Baeldung

Microservice Architecture With Spring Boot

V 4.0.0

1. Spring Cloud – Bootstrapping

2. Exploring the New Spring Cloud Gateway

3. Spring Cloud – Securing Services

4. Spring Cloud – Tracing Services With Zipkin

5. Spring Cloud – Adding Angular 4

1. Spring Cloud – Bootstrapping

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 it's flexible and doesn't require a rebuild of the service on change
- service discovery
- hiding complexity of services deployed on different hosts

In this chapter, we'll 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.

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 can 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're going to be setting up a very simple implementation.

To learn more details and see a more complex example, take a look at our Spring Cloud Configuration chapter.

2.1. Setup

We'll start by navigating to *https://start.spring.io* and selecting Maven and Spring Boot 2.7.x.

We'll set the artifact to "config." In the dependencies section, we'll search for "config server" and add that module. Then we'll press the generate button, which allows us to download a zip file with a preconfigured project already 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>2.7.8</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>2021.0.7</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>
```

For reference, we can find the latest version on Maven Central *(spring-cloud-dependencies, test, config-server).*

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

Now 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://".

2.4. Git Repository

We'll navigate to the folder defined by *spring.cloud.config.server.git.uri* and add the folder *application-config*. Then we'll 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's working. From the command line, we'll type *mvn spring-boot:run*. This will start the server.

We 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'll 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*.

Finally, since Config Server is managing our application properties, one might wonder why even 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.

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'll solve this problem by setting up the *Eureka* discovery server. 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 the Spring Cloud Eureka chapter.

3.1. Setup

Again, we'll navigate to start.spring.io. We'll set the artifact to "discovery." We'll search for "eureka server," and add that dependency. Then we'll search for "config client," and add that dependency. Finally, we'll 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-bootstrap</artifactId>
</dependency>
```

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

For reference, we'll find the bundles on Maven Central (config-client, eureka-server).

3.2. 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 look up the configuration from the config server.

3.3. Properties

Now we'll add two properties files. First, we'll add *bootstrap.properties* into *src/main/resources:*

```
spring.cloud.config.name=discovery
spring.cloud.config.uri=http://localhost:8081
```

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

Second, we'll add *discovery.properties* to 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're telling this server that it's operating in the default zone, which matches the config client's region setting. We're also telling the server not to register with another discovery instance.

In production, we'd 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 won't be detected.

3.4. Add Dependency to the Config Server

We'll add this dependency to the config server POM file:

```
1.  <dependency>
2.      <groupId>org.springframework.cloud</groupId>
3.      <artifactId>spring-cloud-starter-netflix-eureka-client</
4.  artifactId>
5.  </dependency>
6.  <dependency>
7.      <groupId>org.springframework.cloud</groupId>
8.      <artifactId>spring-cloud-starter-bootstrap</artifactId>
9.  </dependency>
```

For reference, we can find the bundle on Maven Central (eureka-client).

We'll 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/
```

3.5. Run

Now we'll 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)
```

Let's stop and rerun the config service. If everything is correct, the 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
```

Even though 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'll 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. We need to navigate to *https://start.spring.io*. We'll then set the artifact to "gateway." Next we'll search for "gateway," "config client," and "eureka discovery client," and add those dependencies. Finally, we'll generate the 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-netflix-eureka-client</
artifactId>
</dependency>
```

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

For reference, we can find the bundle on Maven Central (*config-client, eureka-client, zuul*).

4.2. Spring Config

Let's add the configuration to the main class:

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class GatewayApplication {...}
```

4.3. Properties

Now we'll 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

spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lowerCaseServiceId=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
```

Remember to commit the changes in the repository.

4.4. Run

We'll run the config and discovery applications, and wait until the config application has registered with the discovery server. If they're already running, we don't have to restart them. Once that's complete, we'll 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's easy to make is to start the server before the config server has registered with Eureka. In that case, we'd 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 didn't have an address when the configuration request was made. We can just wait a few seconds and try again; once the config server has registered with Eureka, the problem will resolve.

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

5.1. Setup

To start, we'll navigate to *https://start.spring.io/*. We'll set the artifact to "book-service." We'll search for "web," "config client," and "eureka discovery client," and add those dependencies. Now we'll generate the project.

Alternatively, we can 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-bootstrap</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</
artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

For reference, we can find the bundle on Maven Central (*config-client, eureka-client, web*).

5.2. Spring Config

Let's modify our main class:

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class GatewayApplication {...}
```

```
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'll set during configuration.

Now let's 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*:

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

5.4. 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's up, we can use our browser to access the endpoint we just created. When we navigate to http://localhost:8080/book-service/books, we get back a JSON object with two books we added in our controller. Notice that we're not accessing the book service directly on port 8083, and instead we're going through the gateway server.

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

6.1. Setup

Let's navigate to *https://start.spring.io*. We'll set the artifact to "rating-service." We'll search for "web," "config client," and "eureka discovery client" and add those dependencies. Then we'll generate the project.

Alternatively, we can 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-bootstrap</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</
artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

For reference, we can find the bundle on Maven Central (*config-client, eureka-client, web*).

6.2. Spring Config

Let's modify our main class:

```
@SpringBootApplication
@EnableDiscoveryClient
@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().filter(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'll 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*:

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

6.4. 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's up, we can use our browser to access the endpoint we just created. Let's navigate to *http://localhost:8080/rating-service/ratings/all* and we get back JSON containing all of our ratings. Notice that we're not accessing the rating service directly on port 8084, but we're going through the gateway server.

7. Conclusion

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

As always, we can find this source code over on GitHub.

2. Exploring the New Spring Cloud Gateway

1. Overview

In this chapter, we'll explore the main features of the Spring Cloud Gateway project, a new API based on Spring 6, Spring Boot 3, and Project Reactor.

This tool provides out-of-the-box routing mechanisms often used in microservices applications as a way of hiding multiple services behind a single facade.

For an explanation of the Gateway pattern without the Spring Cloud Gateway project, check out our previous chapter.

Being focused on routing requests, the Spring Cloud Gateway forwards requests to a Gateway Handler Mapping, which determines what should be done with requests matching a specific route.

Let's start with a quick example of how the Gateway Handler resolves route configurations by using *RouteLocator*:

