8

Spring Cloud

Jorge Simão, Ph.D.

Content

Spring Cloud Overview

Spring Cloud is a collection of related projects within the **Spring Framework** ecosystem whose purpose is to simplify application development in cloud environments, including micro-services architectures.

Spring Cloud projects build of top of **Spring Boot**. Consequently, *dependency management* is commonly done via a **Spring Boot** parent and starter dependencies. Modifications to the preferred version numbers for **Spring Cloud** is done using the dependency-management mechanisms of the selected build tool.

» Example: Spring Boot/Cloud Maven POM Base

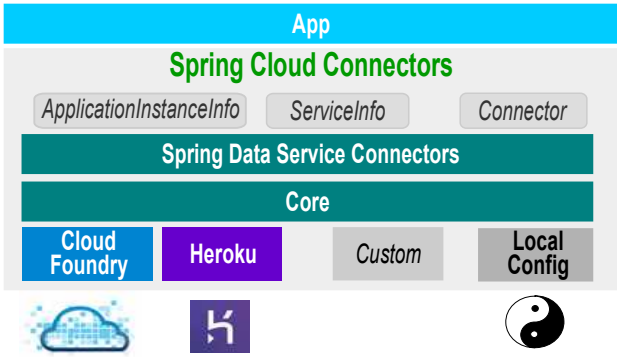
```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.0.RC1</version>
</parent>
```

Spring Cloud Connectors

Spring Cloud Connectors is a library (set) that provides a uniform API for cloud applications to obtain information about services bound to it – such as databases and messages brokers – simplify and automate the creation of connectors (driver wrappers) for those services, and obtain general meta-data about application instances. The motivating goal of the library is to allow application to achieve *cloud portability*. That is, the same application can run unchanged in different cloud platforms. Seemly transition from local development and testing environments – i.e. in the developer's laptop or a *continuous integration* server – to the cloud is also supported (*production-development parity principle*).

The core library is designed from the ground-up to be extensible and support multiple cloud platforms (PasS), using a ***CloudConnector*** abstraction to encapsulate all cloud platform specific behaviour. The project official distribution on **GitHub** includes connectors for **Cloud Foundry** and **Heroku**. Local development and testing is supported with the a local configuration connector.

Figure below depicts the architecture of **Spring Cloud Connectors**.



The **Cloud** abstraction offers most of the API, including access to service descriptors (**ServiceInfo**), application instance meta-data (**ApplicationInstanceInfo**), and creation of service connectors. A **Cloud** object is a wrapper around the low-level **CloudConnector**, and is created by a singleton **CloudFactory** that uses a auto-discovery mechanism to automatically infer the cloud environment where the application is running. Connector objects are specific to each type of service – e.g. a JDBC **DataSource** for relational DB access, and **ConnectionFactory** from **Spring AMQP**. **Spring Data** provided abstractions are used to connect to **NoSql** DBs.

» Example: Creating a Cloud Object

```
CloudFactory cloudFactory = new CloudFactory();
Cloud cloud = cloudFactory.getCloud();
```

» Example: Looking up a ServiceInfo

```
ServiceInfo dbInfo = cloud.getServiceInfo("mydb");
String url = dbInfo.getUrl();
```

» Example: Creating a Connector

```
DataSource dataSource =
  cloud.getServiceConnector("mydb",
DataSource.class, null);
```

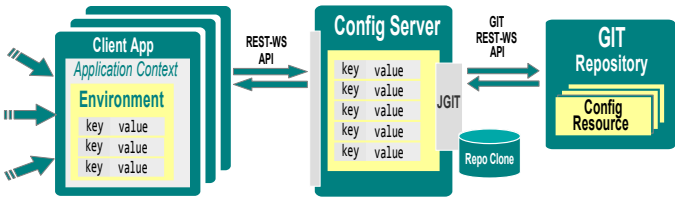
» Example: Cloud Connectors Maven Dependencies

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

Distributed Configuration

Deploying multiple instances of the same application is a common technique to achieve scalability and high-availability in cloud applications. Project functionality is also often split into multiple semi-independent applications – as in a **micro-service architecture**. In both scenarios, application configuration can be challenging since it involves reconfiguration of many systems at the same system. Rather than modify application settings individually (e.g. in environment variables or configuration files), a preferred approach is to have application instances to "download" at deployment time their settings from a configuration service.

