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). 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.

Version: Dalston.SR3

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
- Distributed messaging

Cloud Native Applications

Cloud Native (https://pivotal.io/platform-as-a-service/migrating-to-cloud-native-application-architectures-ebook) 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 (http://12factor.net/) 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 (https://projects.spring.io/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 (http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html)
- Java 7 JCE (http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html)
- Java 8 JCE (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 github (https://github.com/spring-cloud/spring-cloud-commons/tree/master/docs/src/main/asciidoc).

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, by default.

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.

Overriding the Values of Remote Properties

The property sources that are added to you application by the bootstrap context are often "remote" (e.g. from a Config Server), and by default they cannot be overridden locally, except on the command line. If you want to allow your applications to override the remote properties with their own System properties or config files, the remote property source has to grant it permission by setting `spring.cloud.config.allowOverride=true` (it doesn't work to set this locally). Once that flag is set there are some finer grained settings to control the location of the remote properties in relation to System properties and the application's local configuration: `spring.cloud.config.overrideNone=true` to override with any local property source, and `spring.cloud.config.overrideSystemProperties=false` if only System properties and env vars should override the remote settings, but not the local config files.

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.source", "worked as intended"));
    }

}
```

JAVA

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.custom.CustomPropertySourceLocator
```

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 `EnvironmentChangeEvent` 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 `EnvironmentChangeEvent` 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 `EnvironmentChangeEvent` to all the instances instead of having them polling for changes (e.g. using the Spring Cloud Bus (https://github.com/spring-cloud/spring-cloud-bus)).

The `EnvironmentChangeEvent` 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

Spring Cloud 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 an application 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.

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

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`)

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).

@EnableDiscoveryClient

Commons provides the `@EnableDiscoveryClient` annotation. This looks for implementations of the `DiscoveryClient` interface via `META-INF/spring.factories`. Implementations of Discovery Client will add a configuration class to `spring.factories` under the `org.springframework.cloud.client.discovery.EnableDiscoveryClient` key. Examples of `DiscoveryClient` implementations: are Spring Cloud Netflix Eureka (https://cloud.spring.io/spring-cloud-netflix/), Spring Cloud Consul Discovery (https://cloud.spring.io/spring-cloud-consul/) and Spring Cloud Zookeeper Discovery (https://cloud.spring.io/spring-cloud-zookeeper/).

By default, implementations of `DiscoveryClient` will auto-register the local Spring Boot server with the remote discovery server. This can be disabled by setting `autoRegister=false` in `@EnableDiscoveryClient`.

ServiceRegistry

Commons now provides a `ServiceRegistry` interface which provides methods like `register(Registration)` and `deregister(Registration)` which allow you to provide custom registered services. `Registration` is a marker interface.

JAVA

```
@Configuration
@EnableDiscoveryClient(autoRegister=false)
public class MyConfiguration {
    private ServiceRegistry registry;

    public MyConfiguration(ServiceRegistry registry) {
        this.registry = registry;
    }

    // called via some external process, such as an event or a custom actuator endpoint
    public void register() {
        Registration registration = constructRegistration();
        this.registry.register(registration);
    }
}
```

Each `ServiceRegistry` implementation has its own `Registry` implementation.

ServiceRegistry Auto-Registration

By default, the `ServiceRegistry` implementation will auto-register the running service. To disable that behavior, there are two methods. You can set `@EnableDiscoveryClient(autoRegister=false)` to permanently disable auto-registration. You can also set `spring.cloud.service-registry.auto-registration.enabled=false` to disable the behavior via configuration.

Service Registry Actuator Endpoint

A `/service-registry` actuator endpoint is provided by Commons. This endpoint relys on a `Registration` bean in the Spring Application Context. Calling `/service-registry/instance-status` via a GET will return the status of the `Registration`. A POST to the same endpoint with a `String` body will change the status of the current `Registration` to the new value. Please see the documentation of the `ServiceRegistry` implementation you are using for the allowed values for updating the status and the values retured for the status.

Spring RestTemplate as a Load Balancer Client

`RestTemplate` can be automatically configured to use ribbon. To create a load balanced `RestTemplate` create a `RestTemplate` `@Bean` and use the `@LoadBalanced` qualifier.

WARNING A `RestTemplate` bean is no longer created via auto configuration. It must be created by individual applications.

JAVA

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

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 (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/ribbon/RibbonAutoConfiguration.java) for details of how the `RestTemplate` is set up.

Retrying Failed Requests

A load balanced `RestTemplate` can be configured to retry failed requests. By default this logic is disabled, you can enable it by adding Spring Retry (https://github.com/spring-projects/spring-retry) to your application's classpath. The load balanced `RestTemplate` will honor some of the Ribbon configuration values related to retrying failed requests. If you would like to disable the retry logic with Spring Retry on the classpath you can set `spring.cloud.loadbalancer.retry.enabled=false`. The properties you can use are `client.ribbon.MaxAutoRetries`, `client.ribbon.MaxAutoRetriesNextServer`, and `client.ribbon.OkToRetryOnAllOperations`. See the Ribbon documentation (https://github.com/Netflix/ribbon/wiki/Getting-Started#the-properties-file-sample-clientproperties) for a description of what there properties do.

NOTE	`client` in the above examples should be replaced with your Ribbon client's name.

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 `@LoadBalanced` qualifier when you create your `@Bean`.

IMPORTANT	Notice the `@Primary` annotation on the plain `RestTemplate` declaration in the example below, to disambiguate the unqualified `@Autowired` injection.

JAVA

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate loadBalanced() {
        return new RestTemplate();
    }

    @Primary
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

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);
    }
}
```

TIP

If you see errors like `java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.Foo.restTemplate to com.sun.proxy.$Proxy89` try injecting `RestOperations` instead or setting `spring.aop.proxyTargetClass=true`.

Ignore Network Interfaces

Sometimes it is useful to ignore certain named network interfaces so they can be excluded from Service Discovery registration (eg. running in a Docker container). A list of regular expressions can be set that will cause the desired network interfaces to be ignored. The following configuration will ignore the "docker0" interface and all interfaces that start with "veth".

application.yml

```
spring:
  cloud:
    inetutils:
      ignoredInterfaces:
        - docker0
        - veth.*
```

You can also force to use only specified network addresses using list of regular expressions:

application.yml

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
```

You can also force to use only site local addresses. See Inet4Address.html.isSiteLocalAddress() (https://docs.oracle.com/javase/8/docs/api/java/net/Inet4Address.html#isSiteLocalAddress--) for more details what is site local address.

application.yml

```
spring:
  cloud:
    inetutils:
      useOnlySiteLocalInterfaces: true
```

Spring Cloud Config

Dalston.SR3

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.

Quick Start

Start the server:

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

The server is a Spring Boot application so you can run it from your IDE instead if you prefer (the main class is `ConfigServerApplication`). Then try out a client:

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

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".)

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

YAML

```
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

XML

```
   <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.3.5.RELEASE</version>
       <relativePath /> <!-- lookup parent from repository -->
   </parent>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.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:

```
@SpringBootApplication
@RestController
public class Application {

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

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

}
```

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-repo/bar.properties":{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

(a property source called "configService:<URL of remote repository>/<file name>" 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.

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. So this app is a config server:

ConfigServer.java

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServer.class, args);
  }
}
```

JAVA

Like all Spring Boot apps it runs on port 8080 by default, but you can switch it to the conventional port 8888 in various ways. The easiest, which also sets a default configuration repository, is by launching it with `spring.config.name=configserver` (there is a `configserver.yml` in the Config Server jar). Another is to use your own `application.properties`, e.g.

application.properties

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

PROPERTIES

where `${user.home}/config-repo` is a git repository containing YAML and properties files.

NOTE in Windows you need an extra "/" in the file URL if it is absolute with a drive prefix, e.g. `file:///${user.home}/config-repo`.

TIP Here's a recipe for creating the git repository in the example above:

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```

WARNING using the local filesystem for your git repository is intended for testing only. Use a server to host your configuration repositories in production.

WARNING the initial clone of your configuration repository will be quick and efficient if you only keep text files in it. If you start to store binary files, especially large ones, you may experience delays on the first request for configuration and/or out of memory errors in the server.

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.profiles.active" 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

```yaml
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 precedence (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 precedence 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). For example, if the label is `foo/bar`, replacing the slash would result in a label that looks like `foo(_)bar`. 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 ").

Placeholders in Git URI

Spring Cloud Config Server supports a git repository URL with placeholders for the `{application}` and `{profile}` (and `{label}` if you need it, but remember that the label is applied as a git label anyway). So you can easily support a "one repo per application" policy using (for example):

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myorg/{application}
```

or a "one repo per profile" policy using a similar pattern but with `{profile}`.

Pattern Matching and Multiple Repositories

There is also support for more complex requirements with pattern matching on the application and profile name. The pattern format is a comma-separated list of `{application}/{profile}` names with wildcards (where a pattern beginning with a wildcard may need to be quoted). Example:

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

If `{application}/{profile}` does not match any of the patterns, it will use the default uri defined under "spring.cloud.config.server.git.uri". In the above example, for the "simple" repository, the pattern is `simple/*` (i.e. it only matches one application named "simple" in all profiles). The "local" repository matches all application names beginning with "local" in all profiles (the `/*` suffix is added automatically to any pattern that doesn't have a profile matcher).

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.

The `pattern` property in the repo is actually an array, so you can use a YAML array (or `[0]`, `[1]`, etc. suffixes in properties files) to bind to multiple patterns. You may need to do this if you are going to run apps with multiple profiles. Example:

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            development:
              pattern:
                - '*/development'
                - '*/staging'
              uri: https://github.com/development/config-repo
            staging:
              pattern:
                - '*/qa'
                - '*/production'
              uri: https://github.com/staging/config-repo
```

NOTE

Spring Cloud will guess that a pattern containing a profile that doesn't end in `*` implies that you actually want to match a list of profiles starting with this pattern (so `*/staging` is a shortcut for `["*/staging", "*/staging,*"]`). This is common where you need to run apps in the "development" profile locally but also the "cloud" profile remotely, for instance.

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:

```yaml
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:

```yaml
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.

> **NOTE** Setting a repository to be cloned when the Config Server starts up can help to identify a misconfigured configuration source (e.g., an invalid repository URI) quickly, while the Config Server is starting up. With `cloneOnStart` not enabled for a configuration source, the Config Server may start successfully with a misconfigured or invalid configuration source and not detect an error until an application requests configuration from that configuration source.

Authentication

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

```yaml
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. "git@github.com:configuration/cloud-configuration". It is important that an entry for the Git server be present in the `~/.ssh/known_hosts` file and that it is in `ssh-rsa` format. Other formats (like `ecdsa-sha2-nistp256`) are not supported. To avoid surprises, you should ensure that only one entry is present in the `known_hosts` file for the Git server and that it is matching with the URL you provided to the config server. If you used a hostname in the URL, you want to have exactly that in the `known_hosts` file, not the IP. The repository is accessed using JGit, so any documentation you find on that should be applicable. HTTPS proxy settings can be set in `~/.git/config` or in the same way as for any other JVM process via system properties (`-Dhttps.proxyHost` and `-Dhttps.proxyPort`).

> **TIP** If you don't know where your `~/.git` directory is use `git config --global` to manipulate the settings (e.g. `git config --global http.sslVerify false`).

Authentication with AWS CodeCommit

AWS CodeCommit (https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html) authentication can also be done. AWS CodeCommit uses an authentication helper when using Git from the command line. This helper is not used with the JGit library, so a JGit CredentialProvider for AWS CodeCommit will be created if the Git URI matches the AWS CodeCommit pattern. AWS CodeCommit URIs always look like https://git-codecommit.${AWS_REGION}.amazonaws.com/${repopath}.

If you provide a username and password with an AWS CodeCommit URI, then these must be the AWS accessKeyId and secretAccessKey (https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSGettingStartedGuide/AWSCredentials.html) to be used to access the repository. If you do not specify a username and password, then the accessKeyId and secretAccessKey will be retrieved using the AWS Default Credential Provider Chain (https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/credentials.html).

If your Git URI matches the CodeCommit URI pattern (above) then you must provide valid AWS credentials in the username and password, or in one of the locations supported by the default credential provider chain. AWS EC2 instances may use IAM Roles for EC2 Instances (https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html).

Note: The aws-java-sdk-core jar is an optional dependency. If the aws-java-sdk-core jar is not on your classpath, then the AWS Code Commit credential provider will not be created regardless of the git server URI.

Git SSH configuration using properties

By default, the JGit library used by Spring Cloud Config Server uses SSH configuration files such as `~/.ssh/known_hosts` and `/etc/ssh/ssh_config` when connecting to Git repositories using an SSH URI. In cloud environments such as Cloud Foundry, the local filesystem may be ephemeral or not easily accessible. For cases such as these, SSH configuration can be set using Java properties. In order to activate property based SSH configuration, the property `spring.cloud.config.server.git.ignoreLocalSshSettings` must be set to `true`. Example:

YAML

```
spring:
  cloud:
    config:
      server:
        git:
          uri: git@gitserver.com:team/repo1.git
          ignoreLocalSshSettings: true
          hostKey: someHostKey
          hostKeyAlgorithm: ssh-rsa
          privateKey: |
                      -----BEGIN RSA PRIVATE KEY-----
                      MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX
                      IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF
                      ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud
                      1l7R+wxIqmJo1IYyy16xS8WsjyQuyCOlL456qkd5BDZOAg8j2X9H9D5220Ln7s9i
                      oezTipXipS7p7Jekf3Ywx6abJwOmBOrX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W
                      DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd
                      fY6yTiKxFzwb38IQPOojIUWNrq0+9Xt+NsypviLHkXfXXCKKU4zUHeIGVRq5MN9b
                      BO56/RrcQHHOoJdUWuOV2qMqJvPUtCOCpGkD+valhfD75MxoXU7s3FK7yjxy3rsG
                      EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj
                      5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8
                      +AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe
                      pZVnfRixAoGBAO6Uiwt4O/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG
                      ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ
                      xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW
                      dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi
                      PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV2OYIBCVRKD1uX
                      VhEOozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z
                      FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R
                      gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4
                      VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47VO+JJpapLnF0KxGrqeGIjIRV
                      cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee
                      KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N
                      CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs
                      q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J
                      69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT
                      -----END RSA PRIVATE KEY-----
```

Table 1. SSH Configuration properties

Property Name	Remarks
ignoreLocalSshSettings	If true, use property based SSH config instead of file based. Must be set at as `spring.cloud.config.server.git.ignoreLocalSshSettings`, **not** inside a repository definition.
privateKey	Valid SSH private key. Must be set if `ignoreLocalSshSettings` is true and Git URI is SSH format
hostKey	Valid SSH host key. Must be set if `hostKeyAlgorithm` is also set
hostKeyAlgorithm	One of `ssh-dss`, `ssh-rsa`, `ecdsa-sha2-nistp256`, `ecdsa-sha2-nistp384` ,`ecdsa-sha2-nistp521`. Must be set if `hostKey` is also set
strictHostKeyChecking	`true` or `false`. If false, ignore errors with host key

Placeholders in Git Search Paths

Spring Cloud Config Server also supports a search path with placeholders for the `{application}` and `{profile}` (and `{label}` if you need it). Example:

YAML

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: '{application}'
```

searches the repository for files in the same name as the directory (as well as the top level). Wildcards are also valid in a search path with placeholders (any matching directory is included in the search).

Force pull in Git Repositories

As mentioned before Spring Cloud Config Server makes a clone of the remote git repository and if somehow the local copy gets dirty (e.g. folder content changes by OS process) so Spring Cloud Config Server cannot update the local copy from remote repository.

To solve this there is a `force-pull` property that will make Spring Cloud Config Server force pull from remote repository if the local copy is dirty. Example:

YAML

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

If you have a multiple repositories configuration you can configure the `force-pull` property per repository. Example:

YAML

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

NOTE The default value for `force-pull` property is `false`.

Version Control Backend Filesystem Use

WARNING With VCS based backends (git, svn) files are checked out or cloned to the local filesystem. By default they are put in the system temporary directory with a prefix of `config-repo-`. On linux, for example it could be `/tmp/config-repo-<randomid>`. Some operating systems routinely clean out (https://serverfault.com/questions/377348/when-does-tmp-get-cleared/377349#377349) temporary directories. This can lead to unexpected behaviour such as missing properties. To avoid this problem, change the directory Config Server uses, by setting `spring.cloud.config.server.git.basedir` or `spring.cloud.config.server.svn.basedir` to a directory that does not reside in the system temp structure.

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".

NOTE

Remember to use the `file:` prefix for file resources (the default without a prefix is usually the classpath). Just as with any Spring Boot configuration you can embed `${}`-style environment placeholders, but remember that absolute paths in Windows require an extra "/", e.g. `file:///${user.home}/config-repo`

WARNING

The default value of the `searchLocations` is identical to a local Spring Boot application (so `[classpath:/, classpath:/config, file:./, file:./config]`). This does not expose the `application.properties` from the server to all clients because any property sources present in the server are removed before being sent to the client.

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.

The search locations can contain placeholders for `{application}`, `{profile}` and `{label}`. In this way you can segregate the directories in the path, and choose a strategy that makes sense for you (e.g. sub-directory per application, or sub-directory per profile).

If you don't use placeholders in the search locations, this repository also appends 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). Thus the default behaviour with no placeholders is the same as adding a search location ending with `/{label}/`. For example `file:/tmp/config` is the same as `file:/tmp/config,file:/tmp/config/{label}`. This behavior can be disabled by setting `spring.cloud.config.server.native.addLabelLocations=false`.

Vault Backend

Spring Cloud Config Server also supports Vault (https://www.vaultproject.io) as a backend.

> Vault is a tool for securely accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, and more. Vault provides a unified interface to any secret, while providing tight access control and recording a detailed audit log.

For more information on Vault see the Vault quickstart guide (https://www.vaultproject.io/intro/index.html).

To enable the config server to use a Vault backend you must run your config server with the `vault` profile. For example in your config server's `application.properties` you can add `spring.profiles.active=vault`.

By default the config server will assume your Vault server is running at `http://127.0.0.1:8200`. It also will assume that the name of backend is `secret` and the key is `application`. All of these defaults can be configured in your config server's `application.properties`. Below is a table of configurable Vault properties. All properties are prefixed with `spring.cloud.config.server.vault`.

Name	Default Value
host	127.0.0.1
port	8200
scheme	http
backend	secret
defaultKey	application
profileSeparator	,

All configurable properties can be found in `org.springframework.cloud.config.server.environment.VaultEnvironmentRepository`.

With your config server running you can make HTTP requests to the server to retrieve values from the Vault backend. To do this you will need a token for your Vault server.

First place some data in you Vault. For example

```SH
$ vault write secret/application foo=bar baz=bam
$ vault write secret/myapp foo=myappsbar
```

Now make the HTTP request to your config server to retrieve the values.

```
$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"
```

You should see a response similar to this after making the above request.

```JSON
{
   "name":"myapp",
   "profiles":[
      "default"
   ],
   "label":null,
   "version":null,
   "state":null,
   "propertySources":[
      {
         "name":"vault:myapp",
         "source":{
            "foo":"myappsbar"
         }
      },
      {
         "name":"vault:application",
         "source":{
            "baz":"bam",
            "foo":"bar"
         }
      }
   ]
}
```

Multiple Properties Sources

When using Vault you can provide your applications with multiple properties sources. For example, assume you have written data to the following paths in Vault.

```SH
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

Properties written to `secret/application` are available to all applications using the Config Server. An application with the name `myApp` would have any properties written to `secret/myApp` and `secret/application` available to it. When `myApp` has the `dev` profile enabled then properties written to all of the above paths would be available to it, with properties in the first path in the list taking priority over the others.

Sharing Configuration With All Applications

File Based Repositories

With file-based (i.e. git, svn and native) repositories, resources with file names in `application*` are shared between all client applications (so `application.properties`, `application.yml`, `application-*.properties` etc.). You can use resources with these file names to configure global defaults and have them overridden by application-specific files as necessary.

The #_property_overrides[property overrides] feature can also be used for setting global defaults, and with placeholders applications are allowed to override them locally.

> **TIP** With the "native" profile (local file system backend) it is recommended that you use an explicit search location that isn't part of the server's own configuration. Otherwise the `application*` resources in the default search locations are removed because they are part of the server.

Vault Server

When using Vault as a backend you can share configuration with all applications by placing configuration in `secret/application`. For example, if you run this Vault command

```SH
$ vault write secret/application foo=bar baz=bam
```

All applications using the config server will have the properties `foo` and `baz` available to them.

Composite Environment Repositories

In some scenarios you may wish to pull configuration data from multiple environment repositories. To do this just enable multiple profiles in your config server's application properties or YAML file. If, for example, you want to pull configuration data from a Git repository as well as a SVN repository you would set the following properties for your configuration server.

```YAML
spring:
  profiles:
    active: git, svn
  cloud:
    config:
      server:
        svn:
          uri: file:///path/to/svn/repo
          order: 2
        git:
          uri: file:///path/to/git/repo
          order: 1
```

In addition to each repo specifying a URI, you can also specify an `order` property. The `order` property allows you to specify the priority order for all your repositories. The lower the numerical value of the `order` property the higher priority it will have. The priority order of a repository will help resolve any potential conflicts between repositories that contain values for the same properties.

NOTE	Any type of failure when retrieving values from an environment repositoy will result in a failure for the entire composite environment.

NOTE	When using a composite environment it is important that all repos contain the same label(s). If you have an environment similar to the one above and you request configuration data with the label `master` but the SVN repo does not contain a branch called `master` the entire request will fail.

Custom Composite Environment Repositories

It is also possible to provide your own `EnvironmentRepository` bean to be included as part of a composite environment in addition to using one of the environment repositories from Spring Cloud. To do this your bean must implement the `EnvironmentRepository` interface. If you would like to control the priority of you custom `EnvironmentRepository` within the composite environment you should also implement the `Ordered` interface and override the `getOrdered` method. If you do not implement the `Ordered` interface then your `EnvironmentRepository` will be given the lowest priority.

Property Overrides

The Config Server has an "overrides" feature that allows the operator to provide configuration properties to all applications that cannot be accidentally changed by the application using the normal Spring Boot hooks. To declare overrides just add a map of name-value pairs to `spring.cloud.config.server.overrides` . For example

```YAML
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

will cause all applications that are config clients to read `foo=bar` independent of their own configuration. (Of course an application can use the data in the Config Server in any way it likes, so overrides are not enforceable, but they do provide useful default behaviour if they are Spring Cloud Config clients.)

TIP	Normal, Spring environment placeholders with "${}" can be escaped (and resolved on the client) by using backslash ("\") to escape the "$" or the "{", e.g. `\${app.foo:bar}` resolves to "bar" unless the app provides its own "app.foo". Note that in YAML you don't need to escape the backslash itself, but in properties files you do, when you configure the overrides on the server.

You can change the priority of all overrides in the client to be more like default values, allowing applications to supply their own values in environment variables or System properties, by setting the flag `spring.cloud.config.overrideNone=true` (default is false) in the remote repository.

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.

YAML

```
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 encrypted 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 removed from the property source and an additional property is added with the same key, but prefixed with "invalid." and a value that means "not applicable" (usually "<n/a>"). This is 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

YAML

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

Encrypted values in a .properties file must not be wrapped in quotes, otherwise the value will not be decrypted:

application.properties

```
spring.datasource.username: dbuser
spring.datasource.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
```

NOTE	If the value you are encrypting has characters in it that need to be URL encoded you should use the `--data-urlencode` option to `curl` to make sure they are encoded properly.

TIP	Be sure not to include any of the curl command statistics in the encrypted value. Outputting the value to a file can help avoid this problem.

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 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

TIP	If you are testing like this with curl, then use `--data-urlencode` (instead of `-d`) or set an explicit `Content-Type: text/plain` to make sure curl encodes the data correctly when there are special characters ('+' is particularly tricky).

Take the encrypted 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 `/encrypt` 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 (so all encryptions use the same key).

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 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

To use a key in a file (e.g. an RSA public key for encryption) 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:

YAML

```
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:

YAML

```
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.

Serving Encrypted Properties

Sometimes you want the clients to decrypt the configuration locally, instead of doing it in the server. In that case you can still have /encrypt and /decrypt endpoints (if you provide the `encrypt.*` configuration to locate a key), but you need to explicitly switch off the decryption of outgoing properties using `spring.cloud.config.server.encrypt.enabled=false`. If you don't care about the endpoints, then it should work if you configure neither the key nor the enabled flag.

Serving Alternative Formats

The default JSON format from the environment endpoints is perfect for consumption by Spring applications because it maps directly onto the `Environment` abstraction. If you prefer you can consume the same data as YAML or Java properties by adding a suffix to the resource path (".yml", ".yaml" or ".properties"). This can be useful for consumption by applications that do not care about the structure of the JSON endpoints, or the extra metadata they provide, for example an application that is not using Spring might benefit from the simplicity of this approach.

The YAML and properties representations have an additional flag (provided as a boolean query parameter `resolvePlaceholders`) to signal that placeholders in the source documents, in the standard Spring `${…}` form, should be resolved in the output where possible before rendering. This is a useful feature for consumers that don't know about the Spring placeholder conventions.

NOTE

there are limitations in using the YAML or properties formats, mainly in relation to the loss of metadata. The JSON is structured as an ordered list of property sources, for example, with names that correlate with the source. The YAML and properties forms are coalesced into a single map, even if the origin of the values has multiple sources, and the names of the original source files are lost. The YAML representation is not necessarily a faithful representation of the YAML source in a backing repository either: it is constructed from a list of flat property sources, and assumptions have to be made about the form of the keys.

Serving Plain Text

Instead of using the `Environment` abstraction (or one of the alternative representations of it in YAML or properties format) your applications might need generic plain text configuration files, tailored to their environment. The Config Server provides these through an additional endpoint at `/{name}/{profile}/{label}/{path}` where "name", "profile" and "label" have the same meaning as the regular environment endpoint, but "path" is a file name (e.g. `log.xml`). The source files for this endpoint are located in the same way as for the environment endpoints: the same search path is used as for properties or YAML files, but instead of aggregating all matching resources, only the first one to match is returned.

After a resource is located, placeholders in the normal format (`${…}`) are resolved using the effective `Environment` for the application name, profile and label supplied. In this way the resource endpoint is tightly integrated with the environment endpoints. Example, if you have this layout for a GIT (or SVN) repository:

```
application.yml
nginx.conf
```

where `nginx.conf` looks like this:

```
server {
    listen              80;
    server_name         ${nginx.server.name};
}
```

and `application.yml` like this:

YAML

```
nginx:
  server:
    name: example.com
---
spring:
  profiles: development
nginx:
  server:
    name: develop.com
```

then the `/foo/default/master/nginx.conf` resource looks like this:

```
server {
    listen              80;
    server_name         example.com;
}
```

and `/foo/development/master/nginx.conf` like this:

```
server {
    listen              80;
    server_name         develop.com;
}
```

NOTE just like the source files for environment configuration, the "profile" is used to resolve the file name, so if you want a profile-specific file then `/*/development/*/logback.xml` will be resolved by a file called `logback-development.xml` (in preference to `logback.xml`).

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. An 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.

NOTE

It should be obvious, but remember that if you use the bootstrap flag the config server will need to have its name and repository URI configured in `bootstrap.yml` .

To change the location of the server endpoints you can (optionally) set `spring.cloud.config.server.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`).

If you want to read the configuration for an application directly from the backend repository (instead of from the config server) that's basically an embedded config server with no endpoints. You can switch off the endpoints entirely if you don't use the `@EnableConfigServer` annotation (just set `spring.cloud.config.server.bootstrap=true`).

Push Notifications and Spring Cloud Bus

Many source code repository providers (like Github, Gitlab or Bitbucket for instance) will notify you of changes in a repository through a webhook. You can configure the webhook via the provider's user interface as a URL and a set of events in which you are interested. For instance Github (https://developer.github.com/v3/activity/events/types/#pushevent) will POST to the webhook with a JSON body containing a list of commits, and a header "X-Github-Event" equal to "push". If you add a dependency on the `spring-cloud-config-monitor` library and activate the Spring Cloud Bus in your Config Server, then a "/monitor" endpoint is enabled.

When the webhook is activated the Config Server will send a `RefreshRemoteApplicationEvent` targeted at the applications it thinks might have changed. The change detection can be strategized, but by default it just looks for changes in files that match the application name (e.g. "foo.properties" is targeted at the "foo" application, and "application.properties" is targeted at all applications). The strategy if you want to override the behaviour is `PropertyPathNotificationExtractor` which accepts the request headers and body as parameters and returns a list of file paths that changed.

The default configuration works out of the box with Github, Gitlab or Bitbucket. In addition to the JSON notifications from Github, Gitlab or Bitbucket you can trigger a change notification by POSTing to "/monitor" with a form-encoded body parameters `path={name}`. This will broadcast to applications matching the "{name}" pattern (can contain wildcards).

NOTE

the `RefreshRemoteApplicationEvent` will only be transmitted if the `spring-cloud-bus` is activated in the Config Server and in the client application.

NOTE

the default configuration also detects filesystem changes in local git repositories (the webhook is not used in that case but as soon as you edit a config file a refresh will be broadcast).

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").

Discovery First Bootstrap

If you are using a `DiscoveryClient implementation, such as Spring Cloud Netflix and Eureka Service Discovery or Spring Cloud Consul (Spring Cloud Zookeeper does not support this yet), then you can have the Config Server register with the Discovery Service 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 `DiscoveryClient` 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 appropriate discovery configuration. For example, with Spring Cloud Netflix, you need to define 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 the Discovery Service 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`).

The discovery client implementations all support some kind of metadata map (e.g. for Eureka we have `eureka.instance.metadataMap`). Some additional properties of the Config Server may need to be configured in its service registration metadata so that clients can connect correctly. If the Config Server is secured with HTTP Basic you can configure the credentials as "username" and "password". And if the Config Server has a context path you can set "configPath". Example, for a Config Server that is a Eureka client:

bootstrap.yml

```yaml
eureka:
  instance:
    ...
    metadataMap:
      user: osufhalskjrtl
      password: lviuhlszvaorhvlo5847
      configPath: /config
```

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.cloud.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

```yaml
spring:
  cloud:
    config:
     uri: https://user:secret@myconfig.mycompany.com
```

or

bootstrap.yml

```yaml
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

```yaml
spring:
  cloud:
    config:
     uri: ${vcap.services.configserver.credentials.uri:http://user:password@localhost: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).

Health Indicator

The Config Client supplies a Spring Boot Health Indicator that attempts to load configuration from Config Server. The health indicator can be disabled by setting `health.config.enabled=false`. The response is also cached for performance reasons. The default cache time to live is 5 minutes. To change that value set the `health.config.time-to-live` property (in milliseconds).

Providing A Custom RestTemplate

In some cases you might need to customize the requests made to the config server from the client. Typically this involves passing special `Authorization` headers to authenticate requests to the server. To provide a custom `RestTemplate` follow the steps below.

1. Create a new configuration bean with an implementation of `PropertySourceLocator`.

CustomConfigServiceBootstrapConfiguration.java

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
    @Bean
    public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
        ConfigClientProperties clientProperties = configClientProperties();
       ConfigServicePropertySourceLocator configServicePropertySourceLocator =  new
ConfigServicePropertySourceLocator(clientProperties);
        configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));
        return configServicePropertySourceLocator;
    }
}
```
JAVA

1. In `resources/META-INF` create a file called `spring.factories` and specify your custom configuration.

spring.factories

```
org.springframework.cloud.bootstrap.BootstrapConfiguration =
com.my.config.client.CustomConfigServiceBootstrapConfiguration
```
PROPERTIES

Vault

When using Vault as a backend to your config server the client will need to supply a token for the server to retrieve values from Vault. This token can be provided within the client by setting `spring.cloud.config.token` in `bootstrap.yml`.

bootstrap.yml

```
spring:
  cloud:
    config:
      token: YourVaultToken
```
YAML

Vault

Nested Keys In Vault

Vault supports the ability to nest keys in a value stored in Vault. For example

```
echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -
```

This command will write a JSON object to your Vault. To access these values in Spring you would use the traditional dot(.) annotation. For example

```
@Value("${appA.secret}")
String name = "World";
```
JAVA

The above code would set the `name` variable to `appAsecret`.

Spring Cloud Netflix

Dalston.SR3

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).

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.

How to Include Eureka Client

To include Eureka Client in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-eureka` . See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

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:

JAVA

```
@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 (https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java) and EurekaClientConfigBean (https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaClientConfigBean.java) for more details of the configurable options.

Authenticating with the Eureka Server

HTTP basic authentication will be automatically added to your eureka client if one of the `eureka.client.serviceUrl.defaultZone` URLs has credentials embedded in it (curl style, like `http://user:password@localhost:8761/eureka`). For more complex needs you can create a `@Bean` of type `DiscoveryClientOptionalArgs` and inject `ClientFilter` instances into it, all of which will be applied to the calls from the client to the server.

NOTE

Because of a limitation in Eureka it isn't possible to support per-server basic auth credentials, so only the first set that are found will be used.

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.context-path}/info
    healthCheckUrlPath: ${management.context-path}/health
```

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

Registering a Secure Application

If your app wants to be contacted over HTTPS you can set two flags in the `EurekaInstanceConfig`, *viz* `eureka.instance.[nonSecurePortEnabled,securePortEnabled]=[false,true]` respectively. This will make Eureka publish instance information showing an explicit preference for secure communication. The Spring Cloud `DiscoveryClient` will always return an `https://...` URI for a service configured this way, and the Eureka (native) instance information will have a secure health check URL.

Because of the way Eureka works internally, it will still publish a non-secure URL for status and home page unless you also override those explicitly. You can use placeholders to configure the eureka instance urls, e.g.

application.yml

```
eureka:
  instance:
    statusPageUrl: https://${eureka.hostname}/info
    healthCheckUrl: https://${eureka.hostname}/health
    homePageUrl: https://${eureka.hostname}/
```

(Note that `${eureka.hostname}` is a native placeholder only available in later versions of Eureka. You could achieve the same thing with Spring placeholders as well, e.g. using `${eureka.instance.hostName}`.)

NOTE

If your app is running behind a proxy, and the SSL termination is in the proxy (e.g. if you run in Cloud Foundry or other platforms as a service) then you will need to ensure that the proxy "forwarded" headers are intercepted and handled by the application. An embedded Tomcat container in a Spring Boot app does this automatically if it has explicit configuration for the 'X-Forwarded-*` headers. A sign that you got this wrong will be that the links rendered by your app to itself will be wrong (the wrong host, port or protocol).

Eureka's Health Checks

By default, Eureka uses the client heartbeat to determine if a client is up. Unless specified otherwise the Discovery Client will not propagate the current health check status of the application per the Spring Boot Actuator. Which means that after successful registration Eureka will always announce that the application is in 'UP' state. This behaviour can be altered by enabling Eureka health checks, which results in propagating application status to Eureka. As a consequence every other application won't be sending traffic to application in state other then 'UP'.

application.yml

```
eureka:
  client:
    healthcheck:
      enabled: true
```

WARNING	`eureka.client.healthcheck.enabled=true` should only be set in `application.yml`. Setting the value in `bootstrap.yml` will cause undesirable side effects like registering in eureka with an `UNKNOWN` status.

If you require more control over the health checks, you may consider implementing your own `com.netflix.appinfo.HealthCheckHandler`.

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). By default, the `eureka.instance.instanceId` is `vcap.application.instance_id`. For example:

application.yml

```
eureka:
  instance:
    hostname: ${vcap.application.uris[0]}
    nonSecurePort: 80
```

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 (https://run.pivotal.io)).

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 AWS aware and this can be done by customizing the EurekaInstanceConfigBean (https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java) the following way:

JAVA

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

Changing the Eureka Instance ID

A vanilla Netflix Eureka instance is registered with an ID that is equal to its host name (i.e. only one service per host). Spring Cloud Eureka provides a sensible default that looks like this: `${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}`. For example `myhost:myappname:8080`.

Using Spring Cloud you can override this by providing a unique identifier in `eureka.instance.instanceId`. For example:

application.yml

```
eureka:
  instance:
    instanceId:
${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

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 `vcap.application.instance_id` will be populated automatically in a Spring Boot application, so the random value will not be needed.

Using the EurekaClient

Once you have an app that is `@EnableDiscoveryClient` (or `@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.EurekaClient` (as opposed to the Spring Cloud `DiscoveryClient`), e.g.

```
@Autowired
private EurekaClient discoveryClient;

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

> **TIP** Don't use the `EurekaClient` 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 EurekaClient

You don't have to use the raw Netflix `EurekaClient` 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.

```
@Autowired
private DiscoveryClient discoveryClient;

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

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 heartbeats). 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.

Zones

If you have deployed Eureka clients to multiple zones than you may prefer that those clients leverage services within the same zone before trying services in another zone. To do this you need to configure your Eureka clients correctly.

First, you need to make sure you have Eureka servers deployed to each zone and that they are peers of each other. See the section on zones and regions for more information.

Next you need to tell Eureka which zone your service is in. You can do this using the `metadataMap` property. For example if `service 1` is deployed to both `zone 1` and `zone 2` you would need to set the following Eureka properties in `service 1`

Service 1 in Zone 1

```
eureka.instance.metadataMap.zone = zone1
eureka.client.preferSameZoneEureka = true
```

Service 1 in Zone 2

```
eureka.instance.metadataMap.zone = zone2
eureka.client.preferSameZoneEureka = true
```

Service Discovery: Eureka Server

How to Include Eureka Server

To include Eureka Server in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-eureka-server`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

How to Run a Eureka Server

Example eureka server;

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

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

}
```

JAVA

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 (https://github.com/cfregly/fluxcapacitor/wiki/NetflixOSS-FAQ#eureka-service-discovery-load-balancer) and google group discussion (https://groups.google.com/forum/?fromgroups#!topic/eureka_netflix/g3p2r7gHnN0).

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 Boot Gradle plugin must be added and the Spring cloud starter parent bom must be imported like so:

build.gradle

```
buildscript {
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.5.RELEASE")
  }
}

apply plugin: "spring-boot"

dependencyManagement {
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-dependencies:Brixton.RELEASE"
  }
}
```

JAVA

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}/eureka/
```

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.

Circuit Breaker: Hystrix Clients

Netflix has created a library called Hystrix (https://github.com/Netflix/Hystrix) that implements the circuit breaker pattern (http://martinfowler.com/bliki/CircuitBreaker.html). In a microservice architecture it is common to have multiple layers of service calls.

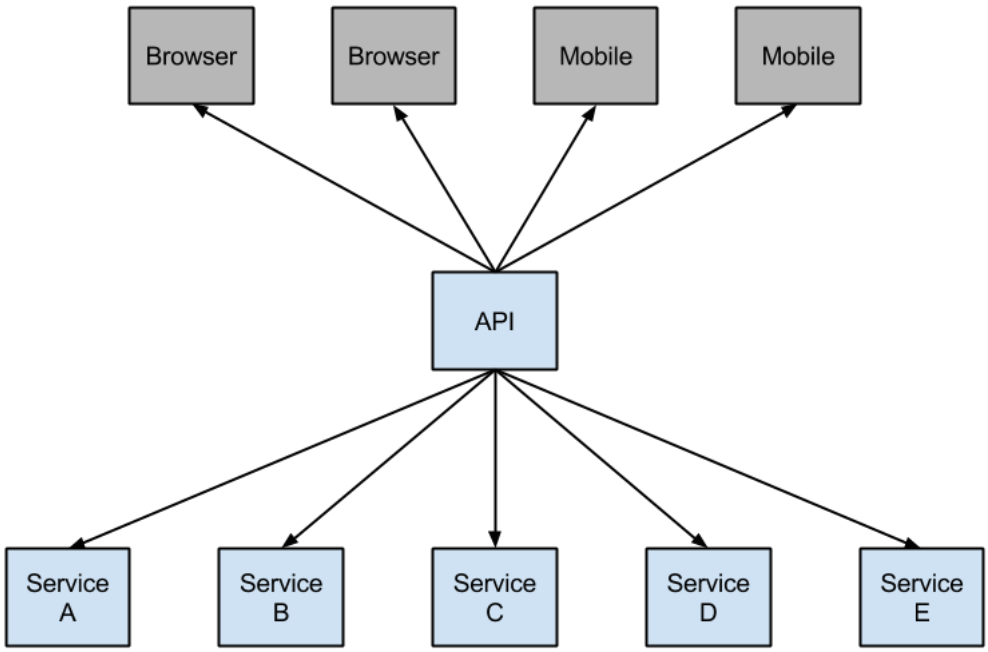


Figure 1. Microservice Graph

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 is greater than `circuitBreaker.requestVolumeThreshold` (default: 20 requests) and failue percentage is greater than `circuitBreaker.errorThresholdPercentage` (default: >50%) in a rolling window defined by `metrics.rollingStats.timeInMilliseconds` (default: 10 seconds), 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.

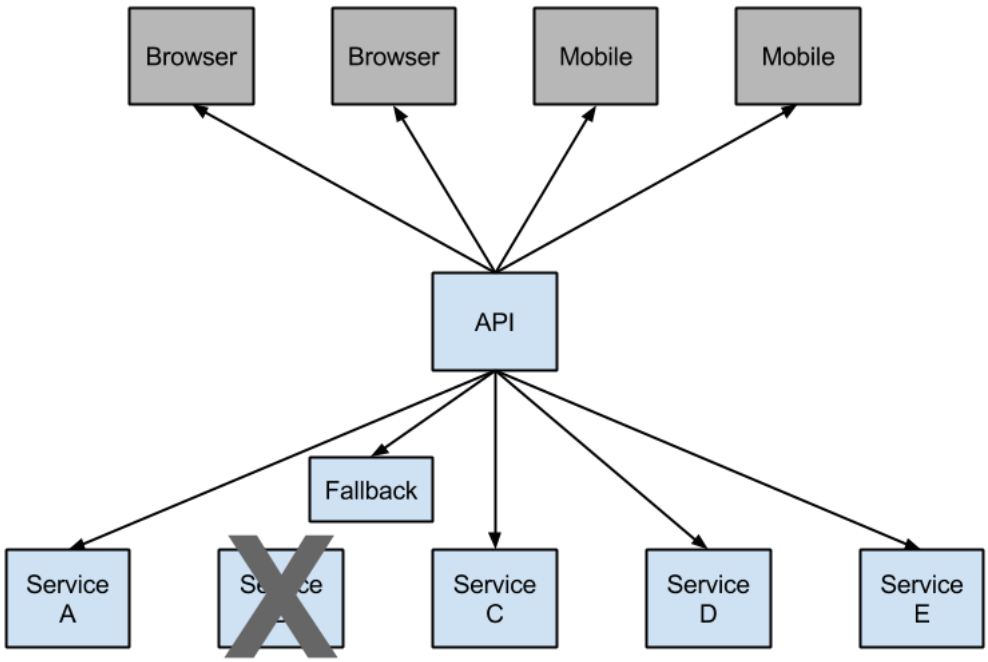


Figure 2. Hystrix fallback prevents cascading failures

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.

How to Include Hystrix

To include Hystrix in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-hystrix`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

Example boot app:

```
@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 */;
    }
}
```

The `@HystrixCommand` is provided by a Netflix contrib library called "javanica" (https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-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 (https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#configuration) for more details. See the Hystrix wiki (https://github.com/Netflix/Hystrix/wiki/Configuration) 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:

JAVA

```
@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.

You also have the option to set the `hystrix.shareSecurityContext` property to `true`. Doing so will auto configure an Hystrix concurrency strategy plugin hook who will transfer the `SecurityContext` from your main thread to the one used by the Hystrix command. Hystrix does not allow multiple hystrix concurrency strategy to be registered so an extension mechanism is available by declaring your own `HystrixConcurrencyStrategy` as a Spring bean. Spring Cloud will lookup for your implementation within the Spring context and wrap it inside its own plugin.

Health Indicator

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

JSON

```
{
    "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.

XML

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

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.

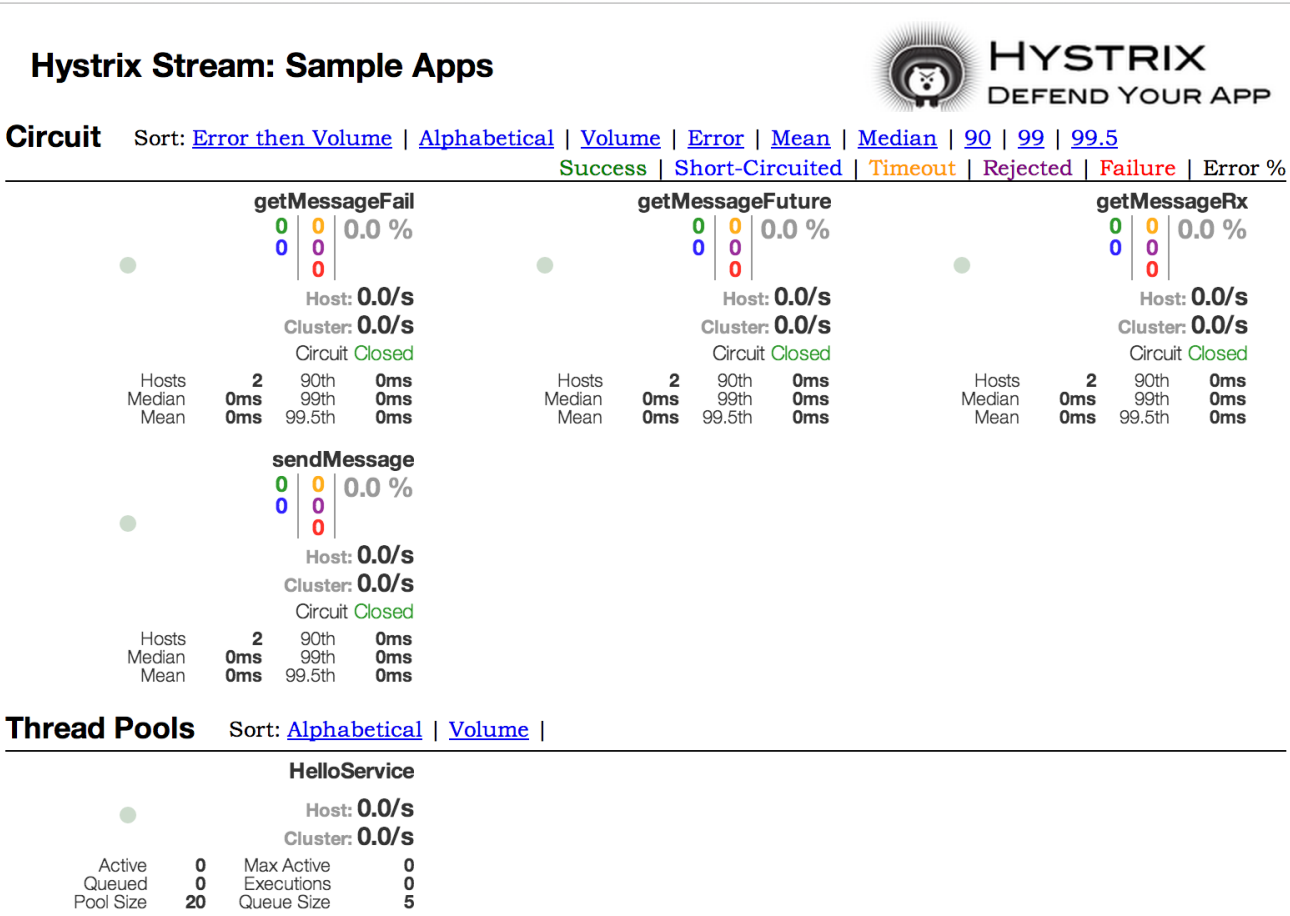


Figure 3. Hystrix Dashboard

Hystrix Timeouts And Ribbon Clients

When using Hystrix commands that wrap Ribbon clients you want to make sure your Hystrix timeout is configured to be longer than the configured Ribbon timeout, including any potential retries that might be made. For example, if your Ribbon connection timeout is one second and the Ribbon client might retry the request three times, than your Hystrix timeout should be slightly more than three seconds.

How to Include Hystrix Dashboard

To include the Hystrix Dashboard in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-hystrix-dashboard`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

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.

> **NOTE** When connecting to a `/hystrix.stream` endpoint which uses HTTPS the certificate used by the server must be trusted by the JVM. If the certificate is not trusted you must import the certificate into the JVM in order for the Hystrix Dashboard to make a successful connection to the stream endpoint.

Turbine

Looking at an individual instances Hystrix data is not very useful in terms of the overall health of the system. Turbine (https://github.com/Netflix/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 (https://github.com/Netflix/Turbine/wiki/Configuration-(1.x)) apply. The only difference is that the `turbine.instanceUrlSuffix` does not need the port prepended as this is handled automatically unless `turbine.instanceInsertPort=false`.

> **NOTE** By default, Turbine looks for the `/hystrix.stream` endpoint on a registered instance by looking up its `homePageUrl` entry in Eureka, then appending `/hystrix.stream` to it. This means that if `spring-boot-actuator` is running on its own port (which is the default), the call to `/hystrix.stream` will fail. To make turbine find the Hystrix stream at the correct port, you need to add `management.port` to the instances' metadata:

```
eureka:
  instance:
    metadata-map:
      management.port: ${management.port:8081}
```

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=<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, and escaped with double quotes if it is in YAML as well):

```
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`.

NOTE

by default Spring Cloud allows Turbine to use the host and port to allow multiple processes per host, per cluster. If you want the native Netflix behaviour built into Turbine that does *not* allow multiple processes per host, per cluster (the key to the instance id is the hostname), then set the property `turbine.combineHostPort=false`.

Turbine Stream

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 messaging. All you need to do on the client is add a dependency to `spring-cloud-netflix-hystrix-stream` and the `spring-cloud-starter-stream-*` of your choice (see Spring Cloud Stream documentation for details on the brokers, and how to configure the client credentials, but it should work out of the box for a local broker).

On the server side Just create a Spring Boot application and annotate it with `@EnableTurbineStream` 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.stream.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 Stream Server instead of individual Hystrix streams. If Turbine Stream is running on port 8989 on myhost, then put `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-stream` that has all the dependencies you need to get a Turbine Stream server running - just add the Stream binder of your choice, e.g. `spring-cloud-starter-stream-rabbit`. You need Java 8 to run the app because it is Netty-based.

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`.

How to Include Ribbon

To include Ribbon in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-ribbon`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

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:

JAVA

```
@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`
- `ServerListUpdater` ribbonServerListUpdater: `PollingServerListUpdater`

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:

JAVA

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

This replaces the `NoOpPing` with `PingUrl`.

Customizing the Ribbon Client using properties

Starting with version 1.2.0, Spring Cloud Netflix now supports customizing Ribbon clients using properties to be compatible with the Ribbon documentation (https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers#components-of-load-balancer).

This allows you to change behavior at start up time in different environments.

The supported properties are listed below and should be prefixed by `<clientName>.ribbon.` :

- `NFLoadBalancerClassName` : should implement `ILoadBalancer`
- `NFLoadBalancerRuleClassName` : should implement `IRule`
- `NFLoadBalancerPingClassName` : should implement `IPing`
- `NIWSServerListClassName` : should implement `ServerList`
- `NIWSServerListFilterClassName` should implement `ServerListFilter`

NOTE	Classes defined in these properties have precedence over beans defined using `@RibbonClient(configuration=MyRibbonConfig.class)` and the defaults provided by Spring Cloud Netflix.

To set the `IRule` for a service name `users` you could set the following:

application.yml

```
users:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule
```

See the Ribbon documentation (https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers) for implementations provided by Ribbon.

Using Ribbon with Eureka

When Eureka is used in conjunction with Ribbon (i.e., both are on the classpath) 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 remote clients 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 `approximateZoneFromHostname` 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` . The zone of the client is determined the same way as the remote instances by default, i.e. via `eureka.instance.metadataMap.zone` .

NOTE	The orthodox "archaius" way to set the client zone is via a configuration property called "@zone", and Spring Cloud will use that in preference to all other settings if it is available (note that the key will have to be quoted in YAML configuration).

NOTE	If there is no other source of zone data then a guess is made based on the client configuration (as opposed to the instance configuration). We take `eureka.client.availabilityZones` , which is a map from region name to a list of zones, and pull out the first zone for the instance's own region (i.e. the `eureka.client.region` , which defaults to "us-east-1" for comatibility with native Netflix).

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:

JAVA

```
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(), instance.getPort()));
        // ... do something with the URI
    }
}
```

Caching of Ribbon Configuration

Each Ribbon named client has a corresponding child Application Context that Spring Cloud maintains, this application context is lazily loaded up on the first request to the named client. This lazy loading behavior can be changed to instead eagerly load up these child Application contexts at startup by specifying the names of the Ribbon clients.

application.yml

```
ribbon:
  eager-load:
    enabled: true
    clients: client1, client2, client3
```

Declarative REST Client: Feign

Feign (https://github.com/Netflix/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.

How to Include Feign

To include Feign in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-feign`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

Example spring boot app

JAVA

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

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

}
```

StoreClient.java

JAVA

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

    @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes = "application/json")
    Store update(@PathVariable("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 name of the bean in the application context is the fully qualified name of the interface. To specify your own alias value you can use the `qualifier` value of the `@FeignClient` annotation.

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).

Overriding Feign Defaults

A central concept in Spring Cloud's Feign support is that of the named client. Each feign client 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 using the `@FeignClient` annotation. Spring Cloud creates a new ensemble as an `ApplicationContext` on demand for each named client using `FeignClientsConfiguration`. This contains (amongst other things) an `feign.Decoder`, a `feign.Encoder`, and a `feign.Contract`.

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

JAVA

```
@FeignClient(name = "stores", configuration = FooConfiguration.class)
public interface StoreClient {
    //..
}
```

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

NOTE | `FooConfiguration` does not need to be annotated with `@Configuration`. However, if it is, then take care to exclude it from any `@ComponentScan` that would otherwise include this configuration as it will become the default source for `feign.Decoder`, `feign.Encoder`, `feign.Contract`, etc., when specified. This can be avoided by putting it in a separate, non-overlapping package from any `@ComponentScan` or `@SpringBootApplication`, or it can be explicitly excluded in `@ComponentScan`.

NOTE | The `serviceId` attribute is now deprecated in favor of the `name` attribute.

WARNING | Previously, using the `url` attribute, did not require the `name` attribute. Using `name` is now required.

Placeholders are supported in the `name` and `url` attributes.

JAVA
```
@FeignClient(name = "${feign.name}", url = "${feign.url}")
public interface StoreClient {
    //..
}
```

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

- `Decoder` feignDecoder: `ResponseEntityDecoder` (which wraps a `SpringDecoder`)
- `Encoder` feignEncoder: `SpringEncoder`
- `Logger` feignLogger: `Slf4jLogger`
- `Contract` feignContract: `SpringMvcContract`
- `Feign.Builder` feignBuilder: `HystrixFeign.Builder`
- `Client` feignClient: if Ribbon is enabled it is a `LoadBalancerFeignClient`, otherwise the default feign client is used.

The OkHttpClient and ApacheHttpClient feign clients can be used by setting `feign.okhttp.enabled` or `feign.httpclient.enabled` to `true`, respectively, and having them on the classpath.

Spring Cloud Netflix *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

- `Logger.Level`
- `Retryer`
- `ErrorDecoder`
- `Request.Options`
- `Collection<RequestInterceptor>`
- `SetterFactory`

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

JAVA
```
@Configuration
public class FooConfiguration {
    @Bean
    public Contract feignContract() {
        return new feign.Contract.Default();
    }

    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "password");
    }
}
```

This replaces the `SpringMvcContract` with `feign.Contract.Default` and adds a `RequestInterceptor` to the collection of `RequestInterceptor`.

Default configurations can be specified in the `@EnableFeignClients` attribute `defaultConfiguration` in a similar manner as described above. The difference is that this configuration will apply to *all* feign clients.

NOTE

If you need to use `ThreadLocal` bound variables in your `RequestInterceptor`s you will need to either set the thread isolation strategy for Hystrix to `SEMAPHORE` or disable Hystrix in Feign.

application.yml

```yaml
# To disable Hystrix in Feign
feign:
  hystrix:
    enabled: false

# To set thread isolation to SEMAPHORE
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: SEMAPHORE
```

Creating Feign Clients Manually

In some cases it might be necessary to customize your Feign Clients in a way that is not possible using the methods above. In this case you can create Clients using the Feign Builder API (https://github.com/OpenFeign/feign/#basics). Below is an example which creates two Feign Clients with the same interface but configures each one with a separate request interceptor.

```java
@Import(FeignClientsConfiguration.class)
class FooController {

    private FooClient fooClient;

    private FooClient adminClient;

    @Autowired
    public FooController(
            Decoder decoder, Encoder encoder, Client client) {
        this.fooClient = Feign.builder().client(client)
                .encoder(encoder)
                .decoder(decoder)
                .requestInterceptor(new BasicAuthRequestInterceptor("user", "user"))
                .target(FooClient.class, "http://PROD-SVC");
        this.adminClient = Feign.builder().client(client)
                .encoder(encoder)
                .decoder(decoder)
                .requestInterceptor(new BasicAuthRequestInterceptor("admin", "admin"))
                .target(FooClient.class, "http://PROD-SVC");
    }
}
```

NOTE

In the above example `FeignClientsConfiguration.class` is the default configuration provided by Spring Cloud Netflix.

NOTE

`PROD-SVC` is the name of the service the Clients will be making requests to.

Feign Hystrix Support

If Hystrix is on the classpath and `feign.hystrix.enabled=true`, Feign will wrap all methods with a circuit breaker. Returning a `com.netflix.hystrix.HystrixCommand` is also available. This lets you use reactive patterns (with a call to `.toObservable()` or `.observe()` or asynchronous use (with a call to `.queue()`).

To disable Hystrix support on a per-client basis create a vanilla `Feign.Builder` with the "prototype" scope, e.g.:

```java
@Configuration
public class FooConfiguration {
    @Bean
    @Scope("prototype")
    public Feign.Builder feignBuilder() {
        return Feign.builder();
    }
}
```

WARNING

Prior to the Spring Cloud Dalston release, if Hystrix was on the classpath Feign would have wrapped all methods in a circuit breaker by default. This default behavior was changed in Spring Cloud Dalston in favor for an opt-in approach.

Feign Hystrix Fallbacks

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given `@FeignClient` set the `fallback` attribute to the class name that implements the fallback. You also need to declare your implementation as a Spring bean.

JAVA

```
@FeignClient(name = "hello", fallback = HystrixClientFallback.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

static class HystrixClientFallback implements HystrixClient {
    @Override
    public Hello iFailSometimes() {
        return new Hello("fallback");
    }
}
```

If one needs access to the cause that made the fallback trigger, one can use the `fallbackFactory` attribute inside `@FeignClient`.

JAVA

```
@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

@Component
static class HystrixClientFallbackFactory implements FallbackFactory<HystrixClient> {
    @Override
    public HystrixClient create(Throwable cause) {
        return new HystrixClientWithFallBackFactory() {
            @Override
            public Hello iFailSometimes() {
                return new Hello("fallback; reason was: " + cause.getMessage());
            }
        };
    }
}
```

WARNING

There is a limitation with the implementation of fallbacks in Feign and how Hystrix fallbacks work. Fallbacks are currently not supported for methods that return `com.netflix.hystrix.HystrixCommand` and `rx.Observable`.

Feign and `@Primary`

When using Feign with Hystrix fallbacks, there are multiple beans in the `ApplicationContext` of the same type. This will cause `@Autowired` to not work because there isn't exactly one bean, or one marked as primary. To work around this, Spring Cloud Netflix marks all Feign instances as `@Primary`, so Spring Framework will know which bean to inject. In some cases, this may not be desirable. To turn off this behavior set the `primary` attribute of `@FeignClient` to false.

JAVA

```
@FeignClient(name = "hello", primary = false)
public interface HelloClient {
    // methods here
}
```

Feign Inheritance Support

Feign supports boilerplate apis via single-inheritance interfaces. This allows grouping common operations into convenient base interfaces.

UserService.java

```
public interface UserService {

    @RequestMapping(method = RequestMethod.GET, value ="/users/{id}")
    User getUser(@PathVariable("id") long id);
}
```

UserResource.java

```
@RestController
public class UserResource implements UserService {

}
```

UserClient.java

```
package project.user;

@FeignClient("users")
public interface UserClient extends UserService {

}
```

NOTE

It is generally not advisable to share an interface between a server and a client. It introduces tight coupling, and also actually doesn't work with Spring MVC in its current form (method parameter mapping is not inherited).

Feign request/response compression

You may consider enabling the request or response GZIP compression for your Feign requests. You can do this by enabling one of the properties:

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

Feign request compression gives you settings similar to what you may set for your web server:

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

These properties allow you to be selective about the compressed media types and minimum request threshold length.

Feign logging

A logger is created for each Feign client created. By default the name of the logger is the full class name of the interface used to create the Feign client. Feign logging only responds to the `DEBUG` level.

application.yml

```
logging.level.project.user.UserClient: DEBUG
```

The `Logger.Level` object that you may configure per client, tells Feign how much to log. Choices are:

- `NONE` , No logging (**DEFAULT**).
- `BASIC` , Log only the request method and URL and the response status code and execution time.
- `HEADERS` , Log the basic information along with request and response headers.
- `FULL` , Log the headers, body, and metadata for both requests and responses.

For example, the following would set the `Logger.Level` to `FULL` :

```
@Configuration
public class FooConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

External Configuration: Archaius

Archaius (https://github.com/Netflix/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 (https://commons.apache.org/proper/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

JAVA

```
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.

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 (https://github.com/Netflix/zuul) is a JVM based router and server side load balancer by Netflix.

Netflix uses Zuul (http://www.slideshare.net/MikeyCohen1/edge-architecture-ieee-international-conference-on-cloud-engineering-32240146/27) 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.

NOTE

The configuration property `zuul.max.host.connections` has been replaced by two new properties, `zuul.host.maxTotalConnections` and `zuul.host.maxPerRouteConnections` which default to 200 and 20 respectively.

NOTE

Default Hystrix isolation pattern (ExecutionIsolationStrategy) for all routes is SEMAPHORE. `zuul.ribbonIsolationStrategy` can be changed to THREAD if this isolation pattern is preferred.

How to Include Zuul

To include Zuul in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-zuul`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

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 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 discovery, 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.

NOTE

the Zuul starter does not include a discovery client, so for routes based on service IDs you need to provide one of those on the classpath as well (e.g. Eureka is one choice).

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

YAML

```
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

YAML
```
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

YAML
```
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 service from discovery) or a "url" (for a physical location), e.g.

application.yml

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

These simple url-routes don't get executed as a `HystrixCommand` nor can you loadbalance multiple URLs 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

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

ribbon:
  eureka:
    enabled: false

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

You can provide convention between serviceId and routes using regexmapper. It uses regular expression named groups to extract variables from serviceId and inject them into a route pattern.

ApplicationConfiguration.java

JAVA
```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper(
        "(?<name>^.+)-(?<version>v.+$)",
        "${version}/${name}");
}
```

This means that a serviceId "myusers-v1" will be mapped to route "/v1/myusers/**". Any regular expression is accepted but all named groups must be present in both servicePattern and routePattern. If servicePattern does not match a serviceId, the default behavior is used. In the example above, a serviceId "myusers" will be mapped to route "/myusers/**" (no version detected) This feature is disable by default and only applies to discovered services.

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
```

YAML

NOTE

`zuul.stripPrefix` only applies to the prefix set in `zuul.prefix`. It does not have any effect on prefixes defined within a given route's `path`.

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 `ZuulProperties`. 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 is 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 `@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.

If more fine-grained ignoring is needed, you can specify specific patterns to ignore. These patterns are evaluated at the start of the route location process, which means prefixes should be included in the pattern to warrant a match. Ignored patterns span all services and supersede any other route specification.

application.yml

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

YAML

This means that all calls such as "/myusers/101" will be forwarded to "/101" on the "users" service. But calls including "/admin/" will not resolve.

WARNING

If you need your routes to have their order preserved you need to use a YAML file as the ordering will be lost using a properties file. For example:

application.yml

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

YAML

If you were to use a properties file, the `legacy` path may end up in front of the `users` path rendering the `users` path unreachable.

Zuul Http Client

The default HTTP client used by zuul is now backed by the Apache HTTP Client instead of the deprecated Ribbon `RestClient`. To use `RestClient` or to use the `okhttp3.OkHttpClient` set `ribbon.restclient.enabled=true` or `ribbon.okhttp.enabled=true` respectively.

Cookies and Sensitive Headers

It's OK to share headers between services in the same system, but you probably don't want sensitive headers leaking downstream into external servers. You can specify a list of ignored headers as part of the route configuration. Cookies play a special role because they have well-defined semantics in browsers, and they are always to be treated as sensitive. If the consumer of your proxy is a browser, then cookies for downstream services also cause problems for the user because they all get jumbled up (all downstream services look like they come from the same place).

If you are careful with the design of your services, for example if only one of the downstream services sets cookies, then you might be able to let them flow from the backend all the way up to the caller. Also, if your proxy sets cookies and all your back end services are part of the same system, it can be natural to simply share them (and for instance use Spring Session to link them up to some shared state). Other than that, any cookies that get set by downstream services are likely to be not very useful to the caller, so it is recommended that you make (at least) "Set-Cookie" and "Cookie" into sensitive headers for routes that are not part of your domain. Even for routes that **are** part of your domain, try to think carefully about what it means before allowing cookies to flow between them and the proxy.

The sensitive headers can be configured as a comma-separated list per route, e.g.

application.yml

YAML

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders: Cookie,Set-Cookie,Authorization
     url: https://downstream
```

NOTE

this is the default value for `sensitiveHeaders`, so you don't need to set it unless you want it to be different. N.B. this is new in Spring Cloud Netflix 1.1 (in 1.0 the user had no control over headers and all cookies flow in both directions).

The `sensitiveHeaders` are a blacklist and the default is not empty, so to make Zuul send all headers (except the "ignored" ones) you would have to explicitly set it to the empty list. This is necessary if you want to pass cookie or authorization headers to your back end. Example:

application.yml

YAML

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders:
     url: https://downstream
```

Sensitive headers can also be set globally by setting `zuul.sensitiveHeaders`. If `sensitiveHeaders` is set on a route, this will override the global `sensitiveHeaders` setting.

Ignored Headers

In addition to the per-route sensitive headers, you can set a global value for `zuul.ignoredHeaders` for values that should be discarded (both request and response) during interactions with downstream services. By default these are empty, if Spring Security is not on the classpath, and otherwise they are initialized to a set of well-known "security" headers (e.g. involving caching) as specified by Spring Security. The assumption in this case is that the downstream services might add these headers too, and we want the values from the proxy. To not discard these well known security headers in case Spring Security is on the classpath you can set `zuul.ignoreSecurityHeaders` to `false`. This can be useful if you disabled the HTTP Security response headers in Spring Security and want the values provided by downstream services

The Routes Endpoint

If you are using `@EnableZuulProxy` with tha Spring Boot Actuator you will enable (by default) an additional endpoint, available via HTTP as `/routes`. A GET to this endpoint will return a list of the mapped routes. A POST will force a refresh of the existing routes (e.g. in case there have been changes in the service catalog). You can disable this endpoint by setting `endpoints.routes.enabled` to `false`.

NOTE

the routes should respond automatically to changes in the service catalog, but the POST to /routes is a way to force the change to happen immediately.

Strangulation Patterns and Local Forwards

A common pattern when migrating an existing application or API is to "strangle" old endpoints, slowly replacing them with different implementations. The Zuul proxy is a useful tool for this because you can use it to handle all traffic from clients of the old endpoints, but redirect some of the requests to new ones.

Example configuration:

application.yml

```
zuul:
 routes:
   first:
     path: /first/**
     url: http://first.example.com
   second:
     path: /second/**
     url: forward:/second
   third:
     path: /third/**
     url: forward:/3rd
   legacy:
     path: /**
     url: http://legacy.example.com
```

YAML

In this example we are strangling the "legacy" app which is mapped to all requests that do not match one of the other patterns. Paths in `/first/**` have been extracted into a new service with an external URL. And paths in `/second/**` are forwarded so they can be handled locally, e.g. with a normal Spring `@RequestMapping`. Paths in `/third/**` are also forwarded, but with a different prefix (i.e. `/third/foo` is forwarded to `/3rd/foo`).

NOTE

The ignored patterns aren't completely ignored, they just aren't handled by the proxy (so they are also effectively forwarded locally).

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
```

YAML

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 "file=@mylarge.iso" localhost:9999/zuul/simple/file
```

Query String Encoding

When processing the incoming request, query params are decoded so they can be available for possible modifications in Zuul filters. They are then re-encoded when building the backend request in the route filters. The result can be different than the original input if it was encoded using Javascript's `encodeURIComponent()` method for example. While this causes no issues in most cases, some web servers can be picky with the encoding of complex query string.

To force the original encoding of the query string, it is possible to pass a special flag to `ZuulProperties` so that the query string is taken as is with the `HttpServletRequest::getQueryString` method :

application.yml

```
zuul:
 forceOriginalQueryStringEncoding: true
```

YAML

Note: This special flag only works with `SimpleHostRoutingFilter` and you loose the ability to easily override query parameters with `RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters)` since the query string is now fetched directly on the original `HttpServletRequest`.

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/**
```

YAML

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 (https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters) 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`.

Providing Hystrix Fallbacks For Routes

When a circuit for a given route in Zuul is tripped you can provide a fallback response by creating a bean of type `ZuulFallbackProvider`. Within this bean you need to specify the route ID the fallback is for and provide a `ClientHttpResponse` to return as a fallback. Here is a very simple `ZuulFallbackProvider` implementation.

JAVA

```
class MyFallbackProvider implements ZuulFallbackProvider {
    @Override
    public String getRoute() {
        return "customers";
    }

    @Override
    public ClientHttpResponse fallbackResponse() {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

And here is what the route configuration would look like.

