Spring Cloud

bush

Published
with GitBook

Table of Contents

undefined# Spring Cloud {#spring-cloud}

undefined# Spring Cloud {#spring-cloud}

Table of Contents

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

undefined## Features {#features}

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging

undefined# Cloud Native Applications {#cloud-native-applications}

Cloud Native is a style of application development that encourages easy adoption of best practices in the areas of continuous delivery and value-driven development. A related discipline is that of building 12-factor Apps in which development practices are aligned with delivery and operations goals, for instance by using declarative programming and management and monitoring. Spring Cloud facilitates these styles of development in a number of specific ways and the starting point is a set of features that all components in a distributed system either need or need easy access to when required.

Many of those features are covered by Spring Boot, which we build on in Spring Cloud. Some more are delivered by Spring Cloud as two libraries: Spring Cloud Context and Spring Cloud Commons. Spring Cloud Context provides utilities and special services for the `ApplicationContext` of a Spring Cloud application (bootstrap context, encryption, refresh scope and environment endpoints). Spring Cloud Commons is a set of abstractions and common classes used in different Spring Cloud implementations (eg. Spring Cloud Netflix vs. Spring Cloud Consul).

If you are getting an exception due to "Illegal key size" and you are using Sun's JDK, you need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. See the following links for more information:

- Java 6 JCE Link http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html
- Java 7 JCE Link http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html
- Java 8 JCE Link http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html

Extract files into JDK/jre/lib/security folder (whichever version of JRE/JDK x64/x86 you are using).

| Note | Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at {githubmaster}/docs/src/main/asciidoc[github]. | | --- | --- |

undefined## Spring Cloud Context: Application Context Services {#spring-cloud-context-application-context-services}

Spring Boot has an opinionated view of how to build an application with Spring: for instance it has conventional locations for common configuration file, and endpoints for common management and monitoring tasks. Spring Cloud builds on top of that and adds a few features that probably all components in a system would use or occasionally need.

The Bootstrap Application Context

A Spring Cloud application operates by creating a "bootstrap" context, which is a parent context for the main application. Out of the box it is responsible for loading configuration properties from the external sources, and also decrypting properties in the local external configuration files. The two contexts share an `Environment` which is the source of external properties for any Spring application. Bootstrap properties are added with high precedence, so they cannot be overridden by local configuration.

The bootstrap context uses a different convention for locating external configuration than the main application context, so instead of `application.yml` (or `.properties`) you use `bootstrap.yml`, keeping the external configuration for bootstrap and main context nicely separate. Example:

bootstrap.yml

```
spring:
  application:
    name: foo
  cloud:
    config:
      uri: ${SPRING_CONFIG_URI:http://localhost:8888}
```

It is a good idea to set the `spring.application.name` (in `bootstrap.yml` or `application.yml`) if your application needs any application-specific configuration from the server.

You can disable the bootstrap process completely by setting `spring.cloud.bootstrap.enabled=false` (e.g. in System properties).

Application Context Hierarchies

If you build an application context from `SpringApplication` or `SpringApplicationBuilder`, then the Bootstrap context is added as a parent to that context. It is a feature of Spring that child contexts inherit property sources and profiles from their parent, so the "main"

application context will contain additional property sources, compared to building the same context without Spring Cloud Config. The additional property sources are:

- "bootstrap": an optional `CompositePropertySource` appears with high priority if any `PropertySourceLocators` are found in the Bootstrap context, and they have non-empty properties. An example would be properties from the Spring Cloud Config Server. See below for instructions on how to customize the contents of this property source.
- "applicationConfig: [classpath:bootstrap.yml]" (and friends if Spring profiles are active). If you have a `bootstrap.yml` (or properties) then those properties are used to configure the Bootstrap context, and then they get added to the child context when its parent is set. They have lower precedence than the `application.yml` (or properties) and any other property sources that are added to the child as a normal part of the process of creating a Spring Boot application. See below for instructions on how to customize the contents of these property sources.

Because of the ordering rules of property sources the "bootstrap" entries take precedence, but note that these do not contain any data from `bootstrap.yml`, which has very low precedence, but can be used to set defaults.

You can extend the context hierarchy by simply setting the parent context of any `ApplicationContext` you create, e.g. using its own interface, or with the `SpringApplicationBuilder` convenience methods (`parent()` , `child()` and `sibling()`). The bootstrap context will be the parent of the most senior ancestor that you create yourself. Every context in the hierarchy will have its own "bootstrap" property source (possibly empty) to avoid promoting values inadvertently from parents down to their descendants. Every context in the hierarchy can also (in principle) have a different `spring.application.name` and hence a different remote property source if there is a Config Server. Normal Spring application context behaviour rules apply to property resolution: properties from a child context override those in the parent, by name and also by property source name (if the child has a property source with the same name as the parent, the one from the parent is not included in the child).

Note that the `SpringApplicationBuilder` allows you to share an `Environment` amongst the whole hierarchy, but that is not the default. Thus, sibling contexts in particular do not need to have the same profiles or property sources, even though they will share common things with their parent.

Changing the Location of Bootstrap Properties

The `bootstrap.yml` (or `.properties) location can be specified using` spring.cloud.bootstrap.name `(default "bootstrap") or` spring.cloud.bootstrap.location `(default empty), e.g. in System properties. Those`

`properties behave like the` spring.config.* `variants with the same name, in fact they are used to set up the bootstrap` ApplicationContext `by setting those properties in its` Environment `. If there is an active profile (from` spring.profiles.active `or through the` Environment `API in the context you are building) then properties in that profile will be loaded as well, just like in a regular Spring Boot app, e.g. from` bootstrap-development.properties` for a "development" profile.

Customizing the Bootstrap Configuration

The bootstrap context can be trained to do anything you like by adding entries to `/META-INF/spring.factories` under the key `org.springframework.cloud.bootstrap.BootstrapConfiguration`. This is a comma-separated list of Spring `@Configuration` classes which will be used to create the context. Any beans that you want to be available to the main application context for autowiring can be created here, and also there is a special contract for `@Beans` of type `ApplicationContextInitializer`. Classes can be marked with an `@Order` if you want to control the startup sequence (the default order is "last").

Warning	**Be careful when adding custom `BootstrapConfiguration` that the classes you add are not `@ComponentScanned` by mistake into your "main" application context, where they might not be needed. Use a separate package name for boot configuration classes that is not already covered by your `@ComponentScan` or `@SpringBootApplication` annotated configuration classes.**

The bootstrap process ends by injecting initializers into the main `SpringApplication` instance (i.e. the normal Spring Boot startup sequence, whether it is running as a standalone app or deployed in an application server). First a bootstrap context is created from the classes found in `spring.factories` and then all `@Beans` of type `ApplicationContextInitializer` are added to the main `SpringApplication` before it is started.

Customizing the Bootstrap Property Sources

The default property source for external configuration added by the bootstrap process is the Config Server, but you can add additional sources by adding beans of type `PropertySourceLocator` to the bootstrap context (via `spring.factories`). You could use this to insert additional properties from a different server, or from a database, for instance.

As an example, consider the following trivial custom locator:

```
@Configuration
public class CustomPropertySourceLocator implements PropertySourceLocator {

    @Override
    public PropertySource<?> locate(Environment environment) {
        return new MapPropertySource("customProperty",
                Collections.<String, Object>singletonMap("property.from.sample.custom.sou
    }

}
```

The `Environment` that is passed in is the one for the `ApplicationContext` about to be created, i.e. the one that we are supplying additional property sources for. It will already have its normal Spring Boot-provided property sources, so you can use those to locate a property source specific to this `Environment` (e.g. by keying it on the `spring.application.name`, as is done in the default Config Server property source locator).

If you create a jar with this class in it and then add a `META-INF/spring.factories` containing:

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=sample.c
```

then the "customProperty" `PropertySource` will show up in any application that includes that jar on its classpath.

Environment Changes

The application will listen for an `EnvironmentChangedEvent` and react to the change in a couple of standard ways (additional `ApplicationListeners` can be added as `@Beans` by the user in the normal way). When an `EnvironmentChangedEvent` is observed it will have a list of key values that have changed, and the application will use those to:

- Re-bind any `@ConfigurationProperties` beans in the context
- Set the logger levels for any properties in `logging.level.*`

Note that the Config Client does not by default poll for changes in the `Environment`, and generally we would not recommend that approach for detecting changes (although you could set it up with a `@Scheduled` annotation). If you have a scaled-out client application then it is better to broadcast the `EnvironmentChangedEvent` to all the instances instead of having them polling for changes (e.g. using the Spring Cloud Bus).

The `EnvironmentChangedEvent` covers a large class of refresh use cases, as long as you can actually make a change to the `Environment` and publish the event (those APIs are public and part of core Spring). You can verify the changes are bound to `@ConfigurationProperties` beans by visiting the `/configprops` endpoint (normal Spring Boot Actuator feature). For instance a `DataSource` can have its `maxPoolSize` changed at runtime (the default `DataSource` created by Spring Boot is an `@ConfigurationProperties` bean) and grow capacity dynamically. Re-binding `@ConfigurationProperties` does not cover another large class of use cases, where you need more control over the refresh, and where you need a change to be atomic over the whole `ApplicationContext`. To address those concerns we have `@RefreshScope`.

Refresh Scope

A Spring `@Bean` that is marked as `@RefreshScope` will get special treatment when there is a configuration change. 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 he gets one with the new URL.

Refresh scope beans are lazy proxies that initialize when they are used (i.e. when a method is called), and the scope acts as a cache of initialized values. To force a bean to re-initialize on the next method call you just need to invalidate its cache entry.

The `RefreshScope` is a bean in the context and it has a public method `refreshAll()` to refresh all beans in the scope by clearing the target cache. There is also a `refresh(String)` method to refresh an individual bean by name. This functionality is exposed in the `/refresh` endpoint (over HTTP or JMX).

Note	**`@RefreshScope` works (technically) on an `@Configuration` class, but it might lead to surprising behaviour: e.g. it does not mean that all the `@Beans` defined in that class are themselves `@RefreshScope`. Specifically, anything that depends on those beans cannot rely on them being updated when a refresh is initiated, unless it is itself in `@RefreshScope` (in which it will be rebuilt on a refresh and its dependencies re-injected, at which point they will be re-initialized from the refreshed `@Configuration`).**

Encryption and Decryption

The Config Client has an `Environment` pre-processor for decrypting property values locally. It follows the same rules as the Config Server, and has the same external configuration via `encrypt.*`. Thus you can use encrypted values in the form `{cipher}*` and as long as there

is a valid key then they will be decrypted before the main application context gets the `Environment`. To use the encryption features in a client you need to include Spring Security RSA in your classpath (Maven co-ordinates "org.springframework.security:spring-security-rsa") and you also need the full strength JCE extensions in your JVM.

include::jce.adoc

Endpoints

For a Spring Boot Actuator application there are some additional management endpoints:

- POST to `/env` to update the `Environment` and rebind `@ConfigurationProperties` and log levels
- `/refresh` for re-loading the boot strap context and refreshing the `@RefreshScope` beans
- `/restart` for closing the `ApplicationContext` and restarting it (disabled by default)
- `/pause` and `/resume` for calling the `Lifecycle` methods (`stop()` and `start()` on the `ApplicationContext`)

undefined## Spring Cloud Commons: Common Abstractions {#spring-cloud-commons-common-abstractions}

Patterns such as service discovery, load balancing and circuit breakers lend themselves to a common abstraction layer that can be consumed by all Spring Cloud clients, independent of the implementation (e.g. discovery via Eureka or Consul).

Spring RestTemplate as a Load Balancer Client

You can use Ribbon indirectly via an autoconfigured `RestTemplate` when RestTemplate is on the classpath and a `LoadBalancerClient` bean is defined):

