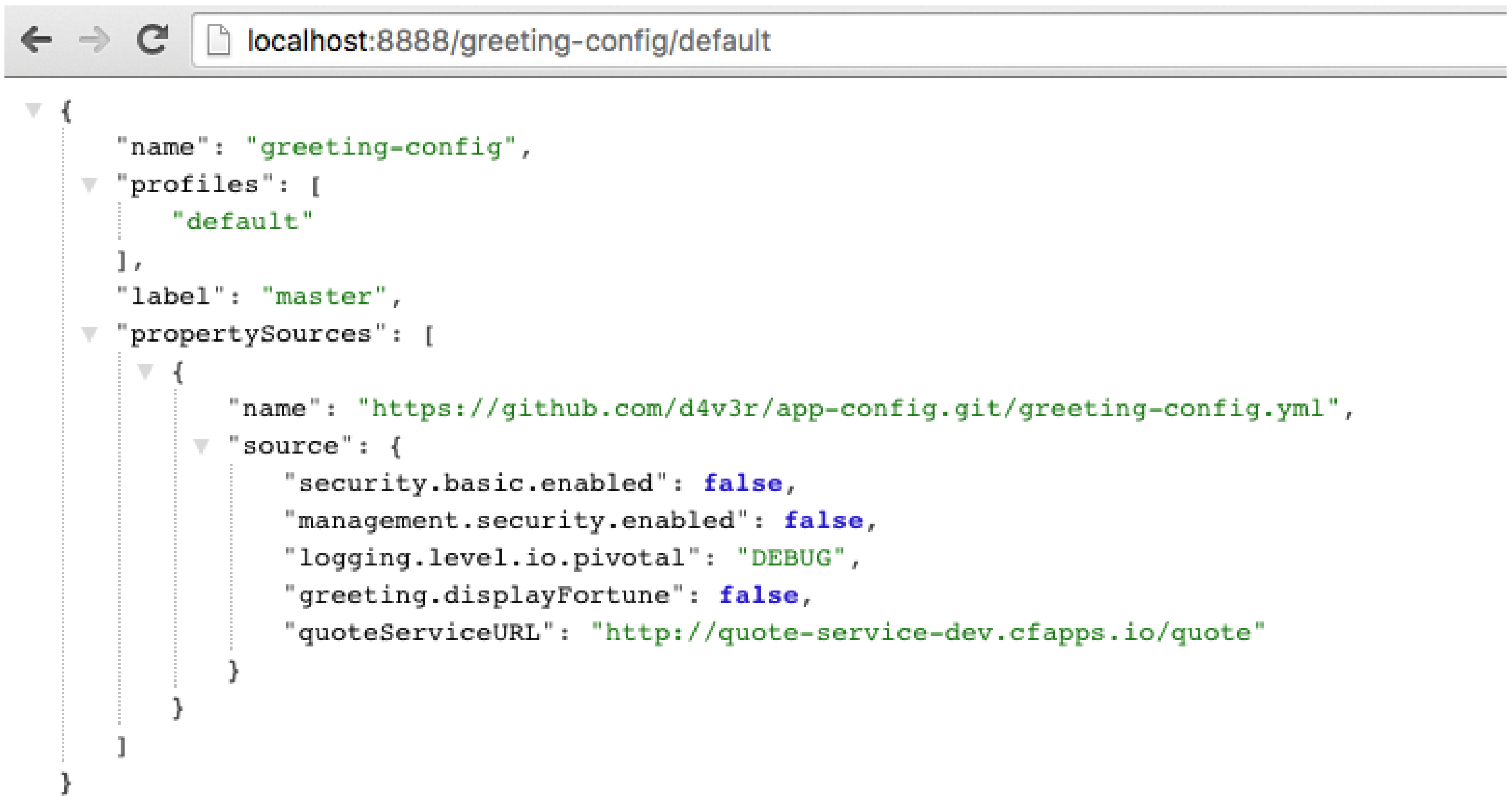
2) In your fork of the `app-config` repo. Add the content below to the `greeting-config.yml` file and push the changes back to GitHub.

```
security:
  basic:
    enabled: false

management:
  security:
    enabled: false

logging: # <----New sections below
  level:
    io:
      pivotal: DEBUG

greeting:
  displayFortune: false

quoteServiceURL: http://quote-service-dev.cfapps.io/quote
```

We have added several configuration parameters that will be used throughout this lab. For this exercise, we have set the log level for classes in the `io.pivotal` package to `DEBUG`.

3) While watching the `greeting-config` terminal, refresh the http://localhost:8080 (http://localhost:8080/) url. Notice there are no `DEBUG` logs yet.

4) Does the `config-server` see the change in your git repo? Let's check what the `config-server` is serving. Browse to http://localhost:8888/greeting-config/default (http://localhost:8888/greeting-config/default)



localhost:8888/greeting-config/default

```
{
    "name": "greeting-config",
    "profiles": [
        "default"
    ],
    "label": "master",
    "propertySources": [
        {
            "name": "https://github.com/d4v3r/app-config.git/greeting-config.yml",
            "source": {
                "security.basic.enabled": false,
                "management.security.enabled": false,
                "logging.level.io.pivotal": "DEBUG",
                "greeting.displayFortune": false,
                "quoteServiceURL": "http://quote-service-dev.cfapps.io/quote"
            }
        }
    ]
}
```

The propertySources value has changed! The `config-server` has picked up the changes to the git repo. (If you don't see the change, verify that you have pushed the greeting-config.yml to GitHub.)

