Externalized Configuration

Context

An application's configuration varies independently from the underlying source code throughout its lifecycle. The desire is for the code to remain immutable as it flows through the continuous delivery pipeline. Any variation in execution behavior between deployment environments should be accomplished without modification to the source code of an application service.

Problem

Traditional techniques for managing configuration will very often tightly couple the orthogonal concepts of independently varying configuration from the execution of an application service. A solution is required that will enable an application service to execute in multiple environments without modification of the deployment artifact.

Forces

Requirements the solution must fulfil:

- Different deployment environments will have different configuration settings. Such as:
  - resource handles to database (e.g. a JDBC URL)
  - credentials to external services (e.g. AWS services)
  - per-deploy values such as a canonical hostname of a resource
  - feature sets that are toggled on or off based on deployment environment
- An application service must run in different deployment environments without modification and/or recompilation:
  - Development, test, QA, staging, production, etc.
- Different environments have different instances of external – 3rd party – services:
  - production database vs. QA database

A common approach is to separate configuration from source code by placing configuration information in an XML or a Java property file. As part of the build process, the configuration information would be bundled within the deployment artifact. In some cases, it's possible to edit these files to change an application's behavior after it's been deployed. However, changes to the configuration require redeployment of the application, often resulting in unacceptable downtime and other administrative overhead. This introduces unnecessary overhead in configuration management of an artifact since it differs between deployment environments.

Another approach would be to modify the build process to take into consideration the deployment environment. The build process will create a deployment artifact with the corresponding configuration information for the targeted environment. However, this approach still suffers from configuration management issues of having a unique artifact for each deployment environment. If we are rebuilding the artifact for each environment there is always a chance the artifact will differ between environments. This leads to reduced confidence that an artifact tested in one environment is the same artifact deployed in another environment.

The deployment pipeline should only build each deployment artifact once, and deploy the same artifact to multiple environments. A one-time-only build eliminates the risk of untracked differences due to

variations in deployment environments, third-party libraries or different compilation contexts; that could potentially result in an unstable or an unpredictable release. As we move an artifact through environments the only thing that should change is its configuration.

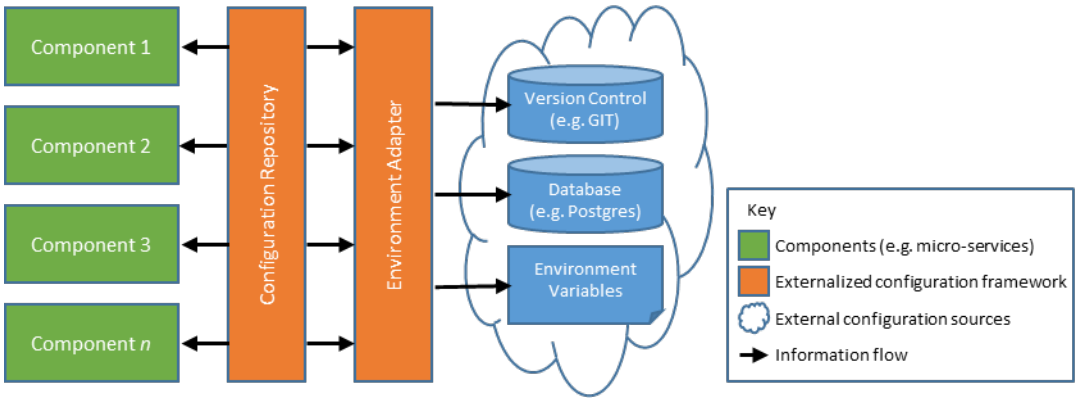
Solution

Configuration management is an important consideration for execution of an evolving public facing service when an application's SLA dictates that the down time should be keep to a minimum (or not tolerated) and maximum flexibility is essential in adjusting runtime settings. The Externalized Configuration pattern addresses this by moving configuration information out of the application deployment package to a centralize location; thus, providing flexibility for the configuration to vary independently throughout the lifecycle of its dependent application services. This approach promotes the sharing of configuration data between multiple services and the versioning of such configuration sets. By versioning your configuration sets you define which services use which set of configuration settings at any given time. Configuration sets can roll back and forth per service as needed.

