Spring Cloud Config

Work

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system.

Spring Cloud Config Quick Start Page

1. Preparation

Install Spring boot by following Spring boot getting started

Linux for example:

```
Install Groovy Environment Manager
$ gvm install springboot
$ spring --version
Spring Boot v1.2.5.RELEASE
```

A simple sample for Spring boot as below:

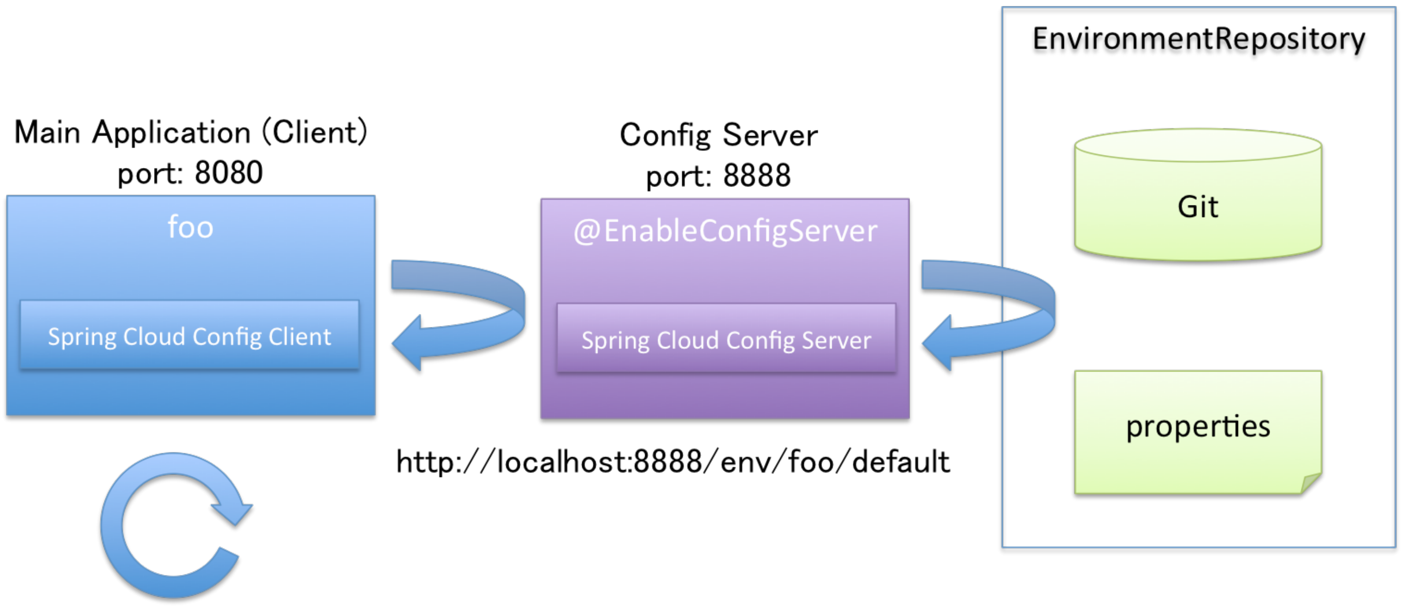
```java
package hello;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@Controller
@EnableAutoConfiguration
public class SampleController {

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World!";
    }

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

Git Clone the Sample Code of Srping Cloud Config

https://github.com/spring-cloud/spring-cloud-config/tree/1.0.2.RELEASE

```
.
├── docs
├── Guardfile
├── pom.xml
├── README.adoc
├── sample.groovy
├── spring-cloud-config-client
├── spring-cloud-config-sample
└── spring-cloud-config-server
```

2. The basic architecture of Spring Cloud Config


EnvironmentRepository
Main Application (Client)
port: 8080
foo
Spring Cloud Config Client
Config Server
port: 8888
@EnableConfigServer
Spring Cloud Config Server
Git
properties
http://localhost:8888/env/foo/default

Apply

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

or

curl -X POST http://localhost:8080/restart

Adhoc update

curl -X POST http://localhost:8080/env -d bar=100

Setup Tips

1. Start Config Server first
2. Then start client app.
3. After Config Server is down, Cient still works.
4. Restarting Config Server will re-clone git properties
5. use POST method instead of GET for curl command above

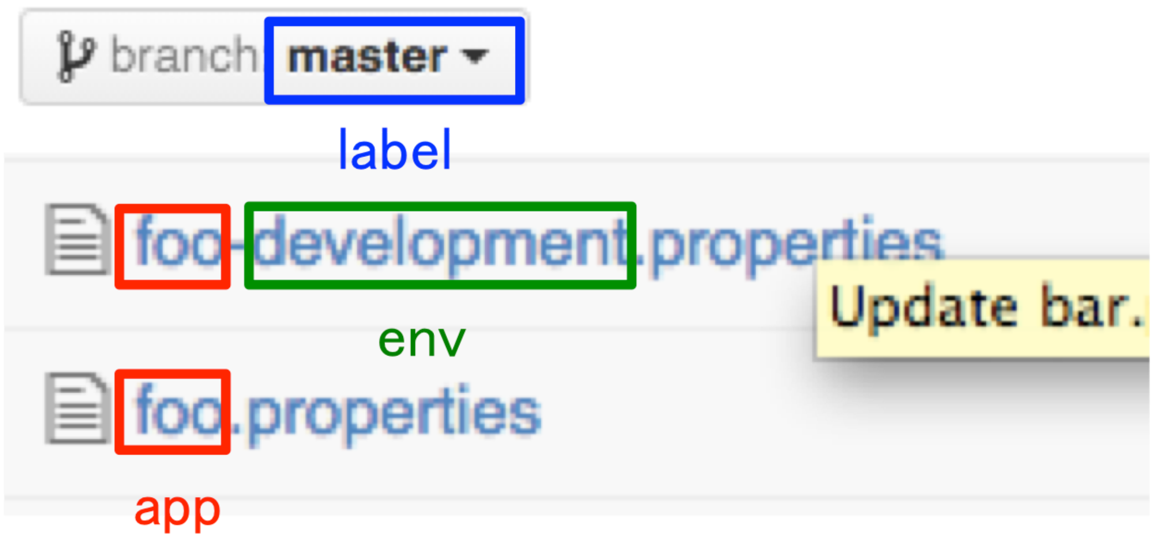
Setup Config Server

1. Start and visit config server