```
public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    public String doOtherStuff() {
        String results = restTemplate.getForObject("http://stores/stores", String.class);
        return results;
    }
}
```

The URI needs to use a virtual host name (ie. service name, not a host name). The Ribbon client is used to create a full physical address. See RibbonAutoConfiguration for details of how the `RestTemplate` is set up.

Multiple RestTemplate objects

If you want a `RestTemplate` that is not load balanced, create a `RestTemplate` bean and inject it as normal. To access the load balanced `RestTemplate use the provided [[email protected]](/cdn-cgi/l/email-protection)`

`**Illegal HTML tag removed :** /* */`

`Qualifier` :

```
public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    @LoadBalanced
    private RestTemplate loadBalanced;

    public String doOtherStuff() {
        return loadBalanced.getForObject("http://stores/stores", String.class);
    }

    public String doStuff() {
        return restTemplate.getForObject("http://example.com", String.class);
    }
}
```

undefined# Spring Cloud Config {#spring-cloud-config}

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the Spring `Environment` and `PropertySource` abstractions, so they fit very well with Spring applications, but can be used with any application running in any language. As an application moves through the deployment pipeline from dev to test and into production you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git so it easily supports labelled versions of configuration environments, as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

| Note | Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at {githubmaster}/docs/src/main/asciidoc[github]. | | --- | --- |

undefined## Quick Start {#quick-start}

Start the server:

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

The server is a Spring Boot application so you can build the jar file and run that (`java -jar …`) or pull it down from a Maven repository. Then try it out as a client:

```
$ curl localhost:8888/foo/development
{"name":"development","label":"master","propertySources":[
  {"name":"https://github.com/scratches/config-repo/foo-development
  {"name":"https://github.com/scratches/config-repo/foo.properties"
]}
```

The default strategy for locating property sources is to clone a git repository (at "spring.cloud.config.server.git.uri") and use it to initialize a mini `SpringApplication` . The mini-application's `Environment` is used to enumerate property sources and publish them via a JSON endpoint.

The HTTP service has resources in the form:

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

where the "application" is injected as the "spring.config.name" in the `SpringApplication` (i.e. what is normally "application" in a regular Spring Boot app), "profile" is an active profile (or comma-separated list of properties), and "label" is an optional git label (defaults to "master".)

The YAML and properties forms are coalesced into a single map, even if the origin of the values (reflected in the "propertySources" of the "standard" form) has multiple sources.

Spring Cloud Config Server pulls configuration for remote clients from a git repository (which must be provided):

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

Client Side Usage

To use these features in an application, just build it as a Spring Boot application that depends on spring-cloud-config-client (e.g. see the test cases for the config-client, or the sample app). The most convenient way to add the dependency is via a Spring Boot starter `org.springframework.cloud:spring-cloud-starter-config`. There is also a parent pom and BOM (`spring-cloud-starter-parent`) for Maven users and a Spring IO version management properties file for Gradle and Spring CLI users. Example Maven configuration:

pom.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 -->
```

Then you can create a standard Spring Boot application, like this simple HTTP server:

<pre>@Configuration @EnableAutoConfiguration @RestController public class Application {

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

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

}</pre>

When it runs it will pick up the external configuration from the default local config server on port 8888 if it is running. To modify the startup behaviour you can change the location of the config server using `bootstrap.properties` (like `application.properties` but for the bootstrap phase of an application context), e.g.

```
spring.cloud.config.uri: http://myconfigserver.com
```

The bootstrap properties will show up in the `/env` endpoint as a high-priority property source, e.g.

```
$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/spring-cloud-samples/config-rep
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

(a property source called "configService:/" contains the property "foo" with value "bar" and is highest priority).

Note	the URL in the property source name is the git repository not the config server URL.

undefined## Spring Cloud Config Server {#spring-cloud-config-server}

The Server provides an HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content). The server is easily embeddable in a Spring Boot application using the `@EnableConfigServer` annotation.

Environment Repository

Where do you want to store the configuration data for the Config Server? The strategy that governs this behaviour is the `EnvironmentRepository`, serving `Environment` objects. This `Environment` is a shallow copy of the domain from the Spring `Environment` (including `propertySources` as the main feature). The `Environment` resources are parametrized by three variables:

- `{application}` maps to "spring.application.name" on the client side;
- `{profile}` maps to "spring.active.profiles" on the client (comma separated list); and
- `{label}` which is a server side feature labelling a "versioned" set of config files.

Repository implementations generally behave just like a Spring Boot application loading configuration files from a "spring.config.name" equal to the `{application}` parameter, and "spring.profiles.active" equal to the `{profiles}` parameter. Precedence rules for profiles are also the same as in a regular Boot application: active profiles take precedence over defaults, and if there are multiple profiles the last one wins (like adding entries to a `Map`).

Example: a client application has this bootstrap configuration:

bootstrap.yml

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

(as usual with a Spring Boot application, these properties could also be set as environment variables or command line arguments).

If the repository is file-based, the server will create an `Environment` from `application.yml` (shared between all clients), and `foo.yml` (with `foo.yml` taking precedence). If the YAML files have documents inside them that point to Spring profiles, those are applied with higher precendence (in order of the profiles listed), and if there are profile-specific YAML (or

properties) files these are also applied with higher precedence than the defaults. Higher precendence translates to a `PropertySource` listed earlier in the `Environment`. (These are the same rules as apply in a standalone Spring Boot application.)

Git Backend

The default implementation of `EnvironmentRepository` uses a Git backend, which is very convenient for managing upgrades and physical environments, and also for auditing changes. To change the location of the repository you can set the "spring.cloud.config.server.git.uri" configuration property in the Config Server (e.g. in `application.yml`). If you set it with a `file:` prefix it should work from a local repository so you can get started quickly and easily without a server, but in that case the server operates directly on the local repository without cloning it (it doesn't matter if it's not bare because the Config Server never makes changes to the "remote" repository). To scale the Config Server up and make it highly available, you would need to have all instances of the server pointing to the same repository, so only a shared file system would work. Even in that case it is better to use the `ssh:` protocol for a shared filesystem repository, so that the server can clone it and use a local working copy as a cache.

This repository implementation maps the `{label}` parameter of the HTTP resource to a git label (commit id, branch name or tag). If the git branch or tag name contains a slash ("/") then the label in the HTTP URL should be specified with the special string "(_)" instead (to avoid ambiguity with other URL paths). Be careful with the brackets in the URL if you are using a command line client like curl (e.g. escape them from the shell with quotes '').

Spring Cloud Config Server supports a single or multiple git repositories:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: pattern*,*pattern1*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

In the above example, if `{application}` does not match any of the patterns, it will use the default uri defined under "spring.cloud.config.server.git.uri". For the "simple" repository, the pattern is "simple" (i.e. it only matches one application named "simple"). The pattern format is a comma-separated list of application names with wildcards (a pattern beginning with a wildcard may need to be quoted).

Note	**the "one-liner" short cut used in the "simple" example above can only be used if the only property to be set is the URI. If you need to set anything else (credentials, pattern, etc.) you need to use the full form.**

Every repository can also optionally store config files in sub-directories, and patterns to search for those directories can be specified as `searchPaths`. For example at the top level:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: foo,bar*
```

In this example the server searches for config files in the top level and in the "foo/" sub-directory and also any sub-directory whose name begins with "bar".

By default the server clones remote repositories when configuration is first requested. The server can be configured to clone the repositories at startup. For example at the top level:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
                pattern: team-a-*
                cloneOnStart: true
                uri: http://git/team-a/config-repo.git
            team-b:
                pattern: team-b-*
                cloneOnStart: false
                uri: http://git/team-b/config-repo.git
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

In this example the server clones team-a’s config-repo on startup before it accepts any requests. All other repositories will not be cloned until configuration from the repository is requested.

To use HTTP basic authentication on the remote repository add the "username" and "password" properties separately (not in the URL), e.g.

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          username: trolley
          password: strongpassword
```

If you don’t use HTTPS and user credentials, SSH should also work out of the box when you store keys in the default directories (`~/.ssh`) and the uri points to an SSH location, e.g. "

Illegal HTML tag removed : / /

:configuration/cloud-configuration". The repository is accessed using JGit, so any documentation you find on that should be applicable.

File System Backend

There is also a "native" profile in the Config Server that doesn’t use Git, but just loads the config files from the local classpath or file system (any static URL you want to point to with "spring.cloud.config.server.native.searchLocations"). To use the native profile just launch the Config Server with "spring.profiles.active=native".

Warning	**The default value of the `searchLocations` is identical to a local Spring Boot application (so `[classpath:/, classpath:/config, file:./, file:./config]`) which will expose the `application.properties` from the server to all clients.**

Tip	**A filesystem backend is great for getting started quickly and for testing. To use it in production you need to be sure that the file system is reliable, and shared across all instances of the Config Server.**

This repository implementation maps the `{label}` parameter of the HTTP resource to a suffix on the search path, so properties files are loaded from each search location **and** a subdirectory with the same name as the label (the labelled properties take precedence in the Spring Environment).

Health Indicator

Config Server comes with a Health Indicator that checks if the configured `EnvironmentRepository` is working. By default it asks the `EnvironmentRepository` for an application named `app` , the `default` profile and the default label provided by the `EnvironmentRepository` implementation.

You can configure the Health Indicator to check more applications along with custom profiles and custom labels, e.g.