The basic solution structure for externalized configuration has the following components:

- Environment – various sources of configuration name/value pairs; such as a version control system, a database, or even environment variables
- Environment adapter – components that understands a particular source of configuration and extracts name/value pairs from it
- Configuration repository – component that manages the aggregate configuration of software components. It takes all the configuration from the various environment adapters, resolving any conflicts, and determines the configuration of a given software component.
- Software components – business components that are interested in the configuration values from the environment. These are the things we would be implementing.

The following diagram illustrates the relationship between these components.



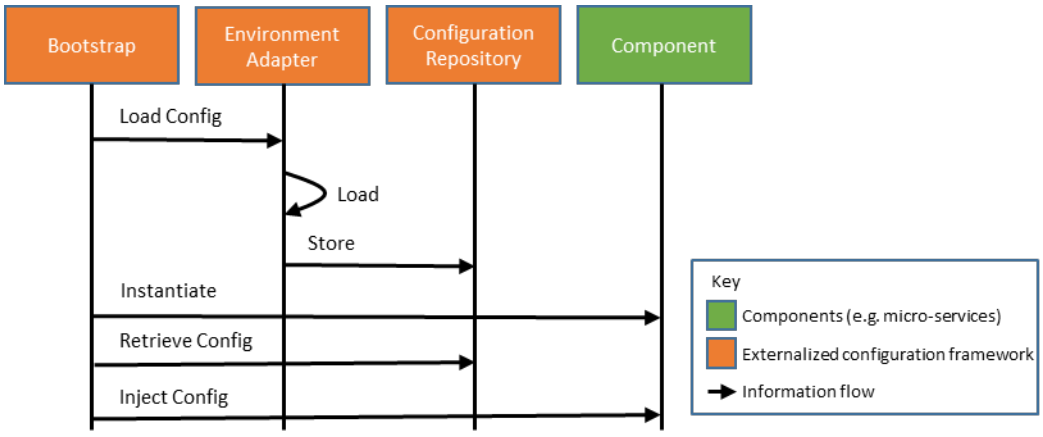
The diagram depicts the environment as a cloud because as far as the software components are concerned it is all one big source of configuration key/value pairs. Example environment sources could be such things as a version control system, a database, or environment variables. From an application perspective, the environment source types are irrelevant since they are external sources of configuration key/value pairs. It is the responsibility of the environment adapter to interface with a

given environment sources to obtain its configuration values. For each environment source, there needs to be an environment adapter that specializes in extracting configuration pairs from that source.

Upon completion of value extraction by all environment adapters, the configuration repository will interface with each and aggregate the configuration sources. However, there could be potential conflicts between the various environment sources. It is the job of the configuration repository to resolve such conflicts by application of a discrepancy rule set. A given configuration repository implementation's discrepancy rule set may be something as simple as using a priority order for each of the configuration sources. For example, we could assign priorities such as: version control system = 1, database = 2, and environment variables = 3. In this scenario, the configuration repository would resolve conflicts by selecting the value with the highest priority. This is not how all configuration repositories operate, but rather an example on how one might be implemented.

The business components receive their aggregated configuration values from the repository. The above diagram illustrates arrows going out from the configuration repository to the components. This shows a Dependency Injection (DI) approach to configuration where the container running the application has responsibility to take values from the configuration repository and inject them into the declared fields of a component. It is by no means a requirement to use DI in the externalized configuration pattern. We could easily reverse the direction of the arrows to illustrate a component's requirement to go and ask the configuration repository for its configuration values.

The following diagram illustrates the runtime interaction of the components that make up the basic solution to externalized configuration.



The bootstrap is any component within an application that is responsible for starting up the application, constructing business components, and wiring up details the business components need to do their job. In the case of a Spring application this would be the Spring context initializer. One of the first responsibilities of bootstrap is to instantiate the environment adapters and direct them to load their configuration from the environment. This diagram illustrates the bootstrap interfacing with a single environment adapter. However, there may be *n*-different environment adapters in play.

An environment adapter will interface with whatever resource it requires to load its configuration values. It will then take these values and store it in the configuration repository. The configuration repository will aggregate these values and resolve any conflicts. After resolution of the configuration by

the configuration repository, the bootstrap will instantiate the business components. This particular sequence diagram illustrates a DI approach to configuration; where the bootstrap node obtains configuration values from the repository and injects them into the declared fields of a component. However, we could easily have had the business components themselves interface with the configuration repository to obtain their configuration values.