5) Review the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/pom.xml`. For the `greeting-config` application to pick up the configuration changes, it must include the `actuator` dependency. The `actuator` adds several additional endpoints to the application for operational visibility and tasks that need to be carried out. In this case, we have added the actuator so that we can use the `/refresh` endpoint, which allows us to refresh the application config on demand.

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

6) For the `greeting-config` application to pick up the configuration changes, it must be told to do so. Notify `greeting-config` app to pick up the new config by POSTing to the `greeting-config /refresh` endpoint. Open a new terminal window and execute the following:

```
$ curl -X POST http://localhost:8080/refresh
```

7) Refresh the `greeting-config` http://localhost:8080 (http://localhost:8080/) url while viewing the `greeting-config` terminal. You should see the debug line “Adding greeting”

Congratulations! You have used the `config-server` and `actuator` to change the logging level of the `greeting-config` application without restarting the `greeting-config` application.

Turning on a Feature with `@ConfigurationProperties`

Use of `@ConfigurationProperties` is a common way to externalize, group, and validate configuration in Spring applications. `@ConfigurationProperties` beans are automatically rebound when application config is refreshed.

1) Review `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/greeting/GreetingProperties.java`. Use of the `@ConfigurationProperties` annotation allows for reading of configuration values. Configuration keys are a combination of the `prefix` and the field names. In this case, there is one field (`displayFortune`). Therefore `greeting.displayFortune` is used to turn the display of fortunes on/off. Remaining code is typical getter/setters for the fields.

```
@ConfigurationProperties(prefix="greeting")
public class GreetingProperties {

        private boolean displayFortune;

        public boolean isDisplayFortune() {
                return displayFortune;
        }

        public void setDisplayFortune(boolean displayFortune) {
                this.displayFortune = displayFortune;
        }
}
```

2) Review `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/greeting/GreetingController.java`. Note how the `greetingProperties.isDisplayFortune()` is used to turn the display of fortunes on/off. There are times when you want to turn features on/off on demand. In this case, we want the fortune feature “on” with our greeting.

```
@EnableConfigurationProperties(GreetingProperties.class)
public class GreetingController {

        Logger logger = LoggerFactory
                        .getLogger(GreetingController.class);



        @Autowired
        GreetingProperties greetingProperties;

        @Autowired
        FortuneService fortuneService;

        @RequestMapping("/")
        String getGreeting(Model model){

                logger.debug("Adding greeting");
                model.addAttribute("msg", "Greetings!!!");

                if(greetingProperties.isDisplayFortune()){
                        logger.debug("Adding fortune");
                        model.addAttribute("fortune", fortuneService.getFortune());
                }

                //resolves to the greeting.vm velocity template
                return "greeting";
        }
}
```

3) Edit your fork of the `app-config` repo. Change `greeting.displayFortune` from `false` to `true` in the `greeting-config.yml` and push the changes back to GitHub.

```
security:
  basic:
    enabled: false

management:
  security:
    enabled: false

logging:
  level:
    io:
      pivotal: DEBUG

greeting:
  displayFortune: true # <----Change to true

quoteServiceURL: http://quote-service-dev.cfapps.io/quote
```

4) Notify `greeting-config` app to pick up the new config by POSTing to the `/refresh` endpoint.

```
$ curl -X POST http://localhost:8080/refresh
```

5) Then refresh the http://localhost:8080 (http://localhost:8080/) url and see the fortune included.

Congratulations! You have turned on a feature without restarting using the `config-server`, `actuator` and `@ConfigurationProperties`.

Reinitializing Beans with `@RefreshScope`

Now you will use the `config-server` to obtain a service URI rather than hardcoding it your application code.

Beans annotated with the `@RefreshScope` will be recreated when refreshed so they can pick up new config values.

1) Review `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/quote/QuoteService.java`. `QuoteService` uses the `@RefreshScope` annotation. Beans with the `@RefreshScope` annotation will be recreated when refreshing configuration. The `@Value` annotation allows for injecting the value of the `quoteServiceURL` configuration parameter.

In this case, we are using a third party service to get quotes. We want to keep our environments aligned with the third party. So we are going to override configuration values by profile (next section).

```
@Service
@RefreshScope
public class QuoteService {
        Logger logger = LoggerFactory
                        .getLogger(QuoteController.class);

        @Value("${quoteServiceURL}")
        private String quoteServiceURL;

        public String getQuoteServiceURI() {
                return quoteServiceURL;
        }

        public Quote getQuote(){
                logger.info("quoteServiceURL: {}", quoteServiceURL);
                RestTemplate restTemplate = new RestTemplate();
                Quote quote = restTemplate.getForObject(
                                quoteServiceURL, Quote.class);
                return quote;
        }
}
```

2) Review `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/src/main/java/io/pivotal/quote/QuoteController.java`. `QuoteController` calls the `QuoteService` for quotes.

```
@Controller
public class QuoteController {

        Logger logger = LoggerFactory
                        .getLogger(QuoteController.class);

        @Autowired
        private QuoteService quoteService;

        @RequestMapping("/random-quote")
        String getView(Model model) {

                model.addAttribute("quote", quoteService.getQuote());
                model.addAttribute("uri", quoteService.getQuoteServiceURI());
                return "quote";
        }
}
```

