A Microservice
Architecture with Spring Boot and Spring Cloud

Table of Contents

In this guide, we'll focus on building out a simple, but fully working base for a microservice architecture with Spring Boot and Spring Cloud.

If you want to dig further into Spring Cloud, definitely go over our Spring Cloud Articles.

The system we're going to start building here is a simple library application, primarily focused on books and reviews. Let's start with the two REST APIs:

- Book API
- Rating API

First, let's take a look at our resources *Book*:

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

And of course the Rating resource, backing the second API:

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

Now, we'll bootstrap the two simple APIs – */books and /ratings.*

First, let's explore our Books API:

```java
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private BookService bookService;

    @GetMapping
    public List<Book> findAllBooks() {
        return bookService.findAllBooks();
    }

    @GetMapping("/{bookId}")
    public Book findBook(@PathVariable Long bookId) {
        return bookService.findBookById(bookId);
    }

    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookService.createBook(book);
    }

    @DeleteMapping("/{bookId}")
    public void deleteBook(@PathVariable Long bookId) {
        bookService.deleteBook(bookId);
    }
    @PutMapping("/{bookId}")
    public Book updateBook(@RequestBody Book book, @PathVariable Long bookId) {
        return bookService.updateBook(book, bookId);
    }
    @PatchMapping("/{bookId}")
    public Book updateBook(
        @RequestBody Map<String, String> updates,
        @PathVariable Long bookId) {
         return bookService.updateBook(updates, bookId);
    }
}
```

And similarly the *RatingController:*

```java
@RestController
@RequestMapping("/ratings")
public class RatingController {

    @Autowired
    private RatingService ratingService;

    @GetMapping
    public List<Rating> findRatingsByBookId(
        @RequestParam(required = false, defaultValue = "0") Long bookId) {
            if (bookId.equals(0L)) {
                return ratingService.findAllRatings();
            }
            return ratingService.findRatingsByBookId(bookId);
    }

    @PostMapping
    public Rating createRating(@RequestBody Rating rating) {
            return ratingService.createRating(rating);
    }

    @DeleteMapping("/{ratingId}")
    public void deleteRating(@PathVariable Long ratingId) {
            ratingService.deleteRating(ratingId);
    }

    @PutMapping("/{ratingId}")
    public Rating updateRating(@RequestBody Rating rating, @PathVariable Long ratingId) {
            return ratingService.updateRating(rating, ratingId);
    }
    @PatchMapping("/{ratingId}")
    public Rating updateRating(
        @RequestBody Map<String, String> updates,
        @PathVariable Long ratingId) {
            return ratingService.updateRating(updates, ratingId);
    }
}
```

Notice that we're not focusing on persistence here – the main focus is the API each application is exposing.

Each API has its own, separate Boot application and is deployment entirely independently of anything else.

When deployed locally, the APIs will be available at:

```
1   http://localhost:8080/book-service/books
2   http://localhost:8080/rating-service/ratings
```

The next step here is to secure the two APIs. Although we may choose to upgrade to an OAuth2 + JWT implementation later on, a good place to start is only using form login. We will see this in section 5.3 when we see the security configuration of the gateway.

First, our Book application security configuration:

```
1   @EnableWebSecurity
2   @Configuration
3   public class SecurityConfig extends WebSecurityConfigurerAdapter {
4
5       @Autowired
6       public void configureGlobal1(AuthenticationManagerBuilder auth)
7         throws Exception {
8           auth.inMemoryAuthentication();
9       }
10
11      @Override
12      protected void configure(HttpSecurity http) throws Exception {
13          http.httpBasic()
14              .disable()
15          .authorizeRequests()
16              .antMatchers(HttpMethod.GET, "/books").permitAll()
17              .antMatchers(HttpMethod.GET, "/books/*").permitAll()
18              .antMatchers(HttpMethod.POST, "/books").hasRole("ADMIN")
19              .antMatchers(HttpMethod.PATCH, "/books/*").hasRole("ADMIN")
20              .antMatchers(HttpMethod.DELETE, "/books/*").hasRole("ADMIN")
21              .anyRequest().authenticated()
22              .and()
23          .csrf()
24              .disable();
25      }
26  }
```