```
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
              profiles: development
```

You can disable the Health Indicator by setting `spring.cloud.config.server.health.enabled=false`.

Security

You are free to secure your Config Server in any way that makes sense to you (from physical network security to OAuth2 bearer tokens), and Spring Security and Spring Boot make it easy to do pretty much anything.

To use the default Spring Boot configured HTTP Basic security, just include Spring Security on the classpath (e.g. through `spring-boot-starter-security`). The default is a username of "user" and a randomly generated password, which isn't going to be very useful in practice, so we recommend you configure the password (via `security.user.password`) and encrypt it (see below for instructions on how to do that).

Encryption and Decryption

Important	**Prerequisites: to use the encryption and decryption features you need the full-strength JCE installed in your JVM (it's not there by default). You can download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" from Oracle, and follow instructions for installation (essentially replace the 2 policy files in the JRE lib/security directory with the ones that you downloaded).**

If the remote property sources contain encryted content (values starting with `{cipher}`) they will be decrypted before sending to clients over HTTP. The main advantage of this set up is that the property values don't have to be in plain text when they are "at rest" (e.g. in a git repository). If a value cannot be decrypted it is replaced with an empty string, largely to prevent cipher text being used as a password and accidentally leaking.

If you are setting up a remote config repository for config client applications it might contain an `application.yml` like this, for instance:

application.yml

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

You can safely push this plain text to a shared git repository and the secret password is protected.

The server also exposes `/encrypt` and `/decrypt` endpoints (on the assumption that these will be secured and only accessed by authorized agents). If you are editing a remote config file you can use the Config Server to encrypt values by POSTing to the `/encrypt` endpoint, e.g.

```
$ curl localhost:8888/encrypt -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```

The inverse operation is also available via `/decrypt` (provided the server is configured with a symmetric key or a full key pair):

```
$ curl localhost:8888/decrypt -d 682bc583f4641835fa2db009355293665d
mysecret
```

Take the encypted value and add the `{cipher}` prefix before you put it in the YAML or properties file, and before you commit and push it to a remote, potentially insecure store. The `/encypt` and `/decrypt` endpoints also both accept paths of the form `/*/{name}/{profiles}` which can be used to control cryptography per application (name) and profile when clients call into the main Environment resource.

Note	**to control the cryptography in this granular way you must also provide a `@Bean` of type `TextEncryptorLocator` that creates a different encryptor per name and profiles. The one that is provided by default does not do this.**

The `spring` command line client (with Spring Cloud CLI extensions installed) can also be used to encrypt and decrypt, e.g.

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647da
mysecret
```

To use a key in a file (e.g. an RSA public key for encyption) prepend the key value with "@" and provide the file path, e.g.

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

The key argument is mandatory (despite having a `--` prefix).

Key Management

The Config Server can use a symmetric (shared) key or an asymmetric one (RSA key pair). The asymmetric choice is superior in terms of security, but it is often more convenient to use a symmetric key since it is just a single property value to configure.

To configure a symmetric key you just need to set `encrypt.key` to a secret String (or use an enviroment variable `ENCRYPT_KEY` to keep it out of plain text configuration files).

To configure an asymmetric key you can either set the key as a PEM-encoded text value (in `encrypt.key`), or via a keystore (e.g. as created by the `keytool` utility that comes with the JDK). The keystore properties are `encrypt.keyStore.*` with `*` equal to

- `location` (a `Resource` location),
- `password` (to unlock the keystore) and
- `alias` (to identify which key in the store is to be used).

The encryption is done with the public key, and a private key is needed for decryption. Thus in principle you can configure only the public key in the server if you only want to do encryption (and are prepared to decrypt the values yourself locally with the private key). In practice you might not want to do that because it spreads the key management process around all the clients, instead of concentrating it in the server. On the other hand it’s a useful option if your config server really is relatively insecure and only a handful of clients need the encrypted properties.

Creating a Key Store for Testing

To create a keystore for testing you can do something like this:

```
$ keytool -genkeypair -alias mytestkey -keyalg RSA \
  -dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US"
  -keypass changeme -keystore server.jks -storepass letmein
```

Put the `server.jks` file in the classpath (for instance) and then in your `application.yml` for the Config Server:

```
encrypt:
  keyStore:
    location: classpath:/server.jks
    password: letmein
    alias: mytestkey
    secret: changeme
```

Using Multiple Keys and Key Rotation

In addition to the `{cipher}` prefix in encrypted property values, the Config Server looks for `{name:value}` prefixes (zero or many) before the start of the (Base64 encoded) cipher text. The keys are passed to a `TextEncryptorLocator` which can do whatever logic it needs to locate a `TextEncryptor` for the cipher. If you have configured a keystore (`encrypt.keystore.location`) the default locator will look for keys in the store with aliases as supplied by the "key" prefix, i.e. with a cipher text like this:

```
foo:
  bar: `{cipher}{key:testkey}...`
```

the locator will look for a key named "testkey". A secret can also be supplied via a `{secret:…}` value in the prefix, but if it is not the default is to use the keystore password (which is what you get when you build a keytore and don't specify a secret). If you **do** supply a secret it is recommended that you also encrypt the secrets using a custom `SecretLocator`.

Key rotation is hardly ever necessary on cryptographic grounds if the keys are only being used to encrypt a few bytes of configuration data (i.e. they are not being used elsewhere), but occasionally you might need to change the keys if there is a security breach for instance. In that case all the clients would need to change their source config files (e.g. in git) and use a new `{key:…}` prefix in all the ciphers, checking beforehand of course that the key alias is available in the Config Server keystore.

Tip	**the `{name:value}` prefixes can also be added to plaintext posted to the `/encrypt` endpoint, if you want to let the Config Server handle all encryption as well as decryption.**

Embedding the Config Server

The Config Server runs best as a standalone application, but if you need to you can embed it in another application. Just use the `@EnableConfigServer` annotation and (optionally) set `spring.cloud.config.server.prefix` to a path prefix, e.g. "/config", to serve the resources under a prefix. The prefix should start but not end with a "/". It is applied to the `@RequestMappings` in the Config Server (i.e. underneath the Spring Boot prefixes `server.servletPath` and `server.contextPath`). Another optional property that can be useful in this case is `spring.cloud.config.server.bootstrap` which is a flag to indicate that the server should configure itself from its own remote repository. The flag is off by default because it can delay startup, but when embedded in another application it makes sense to initialize the same way as any other application.

undefined## Spring Cloud Config Client {#spring-cloud-config-client}

A Spring Boot application can take immediate advantage of the Spring Config Server (or other external property sources provided by the application developer), and it will also pick up some additional useful features related to `Environment` change events.

Config First Bootstrap

This is the default behaviour for any application which has the Spring Cloud Config Client on the classpath. When a config client starts up it binds to the Config Server (via the bootstrap configuration property `spring.cloud.config.uri`) and initializes Spring `Environment` with remote property sources.

The net result of this is that all client apps that want to consume the Config Server need a `bootstrap.yml` (or an environment variable) with the server address in `spring.cloud.config.uri` (defaults to "http://localhost:8888").

Eureka First Bootstrap

If you are using Spring Cloud Netflix and Eureka Service Discovery, then you can have the Config Server register with Eureka if you want to, but in the default "Config First" mode, clients won't be able to take advantage of the registration.

If you prefer to use Eureka to locate the Config Server, you can do that by setting `spring.cloud.config.discovery.enabled=true` (default "false"). The net result of that is that client apps all need a `bootstrap.yml` (or an environment variable) with the Eureka server address, e.g. in `eureka.client.serviceUrl.defaultZone` . The price for using this option is an extra network round trip on start up to locate the service registration. The benefit is that the Config Server can change its co-ordinates, as long as Eureka is a fixed point. The default service id is "CONFIGSERVER" but you can change that on the client with `spring.cloud.config.discovery.serviceId` (and on the server in the usual way for a service, e.g. by setting `spring.application.name`).

Config Client Fail Fast

In some cases, it may be desirable to fail startup of a service if it cannot connect to the Config Server. If this is the desired behavior, set the bootstrap configuration property `spring.cloud.config.failFast=true` and the client will halt with an Exception.

Config Client Retry

If you expect that the config server may occasionally be unavailable when your app starts, you can ask it to keep trying after a failure. First you need to set `spring.cloud.config.failFast=true`, and then you need to add `spring-retry` and `spring-boot-starter-aop` to your classpath. The default behaviour is to retry 6 times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) using `spring.config.retry.*` configuration properties.

Tip	**To take full control of the retry add a `@Bean` of type `RetryOperationsInterceptor` with id "configServerRetryInterceptor". Spring Retry has a `RetryInterceptorBuilder` that makes it easy to create one.**

Locating Remote Configuration Resources

The Config Service serves property sources from `/{name}/{profile}/{label}`, where the default bindings in the client app are

- "name" = `${spring.application.name}`
- "profile" = `${spring.profiles.active}` (actually `Environment.getActiveProfiles()`)
- "label" = "master"

All of them can be overridden by setting `spring.cloud.config.*` (where `*` is "name", "profile" or "label"). The "label" is useful for rolling back to previous versions of configuration; with the default Config Server implementation it can be a git label, branch name or commit id. Label can also be provided as a comma-separated list, in which case the items in the list are tried on-by-one until one succeeds. This can be useful when working on a feature branch, for instance, when you might want to align the config label with your branch, but make it optional (e.g. `spring.cloud.config.label=myfeature,develop`).

Security

If you use HTTP Basic security on the server then clients just need to know the password (and username if it isn't the default). You can do that via the config server URI, or via separate username and password properties, e.g.

bootstrap.yml

```
spring:
  cloud:
    config:
     uri: https://user:[[email protected]](/cdn-cgi/l/email-protect

**Illegal HTML tag removed :** /*  */
```

or

bootstrap.yml

```
spring:
  cloud:
    config:
     uri: https://myconfig.mycompany.com
     username: user
     password: secret
```

The `spring.cloud.config.password` and `spring.cloud.config.username` values override anything that is provided in the URI.

If you deploy your apps on Cloud Foundry then the best way to provide the password is through service credentials, e.g. in the URI, since then it doesn't even need to be in a config file. An example which works locally and for a user-provided service on Cloud Foundry named "configserver":

bootstrap.yml