3) In your browser, hit the http://localhost:8080/random-quote (http://localhost:8080/random-quote) url. Note where the data is being served from: `http://quote-service-dev.cfapps.io/quote`

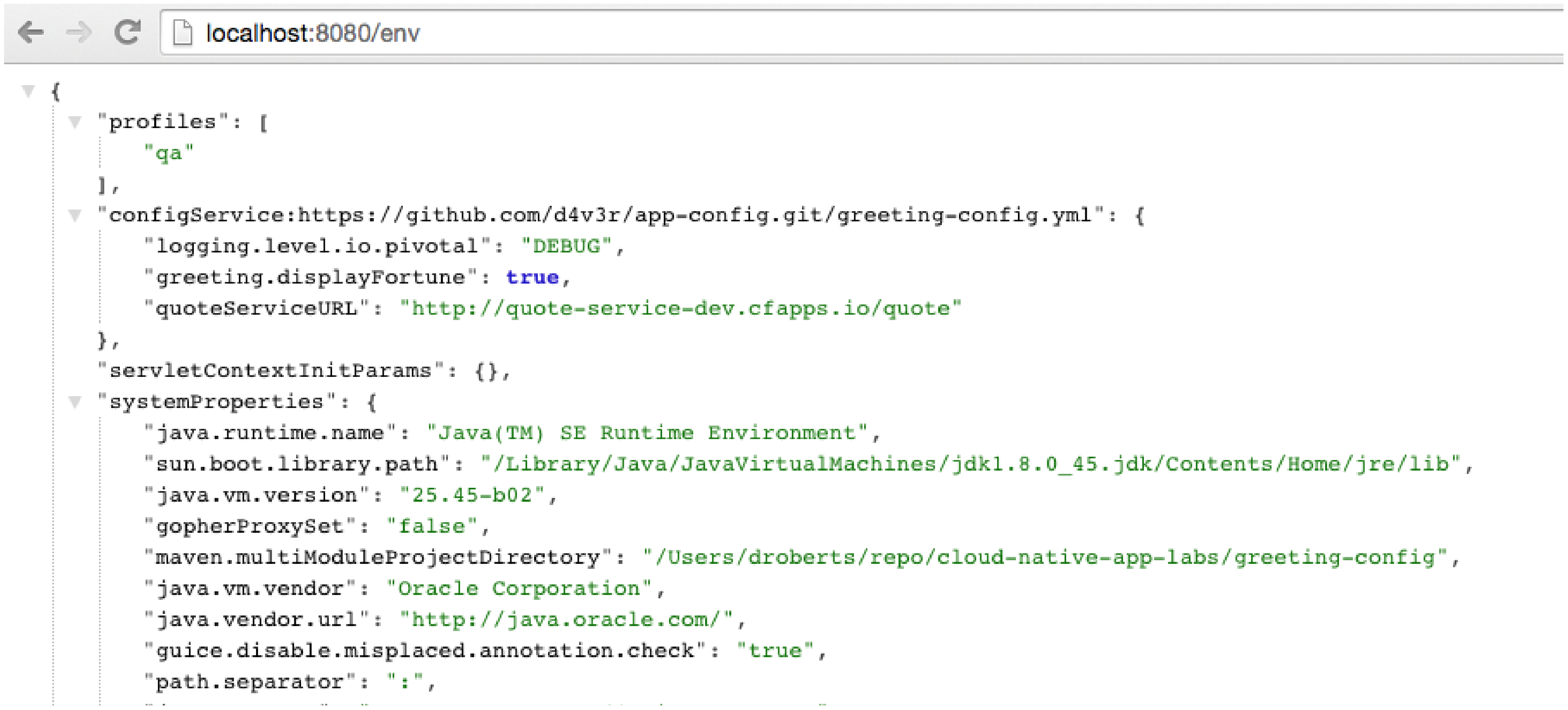
Override Configuration Values By Profile

1) Stop the `greeting-config` application using Command-C or CTRL-C in the terminal window.

2) Set the active profile to qa for the `greeting-config` application. In the example below, we use an environment variable to set the active profile.

```
[mac, linux]
$ SPRING_PROFILES_ACTIVE=qa mvn clean spring-boot:run

[windows]
$ set SPRING_PROFILES_ACTIVE=qa
$ mvn clean spring-boot:run
```

2) Make sure the profile is set by browsing to the http://localhost:8080/env (http://localhost:8080/env) endpoint (provided by `actuator`). Under profiles `qa` should be listed.



localhost:8080/env

```
{
  "profiles": [
      "qa"
  ],
  "configService:https://github.com/d4v3r/app-config.git/greeting-config.yml": {
      "logging.level.io.pivotal": "DEBUG",
      "greeting.displayFortune": true,
      "quoteServiceURL": "http://quote-service-dev.cfapps.io/quote"
  },
  "servletContextInitParams": {},
  "systemProperties": {
      "java.runtime.name": "Java(TM) SE Runtime Environment",
      "sun.boot.library.path": "/Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/jre/lib",
      "java.vm.version": "25.45-b02",
      "gopherProxySet": "false",
      "maven.multiModuleProjectDirectory": "/Users/droberts/repo/cloud-native-app-labs/greeting-config",
      "java.vm.vendor": "Oracle Corporation",
      "java.vendor.url": "http://java.oracle.com/",
      "guice.disable.misplaced.annotation.check": "true",
      "path.separator": ":",
```

3) In your fork of the `app-config` repository, create a new file: `greeting-config-qa.yml`. Fill it in with the following content:

```
quoteServiceURL: http://quote-service-qa.cfapps.io/quote
```

Make sure to commit and push to GitHub.

