(/)

Spring Cloud – Bootstrapping

Last modified: March 28, 2019

by Tim Schimandle (https://www.baeldung.com/author/timothy-schimandle/)

Spring Cloud (https://www.baeldung.com/category/spring/spring-cloud/)

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-start)

1. Overview

Spring Cloud is a framework for building robust cloud applications. The framework facilitates the development of applications by providing solutions to many of the common problems faced when moving to a distributed environment.

Applications that run with microservices architecture aim to simplify development, deployment, and maintenance. The decomposed nature of the application allows developers to focus on one problem at a time. Improvements can be introduced without impacting other parts of a system.

On the other hand, different challenges arise when we take on a microservice approach:

- Externalizing configuration so that is flexible and does not require rebuild of the service on change
- Service discovery
- Hiding complexity of services deployed on different hosts

In this article, we will build five microservices: a configuration server, a discovery server, a gateway server, a book service, and finally a rating service. These five microservices form a solid base application to begin cloud development and address the aforementioned challenges.

2. Config Server

When developing a cloud application, one issue is maintaining and distributing configuration to our services. We really don't want to spend time configuring each environment before scaling our service horizontally or risk security breaches by baking our configuration into our application.

To solve this, we will consolidate all of our configuration into a single Git repository and connect that to one application that manages a configuration for all our applications. We are going to be setting up a very simple implementation.

To learn more details and see a more complex example take a look at Spring Cloud Configuration (/spring-cloud-configuration) article.

2.1. Setup

Navigate to start.spring.io (http://start.spring.io) and select Maven and Spring Boot 1.4.x.

Set the artifact to "*config*". In the dependencies section, search for "*config server*" and add that module. Then press the *generate* button and you will be able to download a zip file with a preconfigured project inside and ready to go.

Alternatively, we can generate a *Spring Boot* project and add some dependencies to the POM file manually.

These dependencies will be shared between all the projects:

```xml
 1  <parent>
 2      <groupId>org.springframework.boot</groupId>
 3      <artifactId>spring-boot-starter-parent</artifactId>
 4      <version>1.4.0.RELEASE</version>
 5      <relativePath/>
 6  </parent>
 7
 8  <dependencies>
 9      <dependency>
10          <groupId>org.springframework.boot</groupId>
11          <artifactId>spring-boot-starter-test</artifactId>
12          <scope>test</scope>
13      </dependency>
14  </dependencies>
15
16  <dependencyManagement>
17      <dependencies>
18          <dependency>
19              <groupId>org.springframework.cloud</groupId>
20              <artifactId>spring-cloud-dependencies</artifactId>
21              <version>Brixton.SR5</version>
22              <type>pom</type>
23              <scope>import</scope>
24          </dependency>
25      </dependencies>
26  </dependencyManagement>
27
28  <build>
29      <plugins>
30          <plugin>
31              <groupId>org.springframework.boot</groupId>
32              <artifactId>spring-boot-maven-plugin</artifactId>
33          </plugin>
34      </plugins>
35  </build>
```

Let's add a dependency for the config server:

```
1   <dependency>
2       <groupId>org.springframework.cloud</groupId>
3       <artifactId>spring-cloud-config-server</artifactId>
4   </dependency>
```

For reference: you can find the latest version on *Maven Central* (spring-cloud-dependencies
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%2
0AND%20a%3A%22spring-cloud-dependencies%22), test
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20
AND%20a%3A%22spring-boot-starter-test%22), config-server
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%2
0AND%20a%3A%22spring-cloud-config-server%22)).

2.2. Spring Config

To enable the configuration server we must add some annotations to the main application class:

```
1   @SpringBootApplication
2   @EnableConfigServer
3   public class ConfigApplication {...}
```

@EnableConfigServer will turn our application into a configuration server.

