Exercise for OAuth2 security

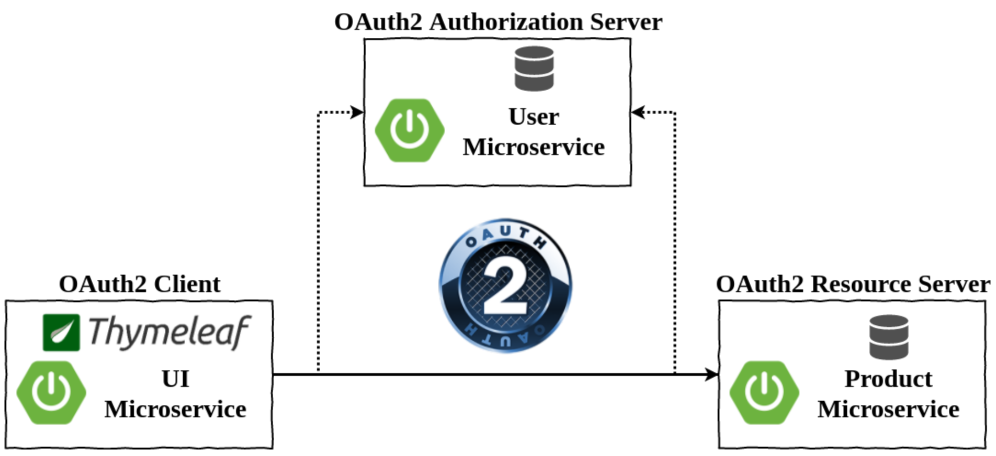
Andreas Falk

Table of Contents

Chapter 1. What we will build

We will extend the existing two microservices to use single sign authentication based on OAuth2.

- OAuth2 Authorization Server: This is the new microservice for single sign on which holds all users with their credentials
- OAuth2 Resource Server (Product Backend): The microservice providing product data maps to a resource server
- OAuth2 Client (UI Microservice): The thymeleaf UI microservice consuming the products maps to an OAuth2 client



These microservices have to be configured to be reachable via the following URL addresses (Port 8080 is the default port in spring boot).

Table 1. Microservice URL Adresses

Microservice	URL
Authorization Server	http://localhost:9999/users
Client (UI)	http://localhost:8081
Resource Server (Products)	http://localhost:8080

You can find more information on building OAuth2 secured microservices with spring in Spring Boot Reference Documentation and in Spring Security OAuth2 Developers Guide

Chapter 2. Step 1

In step 1 we will build a basic OAuth2 secured microservices infrastructure using simple user credentials provided via properties in *application.properties*.

2.1. Authorization Server

You may look into the spring boot reference documentation Spring Boot Reference Documentation on how to implement an authorization server.

To prevent conflicts with different JSESSION cookies the authorization server must run on a separate context path (not '/'). In our example please use '/users' as context path. In spring boot this can be achieved by the *server.context* property

To ensure OAuth2 authorization code grant works correctly with the other components the end points of the authorization server must be as follows:

Table 2. Authorization Server Endpoints

Endpoint	Description	Caller
/oauth/authorize	Authorization endpoint (for login and client authorization)	Client
/oauth/token	Token endpoint (exchanges given authorization code for access token)	Client
/oauth/check_token	Check token endpoint (returns internal contents for access token)	Resource Server

2.1.1. Maven dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>authorizationserver</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>authorizationserver</name>
    <description>OAuth2 Authorization Server</description>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <spring-cloud.version>Dalston.SR1</spring-cloud.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId> ①
            <artifactId>spring-cloud-starter-oauth2</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-hateoas</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
        </dependency>
        <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>${spring-cloud.version}</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>

</project>
```

① Dependency for OAuth2 security support

2.1.2. Java Implementation

```
@EnableAuthorizationServer ①
@SpringBootApplication
public class AuthorizationServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(AuthorizationServerApplication.class, args);
    }

}
```

① Annotation to enable auto configuration of an Authorization Server

2.1.3. Configuration

```
server.port=9999
server.context-path=/users

security.user.name=user ①
security.user.password=secret ②

security.oauth2.client.client-id=productclient ③
security.oauth2.client.client-secret=secretkey ④
security.oauth2.client.scope=read-products ⑤
security.oauth2.authorization.check-token-access=isAuthenticated() ⑥
```

① Definition of username for authentication

② Definition of user password for authentication

③ Client id for OAuth2 authorization code grant

④ Client secret for OAuth2 authorization code grant

⑤ Scopes to authorize for OAuth2

⑥ Open endpoint for getting token details for authenticated users

2.2. Resource Server (Products)

2.2.1. Maven dependencies

Add the following required dependencies to the existing maven pom file.

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security.oauth</groupId>
        <artifactId>spring-security-oauth2</artifactId>
    </dependency>
```

2.2.2. Java Implementation

The existing products service now should act as an OAuth2 resource server. Therefore it has to be marked as such.

```
@EnableResourceServer ①
@SpringBootApplication
public class ProductApplication {

    ...

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

① Annotation to enable this as a OAuth2 resource server.

2.2.3. Configuration

The following additional properties are required to make the resource server work with our new authorization server.