4) Browse to http://localhost:8080/random-quote (http://localhost:8080/random-quote). Quotes are still being served from `http://quote-service-dev.cfapps.io/quote`.

5) Refresh the application configuration values

```
$ curl -X POST http://localhost:8080/refresh
```

6) Refresh the http://localhost:8080/random-quote (http://localhost:8080/random-quote) url. Quotes are now being served from QA.

7) Stop both the `config-server` and `greeting-config` applications.

What Just Happened?

Configuration from `greeting-config.yml` was overridden by a configuration file that was more specific (`greeting-config-qa.yml`).

Deploy the `greeting-config` Application to PCF

1) Package the `greeting-config` application. Execute the following from the `greeting-config` directory:

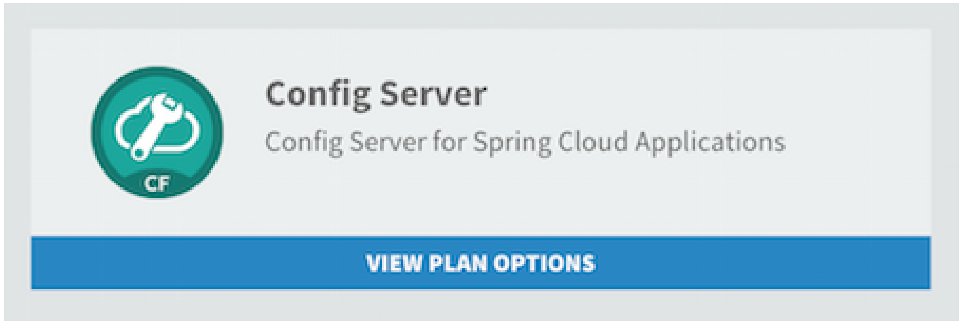
```
$ mvn clean package
```

2) Deploy the `greeting-config` application to PCF, without starting the application:

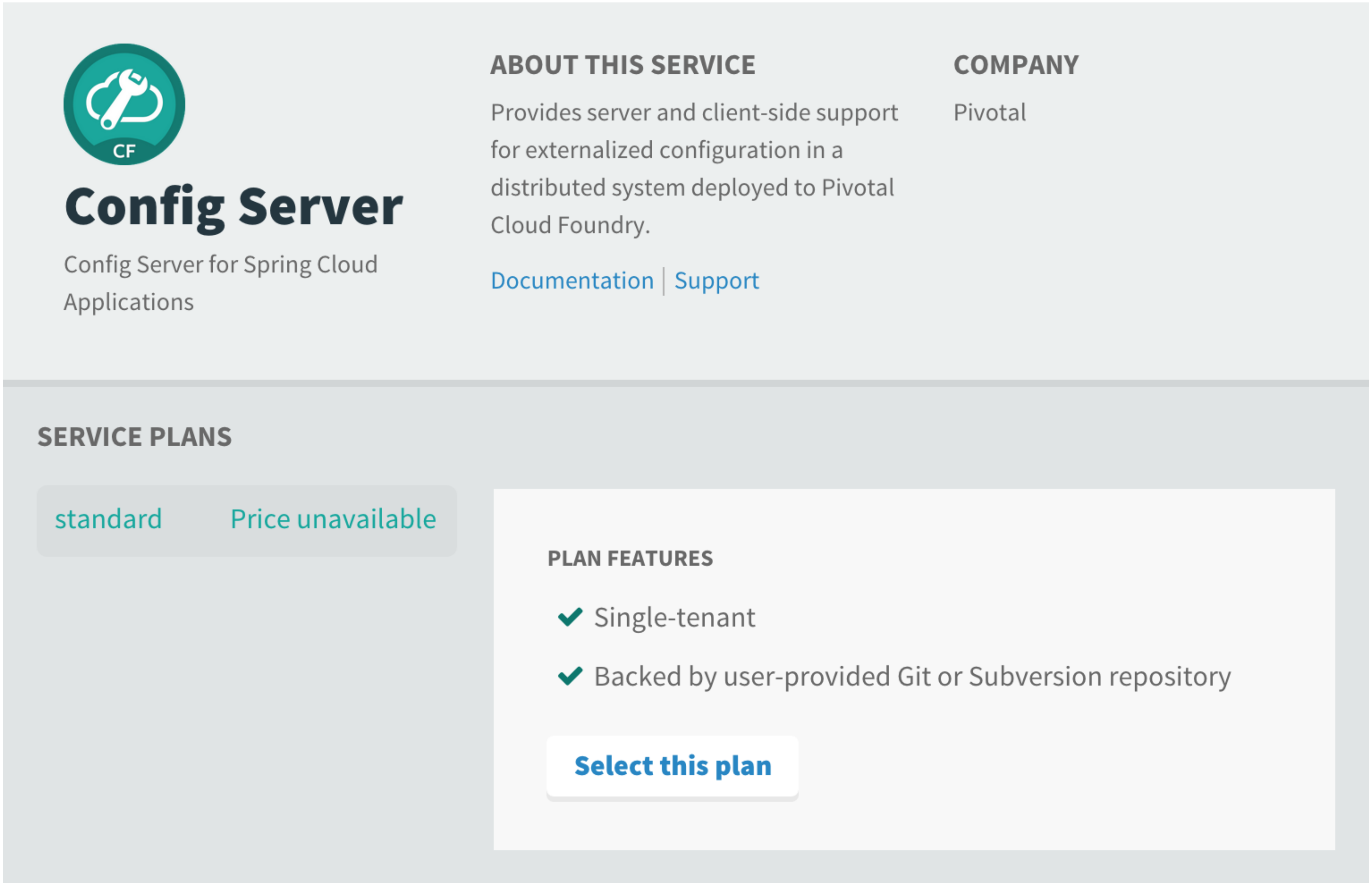
```
$ cf push greeting-config -p target/greeting-config-0.0.1-SNAPSHOT.jar -m 512M --random-rout
e --no-start
```