```
$ cd spring-cloud-config-server
$ mvn spring-boot:run
$ curl localhost:8888/foo/default
$ curl localhost:8888/foo/development
{"name":"development","label":"master","propertySources":[
  {"name":"https://github.com/scratches/config-repo/foo-development.properties","source":{"bar":"spam"}}, # The priority of foo-development.properties is higher than foo.properties
  {"name":"https://github.com/scratches/config-repo/foo.properties","source":{"foo":"bar"}}
]}
```

localhost:8888/foo/development is following this convention:

```
/{application}/{profile}[/{label}]
application: foo
profile: development (environment like develop/qa/release/production)
label: "master" (master branch by default)
```



Explain more below.

2. Configurations in config server

```
/spring-cloud-config-server$ tree
.
├── pom.xml
├── src
│    ├── main
│    │    ├── java
│    │    └── resources
│    │         ├── configserver.yml
```

The content of the configserver.yml

```
info:
  component: Config Server
spring:
  application:
    name: configserver
  jmx:
    default_domain: cloud.config.server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            - patterns: multi-repo-demo-*
              uri: https://github.com/spring-cloud-samples/config-repo

server:
  port: 8888
management:
  context_path: /admin
```

The content of the git repository https://github.com/spring-cloud-samples/config-repo:

```
.
├── application.yml
├── bar.properties
├── configserver.yml
├── eureka.yml
├── foo-development.properties
├── foo.properties
├── processor.yml
```

```
├──── samplebackendservice-development.properties
├──── samplebackendservice.properties
├──── samplefrontendservice.properties
├──── stores.yml
└──── zuul.properties
```

Will be cloned to /tmp/config-repo-{id} in Linux

localhost:8888/foo/development refer to foo-development.properties
localhost:8888/foo/default refer to foo.properties
Updating git repostiory will reflect to localhost:8888 like /tmp/config-repo-{id}

3. Client Side Usage

Simple structure for client side.

```
├──── pom.xml
├──── src
|      ├──── main
|      |      ├──── java
|      |      |      └──── sample
|      |      |             └──── Application.java
|      |      └──── resources
|      |             ├──── application.yml
|      |             └──── bootstrap.yml
```

```
$ cd spring-cloud-config-sample
$ mvn spring-boot:run
```

spring-cloud-config-sample/pom.xml

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.3.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-parent</artifactId>
            <version>1.0.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

<!-- repositories also needed for snapshots and milestones -->
```

Main Client class:

spring-cloud-config-sample/src/main/java/sample/Application.java

```java
@Configuration
@EnableAutoConfiguration
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

spring-cloud-config-sample/src/main/resources/bootstrap.yml

```yaml
spring:
  application:
    name: bar
  cloud:
    config:
      env: default # optional
      label: master # optional
      uri: http://localhost:${config.port:8888}
```

where it specifies application name bar and the uri of spring cloud config server.

```
$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/scratches/config-repo/bar.properties":{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

Usage:

1. Get/Refresh properties (Fetch value on request API call)

```
$ curl localhost:8080/env/foo
bar
$ vi /tmp/config-repo-{id}/bar.properties
.. change value of "bars"
$ curl -X POST localhost:8080/refresh
["foo"]
$ curl localhost:8080/env/foo
bars
```

2. Usage of ClientAppClass

```java
package demo;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@EnableAutoConfiguration
@ComponentScan
@RestController
```

```
public class ClientApp {
    @Value("${bar:World!}")
    String bar;

    @RequestMapping("/")
    String hello() {
        return "Hello " + bar + "!";
    }

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

You can also see a single property.

```
$ curl http://localhost:8080/env/bar
123456
```

When you access to the controller,

```
$ curl http://localhost:8080
Hello 123456!
```

you can find the property on Config Server is injected.

See more usage samples here: http://qiita.com/making@github/items/704d8e254e03c5cce546