2.3. Properties

Let's add the *application.properties* in src/main/resources:

```
1   server.port=8081
2   spring.application.name=config
3
4   spring.cloud.config.server.git.uri=file:// (file://)${user.home}/application-config
```

The most significant setting for the config server is the *git.uri* parameter. This is currently set to a relative file path that generally resolves to *c:\Users\[username]* on Windows or */Users/[username]/* on *nix. This property points to a Git repository where the property files for all the other applications are stored. It can be set to an absolute file path if necessary.

Tip: On a windows machine preface the value with 'file:///', on *nix then use 'file://'.

2.4. Git Repository

Navigate to the folder defined by *spring.cloud.config.server.git.uri* and add the folder 'application-config'. CD into that folder and type *git init*. This will initialize a Git repository where we can store files and track their changes.

2.5. Run

Let's run config server and make sure it is working. From the command line type *mvn spring-boot:run*. This will start the server. You should see this output indicating the server is running:

```
1   Tomcat started on port(s): 8081 (http)
```

2.6. Bootstrapping Configuration

In our subsequent servers, we're going to want their application properties managed by this config server. To do that, we'll actually need to do a bit of chicken-and-egg: **Configure properties in each application that know how to talk back to this server.**

It's a bootstrap process, and **each one of these apps is going to have a file called *bootstrap.properties*.** It will contain properties just like *application.properties* but with a twist:

A parent Spring *ApplicationContext* **loads the *bootstrap.properties* first.** This is critical so that Config Server can start managing the properties in *application.properties*. It's this special *ApplicationContext* that will also decrypt any encrypted application properties.

It's smart to keep these properties files distinct. *bootstrap.properties* is for getting the config server ready, and *application.properties* is for properties specific to our application. Technically, though, it's possible to place application properties in *bootstrap.properties*.

Lastly, since Config Server is managing our application properties, one might wonder why have an *application.properties* at all? The answer is that these still come in handy as default values that perhaps Config Server doesn't have.

3. Discovery

Now that we have configuration taken care of, we need a way for all of our servers to be able to find each other. We will solve this problem by setting the *Eureka* discovery server up. Since our applications could be running on any ip/port combination we need a central address registry that can serve as an application address lookup.

When a new server is provisioned it will communicate with the discovery server and register its address so that others can communicate with it. This way other applications can consume this information as they make requests.

To learn more details and see a more complex discovery implementation take a look at Spring Cloud Eureka article (/spring-cloud-netflix-eureka).

3.1. Setup

Again we'll navigate to start.spring.io (http://start.spring.io/). Set the artifact to 'discovery'. Search for "*eureka server*" and add that dependency. Search for "*config client*" and add that dependency. Generate the project.

Alternatively, we can create a *Spring Boot* project, copy the contents of the *POM* from config server and swap in these dependencies:

```
1  <dependency>
2      <groupId>org.springframework.cloud</groupId>
3      <artifactId>spring-cloud-starter-config</artifactId>
4  </dependency>
5  <dependency>
6      <groupId>org.springframework.cloud</groupId>
7      <artifactId>spring-cloud-starter-eureka-server</artifactId>
8  </dependency>
```

For reference: you will find the bundles on *Maven Central* (config-client (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-config%22), eureka-server (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-eureka-server%22)).

3.2. Spring Config

Let's add Java config to the main class:

```
1  @SpringBootApplication
2  @EnableEurekaServer
3  public class DiscoveryApplication {...}
```

@EnableEurekaServer will configure this server as a discovery server using *Netflix Eureka*. *Spring Boot* will automatically detect the configuration dependency on the classpath and lookup the configuration from the config server.

3.3. Properties

Now we will add two properties files:

bootstrap.properties in src/main/resources

```
1  spring.cloud.config.name=discovery
2  spring.cloud.config.uri=http://localhost:8081 (http://localhost:8081)
```

These properties will let discovery server query the config server at startup.

discovery.properties in our Git repository

```
1  spring.application.name=discovery
2  server.port=8082
3
4  eureka.instance.hostname=localhost
5
6  eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/ (http://localhost:8082/eureka/
7  eureka.client.register-with-eureka=false
8  eureka.client.fetch-registry=false
```

The filename must match the *spring.application.name* property.