```
1.  @Bean
2.  public RouteLocator customRouteLocator(RouteLocatorBuilder
3.  builder) {
4.      return builder.routes()
5.        .route("r1", r -> r.host("**.baeldung.com")
6.          .and()
7.          .path("/baeldung")
8.          .uri("http://baeldung.com"))
9.        .route(r -> r.host("**.baeldung.com")
10.         .and()
11.         .path("/myOtherRouting")
12.         .filters(f -> f.prefixPath("/myPrefix"))
13.         .uri("http://othersite.com")
14.         .id("myOtherID"))
15.     .build();
16. }
```

Notice how we made use of the main building blocks of this API:

- ***Route*** — the primary API of the gateway. It's defined by a given identification (ID), a destination (URI), and a set of predicates and filters.
- ***Predicate*** — a Java 8 *Predicate* used for matching HTTP requests using headers, methods, or parameters
- ***Filter*** — a standard Spring *WebFilter*

Just like Zuul, Spring Cloud Gateway provides a means for routing requests to different services.

The routing configuration can be created by using pure Java (*RouteLocator*, as shown in the example in Section 2), or properties configuration:

```
spring:
  application:
    name: gateway-service
  cloud:
    gateway:
      routes:
      - id: baeldung
        uri: baeldung.com
      - id: myOtherRouting
        uri: localhost:9999
```

Spring Cloud Gateway matches routes using the Spring WebFlux *HandlerMapping infrastructure.*

It also includes many built-in Route Predicate Factories. All these predicates match different attributes of the HTTP request. Multiple Route Predicate Factories can be combined via the logical “and.”

Route matching can be applied both programmatically and via the configuration properties file using a different type of Route Predicate Factories.

Our chapter Spring Cloud Gateway Routing Predicate Factories explores routing factories in more detail.

5. WebFilter Factories

Route filters make the modification of the incoming HTTP request or outgoing HTTP response possible.

Spring Cloud Gateway includes many built-in WebFilter Factories, as well as the possibility to create custom filters.

Our chapter Spring Cloud Gateway WebFilter Factories explores WebFilter factories in more detail.

Spring Cloud Gateway can be easily integrated with Service Discovery and Registry libraries, such as Eureka Server and Consul:

```
1.  @Configuration
2.  @EnableDiscoveryClient
3.  public class GatewayDiscoveryConfiguration {
4.
5.      @Bean
6.      public DiscoveryClientRouteDefinitionLocator
7.        discoveryClientRouteLocator(DiscoveryClient discoveryClient)
8.  {
9.
10.         return new
11. DiscoveryClientRouteDefinitionLocator(discoveryClient);
12.     }
13. }
```

6.1. *LoadBalancerClient* Filter

The *LoadBalancerClientFilter* looks for a URI in the exchange attribute property using *ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR*.

If the URL has a *lb* scheme (e.g., *lb://baeldung-service*), it'll use the Spring Cloud *LoadBalancerClient* to resolve the name (i.e., *baeldung-service*) to an actual host and port.

The unmodified original URL is placed in the *ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR* attribute.

Spring Cloud Gateway makes use of the Actuator API, a well-known Spring Boot library that provides several out-of-the-box services for monitoring the application.

Once the Actuator API is installed and configured, the gateway monitoring features can be visualized by accessing */gateway/* endpoint.

We'll now create a simple example showcasing how to use Spring Cloud Gateway as a proxy server using the *path* predicate.

8.1. Dependencies

The Spring Cloud Gateway is currently in the milestones repository, on version 4.1.1. This is also the version we're using here.

To add the project, we'll use the ***spring-cloud-starter-gateway*** dependency, which will fetch all the required dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
    <version>4.1.1</version>
</dependency>
```

8.2. Code Implementation

And now we'll create a simple routing configuration in the application.yml file:

```
spring:
  cloud:
    gateway:
      routes:
      - id: baeldung_route
        uri: http://baeldung.com
        predicates:
        - Path=/baeldung/
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

Here's the Gateway application code:

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

After the application starts, we can access the url "*http://localhost/actuator/gateway/routes/baeldung_route*" to check all routing configurations created:

```
{
    "id":"baeldung_route",
    "predicates":[{
        "name":"Path",
        "args":{"_genkey_0":"/baeldung"}
    }],
    "filters":[],
    "uri":"http://baeldung.com",
    "order":0
}
```

We can see that the relative url "*/baeldung*" is configured as a route. So hitting the url "*http://localhost/baeldung*", we'll be redirected to "*http://baeldung.com*", as was configured in our example.

In this chapter, we explored some of the features and components that are part of Spring Cloud Gateway. This new API provides out-of-the-box tools for gateway and proxy support.

The examples presented here can be found in our GitHub repository.

3. Spring Cloud – Securing Services

In the previous chapter, Spring Cloud – Bootstrapping, we built a basic *Spring Cloud* application. This chapter shows how to secure it.

We'll use *Spring Security* to share sessions using *Spring Session* and *Redis*. This method is simple to set up and easy to extend to many business scenarios. If you're unfamiliar with *Spring* Session, check out this chapter.

Sharing sessions gives us the ability to log users in our gateway service, and propagate that authentication to any other service of our system.

If you're unfamiliar with *Redis* or *Spring Security*, it's a good idea to do a quick review of these topics at this point. While much of the chapter is copy-paste ready for an application, there's no replacement for understanding what happens under the hood.

For an introduction to *Redis*, read this chapter. For an introduction to *Spring Security*, read spring-security-login, role-and-privilege-for-spring-security-registration, and spring-security-session. To get a complete understanding of *Spring Security*, have a look at the learn-spring-security-the-master-class.

Let's start by adding the spring-boot-starter-security dependency to each module in the system:

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

Because we use *Spring* dependency management, we can omit the versions for *spring-boot-starter* dependencies.

As a second step, we'll modify the *pom.xml* of each application with spring-session, and spring-boot-starter-data-redis dependencies:

```
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Only four of our applications will tie into *Spring Session*: d**iscovery, gateway, book-service**, and **rating-service**.

Next, we'll add a session configuration class in all three services in the same directory as the main application file:

```
@EnableRedisHttpSession
public class SessionConfig
  extends AbstractHttpSessionApplicationInitializer {
}
```

Note that for the gateway service, we need to use a different annotation, namely *@EnableRedisWebSession*.

Finally, we'll add these properties to the three **.properties* files in our git repository:

```
spring.redis.host=localhost
spring.redis.port=6379
```

Now let's jump into service specific configuration.

The config service contains sensitive information often related to database connections and API keys. We can't compromise this information, so let's dive right in and secure this service.

Let's add security properties to the *application.properties* file in *src/main/resources* of the config service:

```
eureka.client.serviceUrl.defaultZone=
  http://discUser:discPassword@localhost:8082/eureka/
security.user.name=configUser
security.user.password=configPassword
security.user.role=SYSTEM
```

This will set up our service to login with discovery. In addition, we're configuring our security with the *application.properties* file.