3) Create a Config Server Service Instance

Using Apps Manager do the following (for help review the docs (http://docs.pivotal.io/spring-cloud-services/config-server/creating-an-instance.html)):

a) Log into Apps Manager as a Space Developer. In the Marketplace, select Config Server for Pivotal Cloud Foundry. 

b) Select the desired plan for the new service.



Config Server

Config Server for Spring Cloud Applications

ABOUT THIS SERVICE

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

Documentation | Support

COMPANY

Pivotal

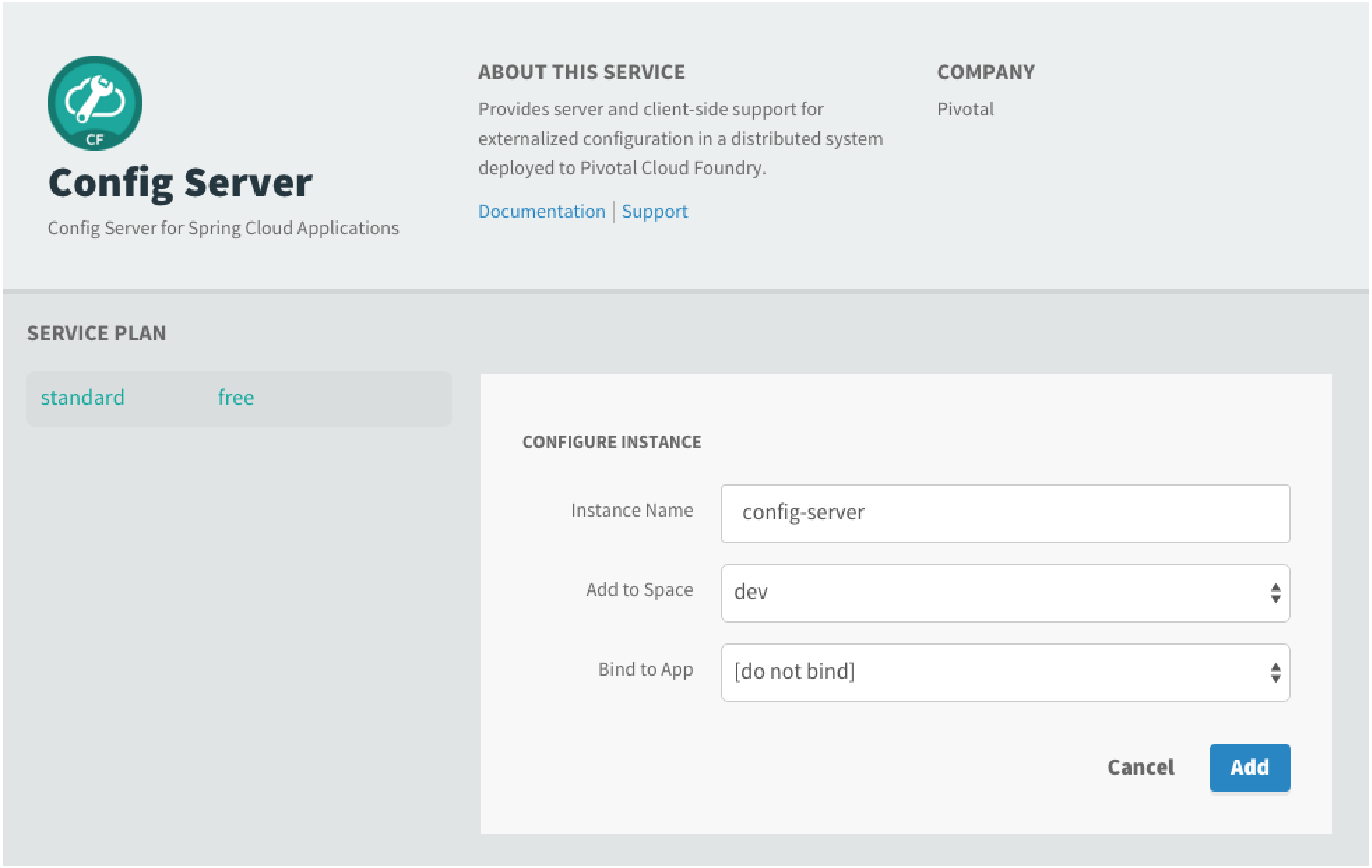
SERVICE PLANS

standard	Price unavailable

PLAN FEATURES

- ✓ Single-tenant
- ✓ Backed by user-provided Git or Subversion repository

c) Name the service `config-server`. Your space may be different. Click the ***Add*** button.



