A Microservice Architecture with Spring Boot and Spring Cloud

Baeldung

www.baeldung.com

Table of Contents

1. Overview

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

2. The Resources

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

```java
public class Book {
    private long id;
    private String title;
    private String author;

    // standard getters and setters
}
```

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

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

    // standard getters and setters
}
```

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

First, let's explore our Books API:

```
@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:*

```
@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:

```
http://localhost:8080/book-service/books
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:

```
@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()
          .antMatchers(HttpMethod.GET, "/books").permitAll()
          .antMatchers(HttpMethod.GET, "/books/*").permitAll()
          .antMatchers(HttpMethod.POST, "/books").hasRole("ADMIN")
          .antMatchers(HttpMethod.PATCH, "/books/*").hasRole("ADMIN")
          .antMatchers(HttpMethod.DELETE, "/books/*").hasRole("ADMIN")
          .anyRequest().authenticated()
          .and()
        .csrf()
          .disable();
    }
}
```

And then the *Rating* configuration:

```
@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

5. Cloud Configuration

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.

5.1. Configuration Server

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

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

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

```
@SpringBootApplication
@EnableConfigServer
@EnableEurekaClient
public class ConfigApplication {...}
```

And here is *application.properties:*

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

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

```
cd ~
mkdir application-config
cd application-config
git init
```

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

5.2. Discovery Server

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

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

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

```
@SpringBootApplication
@EnableEurekaServer
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:

```
@Configuration
public static class AdminSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) {
       http
       .sessionManagement()
         .sessionCreationPolicy(SessionCreationPolicy.NEVER).and()
       .httpBasic().disable()
       .authorizeRequests()
         .antMatchers(HttpMethod.GET, "/").hasRole("ADMIN")
         .antMatchers("/info", "/health").authenticated()
         .anyRequest().denyAll().and()
       .csrf().disable();
   }
}
```

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

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

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

```
spring.application.name=discovery
server.port=8082
eureka.instance.hostname=localhost
eureka.client.serviceUrl.defaultZone=
  http://discUser:discPassword@localhost:8082/eureka/
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
spring.redis.host=localhost
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

5.3. Gateway Server

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

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

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

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

And add a simple security configuration:

```
@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:

```
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:

```
spring.application.name=gateway
server.port=8080
eureka.client.region = default
eureka.client.registryFetchIntervalSeconds = 5
management.security.sessions=always

zuul.routes.book-service.path=/book-service/**
zuul.routes.book-service.sensitive-headers=Set-Cookie,Authorization
hystrix.command.book-service.execution.isolation.thread
  .timeoutInMilliseconds=600000
zuul.routes.rating-service.path=/rating-service/**
zuul.routes.rating-service.sensitive-headers=Set-Cookie,Authorization
hystrix.command.rating-service.execution.isolation.thread
  .timeoutInMilliseconds=600000
zuul.routes.discovery.path=/discovery/**
zuul.routes.discovery.sensitive-headers=Set-Cookie,Authorization
zuul.routes.discovery.url=http://localhost:8082
hystrix.command.discovery.execution.isolation.thread
  .timeoutInMilliseconds=600000

spring.redis.host=localhost
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:

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

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

```
@SpringBootApplication
@EnableEurekaClient
public class ServiceApplication {...}
```

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

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

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

```
spring.cloud.config.name=book-service
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/
```

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

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

and *bootstrap.properties*:

```
spring.cloud.config.name=rating-service
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/
```

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:

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

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

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

For our gateway server, it will be slightly different:

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

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

```
@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;
    }
}
```

7. Test REST API

Finally, we will test our REST API

First, a simple setup:

```java
private final String ROOT_URI = "http://localhost:8080";
private FormAuthConfig formConfig
  = new FormAuthConfig("/login", "username", "password");

@Before
public void setup() {
    RestAssured.config = config().redirect(
      RedirectConfig.redirectConfig().followRedirects(false));
}
```

Next, let's get all books:

```java
@Test
public void whenGetAllBooks_thenSuccess() {
    Response response = RestAssured.get(ROOT_URI + "/book-service/books");

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

Then, try to access protected resource without login:

```java
@Test
public void whenAccessProtectedResourceWithoutLogin_thenRedirectToLogin() {
    Response response = RestAssured.get(ROOT_URI + "/book-service/books/1");

    Assert.assertEquals(HttpStatus.FOUND.value(), response.getStatusCode());
    Assert.assertEquals("http://localhost:8080/login",
      response.getHeader("Location"));
}
```

Then, login and create new Book:

```
@Test
public void whenAddNewBook_thenSuccess() {
    Book book = new Book();
    book.setTitle("How to spring cloud");
    book.setAuthor("Baeldung");

    Response bookResponse = RestAssured.given()
      .auth()
      .form("admin", "admin", formConfig)
      .and()
      .contentType(ContentType.JSON)
      .body(book)
      .post(ROOT_URI + "/book-service/books");

    Book result = bookResponse.as(Book.class);

    Assert.assertEquals(HttpStatus.OK.value(), bookResponse.getStatusCode());
    Assert.assertEquals(book.getAuthor(), result.getAuthor());
    Assert.assertEquals(book.getTitle(), result.getTitle());
}
```

and then access the protected resource after login:

```
@Test
public void whenAccessProtectedResourceAfterLogin_thenSuccess() {
    Response response = RestAssured.given().auth()
      .form("user", "password", formConfig)
      .get(ROOT_URI + "/book-service/books/1");

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

Now, we will create new Rating:

```java
@Test
public void whenAddNewRating_thenSuccess() {
    Rating rating = new Rating();
    rating.setBookId(1L);
    rating.setStars(4);

    Response ratingResponse = RestAssured.given()
      .auth()
      .form("admin", "admin", formConfig)
      .and()
      .contentType(ContentType.JSON)
      .body(rating)
      .post(ROOT_URI + "/rating-service/ratings");

    Rating result = ratingResponse.as(Rating.class);

    Assert.assertEquals(HttpStatus.OK.value(), ratingResponse.getStatusCode());
    Assert.assertEquals(rating.getBookId(), result.getBookId());
    Assert.assertEquals(rating.getStars(), result.getStars());
}
```

Now try access admin protected *Rating* resource:

```java
@Test
public void whenAccessAdminProtectedResource_thenForbidden() {
    Response response = RestAssured.given().auth()
      .form("user", "password", formConfig)
      .get(ROOT_URI + "/rating-service/ratings");

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

Then access protected *Rating* by login using admin:

```
@Test
public void whenAdminAccessProtectedResource_thenSuccess() {
    Response response = RestAssured.given().auth()
      .form("admin", "admin", formConfig)
      .get(ROOT_URI + "/rating-service/ratings");

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

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.

```
git add -A
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”

9. Conclusion

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.