And then the *Rating* configuration:

```java
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal1(AuthenticationManagerBuilder auth)
      throws Exception {
        auth.inMemoryAuthentication();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic()
            .disable()
          .authorizeRequests()
            .regexMatchers("^/ratings\\?bookId.*$").authenticated()
            .antMatchers(HttpMethod.POST,"/ratings").authenticated()
            .antMatchers(HttpMethod.PATCH,"/ratings/*").hasRole("ADMIN")
            .antMatchers(HttpMethod.DELETE,"/ratings/*").hasRole("ADMIN")
            .antMatchers(HttpMethod.GET,"/ratings").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
          .csrf()
            .disable();
    }
}
```

Because the APIs are simple, we can use global matchers right in the security config. However, as they become more and more complex, we should look towards migrating these to a method level annotation implementation.

Right now **the security semantics are very simple:**

- Anyone can read Resources
- Only admins can modify resources

Now, with our two APIs running independently, it's time to look at the using Spring Cloud and bootstrap some very useful components in our microservice topology:

1. **Configuration Server** – provides, manages and centralize the configuration to externalize the configuration of our different modules

2. **Discovery Server** – enable applications to find each other efficiently and with flexibility

3. **Gateway Server** – acts as a reverse proxy and hide complexity of our system by providing all our APIs on one port

4. **Two REST APIs:** Books API and Ratings API

We're going to use Spring Initializr to bootstrap these 3 new applications quickly.

First, we will setup the configuration server, we will need Cloud Config, Eureka and Security:

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

Next, we need to use *@EnableConfigServer* to make our configuration server discoverable via Eureka client – as follows:

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

And here is *application.properties:*

```
1    server.port=8081
2    spring.application.name=config
3    spring.cloud.config.server.git.uri=file:///${user.home}/application-config
4    eureka.client.region=default
5    eureka.client.registryFetchIntervalSeconds=5
6    eureka.client.serviceUrl.defaultZone=
7      http://discUser:discPassword@localhost:8082/eureka/
8    security.user.name=configUser
9    security.user.password=configPassword
10   security.user.role=SYSTEM
```

Next, we need to create a **local Git repository** *application-config* in our HOME directory to hold our configuration files:

```
1   cd ~
2   mkdir application-config
3   cd application-config
4   git init
```

Note: We are using local Git repository for testing purpose.

For the discovery server, we need Eureka, Cloud Config Client and Security:

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

We will configure our discovery server by firstly adding *@EnableEurekaServer* annotation:

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

And make sure we also need to secure our discovery server endpoints:

```
@Configuration
@EnableWebSecurity
@Order(1)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
     public void configureGlobal(AuthenticationManagerBuilder auth) {
         auth.inMemoryAuthentication()
             .withUser("discUser")
             .password("discPassword")
             .roles("SYSTEM");
    }

    @Override
    protected void configure(HttpSecurity http) {
        http
        .sessionManagement()
          .sessionCreationPolicy(SessionCreationPolicy.ALWAYS).and()
        .requestMatchers().antMatchers("/eureka/**").and()
        .authorizeRequests()
          .antMatchers("/eureka/**").hasRole("SYSTEM")
          .anyRequest().denyAll().and()
        .httpBasic().and()
        .csrf().disable();
    }
}
```

and also secure the Eureka dashboard:

```
1   @Configuration
2   public static class AdminSecurityConfig extends WebSecurityConfigurerAdapter {
3       @Override
4       protected void configure(HttpSecurity http) {
5           http
6           .sessionManagement()
7               .sessionCreationPolicy(SessionCreationPolicy.NEVER).and()
8           .httpBasic().disable()
9           .authorizeRequests()
10              .antMatchers(HttpMethod.GET, "/").hasRole("ADMIN")
11              .antMatchers("/info", "/health").authenticated()
12              .anyRequest().denyAll().and()
13          .csrf().disable();
14      }
15  }
```