Config Server

Config Server for Spring Cloud Applications

ABOUT THIS SERVICE

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

Documentation | Support

COMPANY

Pivotal

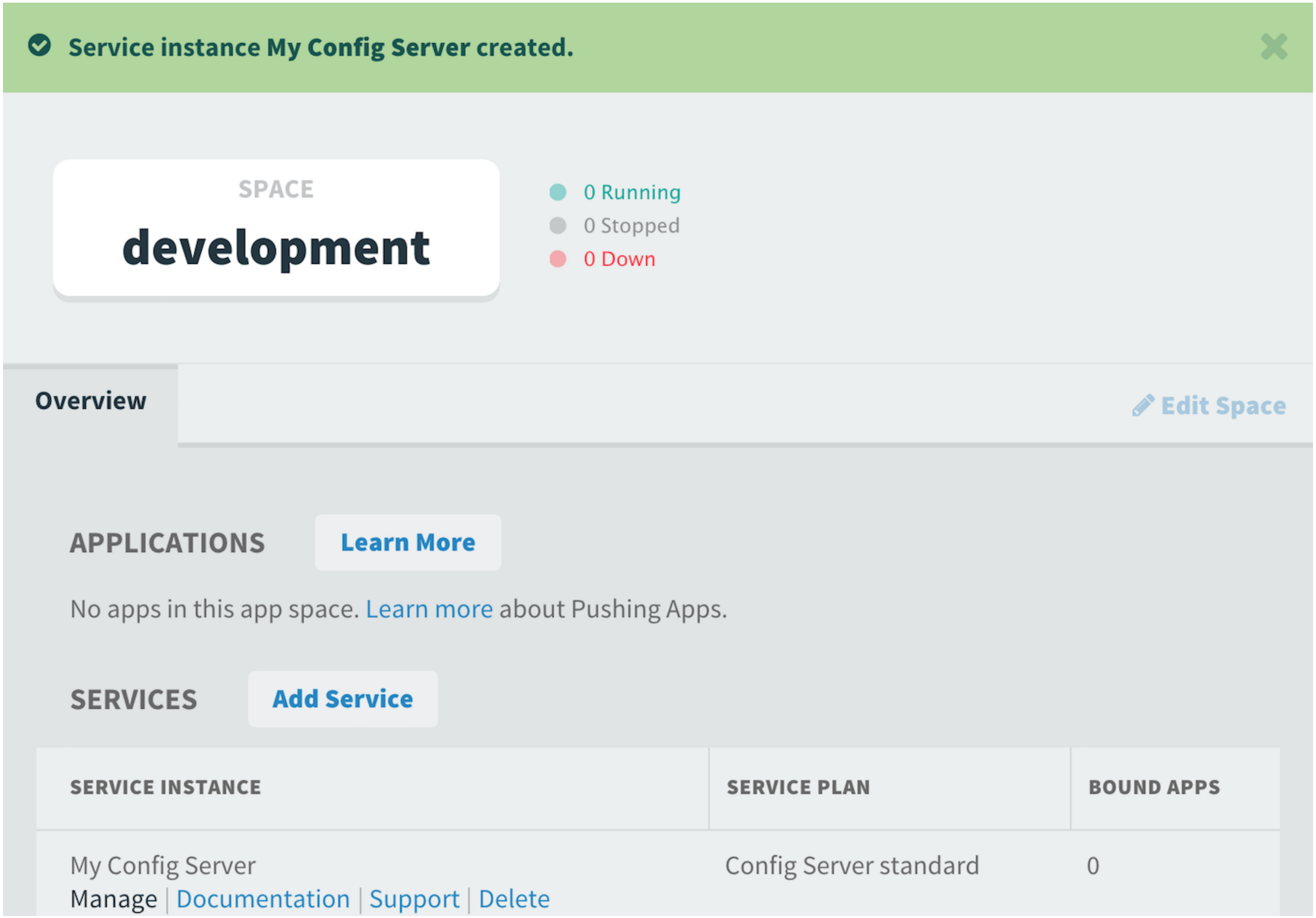
SERVICE PLAN

standard free

CONFIGURE INSTANCE

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

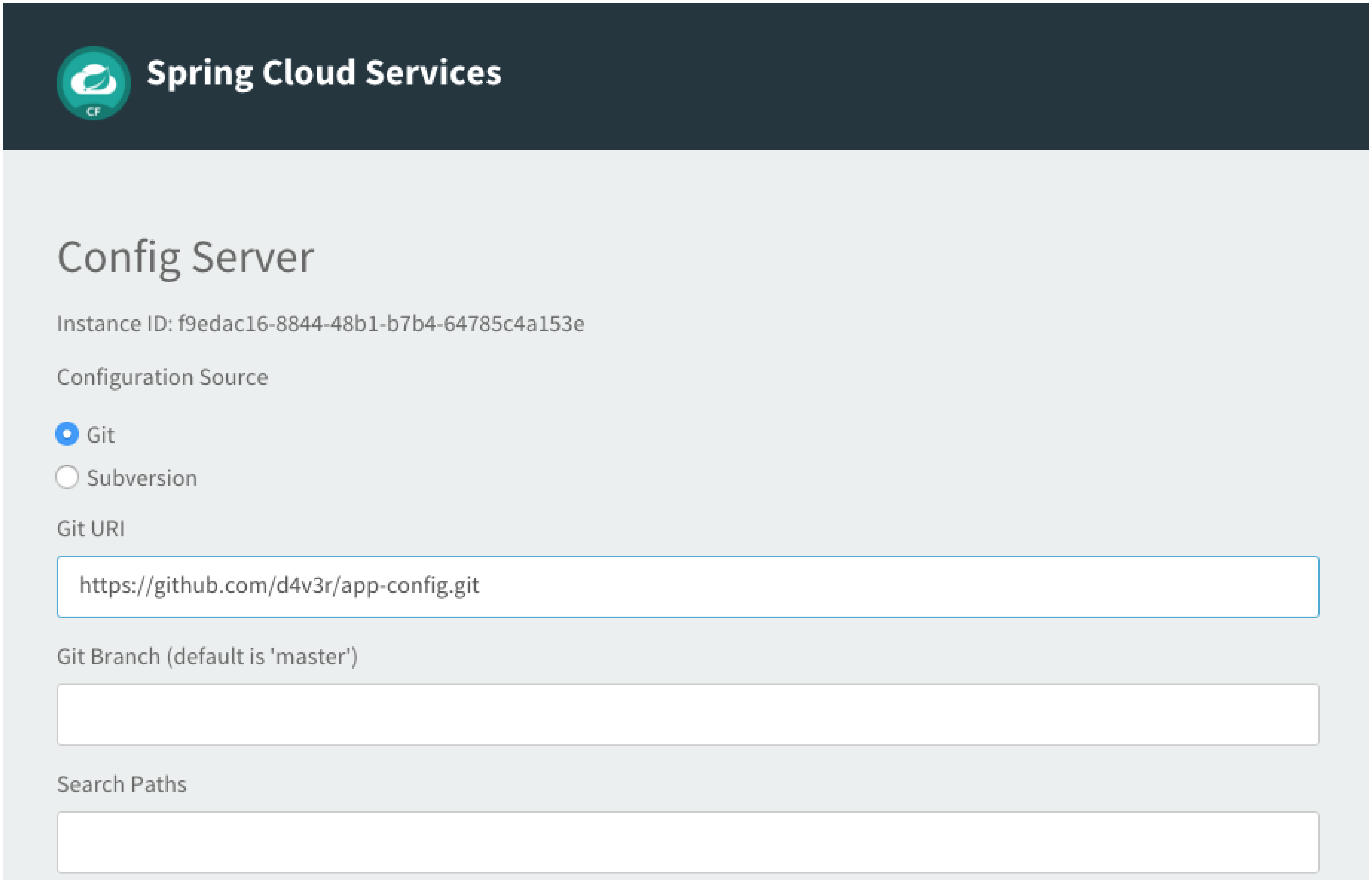
d) In the ***Services*** list, click the ***Manage*** link under the listing for the new service instance. The Config Server may take a few moments to initialize.


Service instance My Config Server created.

No apps in this app space. Learn more about Pushing Apps.