Benefits of Solution

The first benefit is related to 'The Third Way' of DevOps which is a culture that nurtures continuous experimentation and learning. One of the ways we can do experiments is by decoupling deployments from feature releases through such things as feature toggles which is a powerful technique that allows modification of system behavior at runtime without modification of the underlying code. Externalized Configuration gives us a solid mechanism to support feature toggles.

Externalized Configuration provides a process to manage environment-centric configuration. One of the key ingredients of Continuous Delivery (CD) is the notation of configuration management. As an application transitions through different environments we need the ability to vary the configuration in a disciplined way without any modification of the deployment artifact. The Externalized Configuration pattern provides us with such a solution.

Externalized Configuration prevents the creation of configuration *snowflakes*. Traditional techniques for managing configuration might externalize configuration values from code, but then at deployment time recouples them in deployment artifact. Externalizing the configuration allows us to have immutable deployment artifacts throughout the various environments. The configuration for each environment varies in an external way; which keeps the application runtime processes disposal.

Externalized Configuration provides a mechanism for concurrent execution by toggling features in two different versions of a software component that is running concurrently. This allows running of multiple versions of an application at the same time to support different deployment strategies: such as blue/green deployments and/or canary releases.

Liabilities of Solution

Externalized Configuration introduces a new set of artifacts to manage. Since the configuration resides separately from the code we need to have: processes in place for managing the configuration, tools for working with the configuration, and polices on who can interact with the configuration. While Externalized Configuration promotes such things as DevOps and CD practices in the cloud, it does carry overhead.

Very often an external service will be used to maintain the configuration. In the Solution section, we used a Git version control system as an example of an environment source. In our case, a Git repository will be the environment source of our configuration in a Spring Cloud Config solution (discussed in the Solution Implementation section). Inclusion of such external services can add a degree of complexity since it is yet another piece of the overall system that must be managed.

Development can become more cumbersome when we are using Externalized Configuration. Productivity will suffer if the configuration store is not easily accessible in a development (desktop/laptop) environment. In such a case, we will need to come up with a solution to ensure a productive development flow.

Externalized Configuration has several new failure modes to consider. We now have to be concerned with the availability of external services that provide configuration (e.g. Git version control system). Another concern is how to handle incorrect, invalid, corrupted, or missing values in configuration store.

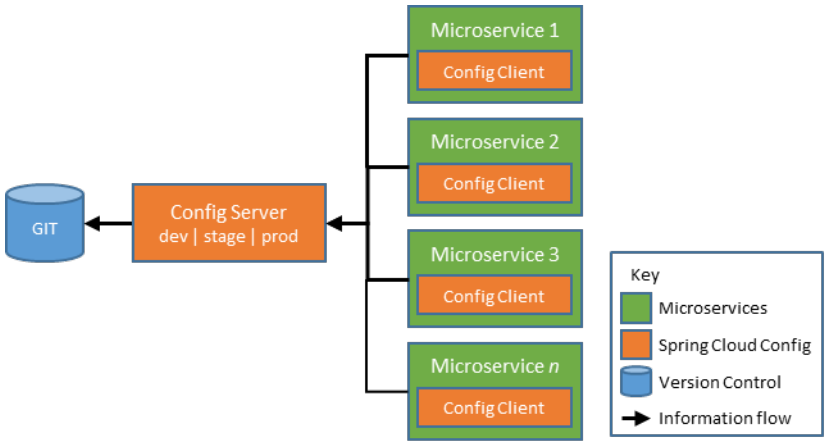
Solution Implementation

We will be using Spring Cloud Config as the externalized configuration implementation. Spring Cloud Config is one of the feature sets under the Spring Cloud project umbrella. As with other Spring Cloud features, it builds on Spring Boot by providing libraries that enhance the behavior of an application when added to the classpath – in this case providing an implementation of the externalized configuration design pattern.

Spring Cloud Config

Spring Cloud Config provides server and client-side support of the externalized configuration in a distributed system. The Spring Cloud Config server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties. The Config server also supports version control of the configuration properties. In our case, the Config server will store properties in a Git version-controlled repository.

The Spring Cloud Config server architecture is shown in the following diagram:



As shown in the preceding diagram, the Config client is embedded in each of our Spring Boot microservices. It does a configuration lookup from a central configuration server using a simple declarative mechanism, and stores properties into the Spring `Environment`. The configuration properties can be application-level configurations such as infrastructure-related configurations, server URLs, credentials, and so on.

