Spring Cloud Config

Table of Contents

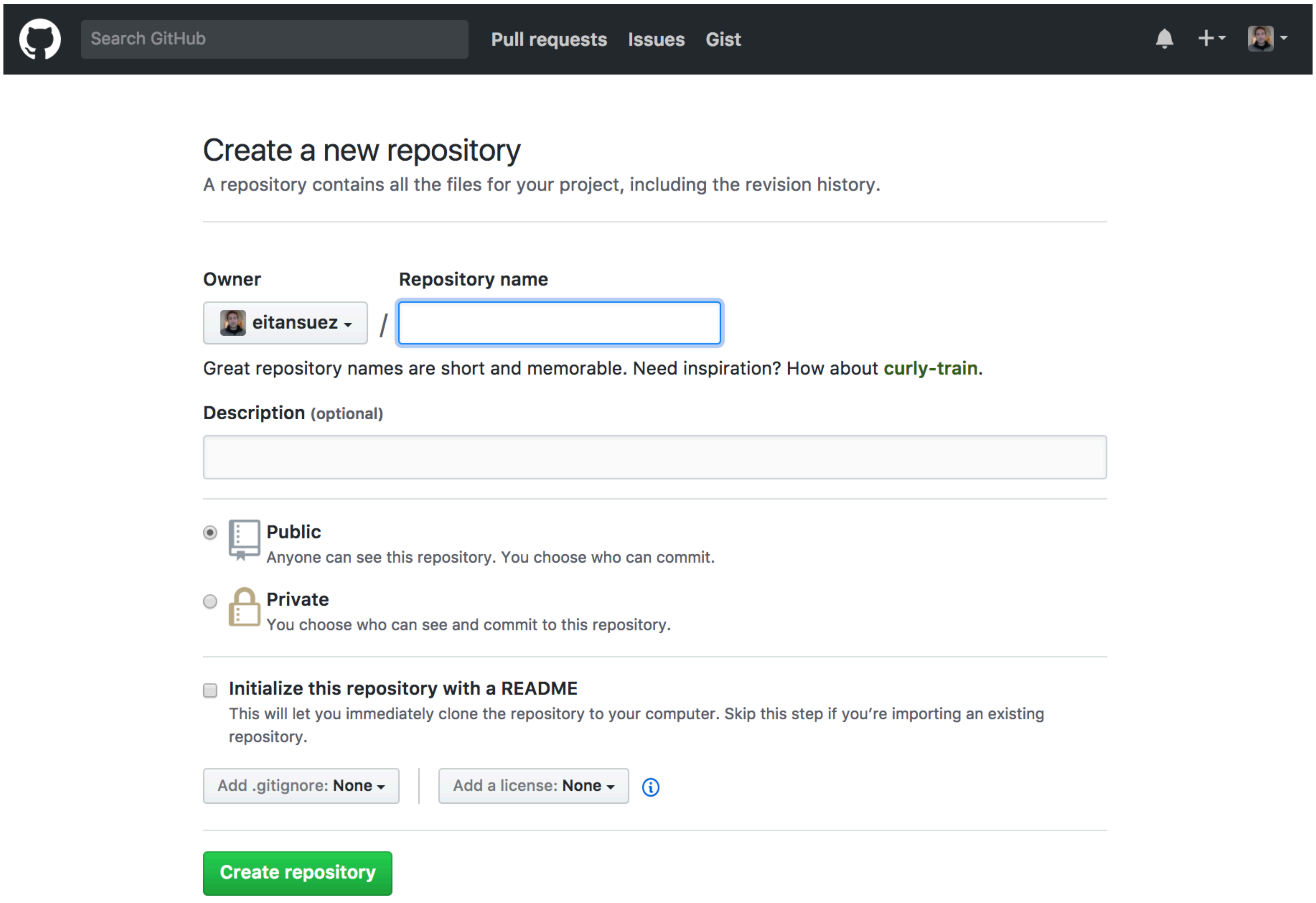
What You Will Learn

- 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

1. Set up the `app-config` Repo

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

1. Create a public repository on github, and name it *app-config*
2. After creating your public repository, follow the instructions that github supplies to create a new (local) repository on the command line and to configure the github repository you just created as its "remote"



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: eitansuez / Repository name:

Great repository names are short and memorable. Need inspiration? How about curly-train.

Description (optional)

(●) Public
Anyone can see this repository. You choose who can commit.

( ) Private
You choose who can see and commit to this repository.

[ ] Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

This repository will serve as the source of configuration data for our Spring applications.