YAML

```
zuul:
  routes:
    customers: /customers/**
```

If you would like to provide a default fallback for all routes than you can create a bean of type `ZuulFallbackProvider` and have the `getRoute` method return `*` or `null`.

JAVA

```
class MyFallbackProvider implements ZuulFallbackProvider {
    @Override
    public String getRoute() {
        return "*";
    }

    @Override
    public ClientHttpResponse fallbackResponse() {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

Zuul Developer Guide

For a general overview of how Zuul works, please see the Zuul Wiki (https://github.com/Netflix/zuul/wiki/How-it-Works).

The Zuul Servlet

Zuul is implemented as a Servlet. For the general cases, Zuul is embedded into the Spring Dispatch mechanism. This allows Spring MVC to be in control of the routing. In this case, Zuul is configured to buffer requests. If there is a need to go through Zuul without buffering requests (e.g. for large file uploads), the Servlet is also installed outside of the Spring Dispatcher. By default, this is located at `/zuul`. This path can be changed with the `zuul.servlet-path` property.

Zuul RequestContext

To pass information between filters, Zuul uses a `RequestContext` (https://github.com/Netflix/zuul/blob/1.x/zuul-core/src/main/java/com/netflix/zuul/context/RequestContext.java). Its data is held in a `ThreadLocal` specific to each request. Information about where to route requests, errors and the actual `HttpServletRequest` and `HttpServletResponse` are stored there. The `RequestContext` extends `ConcurrentHashMap`, so anything can be stored in the context. `FilterConstants` (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters/support/FilterConstants.java) contains the keys that are used by the filters installed by Spring Cloud Netflix (more on these later).

`@EnableZuulProxy` vs. `@EnableZuulServer`

Spring Cloud Netflix installs a number of filters based on which annotation was used to enable Zuul. `@EnableZuulProxy` is a superset of `@EnableZuulServer`. In other words, `@EnableZuulProxy` contains all filters installed by `@EnableZuulServer`. The additional filters in the "proxy" enable routing functionality. If you want a "blank" Zuul, you should use `@EnableZuulServer`.

`@EnableZuulServer` Filters

Creates a `SimpleRouteLocator` that loads route definitions from Spring Boot configuration files.

The following filters are installed (as normal Spring Beans):

Pre filters:

- `ServletDetectionFilter` : Detects if the request is through the Spring Dispatcher. Sets boolean with key `FilterConstants.IS_DISPATCHER_SERVLET_REQUEST_KEY` .
- `FormBodyWrapperFilter` : Parses form data and reencodes it for downstream requests.
- `DebugFilter` : if the `debug` request parameter is set, this filter sets `RequestContext.setDebugRouting()` and `RequestContext.setDebugRequest()` to true.

Route filters:

- `SendForwardFilter` : This filter forwards requests using the Servlet `RequestDispatcher` . The forwarding location is stored in the `RequestContext` attribute `FilterConstants.FORWARD_TO_KEY` . This is useful for forwarding to endpoints in the current application.

Post filters:

- `SendResponseFilter` : Writes responses from proxied requests to the current response.

Error filters:

- `SendErrorFilter` : Forwards to /error (by default) if `RequestContext.getThrowable()` is not null. The default forwarding path (`/error`) can be changed by setting the `error.path` property.

`@EnableZuulProxy` Filters

Creates a `DiscoveryClientRouteLocator` that loads route definitions from a `DiscoveryClient` (like Eureka), as well as from properties. A route is created for each `serviceId` from the `DiscoveryClient` . As new services are added, the routes will be refreshed.

In addition to the filters described above, the following filters are installed (as normal Spring Beans):

Pre filters:

- `PreDecorationFilter` : This filter determines where and how to route based on the supplied `RouteLocator` . It also sets various proxy-related headers for downstream requests.

Route filters:

- `RibbonRoutingFilter` : This filter uses Ribbon, Hystrix and pluggable HTTP clients to send requests. Service ids are found in the `RequestContext` attribute `FilterConstants.SERVICE_ID_KEY` . This filter can use different HTTP clients. They are:
  - Apache `HttpClient` . This is the default client.
  - Squareup `OkHttpClient` v3. This is enabled by having the `com.squareup.okhttp3:okhttp` library on the classpath and setting `ribbon.okhttp.enabled=true` .
  - Netflix Ribbon HTTP client. This is enabled by setting `ribbon.restclient.enabled=true` . This client has limitations, such as it doesn't support the PATCH method, but also has built-in retry.
- `SimpleHostRoutingFilter` : This filter sends requests to predetermined URLs via an Apache HttpClient. URLs are found in `RequestContext.getRouteHost()` .

Custom Zuul Filter examples

Most of the following "How to Write" examples below are included Sample Zuul Filters (https://github.com/spring-cloud-samples/sample-zuul-filters) project. There are also examples of manipulating the request or response body in that repository.

How to Write a Pre Filter

Pre filters are used to set up data in the `RequestContext` for use in filters downstream. The main use case is to set information required for route filters.

JAVA

```
public class QueryParamPreFilter extends ZuulFilter {
    @Override
    public int filterOrder() {
        return PRE_DECORATION_FILTER_ORDER - 1; // run before PreDecoration
    }

    @Override
    public String filterType() {
        return PRE_TYPE;
    }

    @Override
    public boolean shouldFilter() {
        RequestContext ctx = RequestContext.getCurrentContext();
        return !ctx.containsKey(FORWARD_TO_KEY) // a filter has already forwarded
                && !ctx.containsKey(SERVICE_ID_KEY); // a filter has already determined serviceId
    }
    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        if (request.getParameter("foo") != null) {
            // put the serviceId in `RequestContext`
            ctx.put(SERVICE_ID_KEY, request.getParameter("foo"));
        }
        return null;
    }
}
```

The filter above populates `SERVICE_ID_KEY` from the `foo` request parameter. In reality, it's not a good idea to do that kind of direct mapping, but the service id should be looked up from the value of `foo` instead.

Now that `SERVICE_ID_KEY` is populated, `PreDecorationFilter` won't run and `RibbonRoutingFilter` will. If you wanted to route to a full URL instead, call `ctx.setRouteHost(url)` instead.

To modify the path that routing filters will forward to, set the `REQUEST_URI_KEY` .

How to Write a Route Filter

Route filters are run after pre filters and are used to make requests to other services. Much of the work here is to translate request and response data to and from the client required model.

JAVA

```
public class OkHttpRoutingFilter extends ZuulFilter {
    @Autowired
    private ProxyRequestHelper helper;

    @Override
    public String filterType() {
        return ROUTE_TYPE;
    }

    @Override
    public int filterOrder() {
        return SIMPLE_HOST_ROUTING_FILTER_ORDER - 1;
    }

    @Override
    public boolean shouldFilter() {
        return RequestContext.getCurrentContext().getRouteHost() != null
                && RequestContext.getCurrentContext().sendZuulResponse();
    }

    @Override
    public Object run() {
        OkHttpClient httpClient = new OkHttpClient.Builder()
                // customize
                .build();

        RequestContext context = RequestContext.getCurrentContext();
        HttpServletRequest request = context.getRequest();

        String method = request.getMethod();

        String uri = this.helper.buildZuulRequestURI(request);

        Headers.Builder headers = new Headers.Builder();
        Enumeration<String> headerNames = request.getHeaderNames();
        while (headerNames.hasMoreElements()) {
            String name = headerNames.nextElement();
            Enumeration<String> values = request.getHeaders(name);

            while (values.hasMoreElements()) {
                String value = values.nextElement();
                headers.add(name, value);
            }
        }

        InputStream inputStream = request.getInputStream();

        RequestBody requestBody = null;
        if (inputStream != null && HttpMethod.permitsRequestBody(method)) {
            MediaType mediaType = null;
            if (headers.get("Content-Type") != null) {
                mediaType = MediaType.parse(headers.get("Content-Type"));
            }
            requestBody = RequestBody.create(mediaType, StreamUtils.copyToByteArray(inputStream));
        }

        Request.Builder builder = new Request.Builder()
                .headers(headers.build())
                .url(uri)
                .method(method, requestBody);

        Response response = httpClient.newCall(builder.build()).execute();

        LinkedMultiValueMap<String, String> responseHeaders = new LinkedMultiValueMap<>();

        for (Map.Entry<String, List<String>> entry : response.headers().toMultimap().entrySet()) {
            responseHeaders.put(entry.getKey(), entry.getValue());
        }

        this.helper.setResponse(response.code(), response.body().byteStream(),
                responseHeaders);
        context.setRouteHost(null); // prevent SimpleHostRoutingFilter from running
        return null;
    }
}
```

The above filter translates Servlet request information into OkHttp3 request information, executes an HTTP request, then translates OkHttp3 reponse information to the Servlet response. WARNING: this filter might have bugs and not function correctly.

How to Write a Post Filter

Post filters typically manipulate the response. In the filter below, we add a random `UUID` as the `X-Foo` header. Other manipulations, such as transforming the response body, are much more complex and compute-intensive.

JAVA

```
public class AddResponseHeaderFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return POST_TYPE;
    }

    @Override
    public int filterOrder() {
        return SEND_RESPONSE_FILTER_ORDER - 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext context = RequestContext.getCurrentContext();
        HttpServletResponse servletResponse = context.getResponse();
        servletResponse.addHeader("X-Foo", UUID.randomUUID().toString());
        return null;
    }
}
```

How Zuul Errors Work

If an exception is thrown during any portion of the Zuul filter lifecycle, the error filters are executed. The `SendErrorFilter` is only run if `RequestContext.getThrowable()` is not `null`. It then sets specific `javax.servlet.error.*` attributes in the request and forwards the request to the Spring Boot error page.

Zuul Eager Application Context Loading

Zuul internally uses Ribbon for calling the remote url's and Ribbon clients are by default lazily loaded up by Spring Cloud on first call. This behavior can be changed for Zuul using the following configuration and will result in the child Ribbon related Application contexts being eagerly loaded up at application startup time.

application.yml

```
zuul:
  ribbon:
    eager-load:
      enabled: 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 (https://github.com/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 include Sidecar in your project use the dependency with group `org.springframework.cloud` and artifact id `spring-cloud-netflix-sidecar`.

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

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

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

application.yml

```YAML
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}`.

/hosts/customers

```JSON
[
    {
        "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 `/<serviceId>`, so the customers service is available at `/customers`. The Non-jvm app can access the customer service via `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

YAML

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

RxJava with Spring MVC

Spring Cloud Netflix includes RxJava (https://github.com/ReactiveX/RxJava).

> *RxJava is a Java VM implementation of Reactive Extensions (http://reactivex.io/): a library for composing asynchronous and event-based programs by using observable sequences.*

Spring Cloud Netflix provides support for returning `rx.Single` objects from Spring MVC Controllers. It also supports using `rx.Observable` objects for Server-sent events (SSE) (https://en.wikipedia.org/wiki/Server-sent_events). This can be very convenient if your internal APIs are already built using RxJava (see Feign Hystrix Support for examples).

Here are some examples of using `rx.Single`:

JAVA

```
@RequestMapping(method = RequestMethod.GET, value = "/single")
public Single<String> single() {
    return Single.just("single value");
}

@RequestMapping(method = RequestMethod.GET, value = "/singleWithResponse")
public ResponseEntity<Single<String>> singleWithResponse() {
    return new ResponseEntity<>(Single.just("single value"),
            HttpStatus.NOT_FOUND);
}

@RequestMapping(method = RequestMethod.GET, value = "/singleCreatedWithResponse")
public Single<ResponseEntity<String>> singleOuterWithResponse() {
    return Single.just(new ResponseEntity<>("single value", HttpStatus.CREATED));
}

@RequestMapping(method = RequestMethod.GET, value = "/throw")
public Single<Object> error() {
    return Single.error(new RuntimeException("Unexpected"));
}
```

If you have an `Observable`, rather than a single, you can use `.toSingle()` or `.toList().toSingle()`. Here are some examples:

JAVA

```
@RequestMapping(method = RequestMethod.GET, value = "/single")
public Single<String> single() {
    return Observable.just("single value").toSingle();
}

@RequestMapping(method = RequestMethod.GET, value = "/multiple")
public Single<List<String>> multiple() {
    return Observable.just("multiple", "values").toList().toSingle();
}

@RequestMapping(method = RequestMethod.GET, value = "/responseWithObservable")
public ResponseEntity<Single<String>> responseWithObservable() {

    Observable<String> observable = Observable.just("single value");
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(APPLICATION_JSON_UTF8);
    return new ResponseEntity<>(observable.toSingle(), headers, HttpStatus.CREATED);
}

@RequestMapping(method = RequestMethod.GET, value = "/timeout")
public Observable<String> timeout() {
    return Observable.timer(1, TimeUnit.MINUTES).map(new Func1<Long, String>() {
        @Override
        public String call(Long aLong) {
            return "single value";
        }
    });
}
```

If you have a streaming endpoint and client, SSE could be an option. To convert `rx.Observable` to a Spring `SseEmitter` use `RxResponse.sse()`. Here are some examples:

JAVA

```
@RequestMapping(method = RequestMethod.GET, value = "/sse")
public SseEmitter single() {
    return RxResponse.sse(Observable.just("single value"));
}

@RequestMapping(method = RequestMethod.GET, value = "/messages")
public SseEmitter messages() {
    return RxResponse.sse(Observable.just("message 1", "message 2", "message 3"));
}

@RequestMapping(method = RequestMethod.GET, value = "/events")
public SseEmitter event() {
    return RxResponse.sse(APPLICATION_JSON_UTF8,
            Observable.just(new EventDto("Spring io", getDate(2016, 5, 19)),
                    new EventDto("SpringOnePlatform", getDate(2016, 8, 1))));
}
```

Metrics: Spectator, Servo, and Atlas

When used together, Spectator/Servo and Atlas provide a near real-time operational insight platform.

Spectator and Servo are Netflix's metrics collection libraries. Atlas is a Netflix metrics backend to manage dimensional time series data.

Servo served Netflix for several years and is still usable, but is gradually being phased out in favor of Spectator, which is only designed to work with Java 8. Spring Cloud Netflix provides support for both, but Java 8 based applications are encouraged to use Spectator.

Dimensional vs. Hierarchical Metrics

Spring Boot Actuator metrics are hierarchical and metrics are separated only by name. These names often follow a naming convention that embeds key/value attribute pairs (dimensions) into the name separated by periods. Consider the following metrics for two endpoints, root and star-star:

JSON

```
{
    "counter.status.200.root": 20,
    "counter.status.400.root": 3,
    "counter.status.200.star-star": 5,
}
```

The first metric gives us a normalized count of successful requests against the root endpoint per unit of time. But what if the system had 20 endpoints and you want to get a count of successful requests against all the endpoints? Some hierarchical metrics backends would allow you to specify a wild card such as `counter.status.200.` **that would read all 20 metrics and aggregate the results. Alternatively, you could provide a** `HandlerInterceptorAdapter` **that intercepts and records a metric like** `counter.status.200.all` **for all successful requests irrespective of the endpoint, but now you must write 20+1 different metrics. Similarly if you want to know the total number of successful requests for all endpoints in the service, you could specify a wild card such as** `counter.status.2 .*`.

Even in the presence of wildcarding support on a hierarchical metrics backend, naming consistency can be difficult. Specifically the position of these tags in the name string can slip with time, breaking queries. For example, suppose we add an additional dimension to the hierarchical metrics above for HTTP method. Then `counter.status.200.root` becomes `counter.status.200.method.get.root`, etc. Our `counter.status.200.*` suddenly no longer has the same semantic meaning. Furthermore, if the new dimension is not applied uniformly across the codebase, certain queries may become impossible. This can quickly get out of hand.

Netflix metrics are tagged (a.k.a. dimensional). Each metric has a name, but this single named metric can contain multiple statistics and 'tag' key/value pairs that allows more querying flexibility. In fact, the statistics themselves are recorded in a special tag.

Recorded with Netflix Servo or Spectator, a timer for the root endpoint described above contains 4 statistics per status code, where the count statistic is identical to Spring Boot Actuator's counter. In the event that we have encountered an HTTP 200 and 400 thus far, there will be 8 available data points:

JSON

```
{
    "root(status=200,stastic=count)": 20,
    "root(status=200,stastic=max)": 0.7265630630000001,
    "root(status=200,stastic=totalOfSquares)": 0.04759702862580789,
    "root(status=200,stastic=totalTime)": 0.2093076914666667,
    "root(status=400,stastic=count)": 1,
    "root(status=400,stastic=max)": 0,
    "root(status=400,stastic=totalOfSquares)": 0,
    "root(status=400,stastic=totalTime)": 0,
}
```

Default Metrics Collection

Without any additional dependencies or configuration, a Spring Cloud based service will autoconfigure a Servo `MonitorRegistry` and begin collecting metrics on every Spring MVC request. By default, a Servo timer with the name `rest` will be recorded for each MVC request which is tagged with:

1. HTTP method
2. HTTP status (e.g. 200, 400, 500)

3. URI (or "root" if the URI is empty), sanitized for Atlas
4. The exception class name, if the request handler threw an exception
5. The caller, if a request header with a key matching `netflix.metrics.rest.callerHeader` is set on the request. There is no default key for `netflix.metrics.rest.callerHeader`. You must add it to your application properties if you wish to collect caller information.

Set the `netflix.metrics.rest.metricName` property to change the name of the metric from `rest` to a name you provide.

If Spring AOP is enabled and `org.aspectj:aspectjweaver` is present on your runtime classpath, Spring Cloud will also collect metrics on every client call made with `RestTemplate`. A Servo timer with the name of `restclient` will be recorded for each MVC request which is tagged with:

1. HTTP method
2. HTTP status (e.g. 200, 400, 500), "CLIENT_ERROR" if the response returned null, or "IO_ERROR" if an `IOException` occurred during the execution of the `RestTemplate` method
3. URI, sanitized for Atlas
4. Client name

WARNING	Avoid using hardcoded url parameters within `RestTemplate`. When targeting dynamic endpoints use URL variables. This will avoid potential "GC Overhead Limit Reached" issues where `ServoMonitorCache` treats each url as a unique key.

JAVA
```
// recommended
String orderid = "1";
restTemplate.getForObject("http://testeurekabrixtonclient/orders/{orderid}", String.class, orderid)

// avoid
restTemplate.getForObject("http://testeurekabrixtonclient/orders/1", String.class)
```

Metrics Collection: Spectator

To enable Spectator metrics, include a dependency on `spring-boot-starter-spectator`:

XML
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-spectator</artifactId>
</dependency>
```

In Spectator parlance, a meter is a named, typed, and tagged configuration and a metric represents the value of a given meter at a point in time. Spectator meters are created and controlled by a registry, which currently has several different implementations. Spectator provides 4 meter types: counter, timer, gauge, and distribution summary.

Spring Cloud Spectator integration configures an injectable `com.netflix.spectator.api.Registry` instance for you. Specifically, it configures a `ServoRegistry` instance in order to unify the collection of REST metrics and the exporting of metrics to the Atlas backend under a single Servo API. Practically, this means that your code may use a mixture of Servo monitors and Spectator meters and both will be scooped up by Spring Boot Actuator `MetricReader` instances and both will be shipped to the Atlas backend.

Spectator Counter

A counter is used to measure the rate at which some event is occurring.

JAVA
```
// create a counter with a name and a set of tags
Counter counter = registry.counter("counterName", "tagKey1", "tagValue1", ...);
counter.increment(); // increment when an event occurs
counter.increment(10); // increment by a discrete amount
```

The counter records a single time-normalized statistic.

Spectator Timer

A timer is used to measure how long some event is taking. Spring Cloud automatically records timers for Spring MVC requests and conditionally `RestTemplate` requests, which can later be used to create dashboards for request related metrics like latency:

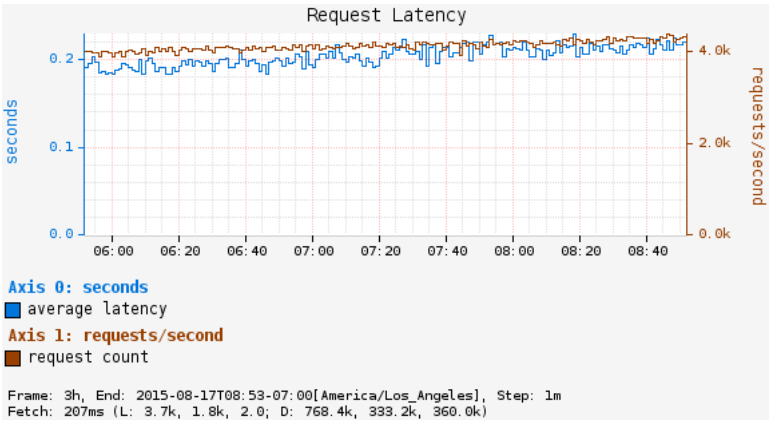


Figure 4. Request Latency

JAVA

```
// create a timer with a name and a set of tags
Timer timer = registry.timer("timerName", "tagKey1", "tagValue1", ...);

// execute an operation and time it at the same time
T result = timer.record(() -> fooReturnsT());

// alternatively, if you must manually record the time
Long start = System.nanoTime();
T result = fooReturnsT();
timer.record(System.nanoTime() - start, TimeUnit.NANOSECONDS);
```

The timer simultaneously records 4 statistics: count, max, totalOfSquares, and totalTime. The count statistic will always match the single normalized value provided by a counter if you had called `increment()` once on the counter for each time you recorded a timing, so it is rarely necessary to count and time separately for a single operation.

For long running operations (https://github.com/Netflix/spectator/wiki/Timer-Usage#longtasktimer), Spectator provides a special `LongTaskTimer`.

Spectator Gauge

Gauges are used to determine some current value like the size of a queue or number of threads in a running state. Since gauges are sampled, they provide no information about how these values fluctuate between samples.

The normal use of a gauge involves registering the gauge once in initialization with an id, a reference to the object to be sampled, and a function to get or compute a numeric value based on the object. The reference to the object is passed in separately and the Spectator registry will keep a weak reference to the object. If the object is garbage collected, then Spectator will automatically drop the registration. See the note (https://github.com/Netflix/spectator/wiki/Gauge-Usage#using-lambda) in Spectator's documentation about potential memory leaks if this API is misused.

JAVA

```
// the registry will automatically sample this gauge periodically
registry.gauge("gaugeName", pool, Pool::numberOfRunningThreads);

// manually sample a value in code at periodic intervals -- last resort!
registry.gauge("gaugeName", Arrays.asList("tagKey1", "tagValue1", ...), 1000);
```

Spectator Distribution Summaries

A distribution summary is used to track the distribution of events. It is similar to a timer, but more general in that the size does not have to be a period of time. For example, a distribution summary could be used to measure the payload sizes of requests hitting a server.

JAVA

```
// the registry will automatically sample this gauge periodically
DistributionSummary ds = registry.distributionSummary("dsName", "tagKey1", "tagValue1", ...);
ds.record(request.sizeInBytes());
```

Metrics Collection: Servo

WARNING If your code is compiled on Java 8, please use Spectator instead of Servo as Spectator is destined to replace Servo entirely in the long term.

In Servo parlance, a monitor is a named, typed, and tagged configuration and a metric represents the value of a given monitor at a point in time. Servo monitors are logically equivalent to Spectator meters. Servo monitors are created and controlled by a `MonitorRegistry`. In spite of the above warning, Servo does have a wider array (https://github.com/Netflix/servo/wiki/Getting-Started) of monitor options than Spectator has meters.

Spring Cloud integration configures an injectable `com.netflix.servo.MonitorRegistry` instance for you. Once you have created the appropriate `Monitor` type in Servo, the process of recording data is wholly similar to Spectator.

Creating Servo Monitors

If you are using the Servo `MonitorRegistry` instance provided by Spring Cloud (specifically, an instance of `DefaultMonitorRegistry`), Servo provides convenience classes for retrieving counters (https://github.com/Netflix/spectator/wiki/Servo-Comparison#dynamiccounter) and timers (https://github.com/Netflix/spectator/wiki/Servo-Comparison#dynamictimer). These convenience classes ensure that only one `Monitor` is registered for each unique combination of name and tags.

To manually create a Monitor type in Servo, especially for the more exotic monitor types for which convenience methods are not provided, instantiate the appropriate type by providing a `MonitorConfig` instance:

JAVA
```
MonitorConfig config = MonitorConfig.builder("timerName").withTag("tagKey1", "tagValue1").build();

// somewhere we should cache this Monitor by MonitorConfig
Timer timer = new BasicTimer(config);
monitorRegistry.register(timer);
```

Metrics Backend: Atlas

Atlas was developed by Netflix to manage dimensional time series data for near real-time operational insight. Atlas features in-memory data storage, allowing it to gather and report very large numbers of metrics, very quickly.

Atlas captures operational intelligence. Whereas business intelligence is data gathered for analyzing trends over time, operational intelligence provides a picture of what is currently happening within a system.

Spring Cloud provides a `spring-cloud-starter-atlas` that has all the dependencies you need. Then just annotate your Spring Boot application with `@EnableAtlas` and provide a location for your running Atlas server with the `netflix.atlas.uri` property.

Global tags

Spring Cloud enables you to add tags to every metric sent to the Atlas backend. Global tags can be used to separate metrics by application name, environment, region, etc.

Each bean implementing `AtlasTagProvider` will contribute to the global tag list:

JAVA
```
@Bean
AtlasTagProvider atlasCommonTags(
    @Value("${spring.application.name}") String appName) {
  return () -> Collections.singletonMap("app", appName);
}
```

Using Atlas

To bootstrap a in-memory standalone Atlas instance:

BASH
```
$ curl -LO https://github.com/Netflix/atlas/releases/download/v1.4.2/atlas-1.4.2-standalone.jar
$ java -jar atlas-1.4.2-standalone.jar
```

> **TIP** An Atlas standalone node running on an r3.2xlarge (61GB RAM) can handle roughly 2 million metrics per minute for a given 6 hour window.

Once running and you have collected a handful of metrics, verify that your setup is correct by listing tags on the Atlas server:

BASH
```
$ curl http://ATLAS/api/v1/tags
```

TIP

After executing several requests against your service, you can gather some very basic information on the request latency of every request by pasting the following url in your browser: `http://ATLAS/api/v1/graph?q=name,rest,:eq,:avg`

The Atlas wiki contains a compilation of sample queries (https://github.com/Netflix/atlas/wiki/Single-Line) for various scenarios.

Make sure to check out the alerting philosophy (https://github.com/Netflix/atlas/wiki/Alerting-Philosophy) and docs on using double exponential smoothing (https://github.com/Netflix/atlas/wiki/DES) to generate dynamic alert thresholds.

Retrying Failed Requests

Spring Cloud Netflix offers a variety of ways to make HTTP requests. You can use a load balanced `RestTemplate`, Ribbon, or Feign. No matter how you choose to your HTTP requests, there is always a chance the request may fail. When a request fails you may want to have the request retried automatically. To accomplish this when using Sping Cloud Netflix you need to include Spring Retry (https://github.com/spring-projects/spring-retry) on your application's classpath. When Spring Retry is present load balanced `RestTemplates`, Feign, and Zuul will automatically retry any failed requests (assuming you configuration allows it to).

Configuration

Anytime Ribbon is used with Spring Retry you can control the retry functionality by configuring certain Ribbon properties. The properties you can use are `client.ribbon.MaxAutoRetries`, `client.ribbon.MaxAutoRetriesNextServer`, and `client.ribbon.OkToRetryOnAllOperations`. See the Ribbon documentation (https://github.com/Netflix/ribbon/wiki/Getting-Started#the-properties-file-sample-clientproperties) for a description of what there properties do.

In addition you may want to retry requests when certain status codes are returned in the response. You can list the response codes you would like the Ribbon client to retry using the property `clientName.ribbon.retryableStatusCodes`. For example

YAML

```
clientName:
  ribbon:
    retryableStatusCodes: 404,502
```

You can also create a bean of type `LoadBalancedRetryPolicy` and implement the `retryableStatusCode` method to determine whether you want to retry a request given the status code.

Zuul

You can turn off Zuul's retry functionality by setting `zuul.retryable` to `false`. You can also disable retry functionality on route by route basis by setting `zuul.routes.routename.retryable` to `false`.

Spring Cloud Stream

This section goes into more detail about how you can work with Spring Cloud Stream. It covers topics such as creating and running stream applications.

Introducing Spring Cloud Stream

Spring Cloud Stream is a framework for building message-driven microservice applications. Spring Cloud Stream builds upon Spring Boot to create standalone, production-grade Spring applications, and uses Spring Integration to provide connectivity to message brokers. It provides opinionated configuration of middleware from several vendors, introducing the concepts of persistent publish-subscribe semantics, consumer groups, and partitions.

You can add the `@EnableBinding` annotation to your application to get immediate connectivity to a message broker, and you can add `@StreamListener` to a method to cause it to receive events for stream processing. The following is a simple sink application which receives external messages.

JAVA

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class VoteRecordingSinkApplication {

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

  @StreamListener(Sink.INPUT)
  public void processVote(Vote vote) {
      votingService.recordVote(vote);
  }
}
```

The `@EnableBinding` annotation takes one or more interfaces as parameters (in this case, the parameter is a single `Sink` interface). An interface declares input and/or output channels. Spring Cloud Stream provides the interfaces `Source`, `Sink`, and `Processor`; you can also define your own interfaces.

The following is the definition of the `Sink` interface:

JAVA

```
public interface Sink {
  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();
}
```

The `@Input` annotation identifies an *input channel*, through which received messages enter the application; the `@Output` annotation identifies an *output channel*, through which published messages leave the application. The `@Input` and `@Output` annotations can take a channel name as a parameter; if a name is not provided, the name of the annotated method will be used.

Spring Cloud Stream will create an implementation of the interface for you. You can use this in the application by autowiring it, as in the following example of a test case.

JAVA

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = VoteRecordingSinkApplication.class)
@WebAppConfiguration
@DirtiesContext
public class StreamApplicationTests {

  @Autowired
  private Sink sink;

  @Test
  public void contextLoads() {
    assertNotNull(this.sink.input());
  }
}
```

Main Concepts

Spring Cloud Stream provides a number of abstractions and primitives that simplify the writing of message-driven microservice applications. This section gives an overview of the following:

- Spring Cloud Stream's application model
- The Binder abstraction
- Persistent publish-subscribe support
- Consumer group support
- Partitioning support
- A pluggable Binder API

Application Model

A Spring Cloud Stream application consists of a middleware-neutral core. The application communicates with the outside world through input and output *channels* injected into it by Spring Cloud Stream. Channels are connected to external brokers through middleware-specific Binder implementations.

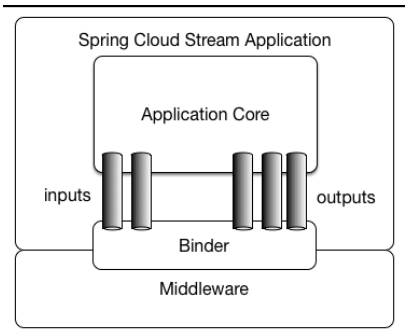


Figure 5. Spring Cloud Stream Application

Fat JAR

Spring Cloud Stream applications can be run in standalone mode from your IDE for testing. To run a Spring Cloud Stream application in production, you can create an executable (or "fat") JAR by using the standard Spring Boot tooling provided for Maven or Gradle.

The Binder Abstraction

Spring Cloud Stream provides Binder implementations for Kafka (https://github.com/spring-cloud/spring-cloud-stream/tree/master/spring-cloud-stream-binders/spring-cloud-stream-binder-kafka) and Rabbit MQ (https://github.com/spring-cloud/spring-cloud-stream/tree/master/spring-cloud-stream-binders/spring-cloud-stream-binder-rabbit). Spring Cloud Stream also includes a TestSupportBinder (https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream-test-support/src/main/java/org/springframework/cloud/stream/test/binder/TestSupportBinder.java), which leaves a channel unmodified so that tests can interact with channels directly and reliably assert on what is received. You can use the extensible API to write your own Binder.

Spring Cloud Stream uses Spring Boot for configuration, and the Binder abstraction makes it possible for a Spring Cloud Stream application to be flexible in how it connects to middleware. For example, deployers can dynamically choose, at runtime, the destinations (e.g., the Kafka topics or RabbitMQ exchanges) to which channels connect. Such configuration can be provided through external configuration properties and in any form supported by Spring Boot (including application arguments, environment variables, and `application.yml` or `application.properties` files). In the sink example from the Introducing Spring Cloud Stream section, setting the application property `spring.cloud.stream.bindings.input.destination` to `raw-sensor-data` will cause it to read from the `raw-sensor-data` Kafka topic, or from a queue bound to the `raw-sensor-data` RabbitMQ exchange.

Spring Cloud Stream automatically detects and uses a binder found on the classpath. You can easily use different types of middleware with the same code: just include a different binder at build time. For more complex use cases, you can also package multiple binders with your application and have it choose the binder, and even whether to use different binders for different channels, at runtime.

Persistent Publish-Subscribe Support

Communication between applications follows a publish-subscribe model, where data is broadcast through shared topics. This can be seen in the following figure, which shows a typical deployment for a set of interacting Spring Cloud Stream applications.

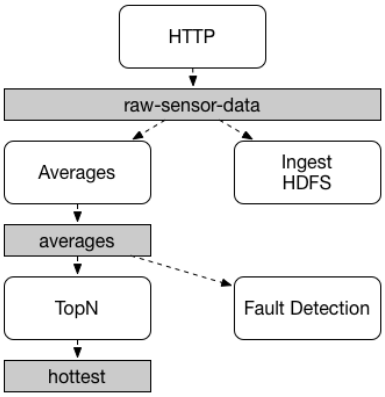


Figure 6. Spring Cloud Stream Publish-Subscribe

Data reported by sensors to an HTTP endpoint is sent to a common destination named `raw-sensor-data`. From the destination, it is independently processed by a microservice application that computes time-windowed averages and by another microservice application that ingests the raw data into HDFS. In order to process the data, both applications declare the topic as their input at runtime.

The publish-subscribe communication model reduces the complexity of both the producer and the consumer, and allows new applications to be added to the topology without disruption of the existing flow. For example, downstream from the average-calculating application, you can add an application that calculates the highest temperature values for display and monitoring. You can then add another application that interprets the same flow of averages for fault detection. Doing all communication through shared topics rather than point-to-point queues reduces coupling between microservices.

While the concept of publish-subscribe messaging is not new, Spring Cloud Stream takes the extra step of making it an opinionated choice for its application model. By using native middleware support, Spring Cloud Stream also simplifies use of the publish-subscribe model across different platforms.

Consumer Groups

While the publish-subscribe model makes it easy to connect applications through shared topics, the ability to scale up by creating multiple instances of a given application is equally important. When doing this, different instances of an application are placed in a competing consumer relationship, where only one of the instances is expected to handle a given message.

Spring Cloud Stream models this behavior through the concept of a *consumer group*. (Spring Cloud Stream consumer groups are similar to and inspired by Kafka consumer groups.) Each consumer binding can use the `spring.cloud.stream.bindings.<channelName>.group` property to specify a group name. For the consumers shown in the following figure, this property would be set as `spring.cloud.stream.bindings.<channelName>.group=hdfsWrite` or `spring.cloud.stream.bindings.<channelName>.group=average`.

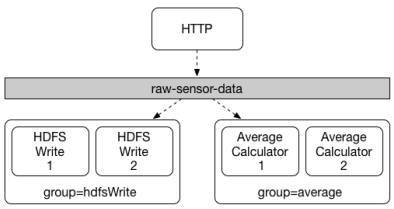


Figure 7. Spring Cloud Stream Consumer Groups

All groups which subscribe to a given destination receive a copy of published data, but only one member of each group receives a given message from that destination. By default, when a group is not specified, Spring Cloud Stream assigns the application to an anonymous and independent single-member consumer group that is in a publish-subscribe relationship with all other consumer groups.

Durability

Consistent with the opinionated application model of Spring Cloud Stream, consumer group subscriptions are *durable*. That is, a binder implementation ensures that group subscriptions are persistent, and once at least one subscription for a group has been created, the group will receive messages, even if they are sent while all applications in the group are stopped.

NOTE

Anonymous subscriptions are non-durable by nature. For some binder implementations (e.g., RabbitMQ), it is possible to have non-durable group subscriptions.

In general, it is preferable to always specify a consumer group when binding an application to a given destination. When scaling up a Spring Cloud Stream application, you must specify a consumer group for each of its input bindings. This prevents the application's instances from receiving duplicate messages (unless that behavior is desired, which is unusual).

Partitioning Support

Spring Cloud Stream provides support for *partitioning* data between multiple instances of a given application. In a partitioned scenario, the physical communication medium (e.g., the broker topic) is viewed as being structured into multiple partitions. One or more producer application instances send data to multiple consumer application instances and ensure that data identified by common characteristics are processed by the same consumer instance.

Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion. Partitioning can thus be used whether the broker itself is naturally partitioned (e.g., Kafka) or not (e.g., RabbitMQ).

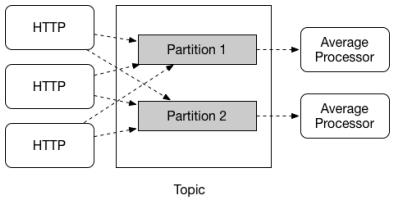


Figure 8. Spring Cloud Stream Partitioning

Partitioning is a critical concept in stateful processing, where it is critiical, for either performance or consistency reasons, to ensure that all related data is processed together. For example, in the time-windowed average calculation example, it is important that all measurements from any given sensor are processed by the same application instance.

NOTE

To set up a partitioned processing scenario, you must configure both the data-producing and the data-consuming ends.

Programming Model

This section describes Spring Cloud Stream's programming model. Spring Cloud Stream provides a number of predefined annotations for declaring bound input and output channels as well as how to listen to channels.

Declaring and Binding Channels

Triggering Binding Via `@EnableBinding`

You can turn a Spring application into a Spring Cloud Stream application by applying the `@EnableBinding` annotation to one of the application's configuration classes. The `@EnableBinding` annotation itself is meta-annotated with `@Configuration` and triggers the configuration of Spring Cloud Stream infrastructure:

JAVA

```
...
@Import(...)
@Configuration
@EnableIntegration
public @interface EnableBinding {
    ...
    Class<?>[] value() default {};
}
```

The `@EnableBinding` annotation can take as parameters one or more interface classes that contain methods which represent bindable components (typically message channels).

NOTE

In Spring Cloud Stream 1.0, the only supported bindable components are the Spring Messaging `MessageChannel` and its extensions `SubscribableChannel` and `PollableChannel`. Future versions should extend this support to other types of components, using the same mechanism. In this documentation, we will continue to refer to channels.

`@Input` and `@Output`

A Spring Cloud Stream application can have an arbitrary number of input and output channels defined in an interface as `@Input` and `@Output` methods:

JAVA

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

Using this interface as a parameter to `@EnableBinding` will trigger the creation of three bound channels named `orders`, `hotDrinks`, and `coldDrinks`, respectively.

JAVA

```
@EnableBinding(Barista.class)
public class CafeConfiguration {

   ...
}
```

Customizing Channel Names

Using the `@Input` and `@Output` annotations, you can specify a customized channel name for the channel, as shown in the following example:

JAVA

```
public interface Barista {
    ...
    @Input("inboundOrders")
    SubscribableChannel orders();
}
```

In this example, the created bound channel will be named `inboundOrders`.

`Source`, `Sink`, and `Processor`

For easy addressing of the most common use cases, which involve either an input channel, an output channel, or both, Spring Cloud Stream provides three predefined interfaces out of the box.

`Source` can be used for an application which has a single outbound channel.

```
public interface Source {

  String OUTPUT = "output";

  @Output(Source.OUTPUT)
  MessageChannel output();

}
```
JAVA

`Sink` can be used for an application which has a single inbound channel.

```
public interface Sink {

  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();

}
```
JAVA

`Processor` can be used for an application which has both an inbound channel and an outbound channel.

```
public interface Processor extends Source, Sink {
}
```
JAVA

Spring Cloud Stream provides no special handling for any of these interfaces; they are only provided out of the box.

Accessing Bound Channels

Injecting the Bound Interfaces

For each bound interface, Spring Cloud Stream will generate a bean that implements the interface. Invoking a `@Input`-annotated or `@Output`-annotated method of one of these beans will return the relevant bound channel.

The bean in the following example sends a message on the output channel when its `hello` method is invoked. It invokes `output()` on the injected `Source` bean to retrieve the target channel.

```
@Component
public class SendingBean {

    private Source source;

    @Autowired
    public SendingBean(Source source) {
        this.source = source;
    }

    public void sayHello(String name) {
         source.output().send(MessageBuilder.withPayload(name).build());
    }
}
```
JAVA

Injecting Channels Directly

Bound channels can be also injected directly:

JAVA
```
@Component
public class SendingBean {

    private MessageChannel output;

    @Autowired
    public SendingBean(MessageChannel output) {
        this.output = output;
    }

    public void sayHello(String name) {
         output.send(MessageBuilder.withPayload(name).build());
    }
}
```

If the name of the channel is customized on the declaring annotation, that name should be used instead of the method name. Given the following declaration:

JAVA
```
public interface CustomSource {
    ...
    @Output("customOutput")
    MessageChannel output();
}
```

The channel will be injected as shown in the following example:

JAVA
```
@Component
public class SendingBean {

    private MessageChannel output;

    @Autowired
    public SendingBean(@Qualifier("customOutput") MessageChannel output) {
        this.output = output;
    }

    public void sayHello(String name) {
         this.output.send(MessageBuilder.withPayload(name).build());
    }
}
```

Producing and Consuming Messages

You can write a Spring Cloud Stream application using either Spring Integration annotations or Spring Cloud Stream's `@StreamListener` annotation. The `@StreamListener` annotation is modeled after other Spring Messaging annotations (such as `@MessageMapping`, `@JmsListener`, `@RabbitListener`, etc.) but adds content type management and type coercion features.

Native Spring Integration Support

Because Spring Cloud Stream is based on Spring Integration, Stream completely inherits Integration's foundation and infrastructure as well as the component itself. For example, you can attach the output channel of a `Source` to a `MessageSource`:

JAVA
```
@EnableBinding(Source.class)
public class TimerSource {

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

  @Bean
  @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "${fixedDelay}", maxMessagesPerPoll = "1"))
  public MessageSource<String> timerMessageSource() {
    return () -> new GenericMessage<>(new SimpleDateFormat(format).format(new Date()));
  }
}
```

Or you can use a processor's channels in a transformer:

JAVA
```
@EnableBinding(Processor.class)
public class TransformProcessor {
  @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
  public Object transform(String message) {
    return message.toUpperCase();
  }
}
```

Spring Integration Error Channel Support

Spring Cloud Stream supports publishing error messages received by the Spring Integration global error channel. Error messages sent to the `errorChannel` can be published to a specific destination at the broker by configuring a binding for the outbound target named `error`. For example, to publish error messages to a broker destination named "myErrors", provide the following property: `spring.cloud.stream.bindings.error.destination=myErrors`

Using @StreamListener for Automatic Content Type Handling

Complementary to its Spring Integration support, Spring Cloud Stream provides its own `@StreamListener` annotation, modeled after other Spring Messaging annotations (e.g. `@MessageMapping`, `@JmsListener`, `@RabbitListener`, etc.). The `@StreamListener` annotation provides a simpler model for handling inbound messages, especially when dealing with use cases that involve content type management and type coercion.

Spring Cloud Stream provides an extensible `MessageConverter` mechanism for handling data conversion by bound channels and for, in this case, dispatching to methods annotated with `@StreamListener`. The following is an example of an application which processes external `Vote` events:

JAVA

```
@EnableBinding(Sink.class)
public class VoteHandler {

  @Autowired
  VotingService votingService;

  @StreamListener(Sink.INPUT)
  public void handle(Vote vote) {
    votingService.record(vote);
  }
}
```

The distinction between `@StreamListener` and a Spring Integration `@ServiceActivator` is seen when considering an inbound `Message` that has a `String` payload and a `contentType` header of `application/json`. In the case of `@StreamListener`, the `MessageConverter` mechanism will use the `contentType` header to parse the `String` payload into a `Vote` object.

As with other Spring Messaging methods, method arguments can be annotated with `@Payload`, `@Headers` and `@Header`.

NOTE

For methods which return data, you must use the `@SendTo` annotation to specify the output binding destination for data returned by the method:

JAVA

```
@EnableBinding(Processor.class)
public class TransformProcessor {

  @Autowired
  VotingService votingService;

  @StreamListener(Processor.INPUT)
  @SendTo(Processor.OUTPUT)
  public VoteResult handle(Vote vote) {
    return votingService.record(vote);
  }
}
```

Using @StreamListener for dispatching messages to multiple methods

Since version 1.2, Spring Cloud Stream supports dispatching messages to multiple `@StreamListener` methods registered on an input channel, based on a condition.

In order to be eligible to support conditional dispatching, a method must satisfy the follow conditions:

- it must not return a value
- it must be an individual message handling method (reactive API methods are not supported)

The condition is specified via a SpEL expression in the `condition` attribute of the annotation and is evaluated for each message. All the handlers that match the condition will be invoked in the same thread and no assumption must be made about the order in which the invocations take place.

An example of using `@StreamListener` with dispatching conditions can be seen below. In this example, all the messages bearing a header `type` with the value `foo` will be dispatched to the `receiveFoo` method, and all the messages bearing a header `type` with the value `bar` will be dispatched to the `receiveBar` method.

JAVA

```
@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class TestPojoWithAnnotatedArguments {

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='foo'")
    public void receiveFoo(@Payload FooPojo fooPojo) {
       // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bar'")
    public void receiveBar(@Payload BarPojo barPojo) {
       // handle the message
    }
}
```

NOTE

Dispatching via `@StreamListener` conditions is only supported for handlers of individual messages, and not for reactive programming support (described below).

Reactive Programming Support

Spring Cloud Stream also supports the use of reactive APIs where incoming and outgoing data is handled as continuous data flows. Support for reactive APIs is available via the `spring-cloud-stream-reactive`, which needs to be added explicitly to your project.

The programming model with reactive APIs is declarative, where instead of specifying how each individual message should be handled, you can use operators that describe functional transformations from inbound to outbound data flows.

Spring Cloud Stream supports the following reactive APIs:

- Reactor
- RxJava 1.x

In the future, it is intended to support a more generic model based on Reactive Streams.

The reactive programming model is also using the `@StreamListener` annotation for setting up reactive handlers. The differences are that:

- the `@StreamListener` annotation must not specify an input or output, as they are provided as arguments and return values from the method;
- the arguments of the method must be annotated with `@Input` and `@Output` indicating which input or output will the incoming and respectively outgoing data flows connect to;
- the return value of the method, if any, will be annotated with `@Output`, indicating the input where data shall be sent.

NOTE

Reactive programming support requires Java 1.8.

NOTE

As of Spring Cloud Stream 1.1.1 and later (starting with release train Brooklyn.SR2), reactive programming support requires the use of Reactor 3.0.4.RELEASE and higher. Earlier Reactor versions (including 3.0.1.RELEASE, 3.0.2.RELEASE and 3.0.3.RELEASE) are not supported. `spring-cloud-stream-reactive` will transitively retrieve the proper version, but it is possible for the project structure to manage the version of the `io.projectreactor:reactor-core` to an earlier release, especially when using Maven. This is the case for projects generated via Spring Initializr with Spring Boot 1.x, which will override the Reactor version to `2.0.8.RELEASE`. In such cases you must ensure that the proper version of the artifact is released. This can be simply achieved by adding a direct dependency on `io.projectreactor:reactor-core` with a version of `3.0.4.RELEASE` or later to your project.

NOTE

The use of term `reactive` is currently referring to the reactive APIs being used and not to the execution model being reactive (i.e. the bound endpoints are still using a 'push' rather than 'pull' model). While some backpressure support is provided by the use of Reactor, we do intend on the long run to support entirely reactive pipelines by the use of native reactive clients for the connected middleware.

Reactor-based handlers

A Reactor based handler can have the following argument types:

- For arguments annotated with `@Input`, it supports the Reactor type `Flux`. The parameterization of the inbound Flux follows the same rules as in the case of individual message handling: it can be the entire `Message`, a POJO which can be the `Message` payload, or a POJO which is the result of a transformation based on the `Message` content-type header. Multiple inputs are provided;
- For arguments annotated with `Output`, it supports the type `FluxSender` which connects a `Flux` produced by the method with an output. Generally speaking, specifying outputs as arguments is only recommended when the method can have multiple outputs;

A Reactor based handler supports a return type of `Flux`, case in which it must be annotated with `@Output`. We recommend using the return value of the method when a single output flux is available.

Here is an example of a simple Reactor-based Processor.

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  @Output(Processor.OUTPUT)
  public Flux<String> receive(@Input(Processor.INPUT) Flux<String> input) {
    return input.map(s -> s.toUpperCase());
  }
}
```
JAVA

The same processor using output arguments looks like this:

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  public void receive(@Input(Processor.INPUT) Flux<String> input,
     @Output(Processor.OUTPUT) FluxSender output) {
     output.send(input.map(s -> s.toUpperCase()));
  }
}
```
JAVA

RxJava 1.x support

RxJava 1.x handlers follow the same rules as Reactor-based one, but will use `Observable` and `ObservableSender` arguments and return types.

So the first example above will become:

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  @Output(Processor.OUTPUT)
  public Observable<String> receive(@Input(Processor.INPUT) Observable<String> input) {
    return input.map(s -> s.toUpperCase());
  }
}
```
JAVA

The second example above will become:

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  public void receive(@Input(Processor.INPUT) Observable<String> input,
     @Output(Processor.OUTPUT) ObservableSender output) {
     output.send(input.map(s -> s.toUpperCase()));
  }
}
```
JAVA

Aggregation

Spring Cloud Stream provides support for aggregating multiple applications together, connecting their input and output channels directly and avoiding the additional cost of exchanging messages via a broker. As of version 1.0 of Spring Cloud Stream, aggregation is supported only for the following types of applications:

- *sources* - applications with a single output channel named `output`, typically having a single binding of the type `org.springframework.cloud.stream.messaging.Source`
- *sinks* - applications with a single input channel named `input`, typically having a single binding of the type `org.springframework.cloud.stream.messaging.Sink`
- *processors* - applications with a single input channel named `input` and a single output channel named `output`, typically having a single binding of the type `org.springframework.cloud.stream.messaging.Processor`.

They can be aggregated together by creating a sequence of interconnected applications, in which the output channel of an element in the sequence is connected to the input channel of the next element, if it exists. A sequence can start with either a *source* or a *processor*, it can contain an arbitrary number of *processors* and must end with either a *processor* or a *sink*.

Depending on the nature of the starting and ending element, the sequence may have one or more bindable channels, as follows:

- if the sequence starts with a source and ends with a sink, all communication between the applications is direct and no channels will be bound
- if the sequence starts with a processor, then its input channel will become the `input` channel of the aggregate and will be bound accordingly
- if the sequence ends with a processor, then its output channel will become the `output` channel of the aggregate and will be bound accordingly

Aggregation is performed using the `AggregateApplicationBuilder` utility class, as in the following example. Let's consider a project in which we have source, processor and a sink, which may be defined in the project, or may be contained in one of the project's dependencies.

> **NOTE** Each component (source, sink or processor) in an aggregate application must be provided in a separate package if the configuration classes use `@SpringBootApplication`. This is required to avoid cross-talk between applications, due to the classpath scanning performed by `@SpringBootApplication` on the configuration classes inside the same package. In the example below, it can be seen that the Source, Processor and Sink application classes are grouped in separate packages. A possible alternative is to provide the source, sink or processor configuration in a separate `@Configuration` class, avoid the use of `@SpringBootApplication` / `@ComponentScan` and use those for aggregation.

JAVA

```
package com.app.mysink;

@SpringBootApplication
@EnableBinding(Sink.class)
public class SinkApplication {

    private static Logger logger = LoggerFactory.getLogger(SinkApplication.class);

    @ServiceActivator(inputChannel=Sink.INPUT)
    public void loggerSink(Object payload) {
        logger.info("Received: " + payload);
    }
}
```

JAVA

```
package com.app.myprocessor;

@SpringBootApplication
@EnableBinding(Processor.class)
public class ProcessorApplication {

    @Transformer
    public String loggerSink(String payload) {
        return payload.toUpperCase();
    }
}
```

```
JAVA
package com.app.mysource;

@SpringBootApplication
@EnableBinding(Source.class)
public class SourceApplication {

    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT)
    public String timerMessageSource() {
        return new SimpleDateFormat().format(new Date());
    }
}
```

Each configuration can be used for running a separate component, but in this case they can be aggregated together as follows:

```
JAVA
package com.app;

@SpringBootApplication
public class SampleAggregateApplication {

    public static void main(String[] args) {
        new AggregateApplicationBuilder()
            .from(SourceApplication.class).args("--fixedDelay=5000")
            .via(ProcessorApplication.class)
            .to(SinkApplication.class).args("--debug=true").run(args);
    }
}
```

The starting component of the sequence is provided as argument to the `from()` method. The ending component of the sequence is provided as argument to the `to()` method. Intermediate processors are provided as argument to the `via()` method. Multiple processors of the same type can be chained together (e.g. for pipelining transformations with different configurations). For each component, the builder can provide runtime arguments for Spring Boot configuration.

Configuring aggregate application

Spring Cloud Stream supports passing properties for the individual applications inside the aggregate application using 'namespace' as prefix.

The namespace can be set for applications as follows:

```
JAVA
@SpringBootApplication
public class SampleAggregateApplication {

    public static void main(String[] args) {
        new AggregateApplicationBuilder()
            .from(SourceApplication.class).namespace("source").args("--fixedDelay=5000")
            .via(ProcessorApplication.class).namespace("processor1")
            .to(SinkApplication.class).namespace("sink").args("--debug=true").run(args);
    }
}
```

Once the 'namespace' is set for the individual applications, the application properties with the `namespace` as prefix can be passed to the aggregate application using any supported property source (commandline, environment properties etc.,)

For instance, to override the default `fixedDelay` and `debug` properties of 'source' and 'sink' applications:

```
java -jar target/MyAggregateApplication-0.0.1-SNAPSHOT.jar --source.fixedDelay=10000 --sink.debug=false
```

Configuring binding service properties for non self contained aggregate application

The non self-contained aggregate application is bound to external broker via either or both the inbound/outbound components (typically, message channels) of the aggregate application while the applications inside the aggregate application are directly bound. For example: a source application's output and a processor application's input are directly bound while the processor's output channel is bound to an external destination at the broker. When passing the binding service properties for non-self contained aggregate application, it is required to pass the binding service properties to the aggregate application instead of setting them as 'args' to individual child application. For instance,

JAVA

```
@SpringBootApplication
public class SampleAggregateApplication {

    public static void main(String[] args) {
        new AggregateApplicationBuilder()
            .from(SourceApplication.class).namespace("source").args("--fixedDelay=5000")
            .via(ProcessorApplication.class).namespace("processor1").args("--debug=true").run(args);
    }
}
```

The binding properties like `--spring.cloud.stream.bindings.output.destination=processor-output` need to be specified as one of the external configuration properties (cmdline arg etc.,).

Binders

Spring Cloud Stream provides a Binder abstraction for use in connecting to physical destinations at the external middleware. This section provides information about the main concepts behind the Binder SPI, its main components, and implementation-specific details.

Producers and Consumers

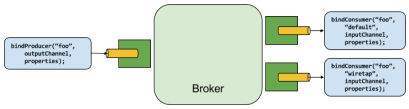


Figure 9. Producers and Consumers

A *producer* is any component that sends messages to a channel. The channel can be bound to an external message broker via a Binder implementation for that broker. When invoking the `bindProducer()` method, the first parameter is the name of the destination within the broker, the second parameter is the local channel instance to which the producer will send messages, and the third parameter contains properties (such as a partition key expression) to be used within the adapter that is created for that channel.

A *consumer* is any component that receives messages from a channel. As with a producer, the consumer's channel can be bound to an external message broker. When invoking the `bindConsumer()` method, the first parameter is the destination name, and a second parameter provides the name of a logical group of consumers. Each group that is represented by consumer bindings for a given destination receives a copy of each message that a producer sends to that destination (i.e., publish-subscribe semantics). If there are multiple consumer instances bound using the same group name, then messages will be load-balanced across those consumer instances so that each message sent by a producer is consumed by only a single consumer instance within each group (i.e., queueing semantics).

Binder SPI

The Binder SPI consists of a number of interfaces, out-of-the box utility classes and discovery strategies that provide a pluggable mechanism for connecting to external middleware.

The key point of the SPI is the `Binder` interface which is a strategy for connecting inputs and outputs to external middleware.

JAVA

```
public interface Binder<T, C extends ConsumerProperties, P extends ProducerProperties> {
    Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C consumerProperties);

    Binding<T> bindProducer(String name, T outboundBindTarget, P producerProperties);
}
```

The interface is parameterized, offering a number of extension points:

- input and output bind targets - as of version 1.0, only `MessageChannel` is supported, but this is intended to be used as an extension point in the future;
- extended consumer and producer properties - allowing specific Binder implementations to add supplemental properties which can be supported in a type-safe manner.

A typical binder implementation consists of the following

- a class that implements the `Binder` interface;
- a Spring `@Configuration` class that creates a bean of the type above along with the middleware connection infrastructure;
- a `META-INF/spring.binders` file found on the classpath containing one or more binder definitions, e.g.

```
kafka:\
org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration
```

Binder Detection

Spring Cloud Stream relies on implementations of the Binder SPI to perform the task of connecting channels to message brokers. Each Binder implementation typically connects to one type of messaging system.

Classpath Detection

By default, Spring Cloud Stream relies on Spring Boot's auto-configuration to configure the binding process. If a single Binder implementation is found on the classpath, Spring Cloud Stream will use it automatically. For example, a Spring Cloud Stream project that aims to bind only to RabbitMQ can simply add the following dependency:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

XML

For the specific maven coordinates of other binder dependencies, please refer to the documentation of that binder implementation.

Multiple Binders on the Classpath

When multiple binders are present on the classpath, the application must indicate which binder is to be used for each channel binding. Each binder configuration contains a `META-INF/spring.binders`, which is a simple properties file:

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration
```

Similar files exist for the other provided binder implementations (e.g., Kafka), and custom binder implementations are expected to provide them, as well. The key represents an identifying name for the binder implementation, whereas the value is a comma-separated list of configuration classes that each contain one and only one bean definition of type `org.springframework.cloud.stream.binder.Binder`.

Binder selection can either be performed globally, using the `spring.cloud.stream.defaultBinder` property (e.g., `spring.cloud.stream.defaultBinder=rabbit`) or individually, by configuring the binder on each channel binding. For instance, a processor application (that has channels with the names `input` and `output` for read/write respectively) which reads from Kafka and writes to RabbitMQ can specify the following configuration:

```
spring.cloud.stream.bindings.input.binder=kafka
spring.cloud.stream.bindings.output.binder=rabbit
```

Connecting to Multiple Systems

By default, binders share the application's Spring Boot auto-configuration, so that one instance of each binder found on the classpath will be created. If your application should connect to more than one broker of the same type, you can specify multiple binder configurations, each with different environment settings.

NOTE

Turning on explicit binder configuration will disable the default binder configuration process altogether. If you do this, all binders in use must be included in the configuration. Frameworks that intend to use Spring Cloud Stream transparently may create binder configurations that can be referenced by name, but will not affect the default binder configuration. In order to do so, a binder configuration may have its `defaultCandidate` flag set to false, e.g. `spring.cloud.stream.binders.<configurationName>.defaultCandidate=false`. This denotes a configuration that will exist independently of the default binder configuration process.

For example, this is the typical configuration for a processor application which connects to two RabbitMQ broker instances:

YML

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: foo
          binder: rabbit1
        output:
          destination: bar
          binder: rabbit2
      binders:
        rabbit1:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host1>
        rabbit2:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host2>
```

Binder configuration properties

The following properties are available when creating custom binder configurations. They must be prefixed with `spring.cloud.stream.binders.<configurationName>`.

type

The binder type. It typically references one of the binders found on the classpath, in particular a key in a `META-INF/spring.binders` file.

By default, it has the same value as the configuration name.

inheritEnvironment

Whether the configuration will inherit the environment of the application itself.

Default `true`.

environment

Root for a set of properties that can be used to customize the environment of the binder. When this is configured, the context in which the binder is being created is not a child of the application context. This allows for complete separation between the binder components and the application components.

Default `empty`.

defaultCandidate

Whether the binder configuration is a candidate for being considered a default binder, or can be used only when explicitly referenced. This allows adding binder configurations without interfering with the default processing.

Default `true`.

Configuration Options

Spring Cloud Stream supports general configuration options as well as configuration for bindings and binders. Some binders allow additional binding properties to support middleware-specific features.

Configuration options can be provided to Spring Cloud Stream applications via any mechanism supported by Spring Boot. This includes application arguments, environment variables, and YAML or .properties files.

Spring Cloud Stream Properties

spring.cloud.stream.instanceCount

The number of deployed instances of an application. Must be set for partitioning and if using Kafka.

Default: `1`.

spring.cloud.stream.instanceIndex

The instance index of the application: a number from `0` to `instanceCount` -1. Used for partitioning and with Kafka. Automatically set in Cloud Foundry to match the application's instance index.

spring.cloud.stream.dynamicDestinations

A list of destinations that can be bound dynamically (for example, in a dynamic routing scenario). If set, only listed destinations can be bound.

Default: empty (allowing any destination to be bound).

spring.cloud.stream.defaultBinder

The default binder to use, if multiple binders are configured. See Multiple Binders on the Classpath.

Default: empty.

spring.cloud.stream.overrideCloudConnectors

This property is only applicable when the `cloud` profile is active and Spring Cloud Connectors are provided with the application. If the property is false (the default), the binder will detect a suitable bound service (e.g. a RabbitMQ service bound in Cloud Foundry for the RabbitMQ binder) and will use it for creating connections (usually via Spring Cloud Connectors). When set to true, this property instructs binders to completely ignore the bound services and rely on Spring Boot properties (e.g. relying on the `spring.rabbitmq.*` properties provided in the environment for the RabbitMQ binder). The typical usage of this property is to be nested in a customized environment when connecting to multiple systems.

Default: false.

Binding Properties

Binding properties are supplied using the format `spring.cloud.stream.bindings.<channelName>.<property>=<value>`. The `<channelName>` represents the name of the channel being configured (e.g., `output` for a `Source`).

To avoid repetition, Spring Cloud Stream supports setting values for all channels, in the format `spring.cloud.stream.default.<property>=<value>`.

In what follows, we indicate where we have omitted the `spring.cloud.stream.bindings.<channelName>.` prefix and focus just on the property name, with the understanding that the prefix will be included at runtime.

Properties for Use of Spring Cloud Stream

The following binding properties are available for both input and output bindings and must be prefixed with `spring.cloud.stream.bindings.<channelName>.`, e.g. `spring.cloud.stream.bindings.input.destination=ticktock`.

Default values can be set by using the prefix `spring.cloud.stream.default`, e.g. `spring.cloud.stream.default.contentType=application/json`.

destination

The target destination of a channel on the bound middleware (e.g., the RabbitMQ exchange or Kafka topic). If the channel is bound as a consumer, it could be bound to multiple destinations and the destination names can be specified as comma separated String values. If not set, the channel name is used instead. The default value of this property cannot be overridden.

group

The consumer group of the channel. Applies only to inbound bindings. See Consumer Groups.

Default: null (indicating an anonymous consumer).

contentType

The content type of the channel.

Default: null (so that no type coercion is performed).

binder

The binder used by this binding. See Multiple Binders on the Classpath for details.

Default: null (the default binder will be used, if one exists).

Consumer properties

The following binding properties are available for input bindings only and must be prefixed with `spring.cloud.stream.bindings.<channelName>.consumer.`, e.g. `spring.cloud.stream.bindings.input.consumer.concurrency=3`.

Default values can be set by using the prefix `spring.cloud.stream.default.consumer`, e.g. `spring.cloud.stream.default.consumer.headerMode=raw`.

concurrency

The concurrency of the inbound consumer.

Default: `1`.

partitioned

Whether the consumer receives data from a partitioned producer.

Default: `false`.

headerMode

When set to `raw`, disables header parsing on input. Effective only for messaging middleware that does not support message headers natively and requires header embedding. Useful when inbound data is coming from outside Spring Cloud Stream applications.

Default: `embeddedHeaders`.

maxAttempts

If processing fails, the number of attempts to process the message (including the first). Set to 1 to disable retry.

Default: `3`.

backOffInitialInterval

The backoff initial interval on retry.

Default: `1000`.

backOffMaxInterval

The maximum backoff interval.

Default: `10000`.

backOffMultiplier

The backoff multiplier.

Default: `2.0`.

instanceIndex

When set to a value greater than equal to zero, allows customizing the instance index of this consumer (if different from `spring.cloud.stream.instanceIndex`). When set to a negative value, it will default to `spring.cloud.stream.instanceIndex`.

Default: `-1`.

instanceCount

When set to a value greater than equal to zero, allows customizing the instance count of this consumer (if different from `spring.cloud.stream.instanceCount`). When set to a negative value, it will default to `spring.cloud.stream.instanceCount`.

Default: `-1`.

Producer Properties

The following binding properties are available for output bindings only and must be prefixed with `spring.cloud.stream.bindings.<channelName>.producer.`, e.g. `spring.cloud.stream.bindings.input.producer.partitionKeyExpression=payload.id`.

Default values can be set by using the prefix `spring.cloud.stream.default.producer`, e.g. `spring.cloud.stream.default.producer.partitionKeyExpression=payload.id`.

partitionKeyExpression

A SpEL expression that determines how to partition outbound data. If set, or if `partitionKeyExtractorClass` is set, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value greater than 1 to be effective. The two options are mutually exclusive. See Partitioning Support.

Default: null.

partitionKeyExtractorClass

A `PartitionKeyExtractorStrategy` implementation. If set, or if `partitionKeyExpression` is set, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value greater than 1 to be effective. The two options are mutually exclusive. See Partitioning Support.

Default: null.

partitionSelectorClass

A `PartitionSelectorStrategy` implementation. Mutually exclusive with `partitionSelectorExpression`. If neither is set, the partition will be selected as the `hashCode(key) % partitionCount`, where `key` is computed via either `partitionKeyExpression` or `partitionKeyExtractorClass`.

Default: null.

partitionSelectorExpression

A SpEL expression for customizing partition selection. Mutually exclusive with `partitionSelectorClass`. If neither is set, the partition will be selected as the `hashCode(key) % partitionCount`, where `key` is computed via either `partitionKeyExpression` or `partitionKeyExtractorClass`.

Default: null.

partitionCount

The number of target partitions for the data, if partitioning is enabled. Must be set to a value greater than 1 if the producer is partitioned. On Kafka, interpreted as a hint; the larger of this and the partition count of the target topic is used instead.

Default: `1`.

requiredGroups

A comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created (e.g., by pre-creating durable queues in RabbitMQ).

headerMode

When set to `raw`, disables header embedding on output. Effective only for messaging middleware that does not support message headers natively and requires header embedding. Useful when producing data for non-Spring Cloud Stream applications.

Default: `embeddedHeaders`.

useNativeEncoding

When set to `true`, the outbound message is serialized directly by client library, which must be configured correspondingly (e.g. setting an appropriate Kafka producer value serializer). When this configuration is being used, the outbound message marshalling is not based on the `contentType` of the binding. When native encoding is used, it is the responsibility of the consumer to use appropriate decoder (ex: Kafka consumer value de-serializer) to deserialize the inbound message. Also, when native encoding/decoding is used the `headerMode` property is ignored and headers will not be embedded into the message.

Default: `false`.

Using dynamically bound destinations

Besides the channels defined via `@EnableBinding`, Spring Cloud Stream allows applications to send messages to dynamically bound destinations. This is useful, for example, when the target destination needs to be determined at runtime. Applications can do so by using the `BinderAwareChannelResolver` bean, registered automatically by the `@EnableBinding` annotation.

The property 'spring.cloud.stream.dynamicDestinations' can be used for restricting the dynamic destination names to a set known beforehand (whitelisting). If the property is not set, any destination can be bound dynamicaly.

The `BinderAwareChannelResolver` can be used directly as in the following example, in which a REST controller uses a path variable to decide the target channel.

JAVA

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

    @Autowired
    private BinderAwareChannelResolver resolver;

    @RequestMapping(path = "/{target}", method = POST, consumes = "*/*")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void handleRequest(@RequestBody String body, @PathVariable("target") target,
           @RequestHeader(HttpHeaders.CONTENT_TYPE) Object contentType) {
        sendMessage(body, target, contentType);
    }

    private void sendMessage(String body, String target, Object contentType) {
        resolver.resolveDestination(target).send(MessageBuilder.createMessage(body,
                new MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT_TYPE, contentType))));
    }
}
```

After starting the application on the default port 8080, when sending the following data:

```
curl -H "Content-Type: application/json" -X POST -d "customer-1" http://localhost:8080/customers

curl -H "Content-Type: application/json" -X POST -d "order-1" http://localhost:8080/orders
```

The destinations 'customers' and 'orders' are created in the broker (for example: exchange in case of Rabbit or topic in case of Kafka) with the names 'customers' and 'orders', and the data is published to the appropriate destinations.

The `BinderAwareChannelResolver` is a general purpose Spring Integration `DestinationResolver` and can be injected in other components. For example, in a router using a SpEL expression based on the `target` field of an incoming JSON message.