Now let's configure our discovery service.

Our discovery service holds sensitive information about the location of all the services in the application. It also registers new instances of those services.

If malicious clients gain access, they'll learn the network location of all the services in our system, and be able to register their own malicious services into our application. It's critical that the discovery service is secured.

4.1. Security Configuration

Let's add a security filter to protect the endpoints the other services will use:

```
1.  @Configuration
2.  @EnableWebSecurity
3.  public class SecurityConfig {
4.
5.
6.     @Autowired
7.     public void configureGlobal(AuthenticationManagerBuilder auth)
8.  throws Exception {
9.           auth.inMemoryAuthentication().withUser("discUser")
10.            .password("{noop}discPassword").roles("SYSTEM");
11.    }
12.    @Bean
13.    public SecurityFilterChain securityFilterChain(HttpSecurity
14. http)  throws Exception {
15.           http.csrf(AbstractHttpConfigurer::disable)
16.                   .sessionManagement(sessionManagement ->
17.                       sessionManagement.
18. sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
19.                   .authorizeHttpRequests(authorizeRequests ->
20.                       authorizeRequests
```

```
21.  .requestMatchers(HttpMethod.GET, "/eureka/**")
22.                                .hasRole("SYSTEM")
23.                           .requestMatchers(HttpMethod.POST, "/
24.  eureka/**")
25.                                .hasRole("SYSTEM")
26.                           .requestMatchers(HttpMethod.PUT, "/
27.  eureka/**")
28.                                .hasRole("SYSTEM")
29.                           .requestMatchers(HttpMethod.DELETE, "/
30.  eureka/**")
31.                                .hasRole("SYSTEM")
31.                           .anyRequest().authenticated())
32.                   .httpBasic(Customizer.withDefaults());
33.          return http.build();
34.      }
35.  }
```

This will set up our service with a 'SYSTEM' user. This is a basic Spring Security configuration with a few twists. Let's take a look at those twists:

- ***.sessionCreationPolicy*** – tells Spring to always create a session when a user logs in on this filter
- ***.requestMatchers*** – limits what endpoints this filter applies to

The security filter we just set up configures an isolated authentication environment that pertains to the discovery service only.

4.2. Securing Eureka Dashboard

Since our discovery application has a nice UI to view currently registered services, we'll expose it using a second security filter, and tie this one into the authentication for the rest of our application. Keep in mind that having no *@Order()* tag present means that this is the last security filter to be evaluated:

```
@Configuration
public static class AdminSecurityConfig {


    public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception {
        auth.inMemoryAuthentication();
    }


    protected void configure(HttpSecurity http) throws Exception {
        http.sessionManagement(session ->
                session.
sessionCreationPolicy(SessionCreationPolicy.NEVER))
            .httpBasic(basic -> basic.disable())
            .authorizeRequests()
            .requestMatchers(HttpMethod.GET, "/").hasRole("ADMIN")
            .requestMatchers("/info", "/health").authenticated()
            .anyRequest().denyAll()
            .and().csrf(csrf -> csrf.disable());
    }
}
```

We'll add this configuration class within the *SecurityConfig* class. This will create a second security filter that will control access to our UI. This filter has a few unusual characteristics:

- ***httpBasic().disable()*** – tells spring security to disable all authentication procedures for this filter
- ***sessionCreationPolicy*** – we set this to *NEVER* to indicate that we require the user to have already authenticated before accessing resources protected by this filter

This filter will never set a user session, and relies on Redis to populate a shared security context. As such, it's dependent on another service, the gateway, to provide authentication.

4.3. Authenticating With Config Service

In the discovery project, we'll append two properties to the *bootstrap.properties* in src/main/resources:

```
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword
```

These properties will let the discovery service authenticate with the config service on startup.

Let's update our *discovery.properties* in our Git repository:

```
eureka.client.serviceUrl.defaultZone=
  http://discUser:discPassword@localhost:8082/eureka/
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

We added basic authentication credentials to our **discovery** service to allow it to communicate with the **config** service. Additionally, we configured *Eureka* to run in standalone mode by telling our service not to register with itself.

Now let's commit the file to the *git* repository; otherwise, the changes won't be detected.

Our gateway service is the only piece of our application we want to expose to the world. As such, it'll need security to ensure that only authenticated users can access sensitive information.

5.1. Security Configuration

Let's create a *SecurityConfig* class like our discovery service, and overwrite the methods with this content:

```
1.  @EnableWebFluxSecurity
2.  @Configuration
3.  public class SecurityConfig {
4.      @Bean
5.      public MapReactiveUserDetailsService userDetailsService() {
6.          UserDetails user = User.withUsername("user")
7.              .password(passwordEncoder().encode("password"))
8.              .roles("USER")
9.              .build();
10.         UserDetails adminUser = User.withUsername("admin")
11.             .password(passwordEncoder().encode("admin"))
12.             .roles("ADMIN")
13.             .build();
14.         return new MapReactiveUserDetailsService(user, adminUser);
15.     }
16.     @Bean
17.     public SecurityWebFilterChain
18. springSecurityFilterChain(ServerHttpSecurity http) {
19.         http.formLogin()
20.             .authenticationSuccessHandler(
21.                 new RedirectServerAuthenticationSuccessHandler("/
22. home/index.html"))
```

```
23.  .and().authorizeExchange()
24.              .pathMatchers("/book-service/**", "/rating-
25.  service/**", "/login*", "/")
26.              .permitAll()
27.              .pathMatchers("/eureka/**").hasRole("ADMIN")
28.              .anyExchange().authenticated().and()
29.              .logout().and().csrf().disable().
30.  httpBasic(withDefaults());
31.          return http.build();
32.      }
33.  }
```

This configuration is pretty straightforward. We declared a security filter with form login that secures a variety of endpoints.

The security on */eureka/** is to protect some static resources we'll serve from our gateway service for the *Eureka* status page. If you're building the project with the chapter, copy the *resource/static* folder from the gateway project on Github to your project.

Now we have to add @EnableRedisWebSession:

```
1.  @Configuration
2.  @EnableRedisWebSession
3.  public class SessionConfig {}
```

The Spring Cloud Gateway filter will automatically grab the request as it's redirected after login, and add the session key as a cookie in the header. This will propagate authentication to any backing service after login.

5.2. Authenticating With Config and Discovery Service

Let's add the following authentication properties to the *bootstrap.properties* file in *src/main/resources* of the gateway service:

```
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword
eureka.client.serviceUrl.defaultZone=
  http://discUser:discPassword@localhost:8082/eureka/
```

Next, let's update our gateway.properties in our Git repository:

```
management.security.sessions=always
spring.redis.host=localhost
spring.redis.port=6379
```

We added session management to always generate sessions because we only have one security filter where we can set that in the properties file. Next, we'll add our *Redis* host and server properties.

We can remove the *serviceUrl.defaultZone* property from the *gateway.properties* file in our configuration git repository. This value is duplicated in the *bootstrap file*.

Let's commit the file to the Git repository; otherwise, the changes won't be detected.

The book service server will hold sensitive information controlled by various users. This service must be secured to prevent leaks of protected information in our system.

6.1. Security Configuration

To secure our book service, we'll copy the *SecurityConfig* class from the gateway and overwrite the method with this content:

```
1.  @EnableWebSecurity
2.  @Configuration
3.  public class SecurityConfig {
4.
5.
6.      @Autowired
7.      public void registerAuthProvider(AuthenticationManagerBuilder
8.  auth) throws Exception {
9.          auth.inMemoryAuthentication();
10.     }
11.
12.
13.     @Bean
14.     public SecurityFilterChain filterChain(HttpSecurity http)
15. throws Exception {
16.         return http.authorizeHttpRequests((auth) ->
17.                 auth.requestMatchers(HttpMethod.GET, "/books")
18.             .permitAll()
19.             .requestMatchers(HttpMethod.GET, "/books/*")
20.             .permitAll()
21.             .requestMatchers(HttpMethod.POST, "/books")
22.             .hasRole("ADMIN")
23.             .requestMatchers(HttpMethod.PATCH, "/books/*")
```

```
24.  .hasRole("ADMIN")
25.                  .requestMatchers(HttpMethod.DELETE, "/books/*")
26.                  .hasRole("ADMIN"))
27.                  .csrf(csrf -> csrf.disable())
28.                  .build();
29.      }
30.  }
```

6.2. Properties

We'll add these properties to the *bootstrap.properties* file in *src/main/resources* of the book service:

```
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword
eureka.client.serviceUrl.defaultZone=
  http://discUser:discPassword@localhost:8082/eureka/
```

Then we'll add properties to our *book-service.properties* file in our git repository:

```
management.security.sessions=never
```

We can remove the *serviceUrl.defaultZone* property from the *book-service.properties* file in our configuration git repository. This value is duplicated in the *bootstrap* file.

Remember to commit these changes so the book-service will pick them up.

The rating service also needs to be secured.

7.1. Security Configuration

To secure our rating service, we'll copy the *SecurityConfig* class from the gateway and overwrite the method with this content:

```
1.  @EnableWebSecurity
2.  @Configuration
3.  public class SecurityConfig {
4.
5.
6.      @Bean
7.      public UserDetailsService users() {
8.          return new InMemoryUserDetailsManager();
9.      }
10.
11.
12.     @Bean
13.     public SecurityFilterChain filterChain(HttpSecurity
14. httpSecurity) throws Exception {
15.         return httpSecurity.authorizeHttpRequests((auth) ->
16.                     auth.requestMatchers("^/ratings\\?bookId.*$")
17.             .authenticated()
18.             .requestMatchers(HttpMethod.POST, "/ratings")
19.             .authenticated()
20.             .requestMatchers(HttpMethod.PATCH, "/ratings/*")
21.             .hasRole("ADMIN")
22.             .requestMatchers(HttpMethod.DELETE, "/ratings/*")
23.             .hasRole("ADMIN")
24.             .requestMatchers(HttpMethod.GET, "/ratings")
25.             .hasRole("ADMIN")
26.             .anyRequest()
```

```
27.  .authenticated())
28.                  .httpBasic(Customizer.withDefaults())
29.                  .csrf(csrf -> csrf.disable())
30.                  .build();
31.      }
32.  }
```

We can delete the *configureGlobal()* method from the **gateway** service.

7.2. Properties

We'll add these properties to the *bootstrap.properties* file in *src/main/resources* of the rating service:

```
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword
eureka.client.serviceUrl.defaultZone=
  http://discUser:discPassword@localhost:8082/eureka/
```

Next, we'll add properties to our *rating-service.properties* file in our git repository:

```
management.security.sessions=never
```

We can remove the *serviceUrl.defaultZone* property from the rating-service.*properties* file in our configuration git repository. This value is duplicated in the *bootstrap* file.

Remember to commit these changes so the rating service will pick them up.

Let's start *Redis* and all the services for the application: **config, discovery, gateway, book-service**, and **rating-service**. Now let's test!

First, we'll create a test class in our **gateway** project, and create a method for our test:

```
01. public class GatewayApplicationLiveTest {
02.     @Test
03.     public void testAccess() {
04.         ...
05.     }
06. }
```

Next, we'll set up our test and validate that we can access our unprotected */book-service/books* resource by adding this code snippet inside our test method:

```
TestRestTemplate testRestTemplate = new TestRestTemplate();
String testUrl = "http://localhost:8080";
```

```
01. ResponseEntity<String> response = testRestTemplate
02.   .getForEntity(testUrl + "/book-service/books", String.class);
03. Assert.assertEquals(HttpStatus.OK, response.getStatusCode());
04. Assert.assertNotNull(response.getBody());
```

We'll run this test and verify the results. If we see failures, we need to confirm that the entire application started successfully, and that configurations were loaded from our configuration git repository.

Now let's test that our users will be redirected to log in when visiting a protected resource as an unauthenticated user by appending this code to the end of the test method:

```
response = testRestTemplate
  .getForEntity(testUrl + "/home/index.html", String.class);
Assert.assertEquals(HttpStatus.FOUND, response.getStatusCode());
Assert.assertEquals("http://localhost:8080/login", response.
getHeaders()
  .get("Location").get(0))
```

We'll run the test again and confirm that it succeeds.

Next, let's actually log in, and then use our session to access the user protected result:

```
MultiValueMap<String, String> form = new LinkedMultiValueMap<>();
form.add("username", "user");
form.add("password", "password");
response = testRestTemplate
  .postForEntity(testUrl + "/login", form, String.class);
```

Now, let's extract the session from the cookie, and propagate it to the following request:

```
String sessionCookie = response.getHeaders().get("Set-Cookie")
  .get(0).split(";")[0];
HttpHeaders headers = new HttpHeaders();
headers.add("Cookie", sessionCookie);
HttpEntity<String> httpEntity = new HttpEntity<>(headers);
```

And request the protected resource:

```
response = testRestTemplate.exchange(testUrl + "/book-service/
books/1",
  HttpMethod.GET, httpEntity, String.class);
Assert.assertEquals(HttpStatus.OK, response.getStatusCode());
Assert.assertNotNull(response.getBody());
```

Let's run the test again to confirm the results.

Now let's try to access the admin section with the same session:

```
01. response = testRestTemplate.exchange(testUrl + "/rating-service/
02. ratings/all",
03.   HttpMethod.GET, httpEntity, String.class);
04. Assert.assertEquals(HttpStatus.FORBIDDEN, response.
05. getStatusCode());
```

Now we'll run the test again, and as expected, we're restricted from accessing admin areas as a plain old user.

The next test will validate that we can log in as the admin and access the admin protected resource:

```
01. form.clear();
02. form.add("username", "admin");
03. form.add("password", "admin");
04. response = testRestTemplate
05.   .postForEntity(testUrl + "/login", form, String.class);
06.
07. sessionCookie = response.getHeaders().get("Set-Cookie").get(0).
08. split(";")[0];
09. headers = new HttpHeaders();
10. headers.add("Cookie", sessionCookie);
11. httpEntity = new HttpEntity<>(headers);
12.
13. response = testRestTemplate.exchange(testUrl + "/rating-service/
14. ratings/all",
15.   HttpMethod.GET, httpEntity, String.class);
16. Assert.assertEquals(HttpStatus.OK, response.getStatusCode());
17. Assert.assertNotNull(response.getBody());
```

Our test is getting big, but we can see when we run it that by logging in as the admin, we gain access to the admin resource.

Our final test is accessing our discovery server through our gateway. To do this, we'll add this code to the end of our test:

```
01. response = testRestTemplate.exchange(testUrl + "/discovery",
02.   HttpMethod.GET, httpEntity, String.class);
03. Assert.assertEquals(HttpStatus.OK, response.getStatusCode());
```

Let's run this test one last time to confirm that everything is working. Success!

We're now able to log in on our gateway service and view content on our book, rating, and discovery services without having to log in on four separate servers!

By utilizing *Spring Session* to propagate our authentication object between servers, we're able to log in once on the gateway and use that authentication to access controllers on any number of backing services.

Security in the cloud certainly becomes more complicated. But with the help of *Spring Security* and *Spring Session*, we can easily solve this critical issue.

We now have a cloud application with security around our services. Using *Spring Cloud Gateway* and *Spring Session*, we can log users into only one service and propagate that authentication to our entire application. This means we can easily break our application into proper domains and secure each of them as we see fit.

As always you can find the source code on GitHub.

4. Spring Cloud – Tracing Services With Zipkin

In this chapter, we're going to add Zipkin to our spring cloud project. *Zipkin* is an open-source project that provides mechanisms for sending, receiving, storing, and visualizing traces. This allows us to correlate activity between servers, and get a much clearer picture of exactly what's happening in our services.

This chapter isn't an introductory chapter to distributed tracing or spring cloud. If you would like more information about distributed tracing, read our introduction to spring sleuth.

Note: **Zipkin project has deprecated the custom server. It's no longer possible to run a custom Zipkin server compatible with Spring Cloud or Spring Boot. The best approach to run a Zipkin server is inside a docker container.**

In section 2, we'll describe how to set up Zipkin for a custom server build (deprecated).

In section 3, we'll describe the preferable default server build, setup, services configuration, and run process.

Our *Zipkin* service will serve as the store for all our spans. Each span is sent to this service and collected into traces for future identification.

2.1. Setup

We'll create a new Spring Boot project and add these dependencies to *pom.xml*:

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
    <scope>runtime</scope>
</dependency>
```

For reference: we can find the latest version on *Maven Central* (zipkin-server, zipkin-autoconfigure-ui). Versions of the dependencies are inherited from spring-boot-starter-parent.

2.2. Enabling Zipkin Server

To enable the *Zipkin* server, we must add some annotations to the main application class:

```
@SpringBootApplication
@EnableZipkinServer
public class ZipkinApplication {...}
```

The new annotation, *@EnableZipkinServer*, will set up this server to listen for incoming spans and act as our UI for querying.

2.3. Configuration

First, let's create a file called *bootstrap.properties* in *src/main/resources*. Remember that this file is needed to fetch our configuration from the config server.

Let's add these properties to it:

```
spring.cloud.config.name=zipkin
spring.cloud.config.discovery.service-id=config
spring.cloud.config.discovery.enabled=true
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword


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

Now let's add a configuration file to our config repo, located at c:\Users*{username}*\ *on Windows, or /home/{username}/ on* *nix.

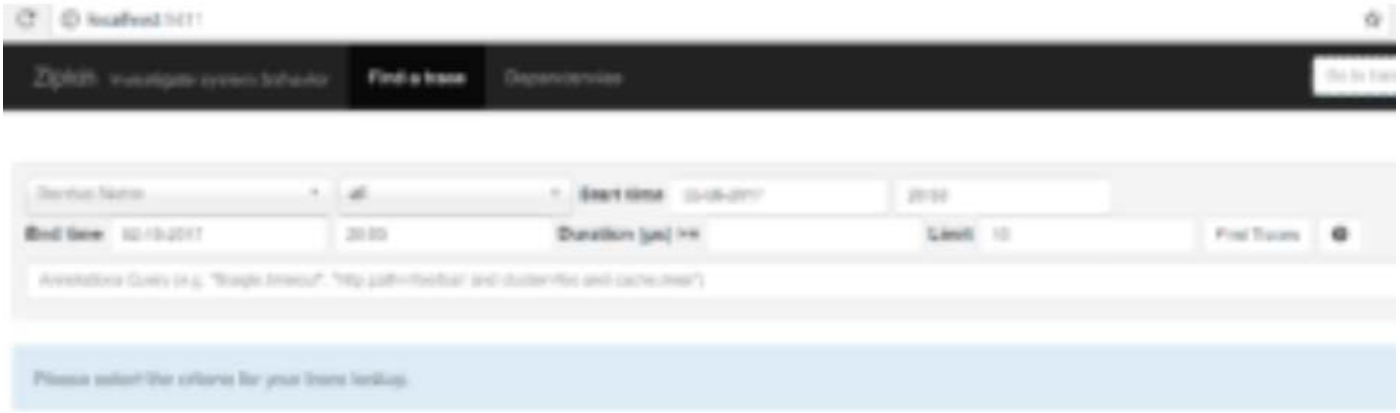
In this directory, we'll add a file named *zipkin.properties* and add these contents:

```
spring.application.name=zipkin
server.port=9411
eureka.client.region=default
eureka.client.registryFetchIntervalSeconds=5
logging.level.org.springframework.web=debug
```

Remember to commit the changes in this directory so that the config service will detect the changes and load the file.

2.4. Run

Now let's run our application, and navigate to http://localhost:9411. We should be greeted with Zipkin's homepage:



Now we're ready to add some dependencies and configurations to our services that we want to trace.

2.5. Services Configuration

The setup for the resource servers is pretty simple. In the following sections, we'll detail how to set up the book-service. After that, we can configure the same for the rating-service and gateway-service.

2.5.1. Setup

To begin sending spans to our *Zipkin* server, we'll add this dependency to our *pom.xml* file:

```
01. <dependency>
02.     <groupId>org.springframework.boot</groupId>
03.     <artifactId>spring-boot-starter-actuator</artifactId>
04. </dependency>
```

```
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
```

2.5.2. Configuration

Now let's add some configuration to our *book-service.properties* file in the config repository:

```
management.tracing.sampling.probability=1
```

Zipkin works by sampling actions on a server. By setting the *management.tracing.sampling.probability* to 1, we're setting the sampling rate to 100%.

As already stated, this is the suggested way to run Zipkin.

3.1. Zipkin Module Setup

In our IDE project, we'll create a "Zipkin" folder and add a docker-compose.yml file with the contents:

```
version: "3.9"
services:
zipkin:
image: openzipkin/zipkin
ports:
- 9411:9411
```

Alternatively, check the official documentation here.

3.2. Service Configuration

Service configuration is the same as in section 2.5 above.

3.3. Run

First, we'll start the Redis server, config, discovery, **gateway, book, rating**, and **Zipkin** docker image with "*docker compose up -d*". Then we'll navigate to *http://localhost:8080/book-service/books* address. After that, we'll open a new tab and navigate to *http://localhost:9411*. On this page, we'll select *book-service*, and press the 'Find Traces' button. We should see a trace appear in the search results. Finally, we can click that trace of opening it:

On the trace page, we can see the request broken down by service. The first two spans are created by the gateway, and the last is created by the book-service. This shows us how much time the request spent processing on the book-service, 18.379 ms, and on the gateway, 87.961 ms.

4. Conclusion

We've seen how easy it is to integrate *Zipkin* into our cloud application.

This gives us some much-needed insight into how communication travels through our application. As our application grows in complexity, *Zipkin* can provide us with much-needed information on where requests are spending their time. This can help us determine where things are slowing down, and indicate what areas of our application need improvement.

As always you can find the source code over on Github.

5. Spring Cloud – Adding Angular 4

In our last Spring Cloud chapter, we added Zipkin support into our application. In this chapter, we're going to be adding a front-end application to our stack.

Up until now, we've been working entirely on the back end to build our cloud application. But what good is a web app if there's no UI? In this chapter, we'll solve that issue by integrating a single page application into our project.

We'll be writing this app using Angular and Bootstrap. The style of Angular code feels a lot like coding a Spring app, which is a natural crossover for a Spring developer. While the front end code will be using Angular, the content of this chapter can be easily extended to any front end framework with minimal effort.

In this chapter, we'll be building an Angular app, and connecting it to our cloud services. We'll demonstrate how to integrate login between a SPA and Spring Security. We'll also show how to access our application's data using Angular's support for HTTP communication.

With the front end in place, we're going to switch to form based login, and secure parts of the UI to privileged users. This requires making changes to our gateway security configuration.

2.1. Update HttpSecurity

First, let's update the configure (HttpSecurity http) method in our gateway SecurityConfig.java class:

```
01. @Bean
02. public SecurityWebFilterChain filterChain(ServerHttpSecurity
03. http) {
04.     http.formLogin()
05.         .authenticationSuccessHandler(
06.             new RedirectServerAuthenticationSuccessHandler("/
07. home/browser/index.html"))
08.         .and().authorizeExchange()
09.         .pathMatchers("/book-service/**", "/rating-service/**",
10. "/login*", "/").permitAll()
11.         .pathMatchers("/eureka/**").hasRole("ADMIN")
12.         .anyExchange().authenticated()
13.         .and().logout().and().csrf().disable()
14.         .httpBasic(withDefaults());
15.     return http.build();
16. }
```

We added a default success URL to point to /home/browser/index.html, as this will be where our Angular app lives. Next, we'll configure the ant matchers to allow any request through the gateway, except for the Eureka resources. This will delegate all security checks to back-end services.

Then we removed the logout success URL, as the default redirect back to the login page will work fine.

2.2. Add a Principal Endpoint

Now let's add an endpoint to return the authenticated user. This will be used in our Angular app to log in and identify the roles our user has.

This will help us control what actions they can do on our site.

In the gateway project, we'll add an AuthenticationController class:

```
01. @RestController
02. public class AuthenticationController {
03.
04.     @GetMapping("/me")
05.     public Principal getMyUser(Principal principal) {
06.         return principal;
07.     }
08. }
```

The controller returns the currently logged in user object to the caller. This gives us all the information we need to control our Angular app.

2.3. Add a Landing Page

Let's add a very simple landing page so that users see something when they go to the root of our application.

In src/main/resources/static, let's add an index.html file with a link to the login page:

```
<!DOCTYPE html>
<html lang=”en”>
<head>
    <meta charset=”UTF-8”>
    <title>Book Rater Landing</title>
</head>
<body>
    <h1>Book Rater</h1>
    <p>So many great things about the books</p>
    <a href=”/login”>Login</a>
</body>
</html>
```

Before starting a new Angular project, let's make sure to install the latest versions of Node.js and npm.

3.1. Install the Angular CLI

To begin, we'll need to use npm to download and install the Angular command line interface. Let's open a terminal and run:

```
npm install -g @angular/cli
```

This will download and install the CLI globally.

3.2. Install a New Project

While still in the terminal, let's navigate to the gateway project and go into the gateway/src/main folder. We'll create a directory called "angular" and navigate to it.

From here, we'll run:

```
ng new ui
```

Be patient; the CLI's setting up a brand new project and downloading all the JavaScript dependencies with npm. It's not uncommon for this process to take many minutes.

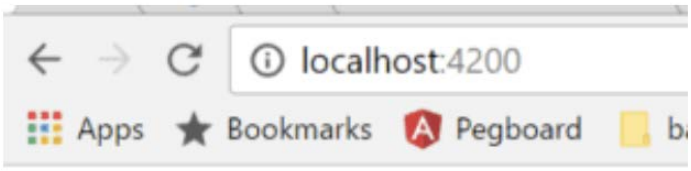
The ng command is the shortcut for the Angular CLI, the new parameter instructs that CLI to create a new project, and the ui command gives our project a name.

3.3. Run the Project

Once the new command is complete, we'll navigate to the ui folder that was created and run:

```
ng serve
```

Once the project builds, we'll navigate to http://localhost:4200. We should see this in the browser:



app works!

Congratulations! We just built an Angular app!

3.4. Install Bootstrap

Now let's use npm to install bootstrap. From the ui directory, we'll run this command:

```
npm install bootstrap@5.3.3 --save
```

This will download bootstrap into the node_modules folder.

In the ui directory, we'll open the angular.json file. This is the file that configures some properties about our project. Let's find the projects > styles property, and add a file location of our Bootstrap CSS class:

```
"styles": [
    "styles.css",
    "../node_modules/bootstrap/dist/css/bootstrap.min.css"
],
```

This will instruct Angular to include Bootstrap in the compiled CSS file that's built with the project.

3.5. Set the Build Output Directory

Next, we need to tell Angular where to put the build files so that our spring boot app can serve them. Spring Boot can serve files from two locations in the resources folder:

src/main/resources/static
src/main/resource/public

Since we're already using the static folder to serve some resources for Eureka, and Angular deletes this folder each time a build is run, let's build our Angular app into the public folder.

Let's open the angular.json file again, and find the options > outputPath property. We'll update that string:

```
"outputPath": "../../resources/static/home",
```

If the Angular project is located in src/main/angular/ui, then it will build to the src/main/resources/public folder. If the app is in another folder, this string will need to be modified to set the location correctly.

3.6. Automate the Build With Maven

Lastly, we'll set up an automated build to run when we compile our code. This ant task will run the Angular CLI build task whenever "mvn compile" is run. We'll add this step to the gateway's POM.xml to ensure that each time we compile, we get the latest ui changes:

```
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <executions>
        <execution>
            <phase>generate-resources</phase>
            <configuration>
                <tasks>
                    <exec executable="cmd" osfamily="windows"
                      dir="${project.basedir}/src/main/angular/
ui">
                        <arg value="/c"/>
                        <arg value="ng"/>
                        <arg value="build"/>
                    </exec>
                    <exec executable="/bin/sh" osfamily="mac"
                      dir="${project.basedir}/src/main/angular/
ui">
                        <arg value="-c"/>
                        <arg value="ng build"/>
                    </exec>
                </tasks>
            </configuration>
            <goals>
                <goal>run</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

We should note that this setup does require that the Angular CLI be available on the classpath. Pushing this script to an environment that doesn't have that dependency will result in build failures.

Now let's start building our Angular application!

In this section of the chapter, we'll build an authentication mechanism for our page. We'll use basic authentication, and follow a simple flow to make it work.

Users have a login form where they can enter their username and password.
Next, we'll use their credentials to create a base64 authentication token, and request the "/me" endpoint. The endpoint returns a Principal object containing the roles of this user.

Finally, we'll store the credentials and the principal on the client to use in subsequent requests.

4.1. Template

In the gateway project, let's navigate to src/main/angular/ui/src/app and open the app.component.html file. This is the first template that Angular loads, and will be where our users will land after logging in.

Here, we're going to add some code to display a navigation bar with a login form:

```
<nav class="navbar navbar-toggleable-md navbar-inverse fixed-top
bg-inverse">
    <button class="navbar-toggler navbar-toggler-right"
type="button"
      data-toggle="collapse" data-target="#navbarCollapse"
      aria-controls="navbarCollapse" aria-expanded="false"
      aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
    </button>
    <a class="navbar-brand" href="#">Book Rater
        <span *ngIf="principal.isAdmin()">Admin</span></a>
```

```
12. <div class="collapse navbar-collapse" id="navbarCollapse">
13.     <ul class="navbar-nav mr-auto">
14.     </ul>
15.     <button *ngIf="principal.authenticated" type="button"
16.       class="btn btn-link" (click)="onLogout()">Logout</button>
17.     </div>
18. </nav>
19.
20. <div class="jumbotron">
21.     <div class="container">
22.         <h1>Book Rater App</h1>
23.         <p *ngIf="!principal.authenticated" class="lead">
24.         Anyone can view the books.
25.         </p>
26.         <p *ngIf="principal.authenticated && !principal.
27. isAdmin()" class="lead">
28.         Users can view and create ratings</p>
29.         <p *ngIf="principal.isAdmin()"  class="lead">Admins can
30. do anything!</p>
31.     </div>
32. </div>
```

This code sets up a navigation bar with Bootstrap classes. Embedded in the bar is an inline login form. Angular uses this markup to interact with JavaScript dynamically to render various parts of the page and control things like form submission.

Statements like (ngSubmit)="onLogin(f)" simply indicate that when the form is submitted, it needs to call the method "onLogin(f)" and pass the form to that function. Within the jumbotron div, we have paragraph tags that will display dynamically depending on the state of our principal object.

Next, let's code up the Typescript file that will support this template.

4.2. Typescript

From the same directory, let's open the app.component.ts file. In this file we'll add all the typescript properties and methods required to make our template function:

```
01. import {Component} from '@angular/core';
02. import {RouterOutlet} from '@angular/router';
03. import {Principal} from "./principal";
04. import {HttpClientModule, HttpResponse} from "@angular/common/
05. http";
06. import {Book} from "./book";
07. import {HttpService} from "./http.service";
08. import {CommonModule} from '@angular/common';
09.
10. @Component({
11.   selector: 'app-root',
12.   standalone: true,
13.   imports: [RouterOutlet, CommonModule, HttpClientModule],
14.   providers: [HttpService],
15.   templateUrl: './app.component.html',
16.   styleUrl: './app.component.css'
17. })
18. export class AppComponent {
19.   selectedBook: Book = null;
20.   principal: Principal = new Principal(false, []);
21.   loginFailed: boolean = false;
22.
23.   constructor(private httpService: HttpService){}
24.
25.   ngOnInit(): void {
26.     this.httpService.me()
27.       .subscribe((response) => {
28.         let principalJson = response.json();
29.         this.principal = new Principal(principalJson.
30. authenticated, principalJson.authorities);
```

```
31.         }, (error) => {
32.           console.log(error);
33.         });
34.     }
35.
36.
37.     onLogout() {
38.       this.httpService.logout()
39.         .subscribe((response) => {
40.           if (response.status === 200) {
41.             this.loginFailed = false;
42.             this.principal = new Principal(false, []);
43.             window.location.replace(response.url);
44.           }
45.         }, (error) => {
46.           console.log(error);
47.         });
48.     }
49. }
```

This class hooks into the Angular life cycle method, ngOnInit(). In this method, we'll call the /me endpoint to get the user's current role and state. This determines what the user sees on the main page. This method will be fired whenever this component is created, which is a great time to be checking the user's properties for permissions in our app.