Spring Cloud Config project provides an out-of-the-box solution for *distributed configuration*. This include provisioning of a **Config Server** (as a **Spring Boot** app) that exports application common or specific configuration settings through a REST API. The actual configuration settings are stored and managed in one (or more) **Git** repositories (accessed internally using the **JGit** library). Settings retrieved by client applications are merged with other settings into the **Environment** abstraction managed by a **Spring ApplicationContext**. Thus, configuration of application components (*beans*) by dependency-injection works unchanged in respect of the source of the settings. Figure below depicts the architecture for **Spring Cloud Config**.



A *Config Server* is setup as a **Spring Boot** app simply by decorating one its **@Configuration** classes which the annotation **@EnableConfigServer**. The environment property `spring.cloud.config.server.git.uri` is set with the URL of the GIT repository where the configuration resources are stored. A local repository can also be used (e.g. for local development or testing purposes) with URL prefix **file://**.

» Example: Starting an Embedded Config Server

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

» Example: Settings for GIT Repository in Config Server

```
spring.cloud.config.server.git.uri:
https://github.com/myuser/myproject-config
```

```
server.port=8888

spring.cloud.config.server.git.uri:
file://myproject/config-repo #dev/testing
```

» Example: GIT Repository Layout

```
myproject-config/
  application.properties
  application.yml
  myapp.properties
  myapp.yml
  myapp-dev.properties
  myapp-prod.properties
  myotherapp.properties
```

» Example: ConfigServer Maven Dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Files stored in the **GIT** repository follows naming conventions similar to **Spring Boot**. Settings common to all applications and all profiles should be stored in a file named **application.properties** (or **.yml** or **.yaml**). Profile specific setting should be stored in files named **appplication-*profile*.properties** (or **.yml**). Settings specific to a particular application with name ***appname*** should be store in files named **appname.properties** and **appname-*profile*.properties** (or YAML). Application and profile specific settings take precedence over common settings.

The **ConfigServer** exposes a REST-API with several endpoints for applications to retrieve settings – e.g. endpoint `GET /{application}/{profiles}`, retrieves application and profile settings combined with the common settings.

» Example: GIT Repository Layout

```
myproject-config/
  application.properties
  application.yml
  myapp.properties
  myapp.yml
  myapp-dev.properties
  myapp-prod.properties
  myotherapp.properties
```

» Example: Config Server REST-WS API

```
curl localhost:8888/myapp/default
```

```
{"name":"myapp","profiles":["default"],
"label":"master", "propertySources":
[{"name":"file:///myproject-
config/application.properties", "source":
{"Greeting":"Hello Cloud!"}}]}
```

Client applications for the **ConfigServer** define the coordinates of the server in file **bootstrap.properties** (or **.yml**) as property

`spring.cloud.config.uri`. Application should also set their name as property `spring.application.name` and the profiles as `spring.application.profiles`.

» Example: Setup of Client App for Config Server

```
#bootstrap.properties
spring.cloud.config.uri: http://config.mydomain.io:8888
spring.application.name: myapp
spring.application.profiles: dev
```