JAVA

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

    @Autowired
    private BinderAwareChannelResolver resolver;


    @RequestMapping(path = "/", method = POST, consumes = "application/json")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void handleRequest(@RequestBody String body, @RequestHeader(HttpHeaders.CONTENT_TYPE) Object contentType) {
        sendMessage(body, contentType);
    }

    private void sendMessage(Object body, Object contentType) {
        routerChannel().send(MessageBuilder.createMessage(body,
                new MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT_TYPE, contentType))));
    }

    @Bean(name = "routerChannel")
    public MessageChannel routerChannel() {
        return new DirectChannel();
    }

    @Bean
    @ServiceActivator(inputChannel = "routerChannel")
    public ExpressionEvaluatingRouter router() {
        ExpressionEvaluatingRouter router =
            new ExpressionEvaluatingRouter(new SpelExpressionParser().parseExpression("payload.target"));
        router.setDefaultOutputChannelName("default-output");
        router.setChannelResolver(resolver);
        return router;
    }
}
```

Content Type and Transformation

To allow you to propagate information about the content type of produced messages, Spring Cloud Stream attaches, by default, a `contentType` header to outbound messages. For middleware that does not directly support headers, Spring Cloud Stream provides its own mechanism of automatically wrapping outbound messages in an envelope of its own. For middleware that does support headers, Spring Cloud Stream applications may receive messages with a given content type from non-Spring Cloud Stream applications.

Spring Cloud Stream can handle messages based on this information in two ways:

- Through its `contentType` settings on inbound and outbound channels
- Through its argument mapping performed for methods annotated with `@StreamListener`

Spring Cloud Stream allows you to declaratively configure type conversion for inputs and outputs using the `spring.cloud.stream.bindings.<channelName>.content-type` property of a binding. Note that general type conversion may also be accomplished easily by using a transformer inside your application. Currently, Spring Cloud Stream natively supports the following type conversions commonly used in streams:

- **JSON** to/from **POJO**
- **JSON** to/from org.springframework.tuple.Tuple (https://github.com/spring-projects/spring-tuple/blob/master/spring-tuple/src/main/java/org/springframework/tuple/Tuple.java)
- **Object** to/from **byte[]** : Either the raw bytes serialized for remote transport, bytes emitted by an application, or converted to bytes using Java serialization(requires the object to be Serializable)
- **String** to/from **byte[]**
- **Object** to **plain text** (invokes the object's *toString()* method)

Where *JSON* represents either a byte array or String payload containing JSON. Currently, Objects may be converted from a JSON byte array or String. Converting to JSON always produces a String.

If no `content-type` property is set on an outbound channel, Spring Cloud Stream will serialize the payload using a serializer based on the Kryo (https://github.com/EsotericSoftware/kryo) serialization framework. Deserializing messages at the destination requires the payload class to be present on the receiver's classpath.

MIME types

`content-type` values are parsed as media types, e.g., `application/json` or `text/plain;charset=UTF-8`. MIME types are especially useful for indicating how to convert to String or byte[] content. Spring Cloud Stream also uses MIME type format to represent Java types, using the general type `application/x-java-object` with a `type` parameter. For example, `application/x-java-object;type=java.util.Map` or `application/x-java-object;type=com.bar.Foo` can be set as the `content-type` property of an input binding. In addition, Spring Cloud Stream provides custom MIME types, notably, `application/x-spring-tuple` to specify a Tuple.

MIME types and Java types

The type conversions Spring Cloud Stream provides out of the box are summarized in the following table: 'Source Payload' means the payload before conversion and 'Target Payload' means the 'payload' after conversion. The type conversion can occur either on the 'producer' side (output) or at the 'consumer' side (input).

Source Payload	**Target Payload**	`content-type` **header (source message)**	`content-type` **header (after conversion)**	**Comments**
POJO	JSON String	ignored	application/json	
Tuple	JSON String	ignored	application/json	JSON is tailored for Tuple
POJO	String (toString())	ignored	text/plain, java.lang.String	

Source Payload	Target Payload	`content-type` header (source message)	`content-type` header (after conversion)	Comments
POJO	byte[] (java.io serialized)	ignored	application/x-java-serialized-object	
JSON byte[] or String	POJO	application/json (or none)	application/x-java-object	
byte[] or String	Serializable	application/x-java-serialized-object	application/x-java-object	
JSON byte[] or String	Tuple	application/json (or none)	application/x-spring-tuple	
byte[]	String	any	text/plain, java.lang.String	will apply any Charset specified in the content-type header
String	byte[]	any	application/octet-stream	will apply any Charset specified in the content-type header

NOTE

Conversion applies to payloads that require type conversion. For example, if an application produces an XML string with outputType=application/json, the payload will not be converted from XML to JSON. This is because the payload send to the outbound channel is already a String so no conversion will be applied at runtime. It is also important to note that when using the default serialization mechanism, the payload class must be shared between the sending and receiving application, and compatible with the binary content. This can create issues when application code changes independently in the two applications, as the binary format and code may become incompatible.

TIP

While conversion is supported for both inbound and outbound channels, it is especially recommended to be used for the conversion of outbound messages. For the conversion of inbound messages, especially when the target is a POJO, the `@StreamListener` support will perform the conversion automatically.

Customizing message conversion

Besides the conversions that it supports out of the box, Spring Cloud Stream also supports registering your own message conversion implementations. This allows you to send and receive data in a variety of custom formats, including binary, and associate them with specific `contentTypes`. Spring Cloud Stream registers all the beans of type `org.springframework.messaging.converter.MessageConverter` as custom message converters along with the out of the box message converters.

If your message converter needs to work with a specific `content-type` and target class (for both input and output), then the message converter needs to extend `org.springframework.messaging.converter.AbstractMessageConverter`. For conversion when using `@StreamListener`, a message converter that implements `org.springframework.messaging.converter.MessageConverter` would suffice.

Here is an example of creating a message converter bean (with the content-type `application/bar`) inside a Spring Cloud Stream application:

JAVA

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter customMessageConverter() {
    return new MyCustomMessageConverter();
  }
```

```
public class MyCustomMessageConverter extends AbstractMessageConverter {

    public MyCustomMessageConverter() {
        super(new MimeType("application", "bar"));
    }

    @Override
  protected boolean supports(Class<?> clazz) {
    return (Bar.class == clazz);
  }

    @Override
    protected Object convertFromInternal(Message<?> message, Class<?> targetClass, Object conversionHint) {
        Object payload = message.getPayload();
        return (payload instanceof Bar ? payload : new Bar((byte[]) payload));
    }
}
```
JAVA

Spring Cloud Stream also provides support for Avro-based converters and schema evolution. See the specific section for details.

`@StreamListener` and Message Conversion

The `@StreamListener` annotation provides a convenient way for converting incoming messages without the need to specify the content type of an input channel. During the dispatching process to methods annotated with `@StreamListener`, a conversion will be applied automatically if the argument requires it.

For example, let's consider a message with the String content `{"greeting":"Hello, world"}` and a `content-type` header of `application/json` is received on the input channel. Let us consider the following application that receives it:

```
public class GreetingMessage {

  String greeting;

  public String getGreeting() {
    return greeting;
  }

  public void setGreeting(String greeting) {
    this.greeting = greeting;
  }
}

@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class GreetingSink {

        @StreamListener(Sink.INPUT)
        public void receive(Greeting greeting) {
            // handle Greeting
        }
    }
```
JAVA

The argument of the method will be populated automatically with the POJO containing the unmarshalled form of the JSON String.

Schema evolution support

Spring Cloud Stream provides support for schema-based message converters through its `spring-cloud-stream-schema` module. Currently, the only serialization format supported out of the box for schema-based message converters is Apache Avro, with more formats to be added in future versions.

Apache Avro Message Converters

The `spring-cloud-stream-schema` module contains two types of message converters that can be used for Apache Avro serialization:

- converters using the class information of the serialized/deserialized objects, or a schema with a location known at startup;
- converters using a schema registry - they locate the schemas at runtime, as well as dynamically registering new schemas as domain objects evolve.

Converters with schema support

The `AvroSchemaMessageConverter` supports serializing and deserializing messages either using a predefined schema or by using the schema information available in the class (either reflectively, or contained in the `SpecificRecord`). If the target type of the conversion is a `GenericRecord`, then a schema must be set.

For using it, you can simply add it to the application context, optionally specifying one ore more `MimeTypes` to associate it with. The default `MimeType` is `application/avro`.

Here is an example of configuring it in a sink application registering the Apache Avro `MessageConverter`, without a predefined schema:

JAVA

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      return new AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));
  }
}
```

Conversely, here is an application that registers a converter with a predefined schema, to be found on the classpath:

JAVA

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      AvroSchemaMessageConverter converter = new AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));
      converter.setSchemaLocation(new ClassPathResource("schemas/User.avro"));
      return converter;
  }
}
```

In order to understand the schema registry client converter, we will describe the schema registry support first.

Schema Registry Support

Most serialization models, especially the ones that aim for portability across different platforms and languages, rely on a schema that describes how the data is serialized in the binary payload. In order to serialize the data and then to interpret it, both the sending and receiving sides must have access to a schema that describes the binary format. In certain cases, the schema can be inferred from the payload type on serialization, or from the target type on deserialization, but in a lot of cases applications benefit from having access to an explicit schema that describes the binary data format. A schema registry allows you to store schema information in a textual format (typically JSON) and makes that information accessible to various applications that need it to receive and send data in binary format. A schema is referenceable as a tuple consisting of:

- a *subject* that is the logical name of the schema;

- the schema *version*;
- the schema *format* which describes the binary format of the data.

Schema Registry Server

Spring Cloud Stream provides a schema registry server implementation. In order to use it, you can simply add the `spring-cloud-stream-schema-server` artifact to your project and use the `@EnableSchemaRegistryServer` annotation, adding the schema registry server REST controller to your application. This annotation is intended to be used with Spring Boot web applications, and the listening port of the server is controlled by the `server.port` setting. The `spring.cloud.stream.schema.server.path` setting can be used to control the root path of the schema server (especially when it is embedded in other applications). The `spring.cloud.stream.schema.server.allowSchemaDeletion` boolean setting enables the deletion of schema. By default this is disabled.

The schema registry server uses a relational database to store the schemas. By default, it uses an embedded database. You can customize the schema storage using the Spring Boot SQL database and JDBC configuration options (https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-sql).

A Spring Boot application enabling the schema registry looks as follows:

JAVA

```
@SpringBootApplication
@EnableSchemaRegistryServer
public class SchemaRegistryServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SchemaRegistryServerApplication.class, args);
    }
}
```

Schema Registry Server API

The Schema Registry Server API consists of the following operations:

`POST /`

Register a new schema.

Accepts JSON payload with the following fields:

- `subject` the schema subject;
- `format` the schema format;
- `definition` the schema definition.

Response is a schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`GET /{subject}/{format}/{version}`

Retrieve an existing schema by its subject, format and version.

Response is a schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`GET /{subject}/{format}`

Retrieve a list of existing schema by its subject and format.

Response is a list of schemas with each schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`GET /schemas/{id}`

Retrieve an existing schema by its id.

Response is a schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`DELETE /{subject}/{format}/{version}`

Delete an existing schema by its subject, format and version.

`DELETE /schemas/{id}`

Delete an existing schema by its id.

`DELETE /{subject}`

Delete existing schemas by their subject.

> **NOTE** This note applies to users of Spring Cloud Stream 1.1.0.RELEASE only. Spring Cloud Stream 1.1.0.RELEASE used the table name `schema` for storing `Schema` objects, which is a keyword in a number of database implementations. To avoid any conflicts in the future, starting with 1.1.1.RELEASE we have opted for the name `SCHEMA_REPOSITORY` for the storage table. Any Spring Cloud Stream 1.1.0.RELEASE users that are upgrading are advised to migrate their existing schemas to the new table before upgrading.

Schema Registry Client

The client-side abstraction for interacting with schema registry servers is the `SchemaRegistryClient` interface, with the following structure:

```
public interface SchemaRegistryClient {

    SchemaRegistrationResponse register(String subject, String format, String schema);

    String fetch(SchemaReference schemaReference);

    String fetch(Integer id);

}
```
JAVA

Spring Cloud Stream provides out of the box implementations for interacting with its own schema server, as well as for interacting with the Confluent Schema Registry.

A client for the Spring Cloud Stream schema registry can be configured using the `@EnableSchemaRegistryClient` as follows:

```
@EnableBinding(Sink.class)
@SpringBootApplication
@EnableSchemaRegistryClient
public static class AvroSinkApplication {
  ...
}
```
JAVA

NOTE

The default converter is optimized to cache not only the schemas from the remote server but also the `parse()` and `toString()` methods that are quite expensive. Because of this, it uses a `DefaultSchemaRegistryClient` that does not caches responses. If you intend to use the client directly on your code, you can request a bean that also caches responses to be created. To do that, just add the property `spring.cloud.stream.schemaRegistryClient.cached=true` to your application properties.

Schema Registry Client properties

The Schema Registry Client supports the following properties:

spring.cloud.stream.schemaRegistryClient.endpoint

The location of the schema-server. Use a full URL when setting this, including protocol (`http` or `https`), port and context path.

Default

`http://localhost:8990/`

spring.cloud.stream.schemaRegistryClient.cached

Whether the client should cache schema server responses. Normally set to `false` , as the caching happens in the message converter. Clients using the schema registry client should set this to `true` .

Default

`true`

Avro Schema Registry Client Message Converters

For Spring Boot applications that have a `SchemaRegistryClient` bean registered with the application context, Spring Cloud Stream will auto-configure an Apache Avro message converter that uses the schema registry client for schema management. This eases schema evolution, as applications that receive messages can get easy access to a writer schema that can be reconciled with their own reader schema.

For outbound messages, the `MessageConverter` will be activated if the content type of the channel is set to `application/*+avro` , e.g.:

```
spring.cloud.stream.bindings.output.contentType=application/*+avro
```
PROPERTIES

During the outbound conversion, the message converter will try to infer the schemas of the outbound messages based on their type and register them to a subject based on the payload type using the `SchemaRegistryClient` . If an identical schema is already found, then a reference to it will be retrieved. If not, the schema will be registered and a new version number will be provided. The message will be sent with a `contentType` header using the scheme `application/[prefix].[subject].v[version]+avro` , where `prefix` is configurable and `subject` is deduced from the payload type.

For example, a message of the type `User` may be sent as a binary payload with a content type of `application/vnd.user.v2+avro` , where `user` is the subject and `2` is the version number.

When receiving messages, the converter will infer the schema reference from the header of the incoming message and will try to retrieve it. The schema will be used as the writer schema in the deserialization process.

Avro Schema Registry Message Converter properties

If you have enabled Avro based schema registry client by setting `spring.cloud.stream.bindings.output.contentType=application/*+avro` you can customize the behavior of the registration with the following properties.

spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled

Enable if you want the converter to use reflection to infer a Schema from a POJO.

Default

`false`

spring.cloud.stream.schema.avro.readerSchema

Avro compares schema versions by looking at a writer schema (origin payload) and a reader schema (your application payload), check Avro (https://avro.apache.org/docs/1.7.6/spec.html) documentation for more information. If set, this overrides any lookups at the schema server and uses the local schema as the reader schema.

Default

`null`

spring.cloud.stream.schema.avro.schemaLocations

Register any `.avsc` files listed in this property with the Schema Server.

Default

`empty`

spring.cloud.stream.schema.avro.prefix

The prefix to be used on the Content-Type header.

Default

`vnd`

Schema Registration and Resolution

To better understand how Spring Cloud Stream registers and resolves new schemas, as well as its use of Avro schema comparison features, we will provide two separate subsections below: one for the registration, and one for the resolution of schemas.

Schema Registration Process (Serialization)

The first part of the registration process is extracting a schema from the payload that is being sent over a channel. Avro types such as `SpecificRecord` or `GenericRecord` already contain a schema, which can be retrieved immediately from the instance. In the case of POJOs a schema will be inferred if the property `spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled` is set to `true` (the default).

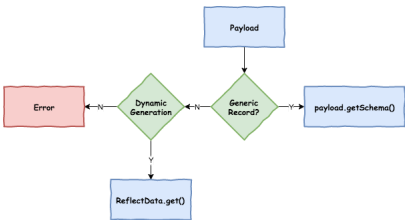


Figure 10. Schema Writer Resolution Process

Once a schema is obtained, the converter will then load its metadata (version) from the remote server. First it queries a local cache, and if not found it then submits the data to the server that will reply with versioning information. The converter will always cache the results to avoid the overhead of querying the Schema Server for every new message that needs to be serialized.

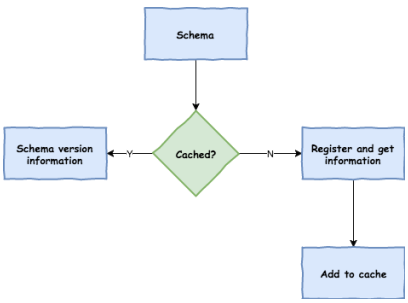


Figure 11. Schema Registration Process

With the schema version information, the converter sets the `contentType` header of the message to carry the version information such as `application/vnd.user.v1+avro`

Schema Resolution Process (Deserialization)

When reading messages that contain version information (i.e. a `contentType` header with a scheme like above), the converter will query the Schema server to fetch the **writer** schema of the message. Once it has found the correct schema of the incoming message, it then retrieves the reader schema and using Avro's schema resolution support reads it into the reader definition (setting defaults and missing properties).

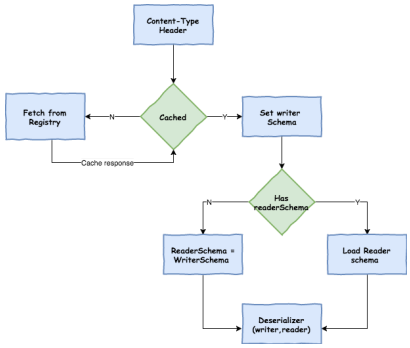


Figure 12. Schema Reading Resolution Process

NOTE

It's important to understand the difference between a writer schema (the application that wrote the message) and a reader schema (the receiving application). Please take a moment to read the Avro terminology (https://avro.apache.org/docs/1.7.6/spec.html) and understand the process. Spring Cloud Stream will always fetch the writer schema to determine how to read a message. If you want to get Avro's schema evolution support working you need to make sure that a readerSchema was properly set for your application.

Inter-Application Communication

Connecting Multiple Application Instances

While Spring Cloud Stream makes it easy for individual Spring Boot applications to connect to messaging systems, the typical scenario for Spring Cloud Stream is the creation of multi-application pipelines, where microservice applications send data to each other. You can achieve this scenario by correlating the input and output destinations of adjacent applications.

Supposing that a design calls for the Time Source application to send data to the Log Sink application, you can use a common destination named `ticktock` for bindings within both applications.

Time Source (that has the channel name `output`) will set the following property:

```
spring.cloud.stream.bindings.output.destination=ticktock
```

Log Sink (that has the channel name `input`) will set the following property:

```
spring.cloud.stream.bindings.input.destination=ticktock
```

Instance Index and Instance Count

When scaling up Spring Cloud Stream applications, each instance can receive information about how many other instances of the same application exist and what its own instance index is. Spring Cloud Stream does this through the `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties. For example, if there are three instances of a HDFS sink application, all three instances will have `spring.cloud.stream.instanceCount` set to `3`, and the individual applications will have `spring.cloud.stream.instanceIndex` set to `0`, `1`, and `2`, respectively.

When Spring Cloud Stream applications are deployed via Spring Cloud Data Flow, these properties are configured automatically; when Spring Cloud Stream applications are launched independently, these properties must be set correctly. By default, `spring.cloud.stream.instanceCount` is `1`, and `spring.cloud.stream.instanceIndex` is `0`.

In a scaled-up scenario, correct configuration of these two properties is important for addressing partitioning behavior (see below) in general, and the two properties are always required by certain binders (e.g., the Kafka binder) in order to ensure that data are split correctly across multiple consumer instances.

Partitioning

Configuring Output Bindings for Partitioning

An output binding is configured to send partitioned data by setting one and only one of its `partitionKeyExpression` or `partitionKeyExtractorClass` properties, as well as its `partitionCount` property. For example, the following is a valid and typical configuration:

```
spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload.id
spring.cloud.stream.bindings.output.producer.partitionCount=5
```

Based on the above example configuration, data will be sent to the target partition using the following logic.

A partition key's value is calculated for each message sent to a partitioned output channel based on the `partitionKeyExpression`. The `partitionKeyExpression` is a SpEL expression which is evaluated against the outbound message for extracting the partitioning key.

If a SpEL expression is not sufficient for your needs, you can instead calculate the partition key value by setting the property `partitionKeyExtractorClass` to a class which implements the `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` interface. While the SpEL expression should usually suffice, more complex cases may use the custom implementation strategy. In that case, the property 'partitionKeyExtractorClass' can be set as follows:

```
spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass=com.example.MyKeyExtractor
spring.cloud.stream.bindings.output.producer.partitionCount=5
```

Once the message key is calculated, the partition selection process will determine the target partition as a value between `0` and `partitionCount - 1`. The default calculation, applicable in most scenarios, is based on the formula `key.hashCode() % partitionCount`. This can be customized on the binding, either by setting a SpEL expression to be evaluated against the 'key' (via the `partitionSelectorExpression` property) or by setting a `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` implementation (via the `partitionSelectorClass` property).

The binding level properties for 'partitionSelectorExpression' and 'partitionSelectorClass' can be specified similar to the way 'partitionKeyExpression' and 'partitionKeyExtractorClass' properties are specified in the above examples. Additional properties can be configured for more advanced scenarios, as described in the following section.

Spring-managed custom `PartitionKeyExtractorClass` implementations

In the example above, a custom strategy such as `MyKeyExtractor` is instantiated by the Spring Cloud Stream directly. In some cases, it is necessary for such a custom strategy implementation to be created as a Spring bean, for being able to be managed by Spring, so that it can perform dependency injection, property binding, etc. This can be done by configuring it as a @Bean in the application context and using the fully qualified class name as the bean's name, as in the following example.

```
@Bean(name="com.example.MyKeyExtractor")
public MyKeyExtractor extractor() {
    return new MyKeyExtractor();
}
```

As a Spring bean, the custom strategy benefits from the full lifecycle of a Spring bean. For example, if the implementation need access to the application context directly, it can make implement 'ApplicationContextAware'.

Configuring Input Bindings for Partitioning

An input binding (with the channel name `input`) is configured to receive partitioned data by setting its `partitioned` property, as well as the `instanceIndex` and `instanceCount` properties on the application itself, as in the following example:

```
spring.cloud.stream.bindings.input.consumer.partitioned=true
spring.cloud.stream.instanceIndex=3
spring.cloud.stream.instanceCount=5
```

The `instanceCount` value represents the total number of application instances between which the data need to be partitioned, and the `instanceIndex` must be a unique value across the multiple instances, between `0` and `instanceCount - 1`. The instance index helps each application instance to identify the unique partition (or, in the case of Kafka, the partition set) from which it receives data. It is important to set both values correctly in order to ensure that all of the data is consumed and that the application instances receive mutually exclusive datasets.

While a scenario which using multiple instances for partitioned data processing may be complex to set up in a standalone case, Spring Cloud Dataflow can simplify the process significantly by populating both the input and output values correctly as well as relying on the runtime infrastructure to provide information about the instance index and instance count.

Testing

Spring Cloud Stream provides support for testing your microservice applications without connecting to a messaging system. You can do that by using the `TestSupportBinder` provided by the `spring-cloud-stream-test-support` library, which can be added as a test dependency to the application:

XML

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```

> **NOTE** The `TestSupportBinder` uses the Spring Boot autoconfiguration mechanism to supersede the other binders found on the classpath. Therefore, when adding a binder as a dependency, make sure that the `test` scope is being used.

The `TestSupportBinder` allows users to interact with the bound channels and inspect what messages are sent and received by the application

For outbound message channels, the `TestSupportBinder` registers a single subscriber and retains the messages emitted by the application in a `MessageCollector`. They can be retrieved during tests and have assertions made against them.

The user can also send messages to inbound message channels, so that the consumer application can consume the messages. The following example shows how to test both input and output channels on a processor.

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ExampleTest {

  @Autowired
  private Processor processor;

  @Autowired
  private MessageCollector messageCollector;

  @Test
  @SuppressWarnings("unchecked")
  public void testWiring() {
    Message<String> message = new GenericMessage<>("hello");
    processor.input().send(message);
    Message<String> received = (Message<String>) messageCollector.forChannel(processor.output()).poll();
    assertThat(received.getPayload(), equalTo("hello world"));
  }


  @SpringBootApplication
  @EnableBinding(Processor.class)
  public static class MyProcessor {

    @Autowired
    private Processor channels;

    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public String transform(String in) {
      return in + " world";
    }
  }
}
```

In the example above, we are creating an application that has an input and an output channel, bound through the `Processor` interface. The bound interface is injected into the test so we can have access to both channels. We are sending a message on the input channel and we are using the `MessageCollector` provided by Spring Cloud Stream's test support to capture the message has been sent to the output channel as a result. Once we have received the message, we can validate that the component functions correctly.

Health Indicator

Spring Cloud Stream provides a health indicator for binders. It is registered under the name of `binders` and can be enabled or disabled by setting the `management.health.binders.enabled` property.

Metrics Emitter

Spring Cloud Stream provides a module called `spring-cloud-stream-metrics` that can be used to emit any available metric from Spring Boot metrics endpoint (https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html) to a named channel. This module allow operators to collect metrics from stream applications without relying on polling their endpoints.

The module is activated when you set the destination name for metrics binding, e.g. `spring.cloud.stream.bindings.applicationMetrics.destination=<DESTINATION_NAME>`. `applicationMetrics` can be configured in a similar fashion to any other producer binding. The default `contentType` setting of `applicationMetrics` is `application/json`.

The following properties can be used for customizing the emission of metrics:

spring.cloud.stream.metrics.key

The name of the metric being emitted. Should be an unique value per application.

Default

`${spring.application.name:${vcap.application.name:${spring.config.name:application}}}`

spring.cloud.stream.metrics.prefix

Prefix string to be prepended to the metrics key.

Default: ``

spring.cloud.stream.metrics.properties

Just like the `includes` option, it allows white listing application properties that will be added to the metrics payload

Default: null.

A detailed overview of the metrics export process can be found in the Spring Boot reference documentation (https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metric-writers). Spring Cloud Stream provides a metric exporter named `application` that can be configured via regular Spring Boot metrics configuration properties (https://github.com/spring-projects/spring-boot/blob/1.5.x/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics/export/TriggerProperties.java) .

The exporter can be configured either by using the global Spring Boot configuration settings for exporters, or by using exporter-specific properties. For using the global configuration settings, the properties should be prefixed by `spring.metric.export` (e.g. `spring.metric.export.includes=integration**`). These configuration options will apply to all exporters (unless they have been configured differently). Alternatively, if it is intended to use configuration settings that are different from the other exporters (e.g. for restricting the number of metrics published), the Spring Cloud Stream provided metrics exporter can be configured using the prefix `spring.metrics.export.triggers.application` (e.g. `spring.metrics.export.triggers.application.includes=integration**`).

> Due to Spring Boot's relaxed binding (https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html#boot-features-external-config-relaxed-binding) the value of a property being included can be slightly different than the original value.
>
> NOTE
>
> As a rule of thumb, the metric exporter will attempt to normalize all the properties in a consistent format using the dot notation (e.g. `JAVA_HOME` becomes `java.home`).
>
> The goal of normalization is to make downstream consumers of those metrics capable of receiving property names consistently, regardless of how they are set on the monitored application (`--spring.application.name` or `SPRING_APPLICATION_NAME` would always yield `spring.application.name`).

Below is a sample of the data published to the channel in JSON format by the following command:

```
java -jar time-source.jar \
    --spring.cloud.stream.bindings.applicationMetrics.destination=someMetrics \
    --spring.cloud.stream.metrics.properties=spring.application** \
    --spring.metrics.export.includes=integration.channel.input**,integration.channel.output**
```

The resulting JSON is:

JAVASCRIPT


```
{
   "name":"time-source",
   "metrics":[
      {
         "name":"integration.channel.output.errorRate.mean",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.max",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.min",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.stdev",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.count",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendCount",
         "value":6.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.mean",
         "value":0.994885872292989,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.max",
         "value":1.006247080013156,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.min",
         "value":1.0012035220116378,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.stdev",
         "value":6.505181111084848E-4,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.count",
         "value":6.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      }
   ],
   "createdTime":"2017-04-11T20:56:35.790Z",
   "properties":{
      "spring.application.name":"time-source",
      "spring.application.index":"0"
   }
}
```

Samples

For Spring Cloud Stream samples, please refer to the spring-cloud-stream-samples (https://github.com/spring-cloud/spring-cloud-stream-samples) repository on GitHub.

Getting Started

To get started with creating Spring Cloud Stream applications, visit the Spring Initializr (https://start.spring.io) and create a new Maven project named "GreetingSource". Select Spring Boot {supported-spring-boot-version} in the dropdown. In the *Search for dependencies* text box type `Stream Rabbit` or `Stream Kafka` depending on what binder you want to use.

Next, create a new class, `GreetingSource`, in the same package as the `GreetingSourceApplication` class. Give it the following code:

JAVA

```
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.integration.annotation.InboundChannelAdapter;

@EnableBinding(Source.class)
public class GreetingSource {

    @InboundChannelAdapter(Source.OUTPUT)
    public String greet() {
        return "hello world " + System.currentTimeMillis();
    }
}
```

The `@EnableBinding` annotation is what triggers the creation of Spring Integration infrastructure components. Specifically, it will create a Kafka connection factory, a Kafka outbound channel adapter, and the message channel defined inside the Source interface:

JAVA

```
public interface Source {

  String OUTPUT = "output";

  @Output(Source.OUTPUT)
  MessageChannel output();

}
```

The auto-configuration also creates a default poller, so that the `greet()` method will be invoked once per second. The standard Spring Integration `@InboundChannelAdapter` annotation sends a message to the source's output channel, using the return value as the payload of the message.

To test-drive this setup, run a Kafka message broker. An easy way to do this is to use a Docker image:

```
# On OS X
$ docker run -p 2181:2181 -p 9092:9092 --env ADVERTISED_HOST=`docker-machine ip \`docker-machine active\`` --env
ADVERTISED_PORT=9092 spotify/kafka

# On Linux
$ docker run -p 2181:2181 -p 9092:9092 --env ADVERTISED_HOST=localhost --env ADVERTISED_PORT=9092 spotify/kafka
```

Build the application:

```
./mvnw clean package
```

The consumer application is coded in a similar manner. Go back to Initializr and create another project, named LoggingSink. Then create a new class, `LoggingSink`, in the same package as the class `LoggingSinkApplication` and with the following code:

JAVA

```
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;

@EnableBinding(Sink.class)
public class LoggingSink {

    @StreamListener(Sink.INPUT)
    public void log(String message) {
        System.out.println(message);
    }
}
```

Build the application:

```
./mvnw clean package
```

To connect the GreetingSource application to the LoggingSink application, each application must share the same destination name. Starting up both applications as shown below, you will see the consumer application printing "hello world" and a timestamp to the console:

```
cd GreetingSource
java -jar target/GreetingSource-0.0.1-SNAPSHOT.jar --spring.cloud.stream.bindings.output.destination=mydest

cd LoggingSink
java -jar target/LoggingSink-0.0.1-SNAPSHOT.jar --server.port=8090 --
spring.cloud.stream.bindings.input.destination=mydest
```

(The different server port prevents collisions of the HTTP port used to service the Spring Boot Actuator endpoints in the two applications.)

The output of the LoggingSink application will look something like the following:

```
[           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090 (http)
[           main] com.example.LoggingSinkApplication       : Started LoggingSinkApplication in 6.828 seconds (JVM running
for 7.371)
hello world 1458595076731
hello world 1458595077732
hello world 1458595078733
hello world 1458595079734
hello world 1458595080735
```

Binder Implementations

Apache Kafka Binder

Usage

For using the Apache Kafka binder, you just need to add it to your Spring Cloud Stream application, using the following Maven coordinates:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

XML

Alternatively, you can also use the Spring Cloud Stream Kafka Starter.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

XML

Apache Kafka Binder Overview

A simplified diagram of how the Apache Kafka binder operates can be seen below.

kafka binder
Figure 13. Kafka Binder

The Apache Kafka Binder implementation maps each destination to an Apache Kafka topic. The consumer group maps directly to the same Apache Kafka concept. Partitioning also maps directly to Apache Kafka partitions as well.

Configuration Options

This section contains the configuration options used by the Apache Kafka binder.

For common configuration options and properties pertaining to binder, refer to the core documentation.

Kafka Binder Properties

spring.cloud.stream.kafka.binder.brokers

A list of brokers to which the Kafka binder will connect.

Default: `localhost`.

spring.cloud.stream.kafka.binder.defaultBrokerPort

`brokers` allows hosts specified with or without port information (e.g., `host1,host2:port2`). This sets the default port when no port is configured in the broker list.

Default: `9092`.

spring.cloud.stream.kafka.binder.zkNodes

A list of ZooKeeper nodes to which the Kafka binder can connect.

Default: `localhost`.

spring.cloud.stream.kafka.binder.defaultZkPort

`zkNodes` allows hosts specified with or without port information (e.g., `host1,host2:port2`). This sets the default port when no port is configured in the node list.

Default: `2181`.

spring.cloud.stream.kafka.binder.configuration

Key/Value map of client properties (both producers and consumer) passed to all clients created by the binder. Due to the fact that these properties will be used by both producers and consumers, usage should be restricted to common properties, especially security settings.

Default: Empty map.

spring.cloud.stream.kafka.binder.headers

The list of custom headers that will be transported by the binder.

Default: empty.

spring.cloud.stream.kafka.binder.offsetUpdateTimeWindow

The frequency, in milliseconds, with which offsets are saved. Ignored if `0`.

Default: `10000`.

spring.cloud.stream.kafka.binder.offsetUpdateCount

The frequency, in number of updates, which which consumed offsets are persisted. Ignored if `0`. Mutually exclusive with `offsetUpdateTimeWindow`.

Default: `0`.

spring.cloud.stream.kafka.binder.requiredAcks

The number of required acks on the broker.

Default: `1`.

spring.cloud.stream.kafka.binder.minPartitionCount

Effective only if `autoCreateTopics` or `autoAddPartitions` is set. The global minimum number of partitions that the binder will configure on topics on which it produces/consumes data. It can be superseded by the `partitionCount` setting of the producer or by the value of `instanceCount * concurrency` settings of the producer (if either is larger).

Default: `1`.

spring.cloud.stream.kafka.binder.replicationFactor

The replication factor of auto-created topics if `autoCreateTopics` is active.

Default: `1`.

spring.cloud.stream.kafka.binder.autoCreateTopics

If set to `true`, the binder will create new topics automatically. If set to `false`, the binder will rely on the topics being already configured. In the latter case, if the topics do not exist, the binder will fail to start. Of note, this setting is independent of the `auto.topic.create.enable` setting of the broker and it does not influence it: if the server is set to auto-create topics, they may be created as part of the metadata retrieval request, with default broker settings.

Default: `true`.

spring.cloud.stream.kafka.binder.autoAddPartitions

If set to `true`, the binder will create add new partitions if required. If set to `false`, the binder will rely on the partition size of the topic being already configured. If the partition count of the target topic is smaller than the expected value, the binder will fail to start.

Default: `false`.

spring.cloud.stream.kafka.binder.socketBufferSize

Size (in bytes) of the socket buffer to be used by the Kafka consumers.

Default: `2097152`.

Kafka Consumer Properties

The following properties are available for Kafka consumers only and must be prefixed with `spring.cloud.stream.kafka.bindings.<channelName>.consumer.`.

autoRebalanceEnabled

When `true`, topic partitions will be automatically rebalanced between the members of a consumer group. When `false`, each consumer will be assigned a fixed set of partitions based on `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex`. This requires both `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties to be set appropriately on each launched instance. The property `spring.cloud.stream.instanceCount` must typically be greater than 1 in this case.

Default: `true`.

autoCommitOffset

Whether to autocommit offsets when a message has been processed. If set to `false`, a header with the key `kafka_acknowledgment` of the type `org.springframework.kafka.support.Acknowledgment` header will be present in the inbound message. Applications may use this header for acknowledging messages. See the examples section for details. When this property is set to `false`, Kafka binder will set the ack mode to `org.springframework.kafka.listener.AbstractMessageListenerContainer.AckMode.MANUAL`.

Default: `true`.

autoCommitOnError

Effective only if `autoCommitOffset` is set to `true`. If set to `false` it suppresses auto-commits for messages that result in errors, and will commit only for successful messages, allows a stream to automatically replay from the last successfully processed message, in case of persistent failures. If set to `true`, it will always auto-commit (if auto-commit is enabled). If not set (default), it effectively has the same value as `enableDlq`, auto-committing erroneous messages if they are sent to a DLQ, and not committing them otherwise.

Default: not set.

recoveryInterval

The interval between connection recovery attempts, in milliseconds.

Default: `5000`.

resetOffsets

Whether to reset offsets on the consumer to the value provided by `startOffset`.

Default: `false`.

startOffset

The starting offset for new groups, or when `resetOffsets` is `true`. Allowed values: `earliest`, `latest`. If the consumer group is set explicitly for the consumer 'binding' (via `spring.cloud.stream.bindings.<channelName>.group`), then 'startOffset' is set to `earliest`; otherwise it is set to `latest` for the `anonymous` consumer group.

Default: null (equivalent to `earliest`).

enableDlq

When set to true, it will send enable DLQ behavior for the consumer. By default, messages that result in errors will be forwarded to a topic named `error.<destination>.<group>`. The DLQ topic name can be configurable via the property `dlqName`. This provides an alternative option to the more common Kafka replay scenario for the case when the number of errors is relatively small and replaying the entire original topic may be too cumbersome.

Default: `false`.

configuration

Map with a key/value pair containing generic Kafka consumer properties.

Default: Empty map.

dlqName

The name of the DLQ topic to receive the error messages.

Default: null (If not specified, messages that result in errors will be forwarded to a topic named `error.<destination>.<group>`).

Kafka Producer Properties

The following properties are available for Kafka producers only and must be prefixed with `spring.cloud.stream.kafka.bindings.<channelName>.producer.`.

bufferSize

Upper limit, in bytes, of how much data the Kafka producer will attempt to batch before sending.

Default: `16384`.

sync

Whether the producer is synchronous.

Default: `false`.

batchTimeout

How long the producer will wait before sending in order to allow more messages to accumulate in the same batch. (Normally the producer does not wait at all, and simply sends all the messages that accumulated while the previous send was in progress.) A non-zero value may increase throughput at the expense of latency.

Default: `0`.

messageKeyExpression

A SpEL expression evaluated against the outgoing message used to populate the key of the produced Kafka message. For example `headers.key` or `payload.myKey`.

Default: `none`.

configuration

Map with a key/value pair containing generic Kafka producer properties.

Default: Empty map.

NOTE

The Kafka binder will use the `partitionCount` setting of the producer as a hint to create a topic with the given partition count (in conjunction with the `minPartitionCount`, the maximum of the two being the value being used). Exercise caution when configuring both `minPartitionCount` for a binder and `partitionCount` for an application, as the larger value will be used. If a topic already exists with a smaller partition count and `autoAddPartitions` is disabled (the default), then the binder will fail to start. If a topic already exists with a smaller partition count and `autoAddPartitions` is enabled, new partitions will be added. If a topic already exists with a larger number of partitions than the maximum of (`minPartitionCount` and `partitionCount`), the existing partition count will be used.

Usage examples

In this section, we illustrate the use of the above properties for specific scenarios.

Example: Setting `autoCommitOffset` false and relying on manual acking.

This example illustrates how one may manually acknowledge offsets in a consumer application.

This example requires that `spring.cloud.stream.kafka.bindings.input.consumer.autoCommitOffset` is set to false. Use the corresponding input channel name for your example.

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class ManuallyAcknowdledgingConsumer {

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

 @StreamListener(Sink.INPUT)
 public void process(Message<?> message) {
     Acknowledgment acknowledgment = message.getHeaders().get(KafkaHeaders.ACKNOWLEDGMENT, Acknowledgment.class);
     if (acknowledgment != null) {
         System.out.println("Acknowledgment provided");
         acknowledgment.acknowledge();
     }
 }
}
```

Example: security configuration

Apache Kafka 0.9 supports secure connections between client and brokers. To take advantage of this feature, follow the guidelines in the Apache Kafka Documentation (https://kafka.apache.org/090/documentation.html#security_configclients) as well as the Kafka 0.9 security guidelines from the Confluent documentation (http://docs.confluent.io/2.0.0/kafka/security.html). Use the

`spring.cloud.stream.kafka.binder.configuration` option to set security properties for all clients created by the binder.

For example, for setting `security.protocol` to `SASL_SSL`, set:

```
spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_SSL
```

All the other security properties can be set in a similar manner.

When using Kerberos, follow the instructions in the reference documentation (https://kafka.apache.org/090/documentation.html#security_sasl_clientconfig) for creating and referencing the JAAS configuration.

Spring Cloud Stream supports passing JAAS configuration information to the application using a JAAS configuration file and using Spring Boot properties.

Using JAAS configuration files

The JAAS, and (optionally) krb5 file locations can be set for Spring Cloud Stream applications by using system properties. Here is an example of launching a Spring Cloud Stream application with SASL and Kerberos using a JAAS configuration file:

```
java -Djava.security.auth.login.config=/path.to/kafka_client_jaas.conf -jar log.jar \
  --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \
  --spring.cloud.stream.kafka.binder.zkNodes=secure.zookeeper:2181 \
  --spring.cloud.stream.bindings.input.destination=stream.ticktock \
  --spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT
```

Using Spring Boot properties

As an alternative to having a JAAS configuration file, Spring Cloud Stream provides a mechanism for setting up the JAAS configuration for Spring Cloud Stream applications using Spring Boot properties.

The following properties can be used for configuring the login context of the Kafka client.

spring.cloud.stream.kafka.binder.jaas.loginModule

The login module name. Not necessary to be set in normal cases.

Default: `com.sun.security.auth.module.Krb5LoginModule`.

spring.cloud.stream.kafka.binder.jaas.controlFlag

The control flag of the login module.

Default: `required`.

spring.cloud.stream.kafka.binder.jaas.options

Map with a key/value pair containing the login module options.

Default: Empty map.

Here is an example of launching a Spring Cloud Stream application with SASL and Kerberos using Spring Boot configuration properties:

```
java --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \
  --spring.cloud.stream.kafka.binder.zkNodes=secure.zookeeper:2181 \
  --spring.cloud.stream.bindings.input.destination=stream.ticktock \
  --spring.cloud.stream.kafka.binder.autoCreateTopics=false \
  --spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT \
  --spring.cloud.stream.kafka.binder.jaas.options.useKeyTab=true \
  --spring.cloud.stream.kafka.binder.jaas.options.storeKey=true \
  --spring.cloud.stream.kafka.binder.jaas.options.keyTab=/etc/security/keytabs/kafka_client.keytab \
  --spring.cloud.stream.kafka.binder.jaas.options.principal=kafka-client-1@EXAMPLE.COM
```

This represents the equivalent of the following JAAS file:

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_client.keytab"
    principal="kafka-client-1@EXAMPLE.COM";
};
```

If the topics required already exist on the broker, or will be created by an administrator, autocreation can be turned off and only client JAAS properties need to be sent. As an alternative to setting `spring.cloud.stream.kafka.binder.autoCreateTopics` you can simply remove the broker dependency from the application. See Excluding Kafka broker jar from the classpath of the binder based application for details.

NOTE

Do not mix JAAS configuration files and Spring Boot properties in the same application. If the `-Djava.security.auth.login.config` system property is already present, Spring Cloud Stream will ignore the Spring Boot properties.

NOTE

Exercise caution when using the `autoCreateTopics` and `autoAddPartitions` if using Kerberos. Usually applications may use principals that do not have administrative rights in Kafka and Zookeeper, and relying on Spring Cloud Stream to create/modify topics may fail. In secure environments, we strongly recommend creating topics and managing ACLs administratively using Kafka tooling.

Using the binder with Apache Kafka 0.10

The default Kafka support in Spring Cloud Stream Kafka binder is for Kafka version 0.10.1.1. The binder also supports connecting to other 0.10 based versions and 0.9 clients. In order to do this, when you create the project that contains your application, include `spring-cloud-starter-stream-kafka` as you normally would do for the default binder. Then add these dependencies at the top of the `<dependencies>` section in the pom.xml file to override the dependencies.

Here is an example for downgrading your application to 0.10.0.1. Since it is still on the 0.10 line, the default `spring-kafka` and `spring-integration-kafka` versions can be retained.

XML

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.11</artifactId>
  <version>0.10.0.1</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.10.0.1</version>
</dependency>
```

Here is another example of using 0.9.0.1 version.

```xml
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>1.0.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-kafka</artifactId>
  <version>2.0.1.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.11</artifactId>
  <version>0.9.0.1</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.9.0.1</version>
</dependency>
```

NOTE

The versions above are provided only for the sake of the example. For best results, we recommend using the most recent 0.10-compatible versions of the projects.

Excluding Kafka broker jar from the classpath of the binder based application

The Apache Kafka Binder uses the administrative utilities which are part of the Apache Kafka server library to create and reconfigure topics. If the inclusion of the Apache Kafka server library and its dependencies is not necessary at runtime because the application will rely on the topics being configured administratively, the Kafka binder allows for Apache Kafka server dependency to be excluded from the application.

If you use non default versions for Kafka dependencies as advised above, all you have to do is not to include the kafka broker dependency. If you use the default Kafka version, then ensure that you exclude the kafka broker jar from the `spring-cloud-starter-stream-kafka` dependency as following.

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka_2.11</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

If you exclude the Apache Kafka server dependency and the topic is not present on the server, then the Apache Kafka broker will create the topic if auto topic creation is enabled on the server. Please keep in mind that if you are relying on this, then the Kafka server will use the default number of partitions and replication factors. On the other hand, if auto topic creation is disabled on the server, then care must be taken before running the application to create the topic with the desired number of partitions.

If you want to have full control over how partitions are allocated, then leave the default settings as they are, i.e. do not exclude the kafka broker jar and ensure that `spring.cloud.stream.kafka.binder.autoCreateTopics` is set to `true`, which is the default.

Dead-Letter Topic Processing

Because it can't be anticipated how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original topic. However, if the problem is a permanent issue, that could cause an infinite loop. The following `spring-boot` application is an example of how to route those messages back to the original topic, but moves them to a third "parking lot" topic after three attempts. The application is simply another spring-cloud-stream application that reads from the dead-letter topic. It terminates when no messages are received for 5 seconds.

The examples assume the original destination is `so8400out` and the consumer group is `so8400`.

There are several considerations.

- Consider only running the rerouting when the main application is not running. Otherwise, the retries for transient errors will be used up very quickly.
- Alternatively, use a two-stage approach - use this application to route to a third topic, and another to route from there back to the main topic.
- Since this technique uses a message header to keep track of retries, it won't work with `headerMode=raw`. In that case, consider adding some data to the payload (that can be ignored by the main application).
- `x-retries` has to be added to the `headers` property `spring.cloud.stream.kafka.binder.headers=x-retries` on both this, and the main application so that the header is transported between the applications.
- Since kafka is publish/subscribe, replayed messages will be sent to each consumer group, even those that successfully processed a message the first time around.

application.properties

```
spring.cloud.stream.bindings.input.group=so8400replay
spring.cloud.stream.bindings.input.destination=error.so8400out.so8400

spring.cloud.stream.bindings.output.destination=so8400out
spring.cloud.stream.bindings.output.producer.partitioned=true

spring.cloud.stream.bindings.parkingLot.destination=so8400in.parkingLot
spring.cloud.stream.bindings.parkingLot.producer.partitioned=true

spring.cloud.stream.kafka.binder.configuration.auto.offset.reset=earliest

spring.cloud.stream.kafka.binder.headers=x-retries
```

Application

JAVA

```
@SpringBootApplication
@EnableBinding(TwoOutputProcessor.class)
public class ReRouteDlqKApplication implements CommandLineRunner {

    private static final String X_RETRIES_HEADER = "x-retries";

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

    private final AtomicInteger processed = new AtomicInteger();

    @Autowired
    private MessageChannel parkingLot;

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public Message<?> reRoute(Message<?> failed) {
        processed.incrementAndGet();
        Integer retries = failed.getHeaders().get(X_RETRIES_HEADER, Integer.class);
        if (retries == null) {
            System.out.println("First retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                    .setHeader(X_RETRIES_HEADER, new Integer(1))
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build();
        }
        else if (retries.intValue() < 3) {
            System.out.println("Another retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                    .setHeader(X_RETRIES_HEADER, new Integer(retries.intValue() + 1))
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build();
        }
        else {
            System.out.println("Retries exhausted for " + failed);
            parkingLot.send(MessageBuilder.fromMessage(failed)
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build());
        }
        return null;
    }

    @Override
    public void run(String... args) throws Exception {
        while (true) {
            int count = this.processed.get();
            Thread.sleep(5000);
            if (count == this.processed.get()) {
                System.out.println("Idle, terminating");
                return;
            }
        }
    }

    public interface TwoOutputProcessor extends Processor {

        @Output("parkingLot")
        MessageChannel parkingLot();

    }

}
```

RabbitMQ Binder

Usage

For using the RabbitMQ binder, you just need to add it to your Spring Cloud Stream application, using the following Maven coordinates:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

XML

Alternatively, you can also use the Spring Cloud Stream RabbitMQ Starter.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

XML

RabbitMQ Binder Overview

A simplified diagram of how the RabbitMQ binder operates can be seen below.

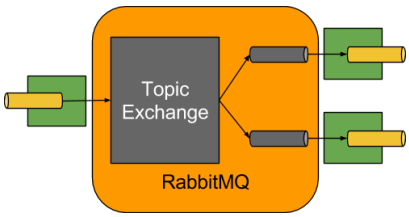


Figure 14. RabbitMQ Binder

The RabbitMQ Binder implementation maps each destination to a `TopicExchange`. For each consumer group, a `Queue` will be bound to that `TopicExchange`. Each consumer instance have a corresponding RabbitMQ `Consumer` instance for its group's `Queue`. For partitioned producers/consumers the queues are suffixed with the partition index and use the partition index as routing key.

Using the `autoBindDlq` option, you can optionally configure the binder to create and configure dead-letter queues (DLQs) (and a dead-letter exchange `DLX`). The dead letter queue has the name of the destination, appended with `.dlq`. If retry is enabled (`maxAttempts > 1`) failed messages will be delivered to the DLQ. If retry is disabled (`maxAttempts = 1`), you should set `requeueRejected` to `false` (default) so that a failed message will be routed to the DLQ, instead of being requeued. In addition, `republishToDlq` causes the binder to publish a failed message to the DLQ (instead of rejecting it); this enables additional information to be added to the message in headers, such as the stack trace in the `x-exception-stacktrace` header. This option does not need retry enabled; you can republish a failed message after just one attempt. Starting with *version 1.2*, you can configure the delivery mode of republished messsages; see property `republishDeliveryMode`.

IMPORTANT	Setting `requeueRejected` to `true` will cause the message to be requeued and redelivered continually, which is likely not what you want unless the failure issue is transient. In general, it's better to enable retry within the binder by setting `maxAttempts` to greater than one, or set `republishToDlq` to `true`.

See RabbitMQ Binder Properties for more information about these properties.

The framework does not provide any standard mechanism to consume dead-letter messages (or to re-route them back to the primary queue). Some options are described in Dead-Letter Queue Processing.

NOTE	When **multiple** RabbitMQ binders are used in a Spring Cloud Stream application, it is important to disable 'RabbitAutoConfiguration' to avoid the same configuration from RabbitAutoConfiguration being applied to the two binders.

Configuration Options

This section contains settings specific to the RabbitMQ Binder and bound channels.

For general binding configuration options and properties, please refer to the Spring Cloud Stream core documentation (https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream-docs/src/main/asciidoc/spring-cloud-stream-overview.adoc#configuration-options)

.

RabbitMQ Binder Properties

By default, the RabbitMQ binder uses Spring Boot's `ConnectionFactory`, and it therefore supports all Spring Boot configuration options for RabbitMQ. (For reference, consult the Spring Boot documentation (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties).) RabbitMQ configuration options use the `spring.rabbitmq` prefix.

In addition to Spring Boot options, the RabbitMQ binder supports the following properties:

spring.cloud.stream.rabbit.binder.adminAddresses

A comma-separated list of RabbitMQ management plugin URLs. Only used when `nodes` contains more than one entry. Each entry in this list must have a corresponding entry in `spring.rabbitmq.addresses`.

Default: empty.

spring.cloud.stream.rabbit.binder.nodes

A comma-separated list of RabbitMQ node names. When more than one entry, used to locate the server address where a queue is located. Each entry in this list must have a corresponding entry in `spring.rabbitmq.addresses`.

Default: empty.

spring.cloud.stream.rabbit.binder.compressionLevel

Compression level for compressed bindings. See `java.util.zip.Deflater`.

Default: `1` (BEST_LEVEL).

RabbitMQ Consumer Properties

The following properties are available for Rabbit consumers only and must be prefixed with `spring.cloud.stream.rabbit.bindings.<channelName>.consumer.`.

acknowledgeMode

The acknowledge mode.

Default: `AUTO`.

autoBindDlq

Whether to automatically declare the DLQ and bind it to the binder DLX.

Default: `false`.

bindingRoutingKey

The routing key with which to bind the queue to the exchange (if `bindQueue` is `true`). for partitioned destinations `-<instanceIndex>` will be appended.

Default: `#`.

bindQueue

Whether to bind the queue to the destination exchange; set to `false` if you have set up your own infrastructure and have previously created/bound the queue.

Default: `true`.

deadLetterQueueName

name of the DLQ

Default: `prefix+destination.dlq`

deadLetterExchange

a DLX to assign to the queue; if autoBindDlq is true

Default: 'prefix+DLX'

deadLetterRoutingKey

a dead letter routing key to assign to the queue; if autoBindDlq is true

Default: `destination`

declareExchange

Whether to declare the exchange for the destination.

Default: `true`.

delayedExchange

Whether to declare the exchange as a `Delayed Message Exchange` - requires the delayed message exchange plugin on the broker. The `x-delayed-type` argument is set to the `exchangeType`.

Default: `false`.

dlqDeadLetterExchange

if a DLQ is declared, a DLX to assign to that queue

Default: `none`

dlqDeadLetterRoutingKey

if a DLQ is declared, a dead letter routing key to assign to that queue; default none

Default: `none`

dlqExpires

how long before an unused dead letter queue is deleted (ms)

Default: `no expiration`

dlqMaxLength

maximum number of messages in the dead letter queue

Default: `no limit`

dlqMaxLengthBytes

maximum number of total bytes in the dead letter queue from all messages

Default: `no limit`

dlqMaxPriority

maximum priority of messages in the dead letter queue (0-255)

Default: `none`

dlqTtl

default time to live to apply to the dead letter queue when declared (ms)

Default: `no limit`

durableSubscription

Whether subscription should be durable. Only effective if `group` is also set.

Default: `true`.

exchangeAutoDelete

If `declareExchange` is true, whether the exchange should be auto-delete (removed after the last queue is removed).

Default: `true`.

exchangeDurable

If `declareExchange` is true, whether the exchange should be durable (survives broker restart).

Default: `true`.

exchangeType

The exchange type; `direct`, `fanout` or `topic` for non-partitioned destinations; `direct` or `topic` for partitioned destinations.

Default: `topic`.

expires

how long before an unused queue is deleted (ms)

Default: `no expiration`

headerPatterns

Patterns for headers to be mapped from inbound messages.

Default: `['*']` (all headers).

maxConcurrency

the maximum number of consumers

Default: `1`.

maxLength

maximum number of messages in the queue

Default: `no limit`

maxLengthBytes

maximum number of total bytes in the queue from all messages

Default: `no limit`

maxPriority

maximum priority of messages in the queue (0-255)

Default

`none`

prefetch

Prefetch count.

Default: `1`.

prefix

A prefix to be added to the name of the `destination` and queues.

Default: "".

recoveryInterval

The interval between connection recovery attempts, in milliseconds.

Default: `5000`.

requeueRejected

Whether delivery failures should be requeued when retry is disabled or republishToDlq is false.

Default: `false`.

republishDeliveryMode

When `republishToDlq` is `true`, specify the delivery mode of the republished message.

Default: `DeliveryMode.PERSISTENT`

republishToDlq

By default, messages which fail after retries are exhausted are rejected. If a dead-letter queue (DLQ) is configured, RabbitMQ will route the failed message (unchanged) to the DLQ. If set to `true`, the binder will republish failed messages to the DLQ with additional headers, including the exception message and stack trace from the cause of the final failure.

Default: false

transacted

Whether to use transacted channels.

Default: `false`.

ttl

default time to live to apply to the queue when declared (ms)

Default: `no limit`

txSize

The number of deliveries between acks.

Default: `1`.

Rabbit Producer Properties

The following properties are available for Rabbit producers only and must be prefixed with `spring.cloud.stream.rabbit.bindings.<channelName>.producer.`.

autoBindDlq

Whether to automatically declare the DLQ and bind it to the binder DLX.

Default: `false`.

batchingEnabled

Whether to enable message batching by producers.

Default: `false`.

batchSize

The number of messages to buffer when batching is enabled.

Default: `100`.

batchBufferLimit

Default: `10000`.

batchTimeout

Default: `5000`.

bindingRoutingKey

The routing key with which to bind the queue to the exchange (if `bindQueue` is `true`). Only applies to non-partitioned destinations. Only applies if `requiredGroups` are provided and then only to those groups.

Default: `#`.

bindQueue

Whether to bind the queue to the destination exchange; set to `false` if you have set up your own infrastructure and have previously created/bound the queue. Only applies if `requiredGroups` are provided and then only to those groups.

Default: `true`.

compress

Whether data should be compressed when sent.

Default: `false`.

deadLetterQueueName

name of the DLQ Only applies if `requiredGroups` are provided and then only to those groups.

Default: `prefix+destination.dlq`

deadLetterExchange

a DLX to assign to the queue; if autoBindDlq is true Only applies if `requiredGroups` are provided and then only to those groups.

Default: 'prefix+DLX'

deadLetterRoutingKey

a dead letter routing key to assign to the queue; if autoBindDlq is true Only applies if `requiredGroups` are provided and then only to those groups.

Default: `destination`

declareExchange

Whether to declare the exchange for the destination.

Default: `true`.

delay

A SpEL expression to evaluate the delay to apply to the message (`x-delay` header) - has no effect if the exchange is not a delayed message exchange.

Default: No `x-delay` header is set.

delayedExchange

Whether to declare the exchange as a `Delayed Message Exchange` - requires the delayed message exchange plugin on the broker. The `x-delayed-type` argument is set to the `exchangeType`.

Default: `false`.

deliveryMode

Delivery mode.

Default: `PERSISTENT`.

dlqDeadLetterExchange

if a DLQ is declared, a DLX to assign to that queue Only applies if `requiredGroups` are provided and then only to those groups.

Default: `none`

dlqDeadLetterRoutingKey

if a DLQ is declared, a dead letter routing key to assign to that queue; default none Only applies if `requiredGroups` are provided and then only to those groups.

Default: `none`

dlqExpires

how long before an unused dead letter queue is deleted (ms) Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no expiration`

dlqMaxLength

maximum number of messages in the dead letter queue Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

dlqMaxLengthBytes

maximum number of total bytes in the dead letter queue from all messages Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

dlqMaxPriority

maximum priority of messages in the dead letter queue (0-255) Only applies if `requiredGroups` are provided and then only to those groups.

Default: `none`

dlqTtl

default time to live to apply to the dead letter queue when declared (ms) Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

exchangeAutoDelete

If `declareExchange` is true, whether the exchange should be auto-delete (removed after the last queue is removed).

Default: `true`.

exchangeDurable

If `declareExchange` is true, whether the exchange should be durable (survives broker restart).

Default: `true`.

exchangeType

The exchange type; `direct`, `fanout` or `topic` for non-partitioned destinations; `direct` or `topic` for partitioned destinations.

Default: `topic`.

expires

how long before an unused queue is deleted (ms) Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no expiration`

headerPatterns

Patterns for headers to be mapped to outbound messages.

Default: `['*']` (all headers).

maxLength

maximum number of messages in the queue Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

maxLengthBytes

maximum number of total bytes in the queue from all messages Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

maxPriority

maximum priority of messages in the queue (0-255) Only applies if `requiredGroups` are provided and then only to those groups.

Default

`none`

prefix

A prefix to be added to the name of the `destination` exchange.

Default: "".

routingKeyExpression

A SpEL expression to determine the routing key to use when publishing messages.

Default: `destination` or `destination-<partition>` for partitioned destinations.

transacted

Whether to use transacted channels.

Default: `false`.

ttl

default time to live to apply to the queue when declared (ms) Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

NOTE

In the case of RabbitMQ, content type headers can be set by external applications. Spring Cloud Stream supports them as part of an extended internal protocol used for any type of transport (including transports, such as Kafka, that do not normally support headers).

Retry With the RabbitMQ Binder

Overview

When retry is enabled within the binder, the listener container thread is suspended for any back off periods that are configured. This might be important when strict ordering is required with a single consumer but for other use cases it prevents other messages from being processed on that thread. An alternative to using binder retry is to set up dead lettering with time to live on the dead-letter queue (DLQ), as well as dead-letter configuration on the DLQ itself. See RabbitMQ Binder Properties for more information about the properties discussed here. Example configuration to enable this feature:

- Set `autoBindDlq` to `true` - the binder will create a DLQ; you can optionally specify a name in `deadLetterQueueName`
- Set `dlqTtl` to the back off time you want to wait between redeliveries
- Set the `dlqDeadLetterExchange` to the default exchange - expired messages from the DLQ will be routed to the original queue since the default `deadLetterRoutingKey` is the queue name (`destination.group`)

To force a message to be dead-lettered, either throw an `AmqpRejectAndDontRequeueException`, or set `requeueRejected` to `true` and throw any exception.

The loop will continue without end, which is fine for transient problems but you may want to give up after some number of attempts. Fortunately, RabbitMQ provides the `x-death` header which allows you to determine how many cycles have occurred.

To acknowledge a message after giving up, throw an `ImmediateAcknowledgeAmqpException`.

Putting it All Together

```
---
spring.cloud.stream.bindings.input.destination=myDestination
spring.cloud.stream.bindings.input.group=consumerGroup
#disable binder retries
spring.cloud.stream.bindings.input.consumer.max-attempts=1
#dlx/dlq setup
spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-ttl=5000
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-dead-letter-exchange=
---
```

This configuration creates an exchange `myDestination` with queue `myDestination.consumerGroup` bound to a topic exchange with a wildcard routing key `#`. It creates a DLQ bound to a direct exchange `DLX` with routing key `myDestination.consumerGroup`. When messages are rejected, they are routed to the DLQ. After 5 seconds, the message expires and is routed to the original queue using the queue name as the routing key.