```
spring:
  cloud:
    config:
     uri: ${vcap.services.configserver.credentials.uri:http://user:

**Illegal HTML tag removed :** /*  */

:8888}
```

If you use another form of security you might need to provide a `RestTemplate` to the `ConfigServicePropertySourceLocator` (e.g. by grabbing it in the bootstrap context and injecting one).

undefined# Spring Cloud Netflix {#spring-cloud-netflix}

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

undefined## Service Discovery: Eureka Clients {#service-discovery-eureka-clients}

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand configure each client or some form of convention can be very difficult to do and can be very brittle. Eureka is the Netflix Service Discovery Server and Client. The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others.

Registering with Eureka

When a client registers with Eureka, it provides meta-data about itself such as host and port, health indicator URL, home page etc. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

Example eureka client:

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
@EnableEurekaClient
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

(i.e. utterly normal Spring Boot app). In this example we use `@EnableEurekaClient` explicitly, but with only Eureka available you could also use `@EnableDiscoveryClient`. Configuration is required to locate the Eureka server. Example:

application.yml

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

where "defaultZone" is a magic string fallback value that provides the service URL for any client that doesn't express a preference (i.e. it's a useful default).

The default application name (service ID), virtual host and non-secure port, taken from the `Environment`, are `${spring.application.name}`, `${spring.application.name}` and `${server.port}` respectively.

`@EnableEurekaClient` makes the app into both a Eureka "instance" (i.e. it registers itself) and a "client" (i.e. it can query the registry to locate other services). The instance behaviour is driven by `eureka.instance.*` configuration keys, but the defaults will be fine if you ensure that your application has a `spring.application.name` (this is the default for the Eureka service ID, or VIP).

See EurekaInstanceConfigBean and EurekaClientConfigBean for more details of the configurable options.

Status Page and Health Indicator

The status page and health indicators for a Eureka instance default to "/info" and "/health" respectively, which are the default locations of useful endpoints in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (e.g. `server.servletPath=/foo`) or management endpoint path (e.g. `management.contextPath=/admin`). Example:

application.yml

```
eureka:
  instance:
    statusPageUrlPath: ${management.contextPath}/info
    healthCheckUrlPath: ${management.contextPath}/health
```

These links show up in the metadata that is consumers by clients, and used in some scenarios to decide whether to send requests to your application, so it's helpful if they are accurate.

Eureka Metadata for Instances and Clients

It's worth spending a bit of time understanding how the Eureka metadata works, so you can use it in a way that makes sense in your platform. There is standard metadata for things like hostname, IP address, port numbers, status page and health check. These are published in the service registry and used by clients to contact the services in a straightforward way. Additional metadata can be added to the instance registration in the

`eureka.instance.metadataMap`, and this will be accessible in the remote clients, but in general will not change the behaviour of the client, unless it is made aware of the meaning of the metadata. There are a couple of special cases described below where Spring Cloud already assigns meaning to the metadata map.

Using Eureka on Cloudfoundry

Cloudfoundry has a global router so that all instances of the same app have the same hostname (it's the same in other PaaS solutions with a similar architecture). This isn't necessarily a barrier to using Eureka, but if you use the router (recommended, or even mandatory depending on the way your platform was set up), you need to explicitly set the hostname and port numbers (secure or non-secure) so that they use the router. You might also want to use instance metadata so you can distinguish between the instances on the client (e.g. in a custom load balancer). For example:

application.yml

```
eureka:
  instance:
    hostname: ${vcap.application.uris[0]}
    nonSecurePort: 80
    metadataMap:
      instanceId: ${vcap.application.instance_id:${spring.applicati
```

Depending on the way the security rules are set up in your Cloudfoundry instance, you might be able to register and use the IP address of the host VM for direct service-to-service calls. This feature is not (yet) available on Pivotal Web Services (PWS).

Using Eureka on AWS

If the application is planned to be deployed to an AWS cloud, then the Eureka instance will have to be configured to be Amazon aware and this can be done by customizing the EurekaInstanceConfigBean the following way:

```
@Bean
@Profile("!default")
public EurekaInstanceConfigBean eurekaInstanceConfig() {
  EurekaInstanceConfigBean b = new EurekaInstanceConfigBean();
  AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
  b.setDataCenterInfo(info);
  return b;
}
```

Making the Eureka Instance ID Unique

By default a eureka instance is registered with an ID that is equal to its host name (i.e. only one service per host). Using Spring Cloud you can override this by providing a unique identifier in `eureka.instance.metadataMap.instanceId`. For example:

application.yml

```
eureka:
  instance:
    metadataMap:
      instanceId: ${spring.application.name}:${spring.application.i
```

With this metadata, and multiple service instances deployed on localhost, the random value will kick in there to make the instance unique. In Cloudfoundry the `spring.application.instance_id` will be populated automatically in a Spring Boot Actuator application, so the random value will not be needed.

Using the DiscoveryClient

Once you have an app that is `@EnableEurekaClient` you can use it to discover service instances from the Eureka Server. One way to do that is to use the native `com.netflix.discovery.DiscoveryClient` (as opposed to the Spring Cloud `DiscoveryClient`), e.g.

<pre>@Autowired private DiscoveryClient discoveryClient;

public String serviceUrl() { InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false); return instance.getHomePageUrl(); }</pre>

Tip	**Don’t use the `DiscoveryClient` in `@PostConstruct` method or in a `@Scheduled` method (or anywhere where the `ApplicationContext` might not be started yet). It is initialized in a `SmartLifecycle` (with `phase=0`) so the earliest you can rely on it being available is in another `SmartLifecycle` with higher phase.**

Alternatives to the native Netflix DiscoveryClient

You don’t have to use the raw Netflix `DiscoveryClient` and usually it is more convenient to use it behind a wrapper of some sort. Spring Cloud has support for Feign (a REST client builder) and also Spring `RestTemplate` using the logical Eureka service identifiers (VIPs) instead of physical URLs. To configure Ribbon with a fixed list of physical servers you can simply set `<client>.ribbon.listOfServers` to a comma-separated list of physical addresses (or hostnames), where `<client>` is the ID of the client.

You can also use the `org.springframework.cloud.client.discovery.DiscoveryClient` which provides a simple API for discovery clients that is not specific to Netflix, e.g.

<pre>@Autowired private DiscoveryClient discoveryClient;

public String serviceUrl() { List<ServiceInstance> list = client.getInstances("STORES"); if (list != null && list.size() > 0) { return list.get(0).getUri(); } return null; }</pre>

Why is it so Slow to Register a Service?

Being an instance also involves a periodic heartbeat to the registry (via the client’s `serviceUrl`) with default duration 30 seconds. A service is not available for discovery by clients until the instance, the server and the client all have the same metadata in their local cache (so it could take 3 hearbeats). You can change the period using `eureka.instance.leaseRenewalIntervalInSeconds` and this will speed up the process of getting clients connected to other services. In production it’s probably better to stick with the default because there are some computations internally in the server that make assumptions about the lease renewal period.

undefined## Service Discovery: Eureka Server {#service-discovery-eureka-server}

Example eureka server (e.g. using spring-cloud-starter-eureka-server to set up the classpath):

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

The server has a home page with a UI, and HTTP API endpoints per the normal Eureka functionality under `/eureka/*`.

Eureka background reading: see flux capacitor and google group discussion.

Tip	**Due to Gradle's dependency resolution rules and the lack of a parent bom feature, simply depending on spring-cloud-starter-eureka-server can cause failures on application startup. To remedy this the Spring dependency management plugin must be added and the Spring cloud starter parent bom must be imported like so:**

High Availability, Zones and Regions

The Eureka server does not have a backend store, but the service instances in the registry all have to send heartbeats to keep their registrations up to date (so this can be done in memory). Clients also have an in-memory cache of eureka registrations (so they don't have to go to the registry for every single request to a service).

By default every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you don't provide it the service will run and work, but it will shower your logs with a lot of noise about not being able to register with the peer.

See also below for details of Ribbon support on the client side for Zones and Regions.

Standalone Mode

The combination of the two caches (client and server) and the heartbeats make a standalone Eureka server fairly resilient to failure, as long as there is some sort of monitor or elastic runtime keeping it alive (e.g. Cloud Foundry). In standalone mode, you might prefer

to switch off the client side behaviour, so it doesn't keep trying and failing to reach its peers. Example:

application.yml (Standalone Eureka Server)

```
server:
  port: 8761

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

Notice that the `serviceUrl` is pointing to the same host as the local instance.

Peer Awareness

Eureka can be made even more resilient and available by running multiple instances and asking them to register with each other. In fact, this is the default behaviour, so all you need to do to make it work is add a valid `serviceUrl` to a peer, e.g.

application.yml (Two Peer Aware Eureka Servers)

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2/eureka/

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/
```

In this example we have a YAML file that can be used to run the same server on 2 hosts (peer1 and peer2), by running it in different Spring profiles. You could use this configuration to test the peer awareness on a single host (there's not much value in doing that in production) by manipulating `/etc/hosts` to resolve the host names. In fact, the `eureka.instance.hostname` is not needed if you are running on a machine that knows its own hostname (it is looked up using `java.net.InetAddress` by default).

You can add multiple peers to a system, and as long as they are all connected to each other by at least one edge, they will synchronize the registrations amongst themselves. If the peers are physically separated (inside a data centre or between multiple data centres) then the system can in principle survive split-brain type failures.

Prefer IP Address

In some cases, it is preferable for Eureka to advertise the IP Adresses of services rather than the hostname. Set `eureka.instance.preferIpAddress` to `true` and when the application registers with eureka, it will use its IP Address rather than its hostname.

undefined## Circuit Breaker: Hystrix Clients {#circuit-breaker-hystrix-clients}

Netflix has created a library called Hystrix that implements the circuit breaker pattern. In a microservice architecture it is common to have multiple layers of service calls.

 Figure 1. Microservice Graph

A service failure in the lower level of services can cause cascading failure all the way up to the user. When calls to a particular service reach a certain threshold (20 failures in 5 seconds is the default in Hystrix), the circuit opens and the call is not made. In cases of error and an open circuit a fallback can be provided by the developer.

 Figure 2. Hystrix fallback prevents cascading failures

Having an open circuit stops cascading failures and allows overwhelmed or failing services time to heal. The fallback can be another Hystrix protected call, static data or a sane empty value. Fallbacks may be chained so the first fallback makes some other business call which in turn falls back to static data.

Example boot app:

<pre>@SpringBootApplication @EnableCircuitBreaker public class Application {