Now, we will add *bootstrap.properties* in our discovery server **resources folder** as follows:

```
1   spring.cloud.config.name=discovery
2   spring.cloud.config.uri=http://localhost:8081
3   spring.cloud.config.username=configUser
4   spring.cloud.config.password=configPassword
```

Finally, we will add *discovery.properties* in our **application-config Git repository:**

```
1   spring.application.name=discovery
2   server.port=8082
3   eureka.instance.hostname=localhost
4   eureka.client.serviceUrl.defaultZone=
5     http://discUser:discPassword@localhost:8082/eureka/
6   eureka.client.register-with-eureka=false
7   eureka.client.fetch-registry=false
8   spring.redis.host=localhost
9   spring.redis.port=6379
```

Note that:

- We are using *@Order(1)* as we have to security configurations for discovery server, one for the endpoints and the other for the dashboard

- *spring.cloud.config.name* should be the same as discovery server properties file in configuration repository

- We have to provide *spring.cloud.config.uri* in the discovery server bootstrap properties to be able to obtain full configuration from configuration server

To setup the gateway server we need Cloud Config Client, Eureka Client, Zuul and Security:

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

Next, we need to need to configure our gateway server as follows:

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

And add a simple security configuration:

```java
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user").password("password").roles("USER")
                .and()
                .withUser("admin").password("admin").roles("ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        .authorizeRequests()
            .antMatchers("/book-service/books").permitAll()
            .antMatchers("/eureka/**").hasRole("ADMIN")
            .anyRequest().authenticated().and()
        .formLogin().and()
        .logout().permitAll().and()
        .csrf().disable();
    }
}
```

We also need to add *bootstrap.properties* in our gateway server **resources folder** as follows:

```properties
spring.cloud.config.name=gateway
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/
```

Finally, we will add *gateway.properties* in our **application-config** Git repository:

```
1   spring.application.name=gateway
2   server.port=8080
3   eureka.client.region = default
4   eureka.client.registryFetchIntervalSeconds = 5
5   management.security.sessions=always
6
7   zuul.routes.book-service.path=/book-service/**
8   zuul.routes.book-service.sensitive-headers=Set-Cookie,Authorization
9   hystrix.command.book-service.execution.isolation.thread
10    .timeoutInMilliseconds=600000
11  zuul.routes.rating-service.path=/rating-service/**
12  zuul.routes.rating-service.sensitive-headers=Set-Cookie,Authorization
13  hystrix.command.rating-service.execution.isolation.thread
14    .timeoutInMilliseconds=600000
15  zuul.routes.discovery.path=/discovery/**
16  zuul.routes.discovery.sensitive-headers=Set-Cookie,Authorization
17  zuul.routes.discovery.url=http://localhost:8082
18  hystrix.command.discovery.execution.isolation.thread
19    .timeoutInMilliseconds=600000
20
21  spring.redis.host=localhost
22  spring.redis.port=6379
```

Note: We are using *zuul.routes.book-service.path* to route any request that comes in on */book-service/*** to our Book Service application, same applies for our Rating Service

We will need the same setup for both APIs: Config Client, Eureka, JPA, Web and Security:

```
1   <dependency>
2       <groupId>org.springframework.cloud</groupId>
3       <artifactId>spring-cloud-starter-config</artifactId>
4   </dependency>
5   <dependency>
6       <groupId>org.springframework.cloud</groupId>
7       <artifactId>spring-cloud-starter-eureka</artifactId>
8   </dependency>
9   <dependency>
10      <groupId>org.springframework.boot</groupId>
11      <artifactId>spring-boot-starter-data-jpa</artifactId>
12  </dependency>
13  <dependency>
14      <groupId>org.springframework.boot</groupId>
15      <artifactId>spring-boot-starter-web</artifactId>
16  </dependency>
17  <dependency>
18      <groupId>org.springframework.boot</groupId>
19      <artifactId>spring-boot-starter-security</artifactId>
20  </dependency>
```