2. Set up `config-server`

1. Review the project `config-server` in the spring cloud services labs you recently cloned.
2. Review the project's `pom.xml` file. Notice the `spring-cloud-config-server` dependency:

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

This dependency turns a spring boot application into a spring configuration service.

3. Look at the class `ConfigServerApplication.java`

```
package io.pivotal;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@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`.

4. Configure the `config-server` with the GitHub repository you just created. This will be the source of the configuration data. Edit the `application.yml` file:

```
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/eitansuez/app-config.git
```

Make sure to use your `app-config` repository url above.

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

```
$ cd config-server
$ mvn spring-boot:run
```

Your `config-server` will be running locally on port 8888 (once you see a *Started ConfigServerApplication..* message). You will not be returned to a command prompt and must leave this window open.

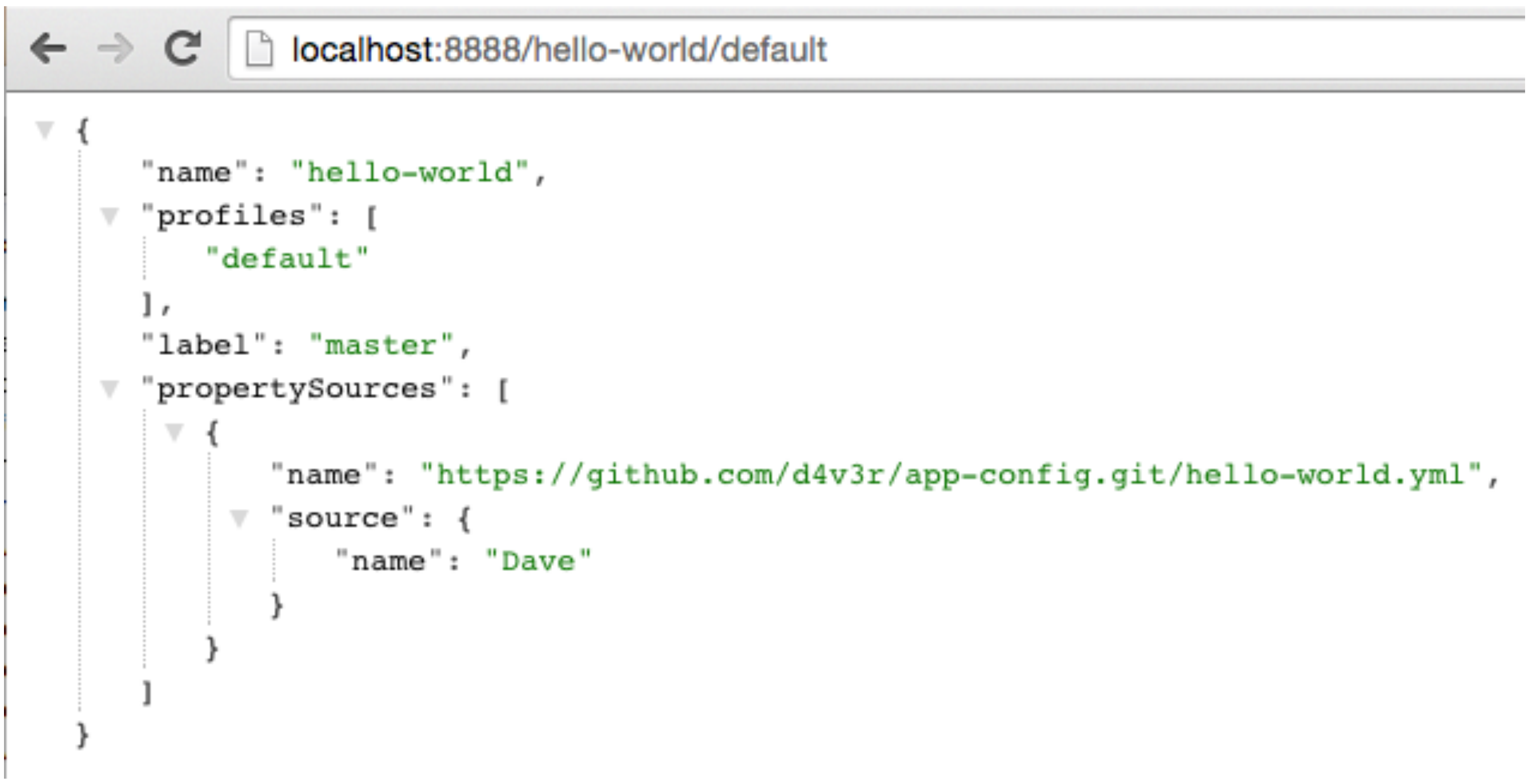
6. Let's add some configuration. Edit your `app-config` repo. Create a file called `hello-world.yml`. Add the content below to the file.