In addition, we are telling this server that it is operating in the default zone, this matches the config client's region setting. We are also telling the server not to register with another discovery instance.

In production, you would have more than one of these to provide redundancy in the event of failure and that setting would be true.

Let's commit the file to the Git repository. Otherwise, the file will not be detected.

3.4. Add Dependency to the Config Server

Add this dependency to the config server POM file:

```
1   <dependency>
2       <groupId>org.springframework.cloud</groupId>
3       <artifactId>spring-cloud-starter-eureka</artifactId>
4   </dependency>
```

For reference: you will find the bundle on *Maven Central* (eureka-client (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%2 0AND%20a%3A%22spring-cloud-starter-eureka%22)).

Add these properties to the *application.properties* file in src/main/resources of the config server:

```
1   eureka.client.region = default
2   eureka.client.registryFetchIntervalSeconds = 5
3   eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/ (http://localhost:8082/eureka/
```

3.5. Run

Start the discovery server using the same command, *mvn spring-boot:run*. The output from the command line should include:

```
1   Fetching config from server at: http://localhost:8081 (http://localhost:8081)
2   ...
3   Tomcat started on port(s): 8082 (http)
```

Stop and rerun the config service. If all is good output should look like:

```
1   DiscoveryClient_CONFIG/10.1.10.235:config:8081: registering service...
2   Tomcat started on port(s): 8081 (http)
3   DiscoveryClient_CONFIG/10.1.10.235:config:8081 - registration status: 204
```

4. Gateway

Now that we have our configuration and discovery issues resolved we still have a problem with clients accessing all of our applications.

If we leave everything in a distributed system, then we will have to manage complex CORS headers to allow cross-origin requests on clients. We can resolve this by creating a *Gateway* server. This will act as a reverse proxy shuttling requests from clients to our back end servers.

A gateway server is an excellent application in microservice architecture as it allows all responses to originate from a single host. This will eliminate the need for CORS and give us a convenient place to handle common problems like authentication.

4.1. Setup

By now we know the drill. Navigate to start.spring.io (http://start.spring.io/). Set the artifact to '*gateway*'. Search for "zuul" and add that dependency. Search for "*config client*" and add that dependency. Search for "*eureka discovery*" and add that dependency. Generate that project.

Alternatively, we could create a *Spring Boot* app with these dependencies:

```
1  <dependency>
2      <groupId>org.springframework.cloud</groupId>
3      <artifactId>spring-cloud-starter-config</artifactId>
4  </dependency>
5  <dependency>
6      <groupId>org.springframework.cloud</groupId>
7      <artifactId>spring-cloud-starter-eureka</artifactId>
8  </dependency>
9  <dependency>
10     <groupId>org.springframework.cloud</groupId>
11     <artifactId>spring-cloud-starter-zuul</artifactId>
12 </dependency>
```

For reference: you will find the bundle on *Maven Central* (config-client (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-config%22), eureka-client (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-eureka%22), zuul (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-zuul%22)).

4.2. Spring Config

Let's add the configuration to the main class:

```
1    @SpringBootApplication
2    @EnableZuulProxy
3    @EnableEurekaClient
4    public class GatewayApplication {...}
```