```
public static void main(String[] args) {
    new SpringApplicationBuilder(Application.class).web(true).run(args);
}
```

}

@Component public class StoreIntegration {

```
@HystrixCommand(fallbackMethod = "defaultStores")
public Object getStores(Map<String, Object> parameters) {
    //do stuff that might fail
}

public Object defaultStores(Map<String, Object> parameters) {
    return /* something useful */;
}
```

}</pre>

The `@HystrixCommand` is provided by a Netflix contrib library called "javanica". Spring Cloud automatically wraps Spring beans with that annotation in a proxy that is connected to the Hystrix circuit breaker. The circuit breaker calculates when to open and close the circuit, and what to do in case of a failure.

To configure the `@HystrixCommand` you can use the `commandProperties` attribute with a list of `@HystrixProperty` annotations. See here for more details. See the Hystrix wiki for details on the properties available.

Propagating the Security Context or using Spring Scopes

If you want some thread local context to propagate into a `@HystrixCommand` the default declaration will not work because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller using some configuration, or directly in the annotation, by asking it to use a different "Isolation Strategy". For example:

```
@HystrixCommand(fallbackMethod = "stubMyService",
    commandProperties = {
      @HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")
    }
)
...
```

The same thing applies if you are using `@SessionScope` or `@RequestScope`. You will know when you need to do this because of a runtime exception that says it can't find the scoped context.

In particular you might be interested

Health Indicator

The state of the connected circuit breakers are also exposed in the `/health` endpoint of the calling application.

```
{
    "hystrix": {
        "openCircuitBreakers": [
            "StoreIntegration::getStoresByLocationLink"
        ],
        "status": "CIRCUIT_OPEN"
    },
    "status": "UP"
}
```

Hystrix Metrics Stream

To enable the Hystrix metrics stream include a dependency on `spring-boot-starter-actuator`. This will expose the `/hystrix.stream` as a management endpoint.

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

undefined## Circuit Breaker: Hystrix Dashboard {#circuit-breaker-hystrix-dashboard}

One of the main benefits of Hystrix is the set of metrics it gathers about each HystrixCommand. The Hystrix Dashboard displays the health of each circuit breaker in an efficient manner.

 Figure 3. Hystrix Dashboard

To run the Hystrix Dashboard annotate your Spring Boot main class with `@EnableHystrixDashboard`. You then visit `/hystrix` and point the dashboard to an individual instances `/hystrix.stream` endpoint in a Hystrix client application.

Turbine

Looking at an individual instances Hystrix data is not very useful in terms of the overall health of the system. Turbine is an application that aggregates all of the relevant `/hystrix.stream` endpoints into a combined `/turbine.stream` for use in the Hystrix Dashboard. Individual instances are located via Eureka. Running Turbine is as simple as annotating your main class with the `@EnableTurbine` annotation (e.g. using spring-cloud-starter-turbine to set up the classpath). All of the documented configuration properties from the Turbine 1 wiki) apply. The only difference is that the `turbine.instanceUrlSuffix` does not need the port prepended as this is handled automatically unless `turbine.instanceInsertPort=false`.

The configuration key `turbine.appConfig` is a list of eureka serviceIds that turbine will use to lookup instances. The turbine stream is then used in the Hystrix dashboard using a url that looks like: `[http://my.turbine.sever:8080/turbine.stream?cluster=&lt;CLUSTERNAME&gt;](http://my.turbine.sever:8080/turbine.stream?cluster=<CLUSTERNAME>");` (the cluster parameter can be omitted if the name is "default"). The `cluster` parameter must match an entry in `turbine.aggregator.clusterConfig`. Values returned from eureka are uppercase, thus we expect this example to work if there is an app registered with Eureka called "customers":

```
turbine:
  aggregator:
    clusterConfig: CUSTOMERS
  appConfig: customers
```

The `clusterName` can be customized by a SPEL expression in `turbine.clusterNameExpression` with root an instance of `InstanceInfo`. The default value is `appName`, which means that the Eureka serviceId ends up as the cluster key (i.e. the

`InstanceInfo` for customers has an `appName` of "CUSTOMERS"). A different example would be `turbine.clusterNameExpression=aSGName`, which would get the cluster name from the AWS ASG name. Another example:

```
turbine:
  aggregator:
    clusterConfig: SYSTEM,USER
  appConfig: customers,stores,ui,admin
  clusterNameExpression: metadata['cluster']
```

In this case, the cluster name from 4 services is pulled from their metadata map, and is expected to have values that include "SYSTEM" and "USER".

To use the "default" cluster for all apps you need a string literal expression (with single quotes):

```
turbine:
  appConfig: customers,stores
  clusterNameExpression: 'default'
```

Spring Cloud provides a `spring-cloud-starter-turbine` that has all the dependencies you need to get a Turbine server running. Just create a Spring Boot application and annotate it with `@EnableTurbine`.

Turbine AMQP

In some environments (e.g. in a PaaS setting), the classic Turbine model of pulling metrics from all the distributed Hystrix commands doesn't work. In that case you might want to have your Hystrix commands push metrics to Turbine, and Spring Cloud enables that with AMQP messaging. All you need to do on the client is add a dependency to `spring-cloud-netflix-hystrix-amqp` and make sure there is a Rabbit broker available (see Spring Boot documentation for details on how to configure the client credentials, but it should work out of the box for a local broker or in Cloud Foundry).

On the server side Just create a Spring Boot application and annotate it with `@EnableTurbineAmqp` and by default it will come up on port 8989 (point your Hystrix dashboard to that port, any path). You can customize the port using either `server.port` or `turbine.amqp.port`. If you have `spring-boot-starter-web` and `spring-boot-starter-actuator` on the classpath as well, then you can open up the Actuator endpoints on a separate port (with Tomcat by default) by providing a `management.port` which is different.

You can then point the Hystrix Dashboard to the Turbine AMQP Server instead of individual Hystrix streams. If Turbine AMQP is running on port 8989 on myhost, then put `[http://myhost:8989](http://myhost:8989)` in the stream input field in the Hystrix Dashboard. Circuits will be prefixed by their respective serviceId, followed by a dot, then the circuit name.

Spring Cloud provides a `spring-cloud-starter-turbine-amqp` that has all the dependencies you need to get a Turbine AMQP server running. You need Java 8 to run the app because it is Netty-based.

undefined## Customizing the AMQP ConnectionFactory {#customizing-the-amqp-connectionfactory}

If you are using AMQP there needs to be a `ConnectionFactory` (from Spring Rabbit) in the application context. If there is a single `ConnectionFactory` it will be used, or if there is a one qualified as `@HystrixConnectionFactory` (on the client) and `@TurbineConnectionFactory` (on the server) it will be preferred over others, otherwise the `@Primary` one will be used. If there are multiple unqualified connection factories there will be an error.

Note that Spring Boot (as of 1.2.2) creates a `ConnectionFactory` that is *not* `@Primary`, so if you want to use one connection factory for the bus and another for business messages, you need to create both, and annotate them `@*ConnectionFactory` and `@Primary` respectively.

undefined## Client Side Load Balancer: Ribbon {#client-side-load-balancer-ribbon}

Ribbon is a client side load balancer which gives you a lot of control over the behaviour of HTTP and TCP clients. Feign already uses Ribbon, so if you are using `@FeignClient` then this section also applies.

A central concept in Ribbon is that of the named client. Each load balancer is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer (e.g. using the `@FeignClient` annotation). Spring Cloud creates a new ensemble as an `ApplicationContext` on demand for each named client using `RibbonClientConfiguration`. This contains (amongst other things) an `ILoadBalancer`, a `RestClient`, and a `ServerListFilter`.

Customizing the Ribbon Client

You can configure some bits of a Ribbon client using external properties in `<client>.ribbon.*`, which is no different than using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in `CommonClientConfigKey` (part of ribbon-core).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the `RibbonClientConfiguration`) using `@RibbonClient`. Example:

```
@Configuration
@RibbonClient(name = "foo", configuration = FooConfiguration.class)
public class TestConfiguration {
}
```

In this case the client is composed from the components already in `RibbonClientConfiguration` together with any in `FooConfiguration` (where the latter generally will override the former).

Warning	**The `FooConfiguration` has to be `@Configuration` but take care that it is not in a `@ComponentScan` for the main application context, otherwise it will be shared by all the `@RibbonClients`. If you use `@ComponentScan` (or `@SpringBootApplication`) you need to take steps to avoid it being included (for instance put it in a separate, non-overlapping package, or specify the packages to scan explicitly in the `@ComponentScan`).**

Spring Cloud Netflix provides the following beans by default for ribbon (`BeanType` beanName: `ClassName`):

- `IClientConfig` ribbonClientConfig: `DefaultClientConfigImpl`

- `IRule` ribbonRule: `ZoneAvoidanceRule`
- `IPing` ribbonPing: `NoOpPing`
- `ServerList<Server> ribbonServerList:` ConfigurationBasedServerList`
- `ServerListFilter<Server>` ribbonServerListFilter: `ZonePreferenceServerListFilter`
- `ILoadBalancer` ribbonLoadBalancer: `ZoneAwareLoadBalancer`

Creating a bean of one of those type and placing it in a `@RibbonClient` configuration (such as `FooConfiguration` above) allows you to override each one of the beans described. Example:

```
@Configuration
public class FooConfiguration {
    @Bean
    public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
    }
}
```

This replaces the `NoOpPing` with `PingUrl`.

Using Ribbon with Eureka

When Eureka is used in conjunction with Ribbon the `ribbonServerList` is overridden with an extension of `DiscoveryEnabledNIWSServerList` which populates the list of servers from Eureka. It also replaces the `IPing` interface with `NIWSDiscoveryPing` which delegates to Eureka to determine if a server is up. The `ServerList` that is installed by default is a `DomainExtractingServerList` and the purpose of this is to make physical metadata available to the load balancer without using AWS AMI metadata (which is what Netflix relies on). By default the server list will be constructed with "zone" information as provided in the instance metadata (so on the client set `eureka.instance.metadataMap.zone`), and if that is missing it can use the domain name from the server hostname as a proxy for zone (if the flag `approximateZoneFromDomain` is set). Once the zone information is available it can be used in a `ServerListFilter` (by default it will be used to locate a server in the same zone as the client because the default is a `ZonePreferenceServerListFilter`).

Example: How to Use Ribbon Without Eureka

Eureka is a convenient way to abstract the discovery of remote servers so you don’t have to hard code their URLs in clients, but if you prefer not to use it, Ribbon and Feign are still quite amenable. Suppose you have declared a `@RibbonClient` for "stores", and Eureka is not in

use (and not even on the classpath). The Ribbon client defaults to a configured server list, and you can supply the configuration like this

application.yml