» Example: Sample App using Remote Settings

```
@RestController
@SpringBootApplication
public class MyApp {
  @Value("${greeting}")
  private String greeting;

  @RequestMapping("/")
  public String home() {
    return "Config Server says: " + greeting;
  }

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

» Example: Client for ConfigServer Maven Dependency

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

Dynamic Configuration

Component configuration (even) when done with the settings fetched from **Spring Cloud Config** server is done only when then bean is created – e.g. at startup for a *singleton* bean. Often, it is desirable to be able to dynamically reconfigure beans without having to restart the application. **Spring Cloud** project extends **Spring Boot Actuator** with additional endpoints to trigger dynamic configuration. Endpoint `POST /env` triggers properties in classes annotated with **@ConfigurationProperties** to be reinitialized, and logging levels to be updated (from `logging.level.*` properties) from local configuration sources (i.e. local configuration files). This mechanism requires beans to actively fetch the properties values from classes annotated with **@ConfigurationProperties**.

For automatic reconfiguration, beans can be annotated with **@RefreshScope**. This makes the beans to be wrapped in a *proxy* that delegates to a dynamic bean instance. Beans cached in this scope can be invalidated by calling **RefreshScope .refreshAll()**. Endpoint `POST /refresh` forces environment settings to be retrieved from the **Config Server**, and trigger the invalidation of **RefreshScope** beans. This in turn makes new bean instances to be initialized with the new values of the environment properties.

When an application is deployed with multiple instances it is convenient to refresh all application instances quasi-simultaneously with a single `POST /refresh`. This can be accomplish by importing project **Spring Cloud Bus**. This makes refresh events to be multi-casted trough a message broker (e.g. **RabbitMQ**/**AMQP**), and propagated to other instances. To connect to the broken, a RabbitMQ **ConnectionFactory** is picked up from the **ApplicationContext** or auto-configured if none is found.

» Example: Refreshing Environment & Beans

```
curl -X POST localhost:8080/refresh
curl localhost:8080/refresh -d ''
```

» Example: RabbitMQ Settings for Spring Boot & Bus

```
spring.rabbitmq.uri= amqp://joe:jopass@localhost:5672/
```

» Example: Spring Cloud Bus Maven Dependency

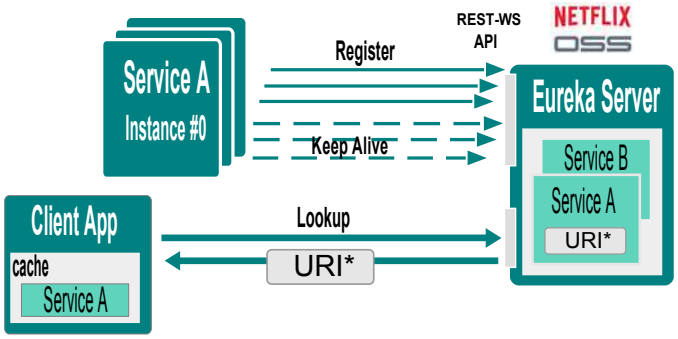
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-rabbitmq</artifactId>
</dependency>
```

Service Discovery – NetFlix Eureka

Micro-service architectures are characterized by multiple semi-independent services working together. A common pattern is to have *gateway* service or UI to perform the integration across the different services, although more complex solutions are possible such as multiple-layer distributed hierarchies and "lateral" communication between services . A pre-condition for these interactions to occur, is for services to have a way to discover each other locations – defined as physical IPs or DNS host-names, and TCP/IP ports. A common approach to accomplish this is by using a *naming service* with an API for service registration and lookup/discovery. **Spring Cloud** provides libraries that integrate with several naming services, of which **Netflix Eureka** was the first to be supported.

Netflix Eureka naming service allow multiple service instances to be registered per service name – which makes it particularly well suitable for replicated services as is done in cloud computing. When looking up a service, clients retrieve the full

list of instance locations for the requested service name. Registrations are also kept updated and "live" by having the service instance to periodically advertise their status to thee **Eureka** server. Multiple **Eureka** server instances can be started for high-availability and load-balancing. Instances can be configured to cross register themselves and to download other instances in-memory registry – thus making service location information to be disseminated "epidemically". Figure below depicts the architecture for service registration with **Eureka**.