We will also use *@EnableEurekaClient* for both APIs- as follows:

```
1   @SpringBootApplication
2   @EnableEurekaClient
3   public class ServiceApplication {...}
```

Here is our first resource service "Book Service" properties configuration *book-service.properties* which will be at **application-config** repository:

```
1   spring.application.name=book-service
2   server.port=8083
3   eureka.client.region=default
4   eureka.client.registryFetchIntervalSeconds=5
5   management.security.sessions=never
```

and here is the Book Service *bootstrap.properties* which will be in our book service **resources folder**:

```
1   spring.cloud.config.name=book-service
2   spring.cloud.config.discovery.service-id=config
3   spring.cloud.config.discovery.enabled=true
4   spring.cloud.config.username=configUser
5   spring.cloud.config.password=configPassword
6   eureka.client.serviceUrl.defaultZone=
7     http://discUser:discPassword@localhost:8082/eureka/
```

Similarly here is our second service "Rating Service" *rating-service.properties*:

```
1   spring.application.name=rating-service
2   server.port=8084
3   eureka.client.region=default
4   eureka.client.registryFetchIntervalSeconds=5
5   management.security.sessions=never
```

and *bootstrap.properties*:

```
1   spring.cloud.config.name=rating-service
2   spring.cloud.config.discovery.service-id=config
3   spring.cloud.config.discovery.enabled=true
4   spring.cloud.config.username=configUser
5   spring.cloud.config.password=configPassword
6   eureka.client.serviceUrl.defaultZone=
7     http://discUser:discPassword@localhost:8082/eureka/
```

We will share sessions between different services in our system using Spring Session. Sharing sessions enable logging users in our gateway service and propagate that authentication to the other services

First, we need to add the following dependencies to discovery server, gateway server, book service and rating service:

```
1  <dependency>
2      <groupId>org.springframework.session</groupId>
3      <artifactId>spring-session</artifactId>
4  </dependency>
5  <dependency>
6      <groupId>org.springframework.boot</groupId>
7      <artifactId>spring-boot-starter-data-redis</artifactId>
8  </dependency>
```

We need to add session configuration to our discovery server and REST APIs:

```
1  @EnableRedisHttpSession
2  public class SessionConfig
3      extends AbstractHttpSessionApplicationInitializer {
4  }
```

For our gateway server, it will be slightly different:

```
1  @Configuration
2  @EnableRedisHttpSession(redisFlushMode = RedisFlushMode.IMMEDIATE)
3  public class SessionConfig extends AbstractHttpSessionApplicationInitializer {
4  }
```

We also add a simple filter to our gateway server to forward the session so that authentication will propagate to another service after login:

```java
@Component
public class SessionSavingZuulPreFilter
  extends ZuulFilter {

    @Autowired
    private SessionRepository repository;

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext context = RequestContext.getCurrentContext();
        HttpSession httpSession = context.getRequest().getSession();
        Session session = repository.getSession(httpSession.getId());

        context.addZuulRequestHeader(
          "Cookie", "SESSION=" + httpSession.getId());
        return null;
    }

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }
}
```

Finally, we will test our REST API

First, a simple setup:

```
1  private final String ROOT_URI = "http://localhost:8080";
2  private FormAuthConfig formConfig
3    = new FormAuthConfig("/login", "username", "password");
4
5  @Before
6  public void setup() {
7      RestAssured.config = config().redirect(
8          RedirectConfig.redirectConfig().followRedirects(false));
9  }
```

Next, let's get all books:

```
1  @Test
2  public void whenGetAllBooks_thenSuccess() {
3      Response response = RestAssured.get(ROOT_URI + "/book-service/books");
4
5      Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
6      Assert.assertNotNull(response.getBody());
7  }
```