Another thing to note is the "`dev | stage | prod`" label of the Config server. These are example profiles that the Config server supports. In Spring, profiles are named logical groupings that provide a way to segregate parts of the application configuration and make it only available in certain environments. Individual or entire collections of component definitions can be flagged to only work based on the profile that is active. A profile is activated by setting the `spring.profiles.active` property to the desired profile name.

Spring Cloud uses a bootstrap context, which is a parent context of the main application. The bootstrap context is responsible for loading configuration properties from the Config server. The bootstrap context looks for `bootstrap.yml` (or `.properties`) for loading initial configuration properties. To make this work in a Spring Boot application, rename the `application.*` file to `bootstrap.*`.

> **Note**
>
> The configuration of a Spring Boot application is externalized to property files that are defined in `application-${profile}.yml` (or `.properties`) – where `${profile}` is the name of the active profile. This promotes support of different target environments based on the name of the profile. In the case where the `spring.profiles.active` property is not set a `default` profile is assumed and is defined in an `application.yml` (or `.properties`) file.
>
> The bootstrap configuration is loaded by the parent Spring application context prior to application specific configuration. It is typically used to enable one or more profiles and reference configuration server attributes to facilitate loading of 'real' configuration data. The bootstrap configuration is defined in a `bootstrap.yml` (or `.properties`) file.

Config Server Setup

The following defines the necessary steps to setup a Config server:

1. Create a Gradle project for our Config server with the following structure:

```
config-server/
  └ build.gradle
  └ src/main/java/
     └ net/edwardsonthe/spring/cloud/config/
        └ ConfigServerApplication.java
  └ src/main/resources/
     └ bootstrap.yml
```

   At a minimum, the `build.gradle` file will have a compile dependency on `org.springframework.cloud:spring-cloud-config-server`. A basic `build.gradle` file for our example will look like the following:

```
buildscript {
  ext {
    springBootVersion = '1.5.6.RELEASE'
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
  }
  repositories {
    jcenter()
  }
}

apply plugin: 'eclipse'
apply plugin: 'java'
apply plugin: 'org.springframework.boot'
```

```
dependencies {
  compile 'org.springframework.cloud:spring-cloud-config-server'
}

dependencyManagement {
  imports {
    mavenBom 'org.springframework.cloud:spring-cloud-dependencies:Dalston.SR2'
  }
}

repositories {
  jcenter()
}
```

The `ConfigServerApplication` Java file will have the following content:

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

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

}
```

2. A local filesystem-based Git repository can be used as an example and for testing purposes. However, an external Git repository will be used in a production scenario. The following steps will use a local filesystem-based Git repository for demonstration purposes.

3. Enter the commands listed below to set up a local Git repository:

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo "message : Hello World" > application.yml
$ git add -A .
$ git commit -m "Added initial application.yml"
```

4. Edit the contents of the `bootstrap.yml` file as follows:

```
server:
  port: ${SERVER_PORT:8888}
spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: ${GIT_URI}
          clone-on-start: true
management:
  security:
    enabled: false
```

```
info:
  app:
    encoding: UTF-8
    java:
      source: 1.8
      target: 1.8
  description: MyProject Externalized Configuration Server
  project: MyProject
```

`server.port` is an optional property that defines the port of the Config server. The default port of `8888` will be used if the `server.port` property is not defined.

`spring.cloud.config.server.git.uri` is a mandatory property that references the URL of the Git repository created in the preceding step. In this example, this property will be set to the URI of the local filesystem-based Git repository – `file://${HOME}/config-repo`.

`clone-on-start` is a flag to indicate that the repository should be cloned on startup (not on demand). Generally, this leads to slower startup but faster first query.

`management.security.enabled` defines whether or not authentication is required to access actuator endpoints. By default, authentication is required, but has been disabled in this example for testing purposes.

`info.*` is customized information about our application that we wish to expose by the `/info` endpoint. The arbitrary properties defined under the `info` key will be automatically exposed.