Annotation **@EnableEurekaServer** is used to start an embedded **Eureka** server in a **Spring Boot** application. **Eureka** instances can be configured like clients to other instances for cross-registration. Otherwise, for standalone operation the environment variables `eureka.client.registerWithEureka` and `eureka.client.fetchRegistry` should be set to `false`.

» Example: Starting an Embedded Eureka Server

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

» Example: Settings for Standalone Eureka Server

```
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

» Example: EurekaServer Maven Dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-eureka-server</artifactId>
</dependency>
```

Registering a **Spring Boot** application as service in **Eureka** is also straightforward – annotation **@EnableEurekaClient** triggers this behaviour declaratively. Environment variable `eureka.client.serviceUrl.defaultZone` specifies the location of one (or more) Eureka server(s) where registration should take place. By default applications are registered under the name `${spring.application.name}`, instance ID `${spring.cloud.client.hostname}:${spring.application.name}:${server.port}`, and location `${spring.cloud.client.hostname}:${server.port}`. This settings can be modified with variables `eureka.instance.{appname, instance_id, hostname, nonSecuredPort}`.

» Example: Registering Boot App as Service in Eureka

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

» Example: Settings for Client to EurekaServer

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

```
eureka:
  instance:
    appname: myservice
    instanceid: ${random.value}
```

» Example: Client Settings – CloudFoundry Routing

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

» Example: Settings – Direct Interaction in CloudFoundry

```
eureka:
  instance:
    hostname: ${CF_INSTANCE_IP}
    nonSecurePort: ${CF_INSTANCE_PORT}
```

» Example: Client for EurekaServer Maven Dependency

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

Service lookup is simplified with annotation **@EnableDiscoveryClient** which makes an instance of **DiscoveryClient** to be automatically created as a managed component. The API of **DiscoveryClient** allows retrieval of service information from an **Eureka** server. The configuration to connect to **Eureka** is the same as for registration of a service.

» Example: Injecting a DiscoveryClient

```
@EnableDiscoveryClient
@SpringBootApplication
public class MyClient {
  @Autowired
  private DiscoveryClient discoveryClient;

  public List<Map<?,?>> getProducts() {
     ServiceInstance instance = chooseOne(
         discoveryClient.getInstances("PRODUCTS"));
    String uri = instance.getUri().toString()
    return new RestTemplate().getForObject(uri, List.class);
  }
  …
}
```

Service Discovery – Consul & ZooKeeper

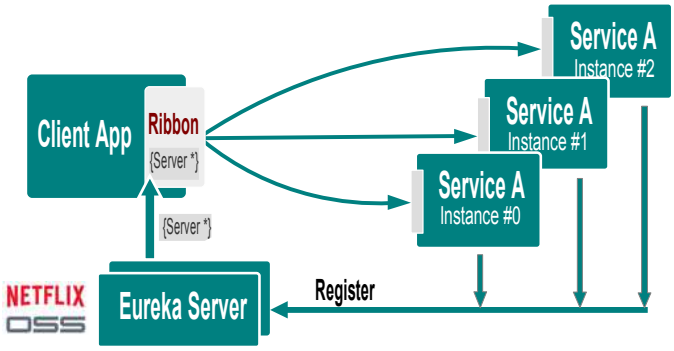
In addition to integrate with **Netflix Eureka**, **Spring Cloud** also integrates with other naming services such as **Hashicorp Consul** and **ZooKeeper** (project part of the **Hadoop** ecosystem). An object of type **DiscoveryClient** is auto-configured if annotation **@EnableDiscoveryClient** is used – thus, following the same programming model as with **Eureka**. Importing the **Spring Cloud Consul** (or **ZooKeeper**) client libraries in the *classpath* is the triggering condition for the server to be contacted. The **Consul** server will be contacted by default in location `localhost:8500`, but this can modified by setting properties **spring.cloud.consul.{host,port}.** **Zookeeper** server will be contacted in the default location `localhost:2181`, or alternatively the location specified in property **zookeeper.connectString**.