Spring Boot application

JAVA

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class XDeathApplication {

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

    @StreamListener(Sink.INPUT)
    public void listen(String in, @Header(name = "x-death", required = false) Map<?,?> death) {
        if (death != null && death.get("count").equals(3L)) {
            // giving up - don't send to DLX
            throw new ImmediateAcknowledgeAmqpException("Failed after 4 attempts");
        }
        throw new AmqpRejectAndDontRequeueException("failed");
    }

}
```

Notice that the count property in the `x-death` header is a `Long`.

Dead-Letter Queue Processing

Because it can't be anticipated how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original queue. However, if the problem is a permanent issue, that could cause an infinite loop. The following `spring-boot` application is an example of how to route those messages back to the original queue, but moves them to a third "parking lot" queue after three attempts. The second example utilizes the RabbitMQ Delayed Message Exchange (https://www.rabbitmq.com/blog/2015/04/16/scheduling-messages-with-rabbitmq/) to introduce a delay to the requeued message. In this example, the delay increases for each attempt. These examples use a `@RabbitListener` to receive messages from the DLQ, you could also use `RabbitTemplate.receive()` in a batch process.

The examples assume the original destination is `so8400in` and the consumer group is `so8400`.

Non-Partitioned Destinations

The first two examples are when the destination is **not** partitioned.

JAVA

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Integer retriesHeader = (Integer) failedMessage.getMessageProperties().getHeaders().get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            failedMessage.getMessageProperties().getHeaders().put(X_RETRIES_HEADER, retriesHeader + 1);
            this.rabbitTemplate.send(ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

JAVA

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    private static final String DELAY_EXCHANGE = "dlqReRouter";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            headers.put(X_RETRIES_HEADER, retriesHeader + 1);
            headers.put("x-delay", 5000 * retriesHeader);
            this.rabbitTemplate.send(DELAY_EXCHANGE, ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public DirectExchange delayExchange() {
        DirectExchange exchange = new DirectExchange(DELAY_EXCHANGE);
        exchange.setDelayed(true);
        return exchange;
    }

    @Bean
    public Binding bindOriginalToDelay() {
        return BindingBuilder.bind(new Queue(ORIGINAL_QUEUE)).to(delayExchange()).with(ORIGINAL_QUEUE);
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

Partitioned Destinations

With partitioned destinations, there is one DLQ for all partitions and we determine the original queue from the headers.

republishToDlq=false

When `republishToDlq` is `false`, RabbitMQ publishes the message to the DLX/DLQ with an `x-death` header containing information about the original destination.

JAVA

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_DEATH_HEADER = "x-death";

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @SuppressWarnings("unchecked")
    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            headers.put(X_RETRIES_HEADER, retriesHeader + 1);
            List<Map<String, ?>> xDeath = (List<Map<String, ?>>) headers.get(X_DEATH_HEADER);
            String exchange = (String) xDeath.get(0).get("exchange");
            List<String> routingKeys = (List<String>) xDeath.get(0).get("routing-keys");
            this.rabbitTemplate.send(exchange, routingKeys.get(0), failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

republishToDlq=true

When `republishToDlq` is `true`, the republishing recoverer adds the original exchange and routing key to headers.

JAVA

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    private static final String X_ORIGINAL_EXCHANGE_HEADER = RepublishMessageRecoverer.X_ORIGINAL_EXCHANGE;

    private static final String X_ORIGINAL_ROUTING_KEY_HEADER = RepublishMessageRecoverer.X_ORIGINAL_ROUTING_KEY;

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            headers.put(X_RETRIES_HEADER, retriesHeader + 1);
            String exchange = (String) headers.get(X_ORIGINAL_EXCHANGE_HEADER);
            String originalRoutingKey = (String) headers.get(X_ORIGINAL_ROUTING_KEY_HEADER);
            this.rabbitTemplate.send(exchange, originalRoutingKey, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

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. Starters are provided for an AMQP broker as the transport or for Kafka, but the same basic feature set (and some more depending on the transport) is on the roadmap for other transports.

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 github (https://github.com/spring-cloud/spring-cloud-config/tree/master/docs/src/main/asciidoc).

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` or `spring-cloud-starter-bus-kafka` to your dependency management and Spring Cloud takes care of the rest. Make sure the broker (RabbitMQ or Kafka) is available and configured: 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. for Rabbit

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 may be added in the future (ie. only service X nodes in data center Y, etc...). There are also some http endpoints under the `/bus/*` actuator namespace. There are currently two implemented. The first, `/bus/env`, sends key/value pairs to update each node's Spring Environment. The second, `/bus/refresh`, will reload each application's configuration, just as if they had all been pinged on their `/refresh` endpoint.

NOTE

The Bus starters cover Rabbit and Kafka, because those are the two most common implementations, but Spring Cloud Stream is quite flexible and binder will work combined with `spring-cloud-bus`.

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.

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.

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).

Customizing the Message Broker

Spring Cloud Bus uses Spring Cloud Stream (https://cloud.spring.io/spring-cloud-stream) to broadcast the messages so to get messages to flow you only need to include the binder implementation of your choice in the classpath. There are convenient starters specifically for the bus with AMQP (RabbitMQ) and Kafka (`spring-cloud-starter-bus-[amqp,kafka]`). Generally speaking Spring Cloud Stream relies on Spring Boot autoconfiguration conventions for configuring middleware, so for instance the AMQP broker address can be changed with `spring.rabbitmq.*` configuration properties. Spring Cloud Bus has a handful of native configuration properties in `spring.cloud.bus.*` (e.g. `spring.cloud.bus.destination` is the name of the topic to use the the externall middleware). Normally the defaults will suffice.

To lean more about how to customize the message broker settings consult the Spring Cloud Stream documentation.

Tracing Bus Events

Bus events (subclasses of `RemoteApplicationEvent`) can be traced by setting `spring.cloud.bus.trace.enabled=true`. If you do this then the Spring Boot `TraceRepository` (if it is present) will show each event sent and all the acks from each service instance. Example (from the `/trace` endpoint):

JSON

```
{
  "timestamp": "2015-11-26T10:24:44.411+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "stores:8081",
    "destination": "*:**"
  }
  },
  {
  "timestamp": "2015-11-26T10:24:41.864+0000",
  "info": {
    "signal": "spring.cloud.bus.sent",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
  },
  {
  "timestamp": "2015-11-26T10:24:41.862+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
}
```

This trace shows that a `RefreshRemoteApplicationEvent` was sent from `customers:9000`, broadcast to all services, and it was received (acked) by `customers:9000` and `stores:8081`.

To handle the ack signals yourself you could add an `@EventListener` for the `AckRemoteApplicationEvent` and `SentApplicationEvent` types to your app (and enable tracing). Or you could tap into the `TraceRepository` and mine the data from there.

NOTE

Any Bus application can trace acks, but sometimes it will be useful to do this in a central service that can do more complex queries on the data. Or forward it to a specialized tracing service.

Broadcasting Your Own Events

The Bus can carry any event of type `RemoteApplicationEvent`, but the default transport is JSON and the deserializer needs to know which types are going to be used ahead of time. To register a new type it needs to be in a subpackage of `org.springframework.cloud.bus.event`.

To customise the event name you can use `@JsonTypeName` on your custom class or rely on the default strategy which is to use the simple name of the class. Note that both the producer and the consumer will need access to the class definition.

Registering events in custom packages

If you cannot or don't want to use a subpackage of `org.springframework.cloud.bus.event` for your custom events, you must specify which packages to scan for events of type `RemoteApplicationEvent` using `@RemoteApplicationEventScan`. Packages specified with `@RemoteApplicationEventScan` include subpackages.

For example, if you have a custom event called `FooEvent`:

JAVA

```
package com.acme;

public class FooEvent extends RemoteApplicationEvent {
    ...
}
```

you can register this event with the deserializer in the following way:

JAVA

```
package com.acme;

@Configuration
@RemoteApplicationEventScan
public class BusConfiguration {
    ...
}
```

Without specifying a value, the package of the class where `@RemoteApplicationEventScan` is used will be registered. In this example `com.acme` will be registered using the package of `BusConfiguration`.

You can also explicitly specify the packages to scan using the `value`, `basePackages` or `basePackageClasses` properties on `@RemoteApplicationEventScan`. For example:

JAVA

```
package com.acme;

@Configuration
//@RemoteApplicationEventScan({"com.acme", "foo.bar"})
//@RemoteApplicationEventScan(basePackages = {"com.acme", "foo.bar", "fizz.buzz"})
@RemoteApplicationEventScan(basePackageClasses = BusConfiguration.class)
public class BusConfiguration {
    ...
}
```

All examples of `@RemoteApplicationEventScan` above are equivalent, in that the `com.acme` package will be registered by explicitly specifying the packages on `@RemoteApplicationEventScan`. Note, you can specify multiple base packages to scan.

Spring Cloud Sleuth

Adrian Cole, Spencer Gibb, Marcin Grzejszczak, Dave Syer

Dalston.SR3

Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud (https://cloud.spring.io).

Terminology

Spring Cloud Sleuth borrows Dapper's (http://research.google.com/pubs/pub36356.html) terminology.

Span: The basic unit of work. For example, sending an RPC is a new span, as is sending a response to an RPC. Span's are identified by a unique 64-bit ID for the span and another 64-bit ID for the trace the span is a part of. Spans also have other data, such as descriptions, timestamped events, key-value annotations (tags), the ID of the span that caused them, and process ID's (normally IP address).

Spans are started and stopped, and they keep track of their timing information. Once you create a span, you must stop it at some point in the future.

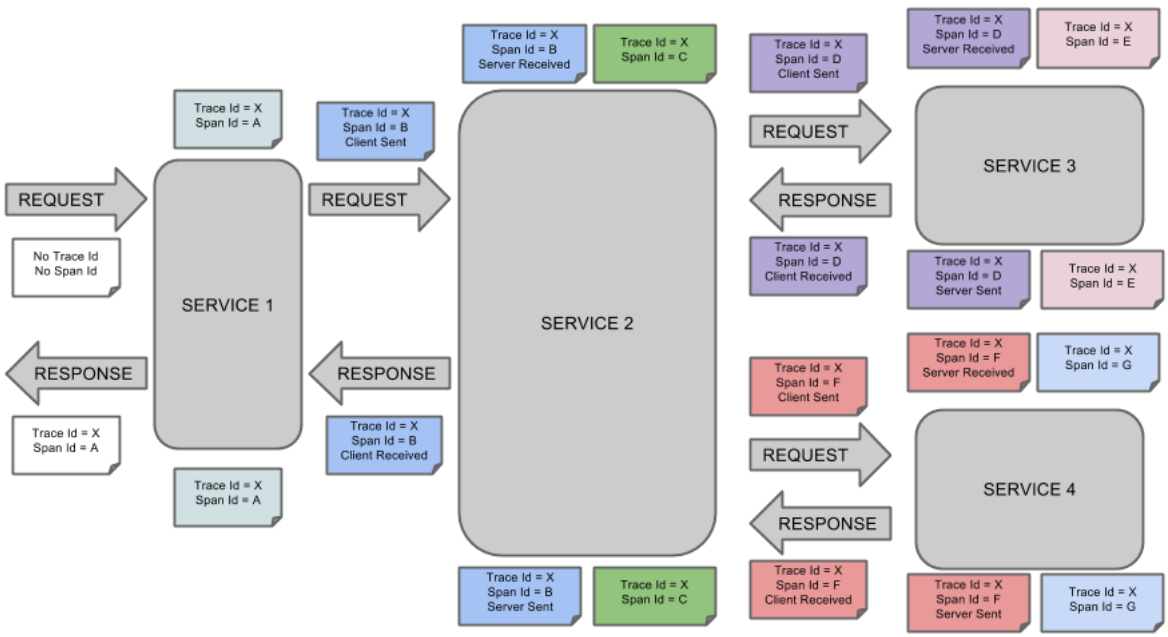
> TIP The initial span that starts a trace is called a `root span`. The value of span id of that span is equal to trace id.

Trace: A set of spans forming a tree-like structure. For example, if you are running a distributed big-data store, a trace might be formed by a put request.

Annotation: is used to record existence of an event in time. Some of the core annotations used to define the start and stop of a request are:

- **cs** - Client Sent - The client has made a request. This annotation depicts the start of the span.
- **sr** - Server Received - The server side got the request and will start processing it. If one subtracts the cs timestamp from this timestamp one will receive the network latency.
- **ss** - Server Sent - Annotated upon completion of request processing (when the response got sent back to the client). If one subtracts the sr timestamp from this timestamp one will receive the time needed by the server side to process the request.
- **cr** - Client Received - Signifies the end of the span. The client has successfully received the response from the server side. If one subtracts the cs timestamp from this timestamp one will receive the whole time needed by the client to receive the response from the server.

Visualization of what **Span** and **Trace** will look in a system together with the Zipkin annotations:

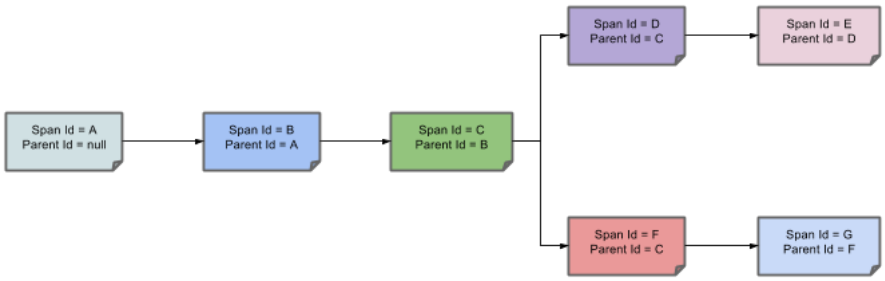


Each color of a note signifies a span (7 spans - from **A** to **G**). If you have such information in the note:

```
Trace Id = X
Span Id = D
Client Sent
```

That means that the current span has **Trace-Id** set to **X**, **Span-Id** set to **D**. It also has emitted **Client Sent** event.

This is how the visualization of the parent / child relationship of spans would look like:

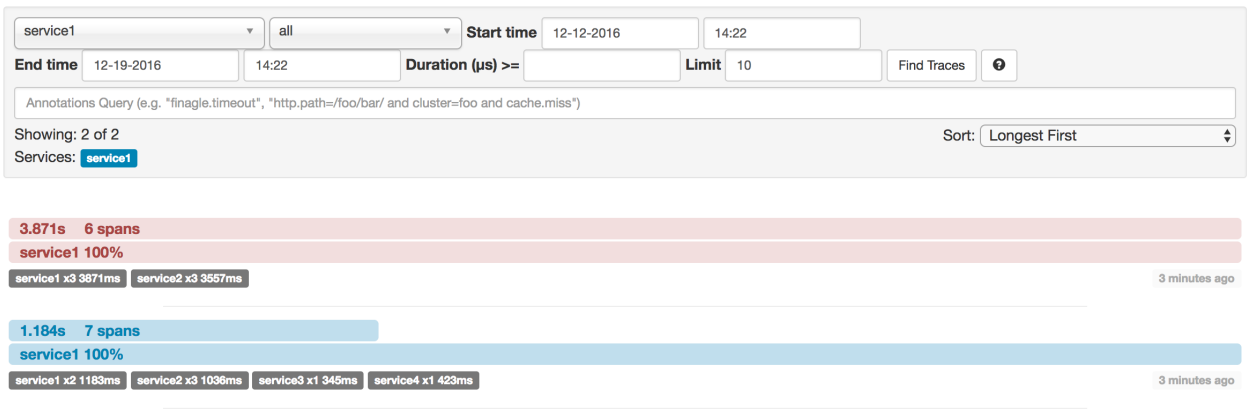


Purpose

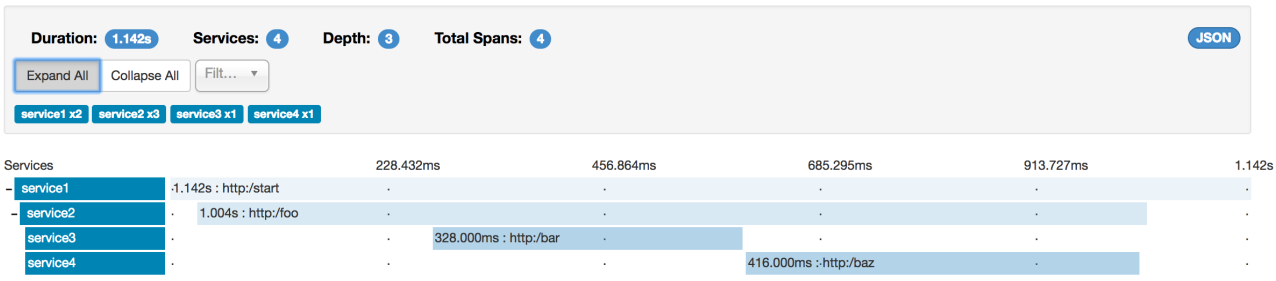
In the following sections the example from the image above will be taken into consideration.

Distributed tracing with Zipkin

Altogether there are 7 **spans** . If you go to traces in Zipkin you will see this number in the second trace:



However if you pick a particular trace then you will see **4 spans**:



NOTE

When picking a particular trace you will see merged spans. That means that if there were 2 spans sent to Zipkin with Server Received and Server Sent / Client Received and Client Sent annotations then they will presented as a single span.

Why is there a difference between the 7 and 4 spans in this case?

- 2 spans come from `http:/start` span. It has the Server Received (SR) and Server Sent (SS) annotations.
- 2 spans come from the RPC call from `service1` to `service2` to the `http:/foo` endpoint. It has the Client Sent (CS) and Client Received (CR) annotations on `service1` side. It also has Server Received (SR) and Server Sent (SS) annotations on the `service2` side. Physically there are 2 spans but they form 1 logical span related to an RPC call.

- 2 spans come from the RPC call from `service2` to `service3` to the `http:/bar` endpoint. It has the Client Sent (CS) and Client Received (CR) annotations on `service2` side. It also has Server Received (SR) and Server Sent (SS) annotations on the `service3` side. Physically there are 2 spans but they form 1 logical span related to an RPC call.
- 2 spans come from the RPC call from `service2` to `service4` to the `http:/baz` endpoint. It has the Client Sent (CS) and Client Received (CR) annotations on `service2` side. It also has Server Received (SR) and Server Sent (SS) annotations on the `service4` side. Physically there are 2 spans but they form 1 logical span related to an RPC call.

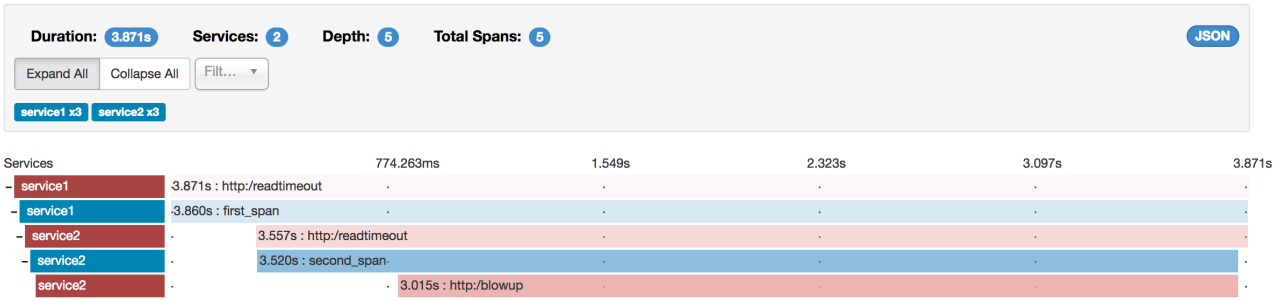
So if we count the physical spans we have **1** from `http:/start` , **2** from `service1` calling `service2` , **2** form `service2` calling `service3` and **2** from `service2` calling `service4` . Altogether 7 spans.

Logically we see the information of **Total Spans: 4** because we have **1** span related to the incoming request to `service1` and **3** spans related to RPC calls.

Visualizing errors

Zipkin allows you to visualize errors in your trace. When an exception was thrown and wasn't caught then we're setting proper tags on the span which Zipkin can properly colorize. You could see in the list of traces one trace that was in red color. That's because there was an exception thrown.

If you click that trace then you'll see a similar picture



Then if you click on one of the spans you'll see the following

service2.http:/readtimeout: 3.557s

AKA: service1,service2

Date Time	Relative Time	Annotation	Address
19/12/2016, 14:19:23	307.000ms	Client Send	127.0.0.1:8081 (service1)
19/12/2016, 14:19:23	310.000ms	Server Receive	127.0.0.1:8082 (service2)
19/12/2016, 14:19:26	3.836s	Server Send	127.0.0.1:8082 (service2)
19/12/2016, 14:19:27	3.864s	Client Receive	127.0.0.1:8081 (service1)

Key	Value
error	Request processing failed; nested exception is org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://localhost:8082/blowup": Read timed out; nested exception is java.net.SocketTimeoutException: Read timed out
http.host	localhost
http.method	GET
http.path	/readtimeout
http.status_code	500
http.url	http://localhost:8082/readtimeout
mvc.controller.class	BasicErrorController
mvc.controller.method	error

As you can see you can easily see the reason for an error and the whole stacktrace related to it.

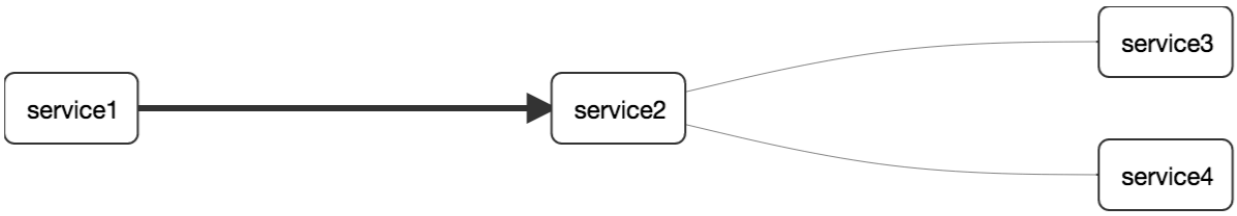
Live examples

(https://docssleuth-zipkin-server.cfapps.io/)

Pivotal **Web Services**

Click Pivotal Web Services icon to see it live!Click Pivotal Web Services icon to see it live!

The dependency graph in Zipkin would look like this:



(https://docssleuth-zipkin-server.cfapps.io/dependency)

Pivotal **Web Services**

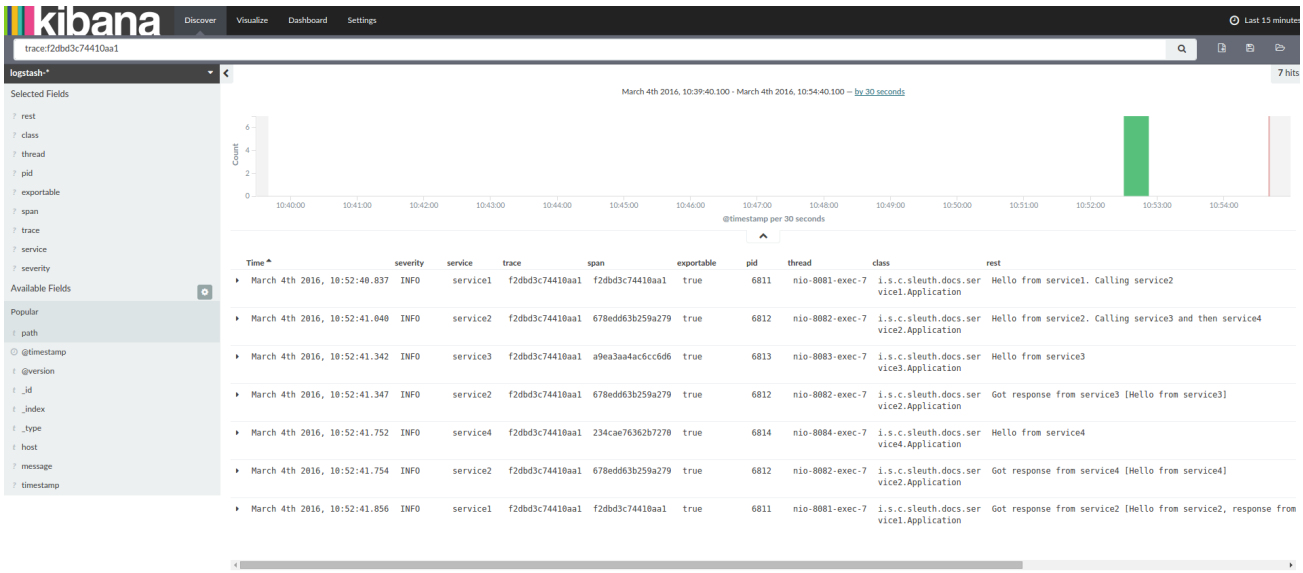
Click Pivotal Web Services icon to see it live!Click Pivotal Web Services icon to see it live!

Log correlation

When grepping the logs of those four applications by trace id equal to e.g. `2485ec27856c56f4` one would get the following:

```
service1.log:2016-02-26 11:15:47.561  INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1]
i.s.c.sleuth.docs.service1.Application   : Hello from service1. Calling service2
service2.log:2016-02-26 11:15:47.710  INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Hello from service2. Calling service3 and then service4
service3.log:2016-02-26 11:15:47.895  INFO [service3,2485ec27856c56f4,1210be13194bfe5,true] 68060 --- [nio-8083-exec-1]
i.s.c.sleuth.docs.service3.Application   : Hello from service3
service2.log:2016-02-26 11:15:47.924  INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Got response from service3 [Hello from service3]
service4.log:2016-02-26 11:15:48.134  INFO [service4,2485ec27856c56f4,1b1845262ffba49d,true] 68061 --- [nio-8084-exec-1]
i.s.c.sleuth.docs.service4.Application   : Hello from service4
service2.log:2016-02-26 11:15:48.156  INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Got response from service4 [Hello from service4]
service1.log:2016-02-26 11:15:48.182  INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1]
i.s.c.sleuth.docs.service1.Application   : Got response from service2 [Hello from service2, response from service3 [Hello
from service3] and from service4 [Hello from service4]]
```

If you're using a log aggregating tool like Kibana (https://www.elastic.co/products/kibana), Splunk (http://www.splunk.com/) etc. you can order the events that took place. An example of Kibana would look like this:



If you want to use Logstash (https://www.elastic.co/guide/en/logstash/current/index.html) here is the Grok pattern for Logstash:

```
filter {
       # pattern matching logback pattern
       grok {
              match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%
{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%
{GREEDYDATA:rest}" }
       }
}
```

NOTE If you want to use Grok together with the logs from Cloud Foundry you have to use this pattern:

```
filter {
       # pattern matching logback pattern
       grok {
              match => { "message" => "(?m)OUT\s+%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%
{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%
{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
       }
}
```

JSON Logback with Logstash

Often you do not want to store your logs in a text file but in a JSON file that Logstash can immediately pick. To do that you have to do the following (for readability we're passing the dependencies in the `groupId:artifactId:version` notation.

Dependencies setup

- Ensure that Logback is on the classpath (`ch.qos.logback:logback-core`)
- Add Logstash Logback encode - example for version `4.6` : `net.logstash.logback:logstash-logback-encoder:4.6`

Logback setup

Below you can find an example of a Logback configuration (file named logback-spring.xml (https://github.com/spring-cloud-samples/sleuth-documentation-apps/blob/master/service1/src/main/resources/logback-spring.xml)) that:

- logs information from the application in a JSON format to a `build/${spring.application.name}.json` file
- has commented out two additional appenders - console and standard log file
- has the same logging pattern as the one presented in the previous section

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>

    <springProperty scope="context" name="springAppName" source="spring.application.name"/>
    <!-- Example for logging into the build folder of your project -->
    <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>

    <!-- You can override this to have a custom pattern -->
    <property name="CONSOLE_LOG_PATTERN"
              value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta}
%clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint}
%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>

    <!-- Appender to log to console -->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <!-- Minimum logging level to be presented in the console logs-->
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <!-- Appender to log to file -->
    <appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_FILE}</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <!-- Appender to log to file in a JSON format -->
    <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_FILE}.json</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp>
                    <timeZone>UTC</timeZone>
                </timestamp>
                <pattern>
                    <pattern>
                        {
                        "severity": "%level",
                        "service": "${springAppName:-}",
                        "trace": "%X{X-B3-TraceId:-}",
                        "span": "%X{X-B3-SpanId:-}",
                        "parent": "%X{X-B3-ParentSpanId:-}",
                        "exportable": "%X{X-Span-Export:-}",
                        "pid": "${PID:-}",
                        "thread": "%thread",
                        "class": "%logger{40}",
                        "rest": "%message"
                        }
                    </pattern>
                </pattern>
            </providers>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="console"/>
        <!-- uncomment this to have also JSON logs -->
        <!--<appender-ref ref="logstash"/>-->
        <!--<appender-ref ref="flatfile"/>-->
    </root>
</configuration>
```

NOTE

If you're using a custom `logback-spring.xml` then you have to pass the `spring.application.name` in `bootstrap` instead of `application` property file. Otherwise your custom logback file won't read the property properly.

Propagating Span Context

The span context is the state that must get propagated to any child Spans across process boundaries. Part of the Span Context is the Baggage. The trace and span IDs are a required part of the span context. Baggage is an optional part.

Baggage is a set of key:value pairs stored in the span context. Baggage travels together with the trace and is attached to every span. Spring Cloud Sleuth will understand that a header is baggage related if the HTTP header is prefixed with `baggage-` and for messaging it starts with `baggage_` .

IMPORTANT

There's currently no limitation of the count or size of baggage items. However, keep in mind that too many can decrease system throughput or increase RPC latency. In extreme cases, it could crash the app due to exceeding transport-level message or header capacity.

Example of setting baggage on a span:

JAVA

```
Span initialSpan = this.tracer.createSpan("span");
initialSpan.setBaggageItem("foo", "bar");
initialSpan.setBaggageItem("UPPER_CASE", "someValue");
```

Baggage vs. Span Tags

Baggage travels with the trace (i.e. every child span contains the baggage of its parent). Zipkin has no knowledge of baggage and will not even receive that information.

Tags are attached to a specific span - they are presented for that particular span only. However you can search by tag to find the trace, where there exists a span having the searched tag value.

If you want to be able to lookup a span based on baggage, you should add corresponding entry as a tag in the root span.

JAVA

```
@Autowired Tracer tracer;

Span span = tracer.getCurrentSpan();
String baggageKey = "key";
String baggageValue = "foo";
span.setBaggageItem(baggageKey, baggageValue);
tracer.addTag(baggageKey, baggageValue);
```

Adding to the project

Only Sleuth (log correlation)

If you want to profit only from Spring Cloud Sleuth without the Zipkin integration just add the `spring-cloud-starter-sleuth` module to your project.

Maven

XML

```
<dependencyManagement> (1)
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Camden.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependency> (2)
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-sleuth</artifactId>
    </dependency>
```

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-starter-sleuth`

Gradle

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Camden.RELEASE"
    }
}

dependencies { (2)
    compile "org.springframework.cloud:spring-cloud-starter-sleuth"
}
```

GROOVY

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-starter-sleuth`

Sleuth with Zipkin via HTTP

If you want both Sleuth and Zipkin just add the `spring-cloud-starter-zipkin` dependency.

Maven

```
<dependencyManagement> (1)
          <dependencies>
              <dependency>
                  <groupId>org.springframework.cloud</groupId>
                  <artifactId>spring-cloud-dependencies</artifactId>
                  <version>Camden.RELEASE</version>
                  <type>pom</type>
                  <scope>import</scope>
              </dependency>
          </dependencies>
    </dependencyManagement>

    <dependency> (2)
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zipkin</artifactId>
    </dependency>
```

XML

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-starter-zipkin`

Gradle

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Camden.RELEASE"
    }
}

dependencies { (2)
    compile "org.springframework.cloud:spring-cloud-starter-zipkin"
}
```

GROOVY

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-starter-zipkin`

Sleuth with Zipkin via Spring Cloud Stream

If you want both Sleuth and Zipkin just add the `spring-cloud-sleuth-stream` dependency.

Maven

XML

```
<dependencyManagement> (1)
         <dependencies>
             <dependency>
                 <groupId>org.springframework.cloud</groupId>
                 <artifactId>spring-cloud-dependencies</artifactId>
                 <version>Camden.RELEASE</version>
                 <type>pom</type>
                 <scope>import</scope>
             </dependency>
         </dependencies>
    </dependencyManagement>

    <dependency> (2)
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-stream</artifactId>
    </dependency>
    <dependency> (3)
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-sleuth</artifactId>
    </dependency>
    <!-- EXAMPLE FOR RABBIT BINDING -->
    <dependency> (4)
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
    </dependency>
```

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-sleuth-stream`
3. Add the dependency to `spring-cloud-starter-sleuth` - that way all dependant dependencies will be downloaded
4. Add a binder (e.g. Rabbit binder) to tell Spring Cloud Stream what it should bind to

Gradle

GROOVY

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Camden.RELEASE"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-sleuth-stream" (2)
    compile "org.springframework.cloud:spring-cloud-starter-sleuth" (3)
    // Example for Rabbit binding
    compile "org.springframework.cloud:spring-cloud-stream-binder-rabbit" (4)
}
```

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-sleuth-stream`
3. Add the dependency to `spring-cloud-starter-sleuth` - that way all dependant dependencies will be downloaded
4. Add a binder (e.g. Rabbit binder) to tell Spring Cloud Stream what it should bind to

Spring Cloud Sleuth Stream Zipkin Collector

If you want to start a Spring Cloud Sleuth Stream Zipkin collector just add the `spring-cloud-sleuth-zipkin-stream` dependency

Maven

XML

```
<dependencyManagement> (1)
         <dependencies>
             <dependency>
                 <groupId>org.springframework.cloud</groupId>
                 <artifactId>spring-cloud-dependencies</artifactId>
                 <version>Camden.RELEASE</version>
                 <type>pom</type>
                 <scope>import</scope>
             </dependency>
         </dependencies>
   </dependencyManagement>

   <dependency> (2)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
   </dependency>
   <dependency> (3)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-starter-sleuth</artifactId>
   </dependency>
   <!-- EXAMPLE FOR RABBIT BINDING -->
   <dependency> (4)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
   </dependency>
```

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-sleuth-zipkin-stream`
3. Add the dependency to `spring-cloud-starter-sleuth` - that way all dependant dependencies will be downloaded
4. Add a binder (e.g. Rabbit binder) to tell Spring Cloud Stream what it should bind to

Gradle

GROOVY

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Camden.RELEASE"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-sleuth-zipkin-stream" (2)
    compile "org.springframework.cloud:spring-cloud-starter-sleuth" (3)
    // Example for Rabbit binding
    compile "org.springframework.cloud:spring-cloud-stream-binder-rabbit" (4)
}
```

1. In order not to pick versions by yourself it's much better if you add the dependency management via the Spring BOM
2. Add the dependency to `spring-cloud-sleuth-zipkin-stream`
3. Add the dependency to `spring-cloud-starter-sleuth` - that way all dependant dependencies will be downloaded
4. Add a binder (e.g. Rabbit binder) to tell Spring Cloud Stream what it should bind to

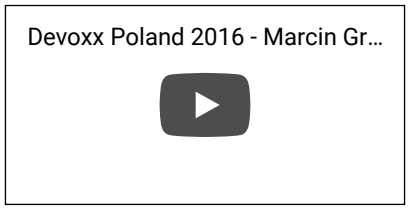
and then just annotate your main class with `@EnableZipkinStreamServer` annotation:

JAVA

```
package example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.sleuth.zipkin.stream.EnableZipkinStreamServer;

@SpringBootApplication
@EnableZipkinStreamServer
public class ZipkinStreamServerApplication {

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

}
```

Additional resources

Marcin Grzejszczak talking about Spring Cloud Sleuth and Zipkin



click here to see the video (https://www.youtube.com/watch?v=eQV71Mw1u1c)

Features

- Adds trace and span ids to the Slf4J MDC, so you can extract all the logs from a given trace or span in a log aggregator. Example logs:

```
2016-02-02 15:30:57.902  INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
2016-02-02 15:31:01.936  INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 --- [nio-8081-exec-4] ...
```

  notice the `[appname,traceId,spanId,exportable]` entries from the MDC:

  - **spanId** - the id of a specific operation that took place
  - **appname** - the name of the application that logged the span
  - **traceId** - the id of the latency graph that contains the span
  - **exportable** - whether the log should be exported to Zipkin or not. When would you like the span not to be exportable? In the case in which you want to wrap some operation in a Span and have it written to the logs only.
- Provides an abstraction over common distributed tracing data models: traces, spans (forming a DAG), annotations, key-value annotations. Loosely based on HTrace, but Zipkin (Dapper) compatible.
- Sleuth records timing information to aid in latency analysis. Using sleuth, you can pinpoint causes of latency in your applications. Sleuth is written to not log too much, and to not cause your production application to crash.
  - propagates structural data about your call-graph in-band, and the rest out-of-band.
  - includes opinionated instrumentation of layers such as HTTP
  - includes sampling policy to manage volume
  - can report to a Zipkin system for query and visualization
- Instruments common ingress and egress points from Spring applications (servlet filter, async endpoints, rest template, scheduled actions, message channels, zuul filters, feign client).
- Sleuth includes default logic to join a trace across http or messaging boundaries. For example, http propagation works via Zipkin-compatible request headers. This propagation logic is defined and customized via `SpanInjector` and `SpanExtractor` implementations.
- Sleuth gives you the possibility to propagate context (also known as baggage) between processes. That means that if you set on a Span a baggage element then it will be sent downstream either via HTTP or messaging to other processes.
- Provides a way to create / continue spans and add tags and logs via annotations.
- Provides simple metrics of accepted / dropped spans.
- If `spring-cloud-sleuth-zipkin` then the app will generate and collect Zipkin-compatible traces. By default it sends them via HTTP to a Zipkin server on localhost (port 9411). Configure the location of the service using `spring.zipkin.baseUrl`.
- If `spring-cloud-sleuth-stream` then the app will generate and collect traces via Spring Cloud Stream (https://github.com/spring-cloud/spring-cloud-stream). Your app automatically becomes a producer of tracer messages that are sent over your broker of choice (e.g. RabbitMQ, Apache Kafka, Redis).

IMPORTANT	If using Zipkin or Stream, configure the percentage of spans exported using `spring.sleuth.sampler.percentage` (default 0.1, i.e. 10%). **Otherwise you might think that Sleuth is not working cause it's omitting some spans.**
NOTE	the SLF4J MDC is always set and logback users will immediately see the trace and span ids in logs per the example above. Other logging systems have to configure their own formatter to get the same result. The default is `logging.pattern.level` set to `%5p [${spring.zipkin.service.name:${spring.application.name:-}},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]` (this is a Spring Boot feature for logback users). **This means that if you're not using SLF4J this pattern WILL NOT be automatically applied**.

Sampling

In distributed tracing the data volumes can be very high so sampling can be important (you usually don't need to export all spans to get a good picture of what is happening). Spring Cloud Sleuth has a `Sampler` strategy that you can implement to take control of the sampling algorithm. Samplers do not stop span (correlation) ids from being generated, but they do prevent the tags and events being attached and exported. By default you get a strategy that continues to trace if a span is already active, but new ones are always marked as non-exportable. If all your apps run with this sampler you will see traces in logs, but not in any remote store. For testing the default is often enough, and it probably is all you need if you are only using the logs (e.g. with an ELK aggregator). If you are exporting span data to Zipkin or Spring Cloud Stream, there is also an `AlwaysSampler` that exports everything and a `PercentageBasedSampler` that samples a fixed fraction of spans.

NOTE

the `PercentageBasedSampler` is the default if you are using `spring-cloud-sleuth-zipkin` or `spring-cloud-sleuth-stream`. You can configure the exports using `spring.sleuth.sampler.percentage`. The passed value needs to be a double from `0.0` to `1.0` so it's not a percentage. For backwards compatibility reasons we're not changing the property name.

A sampler can be installed just by creating a bean definition, e.g:

JAVA

```
@Bean
public Sampler defaultSampler() {
    return new AlwaysSampler();
}
```

TIP

You can set the HTTP header `X-B3-Flags` to `1` or when doing messaging you can set `spanFlags` header to `1`. Then the current span will be forced to be exportable regardless of the sampling decision.

Instrumentation

Spring Cloud Sleuth instruments all your Spring application automatically, so you shouldn't have to do anything to activate it. The instrumentation is added using a variety of technologies according to the stack that is available, e.g. for a servlet web application we use a `Filter`, and for Spring Integration we use `ChannelInterceptors`.

You can customize the keys used in span tags. To limit the volume of span data, by default an HTTP request will be tagged only with a handful of metadata like the status code, host and URL. You can add request headers by configuring `spring.sleuth.keys.http.headers` (a list of header names).

NOTE

Remember that tags are only collected and exported if there is a `Sampler` that allows it (by default there is not, so there is no danger of accidentally collecting too much data without configuring something).

NOTE

Currently the instrumentation in Spring Cloud Sleuth is eager - it means that we're actively trying to pass the tracing context between threads. Also timing events are captured even when sleuth isn't exporting data to a tracing system. This approach may change in the future towards being lazy on this matter.

Span lifecycle

You can do the following operations on the Span by means of **org.springframework.cloud.sleuth.Tracer** interface:

- start - when you start a span its name is assigned and start timestamp is recorded.
- close - the span gets finished (the end time of the span is recorded) and if the span is **exportable** then it will be eligible for collection to Zipkin. The span is also removed from the current thread.
- continue - a new instance of span will be created whereas it will be a copy of the one that it continues.
- detach - the span doesn't get stopped or closed. It only gets removed from the current thread.
- create with explicit parent - you can create a new span and set an explicit parent to it

TIP | Spring creates the instance of `Tracer` for you. In order to use it all you need is to just autowire it.

Creating and closing spans

You can manually create spans by using the **Tracer** interface.

JAVA

```
// Start a span. If there was a span present in this thread it will become
// the `newSpan`'s parent.
Span newSpan = this.tracer.createSpan("calculateTax");
try {
    // ...
    // You can tag a span
    this.tracer.addTag("taxValue", taxValue);
    // ...
    // You can log an event on a span
    newSpan.logEvent("taxCalculated");
} finally {
    // Once done remember to close the span. This will allow collecting
    // the span to send it to Zipkin
    this.tracer.close(newSpan);
}
```

In this example we could see how to create a new instance of span. Assuming that there already was a span present in this thread then it would become the parent of that span.

IMPORTANT | Always clean after you create a span! Don't forget to close a span if you want to send it to Zipkin.

IMPORTANT | If your span contains a name greater than 50 chars, then that name will be truncated to 50 chars. Your names have to be explicit and concrete. Big names lead to latency issues and sometimes even thrown exceptions.

Continuing spans

Sometimes you don't want to create a new span but you want to continue one. Example of such a situation might be (of course it all depends on the use-case):

- **AOP** - If there was already a span created before an aspect was reached then you might not want to create a new span.
- **Hystrix** - executing a Hystrix command is most likely a logical part of the current processing. It's in fact only a technical implementation detail that you wouldn't necessarily want to reflect in tracing as a separate being.

The continued instance of span is equal to the one that it continues:

JAVA

```
Span continuedSpan = this.tracer.continueSpan(spanToContinue);
assertThat(continuedSpan).isEqualTo(spanToContinue);
```

To continue a span you can use the **Tracer** interface.

JAVA

```
// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X
Span continuedSpan = this.tracer.continueSpan(initialSpan);
try {
    // ...
    // You can tag a span
    this.tracer.addTag("taxValue", taxValue);
    // ...
    // You can log an event on a span
    continuedSpan.logEvent("taxCalculated");
} finally {
    // Once done remember to detach the span. That way you'll
    // safely remove it from the current thread without closing it
    this.tracer.detach(continuedSpan);
}
```

IMPORTANT

Always clean after you create a span! Don't forget to detach a span if some work was done started in one thread (e.g. thread X) and it's waiting for other threads (e.g. Y, Z) to finish. Then the spans in the threads Y, Z should be detached at the end of their work. When the results are collected the span in thread X should be closed.

Creating spans with an explicit parent

There is a possibility that you want to start a new span and provide an explicit parent of that span. Let's assume that the parent of a span is in one thread and you want to start a new span in another thread. The `startSpan` method of the `Tracer` interface is the method you are looking for.

JAVA

```
// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X. `initialSpan` will be the parent
// of the `newSpan`
Span newSpan = this.tracer.createSpan("calculateCommission", initialSpan);
try {
    // ...
    // You can tag a span
    this.tracer.addTag("commissionValue", commissionValue);
    // ...
    // You can log an event on a span
    newSpan.logEvent("commissionCalculated");
} finally {
    // Once done remember to close the span. This will allow collecting
    // the span to send it to Zipkin. The tags and events set on the
    // newSpan will not be present on the parent
    this.tracer.close(newSpan);
}
```

IMPORTANT

After having created such a span remember to close it. Otherwise you will see a lot of warnings in your logs related to the fact that you have a span present in the current thread other than the one you're trying to close. What's worse your spans won't get closed properly thus will not get collected to Zipkin.

Naming spans

Picking a span name is not a trivial task. Span name should depict an operation name. The name should be low cardinality (e.g. not include identifiers).

Since there is a lot of instrumentation going on some of the span names will be artificial like:

- `controller-method-name` when received by a Controller with a method name `conrollerMethodName`
- `async` for asynchronous operations done via wrapped `Callable` and `Runnable`.
- `@Scheduled` annotated methods will return the simple name of the class.

Fortunately, for the asynchronous processing you can provide explicit naming.

@SpanName annotation

You can name the span explicitly via the `@SpanName` annotation.

JAVA

```
@SpanName("calculateTax")
class TaxCountingRunnable implements Runnable {

    @Override public void run() {
        // perform logic
    }
}
```

In this case, when processed in the following manner:

JAVA

```
Runnable runnable = new TraceRunnable(tracer, spanNamer, new TaxCountingRunnable());
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

The span will be named `calculateTax`.

toString() method

It's pretty rare to create separate classes for `Runnable` or `Callable`. Typically one creates an anonymous instance of those classes. You can't annotate such classes thus to override that, if there is no `@SpanName` annotation present, we're checking if the class has a custom implementation of the `toString()` method.

So executing such code:

JAVA

```
Runnable runnable = new TraceRunnable(tracer, spanNamer, new Runnable() {
    @Override public void run() {
        // perform logic
    }

    @Override public String toString() {
        return "calculateTax";
    }
});
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

will lead in creating a span named `calculateTax`.

Managing spans with annotations

Rationale

The main arguments for this features are

- api-agnostic means to collaborate with a span
  - use of annotations allows users to add to a span with no library dependency on a span api. This allows Sleuth to change its core api less impact to user code.
- reduced surface area for basic span operations.
  - without this feature one has to use the span api, which has lifecycle commands that could be used incorrectly. By only exposing scope, tag and log functionality, users can collaborate without accidentally breaking span lifecycle.
- collaboration with runtime generated code
  - with libraries such as Spring Data / Feign the implementations of interfaces are generated at runtime thus span wrapping of objects was tedious. Now you can provide annotations over interfaces and arguments of those interfaces

Creating new spans

If you really don't want to take care of creating local spans manually you can profit from the `@NewSpan` annotation. Also we give you the `@SpanTag` annotation to add tags in an automated fashion.

Let's look at some examples of usage.

JAVA

```
@NewSpan
void testMethod();
```

Annotating the method without any parameter will lead to a creation of a new span whose name will be equal to annotated method name.

JAVA

```
@NewSpan("customNameOnTestMethod4")
void testMethod4();
```

If you provide the value in the annotation (either directly or via the `name` parameter) then the created span will have the name as the provided value.

JAVA

```
// method declaration
@NewSpan(name = "customNameOnTestMethod5")
void testMethod5(@SpanTag("testTag") String param);

// and method execution
this.testBean.testMethod5("test");
```

You can combine both the name and a tag. Let's focus on the latter. In this case whatever the value of the annotated method's parameter runtime value will be - that will be the value of the tag. In our sample the tag key will be `testTag` and the tag value will be `test`.

JAVA

```
@NewSpan(name = "customNameOnTestMethod3")
@Override
public void testMethod3() {
}
```

You can place the `@NewSpan` annotation on both the class and an interface. If you override the interface's method and provide a different value of the `@NewSpan` annotation then the most concrete one wins (in this case `customNameOnTestMethod3` will be set).

Continuing spans

If you want to just add tags and annotations to an existing span it's enough to use the `@ContinueSpan` annotation as presented below. Note that in contrast with the `@NewSpan` annotation you can also add logs via the `log` parameter:

```
// method declaration
@ContinueSpan(log = "testMethod11")
void testMethod11(@SpanTag("testTag11") String param);

// method execution
this.testBean.testMethod11("test");
```

JAVA

That way the span will get continued and:

- logs with name `testMethod11.before` and `testMethod11.after` will be created
- if an exception will be thrown a log `testMethod11.afterFailure` will also be created
- tag with key `testTag11` and value `test` will be created

More advanced tag setting

There are 3 different ways to add tags to a span. All of them are controlled by the `SpanTag` annotation. Precedence is:

- try with the bean of `TagValueResolver` type and provided name
- if one hasn't provided the bean name, try to evaluate an expression. We're searching for a `TagValueExpressionResolver` bean. The default implementation uses SPEL expression resolution.
- if one hasn't provided any expression to evaluate just return a `toString()` value of the parameter

Custom extractor

The value of the tag for following method will be computed by an implementation of `TagValueResolver` interface. Its class name has to be passed as the value of the `resolver` attribute.

Having such an annotated method:

```
@NewSpan
public void getAnnotationForTagValueResolver(@SpanTag(key = "test", resolver = TagValueResolver.class) String test) {
}
```

JAVA

and such a `TagValueResolver` bean implementation

```
@Bean(name = "myCustomTagValueResolver")
public TagValueResolver tagValueResolver() {
    return parameter -> "Value from myCustomTagValueResolver";
}
```

JAVA

Will lead to setting of a tag value equal to `Value from myCustomTagValueResolver`.

Resolving expressions for value

Having such an annotated method:

```
@NewSpan
public void getAnnotationForTagValueExpression(@SpanTag(key = "test", expression = "length() + ' characters'") String
test) {
}
```

JAVA

and no custom implementation of a `TagValueExpressionResolver` will lead to evaluation of the SPEL expression and a tag with value `4 characters` will be set on the span. If you want to use some other expression resolution mechanism you can create your own implementation of the bean.

Using toString method

Having such an annotated method:

```
@NewSpan
public void getAnnotationForArgumentToString(@SpanTag("test") Long param) {
}
```

JAVA

if executed with a value of `15` will lead to setting of a tag with a String value of `"15"`.

Customizations

Thanks to the `SpanInjector` and `SpanExtractor` you can customize the way spans are created and propagated.

There are currently two built-in ways to pass tracing information between processes:

- via Spring Integration
- via HTTP

Span ids are extracted from Zipkin-compatible (B3) headers (either `Message` or HTTP headers), to start or join an existing trace. Trace information is injected into any outbound requests so the next hop can extract them.

The key change in comparison to the previous versions of Sleuth is that Sleuth is implementing the Open Tracing's `TextMap` notion. In Sleuth it's called `SpanTextMap` . Basically the idea is that any means of communication (e.g. message, http request, etc.) can be abstracted via a `SpanTextMap` . This abstraction defines how one can insert data into the carrier and how to retrieve it from there. Thanks to this if you want to instrument a new HTTP library that uses a `FooRequest` as a mean of sending HTTP requests then you have to create an implementation of a `SpanTextMap` that delegates calls to `FooRequest` in terms of retrieval and insertion of HTTP headers.

Spring Integration

For Spring Integration there are 2 interfaces responsible for creation of a Span from a `Message` . These are:

- `MessagingSpanTextMapExtractor`
- `MessagingSpanTextMapInjector`

You can override them by providing your own implementation.

HTTP

For HTTP there are 2 interfaces responsible for creation of a Span from a `Message` . These are:

- `HttpSpanExtractor`
- `HttpSpanInjector`

You can override them by providing your own implementation.

Example

Let's assume that instead of the standard Zipkin compatible tracing HTTP header names you have

- for trace id - `correlationId`
- for span id - `mySpanId`

This is a an example of a `SpanExtractor`

JAVA

```
static class CustomHttpSpanExtractor implements HttpSpanExtractor {

    @Override public Span joinTrace(SpanTextMap carrier) {
        Map<String, String> map = TextMapUtil.asMap(carrier);
        long traceId = Span.hexToId(map.get("correlationid"));
        long spanId = Span.hexToId(map.get("myspanid"));
        // extract all necessary headers
        Span.SpanBuilder builder = Span.builder().traceId(traceId).spanId(spanId);
        // build rest of the Span
        return builder.build();
    }
}

static class CustomHttpSpanInjector implements HttpSpanInjector {

    @Override
    public void inject(Span span, SpanTextMap carrier) {
        carrier.put("correlationId", span.traceIdString());
        carrier.put("mySpanId", Span.idToHex(span.getSpanId()));
    }
}
```

And you could register it like this:

JAVA

```
@Bean
HttpSpanInjector customHttpSpanInjector() {
    return new CustomHttpSpanInjector();
}

@Bean
HttpSpanExtractor customHttpSpanExtractor() {
    return new CustomHttpSpanExtractor();
}
```

Spring Cloud Sleuth does not add trace/span related headers to the Http Response for security reasons. If you need the headers then a custom `SpanInjector` that injects the headers into the Http Response and a Servlet filter which makes use of this can be added the following way:

JAVA

```
static class CustomHttpServletResponseSpanInjector extends ZipkinHttpSpanInjector {

    @Override
    public void inject(Span span, SpanTextMap carrier) {
        super.inject(span, carrier);
        carrier.put(Span.TRACE_ID_NAME, span.traceIdString());
        carrier.put(Span.SPAN_ID_NAME, Span.idToHex(span.getSpanId()));
    }
}

static class HttpResponseInjectingTraceFilter extends GenericFilterBean {

    private final Tracer tracer;
    private final HttpSpanInjector spanInjector;

    public HttpResponseInjectingTraceFilter(Tracer tracer, HttpSpanInjector spanInjector) {
        this.tracer = tracer;
        this.spanInjector = spanInjector;
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse servletResponse, FilterChain filterChain) throws
IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        Span currentSpan = this.tracer.getCurrentSpan();
        this.spanInjector.inject(currentSpan, new HttpServletResponseTextMap(response));
        filterChain.doFilter(request, response);
    }

     class HttpServletResponseTextMap implements SpanTextMap {

         private final HttpServletResponse delegate;

         HttpServletResponseTextMap(HttpServletResponse delegate) {
             this.delegate = delegate;
         }

         @Override
         public Iterator<Map.Entry<String, String>> iterator() {
             Map<String, String> map = new HashMap<>();
             for (String header : this.delegate.getHeaderNames()) {
                map.put(header, this.delegate.getHeader(header));
             }
             return map.entrySet().iterator();
         }

         @Override
         public void put(String key, String value) {
            this.delegate.addHeader(key, value);
         }
     }
}
```

And you could register them like this:

```
@Bean HttpSpanInjector customHttpServletResponseSpanInjector() {
    return new CustomHttpServletResponseSpanInjector();
}

@Bean
HttpResponseInjectingTraceFilter responseInjectingTraceFilter(Tracer tracer) {
    return new HttpResponseInjectingTraceFilter(tracer, customHttpServletResponseSpanInjector());
}
```
JAVA

Custom SA tag in Zipkin

Sometimes you want to create a manual Span that will wrap a call to an external service which is not instrumented. What you can do is to create a span with the `peer.service` tag that will contain a value of the service that you want to call. Below you can see an example of a call to Redis that is wrapped in such a span.

```
org.springframework.cloud.sleuth.Span newSpan = tracer.createSpan("redis");
try {
    newSpan.tag("redis.op", "get");
    newSpan.tag("lc", "redis");
    newSpan.logEvent(org.springframework.cloud.sleuth.Span.CLIENT_SEND);
    // call redis service e.g
    // return (SomeObj) redisTemplate.opsForHash().get("MYHASH", someObjKey);
} finally {
    newSpan.tag("peer.service", "redisService");
    newSpan.tag("peer.ipv4", "1.2.3.4");
    newSpan.tag("peer.port", "1234");
    newSpan.logEvent(org.springframework.cloud.sleuth.Span.CLIENT_RECV);
    tracer.close(newSpan);
}
```
JAVA

IMPORTANT Remember not to add both `peer.service` tag and the `SA` tag! You have to add only `peer.service` .

Custom service name

By default Sleuth assumes that when you send a span to Zipkin, you want the span's service name to be equal to `spring.application.name` value. That's not always the case though. There are situations in which you want to explicitly provide a different service name for all spans coming from your application. To achieve that it's enough to just pass the following property to your application to override that value (example for `foo` service name):

```
spring.zipkin.service.name: foo
```
YAML

Customization of reported spans

Before reporting spans to e.g. Zipkin you can be interested in modifying that span in some way. You can achieve that by using the `SpanAdjuster` interface.

Example of usage:

In Sleuth we're generating spans with a fixed name. Some users want to modify the name depending on values of tags. Implementation of the `SpanAdjuster` interface can be used to alter that name. Example:

```
@Bean
SpanAdjuster customSpanAdjuster() {
    return span -> span.toBuilder().name(scrub(span.getName())).build();
}
```
YAML

This will lead in changing the name of the reported span just before it gets sent to Zipkin.

IMPORTANT Your `SpanReporter` should inject the `SpanAdjuster` and allow span manipulation before the actual reporting is done.

Host locator

In order to define the host that is corresponding to a particular span we need to resolve the host name and port. The default approach is to take it from server properties. If those for some reason are not set then we're trying to retrieve the host name from the network interfaces.

If you have the discovery client enabled and prefer to retrieve the host address from the registered instance in a service registry then you have to set the property (it's applicable for both HTTP and Stream based span reporting).

```yaml
spring.zipkin.locator.discovery.enabled: true
```

Span Data as Messages

You can accumulate and send span data over Spring Cloud Stream (https://cloud.spring.io/spring-cloud-stream) by including the `spring-cloud-sleuth-stream` jar as a dependency, and adding a Channel Binder implementation (e.g. `spring-cloud-starter-stream-rabbit` for RabbitMQ or `spring-cloud-starter-stream-kafka` for Kafka). This will automatically turn your app into a producer of messages with payload type `Spans`.

Zipkin Consumer

There is a special convenience annotation for setting up a message consumer for the Span data and pushing it into a Zipkin `SpanStore`. This application

JAVA
```
@SpringBootApplication
@EnableZipkinStreamServer
public class Consumer {
    public static void main(String[] args) {
        SpringApplication.run(Consumer.class, args);
    }
}
```

will listen for the Span data on whatever transport you provide via a Spring Cloud Stream `Binder` (e.g. include `spring-cloud-starter-stream-rabbit` for RabbitMQ, and similar starters exist for Redis and Kafka). If you add the following UI dependency

XML
```
<groupId>io.zipkin.java</groupId>
<artifactId>zipkin-autoconfigure-ui</artifactId>
```

Then you'll have your app a Zipkin server (https://github.com/openzipkin/zipkin), which hosts the UI and api on port 9411.

The default `SpanStore` is in-memory (good for demos and getting started quickly). For a more robust solution you can add MySQL and `spring-boot-starter-jdbc` to your classpath and enable the JDBC `SpanStore` via configuration, e.g.:

YAML
```
spring:
  rabbitmq:
    host: ${RABBIT_HOST:localhost}
  datasource:
    schema: classpath:/mysql.sql
    url: jdbc:mysql://${MYSQL_HOST:localhost}/test
    username: root
    password: root
# Switch this on to create the schema on startup:
    initialize: true
    continueOnError: true
  sleuth:
    enabled: false
zipkin:
  storage:
    type: mysql
```

NOTE

The `@EnableZipkinStreamServer` is also annotated with `@EnableZipkinServer` so the process will also expose the standard Zipkin server endpoints for collecting spans over HTTP, and for querying in the Zipkin Web UI.

Custom Consumer

A custom consumer can also easily be implemented using `spring-cloud-sleuth-stream` and binding to the `SleuthSink`. Example:

JAVA
```
@EnableBinding(SleuthSink.class)
@SpringBootApplication(exclude = SleuthStreamAutoConfiguration.class)
@MessageEndpoint
public class Consumer {

    @ServiceActivator(inputChannel = SleuthSink.INPUT)
    public void sink(Spans input) throws Exception {
        // ... process spans
    }
}
```

NOTE

the sample consumer application above explicitly excludes `SleuthStreamAutoConfiguration` so it doesn't send messages to itself, but this is optional (you might actually want to trace requests into the consumer app).

In order to customize the polling mechanism you can create a bean of `PollerMetadata` type with name equal to `StreamSpanReporter.POLLER`. Here you can find an example of such a configuration.

JAVA

```
@Configuration
public static class CustomPollerConfiguration {

    @Bean(name = StreamSpanReporter.POLLER)
    PollerMetadata customPoller() {
        PollerMetadata poller = new PollerMetadata();
        poller.setMaxMessagesPerPoll(500);
        poller.setTrigger(new PeriodicTrigger(5000L));
        return poller;
    }
}
```

Metrics

Currently Spring Cloud Sleuth registers very simple metrics related to spans. It's using the Spring Boot's metrics support (https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-recording-metrics) to calculate the number of accepted and dropped spans. Each time a span gets sent to Zipkin the number of accepted spans will increase. If there's an error then the number of dropped spans will get increased.

Integrations

Runnable and Callable

If you're wrapping your logic in `Runnable` or `Callable` it's enough to wrap those classes in their Sleuth representative.

Example for `Runnable`:

```java
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // do some work
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceRunnable` creation with explicit "calculateTax" Span name
Runnable traceRunnable = new TraceRunnable(tracer, spanNamer, runnable, "calculateTax");
// Wrapping `Runnable` with `Tracer`. The Span name will be taken either from the
// `@SpanName` annotation or from `toString` method
Runnable traceRunnableFromTracer = tracer.wrap(runnable);
```

Example for `Callable`:

```java
Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        return someLogic();
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceCallable` creation with explicit "calculateTax" Span name
Callable<String> traceCallable = new TraceCallable<>(tracer, spanNamer, callable, "calculateTax");
// Wrapping `Callable` with `Tracer`. The Span name will be taken either from the
// `@SpanName` annotation or from `toString` method
Callable<String> traceCallableFromTracer = tracer.wrap(callable);
```

That way you will ensure that a new Span is created and closed for each execution.

Hystrix

Custom Concurrency Strategy

We're registering a custom `HystrixConcurrencyStrategy` (https://github.com/Netflix/Hystrix/wiki/Plugins#concurrencystrategy) that wraps all `Callable` instances into their Sleuth representative - the `TraceCallable`. The strategy either starts or continues a span depending on the fact whether tracing was already going on before the Hystrix command was called. To disable the custom Hystrix Concurrency Strategy set the `spring.sleuth.hystrix.strategy.enabled` to `false`.

Manual Command setting

Assuming that you have the following `HystrixCommand`:

```java
HystrixCommand<String> hystrixCommand = new HystrixCommand<String>(setter) {
    @Override
    protected String run() throws Exception {
        return someLogic();
    }
};
```

In order to pass the tracing information you have to wrap the same logic in the Sleuth version of the `HystrixCommand` which is the `TraceCommand`:

```
TraceCommand<String> traceCommand = new TraceCommand<String>(tracer, traceKeys, setter) {
    @Override
    public String doRun() throws Exception {
        return someLogic();
    }
};
```
JAVA

RxJava

We're registering a custom `RxJavaSchedulersHook` (https://github.com/ReactiveX/RxJava/wiki/Plugins#rxjavaschedulershook) that wraps all `Action0` instances into their Sleuth representative - the `TraceAction`. The hook either starts or continues a span depending on the fact whether tracing was already going on before the Action was scheduled. To disable the custom RxJavaSchedulersHook set the `spring.sleuth.rxjava.schedulers.hook.enabled` to `false`.

You can define a list of regular expressions for thread names, for which you don't want a Span to be created. Just provide a comma separated list of regular expressions in the `spring.sleuth.rxjava.schedulers.ignoredthreads` property.

HTTP integration

Features from this section can be disabled by providing the `spring.sleuth.web.enabled` property with value equal to `false`.

HTTP Filter

Via the `TraceFilter` all sampled incoming requests result in creation of a Span. That Span's name is `http:` + the path to which the request was sent. E.g. if the request was sent to `/foo/bar` then the name will be `http:/foo/bar`. You can configure which URIs you would like to skip via the `spring.sleuth.web.skipPattern` property. If you have `ManagementServerProperties` on classpath then its value of `contextPath` gets appended to the provided skip pattern.

HandlerInterceptor

Since we want the span names to be precise we're using a `TraceHandlerInterceptor` that either wraps an existing `HandlerInterceptor` or is added directly to the list of existing `HandlerInterceptors`. The `TraceHandlerInterceptor` adds a special request attribute to the given `HttpServletRequest`. If the the `TraceFilter` doesn't see this attribute set it will create a "fallback" span which is an additional span created on the server side so that the trace is presented properly in the UI. Seeing that most likely signifies that there is a missing instrumentation. In that case please file an issue in Spring Cloud Sleuth.

Async Servlet support

If your controller returns a `Callable` or a `WebAsyncTask` Spring Cloud Sleuth will continue the existing span instead of creating a new one.

HTTP client integration

Synchronous Rest Template

We're injecting a `RestTemplate` interceptor that ensures that all the tracing information is passed to the requests. Each time a call is made a new Span is created. It gets closed upon receiving the response. In order to block the synchronous `RestTemplate` features just set `spring.sleuth.web.client.enabled` to `false`.

IMPORTANT	You have to register `RestTemplate` as a bean so that the interceptors will get injected. If you create a `RestTemplate` instance with a `new` keyword then the instrumentation WILL NOT work.

Asynchronous Rest Template

IMPORTANT	A traced version of an `AsyncRestTemplate` bean is registered for you out of the box. If you have your own bean you have to wrap it in a `TraceAsyncRestTemplate` representation. The best solution is to only customize the `ClientHttpRequestFactory` and / or `AsyncClientHttpRequestFactory`. **If you have your own `AsyncRestTemplate` and you don't wrap it your calls WILL NOT GET TRACED**.

Custom instrumentation is set to create and close Spans upon sending and receiving requests. You can customize the `ClientHttpRequestFactory` and the `AsyncClientHttpRequestFactory` by registering your beans. Remember to use tracing compatible implementations (e.g. don't forget to wrap `ThreadPoolTaskScheduler` in a `TraceAsyncListenableTaskExecutor`). Example of custom request factories:

JAVA

```
@EnableAutoConfiguration
@Configuration
public static class TestConfiguration {

    @Bean
    ClientHttpRequestFactory mySyncClientFactory() {
        return new MySyncClientHttpRequestFactory();
    }

    @Bean
    AsyncClientHttpRequestFactory myAsyncClientFactory() {
        return new MyAsyncClientHttpRequestFactory();
    }
}
```

To block the `AsyncRestTemplate` features set `spring.sleuth.web.async.client.enabled` to `false`. To disable creation of the default `TraceAsyncClientHttpRequestFactoryWrapper` set `spring.sleuth.web.async.client.factory.enabled` to `false`. If you don't want to create `AsyncRestClient` at all set `spring.sleuth.web.async.client.template.enabled` to `false`.

Multiple Asynchronous Rest Templates

Sometimes you need to use multiple implementations of Asynchronous Rest Template. In the following snippet you can see an example of how to set up such a custom `AsyncRestTemplate`.

JAVA

```
@Configuration
@EnableAutoConfiguration
static class Config {
    @Autowired Tracer tracer;
    @Autowired HttpTraceKeysInjector httpTraceKeysInjector;
    @Autowired HttpSpanInjector spanInjector;

    @Bean(name = "customAsyncRestTemplate")
    public AsyncRestTemplate traceAsyncRestTemplate(@Qualifier("customHttpRequestFactoryWrapper")
            TraceAsyncClientHttpRequestFactoryWrapper wrapper) {
        return new TraceAsyncRestTemplate(wrapper, this.tracer);
    }

    @Bean(name = "customHttpRequestFactoryWrapper")
    public TraceAsyncClientHttpRequestFactoryWrapper traceAsyncClientHttpRequestFactory() {
        return new TraceAsyncClientHttpRequestFactoryWrapper(this.tracer,
                this.spanInjector,
                asyncClientFactory(),
                clientHttpRequestFactory(),
                this.httpTraceKeysInjector);
    }

    private ClientHttpRequestFactory clientHttpRequestFactory() {
        ClientHttpRequestFactory clientHttpRequestFactory = new CustomClientHttpRequestFactory();
        //CUSTOMIZE HERE
        return clientHttpRequestFactory;
    }

    private AsyncClientHttpRequestFactory asyncClientFactory() {
        AsyncClientHttpRequestFactory factory = new CustomAsyncClientHttpRequestFactory();
        //CUSTOMIZE HERE
        return factory;
    }
}
```

Traverson

If you're using the Traverson (https://docs.spring.io/spring-hateoas/docs/current/reference/html/#client.traverson) library it's enough for you to inject a `RestTemplate` as a bean into your Traverson object. Since `RestTemplate` is already intercepted, you will get full support of tracing in your client. Below you can find a pseudo code of how to do that:

JAVA

```
@Autowired RestTemplate restTemplate;

Traverson traverson = new Traverson(URI.create("http://some/address"),
    MediaType.APPLICATION_JSON, MediaType.APPLICATION_JSON_UTF8).setRestOperations(restTemplate);
// use Traverson
```

Feign

By default Spring Cloud Sleuth provides integration with feign via the `TraceFeignClientAutoConfiguration`. You can disable it entirely by setting `spring.sleuth.feign.enabled` to false. If you do so then no Feign related instrumentation will take place.

Part of Feign instrumentation is done via a `FeignBeanPostProcessor`. You can disable it by providing the `spring.sleuth.feign.processor.enabled` equal to `false`. If you set it like this then Spring Cloud Sleuth will not instrument any of your custom Feign components. All the default instrumentation however will be still there.

Asynchronous communication

@Async annotated methods

In Spring Cloud Sleuth we're instrumenting async related components so that the tracing information is passed between threads. You can disable this behaviour by setting the value of `spring.sleuth.async.enabled` to `false`.

If you annotate your method with `@Async` then we'll automatically create a new Span with the following characteristics:

- the Span name will be the annotated method name
- the Span will be tagged with that method's class name and the method name too

@Scheduled annotated methods

In Spring Cloud Sleuth we're instrumenting scheduled method execution so that the tracing information is passed between threads. You can disable this behaviour by setting the value of `spring.sleuth.scheduled.enabled` to `false`.

If you annotate your method with `@Scheduled` then we'll automatically create a new Span with the following characteristics:

- the Span name will be the annotated method name
- the Span will be tagged with that method's class name and the method name too

If you want to skip Span creation for some `@Scheduled` annotated classes you can set the `spring.sleuth.scheduled.skipPattern` with a regular expression that will match the fully qualified name of the `@Scheduled` annotated class.

> **TIP** If you are using `spring-cloud-sleuth-stream` and `spring-cloud-netflix-hystrix-stream` together, Span will be created for each Hystrix metrics and sent to Zipkin. This may be annoying. You can prevent this by setting `spring.sleuth.scheduled.skipPattern=org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask`

Executor, ExecutorService and ScheduledExecutorService

We're providing `LazyTraceExecutor`, `TraceableExecutorService` and `TraceableScheduledExecutorService`. Those implementations are creating Spans each time a new task is submitted, invoked or scheduled.

Here you can see an example of how to pass tracing information with `TraceableExecutorService` when working with `CompletableFuture`:

JAVA

```
CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {
    // perform some logic
    return 1_000_000L;
}, new TraceableExecutorService(executorService,
        // 'calculateTax' explicitly names the span - this param is optional
        tracer, traceKeys, spanNamer, "calculateTax"));
```

Customization of Executors

Sometimes you need to set up a custom instance of the `AsyncExecutor`. In the following snippet you can see an example of how to set up such a custom `Executor`.

JAVA

```
@Configuration
@EnableAutoConfiguration
@EnableAsync
static class CustomExecutorConfig extends AsyncConfigurerSupport {

    @Autowired BeanFactory beanFactory;

    @Override public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        // CUSTOMIZE HERE
        executor.setCorePoolSize(7);
        executor.setMaxPoolSize(42);
        executor.setQueueCapacity(11);
        executor.setThreadNamePrefix("MyExecutor-");
        // DON'T FORGET TO INITIALIZE
        executor.initialize();
        return new LazyTraceExecutor(this.beanFactory, executor);
    }
}
```

Messaging

Spring Cloud Sleuth integrates with Spring Integration (https://projects.spring.io/spring-integration/). It creates spans for publish and subscribe events. To disable Spring Integration instrumentation, set `spring.sleuth.integration.enabled` to false.

You can provide the `spring.sleuth.integration.patterns` pattern to explicitly provide the names of channels that you want to include for tracing. By default all channels are included.

IMPORTANT	When using the `Executor` to build a Spring Integration `IntegrationFlow` remember to use the **untraced** version of the `Executor`. Decorating Spring Integration Executor Channel with `TraceableExecutorService` will cause the spans to be improperly closed.

Zuul

We're registering Zuul filters to propagate the tracing information (the request header is enriched with tracing data). To disable Zuul support set the `spring.sleuth.zuul.enabled` property to `false`.

Running examples

You can find the running examples deployed in the Pivotal Web Services (https://run.pivotal.io/). Check them out in the following links:

- Zipkin for apps presented in the samples to the top (https://docssleuth-zipkin-server.cfapps.io/)
- Zipkin for Brewery on PWS (https://docsbrewing-zipkin-web.cfapps.io/), its Github Code (https://github.com/spring-cloud-samples/brewery)

Spring Cloud Consul

Dalston.SR3

This project provides Consul 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 Consul based components. The patterns provided include Service Discovery, Control Bus and Configuration. Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon), Circuit Breaker (Hystrix) are provided by integration with Spring Cloud Netflix.

Install Consul

Please see the installation documentation (https://www.consul.io/intro/getting-started/install.html) for instructions on how to install Consul.

Consul Agent

A Consul Agent client must be available to all Spring Cloud Consul applications. By default, the Agent client is expected to be at `localhost:8500`. See the Agent documentation (https://consul.io/docs/agent/basics.html) for specifics on how to start an Agent client and how to connect to a cluster of Consul Agent Servers. For development, after you have installed consul, you may start a Consul Agent using the following command:

```
./src/main/bash/local_run_consul.sh
```

This will start an agent in server mode on port 8500, with the ui available at http://localhost:8500

Service Discovery with Consul

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. Consul provides Service Discovery services via an HTTP API (https://www.consul.io/docs/agent/http.html) and DNS (https://www.consul.io/docs/agent/dns.html). Spring Cloud Consul leverages the HTTP API for service registration and discovery. This does not prevent non-Spring Cloud applications from leveraging the DNS interface. Consul Agents servers are run in a cluster (https://www.consul.io/docs/internals/architecture.html) that communicates via a gossip protocol (https://www.consul.io/docs/internals/gossip.html) and uses the Raft consensus protocol (https://www.consul.io/docs/internals/consensus.html).

How to activate

To activate Consul Service Discovery use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-discovery`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

Registering with Consul

When a client registers with Consul, it provides meta-data about itself such as host and port, id, name and tags. An HTTP Check (https://www.consul.io/docs/agent/checks.html) is created by default that Consul hits the `/health` endpoint every 10 seconds. If the health check fails, the service instance is marked as critical.

Example Consul client:

JAVA

```
@SpringBootApplication
@EnableDiscoveryClient
@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). If the Consul client is located somewhere other than `localhost:8500`, the configuration is required to locate the client. Example:

application.yml

```
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

CAUTION If you use Spring Cloud Consul Config, the above values will need to be placed in `bootstrap.yml` instead of `application.yml`.

The default service name, instance id and port, taken from the `Environment`, are `${spring.application.name}`, the Spring Context ID and `${server.port}` respectively.

`@EnableDiscoveryClient` make the app into both a Consul "service" (i.e. it registers itself) and a "client" (i.e. it can query Consul to locate other services).

HTTP Health Check

The health check for a Consul instance defaults to "/health", which is the default locations of a useful endpoint 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.context-path=/admin`). The interval that Consul uses to check the health endpoint may also be configured. "10s" and "1m" represent 10 seconds and 1 minute respectively. Example:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        healthCheckPath: ${management.context-path}/health
        healthCheckInterval: 15s
```

Metadata and Consul tags

Consul does not yet support metadata on services. Spring Cloud's `ServiceInstance` has a `Map<String, String> metadata` field. Spring Cloud Consul uses Consul tags to approximate metadata until Consul officially supports metadata. Tags with the form `key=value` will be split and used as a `Map` key and value respectively. Tags without the equal `=` sign, will be used as both the key and value.

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        tags: foo=bar, baz
```

The above configuration will result in a map with `foo→bar` and `baz→baz`.

Making the Consul Instance ID Unique

By default a consul instance is registered with an ID that is equal to its Spring Application Context ID. By default, the Spring Application Context ID is `${spring.application.name}:comma,separated,profiles:${server.port}`. For most cases, this will allow multiple instances of one service to run on one machine. If further uniqueness is required, Using Spring Cloud you can override this by providing a unique identifier in `spring.cloud.consul.discovery.instanceId`. For example:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        instanceId:
${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

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 `vcap.application.instance_id` will be populated automatically in a Spring Boot application, so the random value will not be needed.