4.3. Properties

Now we will add two properties files:

bootstrap.properties in src/main/resources

```
1    spring.cloud.config.name=gateway
2    spring.cloud.config.discovery.service-id=config
3    spring.cloud.config.discovery.enabled=true
4
5    eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/ (http://localhost:8082/eureka/
```

gateway.properties in our Git repository

```
 1   spring.application.name=gateway
 2   server.port=8080
 3
 4   eureka.client.region = default
 5   eureka.client.registryFetchIntervalSeconds = 5
 6
 7   zuul.routes.book-service.path=/book-service/**
 8   zuul.routes.book-service.sensitive-headers=Set-Cookie,Authorization
 9   hystrix.command.book-service.execution.isolation.thread.timeoutInMilliseconds=600000
10
11   zuul.routes.rating-service.path=/rating-service/**
12   zuul.routes.rating-service.sensitive-headers=Set-Cookie,Authorization
13   hystrix.command.rating-service.execution.isolation.thread.timeoutInMilliseconds=600000
14
15   zuul.routes.discovery.path=/discovery/**
16   zuul.routes.discovery.sensitive-headers=Set-Cookie,Authorization
17   zuul.routes.discovery.url=http://localhost:8082 (http://localhost:8082)
18   hystrix.command.discovery.execution.isolation.thread.timeoutInMilliseconds=600000
```

The *zuul.routes* property allows us to define an application to route certain requests based on an ant URL matcher. Our property tells *Zuul* to route any request that comes in on '/book-service/**' to an application with the *spring.application.name* of 'book-service'. *Zuul* will then lookup the host from discovery server using the application name and forward the request to that server.

Remember to commit the changes in the repository!

4.4. Run

Run the config and discovery applications and wait until the config application has registered with the discovery server. If they are already running you do not have to restart them. Once that is complete, run the gateway server. The gateway server should start on port 8080 and register itself with the discovery server. The output from the console should contain:

```
1   Fetching config from server at: http://10.1.10.235:8081/ (http://10.1.10.235:8081/)
2   ...
3   DiscoveryClient_GATEWAY/10.1.10.235:gateway:8080: registering service...
4   DiscoveryClient_GATEWAY/10.1.10.235:gateway:8080 - registration status: 204
5   Tomcat started on port(s): 8080 (http)
```

One mistake that is easy to make is to start the server before config server has registered with *Eureka*. In this case, you will see a log with this output:

```
1   Fetching config from server at: http://localhost:8888 (http://localhost:8888)
```

This is the default URL and port for a config server and indicates our discovery service did not have an address when the configuration request was made. Just wait a few seconds and try again, once the config server has registered with Eureka, the problem will resolve.

5. Book Service

In microservice architecture, we are free to make as many applications to meet a business objective. Often engineers will divide their services by domain. We will follow this pattern and create a book service to handle all the operations for books in our application.

5.1. Setup

One more time. Navigate to start.spring.io (http://start.spring.io/). Set the artifact to 'book-service'. Search for "*web*" and add that dependency. Search for "*config client*" and add that dependency. Search for *"eureka discovery"* and add that dependency. Generate that project.

Alternatively, add these dependencies to a project:

```
 1   <dependency>
 2       <groupId>org.springframework.cloud</groupId>
 3       <artifactId>spring-cloud-starter-config</artifactId>
 4   </dependency>
 5   <dependency>
 6       <groupId>org.springframework.cloud</groupId>
 7       <artifactId>spring-cloud-starter-eureka</artifactId>
 8   </dependency>
 9   <dependency>
10       <groupId>org.springframework.boot</groupId>
11       <artifactId>spring-boot-starter-web</artifactId>
12   </dependency>
```

For reference: you will find the bundle on *Maven Central* (config-client
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%2
0AND%20a%3A%22spring-cloud-starter-config%22), eureka-client
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%2
0AND%20a%3A%22spring-cloud-starter-eureka%22), web
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20
AND%20a%3A%22spring-boot-starter-web%22)).