```
stores:
  ribbon:
    listOfServers: example.com,google.com
```

Example: Disable Eureka use in Ribbon

Setting the property `ribbon.eureka.enabled = false` will explicitly disable the use of Eureka in Ribbon.

application.yml

```
ribbon:
  eureka:
   enabled: false
```

Using the Ribbon API Directly

You can also use the `LoadBalancerClient` directly. Example:

```
public class MyClass {
    @Autowired
    private LoadBalancerClient loadBalancer;

    public void doStuff() {
        ServiceInstance instance = loadBalancer.choose("stores");
        URI storesUri = URI.create(String.format("http://%s:%s", instance.getHost(), inst
        // ... do something with the URI
    }
}
```

undefined## Declarative REST Client: Feign {#declarative-rest-client-feign}

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same `HttpMessageConverters` used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

Example spring boot app

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
@EnableEurekaClient
@EnableFeignClients
public class Application {

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

}
```

StoreClient.java

```
@FeignClient("stores")
public interface StoreClient {
    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes =
    Store update(@PathParameter("storeId") Long storeId, Store store);
}
```

In the `@FeignClient` annotation the String value ("stores" above) is an arbitrary client name, which is used to create a Ribbon load balancer (see below for details of Ribbon support). You can also specify a URL using the `url` attribute (absolute value or just a hostname).

The Ribbon client above will want to discover the physical addresses for the "stores" service. If your application is a Eureka client then it will resolve the service in the Eureka service registry. If you don’t want to use Eureka, you can simply configure a list of servers in your external configuration (see above for example).

undefined## External Configuration: Archaius {#external-configuration-archaius}

Archaius is the Netflix client side configuration library. It is the library used by all of the Netflix OSS components for configuration. Archaius is an extension of the Apache Commons Configuration project. It allows updates to configuration by either polling a source for changes or for a source to push changes to the client. Archaius uses Dynamic<Type>Property classes as handles to properties.

Archaius Example

```
class ArchaiusTest {
    DynamicStringProperty myprop = DynamicPropertyFactory
            .getInstance()
            .getStringProperty("my.prop");

    void doSomething() {
        OtherClass.someMethod(myprop.get());
    }
}
```

Archaius has its own set of configuration files and loading priorities. Spring applications should generally not use Archaius directly., but the need to configure the Netflix tools natively remains. Spring Cloud has a Spring Environment Bridge so Archaius can read properties from the Spring Environment. This allows Spring Boot projects to use the normal configuration toolchain, while allowing them to configure the Netflix tools, for the most part, as documented.

undefined## Router and Filter: Zuul {#router-and-filter-zuul}

Routing in an integral part of a microservice architecture. For example, `/` may be mapped to your web application, `/api/users` is mapped to the user service and `/api/shop` is mapped to the shop service. Zuul is a JVM based router and server side load balancer by Netflix.

Netflix uses Zuul for the following:

- Authentication
- Insights
- Stress Testing
- Canary Testing
- Dynamic Routing
- Service Migration
- Load Shedding
- Security
- Static Response handling
- Active/Active traffic management

Zuul's rule engine allows rules and filters to be written in essentially any JVM language, with built in support for Java and Groovy.

Embedded Zuul Reverse Proxy

Spring Cloud has created an embedded Zuul proxy to ease the development of a very common use case where a UI application wants to proxy calls to one or more back end services. This feature is useful for a user interface to proxy to the backend services it requires, avoiding the need to manage CORS and authentication concerns independently for all the backends.

To enable it, annotate a Spring Boot main class with `@EnableZuulProxy`, and this forwards local calls to the appropriate service. By convention, a service with the Eureka ID "users", will receive requests from the proxy located at `/users` (with the prefix stripped). The proxy uses Ribbon to locate an instance to forward to via Eureka, and all requests are executed in a hystrix command, so failures will show up in Hystrix metrics, and once the circuit is open the proxy will not try to contact the service.

To skip having a service automatically added, set `zuul.ignored-services` to a list of service id patterns. If a service matches a pattern that is ignored, but also included in the explicitly configured routes map, then it will be unignored. Example:

application.yml

```
zuul:
 ignoredServices: *
 routes:
   users: /myusers/**
```

In this example, all services are ignored **except** "users".

To augment or change the proxy routes, you can add external configuration like the following:

application.yml

```
zuul:
 routes:
   users: /myusers/**
```

This means that http calls to "/myusers" get forwarded to the "users" service (for example "/myusers/101" is forwarded to "/101").

To get more fine-grained control over a route you can specify the path and the serviceId independently:

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     serviceId: users_service
```

This means that http calls to "/myusers" get forwarded to the "users_service" service. The route has to have a "path" which can be specified as an ant-style pattern, so "/myusers/*" *only matches one level, but "/myusers/**" matches hierarchically.

The location of the backend can be specified as either a "serviceId" (for a Eureka service) or a "url" (for a physical location), e.g.

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     url: http://example.com/users_service
```

These simple url-routes doesn't get executed as HystrixCommand nor can you loadbalance multiple url with Ribbon. To achieve this specify a service-route and configure a Ribbon client for the serviceId (this currently requires disabling Eureka support in Ribbon: see above for more information), e.g.

application.yml

```
zuul:
  routes:
    users:
      path: /myusers/**
      serviceId: users

ribbon:
  eureka:
    enabled: false

users:
  ribbon:
    listOfServers: example.com,google.com
```

To add a prefix to all mappings, set `zuul.prefix` to a value, such as `/api`. The proxy prefix is stripped from the request before the request is forwarded by default (switch this behaviour off with `zuul.stripPrefix=false`). You can also switch off the stripping of the service-specific prefix from individual routes, e.g.

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     stripPrefix: false
```

In this example requests to "/myusers/101" will be forwarded to "/myusers/101" on the "users" service.

The `zuul.routes` entries actually bind to an object of type `ProxyRouteLocator`. If you look at the properties of that object you will see that it also has a "retryable" flag. Set that flag to "true" to have the Ribbon client automatically retry failed requests (and if you need to you can modify the parameters of the retry operations using the Ribbon client configuration).

The `X-Forwarded-Host` header added to the forwarded requests by default. To turn it off set `zuul.addProxyHeaders = false`. The prefix path is stripped by default, and the request to the backend picks up a header "X-Forwarded-Prefix" ("/myusers" in the examples above).

An application with the `@EnableZuulProxy` could act as a standalone server if you set a default route ("/"), for example `zuul.route.home: /` would route all traffic (i.e. "/**") to the "home" service.

Uploading Files through Zuul

If you `@EnableZuulProxy` you can use the proxy paths to upload files and it should just work as long as the files are small. For large files there is an alternative path which bypasses the Spring `DispatcherServlet` (to avoid multipart processing) in "/zuul/**". I.e. if `zuul.routes.customers=/customers/*` then you can POST large files to "/zuul/customers/*". The servlet path is externalized via `zuul.servletPath`. Extremely large files will also require elevated timeout settings if the proxy route takes you through a Ribbon load balancer, e.g.

application.yml

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

Note that for streaming to work with large files, you need to use chunked encoding in the request (which some browsers do not do by default). E.g. on the command line:

```
$ curl -v -H "Transfer-Encoding: chunked" \
    -F "[[email protected]](/cdn-cgi/l/email-protection)

**Illegal HTML tag removed :** /*  */

" localhost:9999/zuul/simple/file
```

Plain Embedded Zuul

You can also run a Zuul server without the proxying, or switch on parts of the proxying platform selectively, if you use `@EnableZuulServer` (instead of `@EnableZuulProxy`). Any beans that you add to the application of type `ZuulFilter` will be installed automatically, as they are with `@EnableZuulProxy`, but without any of the proxy filters being added automatically.

In this case the routes into the Zuul server are still specified by configuring "zuul.routes.*", but there is no service discovery and no proxying, so the "serviceId" and "url" settings are ignored. For example:

application.yml

```
zuul:
 routes:
   api: /api/**
```

maps all paths in "/api/**" to the Zuul filter chain.

Disable Zuul Filters

Zuul for Spring Cloud comes with a number of `ZuulFilter` beans enabled by default in both proxy and server mode. See the zuul filters package for the possible filters that are enabled. If you want to disable one, simply set `zuul.<SimpleClassName>.<filterType>.disable=true`. By convention, the package after `filters` is the Zuul filter type. For example to disable `org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter` set `zuul.SendResponseFilter.post.disable=true`.

Polyglot support with Sidecar

Do you have non-jvm languages you want to take advantage of Eureka, Ribbon and Config Server? The Spring Cloud Netflix Sidecar was inspired by Netflix Prana. It includes a simple http api to get all of the instances (ie host and port) for a given service. You can also proxy service calls through an embedded Zuul proxy which gets its route entries from Eureka. The Spring Cloud Config Server can be accessed directly via host lookup or through the Zuul Proxy. The non-jvm app should implement a health check so the Sidecar can report to eureka if the app is up or down.

To enable the Sidecar, create a Spring Boot application with `@EnableSidecar`. This annotation includes `@EnableCircuitBreaker`, `@EnableDiscoveryClient`, and `@EnableZuulProxy`. Run the resulting application on the same host as the non-jvm application.

To configure the side car add `sidecar.port` and `sidecar.health-uri` to `application.yml`. The `sidecar.port` property is the port the non-jvm app is listening on. This is so the Sidecar can properly register the app with Eureka. The `sidecar.health-uri` is a uri accessible on the non-jvm app that mimicks a Spring Boot health indicator. It should return a json document like the following:

health-uri-document

```
{
  "status":"UP"
}
```

Here is an example application.yml for a Sidecar application:

application.yml

```
server:
  port: 5678
spring:
  application:
    name: sidecar

sidecar:
  port: 8000
  health-uri: http://localhost:8000/health.json
```

The api for the `DiscoveryClient.getInstances()` method is `/hosts/{serviceId}`. Here is an example response for `/hosts/customers` that returns two instances on different hosts. This api is accessible to the non-jvm app (if the sidecar is on port 5678) at `[http://localhost:5678/hosts/{serviceId}](http://localhost:5678/hosts/{serviceId})`.

/hosts/customers