Using the DiscoveryClient

Spring Cloud has support for Feign (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-feign) (a REST client builder) and also Spring `RestTemplate` (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-ribbon) using the logical service names instead of physical URLs.

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.

```
@Autowired
private DiscoveryClient discoveryClient;

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

Distributed Configuration with Consul

Consul provides a Key/Value Store (https://consul.io/docs/agent/http/kv.html) for storing configuration and other metadata. Spring Cloud Consul Config is an alternative to the Config Server and Client (https://github.com/spring-cloud/spring-cloud-config). Configuration is loaded into the Spring Environment during the special "bootstrap" phase. Configuration is stored in the `/config` folder by default. Multiple `PropertySource` instances are created based on the application's name and the active profiles that mimicks the Spring Cloud Config order of resolving properties. For example, an application with the name "testApp" and with the "dev" profile will have the following property sources created:

```
config/testApp,dev/
config/testApp/
config/application,dev/
config/application/
```

The most specific property source is at the top, with the least specific at the bottom. Properties is the `config/application` folder are applicable to all applications using consul for configuration. Properties in the `config/testApp` folder are only available to the instances of the service named "testApp".

Configuration is currently read on startup of the application. Sending a HTTP POST to `/refresh` will cause the configuration to be reloaded. Watching the key value store (which Consul supports) is not currently possible, but will be a future addition to this project.

How to activate

To get started with Consul Configuration use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-config`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

This will enable auto-configuration that will setup Spring Cloud Consul Config.

Customizing

Consul Config may be customized using the following properties:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled` setting this value to "false" disables Consul Config
- `prefix` sets the base folder for configuration values
- `defaultContext` sets the folder name used by all applications
- `profileSeparator` sets the value of the separator used to separate the profile name in property sources with profiles

Config Watch

The Consul Config Watch takes advantage of the ability of consul to watch a key prefix (https://www.consul.io/docs/agent/watches.html#keyprefix). The Config Watch makes a blocking Consul HTTP API call to determine if any relevant configuration data has changed for the current application. If there is new configuration data a Refresh Event is published. This is equivalent to calling the `/refresh` actuator endpoint.

To change the frequency of when the Config Watch is called change `spring.cloud.consul.config.watch.delay`. The default value is 1000, which is in milliseconds.

To disable the Config Watch set `spring.cloud.consul.config.watch.enabled=false`.

YAML or Properties with Config

It may be more convenient to store a blob of properties in YAML or Properties format as opposed to individual key/value pairs. Set the `spring.cloud.consul.config.format` property to `YAML` or `PROPERTIES`. For example to use YAML:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: YAML
```

YAML must be set in the appropriate `data` key in consul. Using the defaults above the keys would look like:

```
config/testApp,dev/data
config/testApp/data
config/application,dev/data
config/application/data
```

You could store a YAML document in any of the keys listed above.

You can change the data key using `spring.cloud.consul.config.data-key`.

git2consul with Config

git2consul is a Consul community project that loads files from a git repository to individual keys into Consul. By default the names of the keys are names of the files. YAML and Properties files are supported with file extensions of `.yml` and `.properties` respectively. Set the `spring.cloud.consul.config.format` property to `FILES`. For example:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: FILES
```

Given the following keys in `/config`, the `development` profile and an application name of `foo`:

```
.gitignore
application.yml
bar.properties
foo-development.properties
foo-production.yml
foo.properties
master.ref
```

the following property sources would be created:

```
config/foo-development.properties
config/foo.properties
config/application.yml
```

The value of each key needs to be a properly formatted YAML or Properties file.

Fail Fast

It may be convenient in certain circumstances (like local development or certain test scenarios) to not fail if consul isn't available for configuration. Setting `spring.cloud.consul.config.failFast=false` in `bootstrap.yml` will cause the configuration module to log a warning rather than throw an exception. This will allow the application to continue startup normally.

Consul Retry

If you expect that the consul agent may occasionally be unavailable when your app starts, you can ask it to keep trying after a failure. 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.cloud.consul.retry.*` configuration properties. This works with both Spring Cloud Consul Config and Discovery registration.

TIP

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

Spring Cloud Bus with Consul

How to activate

To get started with the Consul Bus use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-bus`. See the Spring Cloud Project page (https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

See the Spring Cloud Bus (https://cloud.spring.io/spring-cloud-bus/) documentation for the available actuator endpoints and howto send custom messages.

Circuit Breaker with Hystrix

Applications can use the Hystrix Circuit Breaker provided by the Spring Cloud Netflix project by including this starter in the projects pom.xml: `spring-cloud-starter-hystrix` . Hystrix doesn't depend on the Netflix Discovery Client. The `@EnableHystrix` annotation should be placed on a configuration class (usually the main class). Then methods can be annotated with `@HystrixCommand` to be protected by a circuit breaker. See the documentation (https://projects.spring.io/spring-cloud/spring-cloud.html#_circuit_breaker_hystrix_clients) for more details.

Hystrix metrics aggregation with Turbine and Consul

Turbine (provided by the Spring Cloud Netflix project), aggregates multiple instances Hystrix metrics streams, so the dashboard can display an aggregate view. Turbine uses the `DiscoveryClient` interface to lookup relevant instances. To use Turbine with Spring Cloud Consul, configure the Turbine application in a manner similar to the following examples:

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

Notice that the Turbine dependency is not a starter. The turbine starter includes support for Netflix Eureka.

application.yml

```
spring.application.name: turbine
applications: consulhystrixclient
turbine:
  aggregator:
    clusterConfig: ${applications}
  appConfig: ${applications}
```

The `clusterConfig` and `appConfig` sections must match, so it's useful to put the comma-separated list of service ID's into a separate configuration property.

Turbine.java

```
@EnableTurbine
@EnableDiscoveryClient
@SpringBootApplication
public class Turbine {
    public static void main(String[] args) {
        SpringApplication.run(DemoturbinecommonsApplication.class, args);
    }
}
```

Spring Cloud Zookeeper

This project provides Zookeeper 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 Zookeeper based components. The patterns provided include Service Discovery and Configuration. Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon), Circuit Breaker (Hystrix) are provided by integration with Spring Cloud Netflix.

Install Zookeeper

Please see the installation documentation (https://zookeeper.apache.org/doc/current/zookeeperStarted.html) for instructions on how to install Zookeeper.

Service Discovery with Zookeeper

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. Curator (https://curator.apache.org)(A java library for Zookeeper) provides Service Discovery services via Service Discovery Extension (https://curator.apache.org/curator-x-discovery/). Spring Cloud Zookeeper leverages this extension for service registration and discovery.

How to activate

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` will enable auto-configuration that will setup Spring Cloud Zookeeper Discovery.

NOTE	You still need to include `org.springframework.boot:spring-boot-starter-web` for web functionality.

Registering with Zookeeper

When a client registers with Zookeeper, it provides meta-data about itself such as host and port, id and name.

Example Zookeeper client:

JAVA

```
@SpringBootApplication
@EnableDiscoveryClient
@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). If Zookeeper is located somewhere other than `localhost:2181`, the configuration is required to locate the server. Example:

application.yml

```
spring:
  cloud:
    zookeeper:
      connect-string: localhost:2181
```

CAUTION	If you use Spring Cloud Zookeeper Config, the above values will need to be placed in `bootstrap.yml` instead of `application.yml`.

The default service name, instance id and port, taken from the `Environment`, are `${spring.application.name}`, the Spring Context ID and `${server.port}` respectively.

`@EnableDiscoveryClient` makes the app into both a Zookeeper "service" (i.e. it registers itself) and a "client" (i.e. it can query Zookeeper to locate other services).

Using the DiscoveryClient

Spring Cloud has support for Feign (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-feign) (a REST client builder) and also Spring `RestTemplate` (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-ribbon) using the logical service names instead of physical URLs.

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.

JAVA

```
@Autowired
private DiscoveryClient discoveryClient;

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

Using Spring Cloud Zookeeper with Spring Cloud Netflix Components

Spring Cloud Netflix supplies useful tools that work regardless of which `DiscoveryClient` implementation is used. Feign, Turbine, Ribbon and Zuul all work with Spring Cloud Zookeeper.

Ribbon with Zookeeper

Spring Cloud Zookeeper provides an implementation of Ribbon's `ServerList`. When the `spring-cloud-starter-zookeeper-discovery` is used, Ribbon is auto-configured to use the `ZookeeperServerList` by default.

Spring Cloud Zookeeper and Service Registry

Spring Cloud Zookeeper implements the `ServiceRegistry` interface allowing developers to register arbitrary service in a programmatic way.

The `ServiceInstanceRegistration` class offers a `builder()` method to create a `Registration` object that can be used by the `ServiceRegistry`.

JAVA

```
@Autowired
private ZookeeperServiceRegistry serviceRegistry;

public void registerThings() {
    ZookeeperRegistration registration = ServiceInstanceRegistration.builder()
            .defaultUriSpec()
            .address("anyUrl")
            .port(10)
            .name("/a/b/c/d/anotherservice")
            .build();
    this.serviceRegistry.register(registration);
}
```

Instance Status

Netflix Eureka supports having instances registered with the server that are `OUT_OF_SERVICE` and not returned as active service instances. This is very useful for behaviors such as blue/green deployments. The Curator Service Discovery recipe does not support this behavior. Taking advantage of the flexible payload has let Spring Cloud Zookeeper implement `OUT_OF_SERVICE` by updating some specific metadata and then filtering on that metadata in the Ribbon `ZookeeperServerList`. The `ZookeeperServerList` filters out all non-null instance statuses that do not equal `UP`. If the instance status field is empty, it is considered `UP` for backwards compatibility. To change the status of an instance POST `OUT_OF_SERVICE` to the `ServiceRegistry` instance status actuator endpoint.

```
----
$ echo -n OUT_OF_SERVICE | http POST http://localhost:8081/service-registry/instance-status
----


NOTE: The above example uses the `http` command from https://httpie.org
```

Zookeeper Dependencies

Using the Zookeeper Dependencies

Spring Cloud Zookeeper gives you a possibility to provide dependencies of your application as properties. As dependencies you can understand other applications that are registered in Zookeeper and which you would like to call via Feign (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-feign) (a REST client builder) and also Spring `RestTemplate` (https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-ribbon).

You can also benefit from the Zookeeper Dependency Watchers functionality that lets you control and monitor what is the state of your dependencies and decide what to do with that.

How to activate Zookeeper Dependencies

- Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` will enable auto-configuration that will setup Spring Cloud Zookeeper Dependencies.
- If you have to have the `spring.cloud.zookeeper.dependencies` section properly set up - check the subsequent section for more details then the feature is active
- You can have the dependencies turned off even if you've provided the dependencies in your properties. Just set the property `spring.cloud.zookeeper.dependency.enabled` to false (defaults to `true`).

Setting up Zookeeper Dependencies

Let's take a closer look at an example of dependencies representation:

application.yml

```
spring.application.name: yourServiceName
spring.cloud.zookeeper:
  dependencies:
    newsletter:
      path: /path/where/newsletter/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.newsletter.$version+json
      version: v1
      headers:
        header1:
            - value1
        header2:
            - value2
      required: false
      stubs: org.springframework:foo:stubs
    mailing:
      path: /path/where/mailing/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.mailing.$version+json
      version: v1
      required: true
```

Let's now go through each part of the dependency one by one. The root property name is `spring.cloud.zookeeper.dependencies`.

Aliases

Below the root property you have to represent each dependency has by an alias due to the constraints of Ribbon (the application id has to be placed in the URL thus you can't pass any complex path like /foo/bar/name). The alias will be the name that you will use instead of serviceId for `DiscoveryClient`, `Feign` or `RestTemplate`.

In the aforementioned examples the aliases are `newsletter` and `mailing`. Example of Feign usage with `newsletter` would be:

```
@FeignClient("newsletter")
public interface NewsletterService {
        @RequestMapping(method = RequestMethod.GET, value = "/newsletter")
        String getNewsletters();
}
```

Path

Represented by `path` yaml property.

Path is the path under which the dependency is registered under Zookeeper. Like presented before Ribbon operates on URLs thus this path is not compliant with its requirement. That is why Spring Cloud Zookeeper maps the alias to the proper path.

Load balancer type

Represented by `loadBalancerType` yaml property.

If you know what kind of load balancing strategy has to be applied when calling this particular dependency then you can provide it in the yaml file and it will be automatically applied. You can choose one of the following load balancing strategies

- STICKY - once chosen the instance will always be called
- RANDOM - picks an instance randomly
- ROUND_ROBIN - iterates over instances over and over again

Content-Type template and version

Represented by `contentTypeTemplate` and `version` yaml property.

If you version your api via the `Content-Type` header then you don't want to add this header to each of your requests. Also if you want to call a new version of the API you don't want to roam around your code to bump up the API version. That's why you can provide a `contentTypeTemplate` with a special `$version` placeholder. That placeholder will be filled by the value of the `version` yaml property. Let's take a look at an example.

Having the following `contentTypeTemplate` :

```
application/vnd.newsletter.$version+json
```

and the following `version` :

```
v1
```

Will result in setting up of a `Content-Type` header for each request:

```
application/vnd.newsletter.v1+json
```

Default headers

Represented by `headers` map in yaml

Sometimes each call to a dependency requires setting up of some default headers. In order not to do that in code you can set them up in the yaml file. Having the following `headers` section:

```
headers:
    Accept:
        - text/html
        - application/xhtml+xml
    Cache-Control:
        - no-cache
```

Results in adding the `Accept` and `Cache-Control` headers with appropriate list of values in your HTTP request.

Obligatory dependencies

Represented by `required` property in yaml

If one of your dependencies is required to be up and running when your application is booting then it's enough to set up the `required: true` property in the yaml file.

If your application can't localize the required dependency during boot time it will throw an exception and the Spring Context will fail to set up. In other words your application won't be able to start if the required dependency is not registered in Zookeeper.

You can read more about Spring Cloud Zookeeper Presence Checker in the following sections.

Stubs

You can provide a colon separated path to the JAR containing stubs of the dependency. Example

```
stubs: org.springframework:foo:stubs
```

means that for a particular dependencies can be found under:

- groupId: `org.springframework`
- artifactId: `foo`
- classifier: `stubs` - this is the default value

This is actually equal to

```
stubs: org.springframework:foo
```

since `stubs` is the default classifier.

Configuring Spring Cloud Zookeeper Dependencies

There is a bunch of properties that you can set to enable / disable parts of Zookeeper Dependencies functionalities.

- `spring.cloud.zookeeper.dependencies` - if you don't set this property you won't benefit from Zookeeper Dependencies
- `spring.cloud.zookeeper.dependency.ribbon.enabled` (enabled by default) - Ribbon requires explicit global configuration or a particular one for a dependency. By turning on this property runtime load balancing strategy resolution is possible and you can profit from the `loadBalancerType` section of the Zookeeper Dependencies. The configuration that needs this property has an implementation of `LoadBalancerClient` that delegates to the `ILoadBalancer` presented in the next bullet
- `spring.cloud.zookeeper.dependency.ribbon.loadbalancer` (enabled by default) - thanks to this property the custom `ILoadBalancer` knows that the part of the URI passed to Ribbon might actually be the alias that has to be resolved to a proper path in Zookeeper. Without this property you won't be able to register applications under nested paths.
- `spring.cloud.zookeeper.dependency.headers.enabled` (enabled by default) - this property registers such a `RibbonClient` that automatically will append appropriate headers and content types with version as presented in the Dependency configuration. Without this setting of those two parameters will not be operational.
- `spring.cloud.zookeeper.dependency.resttemplate.enabled` (enabled by default) - when enabled will modify the request headers of `@LoadBalanced` annotated `RestTemplate` so that it passes headers and content type with version set in Dependency configuration. Wihtout this setting of those two parameters will not be operational.

Spring Cloud Zookeeper Dependency Watcher

The Dependency Watcher mechanism allows you to register listeners to your dependencies. The functionality is in fact an implementation of the `Observator` pattern. When a dependency changes its state (UP or DOWN) then some custom logic can be applied.

How to activate

Spring Cloud Zookeeper Dependencies functionality needs to be enabled to profit from Dependency Watcher mechanism.

Registering a listener

In order to register a listener you have to implement an interface `org.springframework.cloud.zookeeper.discovery.watcher.DependencyWatcherListener` and register it as a bean. The interface gives you one method:

```
void stateChanged(String dependencyName, DependencyState newState);
```

If you want to register a listener for a particular dependency then the `dependencyName` would be the discriminator for your concrete implementation. `newState` will provide you with information whether your dependency has changed to `CONNECTED` or `DISCONNECTED`.

Presence Checker

Bound with Dependency Watcher is the functionality called Presence Checker. It allows you to provide custom behaviour upon booting of your application to react accordingly to the state of your dependencies.

The default implementation of the abstract `org.springframework.cloud.zookeeper.discovery.watcher.presence.DependencyPresenceOnStartupVerifier` class is the `org.springframework.cloud.zookeeper.discovery.watcher.presence.DefaultDependencyPresenceOnStartupVerifier` which works in the following way.

- If the dependency is marked us `required` and it's not in Zookeeper then upon booting your application will throw an exception and shutdown
- If dependency is not `required` the `org.springframework.cloud.zookeeper.discovery.watcher.presence.LogMissingDependencyChecker` will log that application is missing at `WARN` level

The functionality can be overriden since the `DefaultDependencyPresenceOnStartupVerifier` is registered only when there is no bean of `DependencyPresenceOnStartupVerifier`.

Distributed Configuration with Zookeeper

Zookeeper provides a hierarchical namespace (https://zookeeper.apache.org/doc/current/zookeeperOver.html#sc_dataModelNameSpace) that allows clients to store arbitrary data, such as configuration data. Spring Cloud Zookeeper Config is an alternative to the Config Server and Client (https://github.com/spring-cloud/spring-cloud-config). Configuration is loaded into the Spring Environment during the special "bootstrap" phase. Configuration is stored in the `/config` namespace by default. Multiple `PropertySource` instances are created based on the application's name and the active profiles that mimicks the Spring Cloud Config order of resolving properties. For example, an application with the name "testApp" and with the "dev" profile will have the following property sources created:

```
config/testApp,dev
config/testApp
config/application,dev
config/application
```

The most specific property source is at the top, with the least specific at the bottom. Properties is the `config/application` namespace are applicable to all applications using zookeeper for configuration. Properties in the `config/testApp` namespace are only available to the instances of the service named "testApp".

Configuration is currently read on startup of the application. Sending a HTTP POST to `/refresh` will cause the configuration to be reloaded. Watching the configuration namespace (which Zookeeper supports) is not currently implemented, but will be a future addition to this project.

How to activate

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-config` will enable auto-configuration that will setup Spring Cloud Zookeeper Config.

Customizing

Zookeeper Config may be customized using the following properties:

bootstrap.yml

```
spring:
  cloud:
    zookeeper:
      config:
        enabled: true
        root: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled` setting this value to "false" disables Zookeeper Config
- `root` sets the base namespace for configuration values
- `defaultContext` sets the name used by all applications
- `profileSeparator` sets the value of the separator used to separate the profile name in property sources with profiles

ACLs

You can add authentication information for Zookeeper ACLs by calling the addAuthInfo method of a CuratorFramework bean. One way to accomplish this is by providing your own CuratorFramework bean:

JAVA

```
@BoostrapConfiguration
public class CustomCuratorFrameworkConfig {

  @Bean
  public CuratorFramework curatorFramework() {
    CuratorFramework curator = new CuratorFramework();
    curator.addAuthInfo("digest", "user:password".getBytes());
    return curator;
  }

}
```

Consult the ZookeeperAutoConfiguration class (https://github.com/spring-cloud/spring-cloud-zookeeper/blob/master/spring-cloud-zookeeper-core/src/main/java/org/springframework/cloud/zookeeper/ZookeeperAutoConfiguration.java) to see how the CuratorFramework bean is configured by default.

Alternatively, you can add your credentials from a class that depends on the existing CuratorFramework bean:

JAVA

```
@BoostrapConfiguration
public class DefaultCuratorFrameworkConfig {

  public ZookeeperConfig(CuratorFramework curator) {
    curator.addAuthInfo("digest", "user:password".getBytes());
  }

}
```

This must occur during the boostrapping phase. You can register configuration classes to run during this phase by annotating them with `@BootstrapConfiguration` and including them in a comma-separated list set as the value of the property `org.springframework.cloud.bootstrap.BootstrapConfiguration` in the file `resources/META-INF/spring.factories`:

resources/META-INF/spring.factories

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
my.project.CustomCuratorFrameworkConfig,\
my.project.DefaultCuratorFrameworkConfig
```

Unresolved directive in spring-cloud.adoc - include::/Users/ryanjbaxter/git-repos/spring-cloud/scripts/docs/../cli/docs/src/main/asciidoc/spring-cloud-cli.adoc[]

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 github (https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/asciidoc).

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'
  }

}
```

JAVA

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'
  }

}
```

JAVA

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

```
security:
  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
```

YAML

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 `security.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 (https://github.com/spring-cloud-samples/sso)).

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

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

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

}
```

and

application.yml

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

More Detail

Single Sign On

NOTE

All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the Spring Boot user guide (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/).

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 is a user facing OAuth2 client (i.e. has declared `@EnableOAuth2Sso` or `@EnableOAuth2Client`) then it has an `OAuth2ClientContext` in request scope from Spring Boot. You can create your own `OAuth2RestTemplate` from this context and an autowired `OAuth2ProtectedResourceDetails`, and then the context will always forward the access token downstream, also refreshing the access token automatically if it expires. (These are features of Spring Security and Spring Boot.)

NOTE

Spring Boot (1.4.1) does not create an `OAuth2ProtectedResourceDetails` automatically if you are using `client_credentials` tokens. In that case you need to create your own `ClientCredentialsResourceDetails` and configure it with `@ConfigurationProperties("security.oauth2.client")`.

Client Token Relay in Zuul Proxy

If your app also has a Spring Cloud Zuul (https://cloud.spring.io/spring-cloud.html#netflix-zuul-reverse-proxy) 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

JAVA

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

}
```

and it will (in addition to logging the user in and grabbing a token) pass the authentication token downstream to the `/proxy/*` services. If those services are implemented with `@EnableResourceServer` 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 filter (https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/OAuth2TokenRelayFilter.java) 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 `@EnableResourceServer` you might want to relay the incoming token downstream to other services. If you use a `RestTemplate` to contact the downstream services then this is just a matter of how to create the template with the right context.

If your service uses `UserInfoTokenServices` to authenticate incoming tokens (i.e. it is using the `security.oauth2.user-info-uri` configuration), then you can simply create an `OAuth2RestTemplate` using an autowired `OAuth2ClientContext` (it will be populated by the authentication process before it hits the backend code). Equivalently (with Spring Boot 1.4), you could inject a `UserInfoRestTemplateFactory` and grab its `OAuth2RestTemplate` in your configuration. For example:

MyConfiguration.java

JAVA

```
@Bean
public OAuth2RestTemplate restTemplate(UserInfoRestTemplateFactory factory) {
    return factory.getUserInfoRestTemplate();
}
```

This rest template will then have the same `OAuth2ClientContext` (request-scoped) that is used by the authentication filter, so you can use it to send requests with the same access token.

If your app is not using `UserInfoTokenServices` but is still a client (i.e. it declares `@EnableOAuth2Client` or `@EnableOAuth2Sso`), then with Spring Security Cloud any `OAuth2RestOperations` that the user creates from an `@Autowired` `@OAuth2Context` will also forward tokens. This feature is implemented by default as an MVC handler interceptor, so it only works in Spring MVC. If you are not using MVC you could use a custom filter or AOP interceptor wrapping an `AccessTokenContextRelay` to provide the same feature.

Here's a basic example showing the use of an autowired rest template created elsewhere ("foo.com" is a Resource Server accepting the same tokens as the surrounding app):

MyController.java

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() + ")";
}
```

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 `OAuth2Context` instead of autowiring the default one.

Feign clients will also pick up an interceptor that uses the `OAuth2ClientContext` if it is available, so they should also do a token relay anywhere where a `RestTemplate` would.

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

```YAML
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 ProxyAuthenticationProperties (https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/ProxyAuthenticationProperties) for full details.

Spring Cloud for Cloud Foundry

Spring Cloud for Cloudfoundry makes it easy to run Spring Cloud (https://github.com/spring-cloud) apps in Cloud Foundry (https://github.com/cloudfoundry) (the Platform as a Service). Cloud Foundry has the notion of a "service", which is middlware that you "bind" to an app, essentially providing it with an environment variable containing credentials (e.g. the location and username to use for the service).

The `spring-cloud-cloudfoundry-web` project provides basic support for some enhanced features of webapps in Cloud Foundry: binding automatically to single-sign-on services and optionally enabling sticky routing for discovery.

The `spring-cloud-cloudfoundry-discovery` project provides an implementation of Spring Cloud Commons `DiscoveryClient` so you can `@EnableDiscoveryClient` and provide your credentials as `spring.cloud.cloudfoundry.discovery.[email,password]` and then you can use the `DiscoveryClient` directly or via a `LoadBalancerClient` (also `*.url` if you are not connecting to Pivotal Web Services (https://run.pivotal.io)).

The first time you use it the discovery client might be slow owing to the fact that it has to get an access token from Cloud Foundry.

Discovery

Here's a Spring Cloud app with Cloud Foundry discovery:

app.groovy

JAVA

```
@Grab('org.springframework.cloud:spring-cloud-cloudfoundry')
@RestController
@EnableDiscoveryClient
class Application {

  @Autowired
  DiscoveryClient client

  @RequestMapping('/')
  String home() {
    'Hello from ' + client.getLocalServiceInstance()
  }

}
```

If you run it without any service bindings:

```
$ spring jar app.jar app.groovy
$ cf push -p app.jar
```

It will show its app name in the home page.

The `DiscoveryClient` can lists all the apps in a space, according to the credentials it is authenticated with, where the space defaults to the one the client is running in (if any). If neither org nor space are configured, they default per the user's profile in Cloud Foundry.

Single Sign On

NOTE

All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the Spring Boot user guide (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/).

This project provides automatic binding from CloudFoundry service credentials to the Spring Boot features. If you have a CloudFoundry service called "sso", for instance, with credentials containing "client_id", "client_secret" and "auth_domain", it will bind automatically to the Spring OAuth2 client that you enable with `@EnableOAuth2Sso` (from Spring Boot). The name of the service can be parameterized using `spring.oauth2.sso.serviceId`.

Spring Cloud Contract

Documentation Authors: Adam Dudczak, Mathias Düsterhöft, Marcin Grzejszczak, Dennis Kieselhorst, Jakub Kubryński, Karol Lassak, Olga Maciaszek-Sharma, Mariusz Smykuła, Dave Syer

Dalston.SR3

1. Spring Cloud Contract

What you always need is confidence in pushing new features into a new application or service in a distributed system. This project provides support for Consumer Driven Contracts and service schemas in Spring applications, covering a range of options for writing tests, publishing them as assets, asserting that a contract is kept by producers and consumers, for HTTP and message-based interactions.

2. Spring Cloud Contract Verifier

2.1. Introduction

> The Accurest project was initially started by Marcin Grzejszczak and Jakub Kubrynski (codearte.io (http://codearte.io))

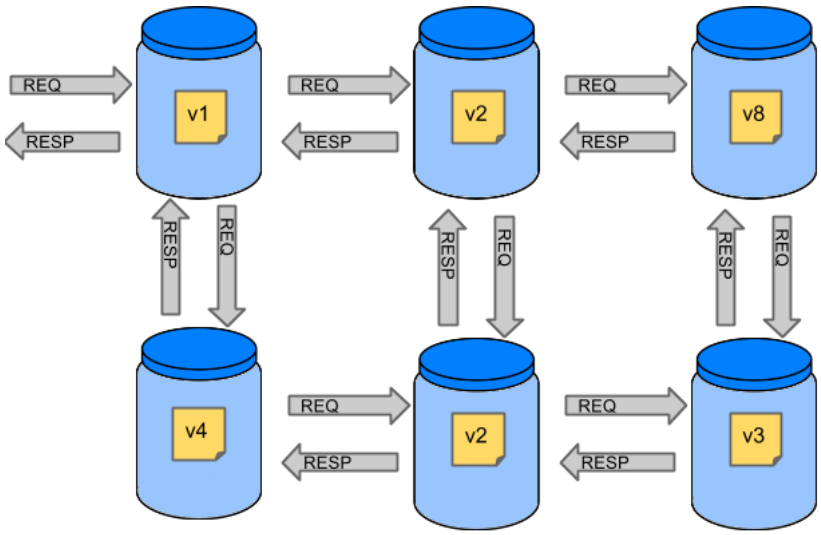
Just to make long story short - Spring Cloud Contract Verifier is a tool that enables Consumer Driven Contract (CDC) development of JVM-based applications. It is shipped with *Contract Definition Language* (DSL). Contract definitions are used to produce following resources:

- JSON stub definitions to be used by WireMock when doing integration testing on the client code (*client tests*). Test code must still be written by hand, test data is produced by Spring Cloud Contract Verifier.
- Messaging routes if you're using one. We're integrating with Spring Integration, Spring Cloud Stream, Spring AMQP and Apache Camel. You can however set your own integrations if you want to
- Acceptance tests (in JUnit or Spock) used to verify if server-side implementation of the API is compliant with the contract (*server tests*). Full test is generated by Spring Cloud Contract Verifier.

Spring Cloud Contract Verifier moves TDD to the level of software architecture.

2.1.1. Why?

Let us assume that we have a system comprising of multiple microservices:



Testing issues

If we wanted to test the application in top left corner if it can communicate with other services then we could do one of two things:

- deploy all microservices and perform end to end tests
- mock other microservices in unit / integration tests

Both have their advantages but also a lot of disadvantages. Let's focus on the latter.

Deploy all microservices and perform end to end tests

Advantages:

- simulates production
- tests real communication between services

Disadvantages:

- to test one microservice we would have to deploy 6 microservices, a couple of databases etc.

- the environment where the tests would be conducted would be locked for a single suite of tests (i.e. nobody else would be able to run the tests in the meantime).
- long to run
- very late feedback
- extremely hard to debug

Mock other microservices in unit / integration tests

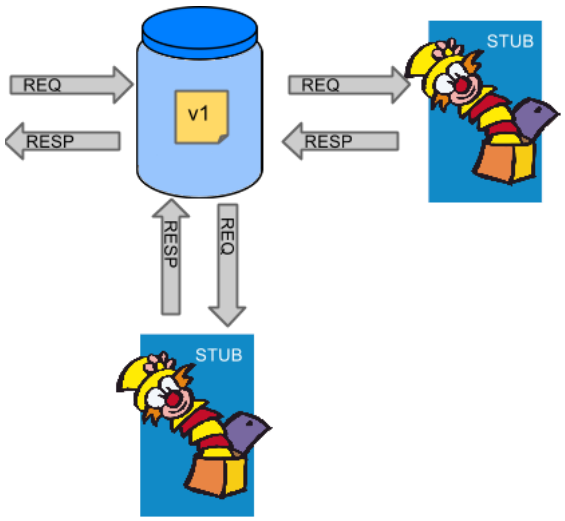
Advantages:

- very fast feedback
- no infrastructure requirements

Disadvantages:

- the implementor of the service creates stubs thus they might have nothing to do with the reality
- you can go to production with passing tests and failing production

To solve the aforementioned issues Spring Cloud Contract Verifier with Stub Runner were created. Their main idea is to give you very fast feedback, without the need to set up the whole world of microservices. If you work on stubs then the only applications you need are those that your application is using directly.



Spring Cloud Contract Verifier gives you the certainty that the stubs that you're using were created by the service that you're calling. Also if you can use them it means that they were tested against the producer's side. In other words - you can trust those stubs.

2.1.2. Purposes

The main purposes of Spring Cloud Contract Verifier with Stub Runner are:

- to ensure that WireMock / Messaging stubs (used when developing the client) are doing exactly what actual server-side implementation will do,
- to promote ATDD method and Microservices architectural style,
- to provide a way to publish changes in contracts that are immediately visible on both sides,
- to generate boilerplate test code used on the server side.

> Spring Cloud Contract Verifier's purpose is NOT to start writing business features in the contracts. Let's assume that we have a business use case of fraud check. If a user can be a fraud for 100 different reasons, we would assume that you would create 2 contracts. One for the positive and one for the negative fraud case. Contract tests are used to test contracts between applications and not to simulate full behaviour.

2.1.3. How

Define the contract

As consumers we need to define what exactly we want to achieve. We need to formulate our expectations. That's why we write the following contract.

Let's assume that we'd like to send the request containing the id of the client and the amount he wants to borrow from us. We'd like to send it to the /fraudcheck url via the PUT method.

GROOVY

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
               "client.id": $(regex('[0-9]{10}')),
               loanAmount: 99999
        ])
        headers { // (5)
            contentType('application/json')
        }
    }
    response { // (6)
        status 200 // (7)
        body([ // (8)
               fraudCheckStatus: "FRAUD",
               "rejection.reason": "Amount too high"
        ])
        headers { // (9)
            contentType('application/json')
        }
    }
}

/*
From the Consumer perspective, when shooting a request in the integration test:

(1) - If the consumer sends a request
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `clientId` that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the response will be sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

(1) - A request will be sent to the producer
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `clientId` that will have a generated value that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the test will assert if the response has been sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` matching `application/json.*`
 */
```

Client Side

Spring Cloud Contract will generate stubs, which you can use during client side testing. You will have a WireMock instance / Messaging route up and running that simulates the service Y. You would like to feed that instance with a proper stub definition.

At some point in time you need to send a request to the Fraud Detection service.

GROOVY

```
ResponseEntity<FraudServiceResponse> response =
        restTemplate.exchange("http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,
                new HttpEntity<>(request, httpHeaders),
                FraudServiceResponse.class);
```

Annotate your test class with `@AutoConfigureStubRunner`. In the annotation provide the group id and artifact id for the Stub Runner to download stubs of your collaborators.

```groovy
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"}, workOffline = true)
@DirtiesContext
public class LoanApplicationServiceTests {
```

After that, during the tests Spring Cloud Contract will automatically find the stubs (simulating the real service) in Maven repository and expose them on configured (or random) port.

Server Side

Being a service Y since you are developing your stub, you need to be sure that it's actually resembling your concrete implementation. You can't have a situation where your stub acts in one way and your application on production behaves in a different way.

That's why from the provided stub acceptance tests will be generated that will ensure that your application behaves in the same way as you define in your stub.

The autogenerated test would look like this:

```java
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
        MockMvcRequestSpecification request = given()
                .header("Content-Type", "application/vnd.fraud.v1+json")
                .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
        ResponseOptions response = given().spec(request)
                .put("/fraudcheck");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
        DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");
        assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");
}
```

2.1.4. Step by step guide to CDC

Let's take an example of Fraud Detection and Loan Issuance process. The business scenario is such that we want to issue loans to people but don't want them to steal the money from us. The current implementation of our system grants loans to everybody.

Let's assume that the `Loan Issuance` is a client to the `Fraud Detection` server. In the current sprint we are required to develop a new feature - if a client wants to borrow too much money then we mark him as fraud.

Technical remark - Fraud Detection will have artifact id `http-server` , Loan Issuance `http-client` and both have group id `com.example` .

Social remark - both client and server development teams need to communicate directly and discuss changes while going through the process. CDC is all about communication.

The server side code is available here (https://github.com/spring-cloud/spring-cloud-contract/tree/1.0.x/samples/standalone/dsl/http-server) and the client side code here (https://github.com/spring-cloud/spring-cloud-contract/tree/1.0.x/samples/standalone/dsl/http-client).

In this case the ownership of the contracts lays on the producer side. It means that physically all the contract are present in the producer's repository

Technical note

If using the **SNAPSHOT** / **Milestone** / **Release Candidate** versions please add the following section to your

Maven

XML

```
<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
</pluginRepositories>
```

Gradle

GROOVY

```
repositories {
    mavenCentral()
    mavenLocal()
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
    maven { url "http://repo.spring.io/release" }
}
```

Consumer side (Loan Issuance)

As a developer of the Loan Issuance service (a consumer of the Fraud Detection server):

start doing TDD by writing a test to your feature

GROOVY
```
@Test
public void shouldBeRejectedDueToAbnormalLoanAmount() {
    // given:
    LoanApplication application = new LoanApplication(new Client("1234567890"),
            99999);
    // when:
    LoanApplicationResult loanApplication = service.loanApplication(application);
    // then:
    assertThat(loanApplication.getLoanApplicationStatus())
            .isEqualTo(LoanApplicationStatus.LOAN_APPLICATION_REJECTED);
    assertThat(loanApplication.getRejectionReason()).isEqualTo("Amount too high");
}
```

We've just written a test of our new feature. If a loan application for a big amount is received we should reject that loan application with some description.

write the missing implementation

At some point in time you need to send a request to the Fraud Detection service. Let's assume that we'd like to send the request containing the id of the client and the amount he wants to borrow from us. We'd like to send it to the `/fraudcheck` url via the `PUT` method.

GROOVY
```
ResponseEntity<FraudServiceResponse> response =
        restTemplate.exchange("http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,
                new HttpEntity<>(request, httpHeaders),
                FraudServiceResponse.class);
```

For simplicity we've hardcoded the port of the Fraud Detection service at `8080` and our application is running on `8090`.

If we'd start the written test it would obviously break since we have no service running on port `8080`.

clone the Fraud Detection service repository locally

We'll start playing around with the server side contract. That's why we need to first clone it.

BASH
```
git clone https://your-git-server.com/server-side.git local-http-server-repo
```

define the contract locally in the repo of Fraud Detection service

As consumers we need to define what exactly we want to achieve. We need to formulate our expectations. That's why we write the following contract.

We're placing the contract under `src/test/resources/contracts/fraud` folder. The `fraud` folder is important cause we'll reference that folder in the producer's test base class name.

GROOVY

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
               "client.id": $(regex('[0-9]{10}')),
               loanAmount: 99999
        ])
        headers { // (5)
            contentType('application/json')
        }
    }
    response { // (6)
        status 200 // (7)
        body([ // (8)
               fraudCheckStatus: "FRAUD",
               "rejection.reason": "Amount too high"
        ])
        headers { // (9)
            contentType('application/json')
        }
    }
}

/*
From the Consumer perspective, when shooting a request in the integration test:

(1) - If the consumer sends a request
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `clientId` that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the response will be sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

(1) - A request will be sent to the producer
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `clientId` that will have a generated value that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to `application/json`
(6) - then the test will assert if the response has been sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
(9) - with header `Content-Type` matching `application/json.*`
 */
```

The Contract is written using a statically typed Groovy DSL. You might be wondering what are those `value(client(…), server(…))` parts. By using this notation Spring Cloud Contract allows you to define parts of a JSON / URL / etc. which are dynamic. In case of an identifier or a timestamp you don't want to hardcode a value. You want to allow some different ranges of values. That's why for the consumer side you can set regular expressions matching those values. You can provide the body either by means of a map notation or String with interpolations. Consult the docs for more information. (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html#_contract_dsl) We highly recommend using the map notation!

It's really important that you understand the map notation to set up contracts. Please read the Groovy docs regarding JSON (http://groovy-lang.org/json.html)

The aforementioned contract is an agreement between two sides that:

- if an HTTP request is sent with
  - a method `PUT` on an endpoint `/fraudcheck`
  - JSON body with `client.id` matching the regular expression `[0-9]{10}` and `loanAmount` equal to `99999`
  - and with a header `Content-Type` equal to `application/vnd.fraud.v1+json`

- then an HTTP response would be sent to the consumer that
  - has status `200`
  - contains JSON body with the `fraudCheckStatus` field containing a value `FRAUD` and the `rejectionReason` field having value `Amount too high`
  - and a `Content-Type` header with a value of `application/vnd.fraud.v1+json`

Once we're ready to check the API in practice in the integration tests we need to just install the stubs locally

add the Spring Cloud Contract Verifier plugin

We can add either Maven or Gradle plugin - in this example we'll show how to add Maven. First we need to add the `Spring Cloud Contract` BOM.

```XML
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud-dependencies.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Next, the `Spring Cloud Contract Verifier` Maven plugin

```XML
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
    </configuration>
</plugin>
```

Since the plugin was added we get the `Spring Cloud Contract Verifier` features which from the provided contracts:

- generate and run tests
- produce and install stubs

We don't want to generate tests since we, as consumers, want only to play with the stubs. That's why we need to skip the tests generation and execution. When we execute:

```BASH
cd local-http-server-repo
./mvnw clean install -DskipTests
```

In the logs we'll see something like this:

```BASH
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-generateStubs) @ http-server --
-
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.4.BUILD-SNAPSHOT:repackage (default) @ http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

This line is extremely important

```
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```
BASH

It's confirming that the stubs of the `http-server` have been installed in the local repository.

run the integration tests

In order to profit from the Spring Cloud Contract Stub Runner functionality of automatic stub downloading you have to do the following in our consumer side project (`Loan Application service`).

Add the `Spring Cloud Contract` BOM

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud-dependencies.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```
XML

Add the dependency to `Spring Cloud Contract Stub Runner`

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
</dependency>
```
XML

Annotate your test class with `@AutoConfigureStubRunner` . In the annotation provide the group id and artifact id for the Stub Runner to download stubs of your collaborators. Also provide the offline work switch since you're playing with the collaborators offline (optional step).

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"}, workOffline = true)
@DirtiesContext
public class LoanApplicationServiceTests {
```
GROOVY

Now if you run your tests you'll see sth like this:

```
2016-07-19 14:22:25.403  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Desired version is +
- will try to resolve the latest version
2016-07-19 14:22:25.438  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolved version is
0.0.1-SNAPSHOT
2016-07-19 14:22:25.439  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolving artifact
com.example:http-server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []
2016-07-19 14:22:25.451  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Resolved artifact
com.example:http-server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
2016-07-19 14:22:25.465  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Unpacking stub from
JAR [URI: file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar]
2016-07-19 14:22:25.475  INFO 41050 --- [           main] o.s.c.c.stubrunner.AetherStubDownloader  : Unpacked file to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contracts100276532569594265]
2016-07-19 14:22:27.737  INFO 41050 --- [           main] o.s.c.c.stubrunner.StubRunnerExecutor    : All stubs are now
running RunningStubs [namesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]
```
BASH

Which means that Stub Runner has found your stubs and started a server for app with group id `com.example` , artifact id `http-server` with version `0.0.1-SNAPSHOT` of the stubs and with `stubs` classifier on port `8080` .

file a PR

What we did until now is an iterative process. We can play around with the contract, install it locally and work on the consumer side until we're happy with the contract.

Once we're satisfied with the results and the test passes publish a PR to the server side. Currently the consumer side work is done.

Producer side (Fraud Detection server)

As a developer of the Fraud Detection server (a server to the Loan Issuance service):

initial implementation

As a reminder here you can see the initial implementation

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

JAVA

take over the PR

```
git checkout -b contract-change-pr master
git pull https://your-git-server.com/server-side-fork.git contract-change-pr
```

BASH

You have to add the dependencies needed by the autogenerated tests

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <scope>test</scope>
</dependency>
```

XML

In the configuration of the Maven plugin we passed the `packageWithBaseClasses` property

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
    </configuration>
</plugin>
```

XML

We've decided to use the "convention based" naming by setting the `packageWithBaseClasses` property. That means that 2 last packages will be combined into a name of the base test class. In our case the contracts were placed under `src/test/resources/contracts/fraud`. Since we don't have 2 packages starting from the `contracts` folder we're picking only one which is `fraud`. We're adding the `Base` suffix and we're capitalizing `fraud`. That gives us the `FraudBase` test class name.

That's because all the generated tests will extend that class. Over there you can set up your Spring Context or whatever is necessary. In our case we're using Rest Assured MVC (http://rest-assured.io/) to start the server side `FraudDetectionController`.

```
package com.example.fraud;

import org.junit.Before;

import io.restassured.module.mockmvc.RestAssuredMockMvc;

public class FraudBase {
    @Before
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(new FraudDetectionController(),
                new FraudStatsController(stubbedStatsProvider()));
    }

    private StatsProvider stubbedStatsProvider() {
        return fraudType -> {
            switch (fraudType) {
            case DRUNKS:
                return 100;
            case ALL:
                return 200;
            }
            return 0;
        };
    }

    public void assertThatRejectionReasonIsNull(Object rejectionReason) {
        assert rejectionReason == null;
    }
}
```

JAVA

Now, if you run the `./mvnw clean install` you would get sth like this:

```
Results :

Tests in error:
  ContractVerifierTest.validate_shouldMarkClientAsFraud:32 » IllegalState Parsed...
```

BASH

That's because you have a new contract from which a test was generated and it failed since you haven't implemented the feature. The autogenerated test would look like this:

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
        MockMvcRequestSpecification request = given()
                .header("Content-Type", "application/vnd.fraud.v1+json")
                .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
        ResponseOptions response = given().spec(request)
                .put("/fraudcheck");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
        DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");
        assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");
}
```

JAVA

As you can see all the `producer()` parts of the Contract that were present in the `value(consumer(…), producer(…))` blocks got injected into the test.

What's important here to note is that on the producer side we also are doing TDD. We have expectations in form of a test. This test is shooting a request to our own application to an URL, headers and body defined in the contract. It also is expecting very precisely defined values in the response. In other words you have is your `red` part of `red`, `green` and `refactor`. Time to convert the `red` into the `green`.

write the missing implementation

Now since we now what is the expected input and expected output let's write the missing implementation.

```java
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
if (amountGreaterThanThreshold(fraudCheck)) {
    return new FraudCheckResult(FraudCheckStatus.FRAUD, AMOUNT_TOO_HIGH);
}
return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

If we execute `./mvnw clean install` again the tests will pass. Since the `Spring Cloud Contract Verifier` plugin adds the tests to the `generated-test-sources` you can actually run those tests from your IDE.

deploy your app

Once you've finished your work it's time to deploy your change. First merge the branch

```bash
git checkout master
git merge --no-ff contract-change-pr
git push origin master
```

Then we assume that your CI would run sth like `./mvnw clean deploy` which would publish both the application and the stub artifcats.

Consumer side (Loan Issuance) final step

As a developer of the Loan Issuance service (a consumer of the Fraud Detection server):

merge branch to master

```bash
git checkout master
git merge --no-ff contract-change-pr
```

work online

Now you can disable the offline work for Spring Cloud Contract Stub Runner and provide where the repository with your stubs is placed. At this moment the stubs of the server side will be automatically downloaded from Nexus / Artifactory. You can switch off the value of the `workOffline` parameter in your annotation. Below you can see an example of achieving the same by changing the properties.

```yaml
stubrunner:
  ids: 'com.example:http-server-dsl:+:stubs:8080'
  repositoryRoot: http://repo.spring.io/libs-snapshot
```

And that's it!

2.1.5. Dependencies

The best way to add the dependencies is to just use the proper `starter` dependency.

For `stub-runner` use `spring-cloud-starter-stub-runner` and when you're using a plugin just add `spring-cloud-starter-contract-verifier`.

2.1.6. Additional links

Below you can find some resources related to Spring Cloud Contract Verifier and Stub Runner. Note that some can be outdated since the Spring Cloud Contract Verifier project is under constant development.

Spring Cloud Contract video

You can check out the video from the Warsaw JUG about Spring Cloud Contract:

Readings

- Slides from Marcin Grzejszczak's talk about Accurest (http://www.slideshare.net/MarcinGrzejszczak/stick-to-the-rules-consumer-driven-contracts-201507-confitura)
- Accurest related articles from Marcin Grzejszczak's blog (http://toomuchcoding.com/blog/categories/accurest/)
- Spring Cloud Contract related articles from Marcin Grzejszczak's blog (http://toomuchcoding.com/blog/categories/spring-cloud-contract/)
- Groovy docs regarding JSON (http://groovy-lang.org/json.html)

2.1.7. Samples

Here you can find some samples (https://github.com/spring-cloud-samples/spring-cloud-contract-samples).

2.2. FAQ

2.2.1. Why use Spring Cloud Contract Verifier and not X ?

For the time being Spring Cloud Contract Verifier is a JVM based tool. So it could be your first pick when you're already creating software for the JVM. This project has a lot of really interesting features but especially quite a few of them definitely make Spring Cloud Contract Verifier stand out on the "market" of Consumer Driven Contract (CDC) tooling. Out of many the most interesting are:

- Possibility to do CDC with messaging
- Clear and easy to use, statically typed DSL
- Possibility to copy paste your current JSON file to the contract and only edit its elements
- Automatic generation of tests from the defined Contract
- Stub Runner functionality - the stubs are automatically downloaded at runtime from Nexus / Artifactory
- Spring Cloud integration - no discovery service is needed for integration tests

2.2.2. What is this value(consumer(), producer()) ?

One of the biggest challenges related to stubs is their reusability. Only if they can be vastly used, will they serve their purpose. What typically makes that difficult are the hard-coded values of request / response elements. For example dates or ids. Imagine the following JSON request

JSON

```
{
    "time" : "2016-10-10 20:10:15",
    "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
    "body" : "foo"
}
```

and JSON response

```JSON
{
    "time" : "2016-10-10 21:10:15",
    "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
    "body" : "bar"
}
```

Imagine the pain required to set proper value of the `time` field (let's assume that this content is generated by the database) by changing the clock in the system or providing stub implementations of data providers. The same is related to the field called `id`. Will you create a stubbed implementation of UUID generator? Makes little sense…

So as a consumer you would like to send a request that matches any form of a time or any UUID. That way your system will work as usual - will generate data and you won't have to stub anything out. Let's assume that in case of the aforementioned JSON the most important part is the `body` field. You can focus on that and provide matching for other fields. In other words you would like the stub to work like this:

```JSON
{
    "time" : "SOMETHING THAT MATCHES TIME",
    "id" : "SOMETHING THAT MATCHES UUID",
    "body" : "foo"
}
```

As far as the response goes as a consumer you need a concrete value that you can operate on. So such a JSON is valid

```JSON
{
    "time" : "2016-10-10 21:10:15",
    "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
    "body" : "bar"
}
```

As you could see in the previous sections we generate tests from contracts. So from the producer's side the situation looks much different. We're parsing the provided contract and in the test we want to send a real request to your endpoints. So for the case of a producer for the request we can't have any sort of matching. We need concrete values that the producer's backend can work on. Such a JSON would be a valid one:

```JSON
{
    "time" : "2016-10-10 20:10:15",
    "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
    "body" : "foo"
}
```

On the other hand from the point of view of the validity of the contract the response doesn't necessarily have to contain concrete values of `time` or `id`. Let's say that you generate those on the producer side - again, you'd have to do a lot of stubbing to ensure that you always return the same values. That's why from the producer's side what you might want is the following response:

```JSON
{
    "time" : "SOMETHING THAT MATCHES TIME",
    "id" : "SOMETHING THAT MATCHES UUID",
    "body" : "bar"
}
```

How can you then provide one time a matcher for the consumer and a concrete value for the producer and vice versa? In Spring Cloud Contract we're allowing you to provide a **dynamic value**. That means that it can differ for both sides of the communication. You can pass the values:

Either via the `value` method

```GROOVY
value(consumer(...), producer(...))
value(stub(...), test(...))
value(client(...), server(...))
```

or using the `$()` method

```GROOVY
$(consumer(...), producer(...))
$(stub(...), test(...))
$(client(...), server(...))
```

You can read more about this in the Contract DSL section (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html#_contract_dsl).

Calling `value()` or `$()` tells Spring Cloud Contract that you will be passing a dynamic value. Inside the `consumer()` method you pass the value that should be used on the consumer side (in the generated stub). Inside the `producer()` method you pass the value that should be used on the producer side (in the generated test).

If on one side you have passed the regular expression and you haven't passed the other, then the other side will get auto-generated.

Most often you will use that method together with the `regex` helper method. E.g. `consumer(regex('[0-9]{10}'))`.

To sum it up the contract for the aforementioned scenario would look more or less like this (the regular expression for time and UUID are simplified and most likely invalid but we want to keep things very simple in this example):

GROOVY
```
org.springframework.cloud.contract.spec.Contract.make {
                request {
                    method 'GET'
                    url '/someUrl'
                    body([
                        time : value(consumer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),
                        id: value(consumer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))
                        body: "foo"
                    ])
                }
            response {
                status 200
                body([
                        time : value(producer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),
                        id: value([producer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))
                        body: "bar"
                    ])
            }
}
```

Please read the Groovy docs related to JSON (http://groovy-lang.org/json.html) to understand how to properly structure the request / response bodies.

2.2.3. How to do Stubs versioning?

API Versioning

Let's try to answer a question what versioning really means. If you're referring to the API version then there are different approaches.

- use Hypermedia, links and do not version your API by any means
- pass versions through headers / urls

I will not try to answer a question which approach is better. Whatever suit your needs and allows you to generate business value should be picked.

Let's assume that you do version your API. In that case you should provide as many contracts as many versions you support. You can create a subfolder for every version or append it to th contract name - whatever suits you more.

JAR versioning

If by versioning you mean the version of the JAR that contains the stubs then there are essentially two main approaches.

Let's assume that you're doing Continuous Delivery / Deployment which means that you're generating a new version of the jar each time you go through the pipeline and that jar can go to production at any time. For example your jar version looks like this (it got built on the 20.10.2016 at 20:15:21) :

GROOVY
```
1.0.0.20161020-201521-RELEASE
```

In that case your generated stub jar will look like this.

```
1.0.0.20161020-201521-RELEASE-stubs.jar
```
GROOVY

In this case you should inside your `application.yml` or `@AutoConfigureStubRunner` when referencing stubs provide the latest version of the stubs. You can do that by passing the `+` sign. Example

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```
JAVA

If the versioning however is fixed (e.g. `1.0.4.RELEASE` or `2.1.1`) then you have to set the concrete value of the jar version. Example for 2.1.1.

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})
```
JAVA

Dev or prod stubs

You can manipulate the classifier to run the tests against current development version of the stubs of other services or the ones that were deployed to production. If you alter your build to deploy the stubs with the `prod-stubs` classifier once you reach production deployment then you can run tests in one case with dev stubs and one with prod stubs.

Example of tests using development version of stubs

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```
JAVA

Example of tests using production version of stubs

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:prod-stubs:8080"})
```
JAVA

You can pass those values also via properties from your deployment pipeline.

2.2.4. Common repo with contracts

Another way of storing contracts other than having them with the producer is keeping them in a common place. It can be related to security issues where the consumers can't clone the producer's code. Also if you keep contracts in a single place then you, as a producer, will know how many consumers you have and which consumer will you break with your local changes.

Repo structure

Let's assume that we have a producer with coordinates `com.example:server` and 3 consumers: `client1`, `client2`, `client3`. Then in the repository with common contracts you would have the following setup (which you can checkout here (https://github.com/spring-cloud/spring-cloud-contract/tree/1.0.x/samples/standalone/contracts):

```
├── com
│   └── example
│       └── server
│           ├── client1
│           │   └── expectation.groovy
│           ├── client2
│           │   └── expectation.groovy
│           ├── client3
│           │   └── expectation.groovy
│           └── pom.xml
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── assembly
        └── contracts.xml
```
BASH

As you can see the under the slash-delimited groupid `/` artifact id folder (`com/example/server`) you have expectations of the 3 consumers (`client1`, `client2` and `client3`). Expectations are the standard Groovy DSL contract files as described throughout this documentation. This repository has to produce a JAR file that maps one to one to the contents of the repo.

Example of a `pom.xml` inside the `server` folder.

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>server</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>Server Stubs</name>
    <description>POM used to install locally stubs for consumer side</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.4.RELEASE</version>
        <relativePath />
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
        <spring-cloud-contract.version>1.2.0.BUILD-SNAPSHOT</spring-cloud-contract.version>
        <spring-cloud-dependencies.version>Edgware.BUILD-SNAPSHOT</spring-cloud-dependencies.version>
        <excludeBuildFolders>true</excludeBuildFolders>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud-dependencies.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-contract-maven-plugin</artifactId>
                <version>${spring-cloud-contract.version}</version>
                <extensions>true</extensions>
                <configuration>
                    <!-- By default it would search under src/test/resources/ -->
                    <contractsDirectory>${project.basedir}</contractsDirectory>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
        <repository>
            <id>spring-releases</id>
            <name>Spring Releases</name>
            <url>https://repo.spring.io/release</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
```

```
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
            <id>spring-releases</id>
            <name>Spring Releases</name>
            <url>https://repo.spring.io/release</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>

</project>
```

As you can see there are no dependencies other than the Spring Cloud Contract Maven Plugin. Those poms are necessary for the consumer side to run `mvn clean install -DskipTests` to locally install stubs of the producer project.

The `pom.xml` in the root folder can look like this:

```
XML
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example.standalone</groupId>
    <artifactId>contracts</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>Contracts</name>
    <description>Contains all the Spring Cloud Contracts, well, contracts. JAR used by the producers to generate tests
and stubs</description>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <executions>
                    <execution>
                        <id>contracts</id>
                        <phase>prepare-package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                        <configuration>
                            <attach>true</attach>
                            <descriptor>${basedir}/src/assembly/contracts.xml</descriptor>
                            <!-- If you want an explicit classifier remove the following line -->
                            <appendAssemblyId>false</appendAssemblyId>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

It’s using the assembly plugin in order to build the JAR with all the contracts. Example of such setup is here:

```xml
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3
http://maven.apache.org/xsd/assembly-1.1.3.xsd">
    <id>project</id>
    <formats>
        <format>jar</format>
    </formats>
    <includeBaseDirectory>false</includeBaseDirectory>
    <fileSets>
        <fileSet>
            <directory>${project.basedir}</directory>
            <outputDirectory>/</outputDirectory>
            <useDefaultExcludes>true</useDefaultExcludes>
            <excludes>
                <exclude>**/${project.build.directory}/**</exclude>
                <exclude>mvnw</exclude>
                <exclude>mvnw.cmd</exclude>
                <exclude>.mvn/**</exclude>
                <exclude>src/**</exclude>
            </excludes>
        </fileSet>
    </fileSets>
</assembly>
```

Workflow

The workflow would look similar to the one presented in the `Step by step guide to CDC`. The only difference is that the producer doesn't own the contracts anymore. So the consumer and the producer have to work on common contracts in a common repository.

Consumer

When the **consumer** wants to work on the contracts offline, instead of cloning the producer code, the consumer team clones the common repository, goes to the required producer's folder (e.g. `com/example/server`) and runs `mvn clean install -DskipTests` to install locally the stubs converted from the contracts.

You need to have Maven installed locally (https://maven.apache.org/download.cgi)

Producer

As a **producer** it's enough to alter the Spring Cloud Contract Verifier to provide the URL and the dependency of the JAR containing the contracts:

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <configuration>
        <contractsRepositoryUrl>http://link/to/your/nexus/or/artifactory/or/sth</contractsRepositoryUrl>
        <contractDependency>
            <groupId>com.example.standalone</groupId>
            <artifactId>contracts</artifactId>
        </contractDependency>
    </configuration>
</plugin>
```

With this setup the JAR with groupid `com.example.standalone` and artifactid `contracts` will be downloaded from `http://link/to/your/nexus/or/artifactory/or/sth`. It will be then unpacked in a local temporary folder and contracts present under the `com/example/server` will be picked as the ones used to generate the tests and the stubs. Due to this convention the producer team will know which consumer teams will be broken when some incompatible changes are done.

The rest of the flow looks the same.

2.2.5. Can I have multiple base classes for tests?

Yes! Check out the Different base classes for contracts (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html#_different_base_classes_for_contracts) sections of either Gradle or Maven plugins.

2.2.6. How can I debug the request/response being sent by the generated tests client?

The generated tests all boil down to RestAssured in some form or fashion which relies on Apache HttpClient (https://hc.apache.org/httpcomponents-client-ga/). HttpClient has a facility called wire logging (https://hc.apache.org/httpcomponents-client-ga/logging.html#Wire_Logging) which logs the entire request and response to HttpClient. Spring Boot has a logging common application property (https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) for doing this sort of thing, just add this to your application properties

```
logging.level.org.apache.http.wire=DEBUG
```
PROPERTIES

2.2.7. Can I reference the request from the response?

Yes! With version 1.1.0 we've added such a possibility. On the HTTP stub server side we're providing support for this for WireMock. In case of other HTTP server stubs you'll have to implement the approach yourself.

2.3. Spring Cloud Contract Verifier HTTP

2.3.1. Gradle Project

Prerequisites

In order to use Spring Cloud Contract Verifier with WireMock you have to use Gradle or Maven plugin.

If you want to use Spock in your projects you have to add separately the `spock-core` and `spock-spring` modules. Check Spock docs for more information (https://spockframework.github.io/)

Add gradle plugin with dependencies

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier_version}"
    }
}

apply plugin: 'groovy'
apply plugin: 'spring-cloud-contract'

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-dependencies:${verifier_version}"
    }
}

dependencies {
    testCompile 'org.codehaus.groovy:groovy-all:2.4.6'
    // example with adding Spock core and Spock Spring
    testCompile 'org.spockframework:spock-core:1.0-groovy-2.4'
    testCompile 'org.spockframework:spock-spring:1.0-groovy-2.4'
    testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'
}
```
GROOVY

Gradle and Rest Assured 3.0

By default Rest Assured 2.x is added to the classpath. However in order to give the users the opportunity to use Rest Assured 3.x it's enough to add it to the plugins classpath.

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier_version}"
        classpath "io.rest-assured:rest-assured:3.0.2"
        classpath "io.rest-assured:spring-mock-mvc:3.0.2"
    }
}

depenendencies {
    // all dependencies
    // you can exclude rest-assured from spring-cloud-contract-verifier
    testCompile "io.rest-assured:rest-assured:3.0.2"
    testCompile "io.rest-assured:spring-mock-mvc:3.0.2"
}
```

That way the plugin will automatically see that Rest Assured 3.x is present on the classpath and will modify the imports accordingly.

Snapshot versions for Gradle

Add the additional snapshot repository to your build.gradle to use snapshot versions which are automatically uploaded after every successful build:

```groovy
buildscript {
    repositories {
        mavenCentral()
        mavenLocal()
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
        maven { url "http://repo.spring.io/release" }
    }
}
```

Add stubs

By default Spring Cloud Contract Verifier is looking for stubs in `src/test/resources/contracts` directory.

Directory containing stub definitions is treated as a class name, and each stub definition is treated as a single test. We assume that it contains at least one directory which will be used as test class name. If there is more than one level of nested directories all except the last one will be used as package name. So with following structure

```groovy
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Spring Cloud Contract Verifier will create test class `defaultBasePackage.MyService` with two methods

- `shouldCreateUser()`
- `shouldReturnUser()`

Run plugin

Plugin registers itself to be invoked before `check` task. You have nothing to do as long as you want it to be part of your build process. If you just want to generate tests please invoke `generateContractTests` task.

Default setup

Default Gradle Plugin setup creates the following Gradle part of the build (it's a pseudocode)

```groovy
contracts {
    targetFramework = 'JUNIT'
    testMode = 'MockMvc'
    generatedTestSourcesDir = project.file("${project.buildDir}/generated-test-sources/contracts")
    contractsDslDir = "${project.rootDir}/src/test/resources/contracts"
    basePackageForTests = 'org.springframework.cloud.verifier.tests'
    stubsOutputDir = project.file("${project.buildDir}/stubs")

    // the following properties are used when you want to provide where the JAR with contract lays
    contractDependency {
        stringNotation = ''
    }
    contractsPath = ''
    contractsWorkOffline = false
    contractRepository {
        cacheDownloadedContracts(true)
    }
}

tasks.create(type: Jar, name: 'verifierStubsJar', dependsOn: 'generateClientStubs') {
    baseName = project.name
    classifier = contracts.stubsSuffix
    from contractVerifier.stubsOutputDir
}

project.artifacts {
    archives task
}

tasks.create(type: Copy, name: 'copyContracts') {
    from contracts.contractsDslDir
    into contracts.stubsOutputDir
}

verifierStubsJar.dependsOn 'copyContracts'

publishing {
    publications {
        stubs(MavenPublication) {
            artifactId project.name
            artifact verifierStubsJar
        }
    }
}
```

Configure plugin

To change default configuration just add `contracts` snippet to your Gradle config

```groovy
contracts {
    testMode = 'MockMvc'
    baseClassForTests = 'org.mycompany.tests'
    generatedTestSourcesDir = project.file('src/generatedContract')
}
```

Configuration options

- **testMode** - defines mode for acceptance tests. By default MockMvc which is based on Spring's MockMvc. It can also be changed to **JaxRsClient** or to **Explicit** for real HTTP calls.
- **imports** - array with imports that should be included in generated tests (for example ['org.myorg.Matchers']). By default empty array []
- **staticImports** - array with static imports that should be included in generated tests(for example ['org.myorg.Matchers.*']). By default empty array []
- **basePackageForTests** - specifies base package for all generated tests. By default set to org.springframework.cloud.verifier.tests
- **baseClassForTests** - base class for all generated tests. By default `spock.lang.Specification` if using Spock tests.
- **packageWithBaseClasses** - instead of providing a fixed value for base class you can provide a package where all the base classes lay. Takes precedence over **baseClassForTests**.
- **baseClassMappings** - explicitly map contract package to a FQN of a base class. Takes precedence over **packageWithBaseClasses** and **baseClassForTests**.
- **ruleClassForTests** - specifies Rule which should be added to generated test classes.
- **ignoredFiles** - Ant matcher allowing defining stub files for which processing should be skipped. By default empty array []

- **contractsDslDir** - directory containing contracts written using the GroovyDSL. By default `$rootDir/src/test/resources/contracts`
- **generatedTestSourcesDir** - test source directory where tests generated from Groovy DSL should be placed. By default `$buildDir/generated-test-sources/contractVerifier`
- **stubsOutputDir** - dir where the generated WireMock stubs from Groovy DSL should be placed
- **targetFramework** - the target test framework to be used; currently Spock and JUnit are supported with JUnit being the default framework

The following properties are used when you want to provide where the JAR with contract lays

- **contractDependency** - the Dependency that provides `groupid:artifactid:version:classifier` coordinates. You can use the `contractDependency` closure to set it up
- **contractsPath** - if contract deps are downloaded will default to `groupid/artifactid` where `groupid` will be slash separated. Otherwise will scan contracts under provided directory
- **contractsWorkOffline** - in order not to download the dependencies each time you can download them once and work offline afterwards (reuse local Maven repo)

Single base class for all tests

When using Spring Cloud Contract Verifier in default MockMvc you need to create a base specification for all generated acceptance tests. In this class you need to point to endpoint which should be verified.

GROOVY

```
abstract class BaseMockMvcSpec extends Specification {

    def setup() {
        RestAssuredMockMvc.standaloneSetup(new PairIdController())
    }

    void isProperCorrelationId(Integer correlationId) {
        assert correlationId == 123456
    }

    void isEmpty(String value) {
        assert value == null
    }

}
```

In case of using `Explicit` mode, you can use base class to initialize the whole tested app similarly as in regular integration tests. In case of `JAXRSCLIENT` mode this base class should also contain `protected WebTarget webTarget` field, right now the only option to test JAX-RS API is to start a web server.

Different base classes for contracts

If your base classes differ between contracts you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- follow a convention by providing the `packageWithBaseClasses`
- provide explicit mapping via `baseClassMappings`

Convention

The convention is such that if you have a contract under e.g. `src/test/resources/contract/foo/bar/baz/` and provide the value of the `packageWithBaseClasses` property to `com.example.base` then we will assume that there is a `BarBazBase` class under `com.example.base` package. In other words we take last two parts of package if they exist and form a class with a `Base` suffix. Takes precedence over **baseClassForTests**. Example of usage in the `contracts` closure:

GROOVY

```
packageWithBaseClasses = 'com.example.base'
```

Mapping

You can manually map a regular expression of the contract's package to fully qualified name of the base class for the matched contract. Let's take a look at the following example:

```
baseClassForTests = "com.example.FooBase"
baseClassMappings {
    baseClassMapping('.*/com/.*', 'com.example.ComBase')
    baseClassMapping('.*/bar/.*':'com.example.BarBase')
}
```
GROOVY

Let's assume that you have contracts under - `src/test/resources/contract/com/` - `src/test/resources/contract/foo/`

By providing the `baseClassForTests` we have a fallback in case mapping didn't succeed (you could also provide the `packageWithBaseClasses` as fallback). That way the tests generated from `src/test/resources/contract/com/` contracts will be extending the `com.example.ComBase` whereas the rest of tests will extend `com.example.FooBase`.

Invoking generated tests

To ensure that provider side is complaint with defined contracts, you need to invoke:

```
./gradlew generateContractTests test
```
BASH

Spring Cloud Contract Verifier on consumer side

In consumer service you need to configure Spring Cloud Contract Verifier plugin in exactly the same way as in case of provider. If you don't want to use Stub Runner then you need to copy contracts stored in `src/test/resources/contracts` and generate WireMock json stubs using:

```
./gradlew generateClientStubs
```
BASH

Note that `stubsOutputDir` option has to be set for stub generation to work.

When present, json stubs can be used in consumer automated tests.

```
@ContextConfiguration(loader == SpringApplicationContextLoader, classes == Application)
class LoanApplicationServiceSpec extends Specification {

 @ClassRule
 @Shared
 WireMockClassRule wireMockRule == new WireMockClassRule()

 @Autowired
 LoanApplicationService sut

 def 'should successfully apply for loan'() {
   given:
    LoanApplication application =
            new LoanApplication(client: new Client(clientPesel: '12345678901'), amount: 123.123)
   when:
    LoanApplicationResult loanApplication == sut.loanApplication(application)
   then:
    loanApplication.loanApplicationStatus == LoanApplicationStatus.LOAN_APPLIED
    loanApplication.rejectionReason == null
 }
}
```
GROOVY

Underneath LoanApplication makes a call to FraudDetection service. This request is handled by WireMock server configured using stubs generated by Spring Cloud Contract Verifier.