5. Build and run the Config server from the command line:

```
config-server$ export GIT_URI=file://${HOME}/config-repo
config-server$ ./gradlew build
config-server$ ./gradlew bootRun
```

6. Verify that the application properties defined in previous step are accessible from our `config-server` using the URL `http://localhost:8888/application/default/master`. This can be done in a web-browser or from the command line using the `curl` command.

```
$ curl -s http://localhost:8888/application/default/master | jq
```

The above command should display contents similar to the following:

```
{
  "name": "application",
  "profiles": [
    "default"
  ],
  "label": "master",
  "version": "5e2293ec93a0141a6ffab9379d68ec492459f4ba",
  "state": null,
  "propertySources": [
    {
      "name": "file:///home/jeff/config-repo/application.yml",
      "source": {
        "message": "Hello World"
      }
    }
  ]
```

```
}
```

7. Spring Boot actuators provide an out-of-the-box mechanism to monitor and manage Spring Boot applications. Here are some of the most common endpoints Spring Boot provides out of the box:
    - `/autoconfig` – displays the auto-configuration report
    - `/dump` – performs a thread dump and displays the result
    - `/env` – list the environment variables and system properties discovered by the application at start-up
    - `/health` – show application health information (a simple 'status' when accessed over an unauthenticated connection or full message details when authenticated)
    - `/info` – display arbitrary information about the application
    - `/mappings` – lists all the HTTP request mappings
    - `/metrics` – show 'metrics' information
    - `/trace` – display trace information (by default the last few HTTP requests)

    As an example, obtain the application's health information as follows:

    ```
    $ curl -s http://localhost:8888/health | jq
    ```

    The above command should display contents similar to the following:

    ```
    {
      "status": "UP",
      "diskSpace": {
        "status": "UP",
        "total": 21463302144,
        "free": 6810849280,
        "threshold": 10485760
      },
      "refreshScope": {
        "status": "UP"
      },
      "configServer": {
        "status": "UP",
        "repositories": [
          {
            "sources": [
              "file:///home/${HOME}/config-repo/application.yml"
            ],
            "name": "app",
            "profiles": [
              "default"
            ],
            "label": null
          }
        ]
      }
    }
    ```

Understanding the Config server URL

In the previous section, we used `http://localhost:8888/application/default/master` to explore the properties. How do we interpret this URL?

The first element in the URL is the application name of the requesting service – not the Config server application, but rather the application that consumes configuration from the Config server. In the given example, the application name should be `application`. The application name is a logical name given to each application, using the `spring.application.name` property in its `bootstrap.yml` file. Each application must have a **unique** name. The Config server will use the name to resolve and pick up the appropriate properties from the Config server repository. For example, if there is an application with the name `myapp`, then there should be a `myapp.yml` in the configuration repository to store all the properties related to that application. The application name is also sometimes referred to as the service ID of the component.

The second part of the URL represents the profile. There can be more than one profile configured within the repository for an application. The profiles can be used in various scenarios. The two common scenarios are segregating different environments such as dev, test, stage, prod, and the like, or segregating server configurations such as primary, secondary, and so on. The first one represents different environments of an application, whereas the second one represents different servers where an application is deployed.

The profile names are logical names that will be used for matching the file name in the repository. The default profile is named `default`. To configure properties for different environments, we have to configure different files as given in the following example. In this example, the first file is for a default environment, the second is for a development environment, and the third is for the production environment:

```
application.yml
application-development.yml
application-production.yml
```

These would be accessible using the following URLs respectively:

```
http://localhost:8888/application/default
http://localhost:8888/application/development
http://localhost:8888/application/production
```

The last part of the URL is the label, and is named `master` by default. The label is an optional Git label that can be used, if required.

In short, the URL is based on the following pattern:

```
http://localhost:8888/{name}/{profile}/{label}
```