```
[
    {
        "host": "myhost",
        "port": 9000,
        "uri": "http://myhost:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    },
    {
        "host": "myhost2",
        "port": 9000,
        "uri": "http://myhost2:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    }
]
```

The Zuul proxy automatically adds routes for each service known in eureka to `/&lt;serviceId&gt;`, so the customers service is available at `/customers`. The Non-jvm app can access the customer service via `[http://localhost:5678/customers](http://localhost:5678/customers)` (assuming the sidecar is listening on port 5678).

If the Config Server is registered with Eureka, non-jvm application can access it via the Zuul proxy. If the serviceId of the ConfigServer is `configserver` and the Sidecar is on port 5678, then it can be accessed at http://localhost:5678/configserver

Non-jvm app can take advantage of the Config Server's ability to return YAML documents. For example, a call to http://sidecar.local.spring.io:5678/configserver/default-master.yml might result in a YAML document like the following

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  password: password
info:
  description: Spring Cloud Samples
  url: https://github.com/spring-cloud-samples
```

undefined# Spring Cloud Bus {#spring-cloud-bus}

Spring Cloud Bus links nodes of a distributed system with a lightweight message broker. This can then be used to broadcast state changes (e.g. configuration changes) or other management instructions. A key idea is that the Bus is like a distributed Actuator for a Spring Boot application that is scaled out, but it can also be used as a communication channel between apps. The only implementation currently is with an AMQP broker as the transport, but the same basic feature set (and some more depending on the transport) is on the roadmap for other transports.

Spring Cloud Bus联系分布式系统节点通过轻量级的消息代理。这样可以用来广播状态变化（如：配置改变）或者其他管理的指令。一个关键的主意是总线就像分布式执行器作为spring boot 应用的扩展，同时也可以用于多个app之间通讯。当前仅实现AMQP服务器作为消息传输，但是其他的消息服务已经准备在支持了，相同特性开发路线。

| Note | Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at {githubmaster}/docs/src/main/asciidoc[github]. | | --- | --- |

undefined## Quick Start {#quick-start}

Spring Cloud Bus works by adding Spring Boot autconfiguration if it detects itself on the classpath. All you need to do to enable the bus is to add `spring-cloud-starter-bus-amqp` to your dependency management and Spring Cloud takes care of the rest. Make sure RabbitMQ is available and configured to provide a `ConnectionFactory`: running on localhost you shouldn't have to do anything, but if you are running remotely use Spring Cloud Connectors, or Spring Boot conventions to define the broker credentials, e.g.

application.yml

```
spring:
  rabbitmq:
    host: mybroker.com
    port: 5672
    username: user
    password: secret
```

The bus currently supports sending messages to all nodes listening or all nodes for a particular service (as defined by Eureka). More selector criteria will be added in the future (ie. only service X nodes in data center Y, etc…). The http endpoints are under the `/bus/*` actuator namespace. There are currently two implemented. The first, `/bus/env`, sends key/values pairs to update each nodes Spring Environment. The second, `/bus/refresh`, will reload each application's configuration, just as if they had all been pinged on their `/refresh` endpoint.

undefined## Addressing an Instance {#addressing-an-instance}

The HTTP endpoints accept a "destination" parameter, e.g. "/bus/refresh?destination=customers:9000", where the destination is an `ApplicationContext` ID. If the ID is owned by an instance on the Bus then it will process the message and all other instances will ignore it. Spring Boot sets the ID for you in the `ContextIdApplicationContextInitializer` to a combination of the `spring.application.name`, active profiles and `server.port` by default.

undefined## Addressing all instances of a service {#addressing-all-instances-of-a-service}

The "destination" parameter is used in a Spring `PathMatcher` (with the path separator as a colon `:`) to determine if an instance will process the message. Using the example from above, "/bus/refresh?destination=customers:**" will target all instances of the "customers" service regardless of the profiles and ports set as the `ApplicationContext` ID.

undefined## Application Context ID must be unique {#application-context-id-must-be-unique}

The bus tries to eliminate processing an event twice, once from the original `ApplicationEvent` and once from the queue. To do this, it checks the sending application context id againts the current application context id. If multiple instances of a service have the same application context id, events will not be processed. Running on a local machine, each service will be on a different port and that will be part of the application context id. Cloud Foundry supplies an index to differentiate. To ensure that the application context id is the unique, set `spring.application.index` to something unique for each instance of a service. For example, in lattice, set `spring.application.index=${INSTANCE_INDEX}` in application.properties (or bootstrap.properties if using configserver).

undefined## Customizing the AMQP ConnectionFactory {#customizing-the-amqp-connectionfactory}

If you are using AMQP there needs to be a `ConnectionFactory` (from Spring Rabbit) in the application context. If there is a single `ConnectionFactory` it will be used, or if there is a one qualified as `@BusConnectionFactory` it will be preferred over others, otherwise the `@Primary` one will be used. If there are multiple unqualified connection factories there will be an error.

Note that Spring Boot (as of 1.2.2) creates a `ConnectionFactory` that is *not* `@Primary`, so if you want to use one connection factory for the bus and another for business messages, you need to create both, and annotate them `@BusConnectionFactory` and `@Primary` respectively.

undefined# Spring Boot Cloud CLI {#spring-boot-cloud-cli}

Spring Boot CLI provides Spring Boot command line features for Spring Cloud. You can write Groovy scripts to run Spring Cloud component applications (e.g. `@EnableEurekaServer`). You can also easily do things like encryption and decryption to support Spring Cloud Config clients with secret configuration values.

| Note | Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at {githubmaster}/docs/src/main/asciidoc[github]. | | --- | --- |

undefined## Installation {#installation}

To install, make sure you have Spring Boot CLI (1.2.0 or better):

```
$ spring version
Spring CLI v1.2.3.RELEASE
```

E.g. for GVM users

```
$ gvm install springboot 1.2.3.RELEASE
$ gvm use springboot 1.2.3.RELEASE
```

and install the Spring Cloud plugin:

```
$ mvn install
$ spring install org.springframework.cloud:spring-cloud-cli:1.0.2.RELEASE
```

| Important | **Prerequisites:** to use the encryption and decryption features you need the full-strength JCE installed in your JVM (it’s not there by default). You can download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" from Oracle, and follow instructions for installation (essentially replace the 2 policy files in the JRE lib/security directory with the ones that you downloaded). | | --- | --- |

undefined## Writing Groovy Scripts and Running Applications {#writing-groovy-scripts-and-running-applications}

Spring Cloud CLI has support for most of the Spring Cloud declarative features, such as the `@Enable*` class of annotations. For example, here is a fully functional Eureka server

app.groovy

```
@EnableEurekaServer
class Eureka {}
```

which you can run from the command line like this

```
$ spring run app.groovy
```

To include additional dependencies, often it suffices just to add the appropriate feature-enabling annotation, e.g. `@EnableConfigServer`, `@EnableOAuth2Sso` or `@EnableEurekaClient`. To manually include a dependency you can use a `@Grab` with the special "Spring Boot" short style artifact co-ordinates, i.e. with just the artifact ID (no need for group or version information), e.g. to set up a client app to listen on AMQP for management events from the Spring CLoud Bus:

app.groovy

```
@Grab('spring-cloud-starter-bus-amqp')
@RestController
class Service {
  @RequestMapping('/')
  def home() { [message: 'Hello'] }
}
```

undefined## Encryption and Decryption {#encryption-and-decryption}

The Spring Cloud CLI comes with an "encrypt" and a "decrypt" command. Both accept arguments in the same form with a key specified as a mandatory "--key", e.g.

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647da
mysecret
```

To use a key in a file (e.g. an RSA public key for encyption) prepend the key value with "@" and provide the file path, e.g.

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

undefined# Spring Cloud Security {#spring-cloud-security}

Spring Cloud Security offers a set of primitives for building secure applications and services with minimum fuss. A declarative model which can be heavily configured externally (or centrally) lends itself to the implementation of large systems of co-operating, remote components, usually with a central indentity management service. It is also extremely easy to use in a service platform like Cloud Foundry. Building on Spring Boot and Spring Security OAuth2 we can quickly create systems that implement common patterns like single sign on, token relay and token exchange.

| Note | Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at {githubmaster}/src/main/asciidoc[github]. | | --- | --- |

undefined## Quickstart {#quickstart}

OAuth2 Single Sign On

Here’s a Spring Cloud "Hello World" app with HTTP Basic authentication and a single user account:

app.groovy

```
@Grab('spring-boot-starter-security')
@Controller
class Application {

  @RequestMapping('/')
  String home() {
    'Hello World'
  }

}
```

You can run it with `spring run app.groovy` and watch the logs for the password (username is "user"). So far this is just the default for a Spring Boot app.

Here’s a Spring Cloud app with OAuth2 SSO:

app.groovy

```
@Controller
@EnableOAuth2Sso
class Application {

  @RequestMapping('/')
  String home() {
    'Hello World'
  }

}
```

Spot the difference? This app will actually behave exactly the same as the previous one, because it doesn’t know it’s OAuth2 credentals yet.

You can register an app in github quite easily, so try that if you want a production app on your own domain. If you are happy to test on localhost:8080, then set up these properties in your application configuration:

application.yml

```
spring:
  oauth2:
    client:
      clientId: bd1c0a783ccdd1c9b9e4
      clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
      accessTokenUri: https://github.com/login/oauth/access_token
      userAuthorizationUri: https://github.com/login/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

run the app above and it will redirect to github for authorization. If you are already signed into github you won't even notice that it has authenticated. These credentials will only work if your app is running on port 8080.

To limit the scope that the client asks for when it obtains an access token you can set `spring.oauth2.client.scope` (comma separated or an array in YAML). By default the scope is empty and it is up to to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.