2.3.2. Using in your Maven project

Add maven plugin

Add the Spring Cloud Contract BOM

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud-dependencies.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```
XML

Next, the `Spring Cloud Contract Verifier` Maven plugin

XML

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
    </configuration>
</plugin>
```

You can read more in the Spring Cloud Contract Maven Plugin Docs
(https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract-maven-plugin/)

Maven and Rest Assured 3.0

By default Rest Assured 2.x is added to the classpath. However in order to give the users the opportunity to use Rest Assured 3.x it's enough to add it to the plugins classpath.

GROOVY

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>com.example</packageWithBaseClasses>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-contract-verifier</artifactId>
            <version>${spring-cloud-contract.version}</version>
        </dependency>
        <dependency>
           <groupId>io.rest-assured</groupId>
           <artifactId>rest-assured</artifactId>
           <version>3.0.2</version>
           <scope>compile</scope>
        </dependency>
        <dependency>
           <groupId>io.rest-assured</groupId>
           <artifactId>spring-mock-mvc</artifactId>
           <version>3.0.2</version>
           <scope>compile</scope>
        </dependency>
    </dependencies>
</plugin>

<dependencies>
    <!-- all dependencies -->
    <!-- you can exclude rest-assured from spring-cloud-contract-verifier -->
    <dependency>
       <groupId>io.rest-assured</groupId>
       <artifactId>rest-assured</artifactId>
       <version>3.0.2</version>
       <scope>test</scope>
    </dependency>
    <dependency>
       <groupId>io.rest-assured</groupId>
       <artifactId>spring-mock-mvc</artifactId>
       <version>3.0.2</version>
       <scope>test</scope>
    </dependency>
</dependencies>
```

That way the plugin will automatically see that Rest Assured 3.x is present on the classpath and will modify the imports accordingly.

Snapshot versions for Maven

For Snapshot / Milestone versions you have to add the following section to your `pom.xml`

```xml
<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
</pluginRepositories>
```

Add stubs

By default Spring Cloud Contract Verifier is looking for stubs in `src/test/resources/contracts` directory. Directory containing stub definitions is treated as a class name, and each stub definition is treated as a single test. We assume that it contains at least one directory which will be used as test class name. If there is more than one level of nested directories all except the last one will be used as package name. So with following structure

```groovy
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Spring Cloud Contract Verifier will create test class `defaultBasePackage.MyService` with two methods - `shouldCreateUser()` - `shouldReturnUser()`

Run plugin

Plugin goal `generateTests` is assigned to be invoked in phase `generate-test-sources`. You have nothing to do as long as you want it to be part of your build process. If you just want to generate tests please invoke `generateTests` goal.

Configure plugin

To change default configuration just add `configuration` section to plugin definition or `execution` definition.

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>convert</goal>
                <goal>generateStubs</goal>
                <goal>generateTests</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <basePackageForTests>org.springframework.cloud.verifier.twitter.place</basePackageForTests>
        <baseClassForTests>org.springframework.cloud.verifier.twitter.place.BaseMockMvcSpec</baseClassForTests>
    </configuration>
</plugin>
```

Important configuration options

- **testMode** - defines mode for acceptance tests. By default `MockMvc` which is based on Spring's MockMvc. It can also be changed to `JaxRsClient` or to `Explicit` for real HTTP calls.
- **basePackageForTests** - specifies base package for all generated tests. By default set to `org.springframework.cloud.verifier.tests`.
- **ruleClassForTests** - specifies Rule which should be added to generated test classes.
- **baseClassForTests** - base class for generated tests. By default `spock.lang.Specification` if using Spock tests.
- **contractsDirectory** - directory containing contracts written using the GroovyDSL. By default `/src/test/resources/contracts`.
- **testFramework** - the target test framework to be used; currently Spock and JUnit are supported with JUnit being the default framework
- **packageWithBaseClasses** - instead of providing a fixed value for base class you can provide a package where all the base classes lay. The convention is such that if you have a contract under `src/test/resources/contract/foo/bar/baz/` and provide the value of this property to `com.example.base` then we will assume that there is a `BarBazBase` class under `com.example.base` package. Takes precedence over **baseClassForTests**
- **baseClassMappings** - list of base class mappings that where you have to provide `contractPackageRegex` which is checked against the package in which the contract lays and `baseClassFQN` that maps to fully qualified name of the base class for the matched contract. If you have a contract under `src/test/resources/contract/foo/bar/baz/` and map the property `.* → com.example.base.BaseClass` then the test class generated from these contracts will extend `com.example.base.BaseClass`. Takes precedence over **packageWithBaseClasses** and **baseClassForTests**.

If you want to download your contract definitions from a Maven repository you can use

- **contractDependency** - the contract dependency that contains all the packaged contracts
- **contractsPath** - path to concrete contracts in the JAR with packaged contracts. Defaults to `groupid/artifactid` where `groupid` is slash separated.
- **contractsWorkOffline** - if the dependencies should be downloaded or local Maven only should be reused
- **contractsRepositoryUrl** - **DEPRECATED PROPERTY - please use the** `contractRepository` **closure** - URL to a repo with the artifacts with contracts, if not provided should use the current Maven ones
- **contractRepository** - closure where you can define properties related to repository with contracts
  - **username** - username to be used to connect to the repo
  - **password** - username to be used to connect to the repo
  - **proxyHost** - proxy host to be used to connect to the repo
  - **proxyPort** - proxy port to be used to connect to the repo
  - **cacheDownloadedContracts** - if you want to reuse download JARs that contain contract definitions. We cache only non-snapshot, explicitly provided versions (e.g. `+` or `1.0.0.BUILD-SNAPSHOT` won't get cached). By default this feature is turned on.

Single base class for all tests

When using Spring Cloud Contract Verifier in default MockMvc you need to create a base specification for all generated acceptance tests. In this class you need to point to endpoint which should be verified.

```groovy
package org.mycompany.tests

import org.mycompany.ExampleSpringController
import com.jayway.restassured.module.mockmvc.RestAssuredMockMvc
import spock.lang.Specification

class  MvcSpec extends Specification {
  def setup() {
   RestAssuredMockMvc.standaloneSetup(new ExampleSpringController())
  }
}
```

In case of using `Explicit` mode, you can use base class to initialize the whole tested app similarly as in regular integration tests. In case of `JAXRSCLIENT` mode this base class should also contain `protected WebTarget webTarget` field, right now the only option to test JAX-RS API is to start a web server.

Different base classes for contracts

If your base classes differ between contracts you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- follow a convention by providing the `packageWithBaseClasses`
- provide explicit mapping via `baseClassMappings`

Convention

The convention is such that if you have a contract under e.g. `src/test/resources/contract/hello/v1/` and provide the value of the `packageWithBaseClasses` property to `hello` then we will assume that there is a `HelloV1Base` class under `hello` package. In other words we take last two parts of package if they exist and form a class with a `Base` suffix. Takes precedence over **baseClassForTests**. Example of usage:

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <configuration>
        <packageWithBaseClasses>hello</packageWithBaseClasses>
    </configuration>
</plugin>
```

Mapping

You can manually map a regular expression of the contract's package to fully qualified name of the base class for the matched contract. You have to provide a list `baseClassMappings` of `baseClassMapping` that takes a `contractPackageRegex` to `baseClassFQN` mapping. Let's take a look at the following example:

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <configuration>
        <baseClassForTests>com.example.FooBase</baseClassForTests>
        <baseClassMappings>
            <baseClassMapping>
                <contractPackageRegex>.*com.*</contractPackageRegex>
                <baseClassFQN>com.example.TestBase</baseClassFQN>
            </baseClassMapping>
        </baseClassMappings>
    </configuration>
</plugin>
```

Let's assume that you have contracts under - `src/test/resources/contract/com/` - `src/test/resources/contract/foo/`

By providing the `baseClassForTests` we have a fallback in case mapping didn't succeed (you could also provide the `packageWithBaseClasses` as fallback). That way the tests generated from `src/test/resources/contract/com/` contracts will be extending the `com.example.ComBase` whereas the rest of tests will extend `com.example.FooBase`.

Invoking generated tests

Spring Cloud Contract Maven Plugin generates verification code into directory `/generated-test-sources/contractVerifier` and attach this directory to `testCompile` goal.

For Groovy Spock code use:

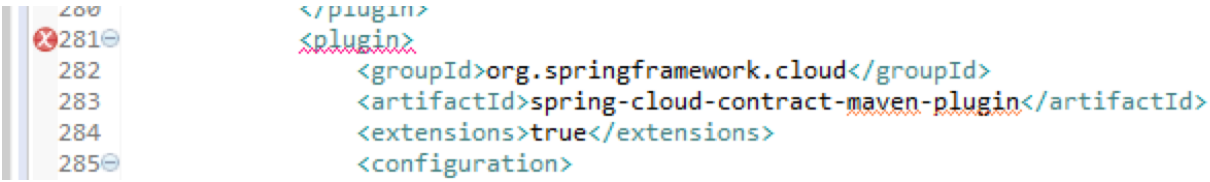
```xml
<plugin>
    <groupId>org.codehaus.gmavenplus</groupId>
    <artifactId>gmavenplus-plugin</artifactId>
    <version>1.5</version>
    <executions>
        <execution>
            <goals>
                <goal>testCompile</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <testSources>
            <testSource>
                <directory>${project.basedir}/src/test/groovy</directory>
                <includes>
                    <include>**/*.groovy</include>
                </includes>
            </testSource>
            <testSource>
                <directory>${project.build.directory}/generated-test-sources/contractVerifier</directory>
                <includes>
                    <include>**/*.groovy</include>
                </includes>
            </testSource>
        </testSources>
    </configuration>
</plugin>
```

To ensure that provider side is complaint with defined contracts, you need to invoke `mvn generateTest test`

FAQ with Maven Plugin

Maven Plugin and STS

In case you see the following exception while using STS



when you click on the marker you should see sth like this

```bash
 plugin:1.1.0.M1:convert:default-convert:process-test-resources) org.apache.maven.plugin.PluginExecutionException:
Execution default-convert of goal org.springframework.cloud:spring-
 cloud-contract-maven-plugin:1.1.0.M1:convert failed. at
org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPluginManager.java:145) at
 org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(MavenImpl.java:331) at
org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(MavenImpl.java:1362) at
...
 org.eclipse.core.internal.jobs.Worker.run(Worker.java:55) Caused by: java.lang.NullPointerException at
 org.eclipse.m2e.core.internal.builder.plexusbuildapi.EclipseIncrementalBuildContext.hasDelta(EclipseIncrementalBuildCont
ext.java:53) at
 org.sonatype.plexus.build.incremental.ThreadBuildContext.hasDelta(ThreadBuildContext.java:59) at
```

In order to fix this issue just provide the following section in your `pom.xml`

```xml
<build>
    <pluginManagement>
        <plugins>
            <!--This plugin's configuration is used to store Eclipse m2e settings
                only. It has no influence on the Maven build itself. -->
            <plugin>
                <groupId>org.eclipse.m2e</groupId>
                <artifactId>lifecycle-mapping</artifactId>
                <version>1.0.0</version>
                <configuration>
                    <lifecycleMappingMetadata>
                        <pluginExecutions>
                             <pluginExecution>
                                <pluginExecutionFilter>
                                    <groupId>org.springframework.cloud</groupId>
                                    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
                                    <versionRange>[1.0,)</versionRange>
                                    <goals>
                                        <goal>convert</goal>
                                    </goals>
                                </pluginExecutionFilter>
                                <action>
                                    <execute />
                                </action>
                            </pluginExecution>
                        </pluginExecutions>
                    </lifecycleMappingMetadata>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

Spring Cloud Contract Verifier on consumer side

You can actually use the Spring Cloud Contract Verifier also for the consumer side! You can use the plugin so that it only converts the contracts and generates the stubs. To achieve that you need to configure Spring Cloud Contract Verifier plugin in exactly the same way as in case of provider. You need to copy contracts stored in `src/test/resources/contracts` and generate WireMock json stubs using: `mvn generateStubs` command. By default generated WireMock mapping is stored in directory `target/mappings`. Your project should create from this generated mappings additional artifact with classifier `stubs` for easy deploy to maven repository.

Sample configuration:

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${verifier-plugin.version}</version>
    <executions>
        <execution>
            <goals>
                <goal>convert</goal>
                <goal>generateStubs</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

When present, json stubs can be used in consumer automated tests.

GROOVY

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureStubRunner
public class LoanApplicationServiceTests {

  @Autowired
  LoanApplicationService service;

  @Test
  public void shouldSuccessfullyApplyForLoan() {
    //given:
    LoanApplication application =
            new LoanApplication(new Client("12345678901"), 123.123);
    //when:
    LoanApplicationResult loanApplication = service.loanApplication(application);
    // then:
    assertThat(loanApplication.loanApplicationStatus).isEqualTo(LoanApplicationStatus.LOAN_APPLIED);
    assertThat(loanApplication.rejectionReason).isNull();
  }
}
```

Underneath `LoanApplication` makes a call to the `FraudDetection` service. This request is handled by a WireMock server configured using stubs generated by Spring Cloud Contract Verifier.

2.3.3. Scenarios

It's possible to handle scenarios with Spring Cloud Contract Verifier. All you need to do is to stick to proper naming convention while creating your contracts. The convention requires to include order number followed by the underscore.

```
my_contracts_dir\
  scenario1\
    1_login.groovy
    2_showCart.groovy
    3_logout.groovy
```

Such tree will cause Spring Cloud Contract Verifier generating WireMock's scenario with name `scenario1` and three steps:

- login marked as `Started` pointing to:
- showCart marked as `Step1` pointing to:
- logout marked as `Step2` which will close the scenario.

More details about WireMock scenarios can be found under http://wiremock.org/stateful-behaviour.html (http://wiremock.org/stateful-behaviour.html)

Spring Cloud Contract Verifier will also generate tests with guaranteed order of execution.

2.3.4. Stubs and transitive dependencies

The Maven and Gradle plugin that we're created are adding the tasks that create the stubs jar for you. What can be problematic is that when reusing the stubs you can by mistake import all of that stub dependencies! When building a Maven artifact even though you have a couple of different jars, all of them share one pom:

BASH

```
├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar
├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar.sha1
├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar
├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar.sha1
├── github-webhook-0.0.1.BUILD-SNAPSHOT.jar
├── github-webhook-0.0.1.BUILD-SNAPSHOT.pom
├── github-webhook-0.0.1.BUILD-SNAPSHOT-stubs.jar
├── ...
└── ...
```

There are three possibilities of working with those dependencies so as not to have any issues with transitive dependencies.

Mark all application dependencies as optional

If in the `github-webhook` application we would mark all of our dependencies as optional, when you include the `github-webhook` stubs in another application (or when that dependency gets downloaded by Stub Runner) then, since all of the depenencies are optional, they will not get downloaded.

Create a separate artifactid for stubs

If you create a separate artifactid then you can set it up in whatever way you wish. For example by having no dependencies at all.

Exclude dependencies on the consumer side

As a consumer, if you add the stub dependency to your classpath you can explicitly exclude the unwanted dependencies.

2.4. Spring Cloud Contract Verifier Messaging

Spring Cloud Contract Verifier allows you to verify your application that uses messaging as means of communication. All of our integrations are working with Spring but you can also create one yourself and use it.

2.4.1. Integrations

You can use one of the four integration configurations:

- Apache Camel
- Spring Integration
- Spring Cloud Stream
- Spring AMQP

Since we're using Spring Boot then if you have added one of the aforementioned libraries to the classpath then automatically all the messaging configuration will be set up.

Remember to put `@AutoConfigureMessageVerifier` on the base class of your generated tests. Otherwise messaging part of Spring Cloud Contract Verifier will not work.

If you want to use Spring Cloud Stream remember to add a `org.springframework.cloud:spring-cloud-stream-test-support` dependency.

Maven

XML

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```

Gradle

GROOVY

```
testCompile "org.springframework.cloud:spring-cloud-stream-test-support"
```

2.4.2. Manual Integration Testing

The main interface used by the tests is the `org.springframework.cloud.contract.verifier.messaging.MessageVerifier` . It defines how to send and receive messages. You can create your own implementation to achieve the same goal.

In the a test you can inject a `ContractVerifierMessageExchange` to send and receive messages that follow the contract. Then add `@AutoConfigureMessageVerifier` to your test, e.g.

JAVA

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureMessageVerifier
public static class MessagingContractTests {

  @Autowired
  private MessageVerifier verifier;
  ...
}
```

If your tests require stubs as well, then `@AutoConfigureStubRunner` includes the messaging configuration, so you only need the one annotation.

2.4.3. Publisher side test generation

Having the `input` or `outputMessage` sections in your DSL will result in creation of tests on the publisher's side. By default JUnit tests will be created, however there is also a possibility to create Spock tests.

There are 3 main scenarios that we should take into consideration:

- Scenario 1: there is no input message that produces an output one. The output message is triggered by a component inside the application (e.g. scheduler)
- Scenario 2: the input message triggers an output message
- Scenario 3: the input message is consumed and there is no output message

> The destination passed to `messageFrom` or `sentTo` can have different meanings for different messaging implementations. For **Stream** and **Integration** it's first resolved as a `destination` of a channel, and then if there is no such `destination` it's resolved as a channel name. For **Camel** that's a certain component (e.x. `jms`).

Example for Camel:

Scenario 1 (no input message)

For the given contract:

```
GROOVY
def contractDsl = Contract.make {
    label 'some_label'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('activemq:output')
        body('''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
            messagingContentType(applicationJson())
        }
    }
}
```

The following JUnit test will be created:

```
GROOVY
'''
 // when:
  bookReturnedTriggered();

 // then:
  ContractVerifierMessage response = contractVerifierMessaging.receive("activemq:output");
  assertThat(response).isNotNull();
  assertThat(response.getHeader("BOOK-NAME")).isNotNull();
  assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");
  assertThat(response.getHeader("contentType")).isNotNull();
  assertThat(response.getHeader("contentType").toString()).isEqualTo("application/json");
 // and:
  DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));
  assertThatJson(parsedJson).field("bookName").isEqualTo("foo");
'''
```

And the following Spock test would be created:

```
GROOVY
'''
 when:
  bookReturnedTriggered()

 then:
  ContractVerifierMessage response = contractVerifierMessaging.receive('activemq:output')
  assert response != null
  response.getHeader('BOOK-NAME')?.toString()  == 'foo'
  response.getHeader('contentType')?.toString()  == 'application/json'
 and:
  DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))
  assertThatJson(parsedJson).field("bookName").isEqualTo("foo")

'''
```

Scenario 2 (output triggered by input)

For the given contract:

```
def contractDsl = Contract.make {
    label 'some_label'
    input {
        messageFrom('jms:input')
        messageBody([
                bookName: 'foo'
        ])
        messageHeaders {
            header('sample', 'header')
        }
    }
    outputMessage {
        sentTo('jms:output')
        body([
                bookName: 'foo'
        ])
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

GROOVY

The following JUnit test will be created:

```
'''
// given:
 ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
  "{\\"bookName\\":\\"foo\\"}"
, headers()
  .header("sample", "header"));

// when:
 contractVerifierMessaging.send(inputMessage, "jms:input");

// then:
 ContractVerifierMessage response = contractVerifierMessaging.receive("jms:output");
 assertThat(response).isNotNull();
 assertThat(response.getHeader("BOOK-NAME")).isNotNull();
 assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");
// and:
 DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));
 assertThatJson(parsedJson).field("bookName").isEqualTo("foo");
'''
```

GROOVY

And the following Spock test would be created:

```
"""\
given:
   ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
    '''{"bookName":"foo"}''',
    ['sample': 'header']
  )

when:
   contractVerifierMessaging.send(inputMessage, 'jms:input')

then:
   ContractVerifierMessage response = contractVerifierMessaging.receive('jms:output')
   assert response != null
   response.getHeader('BOOK-NAME')?.toString()  == 'foo'
and:
   DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))
   assertThatJson(parsedJson).field("bookName").isEqualTo("foo")
"""
```

GROOVY

Scenario 3 (no output message)

For the given contract:

```
def contractDsl = Contract.make {
    label 'some_label'
    input {
        messageFrom('jms:delete')
        messageBody([
                bookName: 'foo'
        ])
        messageHeaders {
            header('sample', 'header')
        }
        assertThat('bookWasDeleted()')
    }
}
```

GROOVY

The following JUnit test will be created:

```
'''
// given:
 ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
    "{\\"bookName\\":\\"foo\\"}"
, headers()
    .header("sample", "header"));

// when:
 contractVerifierMessaging.send(inputMessage, "jms:delete");

// then:
 bookWasDeleted();
'''
```

GROOVY

And the following Spock test would be created:

```
'''
given:
     ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
        \'\'\'{"bookName":"foo"}\'\'\',
        ['sample': 'header']
    )

when:
     contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:
     noExceptionThrown()
     bookWasDeleted()
'''
```

GROOVY

2.4.4. Consumer Stub Side generation

Unlike the HTTP part - in Messaging we need to publish the Groovy DSL inside the JAR with a stub. Then it's parsed on the consumer side and proper stubbed routes are created.

For more information please consult the Stub Runner Messaging sections.

Maven

XML

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-stream-test-support</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Dalston.BUILD-SNAPSHOT</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Gradle

GROOVY

```
ext {
    contractsDir = file("mappings")
    stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder from which JAR will be created
// verifierStubsJar - JAR with a provided stub suffix
// the presented publication is also added by the plugin but you can modify it as you wish

publishing {
    publications {
        stubs(MavenPublication) {
            artifactId "${project.name}-stubs"
            artifact verifierStubsJar
        }
    }
}
```

2.5. Spring Cloud Contract Stub Runner

One of the issues that you could have encountered while using Spring Cloud Contract Verifier was to pass the generated WireMock JSON stubs from the server side to the client side (or various clients). The same takes place in terms of client side generation for messaging.

Copying the JSON files / setting the client side for messaging manually is out of the question.

That's why we'll introduce Spring Cloud Contract Stub Runner that can download and run the stubs automatically for you.

2.5.1. Snapshot versions

Add the additional snapshot repository to your build.gradle to use snapshot versions which are automatically uploaded after every successful build:

Maven

XML

```
<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
</pluginRepositories>
```

Gradle

GROOVY

```
buildscript {
    repositories {
        mavenCentral()
        mavenLocal()
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
        maven { url "http://repo.spring.io/release" }
    }
```

2.5.2. Publishing stubs as JARs

The easiest approach would be to centralize the way stubs are kept. For example you can keep them as JARs in a Maven repository.

For both Maven and Gradle the setup comes out of the box. But you can customize it if you want to.

Maven

XML

```
<!-- First disable the default jar setup in the properties section-->
<!-- we don't want the verifier to do a jar for us -->
<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>

<!-- Next add the assembly plugin to your build -->
<!-- we want the assembly plugin to generate the JAR -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
        <execution>
            <id>stub</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>single</goal>
            </goals>
            <inherited>false</inherited>
            <configuration>
                <attach>true</attach>
                <descriptor>$/Users/ryanjbaxter/git-repos/spring-cloud/scripts/docs/../src/assembly/stub.xml</descriptor>
            </configuration>
        </execution>
    </executions>
</plugin>

<!-- Finally setup your assembly. Below you can find the contents of src/main/assembly/stub.xml -->
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3
http://maven.apache.org/xsd/assembly-1.1.3.xsd">
    <id>stubs</id>
    <formats>
        <format>jar</format>
    </formats>
    <includeBaseDirectory>false</includeBaseDirectory>
    <fileSets>
        <fileSet>
            <directory>src/main/java</directory>
            <outputDirectory>/</outputDirectory>
            <includes>
                <include>**com/example/model/*.*</include>
            </includes>
        </fileSet>
        <fileSet>
            <directory>${project.build.directory}/classes</directory>
            <outputDirectory>/</outputDirectory>
            <includes>
                <include>**com/example/model/*.*</include>
            </includes>
        </fileSet>
        <fileSet>
            <directory>${project.build.directory}/snippets/stubs</directory>
            <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</outputDirectory>
            <includes>
                <include>**/*</include>
            </includes>
        </fileSet>
        <fileSet>
            <directory>$/Users/ryanjbaxter/git-repos/spring-
cloud/scripts/docs/../src/test/resources/contracts</directory>
            <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/contracts</outputDirectory>
            <includes>
                <include>**/*.groovy</include>
            </includes>
        </fileSet>
    </fileSets>
</assembly>
```

Gradle

GROOVY

```
ext {
    contractsDir = file("mappings")
    stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder from which JAR will be created
// verifierStubsJar - JAR with a provided stub suffix
// the presented publication is also added by the plugin but you can modify it as you wish

publishing {
    publications {
        stubs(MavenPublication) {
            artifactId "${project.name}-stubs"
            artifact verifierStubsJar
        }
    }
}
```

2.6. Stub Runner Core

Runs stubs for service collaborators. Treating stubs as contracts of services allows to use stub-runner as an implementation of Consumer Driven Contracts (http://martinfowler.com/articles/consumerDrivenContracts.html).

Stub Runner allows you to automatically download the stubs of the provided dependencies (or pick those from the classpath), start WireMock servers for them and feed them with proper stub definitions. For messaging, special stub routes are defined.

2.6.1. Retrieving stubs

You can pick the following options of acquiring stubs

- Aether based solution that downloads JARs with stubs from Artifactory / Nexus
- Classpath scanning solution that searches classpath via pattern to retrieve stubs
- Write your own implementation of the `org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder` for full customization

The latter example is described in the Custom Stub Runner section.

Stub downloading

If you provide the `stubrunner.repositoryRoot` or `stubrunner.workOffline` flag will be set to `true` then Stub Runner will connect to the given server and download the required jars. It will then unpack the JAR to a temporary folder and reference those files in further contract processing.

Example:

JAVA

```
@AutoConfigureStubRunner(repositoryRoot="http://foo.bar", ids = "com.example:beer-api-producer:+:stubs:8095")
```

Classpath scanning

If you **DON'T** provide the `stubrunner.repositoryRoot` and `stubrunner.workOffline` flag will be set to `false` (that's the default) then classpath will get scanned. Let's look at the following example:

JAVA

```
@AutoConfigureStubRunner(ids = {
    "com.example:beer-api-producer:+:stubs:8095",
    "com.example.foo:bar:1.0.0:superstubs:8096"
})
```

If you've added the dependencies to your classpath

Maven

```xml
<dependency>
    <groupId>com.example</groupId>
    <artifactId>beer-api-producer-restdocs</artifactId>
    <classifier>stubs</classifier>
    <version>0.0.1-SNAPSHOT</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>*</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.example.foo</groupId>
    <artifactId>bar</artifactId>
    <classifier>superstubs</classifier>
    <version>1.0.0</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>*</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Gradle

```groovy
testCompile("com.example:beer-api-producer-restdocs:0.0.1-SNAPSHOT:stubs") {
    transitive = false
}
testCompile("com.example.foo:bar:1.0.0:superstubs") {
    transitive = false
}
```

Then the following locations on your classpath will get scanned. For `com.example:beer-api-producer-restdocs`

- /META-INF/com.example/beer-api-producer-restdocs/*/.*
- /contracts/com.example/beer-api-producer-restdocs/*/.*
- /mappings/com.example/beer-api-producer-restdocs/*/.*

and `com.example.foo:bar`

- /META-INF/com.example.foo/bar/*/.*
- /contracts/com.example.foo/bar/*/.*
- /mappings/com.example.foo/bar/*/.*

As you can see you have to explicitly provide the group and artifact ids when packaging the producer stubs.

The producer would setup the contracts like this:

```bash
└── src
    └── test
        └── resources
            └── contracts
                └── com.example
                    └── beer-api-producer-restdocs
                        └── nested
                            └── contract3.groovy
```

To achieve proper stub packaging.

Or using the Maven `assembly` plugin (https://github.com/spring-cloud-samples/spring-cloud-contract-samples/blob/master/producer_with_restdocs/pom.xml) or Gradle Jar (https://github.com/spring-cloud-samples/spring-cloud-contract-samples/blob/master/producer_with_restdocs/build.gradle) task you have to create the following structure in your stubs jar.

```
└── META-INF
    └── com.example
        └── beer-api-producer-restdocs
            └── 2.0.0
                ├── contracts
                │   └── nested
                │       └── contract2.groovy
                └── mappings
                    └── mapping.json
```

BASH

By maintaining this structure classpath gets scanned and you can profit from the messaging / HTTP stubs without the need to download artifacts.

2.6.2. Running stubs

Limitations

There might be a problem with StubRunner shutting down ports between tests. You might have a situation in which you get port conflicts. As long as you use the same context across tests everything works fine. But when the context are different (e.g. different stubs or different profiles) then you have to either use `@DirtiesContext` to shut down the stub servers, or else run them on different ports per test.

Running using main app

You can set the following options to the main class:

```
-c, --classifier                Suffix for the jar containing stubs (e.
                                  g. 'stubs' if the stub jar would
                                  have a 'stubs' classifier for stubs:
                                  foobar-stubs ). Defaults to 'stubs'
                                  (default: stubs)
--maxPort, --maxp <Integer>     Maximum port value to be assigned to
                                  the WireMock instance. Defaults to
                                  15000 (default: 15000)
--minPort, --minp <Integer>     Minimum port value to be assigned to
                                  the WireMock instance. Defaults to
                                  10000 (default: 10000)
-p, --password                  Password to user when connecting to
                                  repository
--phost, --proxyHost            Proxy host to use for repository
                                  requests
--pport, --proxyPort [Integer]  Proxy port to use for repository
                                  requests
-r, --root                      Location of a Jar containing server
                                  where you keep your stubs (e.g. http:
                                  //nexus.
                                  net/content/repositories/repository)
-s, --stubs                     Comma separated list of Ivy
                                  representation of jars with stubs.
                                  Eg. groupid:artifactid1,groupid2:
                                  artifactid2:classifier
-u, --username                  Username to user when connecting to
                                  repository
--wo, --workOffline             Switch to work offline. Defaults to
                                  'false'
```

GROOVY

HTTP Stubs

Stubs are defined in JSON documents, whose syntax is defined in WireMock documentation (http://wiremock.org/stubbing.html)

Example:

```
{
    "request": {
        "method": "GET",
        "url": "/ping"
    },
    "response": {
        "status": 200,
        "body": "pong",
        "headers": {
            "Content-Type": "text/plain"
        }
    }
}
```

JAVASCRIPT

Viewing registered mappings

Every stubbed collaborator exposes list of defined mappings under `__/admin/` endpoint.

Messaging Stubs

Depending on the provided Stub Runner dependency and the DSL the messaging routes are automatically set up.

2.7. Stub Runner JUnit Rule

Stub Runner comes with a JUnit rule thanks to which you can very easily download and run stubs for given group and artifact id:

JAVA

```
@ClassRule public static StubRunnerRule rule = new StubRunnerRule()
        .repoRoot(repoRoot())
        .downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")
        .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer");
```

After that rule gets executed Stub Runner connects to your Maven repository and for the given list of dependencies tries to:

- download them
- cache them locally
- unzip them to a temporary folder
- start a WireMock server for each Maven dependency on a random port from the provided range of ports / provided port
- feed the WireMock server with all JSON files that are valid WireMock definitions
- can also send messages (remember to pass an implementation of `MessageVerifier` interface)

Stub Runner uses Eclipse Aether (https://wiki.eclipse.org/Aether) mechanism to download the Maven dependencies. Check their docs (https://wiki.eclipse.org/Aether) for more information.

Since the `StubRunnerRule` implements the `StubFinder` it allows you to find the started stubs:

GROOVY

```
package org.springframework.cloud.contract.stubrunner;

import java.net.URL;
import java.util.Collection;
import java.util.Map;

import org.springframework.cloud.contract.spec.Contract;

public interface StubFinder extends StubTrigger {
    /**
     * For the given groupId and artifactId tries to find the matching
     * URL of the running stub.
     *
     * @param groupId - might be null. In that case a search only via artifactId takes place
     * @return URL of a running stub or throws exception if not found
     */
    URL findStubUrl(String groupId, String artifactId) throws StubNotFoundException;

    /**
     * For the given Ivy notation {@code [groupId]:artifactId:[version]:[classifier]} tries to
     * find the matching URL of the running stub. You can also pass only {@code artifactId}.
     *
     * @param ivyNotation - Ivy representation of the Maven artifact
     * @return URL of a running stub or throws exception if not found
     */
    URL findStubUrl(String ivyNotation) throws StubNotFoundException;

    /**
     * Returns all running stubs
     */
    RunningStubs findAllRunningStubs();

    /**
     * Returns the list of Contracts
     */
    Map<StubConfiguration, Collection<Contract>> getContracts();
}
```

Example of usage in Spock tests:

```groovy
@ClassRule @Shared StubRunnerRule rule = new StubRunnerRule()
        .repoRoot(StubRunnerRuleSpec.getResource("/m2repo/repository").toURI().toString())
        .downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")
        .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")

def 'should start WireMock servers'() {
    expect: 'WireMocks are running'
        rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null
        rule.findStubUrl('loanIssuance') != null
        rule.findStubUrl('loanIssuance') == rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance')
        rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null
    and:
        rule.findAllRunningStubs().isPresent('loanIssuance')
        rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')
        rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')
    and: 'Stubs were registered'
        "${rule.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'
        "${rule.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
}
```

Example of usage in JUnit tests:

```java
@Test
public void should_start_wiremock_servers() throws Exception {
    // expect: 'WireMocks are running'
        then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")).isNotNull();
        then(rule.findStubUrl("loanIssuance")).isNotNull();

then(rule.findStubUrl("loanIssuance")).isEqualTo(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance"));
        then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")).isNotNull();
    // and:
        then(rule.findAllRunningStubs().isPresent("loanIssuance")).isTrue();
        then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.verifier.stubs",
"fraudDetectionServer")).isTrue();

then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")).isTr
ue();
    // and: 'Stubs were registered'
        then(httpGet(rule.findStubUrl("loanIssuance").toString() + "/name")).isEqualTo("loanIssuance");
        then(httpGet(rule.findStubUrl("fraudDetectionServer").toString() + "/name")).isEqualTo("fraudDetectionServer");
}
```

Check the **Common properties for JUnit and Spring** for more information on how to apply global configuration of Stub Runner.

To use the JUnit rule together with messaging you have to provide an implementation of the `MessageVerifier` interface to the rule builder (e.g. `rule.messageVerifier(new MyMessageVerifier())`). If you don't do this then whenever you try to send a message an exception will be thrown.

2.7.1. Maven settings

The stub downloader honors Maven settings for a different local repository folder. Authentication details for repositories and profiles are currently not taken into account, so you need to specify it using the properties mentioned above.

2.7.2. Providing fixed ports

You can also run your stubs on fixed ports. You can do it in two different ways. One is to pass it in the properties, and the other via fluent API of JUnit rule.

2.7.3. Fluent API

When using the `StubRunnerRule` you can add a stub to download and then pass the port for the last downloaded stub.

```java
@ClassRule public static StubRunnerRule rule = new StubRunnerRule()
        .repoRoot(repoRoot())
        .downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")
        .withPort(12345)
        .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer:12346");
```

You can see that for this example the following test is valid:

```java
then(rule.findStubUrl("loanIssuance")).isEqualTo(URI.create("http://localhost:12345").toURL());
then(rule.findStubUrl("fraudDetectionServer")).isEqualTo(URI.create("http://localhost:12346").toURL());
```

2.7.4. Stub Runner with Spring

Sets up Spring configuration of the Stub Runner project.

By providing a list of stubs inside your configuration file the Stub Runner automatically downloads and registers in WireMock the selected stubs.

If you want to find the URL of your stubbed dependency you can autowire the `StubFinder` interface and use its methods as presented below:

GROOVY

```
@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)
@SpringBootTest(properties = [" stubrunner.cloud.enabled=false",
        "stubrunner.camel.enabled=false",
        'foo=${stubrunner.runningstubs.fraudDetectionServer.port}'])
@AutoConfigureStubRunner
@DirtiesContext
@ActiveProfiles("test")
class StubRunnerConfigurationSpec extends Specification {

    @Autowired StubFinder stubFinder
    @Autowired Environment environment
    @Value('${foo}') Integer foo

    @BeforeClass
    @AfterClass
    void setupProps() {
        System.clearProperty("stubrunner.repository.root")
        System.clearProperty("stubrunner.classifier")
    }

    def 'should start WireMock servers'() {
        expect: 'WireMocks are running'
            stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null
            stubFinder.findStubUrl('loanIssuance') != null
            stubFinder.findStubUrl('loanIssuance') ==
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance')
            stubFinder.findStubUrl('loanIssuance') ==
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance')
            stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') ==
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs')
            stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null
        and:
            stubFinder.findAllRunningStubs().isPresent('loanIssuance')
            stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs',
'fraudDetectionServer')

stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')
        and: 'Stubs were registered'
            "${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'
            "${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
    }

    def 'should throw an exception when stub is not found'() {
        when:
            stubFinder.findStubUrl('nonExistingService')
        then:
            thrown(StubNotFoundException)
        when:
            stubFinder.findStubUrl('nonExistingGroupId', 'nonExistingArtifactId')
        then:
            thrown(StubNotFoundException)
    }

    def 'should register started servers as environment variables'() {
        expect:
            environment.getProperty("stubrunner.runningstubs.loanIssuance.port") != null
            stubFinder.findAllRunningStubs().getPort("loanIssuance") ==
(environment.getProperty("stubrunner.runningstubs.loanIssuance.port") as Integer)
        and:
            environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") != null
            stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") ==
(environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") as Integer)
    }

    def 'should be able to interpolate a running stub in the passed test property'() {
        given:
            int fraudPort = stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
        expect:
            fraudPort > 0
            environment.getProperty("foo", Integer) == fraudPort
            foo == fraudPort
    }

    @Configuration
    @EnableAutoConfiguration
    static class Config {}
}
```

for the following configuration file:

YML
```
stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids:
    - org.springframework.cloud.contract.verifier.stubs:loanIssuance
    - org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer
    - org.springframework.cloud.contract.verifier.stubs:bootService
  cloud:
    enabled: false
  camel:
    enabled: false

spring.cloud:
  consul.enabled: false
  service-registry.enabled: false
```

Instead of using the properties you can also use the properties inside the `@AutoConfigureStubRunner`. Below you can find an example of achieving the same result by setting values on the annotation.

GROOVY
```
@AutoConfigureStubRunner(
        ids = ["org.springframework.cloud.contract.verifier.stubs:loanIssuance",
        "org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer",
        "org.springframework.cloud.contract.verifier.stubs:bootService"],
        repositoryRoot = "classpath:m2repo/repository/")
```

Stub Runner Spring registers environment variables in the following manner for every registered WireMock server. Example for Stub Runner ids `com.example:foo`, `com.example:bar`.

- `stubrunner.runningstubs.foo.port`
- `stubrunner.runningstubs.bar.port`

Which you can reference in your code.

2.8. Stub Runner Spring Cloud

Stub Runner can integrate with Spring Cloud.

For real life examples you can check the

- producer app sample (https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/master/producer)
- consumer app sample (https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/master/consumer_with_discovery)

2.8.1. Stubbing Service Discovery

The most important feature of `Stub Runner Spring Cloud` is the fact that it's stubbing

- `DiscoveryClient`
- `Ribbon ServerList`

that means that regardless of the fact whether you're using Zookeeper, Consul, Eureka or anything else, you don't need that in your tests. We're starting WireMock instances of your dependencies and we're telling your application whenever you're using `Feign`, load balanced `RestTemplate` or `DiscoveryClient` directly, to call those stubbed servers instead of calling the real Service Discovery tool.

For example this test will pass

GROOVY
```
def 'should make service discovery work'() {
    expect: 'WireMocks are running'
        "${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'
        "${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
    and: 'Stubs can be reached via load service discovery'
        restTemplate.getForObject('http://loanIssuance/name', String) == 'loanIssuance'
        restTemplate.getForObject('http://someNameThatShouldMapFraudDetectionServer/name', String) ==
'fraudDetectionServer'
}
```

for the following configuration file

```yml
spring.cloud:
  zookeeper.enabled: false
  consul.enabled: false
eureka.client.enabled: false
stubrunner:
  camel.enabled: false
  idsToServiceIds:
    ivyNotation: someValueInsideYourCode
    fraudDetectionServer: someNameThatShouldMapFraudDetectionServer
```

Test profiles and service discovery

In your integration tests you typically don't want to call neither a discovery service (e.g. Eureka) or Config Server. That's why you create an additional test configuration in which you want to disable these features.

Due to certain limitations of `spring-cloud-commons` (https://github.com/spring-cloud/spring-cloud-commons/issues/156) to achieve this you have disable these properties via a static block like presented below (example for Eureka)

```java
//Hack to work around https://github.com/spring-cloud/spring-cloud-commons/issues/156
static {
    System.setProperty("eureka.client.enabled", "false");
    System.setProperty("spring.cloud.config.failFast", "false");
}
```

2.8.2. Additional Configuration

You can match the artifactId of the stub with the name of your app by using the `stubrunner.idsToServiceIds:` map. You can disable Stub Runner Ribbon support by providing: `stubrunner.cloud.ribbon.enabled` equal to `false` You can disable Stub Runner support by providing: `stubrunner.cloud.enabled` equal to `false`

By default all service discovery will be stubbed. That means that regardless of the fact if you have an existing `DiscoveryClient` its results will be ignored. However, if you want to reuse it, just set `stubrunner.cloud.delegate.enabled` to `true` and then your existing `DiscoveryClient` results will be merged with the stubbed ones.

2.9. Stub Runner Boot Application

Spring Cloud Contract Verifier Stub Runner Boot is a Spring Boot application that exposes REST endpoints to trigger the messaging labels and to access started WireMock servers.

One of the use-cases is to run some smoke (end to end) tests on a deployed application. You can read more about this in the "Microservice Deployment" article at Too Much Coding blog. (http://toomuchcoding.com/blog/2015/09/27/microservice-deployment/)

2.9.1. How to use it?

Just add the

```groovy
compile "org.springframework.cloud:spring-cloud-starter-stub-runner"
```

Annotate a class with `@EnableStubRunnerServer` , build a fat-jar and you're ready to go!

For the properties check the **Stub Runner Spring** section.

2.9.2. Endpoints

HTTP

- GET `/stubs` - returns a list of all running stubs in `ivy:integer` notation
- GET `/stubs/{ivy}` - returns a port for the given `ivy` notation (when calling the endpoint `ivy` can also be `artifactId` only)

Messaging

For Messaging

- GET `/triggers` - returns a list of all running labels in `ivy : [ label1, label2 …]` notation
- POST `/triggers/{label}` - executes a trigger with `label`

- POST `/triggers/{ivy}/{label}` - executes a trigger with `label` for the given `ivy` notation (when calling the endpoint `ivy` can also be `artifactId` only)

2.9.3. Example

GROOVY

```
@ContextConfiguration(classes = StubRunnerBoot, loader = SpringBootContextLoader)
@SpringBootTest(properties = "spring.cloud.zookeeper.enabled=false")
@ActiveProfiles("test")
class StubRunnerBootSpec extends Specification {

    @Autowired StubRunning stubRunning

    def setup() {
        RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
                new TriggerController(stubRunning))
    }

    def 'should return a list of running stub servers in "full ivy:port" notation'() {
        when:
            String response = RestAssuredMockMvc.get('/stubs').body.asString()
        then:
            def root = new JsonSlurper().parseText(response)
            root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs' instanceof Integer
    }

    def 'should return a port on which a [#stubId] stub is running'() {
        when:
            def response = RestAssuredMockMvc.get("/stubs/${stubId}")
        then:
            response.statusCode == 200
            response.body.as(Integer) > 0
        where:
            stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:+:stubs',
                       'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs',
                       'org.springframework.cloud.contract.verifier.stubs:bootService:+',
                       'org.springframework.cloud.contract.verifier.stubs:bootService',
                       'bootService']
    }

    def 'should return 404 when missing stub was called'() {
        when:
            def response = RestAssuredMockMvc.get("/stubs/a:b:c:d")
        then:
            response.statusCode == 404
    }

    def 'should return a list of messaging labels that can be triggered when version and classifier are passed'() {
        when:
            String response = RestAssuredMockMvc.get('/triggers').body.asString()
        then:
            def root = new JsonSlurper().parseText(response)
            root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-
SNAPSHOT:stubs'?.containsAll(["delete_book","return_book_1","return_book_2"])
    }

    def 'should trigger a messaging label'() {
        given:
            StubRunning stubRunning = Mock()
            RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning), new TriggerController(stubRunning))
        when:
            def response = RestAssuredMockMvc.post("/triggers/delete_book")
        then:
            response.statusCode == 200
        and:
            1 * stubRunning.trigger('delete_book')
    }

    def 'should trigger a messaging label for a stub with [#stubId] ivy notation'() {
        given:
            StubRunning stubRunning = Mock()
            RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning), new TriggerController(stubRunning))
        when:
            def response = RestAssuredMockMvc.post("/triggers/$stubId/delete_book")
        then:
            response.statusCode == 200
        and:
            1 * stubRunning.trigger(stubId, 'delete_book')
        where:
            stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService', 'bootService']
    }

    def 'should throw exception when trigger is missing'() {
        when:
            RestAssuredMockMvc.post("/triggers/missing_label")
        then:
            Exception e = thrown(Exception)
            e.message.contains("Exception occurred while trying to return [missing_label] label.")
            e.message.contains("Available labels are")
```

```
            e.message.contains("org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs=[]")
            e.message.contains("org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs=")
    }

}
```

2.9.4. Stub Runner Boot with Service Discovery

One of the possibilities of using Stub Runner Boot is to use it as a feed of stubs for "smoke-tests". What does it mean? Let's assume that you don't want to deploy 50 microservice to a test environment in order to check if your application is working fine. You've already executed a suite of tests during the build process but you would also like to ensure that the packaging of your application is fine. What you can do is to deploy your application to an environment, start it and run a couple of tests on it to see if it's working fine. We can call those tests smoke-tests since their idea is to check only a handful of testing scenarios.

The problem with this approach is such that if you're doing microservices most likely you're using a service discovery tool. Stub Runner Boot allows you to solve this issue by starting the required stubs and register them in a service discovery tool. Let's take a look at an example of such a setup with Eureka. Let's assume that Eureka was already running.

JAVA

```
@SpringBootApplication
@EnableStubRunnerServer
@EnableEurekaClient
@AutoConfigureStubRunner
public class StubRunnerBootEurekaExample {

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

}
```

As you can see we want to start a Stub Runner Boot server `@EnableStubRunnerServer` , enable Eureka client `@EnableEurekaClient` and we want to have the stub runner feature turned on `@AutoConfigureStubRunner` .

Now let's assume that we want to start this application so that the stubs get automatically registered. We can do it by running the app `java -jar ${SYSTEM_PROPS} stub-runner-boot-eureka-example.jar` where `${SYSTEM_PROPS}` would contain the following list of properties

BASH

```
-Dstubrunner.repositoryRoot=http://repo.spring.io/snapshots (1)
-Dstubrunner.cloud.stubbed.discovery.enabled=false (2)
-
Dstubrunner.ids=org.springframework.cloud.contract.verifier.stubs:loanIssuance,org.springframework.cloud.contract.verifie
r.stubs:fraudDetectionServer,org.springframework.cloud.contract.verifier.stubs:bootService (3)
-Dstubrunner.idsToServiceIds.fraudDetectionServer=someNameThatShouldMapFraudDetectionServer (4)

(1) - we tell Stub Runner where all the stubs reside
(2) - we don't want the default behaviour where the discovery service is stubbed. That's why the stub registration will
be picked
(3) - we provide a list of stubs to download
(4) - we provide a list of artifactId to serviceId mapping
```

That way your deployed application can send requests to started WireMock servers via the service discovery. Most likely points 1-3 could be set by default in `application.yml` cause they are not likely to change. That way you can provide only the list of stubs to download whenever you start the Stub Runner Boot.

2.10. Stubs Per Consumer

There are cases in which 2 consumers of the same endpoint want to have 2 different responses.

This approach also allows you to immediately know which consumer is using which part of your API. You can remove part of a response that your API produces and you can see which of your autogenerated tests fails. If none fails then you can safely delete that part of the response cause nobody is using it.

Let's look at the following example for contract defined for the producer called `producer` . There are 2 consumers: `foo-consumer` and `bar-consumer` .

Consumer `foo-service`

```
request {
   url '/foo'
   method GET()
}
response {
    status 200
    body(
       foo: "foo"
    }
}
```

Consumer `bar-service`

```
request {
   url '/foo'
   method GET()
}
response {
    status 200
    body(
       bar: "bar"
    }
}
```

You can't produce for the same request 2 different responses. That's why you can properly package the contracts and then profit from the `stubsPerConsumer` feature.

On the producer side the consumers can have a folder that contains contracts related only to them. By setting the `stubrunner.stubs-per-consumer` flag to `true` we no longer register all stubs but only those that correspond to the consumer application's name. In other words we'll scan the path of every stub and if it contains the subfolder with name of the consumer in the path only then will it get registered.

On the `foo` producer side the contracts would look like this

```
.
└── contracts
    ├── bar-consumer
    │   ├── bookReturnedForBar.groovy
    │   └── shouldCallBar.groovy
    └── foo-consumer
        ├── bookReturnedForFoo.groovy
        └── shouldCallFoo.groovy
```

Being the `bar-consumer` consumer you can either set the `spring.application.name` or the `stubrunner.consumer-name` to `bar-consumer` Or set the test as follows:

```
@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)
@SpringBootTest(properties = ["spring.application.name=bar-consumer"])
@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
        repositoryRoot = "classpath:m2repo/repository/",
        stubsPerConsumer = true)
@DirtiesContext
class StubRunnerStubsPerConsumerSpec extends Specification {
...
}
```

Then only the stubs registered under a path that contains the `bar-consumer` in its name (i.e. those from the `src/test/resources/contracts/bar-consumer/some/contracts/…` folder) will be allowed to be referenced.

Or set the consumer name explicitly

```
@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)
@SpringBootTest
@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
        repositoryRoot = "classpath:m2repo/repository/",
        consumerName = "foo-consumer",
        stubsPerConsumer = true)
@DirtiesContext
class StubRunnerStubsPerConsumerWithConsumerNameSpec extends Specification {
...
}
```

Then only the stubs registered under a path that contains the `foo-consumer` in its name (i.e. those from the `src/test/resources/contracts/foo-consumer/some/contracts/…` folder) will be allowed to be referenced.

You can check out issue 224 (https://github.com/spring-cloud/spring-cloud-contract/issues/224) for more information about the reasons behind this change.

2.11. Common

2.11.1. Common properties for JUnit and Spring

Some of the properties that are repetitive can be set using system properties or configuration properties (for Spring). Here are their names with their default values:

Property name	Default value	Description
stubrunner.minPort	10000	Minimal value of a port for a started WireMock with stubs
stubrunner.maxPort	15000	Minimal value of a port for a started WireMock with stubs
stubrunner.repositoryRoot		Maven repo url. If blank then will call the local maven repo
stubrunner.classifier	stubs	Default classifier for the stub artifacts
stubrunner.workOffline	false	If true then will not contact any remote repositories to download stubs
stubrunner.ids		Array of Ivy notation stubs to download
stubrunner.username		Optional username to access the tool that stores the JARs with stubs
stubrunner.password		Optional password to access the tool that stores the JARs with stubs
stubrunner.stubsPerConsumer	false	Set to `true` if you want to use different stubs per each consumer instead of registering all stubs for every consumer
stubrunner.consumerName		If you want to use stubs per consumer and want to override the consumer name just change this value

Stub runner stubs ids

You can provide the stubs to download via the `stubrunner.ids` system property. They follow the following pattern:

```
groupId:artifactId:version:classifier:port
```
JAVA

`version`, `classifier` and `port` are optional.

- If you don't provide the `port` then a random one will be picked
- If you don't provide the `classifier` then the default one will be taken. (NOTE that you can pass an empty classifier like this `groupId:artifactId:version:`)
- If you don't provide the `version` then the `+` will be passed and the latest one will be downloaded

Where `port` means the port of the WireMock server.

Starting from version 1.0.4 as a version you can provide a range of versions that you would like the Stub Runner to take into consideration. You can read more about the Aether versioning ranges here (https://wiki.eclipse.org/Aether/New_and_Noteworthy#Version_Ranges).

Taken from Aether Docs (http://download.eclipse.org/aether/aether-core/0.9.0/apidocs/org/eclipse/aether/util/version/GenericVersionScheme.html) :

> *This scheme accepts versions of any form, interpreting a version as a sequence of numeric and alphabetic segments. The characters '-', '_', and '.' as well as the mere transitions from digit to letter and vice versa delimit the version segments. Delimiters are treated as equivalent.*
>
> *Numeric segments are compared mathematically, alphabetic segments are compared lexicographically and case-insensitively. However, the following qualifier strings are recognized and treated specially: "alpha" = "a" < "beta" = "b" < "milestone" = "m" < "cr" = "rc" < "snapshot" < "final" = "ga" < "sp". All of those well-known qualifiers are considered smaller/older than other strings. An empty segment/string is equivalent to 0.*
>
> *In addition to the above mentioned qualifiers, the tokens "min" and "max" may be used as final version segment to denote the smallest/greatest version having a given prefix. For example, "1.2.min" denotes the smallest version in the 1.2 line, "1.2.max" denotes the greatest version in the 1.2 line. A version range of the form "[M.N.*]" is short for "[M.N.min, M.N.max]".*
>
> *Numbers and strings are considered incomparable against each other. Where version segments of different kind would collide, comparison will instead assume that the previous segments are padded with trailing 0 or "ga" segments, respectively, until the kind mismatch is resolved, e.g. "1-alpha" = "1.0.0-alpha" < "1.0.1-ga" = "1.0.1".*

2.12. Stub Runner for Messaging

Stub Runner has the functionality to run the published stubs in memory. It can integrate with the following frameworks out of the box

- Spring Integration
- Spring Cloud Stream
- Apache Camel
- Spring AMQP

It also provides points of entry to integrate with any other solution on the market.

2.12.1. Stub triggering

To trigger a message it's enough to use the `StubTrigger` interface:

```
package org.springframework.cloud.contract.stubrunner;

import java.util.Collection;
import java.util.Map;

public interface StubTrigger {

    /**
     * Triggers an event by a given label for a given {@code groupid:artifactid} notation. You can use only {@code
artifactId} too.
     *
     * Feature related to messaging.
     *
     * @return true - if managed to run a trigger
     */
    boolean trigger(String ivyNotation, String labelName);

    /**
     * Triggers an event by a given label.
     *
     * Feature related to messaging.
     *
     * @return true - if managed to run a trigger
     */
    boolean trigger(String labelName);

    /**
     * Triggers all possible events.
     *
     * Feature related to messaging.
     *
     * @return true - if managed to run a trigger
     */
    boolean trigger();

    /**
     * Returns a mapping of ivy notation of a dependency to all the labels it has.
     *
     * Feature related to messaging.
     */
    Map<String, Collection<String>> labels();
}
```
GROOVY

For convenience the `StubFinder` interface extends `StubTrigger` so it's enough to use only one in your tests.

`StubTrigger` gives you the following options to trigger a message:

Trigger by label

```
stubFinder.trigger('return_book_1')
```
GROOVY

Trigger by group and artifact ids

```
stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:camelService', 'return_book_1')
```
GROOVY

Trigger by artifact ids

```
stubFinder.trigger('camelService', 'return_book_1')
```
GROOVY

Trigger all messages

```
stubFinder.trigger()
```
GROOVY

2.13. Stub Runner Camel

Spring Cloud Contract Verifier Stub Runner's messaging module gives you an easy way to integrate with Apache Camel. For the provided artifacts it will automatically download the stubs and register the required routes.

2.13.1. Adding it to the project

It's enough to have both Apache Camel and Spring Cloud Contract Stub Runner on classpath. Remember to annotate your test class with `@AutoConfigureStubRunner` .

2.13.2. Examples

Stubs structure

Let us assume that we have the following Maven repository with a deployed stubs for the `camelService` application.

```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
                        └── camelService
                            ├── 0.0.1-SNAPSHOT
                            │   ├── camelService-0.0.1-SNAPSHOT.pom
                            │   ├── camelService-0.0.1-SNAPSHOT-stubs.jar
                            │   └── maven-metadata-local.xml
                            └── maven-metadata-local.xml
```

BASH

And the stubs contain the following structure:

```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

BASH

Let's consider the following contracts (let' number it with **1**):

```
Contract.make {
    label 'return_book_1'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('jms:output')
        body('''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

GROOVY

and number **2**

```
Contract.make {
    label 'return_book_2'
    input {
        messageFrom('jms:input')
        messageBody([
                bookName: 'foo'
        ])
        messageHeaders {
            header('sample', 'header')
        }
    }
    outputMessage {
        sentTo('jms:output')
        body([
                bookName: 'foo'
        ])
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

GROOVY

Scenario 1 (no input message)

So as to trigger a message via the `return_book_1` label we'll use the `StubTigger` interface as follows

```
stubFinder.trigger('return_book_1')
```

GROOVY

Next we'll want to listen to the output of the message sent to `jms:output`

```
Exchange receivedMessage = camelContext.createConsumerTemplate().receive('jms:output', 5000)
```
GROOVY

And the received message would pass the following assertions

```
receivedMessage != null
assertThatBodyContainsBookNameFoo(receivedMessage.in.body)
receivedMessage.in.headers.get('BOOK-NAME') == 'foo'
```
GROOVY

Scenario 2 (output triggered by input)

Since the route is set for you it's enough to just send a message to the `jms:output` destination.

```
camelContext.createProducerTemplate().sendBodyAndHeaders('jms:input', new BookReturned('foo'), [sample: 'header'])
```
GROOVY

Next we'll want to listen to the output of the message sent to `jms:output`

```
Exchange receivedMessage = camelContext.createConsumerTemplate().receive('jms:output', 5000)
```
GROOVY

And the received message would pass the following assertions

```
receivedMessage != null
assertThatBodyContainsBookNameFoo(receivedMessage.in.body)
receivedMessage.in.headers.get('BOOK-NAME') == 'foo'
```
GROOVY

Scenario 3 (input with no output)

Since the route is set for you it's enough to just send a message to the `jms:output` destination.

```
camelContext.createProducerTemplate().sendBodyAndHeaders('jms:delete', new BookReturned('foo'), [sample: 'header'])
```
GROOVY

2.14. Stub Runner Integration

Spring Cloud Contract Verifier Stub Runner's messaging module gives you an easy way to integrate with Spring Integration. For the provided artifacts it will automatically download the stubs and register the required routes.

2.14.1. Adding it to the project

It's enough to have both Spring Integration and Spring Cloud Contract Stub Runner on classpath. Remember to annotate your test class with `@AutoConfigureStubRunner` .

2.14.2. Examples

Stubs structure

Let us assume that we have the following Maven repository with a deployed stubs for the `integrationService` application.

```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
                        └── integrationService
                            ├── 0.0.1-SNAPSHOT
                            │   ├── integrationService-0.0.1-SNAPSHOT.pom
                            │   ├── integrationService-0.0.1-SNAPSHOT-stubs.jar
                            │   └── maven-metadata-local.xml
                            └── maven-metadata-local.xml
```
BASH

And the stubs contain the following structure:

```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

BASH

Let's consider the following contracts (let' number it with **1**):

```
Contract.make {
    label 'return_book_1'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('output')
        body('''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

GROOVY

and number **2**

```
Contract.make {
    label 'return_book_2'
    input {
        messageFrom('input')
        messageBody([
                bookName: 'foo'
        ])
        messageHeaders {
            header('sample', 'header')
        }
    }
    outputMessage {
        sentTo('output')
        body([
                bookName: 'foo'
        ])
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

GROOVY

and the following Spring Integration Route:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:beans="http://www.springframework.org/schema/beans"
             xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/spring-integration.xsd">


    <!-- REQUIRED FOR TESTING -->
    <bridge input-channel="output"
            output-channel="outputTest"/>

    <channel id="outputTest">
        <queue/>
    </channel>

</beans:beans>
```

XML

Scenario 1 (no input message)

So as to trigger a message via the `return_book_1` label we'll use the `StubTigger` interface as follows

```
stubFinder.trigger('return_book_1')
```

GROOVY

Next we'll want to listen to the output of the message sent to `output`

```
Message<?> receivedMessage = messaging.receive('outputTest')
```
GROOVY

And the received message would pass the following assertions

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```
GROOVY

Scenario 2 (output triggered by input)

Since the route is set for you it's enough to just send a message to the `output` destination.

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'input')
```
GROOVY

Next we'll want to listen to the output of the message sent to `output`

```
Message<?> receivedMessage = messaging.receive('outputTest')
```
GROOVY

And the received message would pass the following assertions

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```
GROOVY

Scenario 3 (input with no output)

Since the route is set for you it's enough to just send a message to the `input` destination.

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'delete')
```
GROOVY

2.15. Stub Runner Stream

Spring Cloud Contract Verifier Stub Runner's messaging module gives you an easy way to integrate with Spring Stream. For the provided artifacts it will automatically download the stubs and register the required routes.

In Stub Runner's integration with Stream the `messageFrom` or `sentTo` Strings are resolved first as a `destination` of a channel, and then if there is no such `destination` it's resolved as a channel name.

If you want to use Spring Cloud Stream remember to add a `org.springframework.cloud:spring-cloud-stream-test-support` dependency.

Maven

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```
XML

Gradle

```
testCompile "org.springframework.cloud:spring-cloud-stream-test-support"
```
GROOVY

2.15.1. Adding it to the project

It's enough to have both Spring Cloud Stream and Spring Cloud Contract Stub Runner on classpath. Remember to annotate your test class with `@AutoConfigureStubRunner` .

2.15.2. Examples

Stubs structure

Let us assume that we have the following Maven repository with a deployed stubs for the `streamService` application.

```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
                        └── streamService
                            ├── 0.0.1-SNAPSHOT
                            │   ├── streamService-0.0.1-SNAPSHOT.pom
                            │   ├── streamService-0.0.1-SNAPSHOT-stubs.jar
                            │   └── maven-metadata-local.xml
                            └── maven-metadata-local.xml
```

BASH

And the stubs contain the following structure:

```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

BASH

Let's consider the following contracts (let' number it with **1**):

```
Contract.make {
    label 'return_book_1'
    input { triggeredBy('bookReturnedTriggered()') }
    outputMessage {
        sentTo('returnBook')
        body('''{ "bookName" : "foo" }''')
        headers { header('BOOK-NAME', 'foo') }
    }
}
```

GROOVY

and number **2**

```
Contract.make {
    label 'return_book_2'
    input {
        messageFrom('bookStorage')
        messageBody([
            bookName: 'foo'
        ])
        messageHeaders { header('sample', 'header') }
    }
    outputMessage {
        sentTo('returnBook')
        body([
            bookName: 'foo'
        ])
        headers { header('BOOK-NAME', 'foo') }
    }
}
```

GROOVY

and the following Spring configuration:

```yaml
stubrunner.repositoryRoot: classpath:m2repo/repository/
stubrunner.ids: org.springframework.cloud.contract.verifier.stubs:streamService:0.0.1-SNAPSHOT:stubs

spring:
  cloud:
    stream:
      bindings:
        output:
          destination: returnBook
        input:
          destination: bookStorage

server:
  port: 0

debug: true
```

Scenario 1 (no input message)

So as to trigger a message via the `return_book_1` label we'll use the `StubTrigger` interface as follows

```groovy
stubFinder.trigger('return_book_1')
```

Next we'll want to listen to the output of the message sent to a channel whose `destination` is `returnBook`

```groovy
Message<?> receivedMessage = messaging.receive('returnBook')
```

And the received message would pass the following assertions

```groovy
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 2 (output triggered by input)

Since the route is set for you it's enough to just send a message to the `bookStorage destination`.

```groovy
messaging.send(new BookReturned('foo'), [sample: 'header'], 'bookStorage')
```

Next we'll want to listen to the output of the message sent to `returnBook`

```groovy
Message<?> receivedMessage = messaging.receive('returnBook')
```

And the received message would pass the following assertions

```groovy
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 3 (input with no output)

Since the route is set for you it's enough to just send a message to the `output` destination.

```groovy
messaging.send(new BookReturned('foo'), [sample: 'header'], 'delete')
```

2.16. Stub Runner Spring AMQP

Spring Cloud Contract Verifier Stub Runner's messaging module provides an easy way to integrate with Spring AMQP's Rabbit Template. For the provided artifacts it will automatically download the stubs and register the required routes.

The integration tries to work standalone, that is without interaction with a running RabbitMQ message broker. It expects a `RabbitTemplate` on the application context and uses it as a spring boot test `@SpyBean`. Thus it can use the mockito spy functionality to verify and introspect messages sent by the application.

On the message consumer side, it considers all `@RabbitListener` annotated endpoints as well as all `SimpleMessageListenerContainer`s on the application context.

As messages are usually sent to exchanges in AMQP the message contract contains the exchange name as the destination. Message listeners on the other side are bound to queues. Bindings connect an exchange to a queue. If message contracts are triggered the Spring AMQP stub runner integration will look for bindings on the application context that match this exchange. Then it collects the queues from the Spring exchanges and tries to find messages listeners bound to these queues. The message is triggered to all matching message listeners.

2.16.1. Adding it to the project

It's enough to have both Spring AMQP and Spring Cloud Contract Stub Runner on the classpath and set the property `stubrunner.amqp.enabled=true` . Remember to annotate your test class with `@AutoConfigureStubRunner` .

2.16.2. Examples

Stubs structure

Let us assume that we have the following Maven repository with a deployed stubs for the `spring-cloud-contract-amqp-test` application.

```
└── .m2
    └── repository
        └── com
            └── example
                └── spring-cloud-contract-amqp-test
                    ├── 0.4.0-SNAPSHOT
                    │   ├── spring-cloud-contract-amqp-test-0.4.0-SNAPSHOT.pom
                    │   ├── spring-cloud-contract-amqp-test-0.4.0-SNAPSHOT-stubs.jar
                    │   └── maven-metadata-local.xml
                    └── maven-metadata-local.xml
```

BASH

And the stubs contain the following structure:

```
├── META-INF
│   └── MANIFEST.MF
└── contracts
    └── shouldProduceValidPersonData.groovy
```

BASH

Let's consider the following contract:

```
Contract.make {
    // Human readable description
    description 'Should produce valid person data'
    // Label by means of which the output message can be triggered
    label 'contract-test.person.created.event'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('createPerson()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be sent
        sentTo 'contract-test.exchange'
        headers {
            header('contentType': 'application/json')
            header('__TypeId__': 'org.springframework.cloud.contract.stubrunner.messaging.amqp.Person')
        }
        // the body of the output message
        body ([
                id: $(consumer(9), producer(regex("[0-9]+"))),
                name: "me"
        ])
    }
}
```

GROOVY

and the following Spring configuration:

```
stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids: org.springframework.cloud.contract.verifier.stubs.amqp:spring-cloud-contract-amqp-test:0.4.0-SNAPSHOT:stubs
  amqp:
    enabled: true
server:
  port: 0
```

YAML

Triggering the message

So to trigger a message using the contract above we'll use the `StubTrigger` interface as follows.

```
stubTrigger.trigger("contract-test.person.created.event")
```
GROOVY

The message has the destination `contract-test.exchange` so the Spring AMQP stub runner integration looks for bindings related to this exchange.

```
@Bean
public Binding binding() {
    return BindingBuilder.bind(new Queue("test.queue")).to(new DirectExchange("contract-test.exchange")).with("#");
}
```
JAVA

The binding definition binds the queue `test.queue` . So the following listener definition is a match and is invoked with the contract message.

```
@Bean
public SimpleMessageListenerContainer simpleMessageListenerContainer(ConnectionFactory connectionFactory,
                                                                      MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames("test.queue");
    container.setMessageListener(listenerAdapter);

    return container;
}
```
JAVA

Also, the following annotated listener represents a match and would be invoked.

```
@RabbitListener(bindings = @QueueBinding(
        value = @Queue(value = "test.queue"),
        exchange = @Exchange(value = "contract-test.exchange", ignoreDeclarationExceptions = "true")))
public void handlePerson(Person person) {
    this.person = person;
}
```
JAVA

The message is directly handed over to the `onMessage` method of the `MessageListener` associated with the matching `SimpleMessageListenerContainer` .

Spring AMQP Test Configuration

In order to avoid that Spring AMQP is trying to connect to a running broker during our tests we configure a mock `ConnectionFactory` .

To disable the mocked ConnectionFactory set the property `stubrunner.amqp.mockConnection=false`

```
stubrunner:
  amqp:
    mockConnection: false
```
YAML

2.17. Contract DSL

Remember that inside the contract file you have to provide the fully qualified name to the `Contract` class and the `make` static import i.e. `org.springframework.cloud.spec.Contract.make { … }` . You can also provide an import to the `Contract` class `import org.springframework.cloud.spec.Contract` and then call `Contract.make { … }`

Contract DSL is written in Groovy, but don't be alarmed if you didn't use Groovy before. Knowledge of the language is not really needed as our DSL uses only a tiny subset of it (namely literals, method calls and closures). What's more the DSL is designed to be programmer-readable without any knowledge of the DSL itself - it's statically typed.

Spring Cloud Contract supports defining multiple contracts in a single file!

The Contract is present in the `spring-cloud-contract-spec` module of the Spring Cloud Contract Verifier repository.

Let's look at full example of a contract definition.

GROOVY

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/api/12'
        headers {
            header 'Content-Type': 'application/vnd.org.springframework.cloud.contract.verifier.twitter-places-
analyzer.v1+json'
        }
        body '''\
        [{
            "created_at": "Sat Jul 26 09:38:57 +0000 2014",
            "id": 492967299297845248,
            "id_str": "492967299297845248",
            "text": "Gonna see you at Warsaw",
            "place":
            {
                "attributes":{},
                "bounding_box":
                {
                    "coordinates":
                        [[
                            [-77.119759,38.791645],
                            [-76.909393,38.791645],
                            [-76.909393,38.995548],
                            [-77.119759,38.995548]
                        ]],
                    "type":"Polygon"
                },
                "country":"United States",
                "country_code":"US",
                "full_name":"Washington, DC",
                "id":"01fbe706f872cb32",
                "name":"Washington",
                "place_type":"city",
                "url": "http://api.twitter.com/1/geo/id/01fbe706f872cb32.json"
            }
        }]
    '''
    }
    response {
        status 200
    }
}
```

Not all features of the DSL are used in example above. If you didn't find what you are looking for, please check next paragraphs on this page.

