Cloud Native Applications

Table of Contents

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

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

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

- Java 6 JCE
- Java 7 JCE
- Java 8 JCE

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

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.

1. 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.

1.1 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).

1.2 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.

1.3 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.

1.4 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.

1.5 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").

> 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.

1.6 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"));
    }

}
```

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.

1.7 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).

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`.

1.8 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).

`@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`).

1.9 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
- Java 7 JCE
- Java 8 JCE

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

1.10 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`)

2. 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).

2.1 @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, Spring Cloud Consul Discovery and Spring Cloud Zookeeper Discovery.

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`.

The use of `@EnableDiscoveryClient` is no longer required. It is enough to just have a `DiscoveryClient` implementation on the classpath to cause the Spring Boot application to register with the service discovery server.

2.1.1 Health Indicator

Commons creates a Spring Boot `HealthIndicator` that `DiscoveryClient` implementations can participate in by implementing `DiscoveryHealthIndicator`. To disable the composite `HealthIndicator` set `spring.cloud.discovery.client.composite-indicator.enabled=false`. A generic `HealthIndicator` based on `DiscoveryClient` is auto-configured (`DiscoveryClientHealthIndicator). To disable it, set `spring.cloud.discovery.client.health-indicator.enabled=false`. To disable the description field of the `DiscoveryClientHealthIndicator` set `spring.cloud.discovery.client.health-indicator.include-description=false`, otherwise it can bubble up as the `description` of the rolled up `HealthIndicator`.

2.2 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.

```
@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.

2.2.1 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.

2.2.2 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.

2.3 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.

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

```
@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 for details of how the `RestTemplate` is set up.

2.3.1 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 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 for a description of what there properties do.

If you would like to implement a `BackOffPolicy` in your retries you will need to create a bean of type `LoadBalancedBackOffPolicyFactory`, and return the `BackOffPolicy` you would like to use for a given service.

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedBackOffPolicyFactory backOffPolciyFactory() {
        return new LoadBalancedBackOffPolicyFactory() {
            @Override
            public BackOffPolicy createBackOffPolicy(String service) {
                    return new ExponentialBackOffPolicy();
                }
        };
    }
}
```

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

2.3.2 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.

```
@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);
    }
}
```

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

2.4 Spring WebFlux WebClient as a Load Balancer Client

`WebClient` can be configured to use the `LoadBalancerClient. A `LoadBalancerExchangeFilterFunction` is auto-configured if spring-webflux is on the classpath.

```
public class MyClass {
    @Autowired
    private LoadBalancerExchangeFilterFunction lbFunction;

    public Mono<String> doOtherStuff() {
        return WebClient.builder().baseUrl("http://stores")
            .filter(lbFunction)
            .build()
            .get()
            .uri("/stores")
            .retrieve()
            .bodyToMono(String.class);
    }
}
```

The URI needs to use a virtual host name (ie. service name, not a host name). The `LoadBalancerClient` is used to create a full physical address.

2.5 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() for more details what is site local address.

application.yml.

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

2.6 HTTP Client Factories

Spring Cloud Commons provides beans for creating both Apache HTTP clients (`ApacheHttpClientFactory`) as well as OK HTTP clients (`OkHttpClientFactory`). The `OkHttpClientFactory` bean will only be created if the OK HTTP jar is on the classpath. In addition, Spring Cloud Commons provides beans for creating the connection managers used by both clients, `ApacheHttpClientConnectionManagerFactory` for the Apache HTTP client and `OkHttpClientConnectionPoolFactory` for the OK HTTP client. You can provide your own implementation of these beans if you would like to customize how the HTTP clients are created in downstream projects. You can also disable the creation of these beans by setting `spring.cloud.httpclientfactories.apache.enabled` or `spring.cloud.httpclientfactories.ok.enabled` to `false`.