```
name: John Doe
```

7. *Push the changes* back to GitHub.

8. Confirm that the application named `hello-world` is now configured with this name property by visiting this config server url: http://localhost:8888/hello-world/default

> 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.


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).

3. Set up `greeting-config`

1. Review the `greeting-config` project, and specifically its `pom.xml` file.

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

   By adding `spring-cloud-services-starter-config-client` as a dependency, this application will consume configuration from the `config-server`. `greeting-config` is a config client.

2. Notice that the `bootstrap.yml` file defines the spring application's name:

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

   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 greeting-config
   $ mvn spring-boot:run
   ```

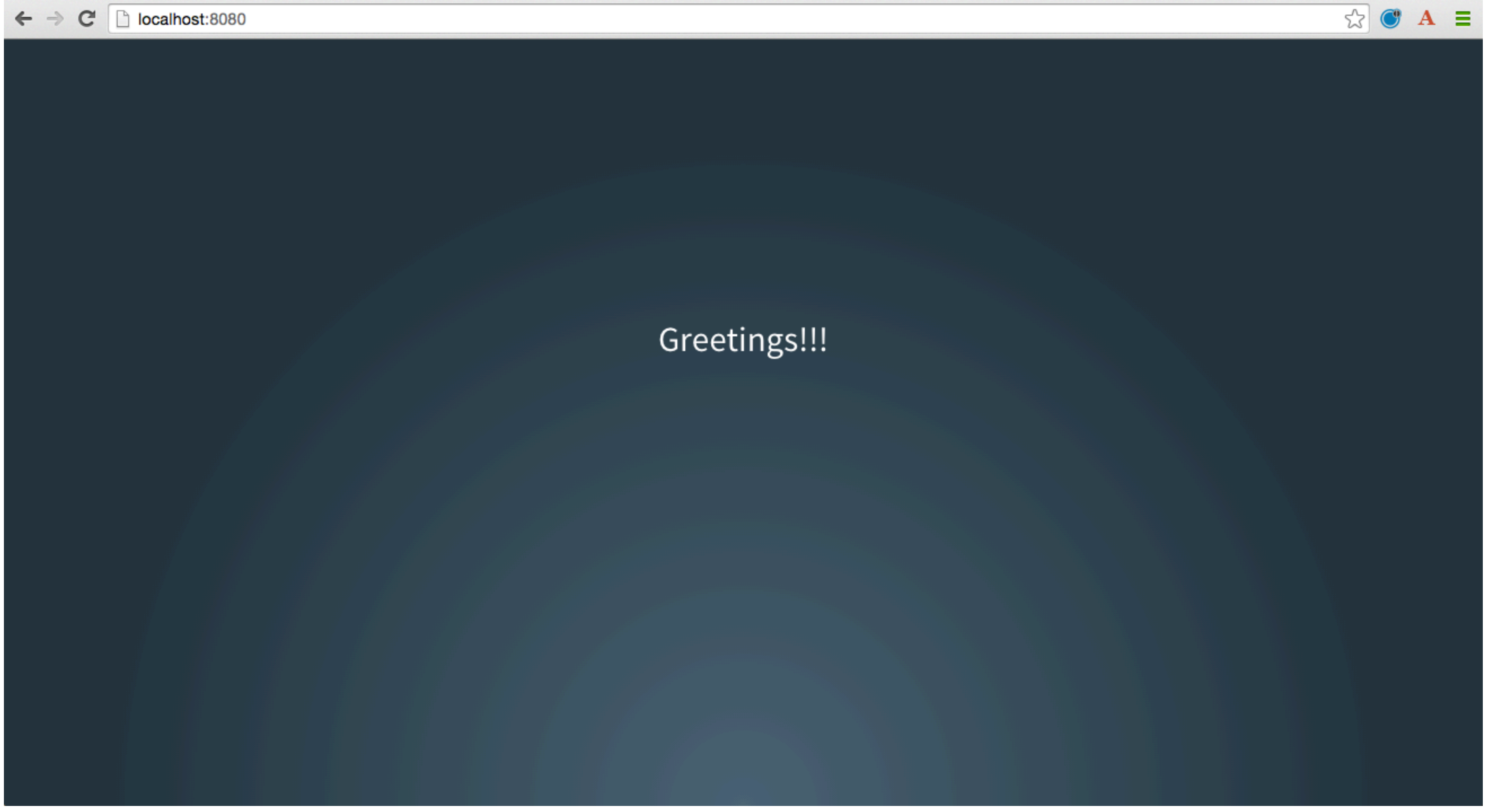
4. Confirm the `greeting-config` app is up. Browse to http://localhost:8080. You should be prompted to authenticate. Why? `spring-cloud-services-starter-config-client` has a dependency on Spring Security (http://projects.spring.io/spring-security/). Unless the given application has other security configuration,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