5.2. Spring Config

Let's modify our main class:

```
 1   @SpringBootApplication
 2   @EnableEurekaClient
 3   @RestController
 4   @RequestMapping("/books")
 5   public class BookServiceApplication {
 6       public static void main(String[] args) {
 7           SpringApplication.run(BookServiceApplication.class, args);
 8       }
 9
10       private List<Book> bookList = Arrays.asList(
11           new Book(1L, "Baeldung goes to the market", "Tim Schimandle"),
12           new Book(2L, "Baeldung goes to the park", "Slavisa")
13       );
14
15       @GetMapping("")
16       public List<Book> findAllBooks() {
17           return bookList;
18       }
19
20       @GetMapping("/{bookId}")
21       public Book findBook(@PathVariable Long bookId) {
22           return bookList.stream().filter(b -> b.getId().equals(bookId)).findFirst().orElse(null);
23       }
24   }
```

We also added a REST controller and a field set by our properties file to return a value we will set during configuration.

Let's now add the book POJO:

```
1  public class Book {
2      private Long id;
3      private String author;
4      private String title;
5
6      // standard getters and setters
7  }
```

5.3. Properties

Now we just need to add our two properties files:

bootstrap.properties in src/main/resources:

```
1  spring.cloud.config.name=book-service
2  spring.cloud.config.discovery.service-id=config
3  spring.cloud.config.discovery.enabled=true
4
5  eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/ (http://localhost:8082/eureka/
```

book-service.properties in our Git repository:

```
1  spring.application.name=book-service
2  server.port=8083
3
4  eureka.client.region = default
5  eureka.client.registryFetchIntervalSeconds = 5
6  eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/ (http://localhost:8082/eureka/
```

Let's commit the changes to the repository.

5.4. Run

Once all the other applications have started we can start the book service. The console output should look like:

```
1  DiscoveryClient_BOOK-SERVICE/10.1.10.235:book-service:8083: registering service...
2  DiscoveryClient_BOOK-SERVICE/10.1.10.235:book-service:8083 - registration status: 204
3  Tomcat started on port(s): 8083 (http)
```

Once it is up we can use our browser to access the endpoint we just created. Navigate to http://localhost:8080/book-service/books and we get back a JSON object with two books we added in out controller. Notice that we are not accessing book service directly on port 8083 but we are going through the gateway server.

6. Rating Service

Like our book service, our rating service will be a domain driven service that will handle operations related to ratings.

6.1. Setup

One more time. Navigate to start.spring.io (http://start.spring.io/). Set the artifact to 'rating-service'. Search for "*web*" and add that dependency. Search for "*config client*" and add that dependency. Search for *"eureka discovery"* and add that dependency. Generate that project.

Alternatively, add these dependencies to a project:

```
 1   <dependency>
 2       <groupId>org.springframework.cloud</groupId>
 3       <artifactId>spring-cloud-starter-config</artifactId>
 4   </dependency>
 5   <dependency>
 6       <groupId>org.springframework.cloud</groupId>
 7       <artifactId>spring-cloud-starter-eureka</artifactId>
 8   </dependency>
 9   <dependency>
10       <groupId>org.springframework.boot</groupId>
11       <artifactId>spring-boot-starter-web</artifactId>
12   </dependency>
```

For reference: you will find the bundle on *Maven Central* (config-client
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%2
0AND%20a%3A%22spring-cloud-starter-config%22), eureka-client
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%2
0AND%20a%3A%22spring-cloud-starter-eureka%22), web
(https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20
AND%20a%3A%22spring-boot-starter-web%22)).

6.2. Spring Config

Let's modify our main class:

```java
@SpringBootApplication
@EnableEurekaClient
@RestController
@RequestMapping("/ratings")
public class RatingServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(RatingServiceApplication.class, args);
    }

    private List<Rating> ratingList = Arrays.asList(
        new Rating(1L, 1L, 2),
        new Rating(2L, 1L, 3),
        new Rating(3L, 2L, 4),
        new Rating(4L, 2L, 5)
    );

    @GetMapping("")
    public List<Rating> findRatingsByBookId(@RequestParam Long bookId) {
        return bookId == null || bookId.equals(0L) ? Collections.EMPTY_LIST : ratingList.stream(
    }

    @GetMapping("/all")
    public List<Rating> findAllRatings() {
        return ratingList;
    }
}
```

We also added a REST controller and a field set by our properties file to return a value we will set during configuration.