» Example: ZooKeeper Discovery Maven Dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-zookeeper-discovery</artifactId>
</dependency>
```

Client-Side Load-Balancing

When a service is deployed with multiple instances – as is common in cloud computing, requests need to be distributed (*load-balanced*) across the instances using some criteria such as round-robin or random. The load-balancing can be done by client applications or by a dedicated load-balancing component (e.g. the Router of a PaaS such as **CloudFoundry**). **Ribbon** is a **Netflix** library to perform *client-side load-balancing*. It integrates with **Netflix Eureka** server to retrieve the list of instances for registered services. Figure below depicts the architecture for **Ribbon**.



Spring Cloud Netflix Ribbon provides a convenient programming model to simplify the use of **Ribbon**. Annotation **@RibbonClient** enables the set up and configuration of **Ribbon**. The default load-balancing criteria (distance based), and other setting can be reconfigured (e.g. defining a bean of type **IPing** modifies the strategy to detect if an instance is reachable). A bean of type **LoadBalancerClient** is auto-configured, and Its API can be used to retrieve the selected single instance for a named service.

» Example: Ribbon Configuration

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

» Example: Consume Load-balanced REST-WS

```
@Service
public class MyClient {
  @Autowired
  private LoadBalancerClient loadBalancer;

  public static List<Map<?,?>> getProducts() {
    ServiceInstance instance = loadBalancer.choose("products");
    String uri = String.format("http://%s:%s/product",
      instance.getHost(), instance.getPort());
    return new RestTemplate().getForObject(uri);
  }
}
```

The location of service instances can be obtained automatically from an Eureka server if the dependency for **eureka-client** is in the *classpath*, and setting **`eureka.client.serviceUrl.defaultZone`** is defined accordingly. Alternatively, **Ribbon** supports manual configuration and integration with **Spring Cloud Config Server**, by defining environment properties named ***<<service-***

***name*>>.ribbon.listOfServer**, with a comma-separated list of URIs (or hostnames) for the service instances as value.

» Example: Manual Service Instance Config (w/o Eureka)

```
products:
  ribbon:
    listOfServers: p0.mysite.io, p1.mysite.io
ribbon:
  eureka:
   enabled: false
```

» Example: Ribbon Maven Dependency

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

Spring Cloud Netflix Eureka and **Ribbon** also integrate, by auto-configuring a **RestTemplate** which is configured internally with an **HttpRequestFactory** that accepts and automatically resolves *logical service names* in URIs. This allows load-balancing to occur automatically without dealing directly with the **LoadBalancerClient** API.

» Example: Access REST-WS RestTemplate – Logical URI

```
@EnableDiscoveryClient
@Configuration
public class RemoteBookManager implements BookManager {
  @Autowired
  private RestTemplate template;

  public List<Map<?,?>> getProducts() {
    return template.getForObject(
                "http://BOOKS/catalog", List.class);
  }
}
```

Declarative REST-WS

Netflix Feign is a library that creates REST-WS client classes automatically from the interface specifications. **Spring Cloud Netflix Feign** extends **Feign** to support **Spring MVC** annotations – like **@RequestMapping** – to map interface methods to REST-WS endpoints. The annotation **@FeignClient** marks interfaces for *proxy generation*. When integrated with **Ribbon** (i.e. **Ribbon** is on *classpath*), the **value()** attribute specifies the logical name of the service to consume. Alternatively, a physical location of a service instance can be specific with attribute **url()**.

» Example: REST-WS Interface w/ Spring MVC Mappings

```
@FeignClient("books")
public interface BookManager {
  @RequestMapping(value="/book",  method=GET)
  List<Map> getBooks();

  @RequestMapping(value="/book", method=POST)
  void newBook(Map<String,Object> book);
}
```

» Example: Hard-Wired REST-WS Location

```
@FeignClient(url="http://books.bookjungle.org")
public interface BooksManager { … }
```

» Example: Feign Maven Dependency

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

Annotation **@EnableFeignClients** enables the declarative creation of the the proxys in a **Spring Boot**/**Spring Cloud** project, which are defined as **Spring** managed components – i.e. added to the **ApplicationContext** and available for dependency-injection in other components (beans).

» Example: Injecting Proxy for REST-WS Client