5. Spring Security automatically generates basic authentication credentials if none have been set explicitly, as in this case. The username is simply `user`, and the password is written by Spring Security to the application's log. Search your application's console output for a line that looks like this:

   ```
   Using default security password: xxxxx-xxxxx-xxxxx-xxxxx-xxxxx
   ```

   The username and password can be explicitly set via the configuration parameters `security.user.name` and `security.user.password`.

6. After logging in, you should see the message "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.PropertySourceBootstrapConfiguration :
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

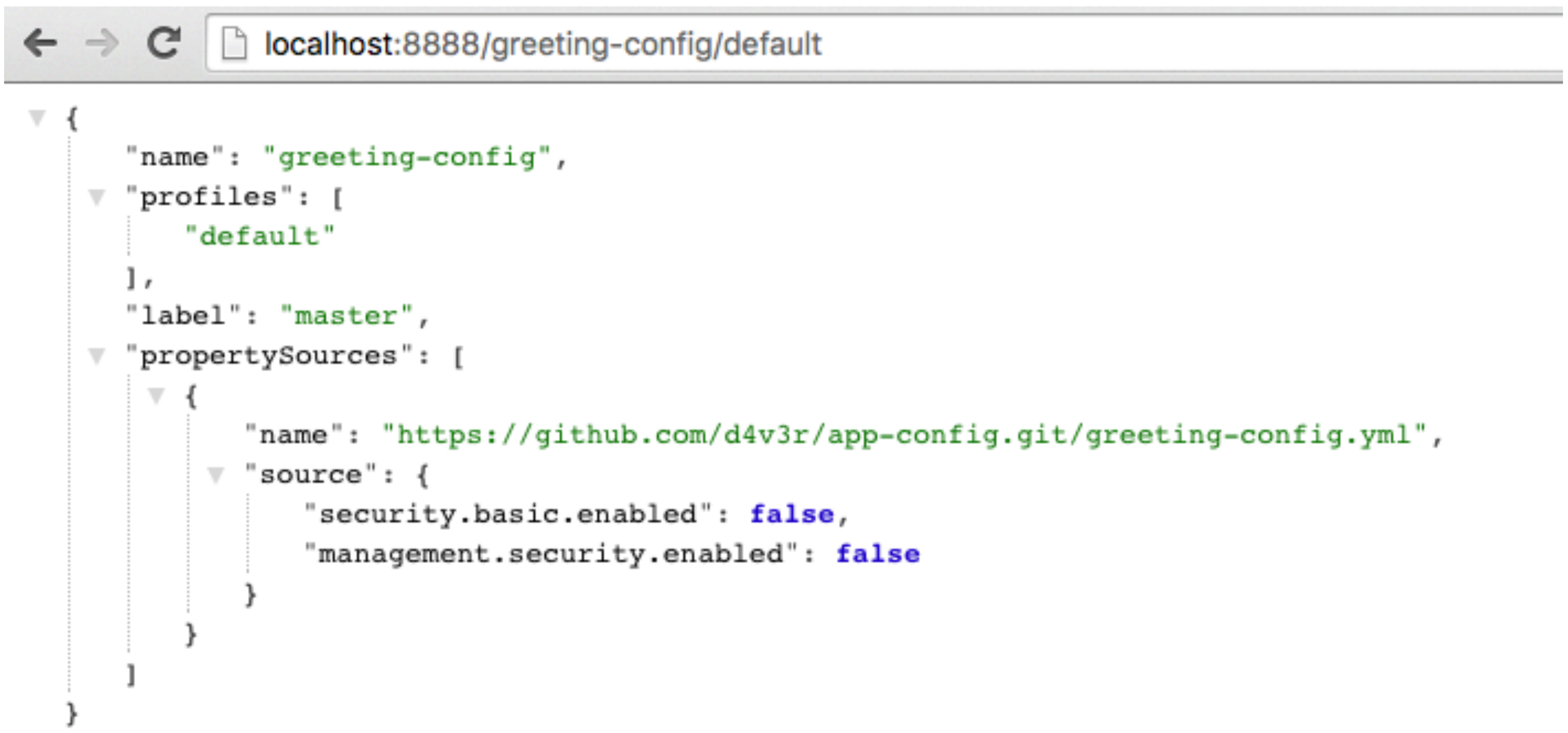
4. Unsecure the Endpoints

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 `app-config` repository. 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 off securing our application endpoints