Then, try to access protected resource without login:

```
1  @Test
2  public void whenAccessProtectedResourceWithoutLogin_thenRedirectToLogin() {
3      Response response = RestAssured.get(ROOT_URI + "/book-service/books/1");
4
5      Assert.assertEquals(HttpStatus.FOUND.value(), response.getStatusCode());
6      Assert.assertEquals("http://localhost:8080/login",
7        response.getHeader("Location"));
8  }
```

Then, login and create new Book:

```
1   @Test
2   public void whenAddNewBook_thenSuccess() {
3       Book book = new Book("Baeldung", "How to spring cloud");
4       Response bookResponse = RestAssured.given().auth()
5          .form("admin", "admin", formConfig).and()
6          .contentType(ContentType.JSON)
7          .body(book)
8          .post(ROOT_URI + "/book-service/books");
9       Book result = bookResponse.as(Book.class);
10
11      Assert.assertEquals(HttpStatus.OK.value(), bookResponse.getStatusCode());
12      Assert.assertEquals(book.getAuthor(), result.getAuthor());
13      Assert.assertEquals(book.getTitle(), result.getTitle());
14  }
```

and then access the protected resource after login:

```
1   @Test
2   public void whenAccessProtectedResourceAfterLogin_thenSuccess() {
3       Response response = RestAssured.given().auth()
4          .form("user", "password", formConfig)
5          .get(ROOT_URI + "/book-service/books/1");
6
7       Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
8       Assert.assertNotNull(response.getBody());
9   }
```

Now, we will create new Rating:

```
1    @Test
2    public void whenAddNewRating_thenSuccess() {
3        Rating rating = new Rating(1L, 4);
4        Response ratingResponse = RestAssured.given().auth()
5          .form("admin", "admin", formConfig).and()
6          .contentType(ContentType.JSON)
7          .body(rating)
8          .post(ROOT_URI + "/rating-service/ratings");
9        Rating result = ratingResponse.as(Rating.class);
10
11       Assert.assertEquals(HttpStatus.OK.value(), ratingResponse.getStatusCode());
12       Assert.assertEquals(rating.getBookId(), result.getBookId());
13       Assert.assertEquals(rating.getStars(), result.getStars());
14   }
```

Now try access admin protected *Rating* resource:

```
1    @Test
2    public void whenAccessAdminProtectedResource_thenForbidden() {
3        Response response = RestAssured.given().auth()
4          .form("user", "password", formConfig)
5          .get(ROOT_URI + "/rating-service/ratings");
6
7        Assert.assertEquals(HttpStatus.FORBIDDEN.value(), response.getStatusCode());
8    }
```

Then access protected *Rating* by login using admin:

```
1    @Test
2    public void whenAdminAccessProtectedResource_thenSuccess() {
3        Response response = RestAssured.given().auth()
4          .form("admin", "admin", formConfig)
5          .get(ROOT_URI + "/rating-service/ratings");
6
7        Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
8        Assert.assertNotNull(response.getBody());
9    }
```

Finally, access discovery resources as the admin:

```
@Test
public void whenAdminAccessDiscoveryResource_thenSuccess() {
    Response response = RestAssured.given().auth()
        .form("admin", "admin", formConfig)
        .get(ROOT_URI + "/discovery");

    Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
}
```

- Create a local Git repository *application-config* in you HOME directory.
- Add the configuration properties files.
- Make sure to commit all changes in the local Git repository before running the servers.

```
1  git add -A

2  git commit -m "the initial configuration"
```

- Make sure to run the modules in the following order:

  - Run the configuration server "port 8081"

  - Then, run discovery server "port 8082"

  - After that, run gateway server "port 8080"

  - Finally, run resource servers "port 8084 , 8083"

We discussed how to build a simple REST API using Spring Boot and Spring Cloud

The source code for all modules presented is available [over on GitHub](#).

V2