SERVICE INSTANCE	SERVICE PLAN	BOUND APPS
My Config Server Manage | Documentation | Support | Delete	Config Server standard	0

e) Select `Git` as the ***Configuration Source*** and enter your fork of the `app-config` repo under ***Git URI***. Do not use the literal below.



Spring Cloud Services

Config Server

Instance ID: f9edac16-8844-48b1-b7b4-64785c4a153e

Configuration Source

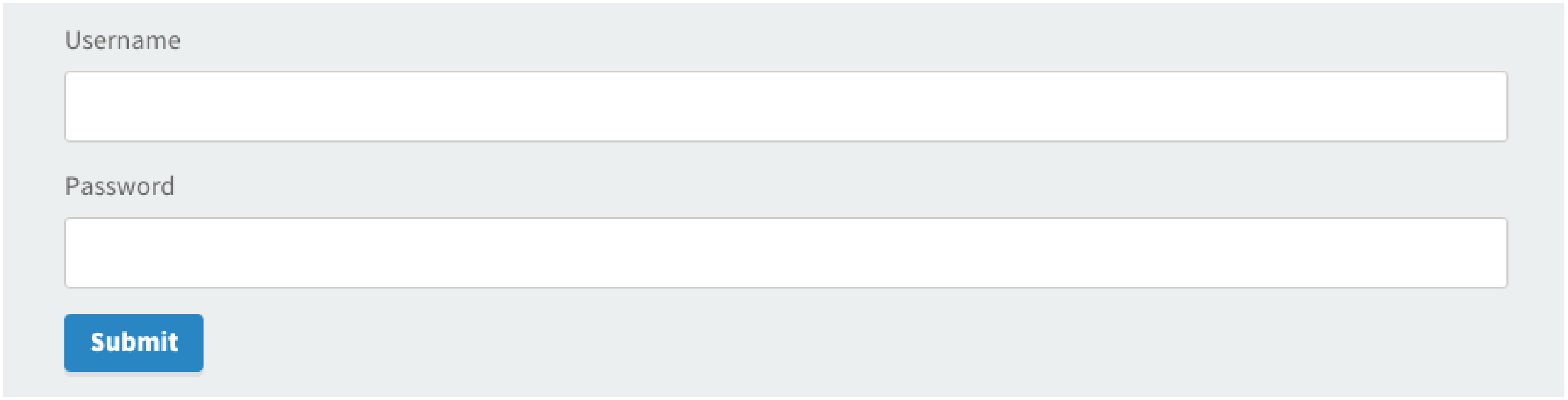
[✓ Git] Subversion

Git URI

https://github.com/d4v3r/app-config.git

Git Branch (default is 'master')

Search Paths



f) The Config Server instance (`config-server`) will take a few moments to initialize and then be ready for use.