> *You can easily compile Contracts to WireMock stubs mapping using standalone maven command:* `mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:convert`.

2.17.1. Limitations

- Spring Cloud Contract Verifier doesn't support XML properly. Please use JSON or help us implement this feature.
- The support for the verification of size of JSON arrays is experimental. If you want to turn it on please provide the value of a system property `spring.cloud.contract.verifier.assert.size` equal to `true`. By default this feature is set to `false`. You can also provide the `assertJsonSize` property in the plugin configuration.
- Due to the fact that JSON structure can have any form it's sometimes impossible to parse it properly when using the `value(consumer(…), producer(…))` notation when using that in GString. That's why we highly recommend using the Groovy Map notation.

2.17.2. Common Top-Level elements

Description

You can add a `description` to your contract that is nothing else but an arbitrary text. Example:

GROOVY
```
        org.springframework.cloud.contract.spec.Contract.make {
            description('''
given:
    An input
when:
    Sth happens
then:
    Output
''')
        }
```

Name

You can provide a name of your contract. Let's assume that you've provided a name `should register a user`. If you do this then the name of the autogenerated test will be equal to `validate_should_register_a_user`. Also the name of the stub will be `should_register_a_user.json` in case of a WireMock stub.

Please ensure that the name doesn't contain any characters that will make the generated test not possible to compile. Also remember that if you provide the same name for multiple contracts then your autogenerated tests will fail to compile and your generated stubs will override each other.

Ignoring contracts

If you want to ignore a contract you can either set a value of ignored contracts in the plugin configuration or just set the `ignored` property on the contract itself:

GROOVY
```
org.springframework.cloud.contract.spec.Contract.make {
    ignored()
}
```

2.17.3. HTTP Top-Level Elements

Following methods can be called in the top-level closure of a contract definition. Request and response are mandatory, priority is optional.

GROOVY
```
org.springframework.cloud.contract.spec.Contract.make {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    request {
        //...
    }

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    response {
        //...
    }

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    priority 1
}
```

2.17.4. Request

HTTP protocol requires only **method and address** to be specified in a request. The same information is mandatory in request definition of the Contract.

GROOVY
```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        // HTTP request method (GET/POST/PUT/DELETE).
        method 'GET'

        // Path component of request URL is specified as follows.
        urlPath('/users')
    }

    response {
        //...
    }
}
```

It is possible to specify whole `url` instead of just path, but `urlPath` is the recommended way as it makes the tests **host-independent**.

```
GROOVY
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'

        // Specifying `url` and `urlPath` in one contract is illegal.
        url('http://localhost:8888/users')
    }

    response {
        //...
    }
}
```

Request may contain **query parameters**, which are specified in a closure nested in a call to `urlPath` or `url`.

```
GROOVY
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...

        urlPath('/users') {

            // Each parameter is specified in form
            // `'paramName' : paramValue` where parameter value
            // may be a simple literal or one of matcher functions,
            // all of which are used in this example.
            queryParameters {

                // If a simple literal is used as value
                // default matcher function is used (equalTo)
                parameter 'limit': 100

                // `equalTo` function simply compares passed value
                // using identity operator (==).
                parameter 'filter': equalTo("email")

                // `containing` function matches strings
                // that contains passed substring.
                parameter 'gender': value(consumer(containing("[mf]")), producer('mf'))

                // `matching` function tests parameter
                // against passed regular expression.
                parameter 'offset': value(consumer(matching("[0-9]+")), producer(123))

                // `notMatching` functions tests if parameter
                // does not match passed regular expression.
                parameter 'loginStartsWith': value(consumer(notMatching(".{0,2}")), producer(3))
            }
        }

        //...
    }

    response {
        //...
    }
}
```

It may contain additional **request headers**...

```groovy
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...

        // Each header is added in form `'Header-Name' : 'Header-Value'`.
        // there are also some helper methods
        headers {
            header 'key': 'value'
            contentType(applicationJson())
        }

        //...
    }

    response {
        //...
    }
}
```

...and a **request body**.

```groovy
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        body '''{ "login" : "john", "name": "John The Contract" }'''
    }

    response {
        //...
    }
}
```

Request may contain **multipart** elements. Just call the `multipart()` method.

```groovy
org.springframework.cloud.contract.spec.Contract contractDsl = org.springframework.cloud.contract.spec.Contract.make {
    request {
        method "PUT"
        url "/multipart"
        headers {
            contentType('multipart/form-data;boundary=AaB03x')
        }
        multipart(
                // key (parameter name), value (parameter value) pair
                formParameter: $(c(regex('".+"')), p('"formParameterValue"')),
                someBooleanParameter: $(c(regex(anyBoolean())), p('true')),
                // a named parameter (e.g. with `file` name) that represents file with
                // `name` and `content`. You can also call `named("fileName", "fileContent")`
                file: named(
                        // name of the file
                        name: $(c(regex(nonEmpty())), p('filename.csv')),
                        // content of the file
                        content: $(c(regex(nonEmpty())), p('file content')))
        )
    }
    response {
        status 200
    }
}
```

In this example we defined parameters either directly by using the map notation, where the value can be a dynamic property (e.g. `formParameter: $(consumer(…), producer(…))`) or by using the `named(…)` method that allows you to set a named parameter. A named parameter can set a `name` and `content`. You can call it either via a method with 2 arguments: e.g. `named("fileName", "fileContent")` or via a map notation `named(name: "fileName", content: "fileContent")`.

From this contract the generated test will look more or less like this:

```java
// given:
 MockMvcRequestSpecification request = given()
   .header("Content-Type", "multipart/form-data;boundary=AaB03x")
   .param("formParameter", "\"formParameterValue\"")
   .param("someBooleanParameter", "true")
   .multiPart("file", "filename.csv", "file content".getBytes());

// when:
 ResponseOptions response = given().spec(request)
   .put("/multipart");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
```

The WireMock stub will look more or less like this:

```json
            '''
{
  "request" : {
    "url" : "/multipart",
    "method" : "PUT",
    "headers" : {
      "Content-Type" : {
        "matches" : "multipart/form-data;boundary=AaB03x.*"
      }
    },
    "bodyPatterns" : [ {
        "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data; name=\\"formParameter\\"\\r\\n(Content-Type:
.*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n\\".+\\"\\r\\n--\\\\1.*"
        }, {
                "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data; name=\\"someBooleanParameter\\"\\r\\n(Content-
Type: .*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n(true|false)\\r\\n--\\\\1.*"
        }, {
      "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data; name=\\"file\\"; filename=\\".+\\"\\r\\n(Content-Type:
.*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n.+\\r\\n--\\\\1.*"
    } ]
  },
  "response" : {
    "status" : 200,
    "transformers" : [ "response-template" ]
  }
}
    '''
```

2.17.5. Response

Minimal response must contain **HTTP status code**.

```groovy
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
    }
    response {
        // Status code sent by the server
        // in response to request specified above.
        status 200
    }
}
```

Besides status response may contain **headers** and **body**, which are specified the same way as in the request (see previous paragraph).

2.17.6. Dynamic properties

The contract can contain some dynamic properties - timestamps / ids etc. You don't want to enforce the consumers to stub their clocks to always return the same value of time so that it gets matched by the stub. That's why we allow you to provide the dynamic parts in your contracts in two ways. One is to pass them directly in the body and one to set them in a separate section called `testMatchers` and `stubMatchers`.

Dynamic properties inside the body

You can set the properties inside the body either via the `value` method

```
value(consumer(...), producer(...))
value(c(...), p(...))
value(stub(...), test(...))
value(client(...), server(...))
```
GROOVY

or if you're using the Groovy map notation for body you can use the `$()` method

```
$(consumer(...), producer(...))
$(c(...), p(...))
$(stub(...), test(...))
$(client(...), server(...))
```
GROOVY

All of the aforementioned approaches are equal. That means that `stub` and `client` methods are aliases over the `consumer` method. Let's take a closer look at what we can do with those values in the subsequent sections.

Regular expressions

You can use regular expressions to write your requests in Contract DSL. It is particularly useful when you want to indicate that a given response should be provided for requests that follow a given pattern. Also, you can use it when you need to use patterns and not exact values both for your test and your server side tests.

Please see the example below:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method('GET')
        url $(consumer(~/\/[0-9]{2}/), producer('/12'))
    }
    response {
        status 200
        body(
                id: $(anyNumber()),
                surname: $(
                        consumer('Kowalsky'),
                        producer(regex('[a-zA-Z]+'))
                ),
                name: 'Jan',
                created: $(consumer('2014-02-02 12:23:43'), producer(execute('currentDate(it)'))),
                correlationId: value(consumer('5d1f9fef-e0dc-4f3d-a7e4-72d2220dd827'),
                        producer(regex('[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'))
                )
        )
        headers {
            header 'Content-Type': 'text/plain'
        }
    }
}
```
GROOVY

You can also provide only one side of the communication using a regular expression. If you do that then automatically we'll provide the generated string that matches the provided regular expression. For example:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url value(consumer(regex('/foo/[0-9]{5}')))
        body([
            requestElement: $(consumer(regex('[0-9]{5}')))
        ])
        headers {
            header('header', $(consumer(regex('application\\/vnd\\.fraud\\.v1\\+json;.*'))))
        }
    }
    response {
        status 200
        body([
            responseElement: $(producer(regex('[0-9]{7}')))
        ])
        headers {
            contentType("application/vnd.fraud.v1+json")
        }
    }
}
```
GROOVY

In this example for request and response the opposite side of the communication will have the respective data generated.

Spring Cloud Contract comes with a series of predefined regular expressions that you can use in your contracts.

GROOVY

```
protected static final Pattern TRUE_OR_FALSE = Pattern.compile(/(true|false)/)
protected static final Pattern ONLY_ALPHA_UNICODE = Pattern.compile(/[\p{L}]*/)
protected static final Pattern NUMBER = Pattern.compile('-?\\d*(\\.\\d+)?')
protected static final Pattern IP_ADDRESS = Pattern.compile('([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-
4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])')
protected static final Pattern HOSTNAME_PATTERN = Pattern.compile('((http[s]?|ftp):/)/?([^:/\\s]+)(:[0-9]{1,5})?')
protected static final Pattern EMAIL = Pattern.compile('[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,6}')
protected static final Pattern URL = UrlHelper.URL
protected static final Pattern UUID = Pattern.compile('[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}')
protected static final Pattern ANY_DATE = Pattern.compile('(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])')
protected static final Pattern ANY_DATE_TIME = Pattern.compile('([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-
3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])')
protected static final Pattern ANY_TIME = Pattern.compile('(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])')
protected static final Pattern NON_EMPTY = Pattern.compile(/.+/)
protected static final Pattern NON_BLANK = Pattern.compile(/.*(\S+|\R).*|!^\R*$/)
protected static final Pattern ISO8601_WITH_OFFSET = Pattern.compile(/([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-
9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.\d{3})?(Z|[+-][01]\d:[0-5]\d)/)

protected static Pattern anyOf(String... values){
    return Pattern.compile(values.collect({"^$it\$"}).join("|"))
}

String onlyAlphaUnicode() {
    return ONLY_ALPHA_UNICODE.pattern()
}

String number() {
    return NUMBER.pattern()
}

String anyBoolean() {
    return TRUE_OR_FALSE.pattern()
}

String ipAddress() {
    return IP_ADDRESS.pattern()
}

String hostname() {
    return HOSTNAME_PATTERN.pattern()
}

String email() {
    return EMAIL.pattern()
}

String url() {
    return URL.pattern()
}

String uuid(){
    return UUID.pattern()
}

String isoDate() {
    return ANY_DATE.pattern()
}

String isoDateTime() {
    return ANY_DATE_TIME.pattern()
}

String isoTime() {
    return ANY_TIME.pattern()
}

String iso8601WithOffset() {
    return ISO8601_WITH_OFFSET.pattern()
}

String nonEmpty() {
    return NON_EMPTY.pattern()
}

String nonBlank() {
    return NON_BLANK.pattern()
}
```

so in your contract you can use it like this

```
Contract dslWithOptionalsInString = Contract.make {
    priority 1
    request {
        method POST()
        url '/users/password'
        headers {
            contentType(applicationJson())
        }
        body(
                email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),
                callback_url: $(consumer(regex(hostname())), producer('http://partners.com'))
        )
    }
    response {
        status 404
        headers {
            contentType(applicationJson())
        }
        body(
                code: value(consumer("123123"), producer(optional("123123"))),
                message: "User not found by email = [${value(producer(regex(email())),
consumer('not.existing@user.com'))}]"
        )
    }
}
```

GROOVY

Passing optional parameters

It is possible to provide optional parameters in your contract. It's only possible to have optional parameter for the:

- *STUB* side of the Request
- *TEST* side of the Response

Example:

```
org.springframework.cloud.contract.spec.Contract.make {
    priority 1
    request {
        method 'POST'
        url '/users/password'
        headers {
            contentType(applicationJson())
        }
        body(
                email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),
                callback_url: $(consumer(regex(hostname())), producer('http://partners.com'))
        )
    }
    response {
        status 404
        headers {
            header 'Content-Type': 'application/json'
        }
        body(
                code: value(consumer("123123"), producer(optional("123123")))
        )
    }
}
```

GROOVY

By wrapping a part of the body with the `optional()` method you are in fact creating a regular expression that should be present 0 or more times.

That way for the example above the following test would be generated if you pick Spock:

```
"""
 given:
  def request = given()
    .header("Content-Type", "application/json")
    .body('''{"email":"abc@abc.com","callback_url":"http://partners.com"}''')

 when:
  def response = given().spec(request)
    .post("/users/password")

 then:
  response.statusCode == 404
  response.header('Content-Type')  == 'application/json'
 and:
  DocumentContext parsedJson = JsonPath.parse(response.body.asString())
  assertThatJson(parsedJson).field("['code']").matches("(123123)?")
"""
```

GROOVY

and the following stub:

```
'''
{
  "request" : {
    "url" : "/users/password",
    "method" : "POST",
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['email'] =~ /([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\\\.[a-zA-Z]{2,6})?/)]"
    }, {
      "matchesJsonPath" : "$[?(@.['callback_url'] =~ /(((http[s]?|ftp):\\\\/)\\\\/?([^:\\\\/\\\\s]+)(:[0-9]{1,5})?/)]"
    } ],
    "headers" : {
      "Content-Type" : {
        "equalTo" : "application/json"
      }
    }
  },
  "response" : {
    "status" : 404,
    "body" : "{\\"code\\":\\"123123\\",\\"message\\":\\"User not found by email == [not.existing@user.com]\\"}",
    "headers" : {
      "Content-Type" : "application/json"
    }
  },
  "priority" : 1
}
'''
```

GROOVY

Executing custom methods on server side

It is also possible to define a method call to be executed on the server side during the test. Such a method can be added to the class defined as "baseClassForTests" in the configuration. Example:

Contract

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url $(consumer(regex('^/api/[0-9]{2}$')), producer('/api/12'))
        headers {
            header 'Content-Type': 'application/json'
        }
        body '''\
                [{
                    "text": "Gonna see you at Warsaw"
                }]
            '''
    }
    response {
        body (
                path: $(consumer('/api/12'), producer(regex('^/api/[0-9]{2}$'))),
                correlationId: $(consumer('1223456'), producer(execute('isProperCorrelationId($it)')))
        )
        status 200
    }
}
```

GROOVY

Base class

```
abstract class BaseMockMvcSpec extends Specification {

    def setup() {
        RestAssuredMockMvc.standaloneSetup(new PairIdController())
    }

    void isProperCorrelationId(Integer correlationId) {
        assert correlationId == 123456
    }

    void isEmpty(String value) {
        assert value == null
    }

}
```

GROOVY

You can't use both a String and `execute` to perform concatenation. E.g. calling `header('Authorization', 'Bearer ' + execute('authToken()'))` will lead to improper results. To make this work just call `header('Authorization', execute('authToken()'))` and ensure that the `authToken()` method returns everything that you need.

The type of the object read from the JSON can be one of the followings depending on the JSON path:

- `String` if you point to a `String` value in a JSON
- `JSONArray` if you point to a `List` in a JSON
- `Map` if you point to a `Map` in a JSON
- proper `Number` if you point to `Integer`, `Double` etc. in a JSON
- `Boolean` if you point to a `Boolean` in a JSON

In the request part of the contract you can specify that the `body` should be taken from a method.

You have to provide both the consumer and the producer side and the `execute` part can be applied for the whole body. Not for parts of it!

Example:

```
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        url '/something'
        body(
                $(c("foo"), p(execute("hashCode()")))
        )
    }
    response {
        status 200
    }
}
```

GROOVY

This will result in calling the `hashCode()` method in the request body. It would more or less like this:

```
// given:
 MockMvcRequestSpecification request = given()
   .body(hashCode());

// when:
 ResponseOptions response = given().spec(request)
   .get("/something");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
```

JAVA

Referencing request from response

The best situation is to provide fixed values but sometimes you need to reference a request in your response. In order to do this you can profit from the `fromRequest()` method that allows you to reference a bunch of elements from the HTTP request. You can use the following options:

- `fromRequest().url()` - return the request URL
- `fromRequest().query(String key)` - return the first query parameter with a given name
- `fromRequest().query(String key, int index)` - return the nth query parameter with a given name
- `fromRequest().header(String key)` - return the first header with a given name
- `fromRequest().header(String key, int index)` - return the nth header with a given name
- `fromRequest().body()` - return the full request body
- `fromRequest().body(String jsonPath)` - return the element from the request that matches the JSON Path

Let's take a look at the following contract

GROOVY

```
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        url('/api/v1/xxxx') {
            queryParameters {
                parameter("foo", "bar")
                parameter("foo", "bar2")
            }
        }
        headers {
            header(authorization(), "secret")
            header(authorization(), "secret2")
        }
        body(foo: "bar", baz: 5)
    }
    response {
        status 200
        headers {
            header(authorization(), "foo ${fromRequest().header(authorization())} bar")
        }
        body(
                url: fromRequest().url(),
                param: fromRequest().query("foo"),
                paramIndex: fromRequest().query("foo", 1),
                authorization: fromRequest().header("Authorization"),
                authorization2: fromRequest().header("Authorization", 1),
                fullBody: fromRequest().body(),
                responseFoo: fromRequest().body('$.foo'),
                responseBaz: fromRequest().body('$.baz'),
                responseBaz2: "Bla bla ${fromRequest().body('$.foo')} bla bla"
        )
    }
}
```

Running a JUnit test generation will lead in creation of a test looking more or less like this

JAVA

```
// given:
 MockMvcRequestSpecification request = given()
   .header("Authorization", "secret")
   .header("Authorization", "secret2")
   .body("{\"foo\":\"bar\",\"baz\":5}");

// when:
 ResponseOptions response = given().spec(request)
   .queryParam("foo","bar")
   .queryParam("foo","bar2")
   .get("/api/v1/xxxx");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
 assertThat(response.header("Authorization")).isEqualTo("foo secret bar");
// and:
 DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
 assertThatJson(parsedJson).field("url").isEqualTo("/api/v1/xxxx");
 assertThatJson(parsedJson).field("fullBody").isEqualTo("{\"foo\":\"bar\",\"baz\":5}");
 assertThatJson(parsedJson).field("paramIndex").isEqualTo("bar2");
 assertThatJson(parsedJson).field("responseFoo").isEqualTo("bar");
 assertThatJson(parsedJson).field("authorization2").isEqualTo("secret2");
 assertThatJson(parsedJson).field("responseBaz").isEqualTo(5);
 assertThatJson(parsedJson).field("responseBaz2").isEqualTo("Bla bla bar bla bla");
 assertThatJson(parsedJson).field("param").isEqualTo("bar");
 assertThatJson(parsedJson).field("authorization").isEqualTo("secret");
```

As you can see elements from the request have been properly referenced in the response.

The generated WireMock stub will look more or less like this:

```
JSON
{
  "request" : {
    "urlPath" : "/api/v1/xxxx",
    "method" : "POST",
    "headers" : {
      "Authorization" : {
        "equalTo" : "secret2"
      }
    },
    "queryParameters" : {
      "foo" : {
        "equalTo" : "bar2"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.baz == 5)]"
    }, {
      "matchesJsonPath" : "$[?(@.foo == 'bar')]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" : "{\"url\":\"{{{request.url}}}\",\"param\":\"{{{request.query.foo.[0]}}}\",\"paramIndex\":\"
{{{request.query.foo.[1]}}}\",\"authorization\":\"{{{request.headers.Authorization.[0]}}}\",\"authorization2\":\"
{{{request.headers.Authorization.[1]}}}\",\"fullBody\":\"{{{escapejsonbody}}}\",\"responseFoo\":\"{{{jsonpath this
'$.foo'}}}\",\"responseBaz\":{{{jsonpath this '$.baz'}}} ,\"responseBaz2\":\"Bla bla {{{jsonpath this '$.foo'}}} bla
bla\"}",
    "headers" : {
      "Authorization" : "{{{request.headers.Authorization.[0]}}}"
    },
    "transformers" : [ "response-template" ]
  }
}
```

So sending a request as the one presented in the `request` part of the contract will lead in sending the following response body

```
JSON
{
  "url" : "/api/v1/xxxx?foo=bar&foo=bar2",
  "param" : "bar",
  "paramIndex" : "bar2",
  "authorization" : "secret",
  "authorization2" : "secret2",
  "fullBody" : "{\"foo\":\"bar\",\"baz\":5}",
  "responseFoo" : "bar",
  "responseBaz" : 5,
  "responseBaz2" : "Bla bla bar bla bla"
}
```

This feature will work only with WireMock having version greater or equal to 2.5.1. We're using WireMock's `response-template` response transformer. It's using Handlebars to convert the Mustache `{{{ }}}` templates into proper values. Additionally we're registering 2 helper functions. `escapejsonbody` - that escapes the request body in a format that can be embedded in a JSON. Another is `jsonpath` that for a given parameter knows how to find an object in the request body.

Dynamic properties in matchers sections

If you've been working with Pact (https://docs.pact.io/) this might seem familiar. Quite a few users are used to having a separation between the body and setting dynamic parts of your contract.

That's why you can profit from two separate sections. One is called `stubMatchers` where you can define the dynamic values that should end up in a stub. You can set it in the `request` or `inputMessage` part of your contract. The other is called `testMatchers` which is present in the `response` or `outputMessage` side of the contract.

Currently we support only JSON Path based matchers with the following matching possibilities. For `stubMatchers` :

- `byEquality()` - the value taken from the response via the provided JSON Path needs to be equal to the provided value in the contract
- `byRegex(…)` - the value taken from the response via the provided JSON Path needs to match the regex
- `byDate()` - the value taken from the response via the provided JSON Path needs to match the regex for ISO Date
- `byTimestamp()` - the value taken from the response via the provided JSON Path needs to match the regex for ISO DateTime

- `byTime()` - the value taken from the response via the provided JSON Path needs to match the regex for ISO Time

For `testMatchers`:

- `byEquality()` - the value taken from the response via the provided JSON Path needs to be equal to the provided value in the contract
- `byRegex(…)` - the value taken from the response via the provided JSON Path needs to match the regex
- `byDate()` - the value taken from the response via the provided JSON Path needs to match the regex for ISO Date
- `byTimestamp()` - the value taken from the response via the provided JSON Path needs to match the regex for ISO DateTime
- `byTime()` - the value taken from the response via the provided JSON Path needs to match the regex for ISO Time
- `byType()` - the value taken from the response via the provided JSON Path needs to be of the same type as the type defined in the body of the response in the contract. `byType` can take a closure where you can set `minOccurrence` and `maxOccurrence`. That way you can assert on the size of the collection.
- `byCommand(…)` - the value taken from the response via the provided JSON Path will be passed as an input to the custom method that you're providing. E.g. `byCommand('foo($it)')` will result in calling a `foo` method to which the value matching the JSON Path will get passed.
  - The type of the object read from the JSON can be one of the followings depending on the JSON path:
    - `String` if you point to a `String` value in a JSON
    - `JSONArray` if you point to a `List` in a JSON
    - `Map` if you point to a `Map` in a JSON
    - proper `Number` if you point to `Integer`, `Double` etc. in a JSON
    - `Boolean` if you point to a `Boolean` in a JSON

Let's take a look at the following example:

GROOVY

```
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        urlPath '/get'
        body([
                duck: 123,
                alpha: "abc",
                number: 123,
                aBoolean: true,
                date: "2017-01-01",
                dateTime: "2017-01-01T01:23:45",
                time: "01:02:34",
                valueWithoutAMatcher: "foo",
                valueWithTypeMatch: "string",
                key: [
                        'complex.key' : 'foo'
                ]
        ])
        stubMatchers {
            jsonPath('$.duck', byRegex("[0-9]{3}"))
            jsonPath('$.duck', byEquality())
            jsonPath('$.alpha', byRegex(onlyAlphaUnicode()))
            jsonPath('$.alpha', byEquality())
            jsonPath('$.number', byRegex(number()))
            jsonPath('$.aBoolean', byRegex(anyBoolean()))
            jsonPath('$.date', byDate())
            jsonPath('$.dateTime', byTimestamp())
            jsonPath('$.time', byTime())
            jsonPath("\$.['key'].['complex.key']", byEquality())
        }
        headers {
            contentType(applicationJson())
        }
    }
    response {
        status 200
        body([
                duck: 123,
                alpha: "abc",
                number: 123,
                aBoolean: true,
                date: "2017-01-01",
                dateTime: "2017-01-01T01:23:45",
                time: "01:02:34",
                valueWithoutAMatcher: "foo",
                valueWithTypeMatch: "string",
                valueWithMin: [
                    1,2,3
                ],
                valueWithMax: [
                    1,2,3
                ],
                valueWithMinMax: [
                    1,2,3
                ],
                valueWithMinEmpty: [],
                valueWithMaxEmpty: [],
                key: [
                        'complex.key' : 'foo'
                ]
        ])
        testMatchers {
            // asserts the jsonpath value against manual regex
            jsonPath('$.duck', byRegex("[0-9]{3}"))
            // asserts the jsonpath value against the provided value
            jsonPath('$.duck', byEquality())
            // asserts the jsonpath value against some default regex
            jsonPath('$.alpha', byRegex(onlyAlphaUnicode()))
            jsonPath('$.alpha', byEquality())
            jsonPath('$.number', byRegex(number()))
            jsonPath('$.aBoolean', byRegex(anyBoolean()))
            // asserts vs inbuilt time related regex
            jsonPath('$.date', byDate())
            jsonPath('$.dateTime', byTimestamp())
            jsonPath('$.time', byTime())
            // asserts that the resulting type is the same as in response body
            jsonPath('$.valueWithTypeMatch', byType())
            jsonPath('$.valueWithMin', byType {
                // results in verification of size of array (min 1)
                minOccurrence(1)
            })
            jsonPath('$.valueWithMax', byType {
                // results in verification of size of array (max 3)
                maxOccurrence(3)
```

```
            })
            jsonPath('$.valueWithMinMax', byType {
                // results in verification of size of array (min 1 & max 3)
                minOccurrence(1)
                maxOccurrence(3)
            })
            jsonPath('$.valueWithMinEmpty', byType {
                // results in verification of size of array (min 0)
                minOccurrence(0)
            })
            jsonPath('$.valueWithMaxEmpty', byType {
                // results in verification of size of array (max 0)
                maxOccurrence(0)
            })
            // will execute a method `assertThatValueIsANumber`
            jsonPath('$.duck', byCommand('assertThatValueIsANumber($it)'))
            jsonPath("\$.['key'].['complex.key']", byEquality())
        }
        headers {
            contentType(applicationJson())
        }
    }
}
```

In this example we're providing the dynamic portions of the contract in the matchers sections. For the request part you can see that for all fields but `valueWithoutAMatcher` we're setting explicitly the values of regular expressions we'd like the stub to contain. For the `valueWithoutAMatcher` the verification will take place in the same way as without the usage of matchers - the test will perform an equality check in this case.

For the response side in the `testMatchers` section we're defining all the dynamic parts in a similar manner. The only difference is that we have the `byType` matchers too. In that case we're checking 4 fields in the way that we're verifying whether the response from the test has a value whose JSON path matching the given field is of the same type as the one defined in the response body and:

- for `$.valueWithTypeMatch` - we're just checking the whether the type is the same
- for `$.valueWithMin` - we're checking the type and assert if the size is greater or equal to the min occurrence
- for `$.valueWithMax` - we're checking the type and assert if the size is smaller or equal to the max occurrence
- for `$.valueWithMinMax` - we're checking the type and assert if the size is between the min and max occurrence

The resulting test would look more or less like this (note that we're separating the autogenerated assertions and the one from matchers with an `and` section):

JAVA

```
// given:
 MockMvcRequestSpecification request = given()
   .header("Content-Type", "application/json")
   .body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-
01-01T01:23:45\",\"time\":\"01:02:34\",\"valueWithoutAMatcher\":\"foo\",\"valueWithTypeMatch\":\"string\"}");

// when:
 ResponseOptions response = given().spec(request)
   .get("/get");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
 assertThat(response.header("Content-Type")).matches("application/json.*");
// and:
 DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
 assertThatJson(parsedJson).field("valueWithoutAMatcher").isEqualTo("foo");
// and:
 assertThat(parsedJson.read("$.duck", String.class)).matches("[0-9]{3}");
 assertThat(parsedJson.read("$.duck", Integer.class)).isEqualTo(123);
 assertThat(parsedJson.read("$.alpha", String.class)).matches("[\\p{L}]*");
 assertThat(parsedJson.read("$.alpha", String.class)).isEqualTo("abc");
 assertThat(parsedJson.read("$.number", String.class)).matches("-?\\d*(\\.\\d+)?");
 assertThat(parsedJson.read("$.aBoolean", String.class)).matches("(true|false)");
 assertThat(parsedJson.read("$.date", String.class)).matches("(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");
 assertThat(parsedJson.read("$.dateTime", String.class)).matches("([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-
9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
 assertThat(parsedJson.read("$.time", String.class)).matches("(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
 assertThat((Object) parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(java.lang.String.class);
 assertThat((Object) parsedJson.read("$.valueWithMin")).isInstanceOf(java.util.List.class);
 assertThat(parsedJson.read("$.valueWithMin", java.util.Collection.class)).hasSizeGreaterThanOrEqualTo(1);
 assertThat((Object) parsedJson.read("$.valueWithMax")).isInstanceOf(java.util.List.class);
 assertThat(parsedJson.read("$.valueWithMax", java.util.Collection.class)).hasSizeLessThanOrEqualTo(3);
 assertThat((Object) parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.util.List.class);
 assertThat(parsedJson.read("$.valueWithMinMax", java.util.Collection.class)).hasSizeBetween(1, 3);
 assertThat((Object) parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java.util.List.class);
 assertThat(parsedJson.read("$.valueWithMinEmpty", java.util.Collection.class)).hasSizeGreaterThanOrEqualTo(0);
 assertThat((Object) parsedJson.read("$.valueWithMaxEmpty")).isInstanceOf(java.util.List.class);
 assertThat(parsedJson.read("$.valueWithMaxEmpty", java.util.Collection.class)).hasSizeLessThanOrEqualTo(0);
 assertThatValueIsANumber(parsedJson.read("$.duck"));
```

and the WireMock stub like this:

JSON

```
                '''
{
  "request" : {
    "urlPath" : "/get",
    "method" : "POST",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/json.*"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['valueWithoutAMatcher'] == 'foo')]"
    }, {
      "matchesJsonPath" : "$[?(@.['valueWithTypeMatch'] == 'string')]"
    }, {
      "matchesJsonPath" : "$.['list'].['some'].['nested'][?(@.['anothervalue'] == 4)]"
    }, {
      "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['anothervalue'] == 4)]"
    }, {
      "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['json'] == 'with value')]"
    }, {
      "matchesJsonPath" : "$[?(@.duck =~ /([0-9]{3})/)]"
    }, {
      "matchesJsonPath" : "$[?(@.duck == 123)]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha =~ /([\\\\p{L}]*)/)]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha == 'abc')]"
    }, {
      "matchesJsonPath" : "$[?(@.number =~ /(-?\\\\d*(\\\\.\\\\d+)?)/)]"
    }, {
      "matchesJsonPath" : "$[?(@.aBoolean =~ /((true|false))/)]"
    }, {
      "matchesJsonPath" : "$[?(@.date =~ /((\\\\d\\\\d\\\\d\\\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01]))/)]"
    }, {
      "matchesJsonPath" : "$[?(@.dateTime =~ /(([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):
([0-5][0-9]):([0-5][0-9]))/)]"
    }, {
      "matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
    }, {
      "matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.*)/)]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" : "{\\"duck\\":123,\\"alpha\\":\\"abc\\",\\"number\\":123,\\"aBoolean\\":true,\\"date\\":\\"2017-01-
01\\",\\"dateTime\\":\\"2017-01-
01T01:23:45\\",\\"time\\":\\"01:02:34\\",\\"valueWithoutAMatcher\\":\\"foo\\",\\"valueWithTypeMatch\\":\\"string\\",\\"va
lueWithMin\\":[1,2,3],\\"valueWithMax\\":[1,2,3],\\"valueWithMinMax\\":[1,2,3]}",
    "headers" : {
      "Content-Type" : "application/json"
    }
  }
}
'''
```

> If you use a `matcher` then the part of the request / response that the `matcher` is addressing via the JSON Path will get removed from assertion. In case of verifying a collection you have to create matchers for **all** elements of the collection.

Let's look at the following example:

```groovy
Contract.make {
    request {
        method 'GET'
        url("/foo")
    }
    response {
        status 200
        body(events: [[
                                operation          : 'EXPORT',
                                eventId            : '16f1ed75-0bcc-4f0d-a04d-3121798faf99',
                                status             : 'OK'
                        ], [
                                operation          : 'INPUT_PROCESSING',
                                eventId            : '3bb4ac82-6652-462f-b6d1-75e424a0024a',
                                status             : 'OK'
                        ]
                ]
        )
        testMatchers {
            jsonPath('$.events[0].operation', byRegex('.+'))
            jsonPath('$.events[0].eventId', byRegex('^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-
F0-9]{12})$'))
            jsonPath('$.events[0].status', byRegex('.+'))
        }
    }
}
```

This will lead in creating the following test (showing just the assertion section)

```java
and:
    DocumentContext parsedJson = JsonPath.parse(response.body.asString())
    assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("16f1ed75-0bcc-4f0d-a04d-
3121798faf99")
    assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("EXPORT")
    assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("INPUT_PROCESSING")
    assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("3bb4ac82-6652-462f-b6d1-
75e424a0024a")
    assertThatJson(parsedJson).array("['events']").contains("['status']").isEqualTo("OK")
and:
    assertThat(parsedJson.read("\$.events[0].operation", String.class)).matches(".+")
    assertThat(parsedJson.read("\$.events[0].eventId", String.class)).matches("^([a-fA-F0-9]{8}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})\$")
    assertThat(parsedJson.read("\$.events[0].status", String.class)).matches(".+")
```

As you can see the assertion is malformed. That's because only the first element of the array got asserted. In order to fix this it's best to apply the assertion to the whole `$.events` collection and assert it via the `byCommand(…)` method.

2.17.7. JAX-RS support

We support JAX-RS 2 Client API. Base class needs to define `protected WebTarget webTarget` and server initialization, right now the only option how to test JAX-RS API is to start a web server.

Request with a body needs to have a content type set otherwise `application/octet-stream` is going to be used.

In order to use JAX-RS mode, use the following settings:

```groovy
testMode === 'JAXRSCLIENT'
```

Example of a test API generated:

```
'''
 // when:
  Response response = webTarget
    .path("/users")
    .queryParam("limit", "10")
    .queryParam("offset", "20")
    .queryParam("filter", "email")
    .queryParam("sort", "name")
    .queryParam("search", "55")
    .queryParam("age", "99")
    .queryParam("name", "Denis.Stepanov")
    .queryParam("email", "bob@email.com")
    .request()
    .method("GET");

  String responseAsString = response.readEntity(String.class);

 // then:
  assertThat(response.getStatus()).isEqualTo(200);
 // and:
  DocumentContext parsedJson = JsonPath.parse(responseAsString);
  assertThatJson(parsedJson).field("['property1']").isEqualTo("a");
'''
```

GROOVY

2.17.8. Async support

If you're using asynchronous communication on the server side (your controllers are returning `Callable`, `DeferredResult` etc. then inside your contract you have to provide in the `response` section a `async()` method. Example:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status 200
        body 'Passed'
        async()
    }
}
```

GROOVY

2.17.9. Working with Context Paths

Spring Cloud Contract supports context paths.

The only thing that changes in order to fully support context paths is the switch on the **PRODUCER** side. The autogenerated tests need to be using the **EXPLICIT** mode.

The consumer side remains untouched, in order for the generated test to pass you have to switch the **EXPLICIT** mode.

Maven

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <testMode>EXPLICIT</testMode>
    </configuration>
</plugin>
```

XML

Gradle

```
contracts {
        testMode = 'EXPLICIT'
}
```

GROOVY

That way you'll generate a test that **DOES NOT** use MockMvc. It means that you're generating real requests and you need to setup your generated test's base class to work on a real socket.

Let's imagine the following contract:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/my-context-path/url'
    }
    response {
        status 200
    }
}
```

GROOVY

Here is an example of how to set up a base class and Rest Assured for everything to work correctly.

```
import com.jayway.restassured.RestAssured;
import org.junit.Before;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = ContextPathTestingBaseClass.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class ContextPathTestingBaseClass {

    @LocalServerPort int port;

    @Before
    public void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = this.port;
    }
}
```

GROOVY

That way all:

- all your requests in the autogenerated tests will be sent to the real endpoint with your context path included (e.g. `/my-context-path/url`)
- your contracts reflect that you have a context path, thus your generated stubs will also have that information (e.g. in the stubs you'll see that you have too call `/my-context-path/url`)

2.17.10. Messaging Top-Level Elements

The DSL for messaging looks a little bit different than the one that focuses on HTTP.

Output triggered by a method

The output message can be triggered by calling a method (e.g. a Scheduler was started and a message was sent)

```
def dsl = Contract.make {
    // Human readable description
    description 'Some description'
    // Label by means of which the output message can be triggered
    label 'some_label'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('bookReturnedTriggered()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be sent
        sentTo('output')
        // the body of the output message
        body('''{ "bookName" : "foo" }''')
        // the headers of the output message
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

GROOVY

In this case the output message will be sent to `output` if a method called `bookReturnedTriggered` will be executed. In the message **publisher's** side we will generate a test that will call that method to trigger the message. On the **consumer** side you can use the `some_label` to trigger the message.

Output triggered by a message

The output message can be triggered by receiving a message.

GROOVY

```
def dsl = Contract.make {
    description 'Some Description'
    label 'some_label'
    // input is a message
    input {
        // the message was received from this destination
        messageFrom('input')
        // has the following body
        messageBody([
                bookName: 'foo'
        ])
        // and the following headers
        messageHeaders {
            header('sample', 'header')
        }
    }
    outputMessage {
        sentTo('output')
        body([
                bookName: 'foo'
        ])
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

In this case the output message will be sent to `output` if a proper message will be received on the `input` destination. In the message **publisher's** side we will generate a test that will send the input message to the defined destination. On the **consumer** side you can either send a message to the input destination or use the `some_label` to trigger the message.

Consumer / Producer

In HTTP you have a notion of `client` / `stub and `server` / `test` notation. You can use them also in messaging but we're providing also the `consumer` and `produer` methods as presented below (note you can use either `$` or `value` methods to provide `consumer` and `producer` parts)

GROOVY

```
Contract.make {
    label 'some_label'
    input {
        messageFrom value(consumer('jms:output'), producer('jms:input'))
        messageBody([
                bookName: 'foo'
        ])
        messageHeaders {
            header('sample', 'header')
        }
    }
    outputMessage {
        sentTo $(consumer('jms:input'), producer('jms:output'))
        body([
                bookName: 'foo'
        ])
    }
}
```

2.17.11. Multiple contracts in one file

It's possible to define multiple contracts in one file. An example of such a contract can look like this

```groovy
import org.springframework.cloud.contract.spec.Contract

[
        Contract.make {
            name("should post a user")
            request {
                method 'POST'
                url('/users/1')
            }
            response {
                status 200
            }
        },
        Contract.make {
            request {
                method 'POST'
                url('/users/2')
            }
            response {
                status 200
            }
        }
]
```

In this example one contract has the `name` field and the other doesn't. This will lead to generation of two tests that will look more or less like this:

```java
package org.springframework.cloud.contract.verifier.tests.com.hello;

import com.example.TestBase;
import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import com.jayway.restassured.module.mockmvc.specification.MockMvcRequestSpecification;
import com.jayway.restassured.response.ResponseOptions;
import org.junit.Test;

import static com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static org.assertj.core.api.Assertions.assertThat;

public class V1Test extends TestBase {

    @Test
    public void validate_should_post_a_user() throws Exception {
        // given:
            MockMvcRequestSpecification request = given();

        // when:
            ResponseOptions response = given().spec(request)
                    .post("/users/1");

        // then:
            assertThat(response.statusCode()).isEqualTo(200);
    }

    @Test
    public void validate_withList_1() throws Exception {
        // given:
            MockMvcRequestSpecification request = given();

        // when:
            ResponseOptions response = given().spec(request)
                    .post("/users/2");

        // then:
            assertThat(response.statusCode()).isEqualTo(200);
    }

}
```

Notice that for the contract that has the `name` field the generated test method is named `validate_should_post_a_user` . For the one that doesn't have the name it's called `validate_withList_1` . It corresponds to the name of the file `WithList.groovy` and the index of the contract in the list.

The generated stubs will look like this

```
should post a user.json
1_WithList.json
```

As you can see the first file got the `name` parameter from the contract. The second got the name of the contract file `WithList.groovy` prefixed with the index (in this case contract had index `1` in the list of contracts in the file).

As you can see it's much better if you name your contracts since then your tests are far more meaningful.

2.18. Customization

2.18.1. Extending the DSL

It is possible to provide your own functions to the DSL. The key requirement for this feature was to maintain the static compatibility. Below you will be able to see an example of:

- creation of a JAR with reusable classes
- referencing of these classes in the DSLs

The full example can be found here (https://github.com/spring-cloud-samples/spring-cloud-contract-samples).

Common JAR

Below you can find three classes that we will reuse in the DSLs.

PatternUtils contains functions used by both the **consumer** and the **producer**.

JAVA

```
package com.example;

import java.util.regex.Pattern;

/**
 * If you want to use {@link Pattern} directly in your tests
 * then you can create a class resembling this one. It can
 * contain all the {@link Pattern} you want to use in the DSL.
 *
 * <pre>
 * {@code
 * request {
 *     body(
 *         [ age: $(c(PatternUtils.oldEnough()))]
 *     )
 * }
 * </pre>
 *
 * Notice that we're using both {@code $()} for dynamic values
 * and {@code c()} for the consumer side.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class PatternUtils {

    public static String tooYoung() {
        //remove::start[]
        return "[0-1][0-9]";
        //remove::end[return]
    }

    public static Pattern oldEnough() {
        //remove::start[]
        return Pattern.compile("[2-9][0-9]");
        //remove::end[return]
    }

    /**
     * Makes little sense but it's just an example ;)
     */
    public static Pattern ok() {
        //remove::start[]
        return Pattern.compile("OK");
        //remove::end[return]
    }
}
//end::impl[]
```

ConsumerUtils contains functions used by the **consumer**.

JAVA

```
package com.example;

import org.springframework.cloud.contract.spec.internal.ClientDslProperty;

/**
 * DSL Properties passed to the DSL from the consumer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you can have a regular expression.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you have to have a concrete value.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ConsumerUtils {
    /**
     * Consumer side property. By using the {@link ClientDslProperty}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * request {
     *     body(
     *         [ age: $(ConsumerUtils.oldEnough())]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass to the consumer
     * and which one to the producer.
     *
     * @author Marcin Grzejszczak
     */
    public static ClientDslProperty oldEnough() {
        //remove::start[]
        // this example is not the best one and
        // theoretically you could just pass the regex instead of `ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ClientDslProperty(PatternUtils.oldEnough(), 40);
        //remove::end[return]
    }

}
//end::impl[]
```

ProducerUtils contains functions used by the **producer**.

JAVA
```
package com.example;

import org.springframework.cloud.contract.spec.internal.ServerDslProperty;

/**
 * DSL Properties passed to the DSL from the producer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you have to have a concrete value.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you can have a regular expression.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ProducerUtils {

    /**
     * Producer side property. By using the {@link ProducerUtils}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * response {
     *     body(
     *         [ status: $(ProducerUtils.ok())]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass to the consumer
     * and which one to the producer.
     */
    public static ServerDslProperty ok() {
        // this example is not the best one and
        // theoretically you could just pass the regex instead of `ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ServerDslProperty( PatternUtils.ok(), "OK");
    }
}
//end::impl[]
```

Adding the dependency to project

In order for the plugins and IDE to be able to reference the common JAR classes you need to pass the dependency to your project.

Test dependency in project's dependencies

First add the common jar dependency as a test dependency. That way since your contracts files are available at test resources path, automatically the common jar classes will be visible in your Groovy files.

Maven

XML
```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>beer-common</artifactId>
    <version>${project.version}</version>
    <scope>test</scope>
</dependency>
```

Gradle

GROOVY
```
testCompile("com.example:beer-common:0.0.1-SNAPSHOT")
```

Test dependency in plugin's dependencies

Now you have to add the dependency for the plugin to reuse at runtime.

Maven

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>com.example</packageWithBaseClasses>
        <baseClassMappings>
            <baseClassMapping>
                <contractPackageRegex>.*intoxication.*</contractPackageRegex>
                <baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>
            </baseClassMapping>
        </baseClassMappings>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>com.example</groupId>
            <artifactId>beer-common</artifactId>
            <version>${project.version}</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
</plugin>
```

Gradle

```groovy
classpath "com.example:beer-common:0.0.1-SNAPSHOT"
```

Referencing classes in DSLs

Now you can reference your classes in your DSL. Example:

```groovy
package contracts.beer.rest

import com.example.ConsumerUtils
import com.example.ProducerUtils
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description("""
Represents a successful scenario of getting a beer

```
given:
    client is old enough
when:
    he applies for a beer
then:
    we'll grant him the beer
```

""")
    request {
        method 'POST'
        url '/check'
        body(
                age: $(ConsumerUtils.oldEnough())
)
        headers {
            contentType(applicationJson())
        }
    }
    response {
        status 200
        body("""
            {
                "status": "${value(ProducerUtils.ok())}"
            }
            """)
        headers {
            contentType(applicationJson())
        }
    }
}
```

2.19. Pluggable architecture

There are cases where you have your contracts defined in other formats like YAML, RAML or PACT. On the other hand you'd like to profit from the test and stubs generation. It's really easy to add your own implementation of either of those. Also you can customize the way tests are generated (for example you can generate tests for other languages) and you can do the same for stubs

generation (you can generate stubs for other stub http server implementations).

2.19.1. Custom contract converter

Let's assume that your contract is written in a YAML file like this:

```yml
request:
  url: /foo
  method: PUT
  headers:
    foo: bar
  body:
    foo: bar
response:
  status: 200
  headers:
    foo2: bar
  body:
    foo2: bar
```

Thanks to the interface

```groovy
package org.springframework.cloud.contract.spec

/**
 * Converter to be used to convert FROM {@link File} TO {@link Contract}
 * and from {@link Contract} to {@code T}
 *
 * @param <T> - type to which we want to convert the contract
 *
 * @author Marcin Grzejszczak
 * @since 1.1.0
 */
interface ContractConverter<T> {

    /**
     * Should this file be accepted by the converter. Can use the file extension
     * to check if the conversion is possible.
     *
     * @param file - file to be considered for conversion
     * @return - {@code true} if the given implementation can convert the file
     */
    boolean isAccepted(File file)

    /**
     * Converts the given {@link File} to its {@link Contract} representation
     *
     * @param file - file to convert
     * @return - {@link Contract} representation of the file
     */
    Collection<Contract> convertFrom(File file)

    /**
     * Converts the given {@link Contract} to a {@link T} representation
     *
     * @param contract - the parsed contract
     * @return - {@link T} the type to which we do the conversion
     */
    T convertTo(Collection<Contract> contract)
}
```

you can register your own implementation of a contract structure converter. Your implementation needs to state the condition on which it should start the conversion. Also you have to define how to perform that conversion in both ways.

Once you create your implementation you have to create a `/META-INF/spring.factories` file in which you provide the fully qualified name of your implementation.

Example of a `spring.factories` file

```
# Converters
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.verifier.converter.YamlContractConverter
```

and the YAML implementation

GROOVY

```
package org.springframework.cloud.contract.verifier.converter

import groovy.transform.CompileStatic
import org.springframework.cloud.contract.spec.Contract
import org.springframework.cloud.contract.spec.ContractConverter
import org.springframework.cloud.contract.spec.internal.Headers
import org.yaml.snakeyaml.Yaml

/**
 * Simple converter from and to a {@link YamlContract} to a collection of {@link Contract}
 */
@CompileStatic
class YamlContractConverter implements ContractConverter<List<YamlContract>> {

    @Override
    public boolean isAccepted(File file) {
        String name = file.getName()
        return name.endsWith(".yml") || name.endsWith(".yaml")
    }

    @Override
    public Collection<Contract> convertFrom(File file) {
        try {
            YamlContract yamlContract = new Yaml().loadAs(new FileInputStream(file), YamlContract.class)
            return [Contract.make {
                request {
                    method(yamlContract?.request?.method)
                    url(yamlContract?.request?.url)
                    headers {
                        yamlContract?.request?.headers?.each { String key, Object value ->
                            header(key, value)
                        }
                    }
                    body(yamlContract?.request?.body)
                }
                response {
                    status(yamlContract?.response?.status)
                    headers {
                        yamlContract?.response?.headers?.each { String key, Object value ->
                            header(key, value)
                        }
                    }
                    body(yamlContract?.response?.body)
                }
            }]
        }
        catch (FileNotFoundException e) {
            throw new IllegalStateException(e)
        }
    }

    @Override
    public List<YamlContract> convertTo(Collection<Contract> contracts) {
        return contracts.collect { Contract contract ->
            YamlContract yamlContract = new YamlContract()
            yamlContract.request.with {
                method = contract?.request?.method?.clientValue
                url = contract?.request?.url?.clientValue
                headers = (contract?.request?.headers as Headers)?.asStubSideMap()
                body = contract?.request?.body?.clientValue as Map
            }
            yamlContract.response.with {
                status = contract?.response?.status?.clientValue as Integer
                headers = (contract?.response?.headers as Headers)?.asStubSideMap()
                body = contract?.response?.body?.clientValue as Map
            }
            return yamlContract
        }
    }
}
```

Pact converter

Spring Cloud Contract comes with an out of the box support for Pact (https://docs.pact.io/) representation of contracts. In other words instead of using the Groovy DSL you can use Pact files. In this section we will present how to add such a support for your project.

Pact contract

We will be working on the following example of a Pact contract. We've placed this file under the `src/test/resources/contracts` folder.

JAVASCRIPT

```
{
  "provider": {
    "name": "Provider"
  },
  "consumer": {
    "name": "Consumer"
  },
  "interactions": [
    {
      "description": "",
      "request": {
        "method": "PUT",
        "path": "/fraudcheck",
        "headers": {
          "Content-Type": "application/vnd.fraud.v1+json"
        },
        "body": {
          "clientId": "1234567890",
          "loanAmount": 99999
        },
        "matchingRules": {
          "$.body.clientId": {
            "match": "regex",
            "regex": "[0-9]{10}"
          }
        }
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/vnd.fraud.v1+json;charset=UTF-8"
        },
        "body": {
          "fraudCheckStatus": "FRAUD",
          "rejectionReason": "Amount too high"
        },
        "matchingRules": {
          "$.body.fraudCheckStatus": {
            "match": "regex",
            "regex": "FRAUD"
          }
        }
      }
    }
  ],
  "metadata": {
    "pact-specification": {
      "version": "2.0.0"
    },
    "pact-jvm": {
      "version": "2.4.18"
    }
  }
}
```

Pact for producers

On the producer side you have add to your plugin configuration two additional dependencies. One is the Spring Cloud Contract Pact support and the other represents the current Pact version that you're using.

Maven

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-contract-spec-pact</artifactId>
            <version>${spring-cloud-contract.version}</version>
        </dependency>
        <dependency>
            <groupId>au.com.dius</groupId>
            <artifactId>pact-jvm-model</artifactId>
            <version>2.4.18</version>
        </dependency>
    </dependencies>
</plugin>
```

Gradle

```groovy
classpath "org.springframework.cloud:spring-cloud-contract-spec-pact:${findProperty('verifierVersion') ?:
verifierVersion}"
classpath 'au.com.dius:pact-jvm-model:2.4.18'
```

When you execute the build of your application a test, looking more or less like this, will be generated

```java
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
        MockMvcRequestSpecification request = given()
                .header("Content-Type", "application/vnd.fraud.v1+json")
                .body("{\"clientId\":\"1234567890\",\"loanAmount\":99999}");

    // when:
        ResponseOptions response = given().spec(request)
                .put("/fraudcheck");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-Type")).isEqualTo("application/vnd.fraud.v1+json;charset=UTF-8");
    // and:
        DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("rejectionReason").isEqualTo("Amount too high");
    // and:
        assertThat(parsedJson.read("$.fraudCheckStatus", String.class)).matches("FRAUD");
}
```

and the stub looking like this

```javascript
{
  "uuid" : "996ae5ae-6834-4db6-8fac-358ca187ab62",
  "request" : {
    "url" : "/fraudcheck",
    "method" : "PUT",
    "headers" : {
      "Content-Type" : {
        "equalTo" : "application/vnd.fraud.v1+json"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.loanAmount == 99999)]"
    }, {
      "matchesJsonPath" : "$[?(@.clientId =~ /([0-9]{10})/)]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" : "{\"fraudCheckStatus\":\"FRAUD\",\"rejectionReason\":\"Amount too high\"}",
    "headers" : {
      "Content-Type" : "application/vnd.fraud.v1+json;charset=UTF-8"
    }
  }
}
```

Pact for consumers

On the producer side you have add to your project dependencies two additional dependencies. One is the Spring Cloud Contract Pact support and the other represents the current Pact version that you're using.

Maven

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-spec-pact</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>au.com.dius</groupId>
    <artifactId>pact-jvm-model</artifactId>
    <version>2.4.18</version>
    <scope>test</scope>
</dependency>
```

Gradle

```groovy
testCompile "org.springframework.cloud:spring-cloud-contract-spec-pact"
testCompile 'au.com.dius:pact-jvm-model:2.4.18'
```

2.19.2. Custom test generator

If you want to generate tests for different languages than Java or you're not happy with the way we're building Java tests for you then you can register your own implementation to do that.

Thanks to the interface

```groovy
package org.springframework.cloud.contract.verifier.builder

import org.springframework.cloud.contract.verifier.config.ContractVerifierConfigProperties
import org.springframework.cloud.contract.verifier.file.ContractMetadata
/**
 * Builds a single test.
 *
 * @since 1.1.0
 */
interface SingleTestGenerator {

    /**
     * Creates contents of a single test class in which all test scenarios from
     * the contract metadata should be placed.
     *
     * @param properties - properties passed to the plugin
     * @param listOfFiles - list of parsed contracts with additional metadata
     * @param className - the name of the generated test class
     * @param classPackage - the name of the package in which the test class should be stored
     * @param includedDirectoryRelativePath - relative path to the included directory
     * @return contents of a single test class
     */
    String buildClass(ContractVerifierConfigProperties properties, Collection<ContractMetadata> listOfFiles,
                      String className, String classPackage, String includedDirectoryRelativePath)

    /**
     * Extension that should be appended to the generated test class. E.g. {@code .java} or {@code .php}
     *
     * @param properties - properties passed to the plugin
     */
    String fileExtension(ContractVerifierConfigProperties properties)
}
```

you can register your own implementation that generates a test. Again, it's enough to provide a proper `spring.factories` file. Example:

```
org.springframework.cloud.contract.verifier.builder.SingleTestGenerator=/
com.example.MyGenerator
```

2.19.3. Custom stub generator

If you want to generate stubs for other stub server than WireMock it's enough to plug in your own implementation of this interface:

GROOVY

```
package org.springframework.cloud.contract.verifier.converter

import groovy.transform.CompileStatic
import org.springframework.cloud.contract.spec.Contract
import org.springframework.cloud.contract.verifier.file.ContractMetadata

/**
 * Converts contracts into their stub representation.
 *
 * @since 1.1.0
 */
@CompileStatic
interface StubGenerator {

    /**
     * Returns {@code true} if the converter can handle the file to convert it into a stub.
     */
    boolean canHandleFileName(String fileName)

    /**
     * Returns the collection of converted contracts into stubs. One contract can
     * result in multiple stubs.
     */
    Map<Contract, String> convertContents(String rootName, ContractMetadata content)

    /**
     * Returns the name of the converted stub file. If you have multiple contracts
     * in a single file then a prefix will be added to the generated file. If you
     * provide the {@link Contract#name} field then that field will override the
     * generated file name.
     *
     * Example: name of file with 2 contracts is {@code foo.groovy}, it will be
     * converted by the implementation to {@code foo.json}. The recursive file
     * converter will create two files {@code 0_foo.json} and {@code 1_foo.json}
     */
    String generateOutputFileNameForInput(String inputFileName)
}
```

you can register your own implementation that generate Stubs. Again, it's enough to provide a proper `spring.factories` file. Example:

```
# Stub converters
org.springframework.cloud.contract.verifier.converter.StubGenerator=\
org.springframework.cloud.contract.verifier.wiremock.DslToWireMockClientConverter
```

The default implementation is the WireMock stub generation.

You can provide multiple stub generator implementations. That way for example from a single DSL as input you can e.g. produce WireMock stubs and Pact files too!

2.19.4. Custom Stub Runner

If you decide to have a custom stub generation you also need a custom way of running stubs with your different stub provider.

Let us assume that you're using Moco (https://github.com/dreamhead/moco) to build your stubs. You wrote a proper stub generator and your stubs got placed in a JAR file.

In order for Stub Runner to know how to run your stubs you have to define a custom HTTP Stub server implementation. It can look like this:

GROOVY

```
package org.springframework.cloud.contract.stubrunner.provider.moco

import com.github.dreamhead.moco.bootstrap.arg.HttpArgs
import com.github.dreamhead.moco.runner.JsonRunner
import com.github.dreamhead.moco.runner.RunnerSetting
import groovy.util.logging.Slf4j
import org.springframework.cloud.contract.stubrunner.HttpServerStub
import org.springframework.util.SocketUtils

@Slf4j
class MocoHttpServerStub implements HttpServerStub {

    private boolean started
    private JsonRunner runner
    private int port

    @Override
    int port() {
        if (!isRunning()) {
            return -1
        }
        return port
    }

    @Override
    boolean isRunning() {
        return started
    }

    @Override
    HttpServerStub start() {
        return start(SocketUtils.findAvailableTcpPort())
    }

    @Override
    HttpServerStub start(int port) {
        this.port = port
        return this
    }

    @Override
    HttpServerStub stop() {
        if (!isRunning()) {
            return this
        }
        this.runner.stop()
        return this
    }

    @Override
    HttpServerStub registerMappings(Collection<File> stubFiles) {
        List<RunnerSetting> settings = stubFiles.findAll { it.name.endsWith("json") }
                .collect {
            log.info("Trying to parse [{}]", it.name)
            try {
                return RunnerSetting.aRunnerSetting().withStream(it.newInputStream()).build()
            } catch (Exception e) {
                log.warn("Exception occurred while trying to parse file [{}]", it.name, e)
                return null
            }
        }.findAll { it }
        this.runner = JsonRunner.newJsonRunnerWithSetting(settings,
                HttpArgs.httpArgs().withPort(this.port).build())
        this.runner.run()
        this.started = true
        return this
    }

    @Override
    boolean isAccepted(File file) {
        return file.name.endsWith(".json")
    }
}
```

and just register it in your `spring.factories` file

```
org.springframework.cloud.contract.stubrunner.HttpServerStub=\
org.springframework.cloud.contract.stubrunner.provider.moco.MocoHttpServerStub
```

that way you'll be able to run stubs using Moco.

If you don't provide any implementation then the default one - WireMock based will be picked. If you provide more than one then the first one on the list will be picked.

2.19.5. Custom Stub Downloader

You can customize the way your stubs are downloaded. It's enough to create an implementation of the `StubDownloaderBuilder`

```
JAVA
package com.example;

class CustomStubDownloaderBuilder implements StubDownloaderBuilder {

    @Override
    public StubDownloader build(final StubRunnerOptions stubRunnerOptions) {
        return new StubDownloader() {
            @Override
            public Map.Entry<StubConfiguration, File> downloadAndUnpackStubJar(
                    StubConfiguration config) {
                File unpackedStubs = retrieveStubs();
                return new AbstractMap.SimpleEntry<>(
                        new StubConfiguration(config.getGroupId(), config.getArtifactId(), version,
                                config.getClassifier()), unpackedStubs);
            }

            File retrieveStubs() {
                // here goes your custom logic to provide a folder where all the stubs reside
            }
}
```

and just register it in your `spring.factories` file

```
# Example of a custom Stub Downloader Provider
org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder=\
com.example.CustomStubDownloaderBuilder
```

that way you'll be able to pick a folder with the source of your stubs.

If you don't provide any implementation then the default one will be picked. If you provide `repositoryRoot` property or `workOffline` flag then Aether based that will download stubs from a remote repo will be picked. If you don't provide these values then the `ClasspathStubProvider` will be picked that will scan the classpath. If you provide more than one, then the first one on the list will be picked.

2.20. Links

Here you can find interesting links related to Spring Cloud Contract Verifier:

- Spring Cloud Contract Github Repository (https://github.com/spring-cloud/spring-cloud-contract/)
- Spring Cloud Contract Samples (https://github.com/spring-cloud-samples/spring-cloud-contract-samples/)
- Spring Cloud Contract Documentation (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html)
- Accurest Legacy Documentation (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html/deprecated)
- Spring Cloud Contract Stub Runner Documentation (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html/#spring-cloud-contract-stub-runner)
- Spring Cloud Contract Stub Runner Messaging Documentation (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html/#stub-runner-for-messaging)
- Spring Cloud Contract Gitter (https://gitter.im/spring-cloud/spring-cloud-contract)
- Spring Cloud Contract Maven Plugin (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract-maven-plugin/)
- Spring Cloud Contract WJUG Presentation by Marcin Grzejszczak (https://www.youtube.com/watch?v=sAAklvxmPmk)

3. Spring Cloud Contract WireMock

Modules giving you the possibility to use WireMock (http://wiremock.org) with different servers by using the "ambient" server embedded in a Spring Boot application. Check out the samples (https://github.com/spring-cloud/spring-cloud-contract/tree/1.0.x/samples) for more details.

The Spring Cloud Release Train BOM imports `spring-cloud-contract-dependencies` which in turn has exclusions for the dependencies needed by WireMock. This might lead to a situation that even if you're not using Spring Cloud Contract then your dependencies will be influenced anyways.

If you have a Spring Boot application that uses Tomcat as an embedded server, for example (the default with `spring-boot-starter-web`), then you can simply add `spring-cloud-contract-wiremock` to your classpath and add `@AutoConfigureWireMock` in order to be able to use Wiremock in your tests. Wiremock runs as a stub server and you can register stub behaviour using a Java API or via static JSON declarations as part of your test. Here's a simple example:

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 0)
public class WiremockForDocsTests {
    // A service that calls out over HTTP
    @Autowired private Service service;

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        stubFor(get(urlEqualTo("/resource"))
                .willReturn(aResponse().withHeader("Content-Type", "text/plain").withBody("Hello World!")));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }

}
```

To start the stub server on a different port use `@AutoConfigureWireMock(port=9999)` (for example), and for a random port use the value 0. The stub server port will be bindable in the test application context as "wiremock.server.port". Using `@AutoConfigureWireMock` adds a bean of type `WiremockConfiguration` to your test application context, where it will be cached in between methods and classes having the same context, just like for normal Spring integration tests.

3.1. Registering Stubs Automatically

If you use `@AutoConfigureWireMock` then it will register WireMock JSON stubs from the file system or classpath, by default from `file:src/test/resources/mappings`. You can customize the locations using the `stubs` attribute in the annotation, which can be a resource pattern (ant-style) or a directory, in which case `**/.json` is appended. Example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWireMock(stubs="classpath:/stubs")
public class WiremockImportApplicationTests {

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }

}
```


Actually WireMock always loads mappings from `src/test/resources/mappings` **as well as** the custom locations in the stubs attribute. To change this behaviour you have to also specify a files root as described next.

3.2. Using Files to Specify the Stub Bodies

WireMock can read response bodies from files on the classpath or file system. In that case you will see in the JSON DSL that the response has a "bodyFileName" instead of a (literal) "body". The files are resolved relative to a root directory `src/test/resources/__files` by default. To customize this location you can set the `files` attribute in the

`@AutoConfigureWireMock` annotation to the location of the parent directory (i.e. the place `__files` is a subdirectory). You can use Spring resource notation to refer to `file:…` or `classpath:…` locations (but generic URLs are not supported). A list of values can be given and WireMock will resolve the first file that exists when it needs to find a response body.

when you configure the `files` root, then it affects the automatic loading of stubs as well (they come from the root location in a subdirectory called "mappings"). The value of `files` has no effect on the stubs loaded explicitly from the `stubs` attribute.

3.3. Alternative: Using JUnit Rules

For a more conventional WireMock experience, using JUnit `@Rules` to start and stop the server, just use the `WireMockSpring` convenience class to obtain an `Options` instance:

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class WiremockForDocsClassRuleTests {

    // Start WireMock on some dynamic port
    // for some reason `dynamicPort()` is not working properly
    @ClassRule
    public static WireMockClassRule wiremock = new WireMockClassRule(
            WireMockSpring.options().dynamicPort());
    // A service that calls out over HTTP to localhost:${wiremock.port}
    @Autowired
    private Service service;

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        wiremock.stubFor(get(urlEqualTo("/resource"))
                .willReturn(aResponse().withHeader("Content-Type", "text/plain").withBody("Hello World!")));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }

}
```

The use `@ClassRule` means that the server will shut down after all the methods in this class.

4. Relaxed SSL Validation for Rest Template

WireMock allows you to stub a "secure" server with an "https" URL protocol. If your application wants to contact that stub server in an integration test, then it will find that the SSL certificates are not valid (it's the usual problem with self-installed certificates). The best option is often to just re-configure the client to use "http", but if that's not open to you then you can ask Spring to configure an HTTP client that ignores SSL validation errors (just for tests).

To make this work with minimum fuss you need to be using the Spring Boot `RestTemplateBuilder` in your app, e.g.

JAVA

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
```

This is because the builder is passed through callbacks to initalize it, so the SSL validation can be set up in the client at that point. This will happen automatically in your test if you are using the `@AutoConfigureWireMock` annotation (or the stub runner). If you are using the JUnit `@Rule` approach you need to add the `@AutoConfigureHttpClient` annotation as well:

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest("app.baseUrl=https://localhost:6443")
@AutoConfigureHttpClient
public class WiremockHttpsServerApplicationTests {

    @ClassRule
    public static WireMockClassRule wiremock = new WireMockClassRule(
            WireMockSpring.options().httpsPort(6443));
...
}
```

If you are using `spring-boot-starter-test` then you will have the Apache HTTP client on the classpath and it will be selected by the `RestTemplateBuilder` and configured to ignore SSL errors. If you are using the default `java.net` client you don't need the annotation (but it won't do any harm). There is no support currently for other clients, but it may be added in future releases.

5. WireMock and Spring MVC Mocks

Spring Cloud Contract provides a convenience class that can load JSON WireMock stubs into a Spring `MockRestServiceServer`. Here's an example:

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class WiremockForDocsMockServerApplicationTests {

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        // will read stubs classpath
        MockRestServiceServer server = WireMockRestServiceServer.with(this.restTemplate)
                .baseUrl("http://example.org").stubs("classpath:/stubs/resource.json")
                .build();
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World");
        server.verify();
    }
}
```

The `baseUrl` is prepended to all mock calls, and the `stubs()` method takes a stub path resource pattern as an argument. So in this example the stub defined at `/stubs/resource.json` is loaded into the mock server, so if the `RestTemplate` is asked to visit `http://example.org/` it will get the responses as declared there. More than one stub pattern can be specified, and each one can be a directory (for a recursive list of all ".json"), or a fixed filename (like in the example above) or an ant-style pattern. The JSON format is the normal WireMock format which you can read about in the WireMock website.

Currently we support Tomcat, Jetty and Undertow as Spring Boot embedded servers, and Wiremock itself has "native" support for a particular version of Jetty (currently 9.2). To use the native Jetty you need to add the native wiremock dependencies and exclude the Spring Boot container if there is one.

6. Generating Stubs using RestDocs

Spring RestDocs (https://projects.spring.io/spring-restdocs) can be used to generate documentation (e.g. in asciidoctor format) for an HTTP API with Spring MockMvc or Rest Assured. At the same time as you generate documentation for your API, you can also generate WireMock stubs, by using Spring Cloud Contract WireMock. Just write your normal RestDocs test cases and use `@AutoConfigureRestDocs` to have stubs automatically in the restdocs output directory. For example:

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(get("/resource"))
                .andExpect(content().string("Hello World"))
                .andDo(document("resource"));
    }
}
```

From this test will be generated a WireMock stub at "target/snippets/stubs/resource.json". It matches all GET requests to the "/resource" path.