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

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



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
            }
        }
    ]
}
```

3. Start the `greeting-config` application:

```
$ mvn 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.PropertySourceBootstrapConfiguration : Located property source: CompositePropertySource
[name='configService', propertySources=[MapPropertySource [name='https://github.com/d4v3r/app-
config.git/greeting-config.yml']]]
```

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

5. Changing Logging Levels

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

1. View the `getGreeting()` method of the `GreetingController` class:

```
@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());
  }

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

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).

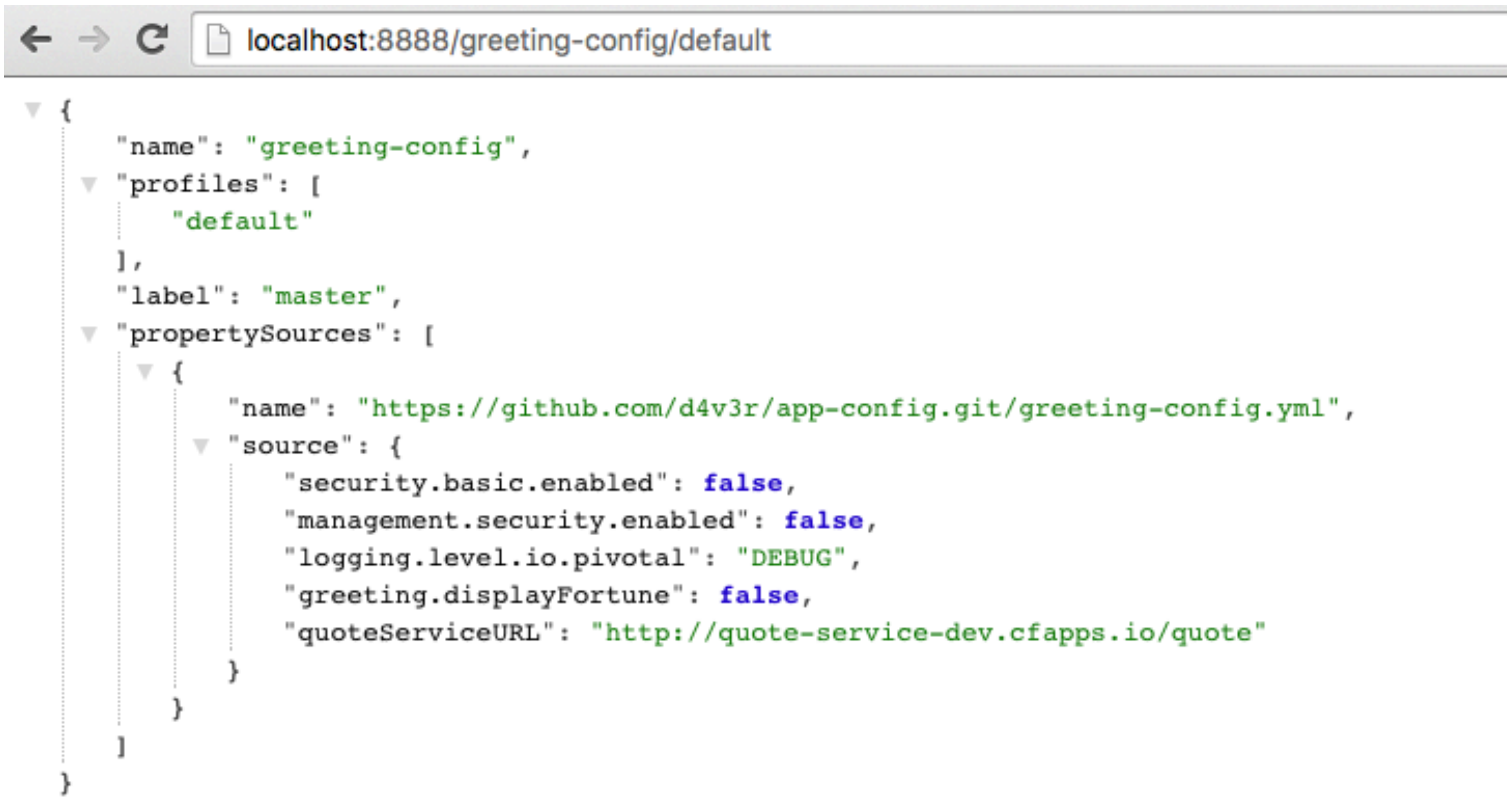
2. In your `app-config` repository, add the content below to the `greeting-config.yml` file and push the changes back to GitHub.

```yaml
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/ 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


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: `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/ 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.

6. 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 `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.