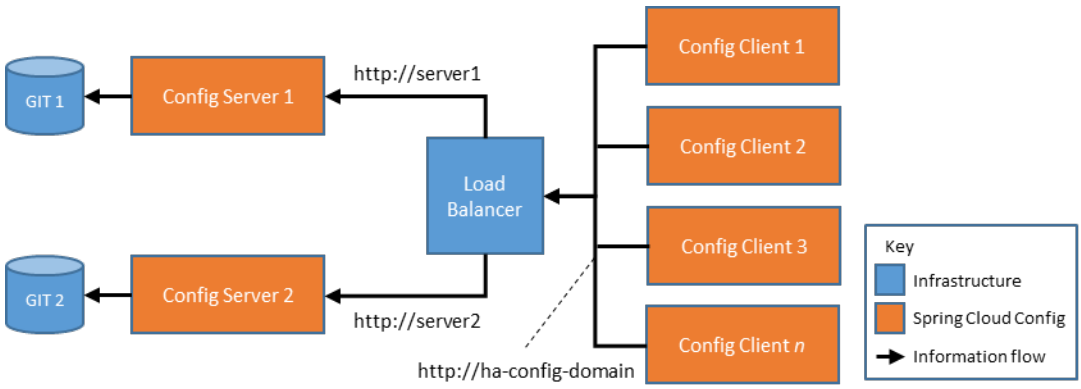
The configuration can also be accessed by ignoring the profile. In the preceding example, all the following three URLs point to the same configuration:

```
http://localhost:8888/application/default
http://localhost:8888/application/master
http://localhost:8888/application/default/master
```

There is an option to have different Git repositories for different profiles. This would makes sense in a production environment, since the access to different repositories could be different for a given profile.

High Availability of the Config Server

There are two single points of failure in the architecture that was established in the preceding section. One of them is the availability of the Config server itself, and the second one is the Git repository. The architecture needs to be enhanced to ensure high availability of both of these services. The following diagram illustrates a high availability architecture for both the Config server and the Git repository.



The Config server requires high availability, since Config clients won't be able to bootstrap if the Config server is not available. Hence, redundant Config servers are required for high availability. However, the applications can continue to run if the Config server is unavailable after their services are bootstrapped. In this case, services will run with the last known configuration state. Hence, the Config server availability is not at the same critical level as the microservices availability.

In order to make the Config server highly available, we need multiple instances of both the Config server as well as the Git repository. Since the Config server is a stateless HTTP service, multiple instances of configuration servers can be run in parallel. Based on the load on the configuration server, a number of instances have to be adjusted. A Config client's `bootstrap.yml` file is not capable of handling more than one server address. Hence, multiple configuration servers should be configured to run behind a load balancer or behind a local DNS with failover and fallback capabilities. The load balancer or DNS server URL will be configured in the microservices' `bootstrap.yml` file. This is with the assumption that the DNS or the load balancer is highly available and capable of handling failovers.

Config Client Setup

The following defines the necessary steps to setup a Config client that will obtain its configuration from the Config server implemented in the preceding section:

1. Create a Gradle project for our Config client with the following structure:

```
config-client/
  └ build.gradle
  └ src/main/java/
     └ net/edwardsonthe/spring/cloud/config/
        └ ConfigClientApplication.java
  └ src/main/resources/
     └ bootstrap.yml
```

   At a minimum, the `build.gradle` file will have a compile dependency on `org.springframework.boot:spring-boot-starter-actuator` and `org.springframework.cloud:spring-cloud-config-client`. In addition to the required dependencies, we

will also include the `org.springframework.boot:spring-boot-starter-web` dependency to expose a web service end-point of the Config client. The `build.gradle` file for our Config client will look like the following:

```
 buildscript {
  ext {
    springBootVersion = '1.5.6.RELEASE'
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
  }
  repositories {
    jcenter()
  }
}

apply plugin: 'eclipse'
apply plugin: 'java'
apply plugin: 'org.springframework.boot'

dependencies {
  compile 'org.springframework.boot:spring-boot-starter-actuator'
  compile 'org.springframework.boot:spring-boot-starter-web'
  compile 'org.springframework.cloud:spring-cloud-config-client'
}

dependencyManagement {
  imports {
    mavenBom 'org.springframework.cloud:spring-cloud-dependencies:Dalston.SR2'
  }
}

repositories {
  jcenter()
}
```

The `ConfigClientApplication` Java file will have the following content:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class ConfigClientApplication {

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

}

@RefreshScope
@RestController
class MessageRestController {
```

```
  @Value("${message}")
  private String message;

  @RequestMapping("/message")
  String getMessage() {
    return this.message + "\n";
  }

}
```

Note

In Spring, a bean annotated with `@RefreshScope` can have its configuration refreshed at runtime. This addresses the problem of stateful beans that only get their configuration injected when they are initialized. For instance, if a `DataSource` has open connections when the database URL is changed via the `Environment`, we probably want the holders of those connections to be able to complete what they are doing. Then the next time someone borrows a connection from the pool they get one with the new URL. This approach follows the Runtime Reconfiguration design pattern, which is a design pattern built on top of Externalized Configuration.

2. Edit the contents of the `bootstrap.yml` file as follows:

```
server:
  port: ${SERVER_PORT:8889}
spring:
  application:
    name: config-client
  cloud:
    config:
      uri: ${CONFIG_SERVER_URL:http://localhost:8888}
management:
  security:
    enabled: false
```

The `spring.application.name` property is the logical, unique name given to our example Config client. It is used by the Config server to resolve the appropriate properties from the Config server repository for our Config client.

`spring.cloud.config.uri` is the URI of the remote Config server.

3. Enter the commands listed below to update our local Git repository:

```
$ cd $HOME
$ cd config-repo
$ echo "message : Config Client Message" > config-client.yml
$ git add -A .
$ git commit -m "Added initial config-client.yml"
```

Note

The prefix of the generated YAML file is equivalent to the `spring.application.name` property defined in the preceding step.

4. Build and run the Config client from the command line:

```
config-client$ export CONFIG_SERVER_URL=http://localhost:8888
config-client$ ./gradlew build
config-client$ ./gradlew bootRun
```

5. Verify that the web-service endpoint of the Config client is accessible using the URL `http://localhost:8889/message` and the appropriate `message` property is obtained from the Config server. This can be done in a web-browser or from the command line using the `curl` command.

```
$ curl -s http://localhost:8889/message
```

   The above command should display the following:

```
Config Client Message
```

6. Enter the commands listed below to update the `config-client.yml` file:

```
$ cd $HOME
$ cd config-repo
$ echo "message : Updated Config Client Message" > config-client.yml
$ git add -A .
$ git commit -m "Updated config-client.yml"
```

7. Invoke the `/refresh` Spring Boot Actuator operational endpoint of the Config client in order to force the refresh of its configuration values. This is accomplished by sending an empty HTTP POST to the client's `/refresh` endpoint with the URL `http://localhost:8889/refresh`.

```
$ curl -s -X POST http://localhost:8889/refresh
```

   Confirm that the `refresh` worked by reviewing the `http://localhost:8889/message` endpoint:

```
$ curl -s http://localhost:8889/message
```

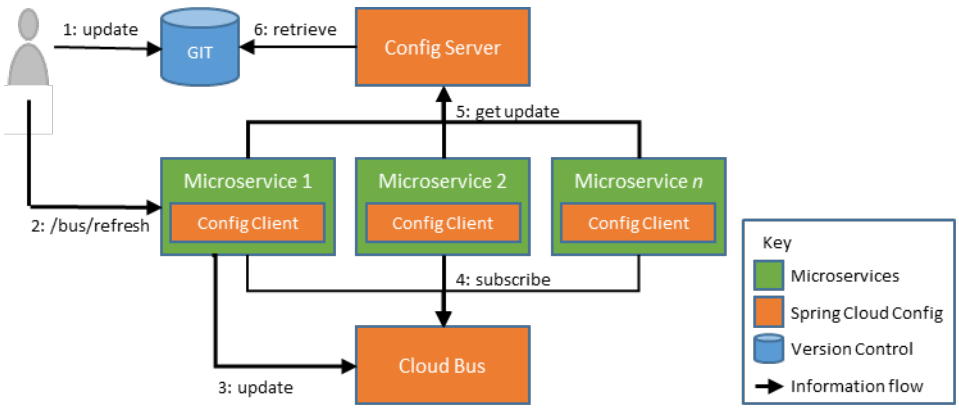
   The above command should display the following:

```
Updated Config Client Message
```

Spring Cloud Bus

With the preceding approach, configuration parameters can be changed without restarting the microservices. This is good when there are only one or two instances of the services running. However, what happens if there are many instances? If there are five instances, then we have to hit `/refresh` against each service instance.

The following diagram shows a solution to this problem using the Spring Cloud Bus architecture:



Spring Cloud Bus, an implementation of Runtime Reconfiguration design pattern, provides a mechanism to refresh configurations across multiple instances without knowing how many instances there are, or their locations. This is particularly handy when there are many service instances of a microservice running or when there are many microservices of different types running. This is done by connecting all service instances through a single message broker. Each instance subscribes for change events, and refreshes its local configuration when required. This refresh is triggered by making a call to any one instance by hitting the `/bus/refresh` endpoint, which then propagates the changes through the cloud bus and the common message broker.

Currently, Spring Cloud Bus supports both an AMQP broker and Apache Kafka as the transport mechanism. To enable the bus, add `org.springframework.cloud:spring-cloud-starter-bus-amqp` or `org.springframework.cloud:spring-cloud-starter-bus-kafka` to the dependency management section of your `build.gradle` file. It is required that the corresponding broker – RabbitMQ or Kafka – is available and configured: when running locally you shouldn't have to do anything, but if you are running remotely you will need to use the Spring Cloud Connectors, or Spring Boot conventions to define the broker credentials. For example, for RabbitMQ the following would be required to be added in the client's `bootstrap.yml` file:

```
spring:
  rabbitmq:
    host: broker-hostname
    port: broker-port
    username: broker-username
    password: broker-password
```

The following steps illustrate how to use RabbitMQ as the AMQP message broker for Spring Cloud Bus. A precondition is to ensure that RabbitMQ is running on your local machine with a default configuration.

1. Add an `org.springframework.cloud:spring-cloud-starter-bus-amqp` dependency to your `build.gradle` file.

2. Rebuild and restart the Config client microservice. In this case, we will run two instances of the Config client from a command line, as follows:

```
config-server$ ./gradlew build
config-server$ java -jar -Dserver.port=8889 build/libs/config-client.jar
config-server$ java -jar -Dserver.port=8890 build/libs/config-client.jar
```

The two instances of the Config client microservice will be now running – one on port 8889 and another one on 8890.

> **Note**
>
> The system property `server.port` that is passed in on the command line maps to the `SERVER_PORT` property referenced in the `bootstrap.yml` file. As with many properties referenced in the `bootstrap.yml` file, we have flexibility to associate values as either system properties or environment variables.
>
> The `config-client.jar` referenced above is the Spring Boot über JAR created by the `org.springframework.boot` Gradle plugin. The `org.springframework.boot` plugin assembles the über JAR as part of the `build` task.

3. Execute the following from the command line to refresh both Config clients:

```
$ curl -s -X  POST http://localhost:8889/bus/refresh
```

4. Verify the following message is displayed for both instances:

```
Received remote refresh request. Keys refreshed
```

Although the `/bus/refresh` command was sent to a single endpoint, both running Config clients refreshed their configurations. As illustrated in the preceding Spring Cloud Bus architecture diagram, the bus endpoint that receives the refresh command sends a message to the message broker internally, which is eventually consumed by all instances, resulting in the reloading of their configuration `Environment`.

Config Client Bootstrap

There are two means to bootstrap a Config client: Config first or Discovery first.

Config First Bootstrap

Config first bootstrapping is the default behavior for any application having the `spring-cloud-config-client` dependency on its classpath. When a client starts up it binds to the Config server via the bootstrap configuration property `spring.cloud.config.uri` and initializes Spring's `Environment` with the remote property sources. This approach assumes the Config server has a *well-know* URI location.

With config first bootstrapping, a Config client can be configured to retry a Config server connection. This is ideal in a scenario where the Config client starts up prior to the Config server. To accomplish this, the dependencies `org.springframework.boot:spring-boot-starter-aop` and `org.springframework.retry:spring-retry` are required on the classpath. You will also need to set '`spring.cloud.config.failFast: true`' in `bootstrap.yml` file. Retry behavior can further be customized using the `spring.cloud.config.retry.*` properties:

- `initialInterval` - initial retry interval in milliseconds (default of 1000)
- `multiplier` - multiplier for next interval (default of 1.1)
- `maxInterval` - maximum interval for back-off in milliseconds (default of 2000)
- `maxAttempts` - maximum number of attempts (default of 6)

Discover First Bootstrap

Discovery first bootstrapping is when the Config server registers with a discovery service such as Spring Cloud Netflix and Eureka Service Discovery. Config clients would then be configured to use the `DiscoveryService` to locate the Config server. The benefit of this approach is that it provides the flexibility for the Config server to change its co-ordinates, as long as the `DiscoveryService` has a *well-know* location. However, it costs an extra network round trip on startup – for each Config client – to locate the Config server registration.

> **Note**
>
> The Config first bootstrapping approach seems to be the ideal candidate. Having the Config server defined with a *well-know* URI location seems like the logical choice and is complementary to the discussion above regarding High Availability of the Config Server.