Without any additional configuration this will create a stub with a request matcher for the HTTP method and all headers except "host" and "content-length". To match the request more precisely, for example to match the body of a POST or PUT, we need to explicitly create a request matcher. This will do two things: 1) create a stub that only matches the way you specify, 2) assert that the request in the test case also matches the same conditions.

The main entry point for this is `WireMockRestDocs.verify()` which can be used as a substitute for the `document()` convenience method. For example:

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(post("/resource")
                .content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))
                .andExpect(status().isOk())
                .andDo(verify().jsonPath("$.id")
                        .stub("resource"));
    }
}
```

So this contract is saying: any valid POST with an "id" field will get back an the same response as in this test. You can chain together calls to `.jsonPath()` to add additional matchers. The JayWay documentation (https://github.com/jayway/JsonPath) can help you to get up to speed with JSON Path if it is unfamiliar to you.

Instead of the `jsonPath` and `contentType` convenience methods, you can also use the WireMock APIs to verify the request matches the created stub. Example:

```java
@Test
public void contextLoads() throws Exception {
    mockMvc.perform(post("/resource")
               .content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))
            .andExpect(status().isOk())
            .andDo(verify()
                    .wiremock(WireMock.post(
                        urlPathEquals("/resource"))
                        .withRequestBody(matchingJsonPath("$.id"))
                       .stub("post-resource"));
}
```

The WireMock API is rich - you can match headers, query parameters, and request body by regex as well as by json path - so this can useful to create stubs with a wider range of parameters. The above example will generate a stub something like this:

post-resource.json

```json
{
  "request" : {
    "url" : "/resource",
    "method" : "POST",
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.id"
    }]
  },
  "response" : {
    "status" : 200,
    "body" : "Hello World",
    "headers" : {
      "X-Application-Context" : "application:-1",
      "Content-Type" : "text/plain"
    }
  }
}
```

You can use either the `wiremock()` method or the `jsonPath()` and `contentType()` methods to create request matchers, but not both.

On the consumer side, you can make the `resource.json` generated above available on the classpath (by publishing stubs as JARs (https://cloud.spring.io/spring-cloud-contract/spring-cloud-contract.html#_publishing_stubs_as_jars) for example). After that, you can create a stub using WireMock in a number of different ways, including as described above using `@AutoConfigureWireMock(stubs="classpath:resource.json")`.

7. Generating Contracts using RestDocs

Another thing that can be generated with Spring RestDocs is the Spring Cloud Contract DSL file and documentation. If you combine that with Spring Cloud WireMock then you're getting both the contracts and stubs.

Why would you want to use this feature? Some people in the community asked questions about situation in which they would like to move to DSL based contract definition but they already have a lot of Spring MVC tests. Using this feature allows you to generate the contract files that you can later modify and move to proper folders so that the plugin picks them up.

> You might wonder why this functionality is in the WireMock module. Come to think of it, it does make sense since it makes little sense to generate only contracts and not generate the stubs. That's why we suggest to do both.

Let's imagine the following test:

JAVA

```
        this.mockMvc.perform(post("/foo")
                    .accept(MediaType.APPLICATION_PDF)
                    .accept(MediaType.APPLICATION_JSON)
                    .contentType(MediaType.APPLICATION_JSON)
                    .content("{\"foo\": 23 }"))
                .andExpect(status().isOk())
                .andExpect(content().string("bar"))
                // first WireMock
                .andDo(WireMockRestDocs.verify()
                        .jsonPath("$[?(@.foo >= 20)]")
                        .contentType(MediaType.valueOf("application/json"))
                        .stub("shouldGrantABeerIfOldEnough"))
                // then Contract DSL documentation
                .andDo(document("index", SpringCloudContractRestDocs.dslContract()));
```

This will lead in the creation of the stub as presented in the previous section, contract will get generated and a documentation file too.

The contract will be called `index.groovy` and look more like this.

GROOVY

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method 'POST'
        url '/foo'
        body('''
            {"foo": 23 }
        ''')
        headers {
            header('''Accept''', '''application/json''')
            header('''Content-Type''', '''application/json''')
        }
    }
    response {
        status 200
        body('''
        bar
        ''')
        headers {
            header('''Content-Type''', '''application/json;charset=UTF-8''')
            header('''Content-Length''', '''3''')
        }
        testMatchers {
            jsonPath('$[?(@.foo >= 20)]', byType())
        }
    }
}
```

the generated document (example for Asciidoc) will contain a formatted contract (the location of this file would be `index/dsl-contract.adoc`).

Spring Cloud Vault

Spring Cloud Vault Config provides client-side support for externalized configuration in a distributed system. With HashiCorp's Vault (https://www.vaultproject.io) you have a central place to manage external secret properties for applications across all environments. Vault can manage static and dynamic secrets such as username/password for remote applications/resources and provide credentials for external services such as MySQL, PostgreSQL, Apache Cassandra, MongoDB, Consul, AWS and more.

8. Quick Start

Prerequisites

To get started with Vault and this guide you need a *NIX-like operating systems that provides:

- `wget`, `openssl` and `unzip`
- at least Java 7 and a properly configured `JAVA_HOME` environment variable

Install Vault

```
$ src/test/bash/install_vault.sh
```
BASH

Create SSL certificates for Vault

```
$ src/test/bash/create_certificates.sh
```
BASH

`create_certificates.sh` creates certificates in `work/ca` and a JKS truststore `work/keystore.jks`. If you want to run Spring Cloud Vault using this quickstart guide you need to configure the truststore the `spring.cloud.vault.ssl.trust-store` property to `file:work/keystore.jks`.

Start Vault server

```
$ src/test/bash/local_run_vault.sh
```
BASH

Vault is started listening on `0.0.0.0:8200` using the `inmem` storage and `https`. Vault is sealed and not initialized when starting up.

If you want to run tests, leave Vault uninitialized. The tests will initialize Vault and create a root token `00000000-0000-0000-0000-000000000000`.

If you want to use Vault for your application or give it a try then you need to initialize it first.

```
$ export VAULT_ADDR="https://localhost:8200"
$ export VAULT_SKIP_VERIFY=true # Don't do this for production
$ vault init
```
BASH

You should see something like:

```
Key 1: 7149c6a2e16b8833f6eb1e76df03e47f6113a3288b3093faf5033d44f0e70fe701
Key 2: 901c534c7988c18c20435a85213c683bdcf0efcd82e38e2893779f152978c18c02
Key 3: 03ff3948575b1165a20c20ee7c3e6edf04f4cdbe0e82dbff5be49c63f98bc03a03
Key 4: 216ae5cc3ddaf93ceb8e1d15bb9fc3176653f5b738f5f3d1ee00cd7dccbe926e04
Key 5: b2898fc8130929d569c1677ee69dc5f3be57d7c4b494a6062693ce0b1c4d93d805
Initial Root Token: 19aefa97-cccc-bbbb-aaaa-225940e63d76

Vault initialized with 5 keys and a key threshold of 3. Please
securely distribute the above keys. When the Vault is re-sealed,
restarted, or stopped, you must provide at least 3 of these keys
to unseal it again.

Vault does not store the master key. Without at least 3 keys,
your Vault will remain permanently sealed.
```
BASH

Vault will initialize and return a set of unsealing keys and the root token. Pick 3 keys and unseal Vault. Store the Vault token in the `VAULT_TOKEN` environment variable.

BASH

```
$ vault unseal (Key 1)
$ vault unseal (Key 2)
$ vault unseal (Key 3)
$ export VAULT_TOKEN=(Root token)
# Required to run Spring Cloud Vault tests after manual initialization
$ vault token-create -id="00000000-0000-0000-0000-000000000000" -policy="root"
```

Spring Cloud Vault accesses different resources. By default, the secret backend is enabled which accesses secret config settings via JSON endpoints.

The HTTP service has resources in the form:

```
/secret/{application}/{profile}
/secret/{application}
/secret/{defaultContext}/{profile}
/secret/{defaultContext}
```

where the "application" is injected as the `spring.application.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). Properties retrieved from Vault will be used "as-is" without further prefixing of the property names.

9. Client Side Usage

To use these features in an application, just build it as a Spring Boot application that depends on `spring-cloud-vault-config` (e.g. see the test cases). Example Maven configuration:

Example 1. pom.xml

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.4.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-vault-config</artifactId>
        <version>Dalston.SR3</version>
    </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:

```java
@SpringBootApplication
@RestController
public class Application {

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

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

When it runs it will pick up the external configuration from the default local Vault server on port `8200` if it is running. To modify the startup behavior you can change the location of the Vault server using `bootstrap.properties` (like `application.properties` but for the bootstrap phase of an application context), e.g.

Example 2. bootstrap.yml

```yaml
spring.cloud.vault:
    host: localhost
    port: 8200
    scheme: https
    connection-timeout: 5000
    read-timeout: 15000
    config:
        order: -10
```

- `host` sets the hostname of the Vault host. The host name will be used for SSL certificate validation
- `port` sets the Vault port
- `scheme` setting the scheme to `http` will use plain HTTP. Supported schemes are `http` and `https`.
- `connection-timeout` sets the connection timeout in milliseconds
- `read-timeout` sets the read timeout in milliseconds
- `config.order` sets the order for the property source

Enabling further integrations requires additional dependencies and configuration. Depending on how you have set up Vault you might need additional configuration like SSL (https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html#vault.config.ssl) and authentication (https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html#vault.config.authentication).

If the application imports the `spring-boot-starter-actuator` project, the status of the vault server will be available via the `/health` endpoint.

The vault health indicator can be enabled or disabled through the property `health.vault.enabled` (default `true`).

9.1. Authentication

Vault requires an authentication mechanism (https://www.vaultproject.io/docs/concepts/auth.html) to authorize client requests (https://www.vaultproject.io/docs/concepts/tokens.html).

Spring Cloud Vault supports multiple authentication mechanisms (https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html#vault.config.authentication) to authenticate applications with Vault.

For a quickstart, use the root token printed by the Vault initialization.

Example 3. bootstrap.yml

```yaml
spring.cloud.vault:
    token: 19aefa97-cccc-bbbb-aaaa-225940e63d76
```

Consider carefully your security requirements. Static token authentication is fine if you want quickly get started with Vault, but a static token is not protected any further. Any disclosure to unintended parties allows Vault use with the associated token roles.

10. Authentication methods

Different organizations have different requirements for security and authentication. Vault reflects that need by shipping multiple authentication methods. Spring Cloud Vault supports token and AppId authentication.

10.1. Token authentication

Tokens are the core method for authentication within Vault. Token authentication requires a static token to be provided using the Bootstrap Application Context (https://github.com/spring-cloud/spring-cloud-commons/blob/master/docs/src/main/asciidoc/spring-cloud-commons.adoc#the-bootstrap-application-context)
.

Token authentication is the default authentication method. If a token is disclosed an unintended party gains access to Vault and can access secrets for the intended client.

Example 4. bootstrap.yml

```
YAML
spring.cloud.vault:
    authentication: TOKEN
    token: 00000000-0000-0000-0000-000000000000
```

- `authentication` setting this value to `TOKEN` selects the Token authentication method
- `token` sets the static token to use

See also: Vault Documentation: Tokens (https://www.vaultproject.io/docs/concepts/tokens.html)

10.2. AppId authentication

Vault supports AppId (https://www.vaultproject.io/docs/auth/app-id.html) authentication that consists of two hard to guess tokens. The AppId defaults to `spring.application.name` that is statically configured. The second token is the UserId which is a part determined by the application, usually related to the runtime environment. IP address, Mac address or a Docker container name are good examples. Spring Cloud Vault Config supports IP address, Mac address and static UserId's (e.g. supplied via System properties). The IP and Mac address are represented as Hex-encoded SHA256 hash.

IP address-based UserId's use the local host's IP address.

Example 5. bootstrap.yml using SHA256 IP-Address UserId's

```
YAML
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: IP_ADDRESS
```

- `authentication` setting this value to `APPID` selects the AppId authentication method
- `app-id-path` sets the path of the AppId mount to use
- `user-id` sets the UserId method. Possible values are `IP_ADDRESS`, `MAC_ADDRESS` or a class name implementing a custom `AppIdUserIdMechanism`

The corresponding command to generate the IP address UserId from a command line is:

```
$ echo -n 192.168.99.1 | sha256sum
```

Including the line break of `echo` leads to a different hash value so make sure to include the `-n` flag.

Mac address-based UserId's obtain their network device from the localhost-bound device. The configuration also allows specifying a `network-interface` hint to pick the right device. The value of `network-interface` is optional and can be either an interface name or interface index (0-based).

Example 6. bootstrap.yml using SHA256 Mac-Address UserId's

```
YAML
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: MAC_ADDRESS
        network-interface: eth0
```

- `network-interface` sets network interface to obtain the physical address

The corresponding command to generate the IP address UserId from a command line is:

```
$ echo -n 0AFEDE1234AC | sha256sum
```

The Mac address is specified uppercase and without colons. Including the line break of `echo` leads to a different hash value so make sure to include the `-n` flag.

10.2.1. Custom UserId

The UserId generation is an open mechanism. You can set `spring.cloud.vault.app-id.user-id` to any string and the configured value will be used as static UserId.

A more advanced approach lets you set `spring.cloud.vault.app-id.user-id` to a classname. This class must be on your classpath and must implement the `org.springframework.cloud.vault.AppIdUserIdMechanism` interface and the `createUserId` method. Spring Cloud Vault will obtain the UserId by calling `createUserId` each time it authenticates using AppId to obtain a token.

Example 7. bootstrap.yml

```
YAML
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: com.examlple.MyUserIdMechanism
```

Example 8. MyUserIdMechanism.java

```
YAML
public class MyUserIdMechanism implements AppIdUserIdMechanism {

  @Override
  public String createUserId() {
    String userId = ...
    return userId;
  }
}
```

See also: Vault Documentation: Using the App ID auth backend (https://www.vaultproject.io/docs/auth/app-id.html)

10.3. AppRole authentication

AppRole (https://www.vaultproject.io/docs/auth/app-id.html) is intended for machine authentication, like the deprecated (since Vault 0.6.1) AppId authentication. AppRole authentication consists of two hard to guess (secret) tokens: RoleId and SecretId.

Spring Vault supports AppRole authentication by providing either RoleId only or together with a provided SecretId (push or pull mode).

RoleId and optionally SecretId must be provided by configuration, Spring Vault will not look up these or create a custom SecretId.

Example 9. bootstrap.yml with AppRole authentication properties

```
spring.cloud.vault:
    authentication: APPROLE
    app-role:
        role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
```
YAML

- `role-id` sets the RoleId.

Example 10. bootstrap.yml with all AppRole authentication properties

```
spring.cloud.vault:
    authentication: APPROLE
    app-role:
        role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
        secret-id: 1696536f-1976-73b1-b241-0b4213908d39
        app-auth-path: approle
```
YAML

- `role-id` sets the RoleId.
- `secret-id` sets the SecretId. SecretId can be omitted if AppRole is configured without requiring SecretId (See `bind_secret_id`)
- `approle-path` sets the path of the approle authentication mount to use

See also: Vault Documentation: Using the AppRole auth backend (https://www.vaultproject.io/docs/auth/approle.html)

10.4. AWS-EC2 authentication

The aws-ec2 (https://www.vaultproject.io/docs/auth/aws-ec2.html) auth backend provides a secure introduction mechanism for AWS EC2 instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each EC2 instance.

Example 11. bootstrap.yml using AWS-EC2 Authentication

```
spring.cloud.vault:
    authentication: AWS_EC2
```
YAML

AWS-EC2 authentication enables nonce by default to follow the Trust On First Use (TOFU) principle. Any unintended party that gains access to the PKCS#7 identity metadata can authenticate against Vault.

During the first login, Spring Cloud Vault generates a nonce that is stored in the auth backend aside the instance Id. Re-authentication requires the same nonce to be sent. Any other party does not have the nonce and can raise an alert in Vault for further investigation.

The nonce is kept in memory and is lost during application restart.

AWS-EC2 authentication roles are optional and default to the AMI. You can configure the authentication role by setting the `spring.cloud.vault.aws-ec2.role` property.

Example 12. bootstrap.yml with configured role

```
spring.cloud.vault:
    authentication: AWS_EC2
    aws-ec2:
        role: application-server
```
YAML

Example 13. bootstrap.yml with all AWS EC2 authentication properties

```yaml
spring.cloud.vault:
    authentication: AWS_EC2
    aws-ec2:
        role: application-server
        aws-ec2-path: aws-ec2
        identity-document: http://...
```

- `authentication` setting this value to `AWS_EC2` selects the AWS EC2 authentication method
- `role` sets the role name of the AWS EC2 role definition
- `aws-ec2-path` sets the path of the AWS EC2 mount to use
- `identity-document` sets URL of the PKCS#7 AWS EC2 identity document

See also: Vault Documentation: Using the aws-ec2 auth backend (https://www.vaultproject.io/docs/auth/aws-ec2.html)

10.5. TLS certificate authentication

The `cert` auth backend allows authentication using SSL/TLS client certificates that are either signed by a CA or self-signed.

To enable `cert` authentication you need to:

1. Use SSL, see Vault Client SSL configuration
2. Configure a Java `Keystore` that contains the client certificate and the private key
3. Set the `spring.cloud.vault.authentication` to `CERT`

Example 14. bootstrap.yml

```yaml
spring.cloud.vault:
    authentication: CERT
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: changeit
        cert-auth-path: cert
```

See also: Vault Documentation: Using the Cert auth backend (https://www.vaultproject.io/docs/auth/cert.html)

10.6. Cubbyhole authentication

Cubbyhole authentication uses Vault primitives to provide a secured authentication workflow. Cubbyhole authentication uses tokens as primary login method. An ephemeral token is used to obtain a second, login VaultToken from Vault's Cubbyhole secret backend. The login token is usually longer-lived and used to interact with Vault. The login token will be retrieved from a wrapped response stored at `/cubbyhole/response`.

Creating a wrapped token

Response Wrapping for token creation requires Vault 0.6.0 or higher.

Example 15. Crating and storing tokens

```shell
$ vault token-create -wrap-ttl="10m"
Key                           Value
---                           -----
wrapping_token:               397ccb93-ff6c-b17b-9389-380b01ca2645
wrapping_token_ttl:           0h10m0s
wrapping_token_creation_time: 2016-09-18 20:29:48.652957077 +0200 CEST
wrapped_accessor:             46b6aebb-187f-932a-26d7-4f3d86a68319
```

Example 16. bootstrap.yml

```yaml
spring.cloud.vault:
    authentication: CUBBYHOLE
    token: 397ccb93-ff6c-b17b-9389-380b01ca2645
```

See also:

- Vault Documentation: Tokens (https://www.vaultproject.io/docs/concepts/tokens.html)
- Vault Documentation: Cubbyhole Secret Backend (https://www.vaultproject.io/docs/secrets/cubbyhole/index.html)
- Vault Documentation: Response Wrapping (https://www.vaultproject.io/docs/concepts/response-wrapping.html)

11. Secret Backends

11.1. Generic Backend

Spring Cloud Vault supports at the basic level the generic secret backend. The generic secret backend allows storage of arbitrary values as key-value store. A single context can store one or many key-value tuples. Contexts can be organized hierarchically. Spring Cloud Vault allows using the Application name and a default context name (`application`) in combination with active profiles.

```
/secret/{application}/{profile}
/secret/{application}
/secret/{default-context}/{profile}
/secret/{default-context}
```

The application name is determined by the properties:

- `spring.cloud.vault.generic.application-name`
- `spring.cloud.vault.application-name`
- `spring.application.name`

Secrets can be obtained from other folders within the generic backend by adding their paths to the application name, separated by commas. For example, given the application name `usefulapp,mysql1,projectx/aws` , each of these folders will be used:

- `/secret/usefulapp`
- `/secret/mysql1`
- `/secret/projectx/aws`

Spring Cloud Vault adds all active profiles to the list of possible context paths. No active profiles will skip accessing contexts with a profile name.

Properties are exposed like they are stored (i.e. without additional prefixes).

```yaml
spring.cloud.vault:
    generic:
        enabled: true
        backend: secret
        profile-separator: '/'
        default-context: application
        application-name: my-app
```

- `enabled` setting this value to `false` disables the secret backend config usage
- `backend` sets the path of the secret mount to use
- `default-context` sets the context name used by all applications
- `application-name` overrides the application name for use in the generic backend
- `profile-separator` separates the profile name from the context in property sources with profiles

See also: Vault Documentation: Using the generic secret backend (https://www.vaultproject.io/docs/secrets/generic/index.html)

11.2. Consul

Spring Cloud Vault can obtain credentials for HashiCorp Consul. The Consul integration requires the `spring-cloud-vault-config-consul` dependency.

Example 17. pom.xml

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-consul</artifactId>
        <version>Dalston.SR3</version>
    </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.consul.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.consul.role=…`.

The obtained token is stored in `spring.cloud.consul.token` so using Spring Cloud Consul can pick up the generated credentials without further configuration. You can configure the property name by setting `spring.cloud.vault.consul.token-property`.

```yaml
spring.cloud.vault:
    consul:
        enabled: true
        role: readonly
        backend: consul
        token-property: spring.cloud.consul.token
```

- `enabled` setting this value to `true` enables the Consul backend config usage
- `role` sets the role name of the Consul role definition
- `backend` sets the path of the Consul mount to use
- `token-property` sets the property name in which the Consul ACL token is stored

See also: Vault Documentation: Setting up Consul with Vault (https://www.vaultproject.io/docs/secrets/consul/index.html)

11.3. RabbitMQ

Spring Cloud Vault can obtain credentials for RabbitMQ.

The RabbitMQ integration requires the `spring-cloud-vault-config-rabbitmq` dependency.

Example 18. pom.xml

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-rabbitmq</artifactId>
        <version>Dalston.SR3</version>
    </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.rabbitmq.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.rabbitmq.role=…`.

Username and password are stored in `spring.rabbitmq.username` and `spring.rabbitmq.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.rabbitmq.username-property` and `spring.cloud.vault.rabbitmq.password-property`.

```yaml
spring.cloud.vault:
    rabbitmq:
        enabled: true
        role: readonly
        backend: rabbitmq
        username-property: spring.rabbitmq.username
        password-property: spring.rabbitmq.password
```

- `enabled` setting this value to `true` enables the RabbitMQ backend config usage
- `role` sets the role name of the RabbitMQ role definition
- `backend` sets the path of the RabbitMQ mount to use
- `username-property` sets the property name in which the RabbitMQ username is stored
- `password-property` sets the property name in which the RabbitMQ password is stored

See also: Vault Documentation: Setting up RabbitMQ with Vault (https://www.vaultproject.io/docs/secrets/rabbitmq/index.html)

11.4. AWS

Spring Cloud Vault can obtain credentials for AWS.

The AWS integration requires the `spring-cloud-vault-config-aws` dependency.

Example 19. pom.xml

```
XML
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-aws</artifactId>
        <version>Dalston.SR3</version>
    </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.aws=true` (default `false`) and providing the role name with `spring.cloud.vault.aws.role=…`.

The access key and secret key are stored in `cloud.aws.credentials.accessKey` and `cloud.aws.credentials.secretKey` so using Spring Cloud AWS will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.aws.access-key-property` and `spring.cloud.vault.aws.secret-key-property`.

```
YAML
spring.cloud.vault:
    aws:
        enabled: true
        role: readonly
        backend: aws
        access-key-property: cloud.aws.credentials.accessKey
        secret-key-property: cloud.aws.credentials.secretKey
```

- `enabled` setting this value to `true` enables the AWS backend config usage
- `role` sets the role name of the AWS role definition
- `backend` sets the path of the AWS mount to use
- `access-key-property` sets the property name in which the AWS access key is stored
- `secret-key-property` sets the property name in which the AWS secret key is stored

See also: Vault Documentation: Setting up AWS with Vault (https://www.vaultproject.io/docs/secrets/aws/index.html)

12. Database backends

Vault supports several database secret backends to generate database credentials dynamically based on configured roles. This means services that need to access a database no longer need to configure credentials: they can request them from Vault, and use Vault's leasing mechanism to more easily roll keys.

Spring Cloud Vault integrates with these backends:

- Apache Cassandra
- MongoDB
- MySQL
- PostgreSQL

Using a database secret backend requires to enable the backend in the configuration and the `spring-cloud-vault-config-databases` dependency.

Example 20. pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-databases</artifactId>
        <version>Dalston.SR3</version>
    </dependency>
</dependencies>
```

XML

Enabling multiple JDBC-compliant databases will generate credentials and store them by default in the same property keys hence property names for JDBC secrets need to be configured separately.

12.1. Apache Cassandra

Spring Cloud Vault can obtain credentials for Apache Cassandra. The integration can be enabled by setting `spring.cloud.vault.cassandra.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.cassandra.role=…`.

Username and password are stored in `spring.data.cassandra.username` and `spring.data.cassandra.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.cassandra.username-property` and `spring.cloud.vault.cassandra.password-property`.

```
spring.cloud.vault:
    cassandra:
        enabled: true
        role: readonly
        backend: cassandra
        username-property: spring.data.cassandra.username
        password-property: spring.data.cassandra.username
```

YAML

- `enabled` setting this value to `true` enables the Cassandra backend config usage
- `role` sets the role name of the Cassandra role definition
- `backend` sets the path of the Cassandra mount to use
- `username-property` sets the property name in which the Cassandra username is stored
- `password-property` sets the property name in which the Cassandra password is stored

See also: Vault Documentation: Setting up Apache Cassandra with Vault (https://www.vaultproject.io/docs/secrets/cassandra/index.html)

12.2. MongoDB

Spring Cloud Vault can obtain credentials for MongoDB. The integration can be enabled by setting `spring.cloud.vault.mongodb.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.mongodb.role=…`.

Username and password are stored in `spring.data.mongodb.username` and `spring.data.mongodb.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.mongodb.username-property` and `spring.cloud.vault.mongodb.password-property`.

```
YAML
spring.cloud.vault:
    mongodb:
        enabled: true
        role: readonly
        backend: mongodb
        username-property: spring.data.mongodb.username
        password-property: spring.data.mongodb.password
```

- `enabled` setting this value to `true` enables the MongodB backend config usage
- `role` sets the role name of the MongoDB role definition
- `backend` sets the path of the MongoDB mount to use
- `username-property` sets the property name in which the MongoDB username is stored
- `password-property` sets the property name in which the MongoDB password is stored

See also: Vault Documentation: Setting up MongoDB with Vault (https://www.vaultproject.io/docs/secrets/mongodb/index.html)

12.3. MySQL

Spring Cloud Vault can obtain credentials for MySQL. The integration can be enabled by setting `spring.cloud.vault.mysql.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.mysql.role=…`.

Username and password are stored in `spring.datasource.username` and `spring.datasource.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.mysql.username-property` and `spring.cloud.vault.mysql.password-property`.

```
YAML
spring.cloud.vault:
    mysql:
        enabled: true
        role: readonly
        backend: mysql
        username-property: spring.datasource.username
        password-property: spring.datasource.username
```

- `enabled` setting this value to `true` enables the MySQL backend config usage
- `role` sets the role name of the MySQL role definition
- `backend` sets the path of the MySQL mount to use
- `username-property` sets the property name in which the MySQL username is stored
- `password-property` sets the property name in which the MySQL password is stored

See also: Vault Documentation: Setting up MySQL with Vault (https://www.vaultproject.io/docs/secrets/mysql/index.html)

12.4. PostgreSQL

Spring Cloud Vault can obtain credentials for PostgreSQL. The integration can be enabled by setting `spring.cloud.vault.postgresql.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.postgresql.role=…`.

Username and password are stored in `spring.datasource.username` and `spring.datasource.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.postgresql.username-property` and `spring.cloud.vault.postgresql.password-property`.

```yaml
spring.cloud.vault:
    postgresql:
        enabled: true
        role: readonly
        backend: postgresql
        username-property: spring.datasource.username
        password-property: spring.datasource.username
```

- `enabled` setting this value to `true` enables the PostgreSQL backend config usage
- `role` sets the role name of the PostgreSQL role definition
- `backend` sets the path of the PostgreSQL mount to use
- `username-property` sets the property name in which the PostgreSQL username is stored
- `password-property` sets the property name in which the PostgreSQL password is stored

See also: Vault Documentation: Setting up PostgreSQL with Vault (https://www.vaultproject.io/docs/secrets/postgresql/index.html)

13. Vault Client Fail Fast

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

```yaml
spring.cloud.vault:
    fail-fast: true
```

14. Vault Client SSL configuration

SSL can be configured declaratively by setting various properties. You can set either `javax.net.ssl.trustStore` to configure JVM-wide SSL settings or `spring.cloud.vault.ssl.trust-store` to set SSL settings only for Spring Cloud Vault Config.

```
YAML
spring.cloud.vault:
    ssl:
        trust-store: classpath:keystore.jks
        trust-store-password: changeit
```

- `trust-store` sets the resource for the trust-store. SSL-secured Vault communication will validate the Vault SSL certificate with the specified trust-store.
- `trust-store-password` sets the trust-store password

Please note that configuring `spring.cloud.vault.ssl.*` can be only applied when either Apache Http Components or the OkHttp client is on your class-path.

15. Lease lifecycle management (renewal and revocation)

With every secret, Vault creates a lease: metadata containing information such as a time duration, renewability, and more.

Vault promises that the data will be valid for the given duration, or Time To Live (TTL). Once the lease is expired, Vault can revoke the data, and the consumer of the secret can no longer be certain that it is valid.

Spring Cloud Vault maintains a lease lifecycle beyond the creation of login tokens and secrets. That said, login tokens and secrets associated with a lease are scheduled for renewal just before the lease expires until terminal expiry. Application shutdown revokes obtained login tokens and renewable leases.

Secret service and database backends (such as MongoDB or MySQL) usually generate a renewable lease so generated credentials will be disabled on application shutdown.

Static tokens are not renewed or revoked.

Lease renewal and revocation is enabled by default and can be disabled by setting `spring.cloud.vault.config.lifecycle.enabled` to `false`. This is not recommended as leases can expire and Spring Cloud Vault cannot longer access Vault or services using generated credentials and valid credentials remain active after application shutdown.

```
YAML
spring.cloud.vault:
    config.lifecycle.enabled: true
```

See also: Vault Documentation: Lease, Renew, and Revoke (https://www.vaultproject.io/docs/concepts/lease.html)

Appendix: Compendium of Configuration Properties

Name	Default	Description
encrypt.fail-on-error	true	Flag to say that a process should fail if the decryption error.
encrypt.key		A symmetric key. As a stronger alternative keystore.
encrypt.key-store.alias		Alias for a key in the store.
encrypt.key-store.location		Location of the key store file, e.g. classpath
encrypt.key-store.password		Password that locks the keystore.
encrypt.key-store.secret		Secret protecting the key (defaults to the s
encrypt.rsa.algorithm		The RSA algorithm to use (DEFAULT or OE not change it (or existing ciphers will not a
encrypt.rsa.salt	deadbeef	Salt for the random secret used to encrypt set do not change it (or existing ciphers wi
encrypt.rsa.strong	false	Flag to indicate that "strong" AES encrypti internally. If true then the GCM algorithm encrypted bytes. Default is false (in which used instead). Once it is set do not change will not a decryptable).
endpoints.bus.enabled		
endpoints.bus.id		
endpoints.bus.sensitive		
endpoints.consul.enabled		
endpoints.consul.id		
endpoints.consul.sensitive		
endpoints.env.post.enabled	true	Enable changing the Environment through
endpoints.features.enabled		
endpoints.features.id		
endpoints.features.sensitive		
endpoints.pause.enabled	true	Enable the /pause endpoint (to send Lifecy
endpoints.pause.id		
endpoints.pause.sensitive		
endpoints.refresh.enabled	true	Enable the /refresh endpoint to refresh co initialize refresh scoped beans.
endpoints.refresh.id		
endpoints.refresh.sensitive		
endpoints.restart.enabled	true	Enable the /restart endpoint to restart the

Name	Default	Description
endpoints.restart.id		
endpoints.restart.pause-endpoint.enabled		
endpoints.restart.pause-endpoint.id		
endpoints.restart.pause-endpoint.sensitive		
endpoints.restart.resume-endpoint.enabled		
endpoints.restart.resume-endpoint.id		
endpoints.restart.resume-endpoint.sensitive		
endpoints.restart.sensitive		
endpoints.restart.timeout	0	
endpoints.resume.enabled	true	Enable the /resume endpoint (to send Life
endpoints.resume.id		
endpoints.resume.sensitive		
endpoints.zookeeper.enabled	true	Enable the /zookeeper endpoint to inspect
eureka.client.allow-redirects	false	Indicates whether server can redirect a cli backup server/cluster. If set to false, the se request directly, If set to true, it may send client, with a new server location.
eureka.client.availability-zones		Gets the list of availability zones (used in A the region in which this instance resides. The changes are effective at runtime at th cycle as specified by registryFetchInterval
eureka.client.backup-registry-impl		Gets the name of the implementation whi BackupRegistry to fetch the registry inforr option for only the first time when the eur This may be needed for applications whicl resiliency for registry information withou operate.
eureka.client.cache-refresh-executor-exponential-back-off-bound	10	Cache refresh executor exponential back o is a maximum multiplier value for retry d sequence of timeouts occurred.
eureka.client.cache-refresh-executor-thread-pool-size	2	The thread pool size for the cacheRefreshI with
eureka.client.client-data-accept		EurekaAccept name for client data accept
eureka.client.decoder-name		This is a transient config and once the late can be removed (as there will only be one

Name	Default	Description
eureka.client.disable-delta	false	Indicates whether the eureka client shoul delta and should rather resort to getting th information. Note that the delta fetches can reduce the because the rate of change with the eurek much lower than the rate of fetches. The changes are effective at runtime at th cycle as specified by registryFetchInterval
eureka.client.dollar-replacement	_-	Get a replacement string for Dollar sign <c serializing/deserializing information in eu
eureka.client.enabled	true	Flag to indicate that the Eureka client is en
eureka.client.encoder-name		This is a transient config and once the late can be removed (as there will only be one
eureka.client.escape-char-replacement	__	Get a replacement string for underscore si during serializing/deserializing informatic
eureka.client.eureka-connection-idle-timeout-seconds	30	Indicates how much time (in seconds) that to eureka server can stay idle before it can In the AWS environment, it is recommend seconds or less, since the firewall cleans u information after a few mins leaving the c limbo
eureka.client.eureka-server-connect-timeout-seconds	5	Indicates how long to wait (in seconds) be eureka server needs to timeout. Note that client are pooled by org.apache.http.client setting affects the actual connection creati time to get the connection from the pool.
eureka.client.eureka-server-d-n-s-name		Gets the DNS name to be queried to get th servers.This information is not required if the service urls by implementing serviceU The DNS mechanism is used when useDnsForFetchingServiceUrls is set to tru client expects the DNS to configured a cert fetch changing eureka servers dynamicall The changes are effective at runtime.
eureka.client.eureka-server-port		Gets the port to be used to construct the se eureka server when the list of eureka serv DNS.This information is not required if th service urls eurekaServerServiceUrls(Strin The DNS mechanism is used when useDnsForFetchingServiceUrls is set to tru client expects the DNS to configured a cert fetch changing eureka servers dynamicall The changes are effective at runtime.
eureka.client.eureka-server-read-timeout-seconds	8	Indicates how long to wait (in seconds) be eureka server needs to timeout.

Name	Default	Description
eureka.client.eureka-server-total-connections	200	Gets the total number of connections that eureka client to all eureka servers.
eureka.client.eureka-server-total-connections-per-host	50	Gets the total number of connections that eureka client to a eureka server host.
eureka.client.eureka-server-u-r-l-context		Gets the URL context to be used to constru contact eureka server when the list of eur the DNS. This information is not required the service urls from eurekaServerService The DNS mechanism is used when useDnsForFetchingServiceUrls is set to tru client expects the DNS to configured a cert fetch changing eureka servers dynamicall effective at runtime.
eureka.client.eureka-service-url-poll-interval-seconds	0	Indicates how often(in seconds) to poll for server information. Eureka servers could and this setting controls how soon the eur know about it.
eureka.client.fetch-registry	true	Indicates whether this client should fetch information from eureka server.
eureka.client.fetch-remote-regions-registry		Comma separated list of regions for which information will be fetched. It is mandato availability zones for each of these region availabilityZones. Failing to do so, will res discovery client startup.
eureka.client.filter-only-up-instances	true	Indicates whether to get the applications a applications for instances with only Instar
eureka.client.g-zip-content	true	Indicates whether the content fetched fro be compressed whenever it is supported b registry information from the eureka serv optimum network traffic.
eureka.client.heartbeat-executor-exponential-back-off-bound	10	Heartbeat executor exponential back off r maximum multiplier value for retry delay sequence of timeouts occurred.
eureka.client.heartbeat-executor-thread-pool-size	2	The thread pool size for the heartbeatExec
eureka.client.initial-instance-info-replication-interval-seconds	40	Indicates how long initially (in seconds) to to the eureka server
eureka.client.instance-info-replication-interval-seconds	30	Indicates how often(in seconds) to replica be replicated to the eureka server.

Name	Default	Description
eureka.client.log-delta-diff	false	Indicates whether to log differences betwe and the eureka client in terms of registry i Eureka client tries to retrieve only delta cl server to minimize network traffic. After r eureka client reconciles the information fi verify it has not missed out some informat failures could happen when the client has communicating to server.If the reconciliat gets the full registry information. While getting the full registry information log the differences between the client and setting controls that. The changes are effective at runtime at th cycle as specified by registryFetchInterval
eureka.client.on-demand-update-status-change	true	If set to true, local status updates via Appli will trigger on-demand (but rate limited) r remote eureka servers
eureka.client.prefer-same-zone-eureka	true	Indicates whether or not this instance sho eureka server in the same zone for latency Ideally eureka clients are configured to ta same zone The changes are effective at runtime at th cycle as specified by registryFetchInterval
eureka.client.property-resolver		
eureka.client.proxy-host		Gets the proxy host to eureka server if any
eureka.client.proxy-password		Gets the proxy password if any.
eureka.client.proxy-port		Gets the proxy port to eureka server if any
eureka.client.proxy-user-name		Gets the proxy user name if any.
eureka.client.region	us-east-1	Gets the region (used in AWS datacenters) resides.
eureka.client.register-with-eureka	true	Indicates whether or not this instance sho information with eureka server for discov In some cases, you do not want your insta whereas you just want do discover other i
eureka.client.registry-fetch-interval-seconds	30	Indicates how often(in seconds) to fetch th from the eureka server.
eureka.client.registry-refresh-single-vip-address		Indicates whether the client is only interes information for a single VIP.

Name	Default	Description
eureka.client.service-url		Map of availability zone to list of fully qua communicate with eureka server. Each va URL or a comma separated list of alternati Typically the eureka server URLs carry protocol,host,port,context and version inf Example: http://ec2-256-156-243-129.comp 1.amazonaws.com:7001/eureka/ The changes are effective at runtime at th refresh cycle as specified by eurekaServiceUrlPollIntervalSeconds.
eureka.client.transport		
eureka.client.use-dns-for-fetching-service-urls	false	Indicates whether the eureka client shoul mechanism to fetch a list of eureka server DNS name is updated to have additional s information is used immediately after the that information as specified in eurekaServiceUrlPollIntervalSeconds. Alternatively, the service urls can be retur the users should implement their own me updated list in case of changes. The changes are effective at runtime.
eureka.dashboard.enabled	true	Flag to enable the Eureka dashboard. Defa
eureka.dashboard.path	/	The path to the Eureka dashboard (relativ Defaults to "/".
eureka.instance.a-s-g-name		Gets the AWS autoscaling group name ass instance. This information is specifically u environment to automatically put an insta after the instance is launched and it has b traffic..
eureka.instance.app-group-name		Get the name of the application group to b eureka.
eureka.instance.appname	unknown	Get the name of the application to be regis
eureka.instance.data-center-info		Returns the data center this instance is de information is used to get some AWS speci information if the instance is deployed in
eureka.instance.default-address-resolution-order	[]	
eureka.instance.environment		

Name	Default	Description
eureka.instance.health-check-url		Gets the absolute health check page URL f users can provide the healthCheckUrlPath page resides in the same instance talking t cases where the instance is a proxy for so can provide the full URL. If the full URL is precedence. <p> It is normally used for making educat the health of the instance - for example, it determine whether to proceed deploymen stop the deployments without causing furt URL should follow the format http://${eur where the value ${eureka.hostname} is re
eureka.instance.health-check-url-path	/health	Gets the relative health check URL path fo health check page URL is then constructed and the type of communication - secure o in securePort and nonSecurePort. It is normally used for making educated d health of the instance - for example, it can whether to proceed deployments to an en deployments without causing further dam
eureka.instance.home-page-url		Gets the absolute home page URL for this provide the homePageUrlPath if the home same instance talking to eureka, else in th instance is a proxy for some other server, full URL. If the full URL is provided it take It is normally used for informational purp to use it as a landing page. The full URL sh http://${eureka.hostname}:7001/ where th ${eureka.hostname} is replaced at runtim
eureka.instance.home-page-url-path	/	Gets the relative home page URL Path for home page URL is then constructed out of type of communication - secure or unsecu It is normally used for informational purp to use it as a landing page.
eureka.instance.host-info		
eureka.instance.hostname		The hostname if it can be determined at c (otherwise it will be guessed from OS prim
eureka.instance.inet-utils		
eureka.instance.initial-status		Initial status to register with rmeote Eurek
eureka.instance.instance-enabled-onit	false	Indicates whether the instance should be traffic as soon as it is registered with eure application might need to do some pre-pr ready to take traffic.
eureka.instance.instance-id		Get the unique Id (within the scope of the instance to be registered with eureka.

Name	Default	Description
eureka.instance.ip-address		Get the IPAdress of the instance. This infor academic purposes only as the communic instances primarily happen using the info {@link #getHostName(boolean)}.
eureka.instance.lease-expiration-duration-in-seconds	90	Indicates the time in seconds that the eure it received the last heartbeat before it can from its view and there by disallowing tra Setting this value too long could mean tha routed to the instance even though the ins Setting this value too small could mean, th taken out of traffic because of temporary r value to be set to atleast higher than the v leaseRenewalIntervalInSeconds.
eureka.instance.lease-renewal-interval-in-seconds	30	Indicates how often (in seconds) the eurek heartbeats to eureka server to indicate tha heartbeats are not received for the period leaseExpirationDurationInSeconds, eurek the instance from its view, there by disallo instance. Note that the instance could still not take t HealthCheckCallback and then decides to unavailable.
eureka.instance.metadata-map		Gets the metadata name/value pairs assoc instance. This information is sent to eurek used by other instances.
eureka.instance.namespace	eureka	Get the namespace used to find properties Cloud.
eureka.instance.non-secure-port	80	Get the non-secure port on which the insta traffic.
eureka.instance.non-secure-port-enabled	true	Indicates whether the non-secure port sho traffic or not.
eureka.instance.prefer-ip-address	false	Flag to say that, when guessing a hostnam server should be used in prference to the the OS.
eureka.instance.registry.default-open-for-traffic-count	1	Value used in determining when leases ar 1 for standalone. Should be set to 0 for pee
eureka.instance.registry.expected-number-of-renews-per-min	1	

Name	Default	Description
eureka.instance.secure-health-check-url		Gets the absolute secure health check pag The users can provide the secureHealthCh check page resides in the same instance ta the cases where the instance is a proxy fo users can provide the full URL. If the full U precedence. <p> It is normally used for making educat the health of the instance - for example, it determine whether to proceed deploymen stop the deployments without causing furt URL should follow the format http://${eur where the value ${eureka.hostname} is re
eureka.instance.secure-port	443	Get the Secure port on which the instance
eureka.instance.secure-port-enabled	false	Indicates whether the secure port should or not.
eureka.instance.secure-virtual-host-name	unknown	Gets the secure virtual host name defined This is typically the way other instance wo by using the secure virtual host name.Thi the fully qualified domain name, that the will need to find this instance.
eureka.instance.status-page-url		Gets the absolute status page URL path for users can provide the statusPageUrlPath i resides in the same instance talking to eur where the instance is a proxy for some oth provide the full URL. If the full URL is prov precedence. It is normally used for informational purp to find about the status of this instance. Us simple HTML indicating what is the curre instance.
eureka.instance.status-page-url-path	/info	Gets the relative status page URL path for status page URL is then constructed out of type of communication - secure or unsecu securePort and nonSecurePort. It is normally used for informational purp to find about the status of this instance. Us simple HTML indicating what is the curre instance.
eureka.instance.virtual-host-name	unknown	Gets the virtual host name defined for this This is typically the way other instance wo by using the virtual host name.Think of th fully qualified domain name, that the user need to find this instance.
eureka.server.a-s-g-cache-expiry-timeout-ms	0	
eureka.server.a-s-g-query-timeout-ms	300	
eureka.server.a-s-g-update-interval-ms	0	
eureka.server.a-w-s-access-id		

Name	Default	Description
eureka.server.a-w-s-secret-key		
eureka.server.batch-replication	false	
eureka.server.binding-strategy		
eureka.server.delta-retention-timer-interval-in-ms	0	
eureka.server.disable-delta	false	
eureka.server.disable-delta-for-remote-regions	false	
eureka.server.disable-transparent-fallback-to-other-region	false	
eureka.server.e-i-p-bind-rebind-retries	3	
eureka.server.e-i-p-binding-retry-interval-ms	0	
eureka.server.e-i-p-binding-retry-interval-ms-when-unbound	0	
eureka.server.enable-replicated-request-compression	false	
eureka.server.enable-self-preservation	true	
eureka.server.eviction-interval-timer-in-ms	0	
eureka.server.g-zip-content-from-remote-region	true	
eureka.server.json-codec-name		
eureka.server.list-auto-scaling-groups-role-name	ListAutoScalingGroups	
eureka.server.log-identity-headers	true	
eureka.server.max-elements-in-peer-replication-pool	10000	
eureka.server.max-elements-in-status-replication-pool	10000	
eureka.server.max-idle-thread-age-in-minutes-for-peer-replication	15	
eureka.server.max-idle-thread-in-minutes-age-for-status-replication	10	
eureka.server.max-threads-for-peer-replication	20	
eureka.server.max-threads-for-status-replication	1	
eureka.server.max-time-for-replication	30000	
eureka.server.min-threads-for-peer-replication	5	
eureka.server.min-threads-for-status-replication	1	
eureka.server.number-of-replication-retries	5	
eureka.server.peer-eureka-nodes-update-interval-ms	0	

Name	Default	Description
eureka.server.peer-eureka-status-refresh-time-interval-ms	0	
eureka.server.peer-node-connect-timeout-ms	200	
eureka.server.peer-node-connection-idle-timeout-seconds	30	
eureka.server.peer-node-read-timeout-ms	200	
eureka.server.peer-node-total-connections	1000	
eureka.server.peer-node-total-connections-per-host	500	
eureka.server.prime-aws-replica-connections	true	
eureka.server.property-resolver		
eureka.server.rate-limiter-burst-size	10	
eureka.server.rate-limiter-enabled	false	
eureka.server.rate-limiter-full-fetch-average-rate	100	
eureka.server.rate-limiter-privileged-clients		
eureka.server.rate-limiter-registry-fetch-average-rate	500	
eureka.server.rate-limiter-throttle-standard-clients	false	
eureka.server.registry-sync-retries	0	
eureka.server.registry-sync-retry-wait-ms	0	
eureka.server.remote-region-app-whitelist		
eureka.server.remote-region-connect-timeout-ms	1000	
eureka.server.remote-region-connection-idle-timeout-seconds	30	
eureka.server.remote-region-fetch-thread-pool-size	20	
eureka.server.remote-region-read-timeout-ms	1000	
eureka.server.remote-region-registry-fetch-interval	30	
eureka.server.remote-region-total-connections	1000	
eureka.server.remote-region-total-connections-per-host	500	
eureka.server.remote-region-trust-store		
eureka.server.remote-region-trust-store-password	changeit	
eureka.server.remote-region-urls		
eureka.server.remote-region-urls-with-name		

Name	Default	Description
eureka.server.renewal-percent-threshold	0.85	
eureka.server.renewal-threshold-update-interval-ms	0	
eureka.server.response-cache-auto-expiration-in-seconds	180	
eureka.server.response-cache-update-interval-ms	0	
eureka.server.retention-time-in-m-s-in-delta-queue	0	
eureka.server.route53-bind-rebind-retries	3	
eureka.server.route53-binding-retry-interval-ms	0	
eureka.server.route53-domain-t-t-l	30	
eureka.server.sync-when-timestamp-differs	true	
eureka.server.use-read-only-response-cache	true	
eureka.server.wait-time-in-ms-when-sync-empty	0	
eureka.server.xml-codec-name		
feign.compression.request.mime-types	[text/xml, application/xml, application/json]	The list of supported mime types.
feign.compression.request.min-request-size	2048	The minimum threshold content size.
health.config.enabled	false	Flag to indicate that the config server heal installed.
health.config.time-to-live	0	Time to live for cached result, in milliseco min).
hystrix.metrics.enabled	true	Enable Hystrix metrics polling. Defaults to
hystrix.metrics.polling-interval-ms	2000	Interval between subsequent polling of m ms.
management.health.refresh.enabled	true	Enable the health endpoint for the refresh
management.health.zookeeper.enabled	true	Enable the health endpoint for zookeeper.
netflix.atlas.batch-size	10000	
netflix.atlas.enabled	true	
netflix.atlas.uri		
netflix.metrics.servo.cache-warning-threshold	1000	When the `ServoMonitorCache` reaches th logged. This will be useful if you are using in RestTemplate urls.
netflix.metrics.servo.registry-class	com.netflix.servo.BasicMonitorRegistry	Fully qualified class name for monitor reg
proxy.auth.load-balanced		
proxy.auth.routes		Authentication strategy per route.
spring.cloud.bus.ack.destination-service		Service that wants to listen to acks. By def services).

Name	Default	Description
spring.cloud.bus.ack.enabled	true	Flag to switch off acks (default on).
spring.cloud.bus.destination	springCloudBus	Name of Spring Cloud Stream destination
spring.cloud.bus.enabled	true	Flag to indicate that the bus is enabled.
spring.cloud.bus.env.enabled	true	Flag to switch off environment change eve
spring.cloud.bus.refresh.enabled	true	Flag to switch off refresh events (default o
spring.cloud.bus.trace.enabled	false	Flag to switch on tracing of acks (default o
spring.cloud.cloudfoundry.discovery.enabled	true	Flag to indicate that discovery is enabled.
spring.cloud.cloudfoundry.discovery.heartbeat-frequency	5000	Frequency in milliseconds of poll for hear poll on this frequency and broadcast a list
spring.cloud.cloudfoundry.discovery.org		Organization name to authenticate with (d default).
spring.cloud.cloudfoundry.discovery.password		Password for user to authenticate and obt
spring.cloud.cloudfoundry.discovery.space		Space name to authenticate with (default t
spring.cloud.cloudfoundry.discovery.url	https://api.run.pivotal.io	URL of Cloud Foundry API (Cloud Controll
spring.cloud.cloudfoundry.discovery.username		Username to authenticate (usually an ema
spring.cloud.config.allow-override	true	Flag to indicate that {@link #isSystemProp systemPropertiesOverride} can be used. S users from changing the default accidenta
spring.cloud.config.authorization		Authorization token used by the client to c
spring.cloud.config.discovery.enabled	false	Flag to indicate that config server discover server URL will be looked up via discovery
spring.cloud.config.discovery.service-id	configserver	Service id to locate config server.
spring.cloud.config.enabled	true	Flag to say that remote configuration is en
spring.cloud.config.fail-fast	false	Flag to indicate that failure to connect to t (default false).
spring.cloud.config.label		The label name to use to pull remote confi The default is set on the server (generally based server).
spring.cloud.config.name		Name of application used to fetch remote
spring.cloud.config.override-none	false	Flag to indicate that when {@link #setAllo allowOverride} is true, external properties priority, and not override any existing pro (including local config files). Default false.
spring.cloud.config.override-system-properties	true	Flag to indicate that the external propertie system properties. Default true.
spring.cloud.config.password		The password to use (HTTP Basic) when co server.
spring.cloud.config.profile	default	The default profile to use when fetching re (comma-separated). Default is "default".
spring.cloud.config.retry.initial-interval	1000	Initial retry interval in milliseconds.

Name	Default	Description
spring.cloud.config.retry.max-attempts	6	Maximum number of attempts.
spring.cloud.config.retry.max-interval	2000	Maximum interval for backoff.
spring.cloud.config.retry.multiplier	1.1	Multiplier for next interval.
spring.cloud.config.server.bootstrap	false	Flag indicating that the config server shou Environment with properties from the rer default because it delays startup but can b embedding the server in another applicati
spring.cloud.config.server.default-application-name	application	Default application name when incoming specific one.
spring.cloud.config.server.default-label		Default repository label when incoming re specific label.
spring.cloud.config.server.default-profile	default	Default application profile when incoming a specific one.
spring.cloud.config.server.encrypt.enabled	true	Enable decryption of environment proper client.
spring.cloud.config.server.git.basedir		Base directory for local working copy of re
spring.cloud.config.server.git.clone-on-start		Flag to indicate that the repository should (not on demand). Generally leads to slowe first query.
spring.cloud.config.server.git.default-label		
spring.cloud.config.server.git.environment		
spring.cloud.config.server.git.force-pull		Flag to indicate that the repository should discard any local changes and take from r
spring.cloud.config.server.git.git-factory		
spring.cloud.config.server.git.password		Password for authentication with remote
spring.cloud.config.server.git.repos		Map of repository identifier to location an
spring.cloud.config.server.git.search-paths		Search paths to use within local working c searches only the root.
spring.cloud.config.server.git.timeout		Timeout (in seconds) for obtaining HTTP c applicable). Default 5 seconds.
spring.cloud.config.server.git.uri		URI of remote repository.
spring.cloud.config.server.git.username		Username for authentication with remote
spring.cloud.config.server.health.repositories		
spring.cloud.config.server.native.fail-on-error	false	Flag to determine how to handle exceptio (default false).
spring.cloud.config.server.native.search-locations	[]	Locations to search for configuration files as a Spring Boot app so [classpath:/,classpath:/config/,file:./,file:./co
spring.cloud.config.server.native.version		Version string to be reported for native re

Name	Default	Description
spring.cloud.config.server.overrides		Extra map for a property source to be sen unconditionally.
spring.cloud.config.server.prefix		Prefix for configuration resource paths (d when embedding in another application v change the context path or servlet path.
spring.cloud.config.server.strip-document-from-yaml	true	Flag to indicate that YAML documents tha (not a map) should be returned in "native'
spring.cloud.config.server.svn.basedir		Base directory for local working copy of re
spring.cloud.config.server.svn.default-label	trunk	The default label for environment propert
spring.cloud.config.server.svn.environment		
spring.cloud.config.server.svn.password		Password for authentication with remote
spring.cloud.config.server.svn.search-paths		Search paths to use within local working c searches only the root.
spring.cloud.config.server.svn.uri		URI of remote repository.
spring.cloud.config.server.svn.username		Username for authentication with remote
spring.cloud.config.token		Security Token passed thru to underlying repository.
spring.cloud.config.uri	http://localhost:8888	The URI of the remote server (default http
spring.cloud.config.username		The username to use (HTTP Basic) when c server.
spring.cloud.consul.config.acl-token		
spring.cloud.consul.config.data-key	data	If format is Format.PROPERTIES or Forma following field is used as key to look up co
spring.cloud.consul.config.default-context	application	
spring.cloud.consul.config.enabled	true	
spring.cloud.consul.config.fail-fast	true	Throw exceptions during config lookup if warnings.
spring.cloud.consul.config.format		
spring.cloud.consul.config.prefix	config	
spring.cloud.consul.config.profile-separator	,	
spring.cloud.consul.config.watch.delay	1000	The value of the fixed delay for the watch 1000.
spring.cloud.consul.config.watch.enabled	true	If the watch is enabled. Defaults to true.
spring.cloud.consul.config.watch.wait-time	55	The number of seconds to wait (or block) f defaults to 55. Needs to be less than defaul (defaults to 60). To increase ConsulClient t ConsulClient bean with a custom ConsulRa HttpClient.
spring.cloud.consul.discovery.acl-token		

Name	Default	Description
spring.cloud.consul.discovery.catalog-services-watch-delay	10	
spring.cloud.consul.discovery.catalog-services-watch-timeout	2	
spring.cloud.consul.discovery.default-query-tag		Tag to query for in service list if one is not serverListQueryTags.
spring.cloud.consul.discovery.default-zone-metadata-name	zone	Service instance zone comes from metada changing the metadata tag name.
spring.cloud.consul.discovery.enabled	true	Is service discovery enabled?
spring.cloud.consul.discovery.fail-fast	true	Throw exceptions during service registrat log warnings (defaults to true).
spring.cloud.consul.discovery.health-check-interval	10s	How often to perform the health check (e.
spring.cloud.consul.discovery.health-check-path	/health	Alternate server path to invoke for health
spring.cloud.consul.discovery.health-check-timeout		Timeout for health check (e.g. 10s)
spring.cloud.consul.discovery.health-check-url		Custom health check url to override defau
spring.cloud.consul.discovery.heartbeat.enabled	false	
spring.cloud.consul.discovery.heartbeat.heartbeat-interval		
spring.cloud.consul.discovery.heartbeat.interval-ratio		
spring.cloud.consul.discovery.heartbeat.ttl-unit	s	
spring.cloud.consul.discovery.heartbeat.ttl-value	30	
spring.cloud.consul.discovery.host-info		
spring.cloud.consul.discovery.hostname		Hostname to use when accessing server
spring.cloud.consul.discovery.instance-id		Unique service instance id
spring.cloud.consul.discovery.instance-zone		Service instance zone
spring.cloud.consul.discovery.ip-address		IP address to use when accessing service (preferIpAddress to use)
spring.cloud.consul.discovery.lifecycle.enabled	true	
spring.cloud.consul.discovery.management-port		Port to register the management service u management port)
spring.cloud.consul.discovery.management-suffix	management	Suffix to use when registering manageme
spring.cloud.consul.discovery.management-tags		Tags to use when registering management
spring.cloud.consul.discovery.port		Port to register the service under (defaults
spring.cloud.consul.discovery.prefer-agent-address	false	Source of how we will determine the addr
spring.cloud.consul.discovery.prefer-ip-address	false	Use ip address rather than hostname duri

Name	Default	Description
spring.cloud.consul.discovery.query-passing	false	Add the 'passing` parameter to /v1/health/ This pushes health check passing to the se
spring.cloud.consul.discovery.register	true	Register as a service in consul.
spring.cloud.consul.discovery.register-health-check	true	Register health check in consul. Useful du service.
spring.cloud.consul.discovery.scheme	http	Whether to register an http or https servic
spring.cloud.consul.discovery.server-list-query-tags		Map of serviceId's → tag to query for in se filtering services by a single tag.
spring.cloud.consul.discovery.service-name		Service name
spring.cloud.consul.discovery.tags		Tags to use when registering service
spring.cloud.consul.enabled	true	Is spring cloud consul enabled
spring.cloud.consul.host	localhost	Consul agent hostname. Defaults to 'localh
spring.cloud.consul.port	8500	Consul agent port. Defaults to '8500'.
spring.cloud.consul.retry.initial-interval	1000	Initial retry interval in milliseconds.
spring.cloud.consul.retry.max-attempts	6	Maximum number of attempts.
spring.cloud.consul.retry.max-interval	2000	Maximum interval for backoff.
spring.cloud.consul.retry.multiplier	1.1	Multiplier for next interval.
spring.cloud.hypermedia.refresh.fixed-delay	5000	
spring.cloud.hypermedia.refresh.initial-delay	10000	
spring.cloud.inetutils.default-hostname	localhost	The default hostname. Used in case of erro
spring.cloud.inetutils.default-ip-address	127.0.0.1	The default ipaddress. Used in case of erro
spring.cloud.inetutils.ignored-interfaces		List of Java regex expressions for network ignored.
spring.cloud.inetutils.preferred-networks		List of Java regex expressions for network ignored.
spring.cloud.inetutils.timeout-seconds	1	Timeout in seconds for calculating hostna
spring.cloud.inetutils.use-only-site-local-interfaces	false	Use only interfaces with site local address InetAddress#isSiteLocalAddress()} for mo
spring.cloud.loadbalancer.retry.enabled	false	
spring.cloud.stream.binders		
spring.cloud.stream.bindings		
spring.cloud.stream.consul.binder.event-timeout	5	
spring.cloud.stream.consumer-defaults		
spring.cloud.stream.default-binder		
spring.cloud.stream.dynamic-destinations	[]	
spring.cloud.stream.ignore-unknown-properties	true	

Name	Default	Description
spring.cloud.stream.instance-count	1	
spring.cloud.stream.instance-index	0	
spring.cloud.stream.producer-defaults		
spring.cloud.stream.rabbit.binder.admin-adresses	[]	
spring.cloud.stream.rabbit.binder.compression-level	0	
spring.cloud.stream.rabbit.binder.nodes	[]	
spring.cloud.stream.rabbit.bindings		
spring.cloud.zookeeper.base-sleep-time-ms	50	Initial amount of time to wait between ret
spring.cloud.zookeeper.block-until-connected-unit		The unit of time related to blocking on cor
spring.cloud.zookeeper.block-until-connected-wait	10	Wait time to block on connection to Zooke
spring.cloud.zookeeper.connect-string	localhost:2181	Connection string to the Zookeeper cluste
spring.cloud.zookeeper.default-health-endpoint		Default health endpoint that will be check dependency is alive
spring.cloud.zookeeper.dependencies		Mapping of alias to ZookeeperDependency perspective the alias is actually serviceID s accept nested structures in serviceID
spring.cloud.zookeeper.dependency-configurations		
spring.cloud.zookeeper.dependency-names		
spring.cloud.zookeeper.discovery.enabled	true	
spring.cloud.zookeeper.discovery.instance-host		Predefined host with which a service can Zookeeper. Corresponds to the {code addr
spring.cloud.zookeeper.discovery.instance-port		Port to register the service under (defaults
spring.cloud.zookeeper.discovery.metadata		Gets the metadata name/value pairs assoc instance. This information is sent to zooke by other instances.
spring.cloud.zookeeper.discovery.register	true	Register as a service in zookeeper.
spring.cloud.zookeeper.discovery.root	/services	Root Zookeeper folder in which all instanc
spring.cloud.zookeeper.discovery.uri-spec	{scheme}://{address}:{port}	The URI specification to resolve during ser Zookeeper
spring.cloud.zookeeper.enabled	true	Is Zookeeper enabled
spring.cloud.zookeeper.max-retries	10	Max number of times to retry
spring.cloud.zookeeper.max-sleep-ms	500	Max time in ms to sleep on each retry
spring.cloud.zookeeper.prefix		Common prefix that will be applied to all dependencies' paths
spring.integration.poller.fixed-delay	1000	Fixed delay for default poller.
spring.integration.poller.max-messages-per-poll	1	Maximum messages per poll for the defau

Name	Default	Description
spring.sleuth.integration.enabled	true	Enable Spring Integration sleuth instrume
spring.sleuth.integration.patterns	*	An array of simple patterns against which matched. Default is * (all channels). See org.springframework.util.PatternMatchUt String).
spring.sleuth.keys.async.class-name-key	class	Simple name of the class with a method a @Async} from which the asynchronous p @see org.springframework.scheduling.an
spring.sleuth.keys.async.method-name-key	method	Name of the method annotated with {@co @see org.springframework.scheduling.an
spring.sleuth.keys.async.prefix		Prefix for header names if they are added
spring.sleuth.keys.async.thread-name-key	thread	Name of the thread that executed the asyr @see org.springframework.scheduling.an
spring.sleuth.keys.http.headers		Additional headers that should be added a the header value is multi-valued, the tag v separated, single-quoted list.
spring.sleuth.keys.http.host	http.host	The domain portion of the URL or host he "mybucket.s3.amazonaws.com". Used to fi to ip address.
spring.sleuth.keys.http.method	http.method	The HTTP method, or verb, such as "GET" filter against an http route.
spring.sleuth.keys.http.path	http.path	The absolute http path, without any query "/objects/abcd-ff". Used to filter against an with zipkin v1. In zipkin v1, only equals fi Dropping query parameters makes the nu less. For example, one can query for the sa regardless of signing parameters encoded does not reduce cardinality to a HTTP sing it is common to express a route as an http "/resource/{resource_id}". In systems whe are available, searching for {@code http.u match if the actual request was "/resource note: This was commonly expressed as "ht eventhough it was most often just a path.
spring.sleuth.keys.http.prefix	http.	Prefix for header names if they are added
spring.sleuth.keys.http.request-size	http.request.size	The size of the non-empty HTTP request b "16384" <p>Large uploads can exceed limits or cor latency.
spring.sleuth.keys.http.response-size	http.response.size	The size of the non-empty HTTP response "16384" <p>Large downloads can exceed limits or latency.

Name	Default	Description
spring.sleuth.keys.http.status-code	http.status_code	The HTTP response code, when not in 2xx to filter for error status. 2xx range are not codes are less interesting for latency troub saves at least 20 bytes per span.
spring.sleuth.keys.http.url	http.url	The entire URL, including the scheme, hos parameters if available. Ex. "https://mybucket.s3.amazonaws.com/obje Algorithm=AWS4-HMAC-SHA256&X-Amz- HMAC-SHA256..." Combined with {@link # understand the fully-qualified request line may include private data or be of consider
spring.sleuth.keys.hystrix.command-group	commandGroup	Name of the command group. Hystrix use key to group together commands such as f dashboards, or team/library ownership. @see com.netflix.hystrix.HystrixComman
spring.sleuth.keys.hystrix.command-key	commandKey	Name of the command key. Describes the command. A key to represent a {@link com.netflix.hystrix.HystrixCommand} for breakers, metrics publishing, caching and @see com.netflix.hystrix.HystrixComman
spring.sleuth.keys.hystrix.prefix		Prefix for header names if they are added
spring.sleuth.keys.hystrix.thread-pool-key	threadPoolKey	Name of the thread pool key. The thread-p {@link com.netflix.hystrix.HystrixThreadP metrics publishing, caching, and other suc com.netflix.hystrix.HystrixCommand} is a {@link com.netflix.hystrix.HystrixThreadP the {@link com.netflix.hystrix.HystrixThre into it, or it defaults to one created using t com.netflix.hystrix.HystrixCommandGrou with. @see com.netflix.hystrix.HystrixThreadPo
spring.sleuth.keys.message.headers		Additional headers that should be added a the header value is not a String it will be c using its toString() method.
spring.sleuth.keys.message.payload.size	message/payload-size	An estimate of the size of the payload if av
spring.sleuth.keys.message.payload.type	message/payload-type	The type of the payload.
spring.sleuth.keys.message.prefix	message/	Prefix for header names if they are added
spring.sleuth.keys.mvc.controller-class	mvc.controller.class	The lower case, hyphen delimited name o processes the request. Ex. class named "Bo result in "book-controller" tag value.
spring.sleuth.keys.mvc.controller-method	mvc.controller.method	The lower case, hyphen delimited name o processes the request. Ex. method named result in "list-of-books" tag value.
spring.sleuth.metric.span.accepted-name	counter.span.accepted	
spring.sleuth.metric.span.dropped-name	counter.span.dropped	

Name	Default	Description
spring.sleuth.sampler.percentage	0.1	Percentage of requests that should be sam requests should be sampled. The precisior only (i.e. there's no support for 0.1% of the
spring.sleuth.trace-id128	false	When true, generate 128-bit trace IDs inst
zuul.add-host-header	false	Flag to determine whether the proxy forw
zuul.add-proxy-headers	true	Flag to determine whether the proxy adds headers.
zuul.host.max-per-route-connections	20	The maximum number of connections tha single route.
zuul.host.max-total-connections	200	The maximum number of total connectior open to backends.
zuul.ignore-local-service	true	
zuul.ignore-security-headers	true	Flag to say that SECURITY_HEADERS are a headers if spring security is on the classpa ignoreSecurityHeaders to false we can swi behaviour. This should be used together w default spring security headers see https:/ security/site/docs/current/reference/html/l security-headers
zuul.ignored-headers		Names of HTTP headers to ignore complet of downstream requests and drop them fr responses).
zuul.ignored-patterns		
zuul.ignored-services		Set of service names not to consider for pr By default all services in the discovery clie
zuul.prefix		A common prefix for all routes.
zuul.remove-semicolon-content	true	Flag to say that path elements past the firs dropped.
zuul.retryable		Flag for whether retry is supported by def routes themselves support it).
zuul.ribbon-isolation-strategy		
zuul.routes		Map of route names to properties.
zuul.s-e-c-u-r-i-t-y-h-e-a-d-e-r-s		Headers that are generally expected to be Security, and hence often duplicated if the backend are secured with Spring. By defau the ignored headers if Spring Security is p ignoreSecurityHeaders = true.
zuul.semaphore.max-semaphores	100	The maximum number of total semaphor
zuul.sensitive-headers		List of sensitive headers that are not passe requests. Defaults to a "safe" set of header contain user credentials. It's OK to remove the downstream service is part of the sam so they are sharing authentication data. If outside your own domain, then generally to leak user credentials.

Name	Default	Description
zuul.servlet-path	/zuul	Path to install Zuul as a servlet (not part o servlet is more memory efficient for requ e.g. file uploads.
zuul.ssl-hostname-validation-enabled	true	Flag to say whether the hostname for ssl c verified or not. Default is true. This should setups!
zuul.strip-prefix	true	Flag saying whether to strip the prefix fro forwarding.
zuul.trace-request-body	true	Flag to say that request bodies can be trac