```java
package io.pivotal.greeting;

import org.springframework.boot.context.properties.ConfigurationProperties;

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

  private boolean displayFortune;

  public boolean isDisplayFortune() {
    return displayFortune;
  }

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

2. Review `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.

```
package io.pivotal.greeting;

import io.pivotal.fortune.FortuneService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

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

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

  private final GreetingProperties greetingProperties;
  private final FortuneService fortuneService;

  public GreetingController(GreetingProperties greetingProperties, FortuneService
fortuneService) {
    this.greetingProperties = greetingProperties;
    this.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());
    }

    return "greeting"; // resolves to the greeting.ftl template
  }

}
```

3. Edit your `app-config` repository. 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/ url and see the fortune included.

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

7. Reinitializing Beans with `@RefreshScope`

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

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

1. Review `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).

```java
package io.pivotal.quote;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
@RefreshScope
public class QuoteService {
  private final Logger logger = LoggerFactory.getLogger(QuoteService.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 `greeting-config/src/main/java/io/pivotal/quote/QuoteController.java`. `QuoteController` calls the `QuoteService` for quotes.

```
package io.pivotal.quote;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class QuoteController {

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

  private final QuoteService quoteService;

  public QuoteController(QuoteService quoteService) {
    this.quoteService = quoteService;
  }

  @RequestMapping("/random-quote")
  String getView(Model model) {
    logger.debug("returning random quote");
    model.addAttribute("quote", quoteService.getQuote());
    model.addAttribute("uri", quoteService.getQuoteServiceURI());
    return "quote";
  }
}
```

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

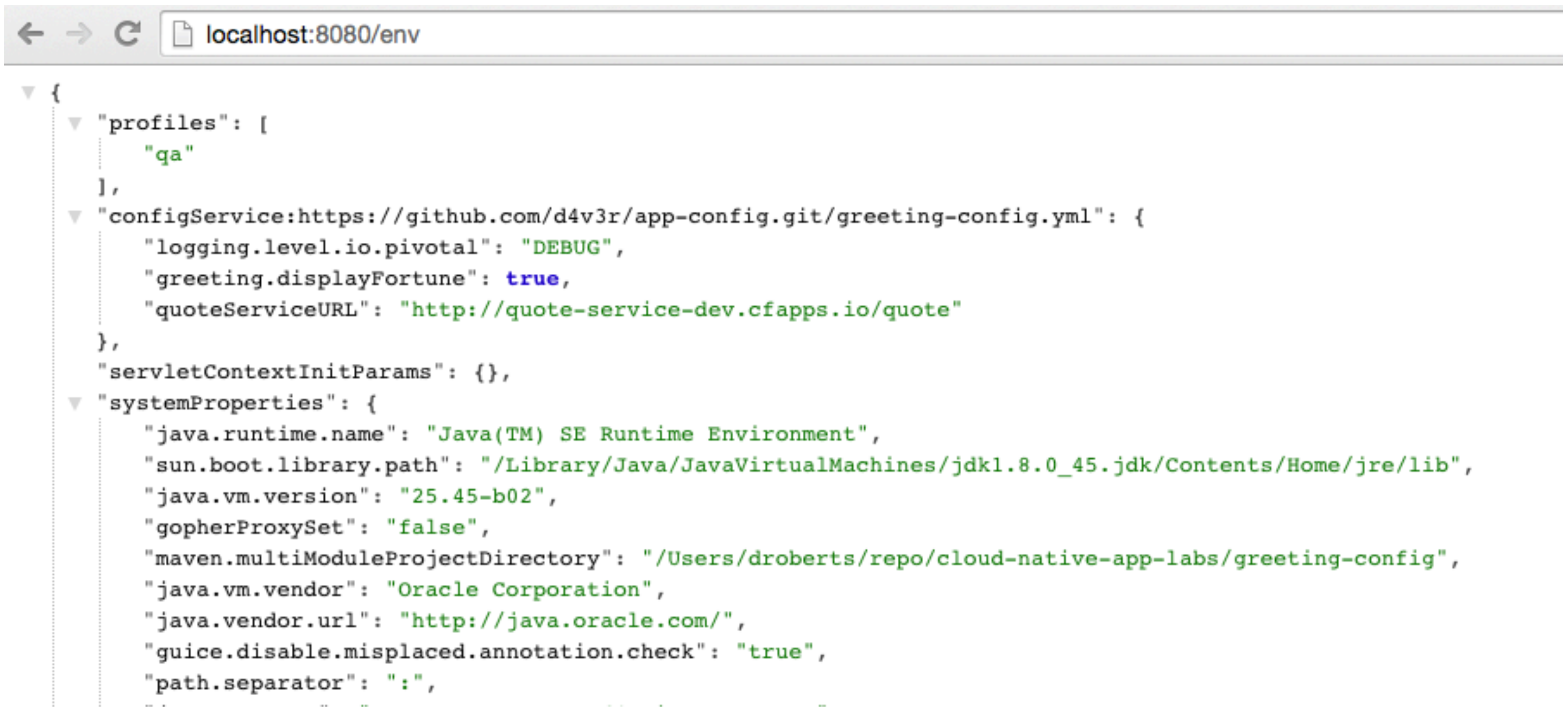
8. Override Configuration Values By Profile

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 | windows