We also have an onLogout() method that logs our user out, and restores the state of this page to its original settings.

There's some magic going on here with the httpService property that's declared in the constructor. Angular is injecting this property into our class at runtime. Angular manages singleton instances of service classes, and injects them using constructor injection, just like Spring.

Next, we need to define the HttpService class.

4.3. HttpService

In the same directory, let's create a file named "http.service.ts". In this file, we'll add this code to support the login and logout methods:

```
01. import {Injectable} from "@angular/core";
02. import {Observable} from "rxjs";
03. import {HttpClientModule, HttpResponse, HttpClient,
04. HttpHeaders, HttpRequest, HttpContext} from "@angular/common/
05. http";
06.
07. @Injectable({providedIn: 'root'})
08. export class HttpService {
09.
10.   constructor(private http: HttpClient) { }
11.
12.   me(): Observable {
13.     return this.http.get("/me", {'headers': this.
14. makeOptions()})
15.   }
16.
17.   logout(): Observable {
18.     return this.http.post("/logout", '', {'headers': this.
19. makeOptions()})
20.   }
21.
22.   private makeOptions(): HttpHeaders {
23.     return new HttpHeaders({'Content-Type': 'application/
24. json'});
25.   }
26. }
```

In this class, we're injecting another dependency using Angular's DI construct. This time it's the HttpClient class. This class handles all HTTP communication and is provided to us by the framework.

These methods each perform an HTTP request using angular's HTTP library. Each request also specifies a content type in the headers.

Now we need to do one more thing to get the HttpService registered in the dependency injection system. We'll open the app.component.ts file and find the providers property. Then we'll add the HttpService to that array. The result should look like this:

```
providers: [HttpService],
```

4.4. Add Principal

Next, let's add our Principal DTO object in our Typescript code. In the same directory, we'll add a file called "principal.ts" and add this code:

```
01. export class Principal {
02.     public authenticated: boolean;
03.     public authorities: Authority[] = [];
04.     public credentials: any;
05.
06.     constructor(authenticated: boolean, authorities: any[],
07. credentials: any) {
08.         this.authenticated = authenticated;
09.         authorities.map(
10.           auth => this.authorities.push(new Authority(auth.
11. authority)))
12.         this.credentials = credentials;
13.   }
```

```
14.     isAdmin() {
15.         return this.authorities.some(
16.           (auth: Authority) => auth.authority.indexOf('ADMIN')
17. > -1)
18.     }
19. }
20.
21.
22. export class Authority {
23.     public authority: String;
24.
25.
26.     constructor(authority: String) {
27.         this.authority = authority;
28.     }
29. }
```

We added the Principal class and an Authority class. These are two DTO classes, much like POJOs in a Spring app. Because of that, we don't need to register these classes with the DI system in angular.

Next, let's configure a redirect rule to redirect unknown requests to the root of our application.

4.5. 404 Handling

Now let's navigate back into the Java code for the gateway service. Where the GatewayApplication class resides, we'll add a new class called ErrorPageConfig:

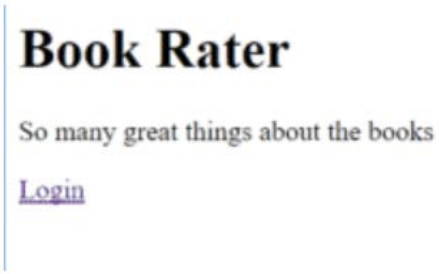
```
01. @Component
02. public class ErrorPageConfig implements ErrorPageRegistrar {
03.
04.     @Override
05.     public void registerErrorPages(ErrorPageRegistry registry)
06. {
07.         registry.addErrorPages(new ErrorPage(HttpStatus.NOT_
08. FOUND,
09.           "/home/index.html"));
10.     }
11.
12.
13. }
```

This class will identify any 404 response, and redirect the user to "/home/index.html". In a single page app, this is how we handle all traffic not going to a dedicated resource, since the client should be handling all navigable routes.

Now we're ready to fire this app up and see what we built.

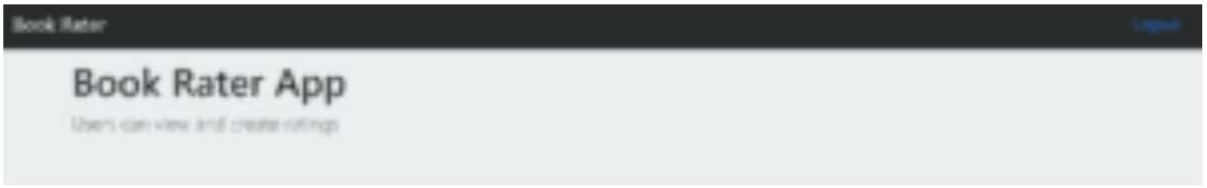
4.6. Build and View

Let's run "mvn compile" from the gateway folder. This will compile our java source, and build the Angular app to the public folder. Next, let's start the other cloud applications: config, discovery, and zipkin. Then we'll run the gateway project. When the service starts, we'll navigate to http://localhost:8080 to see our app. We should see something like this:

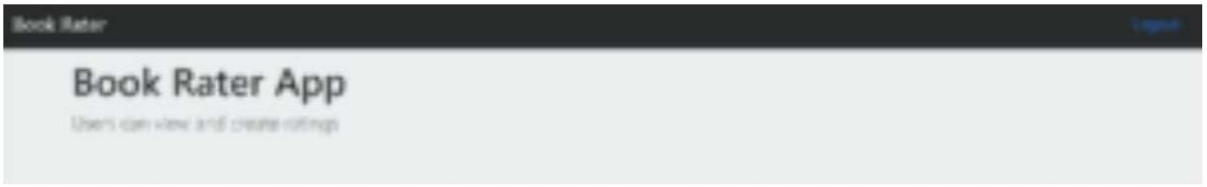


Next, let's follow the link to the login page:

We'll log in using the user/password credentials. Click "Login," and we should be redirected to /home/index.html, where our single page app loads:



It looks like our jumbotron is indicating we're logged in as a user. Now we'll log out by clicking the link in the upper right corner, and log in using the admin/admin credentials this time:



Looks good! Now we're logged in as an admin.

In this chapter, we demonstrated how easy it is to integrate a single page app into our cloud system. We took a modern framework, and integrated a working security configuration into our application.

Using these examples, try to write some code to make a call to the book-service or rating-service. Since we now have examples of making HTTP calls and wiring data to the templates, this should be relatively easy.

If you would like to see how the rest of the site is built, as always, you can find the source code over on Github.