Note	**The examples above are all Groovy scripts. If you want to write the same code in Java (or Groovy) you need to add Spring Security OAuth2 to the classpath (e.g. see the sample here).**

OAuth2 Protected Resource

You want to protect an API resource with an OAuth2 token? Here's a simple example (paired with the client above):

app.groovy

```
@Grab('spring-cloud-starter-security')
@RestController
@EnableOAuth2Resource
class Application {

  @RequestMapping('/')
  def home() {
    [message: 'Hello World']
  }

}
```

and

application.yml

```
spring:
  oauth2:
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

undefined## More Detail {#more-detail}

Single Sign On

An app will activate `@EnableOAuth2Sso` if you bind provide the following properties in the `Environment`:

- `spring.oauth2.client.*` with `*` equal to `clientId`, `clientSecret`, `accessTokenUri`, `userAuthorizationUri` and one of:
  - `spring.oauth2.resource.userInfoUri` to use the "/me" resource (e.g. "https://uaa.run.pivotal.io/userinfo" on PWS), or
  - `spring.oauth2.resource.tokenInfoUri` to use the token decoding endpoint (e.g. "https://uaa.run.pivotal.io/check_token" on PWS).

  If you specify both the `userInfoUri` and the `tokenInfoUri` then you can set a flag to say that one is preferred over the other (`preferTokenInfo=true` is the default). Or
- `spring.oauth2.resource.jwt.keyValue` to decode a JWT token locally, where the key is a verification key. The verification key value is either a symmetric secret or PEM-encoded RSA public key. If you don't have the key and it's public you can provide a URI where it can be downloaded (as a JSON object with a "value" field) with `spring.oauth2.resource.jwt.keyUri`. E.g. on PWS:

```
$ curl https://uaa.run.pivotal.io/token_key
{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMII
```

Warning	**If you use the `spring.oauth2.resource.jwt.keyUri` the authorization server needs to be running when your application starts up. It will log a warning if it can't find the key, and tell you what to do to fix it.**

You can set the preferred scope (as a comma-separated list or YAML array) in `spring.oauth2.client.scope`. It defaults to empty, in which case most Authorization Servers will ask the user for approval for the maximum allowed scope for the client.

There is also a setting for `spring.oauth2.client.clientAuthenticationScheme` which defaults to "header" (but you might need to set it to "form" if, like Github for instance, your OAuth2 provider doesn't like header authentication). The `spring.oauth2.client.*` properties are bound to an instance of `AuthorizationCodeResourceDetails` so all its properties can be specified.

Token Type in User Info

Google (and certain other 3rd party identity providers) is more strict about the token type name that is sent in the headers to the user info endpoint. The default is "Bearer" which suits most providers and matches the spec, but if you need to change it you can set `spring.oauth2.resource.tokenType`.

Customizing the RestTemplate

The SSO (and Resource Server) features use an `OAuth2RestTemplate` internally to fetch user details for authentication. This is provided as a qualified `@Bean` with id "userInfoRestTemplate", but you shouldn't need to know that to just use it. The default should be fine for most providers, but occasionally you might need to add additional interceptors, or change the request authenticator (which is how the token gets attached to outgoing requests). To add a customization just create a bean of type `UserInfoRestTemplateCustomizer` - it has a single method that will be called after the bean is created but before it is initialized. The rest template that is being customized here is *only* used internally to carry out authentication (in the SSO or Resource Server use cases).

A second **Illegal HTML tag removed :** / / OAuth2RestTemplate} is available for autowiring if you want to use it for back channel calls, and if there is a token-authenticated user (in a web application) it will have the token injected for you.

Tip	To set an RSA key value in YAML use the "pipe" continuation marker to split it over multiple lines ("	") and remember to indent the key value (it's a standard YAML language feature). Example:

Access Decision Rules

By default the whole application will be secured with OAuth2 with the same access rule ("authenticated"). This includes the Actuator endpoints, which you might prefer to be secured differently, so Spring Cloud Security provides a configurer callback that lets you change the matching and access rules for OAuth2 authentication. Any bean of type `OAuth2SsoConfigurer` (there is a convenient empty base class) will get 2 callbacks, one to set the request matchers for the OAuth2 filter, and one with the full `HttpSecurity` builder (so you can set up all sorts of behaviour, but the main application is to control access rules).

The default login path, i.e. the one that triggers the redirect to the OAuth2 Authorization Server, is "/login". It will always be added to the matching patterns for the OAuth2 SSO, even if you have `OAuth2SsoConfigurer` beans as well. The default logout path is "/logout"

and it gets similar treatment, as does the "home" page (which is the logout success page, defaults to "/"). Those paths can be overriden by setting `spring.oauth2.sso.*' ( loginPath , logoutPath and home.path`).

For example if you want the resources under "/ui/**" to be protected with OAuth2:

```
@Configuration
@EnableOAuth2Sso
@EnableAutoConfiguration
protected static class TestConfiguration extends OAuth2SsoConfigurerAdapter {
    @Override
    public void match(RequestMatchers matchers) {
        matchers.antMatchers("/ui/**");
    }
}
```

In this case the rest of the application will default to the normal Spring Boot access control (Basic authentication, or whatever custom filters you put in place).

Integrating with the Actuator Endpoints

The Spring Boot Actuator endpoints ("/env", "/metrics", etc.) if present will, by default, be protected by the standard Spring Boot basic authentication. The SSO authentication filter is added in a position directly behind the filter that intercepts requests to the Actuator endpoints by default (i.e. `ManagementProperties.BASIC_AUTH_ORDER + 1` which is `Ordered.LOWEST_PRECEDENCE-9` or `2147483636`). If you want to change the order you can set `spring.oauth2.sso.filterOrder` . If you do that and the value is less than the default, then you will need to consider setting the access rules for the Actuator, since they will become accessible to all authenticated users who sign on with the external provider. One way to do that would be to set `management.contextPath=/admin` (for instance) and use an `OAuth2SsoConfigurer` to set the access rules, e.g.

```
    @Configuration
    @EnableOAuth2Sso
    @EnableAutoConfiguration
    protected static class TestConfiguration extends OAuth2SsoConfigurerAdapter {
        @Override
        public void configure(HttpSecurity http) {
             http.authorizeRequests()
                 .antMatchers("/admin/**").role("ADMIN")
                 .anyRequest().authenticated();
        }
    }
```

Resource Server

The `@EnableOAuth2Resource` annotation will protect your API endpoints if you have the same environment settings as the SSO client, except that it doesn't need a `tokenUri` or `authorizationUri`, and it also doesn't need a `clientId` and `clientSecret` if it isn't using the `tokenInfoUri` (i.e. if it has `jwt.*` or `userInfoUri`).

By default **all** your endpoints are protected (i.e. "/**") but you can pick and choose by adding a `ResourceServerConfigurerAdapter` (standard Spring OAuth feature), e.g. to protect only the "/api/" resources

Application.java

```
@RestController
@EnableOAuth2Resource
class Application extends ResourceServerConfigurerAdapter {

  @Override
  public void configure(HttpSecurity http) throws Exception {
    http.requestMatchers()
      .antMatchers("/api/**")
   .and()
     .authorizeRequests()
       .anyRequest().authenticated();
  }

  @RequestMapping("/api")
  public String home() {
    return "Hello World";
  }

}
```

Customizing the JWT Token Converter

When a resource server accepts an access token as a JWT, it has to convert it to an `Authentication` so that Spring Security can do its access decisions. Different token providers might support JWT tokens with different contents, so Spring OAuth2 has an abstraction for converting the token into security domain objects (`AccessTokenConverter`). You can modify the default behaviour easily by providing a `@Bean` of type `JwtAccessTokenConverterConfigurer`, e.g.

```
@Component
public class JwtCustomization extends DefaultAccessTokenConverter implements
        JwtAccessTokenConverterConfigurer {

    @Override
    public void configure(JwtAccessTokenConverter converter) {
        converter.setAccessTokenConverter(this);
    }

    ... // implement custom AccessTokenConverter here

}
```

Token Relay

A Token Relay is where an OAuth2 consumer acts as a Client and forwards the incoming token to outgoing resource requests. The consumer can be a pure Client (like an SSO application) or a Resource Server.

Client Token Relay

If your app has a Spring Cloud Zuul embedded reverse proxy (using `@EnableZuulProxy`) then you can ask it to forward OAuth2 access tokens downstream to the services it is proxying. Thus the SSO app above can be enhanced simply like this:

app.groovy

```
@Controller
@EnableOAuth2Sso
@EnableZuulProxy
class Application {

}
```

and it will (in addition to loggin the user in and grabbing a token) pass the authentication token downstream to the `/proxy/*` services. If those services are implemented with `@EnableOAuth2Resource` then they will get a valid token in the correct header.

How does it work? The `@EnableOAuth2Sso` annotation pulls in `spring-cloud-starter-security` (which you could do manually in a traditional app), and that in turn triggers some autoconfiguration for a `ZuulFilter`, which itself is activated because Zuul is on the classpath (via `@EnableZuulProxy`). The

{github}/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/OAuth2TokenRelayFilter.java[filter] just extracts an access token from the currently authenticated user, and puts it in a request header for the downstream requests.

Resource Server Token Relay

If your app has `@EnableOAuth2Resource` and also is a Client (i.e. it has a `spring.oauth2.client.clientId`, even if it doesn't use it), then the `OAuth2RestOperations` that is provided for `@Autowired` users by Spring Cloud (it is declared as `@Primary`) will also forward tokens. If you don't want to forward tokens (and that is a valid choice, since you might want to act as yourself, rather than the client that sent you the token), then you only need to create your own `OAuth2RestOperations` instead of autowiring the default one. Here's a basic example showing the use of the autowired rest template ("foo.com" is a Resource Server accepting the same tokens as the surrounding app):

MyController.java

```
@Autowired
private OAuth2RestOperations restTemplate;

@RequestMapping("/relay")
public String relay() {
    ResponseEntity<String> response =
      restTemplate.getForEntity("https://foo.com/bar", String.class);
    return "Success! (" + response.getBody() + ")";
}
```

undefined## Configuring Authentication Downstream of a Zuul Proxy {#configuring-authentication-downstream-of-a-zuul-proxy}

You can control the authorization behaviour downstream of an `@EnableZuulProxy` through the `proxy.auth.*` settings. Example:

application.yml

```
proxy:
  auth:
    routes:
      customers: oauth2
      stores: passthru
      recommendations: none
```

In this example the "customers" service gets an OAuth2 token relay, the "stores" service gets a passthrough (the authorization header is just passed downstream), and the "recommendations" service has its authorization header removed. The default behaviour is to do a token relay if there is a token available, and passthru otherwise.

See {github}/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/ProxyAuthenticationProperties[ProxyAuthenticationProperties] for full details.

Last updated 2016-02-25 11:46:04 UTC

Illegal HTML tag removed : / /