```
$ SPRING_PROFILES_ACTIVE=qa mvn spring-boot:run
```

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



```
{
  "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": ":",
```

4. In your `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.

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

6. Refresh the application configuration values

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

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

8. 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`).

9. 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-route --no-start
   ```

3. Create a Config Server Service Instance

   Spring Cloud Services provides the `p-config-server` managed service for provisioning config servers on demand. Pass it as an argument to the `create-service` cf command.

   First, familiarize yourself with the command:

   ```
   cf help create-service
   ```

   As you probably suspect, this config server must be configured with the uri of its backing git repository. This configuration is provided with the `-c` command flag. The information is json-encoded, like so:

   ```
   { "git": { "uri": "https://github.com/{{github_username}}/app-config.git" } }
   ```

   Here is the full command:

   mac, linux | windows

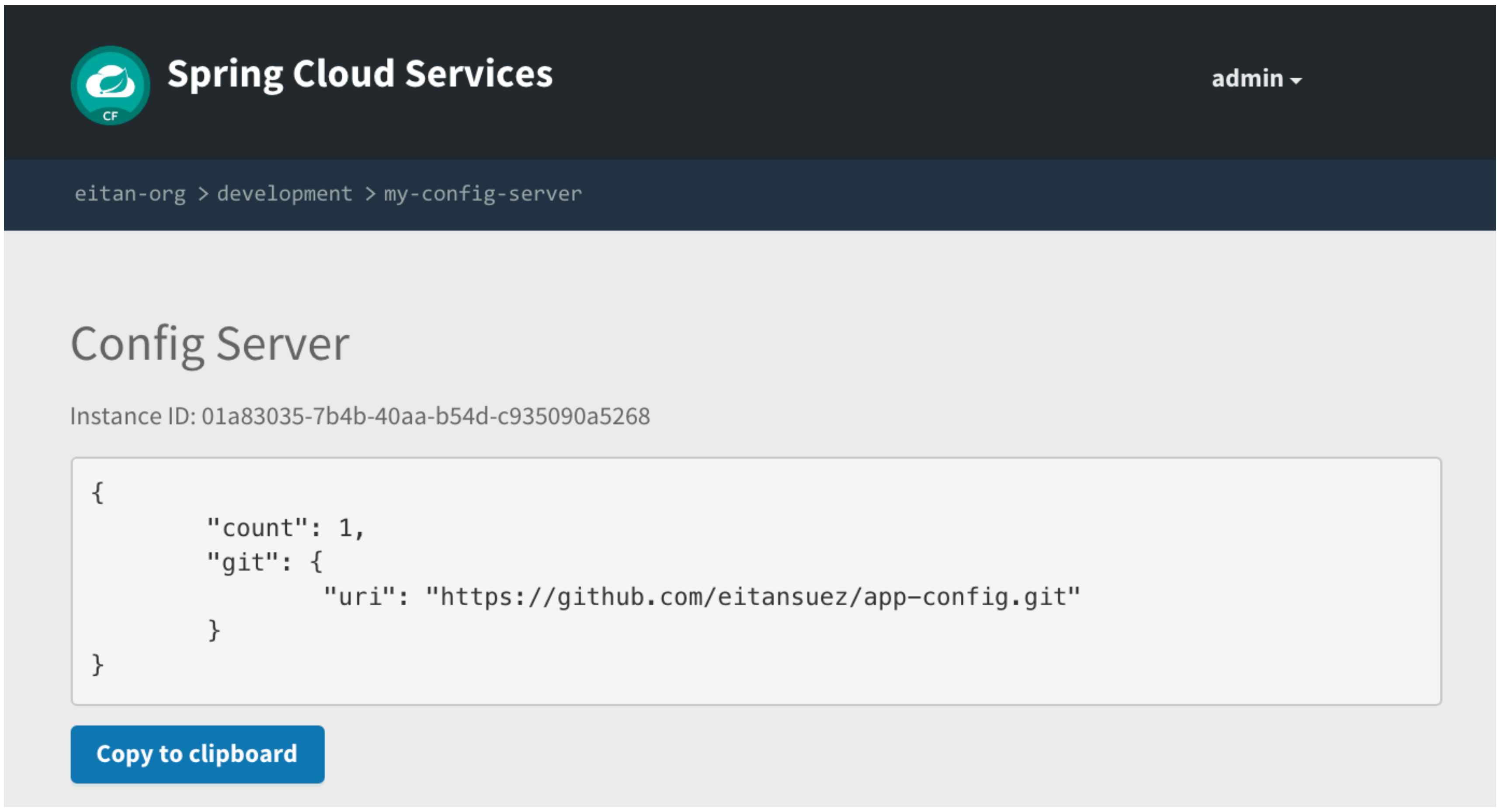
   ```
   $ cf create-service p-config-server standard config-server -c '{ "git": { "uri": "https://github.com/{{github_username}}/app-config.git" } }'
   ```