```
@EnableFeignClients
@SpringBootApplication
public class BooksWebApp { … }
```

» Example: Injecting Proxy for REST-WS Client

```
@Controller
public class BookController {
   @Autowired
   private BookManager manager;

   @RequestMapping(value="/catalog", method=GET)
   public static String catalog(Model model) {
      model.addAttribute("books", manager.getBooks();
      return "book/catalog";
   }
}
```

A dedicated routing component can also created by integration with **Netflix Zuul** project. Annotation **@EnableZuulProxy** creates an embedded **Zuul** server instance, that maps and splits URL namespace across services registered in Eureka – e.g. /products and /users will be mapped and routed to intances of service PRODUCTS an USERS. This is specially useful to simplify the creation of API gateways.

Reliability with Circuit-Breakers

When invoking a remote service that might be faulty, it is often preferred to not make the invocation and fallback to alternative strategy than to wait until the service comes back. The *circuit-*

breaker concept – as implemented in the **Netflix Hystrix** library – captures this idea by defining a state-machine whose state determine if/when a possibly faulty operation should be attempted. **Hystrix** project integrates seemly with **Spring**, by using **Spring AOP** to wrap beans into proxies that "protect" methods with circuit-breakers. Annotation **@HystrixCommand** marks and configures methods to be made resilient by a circuit-breaker. Attribute **fallbackMethod** is used to specify an alternative method/strategy to call when the method is invoked, but the circuit-breaker is currently open. Detailed configuration of the properties that control the circuit-breaker is defined in attribute **commandProperties()** having as value annotation **@HystrixProperty**.

» Example: Using Circuit-Breaker in REST-WS Invocation

```
@HystrixCommand(fallbackMethod="getBookFallback")
@RequestMapping(method = RequestMethod.GET)
public Map getBook(Long bookId) {
  return restTemplate.getForObject(bookUri, Map.class, bookId);
}

public Map[] getBookFallback(Long bookId) { return ...; }
```

Hystrix project includes a *dashboard* (web UI) that can be used to monitor the state and metrics of the circuit-breakers in an app, reachable on endpoint **/hystrix**. The state of individual circuit-breakers can also be captured in REST endpoint **/hystrix.stream**. Annotation **@EnableHystrixDashboard** is used in a **Spring Cloud** project to enable the **Hystrix** dashboard endpoints. Project **Turbine** can also be used to aggregate the metrics from multiple instances of a service (or several services) looked-up in an Eureka server.

Cloud Monitoring and Tracing

Spring Cloud Sleuth is a project that integrates with the distributed tracing tool **Zipkin** (inspired in google project **Dapper**). **Sleuth** & **Zipkin** captures detailed information of the locations a distributed request reaches – directly and by cascading of requests. Each direct interaction between services is designated a *span*, and the aggregate tree of cascaded spans produced from a single initial request a *trace*. Importing the **Spring Cloud Sleuth** dependency in a Spring boot project make tracing information be registered in log files, and it is also registered in the **Zipkin** server for dashboard inspection.

Resources

- Spring Cloud Project home page: http://projects.spring.io/spring-cloud/
- Spring Cloud Reference Manual: https://raw.githubusercontent.com/antirez/redis/3.2/redis.conf
- Spring Cloud on GitHub: https://raw.githubusercontent.com/antirez/redis/3.2/redis.conf

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consultancy) at **EInnovator**. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Spring Micro-Services Training

The **Spring Micro-Services** course offers in-depth study and hand-ons working experience using **Spring Boot** and **Spring Cloud** to build *cloud-native applications* and micro-service architectures. Applications are developed and tested in a local environment, and deployed into PWS Cloud Foundry environment.
Book for a training event in a date&location of your choice:
www.einnovator.org/course/spring-microservices

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org