Let's add the rating POJO:

```
1   public class Rating {
2       private Long id;
3       private Long bookId;
4       private int stars;
5
6       //standard getters and setters
7   }
```

6.3. Properties

Now we just need to add our two properties files:

bootstrap.properties in src/main/resources:

```
1   spring.cloud.config.name=rating-service
2   spring.cloud.config.discovery.service-id=config
3   spring.cloud.config.discovery.enabled=true
4
5   eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/ (http://localhost:8082/eureka/
```

rating-service.properties in our Git repository:

```
1   spring.application.name=rating-service
2   server.port=8084
3
4   eureka.client.region = default
5   eureka.client.registryFetchIntervalSeconds = 5
6   eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/ (http://localhost:8082/eureka/
```

Let's commit the changes to the repository.

6.4. Run

Once all the other applications have started we can start the rating service. The console output should look like:

```
1  DiscoveryClient_RATING-SERVICE/10.1.10.235:rating-service:8083: registering service...
2  DiscoveryClient_RATING-SERVICE/10.1.10.235:rating-service:8083 - registration status: 204
3  Tomcat started on port(s): 8084 (http)
```

Once it is up we can use our browser to access the endpoint we just created. Navigate to http://localhost:8080/rating-service/ratings/all and we get back JSON containing all our ratings. Notice that we are not accessing the rating service directly on port 8084 but we are going through the gateway server.
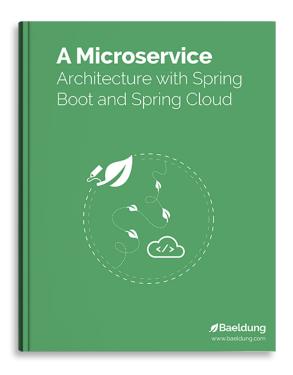
7. Conclusion

Now we are able to connect the various pieces of *Spring Cloud* into a functioning microservice application. This forms a base we can use to begin building more complex applications.

As always, you can find this source code over on Github (https://github.com/eugenp/tutorials/tree/master/spring-cloud/spring-cloud-bootstrap).

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-end)

Build your Microservice Architecture with

Spring Boot and Spring Cloud

Enter your email address

>> Download Now

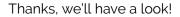▲ newest ▲ oldest ▲ most voted

Mervin Felix 🔗

Guest

Section 3.4 seems to be missing the @EnableEurekaClient in the Config Server to register itself to the Eureka Server

+ 9 **—**

🕐 2 years ago ∧

Grzegorz Piwowarek 🔗

Guest

Thanks, we'll have a look!

+ 0 **—**

🕐 2 years ago

Timothy Schimandle 🔗

Guest

Mervin,
Section 3 discusses the Eureka Server. Since the Eureka server is the application that holds all the route information it does not need to be a client as well. Each service that connects to the Eureka service

needs to be configured as a Eureka client. This is accomplished by adding the spring-cloud-starter-eureka dependency. Adding the @EnableEurekaClient annotation is optional in a boot application as adding the dependency is enough to trigger client construction on start up.

+ -6 **−**   🕐 2 years ago

Logan   🔗

Guest

hystrix.command.resource.execution.isolation.thread.timeoutInMilliseconds: 5000
YAML syntax in gateway.properties file. In github it's normal

+ 0 **−**   🕐 2 years ago ∧

Grzegorz Piwowarek   🔗

Guest

Thanks, good catch. We will update the article

+ 0 **−**   🕐 2 years ago

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)

REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)

JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)

SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)

PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)

JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)

HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)

JACKSON JSON TUTORIAL (/JACKSON)

HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)

REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)

SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)

SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE COURSES (HTTPS://COURSES.BAELDUNG.COM)

CONSULTING WORK (/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS)

ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)