For more information on creating a config server service instance, consult the documentation here (http://docs.pivotal.io/spring-cloud-services/1-3/common/config-server/creating-an-instance.html).

Feel free to name your service anything you like, it doesn't have to be named `config-server`. The Config Server instance will take a few moments to initialize and then be ready for use.

Invoke either the `cf services` command or `cf service config-server` to view the status of the service you just created.

The Apps Manager also provides a means to access a dashboard for your Config Server. In a browser, navigate to the apps manager, and to your space. You should see your config server service displayed in there (it may be in a separate tab named `services`). Click on the service, and in the subsequent view, select the `Manage` link.



Config Server

Instance ID: 01a83035-7b4b-40aa-b54d-c935090a5268

```
{
        "count": 1,
        "git": {
                "uri": "https://github.com/eitansuez/app-config.git"
        }
}
```

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 because it isn't currently running.

5. Our PCF instance is using self-signed SSL certificates. Set the `TRUST_CERTS` environment variable to API endpoint of your Elastic Runtime instance.

   > You can quickly retrieve the API endpoint by running the command `cf api`.

   ```
   $ cf set-env greeting-config TRUST_CERTS api.sys.gcp.esuez.org
   ```

   > Make sure to specify only the hostname part of your api endpoint (i.e. without the `https://` scheme/prefix)

   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.

   > 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 `TRUST_CERTS` environment variable a trusted domain is added to the Java trust store. For more information see the this portion (https://docs.pivotal.io/spring-cloud-services/config-server/writing-client-applications.html#self-signed-ssl-certificate) of the SCS documentation.

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` root endpoint.

- 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

10. 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:

   ```
   $ cf create-service p-rabbitmq standard cloud-bus
   ```

2. Bind it to `greeting-config`:

```
$ cf bind-service 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.

3. Include the cloud bus dependency in the `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>
```

4. Repackage the `greeting-config` application:

```
$ mvn clean package
```

5. Deploy the application and scale the number of instances.

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

> Invoke the command `cf help push` and study the command line arguments that can be passed to the push command. What does the -i flag control?

6. Observe your application's logs, specifically what GreetingController is emitting:

```
$ cf logs greeting-config | grep GreetingController
```

7. Generate log messages by refreshing the `greeting-config` root endpoint several times in your browser.

   All app instances are creating debug statements. Notice the `[App/X]` portion of each log statement, which denotes which application 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
```

8. Turn logging down. In your `app-config` repository, edit the `greeting-config.yml`. Set the log level to `INFO`.

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

9. Don't forget to push your commit back to Github.

10. 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 | httpie

```
$ curl -k -X POST https://{{greeting_config_hostname}}/bus/refresh
```

11. Refresh the `greeting-config` root endpoint several times in your browser. No more logs!

12. Stop tailing logs from the `greeting-config` application.

Last updated 2017-06-23 16:30:28 CEST