Spring Cloud – Bootstrapping

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.

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.

Setup

Navigate to 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:

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.4.0.RELEASE</version>
   <relativePath/>
</parent>

<dependencies>
   <dependency>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Let’s add a dependency for the config server:

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

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

```
@SpringBootApplication
@EnableConfigServer
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:

```
server.port=8081
spring.application.name=config
```

spring.cloud.config.server.git.uri=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://'.

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.

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:

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.

Setup
Again we'll navigate to 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:

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

Spring Config
Let's add Java config to the main class:

```
@SpringBootApplication
@EnableEurekaServer
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.

Properties
Now we will add two properties files:

bootstrap.properties in src/main/resources
```
spring.cloud.config.name=discovery
spring.cloud.config.uri=http://localhost:8081
```

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

discovery.properties in our Git repository
```
spring.application.name=discovery
server.port=8082

eureka.instance.hostname=localhost

eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/
eureka.client.register-with-eureka=false
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.

Add Dependency to the Config Server
Add this dependency to the config server POM file:

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

For reference: you will find the bundle on Maven Central (eureka-client).

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

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

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

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

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

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

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.

Setup
By now we know the drill. Navigate to 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:

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

For reference: you will find the bundle on Maven Central (config-client, eureka-client, zuul).

Spring Config

Let’s add the configuration to the main class:

```
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
public class GatewayApplication {...}
```

Properties

Now we will add two properties files:

bootstrap.properties in src/main/resources

```
spring.cloud.config.name=gateway
spring.cloud.config.discovery.service-id=config
spring.cloud.config.discovery.enabled=true

eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/
```

gateway.properties in our Git repository

```
spring.application.name=gateway
server.port=8080

eureka.client.region = default
eureka.client.registryFetchIntervalSeconds = 5

zuul.routes.book-service.path=/book-service/**
zuul.routes.book-service.sensitive-headers=Set-Cookie,Authorization
hystrix.command.book-service.execution.isolation.thread.timeoutInMilliseconds=600000

zuul.routes.rating-service.path=/rating-service/**
zuul.routes.rating-service.sensitive-headers=Set-Cookie,Authorization
hystrix.command.rating-service.execution.isolation.thread.timeoutInMilliseconds=600000
```

```
zuul.routes.discovery.path=/discovery/**
zuul.routes.discovery.sensitive-headers=Set-Cookie,Authorization
zuul.routes.discovery.url=http://localhost:8082
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!

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:

```
Fetching config from server at: http://10.1.10.235:8081/
...
DiscoveryClient_GATEWAY/10.1.10.235:gateway:8080: registering service...
DiscoveryClient_GATEWAY/10.1.10.235:gateway:8080 - registration status: 204
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:

```
Fetching config from server at: 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.

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.

Setup
One more time. Navigate to 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:

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

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

For reference: you will find the bundle on Maven Central (config-client, eureka-client, web).

Spring Config
Let's modify our main class:

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

    private List<Book> bookList = Arrays.asList(
        new Book(1L, "Baeldung goes to the market", "Tim Schimandle"),
        new Book(2L, "Baeldung goes to the park", "Slavisa")
    );

    @GetMapping("")
    public List<Book> findAllBooks() {
        return bookList;
    }

    @GetMapping("/{bookId}")
    public Book findBook(@PathVariable Long bookId) {
        return bookList.stream().filter(b -> b.getId().equals(bookId)).findFirst().orElse(null);
    }
}
```

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:

```
public class Book {
    private Long id;
    private String author;
    private String title;

    // standard getters and setters
}
```

Properties

Now we just need to add our two properties files:

bootstrap.properties in src/main/resources:
```
spring.cloud.config.name=book-service
spring.cloud.config.discovery.service-id=config
spring.cloud.config.discovery.enabled=true

eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/
```

book-service.properties in our Git repository:
```
spring.application.name=book-service
server.port=8083

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

Let's commit the changes to the repository.

Run
Once all the other applications have started we can start the book service. The console output should look like:
```
DiscoveryClient_BOOK-SERVICE/10.1.10.235:book-service:8083: registering service...
DiscoveryClient_BOOK-SERVICE/10.1.10.235:book-service:8083 - registration status: 204
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.

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

Setup
One more time. Navigate to 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:
```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

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

For reference: you will find the bundle on Maven Central (config-client, eureka-client, web).

Spring Config
Let's modify our main class:

```
@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().fil-
ter(r -> r.getBookId().equals(bookId)).collect(Collectors.toList());
   }

   @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:

```
public class Rating {
   private Long id;
   private Long bookId;
   private int stars;

   //standard getters and setters
}
```

Properties
Now we just need to add our two properties files:

bootstrap.properties in src/main/resources:

```
spring.cloud.config.name=rating-service
spring.cloud.config.discovery.service-id=config
spring.cloud.config.discovery.enabled=true

eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/
```

rating-service.properties in our Git repository:

```
spring.application.name=rating-service
server.port=8084

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

Let’s commit the changes to the repository.

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

```
DiscoveryClient_RATING-SERVICE/10.1.10.235:rating-service:8083: registering service...
DiscoveryClient_RATING-SERVICE/10.1.10.235:rating-service:8083 - registration status: 204
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.