4) Bind the `config-server` service to the `greeting-config` app. This will enable the `greeting-config` app to read configuration values from the `config-server` .

```
$ cf bind-service greeting-config config-server
```

You can safely ignore the *TIP: Use ‘cf restage’ to ensure your env variable changes take effect* message from the CLI. Our app doesn’t need to be restaged at this time.

5) If using self signed certificates, set the `CF_TARGET` environment variable to API endpoint of your Elastic Runtime instance. Make sure to use `https://` not `http://` . You can quickly retrieve the API endpoint by running the command `cf t` .

```
cf set-env greeting-config CF_TARGET <your api endpoint - make sure it starts with "http
s://">
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. Our app doesn't need to be restaged at this time.

NOTE:

All communication between Spring Cloud Services components are made through HTTPS. If you are on an environment that uses self-signed certs, the Java SSL trust store will not have those certificates. By adding the `CF_TARGET` environment variable a trusted domain is added to the Java trust store.

6) Start the `greeting-config` app.

```
$ cf start greeting-config
```

7) Browse to your `greeting-config` application. Are your configuration settings that were set when developing locally mirrored on PCF?

- Is the log level for `io.pivotal` package set to `DEBUG`? Yes, this can be confirmed with `cf logs` command while refreshing the `greeting-config /` endpoint (`http://<your-random-greeting-config-url/`).
- Is `greeting-config` app displaying the fortune? Yes, this can be confirmed by visiting the `greeting-`

config / endpoint.
- Is the `greeting-config` app serving quotes from `http://quote-service-qa.cfapps.io/quote`? No, this can be confirmed by visiting the `greeting-config /random-quote` endpoint. Why not? When developing locally we used an environment variable to set the active profile, we need to do the same on PCF.

```
$ cf set-env greeting-config SPRING_PROFILES_ACTIVE qa
$ cf restart greeting-config
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. Our app doesn't need to be restaged but just re-started.

Then confirm quotes are being served from `http://quote-service-qa.cfapps.io/quote`

Refreshing Application Configuration at Scale with Cloud Bus

Until now you have been notifying your application to pick up new configuration by POSTing to the `/refresh` endpoint.

When running several instances of your application, this poses several problems:

- Refreshing each individual instance is time consuming and too much overhead
- When running on Cloud Foundry you don't have control over which instances you hit when sending the POST request due to load balancing provided by the `router`

Cloud Bus addresses the issues listed above by providing a single endpoint to refresh all application instances via a pub/sub notification.

1) Create a RabbitMQ service instance, bind it to `greeting-config`

```
$ cf cs p-rabbitmq standard cloud-bus
$ cf bs greeting-config cloud-bus
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. Our app doesn't need to be restaged. We will push it again with new functionality in a moment.

2) Include the cloud bus dependency in the `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-config/pom.xml`. *You will need to paste this in your file.*

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

3) Repackage the `greeting-config` application:

```
$ mvn clean package
```

4) Deploy the application and scale the number of instances.

```
$ cf push greeting-config -p target/greeting-config-0.0.1-SNAPSHOT.jar -i 3
```

5) Observe the logs that are generated by refreshing the `greeting-config /` endpoint several times in your browser and tailing the logs. Allow this process to run through the next few steps.

```
[mac, linux]
$ cf logs greeting-config | grep GreetingController

[windows]
$ cf logs greeting-config
# then search output for "GreetingController"
```

All app instances are creating debug statements. Notice the `[App/X]`. It denotes which app instance is logging.

```
2015-09-28T20:53:06.07-0500 [App/2]      OUT 2015-09-29 01:53:06.071 DEBUG 34 --- [io-64495-
exec-6] io.pivotal.greeting.GreetingController   : Adding fortune
2015-09-28T20:53:06.16-0500 [App/1]      OUT 2015-09-29 01:53:06.160 DEBUG 33 --- [io-63186-
exec-5] io.pivotal.greeting.GreetingController   : Adding greeting
2015-09-28T20:53:06.16-0500 [App/1]      OUT 2015-09-29 01:53:06.160 DEBUG 33 --- [io-63186-
exec-5] io.pivotal.greeting.GreetingController   : Adding fortune
2015-09-28T20:53:06.24-0500 [App/1]      OUT 2015-09-29 01:53:06.246 DEBUG 33 --- [io-63186-
exec-9] io.pivotal.greeting.GreetingController   : Adding greeting
2015-09-28T20:53:06.24-0500 [App/1]      OUT 2015-09-29 01:53:06.247 DEBUG 33 --- [io-63186-
exec-9] io.pivotal.greeting.GreetingController   : Adding fortune
2015-09-28T20:53:06.41-0500 [App/0]      OUT 2015-09-29 01:53:06.410 DEBUG 33 --- [io-63566-
exec-3] io.pivotal.greeting.GreetingController   : Adding greeting
```

7) Turn logging down. In your fork of the `app-config` repo edit the `greeting-config.yml`. Set the log level to `INFO`. Make sure to push back to Github.

```
logging:
  level:
    io:
      pivotal: INFO
```

8) Notify applications to pickup the change. Open a new terminal window. Send a POST to the `greeting-config` `/bus/refresh` endpoint. Use your `greeting-config` URL not the literal below.

```
$ curl -X POST http://greeting-config-hypodermal-subcortex.cfapps.io/bus/refresh
```

9) Refresh the `greeting-config /` endpoint several times in your browser. No more logs!

10) Stop tailing logs from the `greeting-config` application.

Back to TOP