```
security.user.password=none ①

security.oauth2.resource.token-info-uri=http://localhost:9999/users/oauth/check_token
②
security.oauth2.client.client-id=productclient ③
security.oauth2.client.client-secret=secretkey ④
```

① This password won't be used as this microservice is protected by OAuth2 now

② Endpoint for getting token details required for products resource server

③ Client id for OAuth2 authorization code grant

④ Client secret for OAuth2 authorization code grant

2.3. OAuth2 Client (Thymeleaf UI)

2.3.1. Maven dependencies

Add the following required dependencies to the existing maven pom file.

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-oauth2</artifactId>
    </dependency>
```

2.3.2. Java Implementation

The UI client now acts as OAuth2 client and must be marked as such. Additionally to automatically use the OAuth2 tokens with all calls to REST services the currently used *RestTemplate* has to be replaced with new *OAuth2RestTemplate*

In more advanced cloud scenarios you may also use Feign Client instead of *OAuth2RestTemplate* to call the REST Api. In that case you have to add an interceptor for OAuth2.

```
@EnableOAuth2Sso ①
@SpringBootApplication
public class UiApplication {

    public static void main(String[] args) {
        SpringApplication.run(UiApplication.class, args);
    }

    @Bean
    public OAuth2RestTemplate oauth2RestTemplate(OAuth2ClientContext
oauth2ClientContext, ②
                                                 OAuth2ProtectedResourceDetails
details) {
        return new OAuth2RestTemplate(details, oauth2ClientContext);
    }
}
```

① Add *@EnableOAuth2Sso* annotation to secure complete UI using OAuth2

② Add new *@Bean* configuration for OAuth2RestTemplate (will replace the standard *RestTemplate*)

```
@Service
public class ProductService {

    //private RestTemplate template = new RestTemplate();
    private final OAuth2RestTemplate template; ①

    @Autowired
    public ProductService(OAuth2RestTemplate template) {
        this.template = template;
    }

    @HystrixCommand(fallbackMethod = "fallbackProducts",
            commandProperties = {  ②
                    @HystrixProperty(name="execution.isolation.strategy",
value="SEMAPHORE")
            })
    public Collection<Product> getAllProducts() {

        ResponseEntity<Product[]> response = template.getForEntity(
                "http://localhost:8080/products", Product[].class);

        return Arrays.asList(response.getBody());
    }

    public Collection<Product> fallbackProducts() {
        return Collections.emptyList();
    }
}
```

① Replace standard *RestTemplate* with *OAuth2RestTemplate*

② Reconfigure HystrixCommand to use Semaphore to propagate the security context

2.3.3. Configuration

The following additional properties are required to make the UI client work with our new authorization server.

```
server.port=8081

security.user.password=none ①

security.oauth2.client.client-id=productclient ②
security.oauth2.client.client-secret=secretkey ③
security.oauth2.client.access-token-uri=http://localhost:9999/users/oauth/token ④
security.oauth2.client.user-authorization-
uri=http://localhost:9999/users/oauth/authorize ⑤
security.oauth2.client.scope=read-products ⑥
security.oauth2.resource.token-info-uri=http://localhost:9999/users/oauth/check_token
⑦
```

① This password won't be used as this microservice is protected by OAuth2 now

② Client id for OAuth2 authorization code grant

③ Client secret for OAuth2 authorization code grant

④ Endpoint for exchanging authorization code for access token

⑤ Endpoint for redirecting to authorization server for authentication

⑥ Scope for the request

⑦ Endpoint for validating token details

Chapter 3. Step 2

To make the sample application even more secure we will enhance the authorization server to…

- …enable login using a form login page
- …use a persistent store for users
- …encrypt the passwords

3.1. Provide form based login

A form based login is more user friendly authentication and should always be preferred over basic authentication popups. To provide an automatically generated form based login just extend the predefined class *WebSecurityConfigurerAdapter*.

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {
}
```

3.2. Use persistent store

Until now we have authenticated ourselves using credentials that are defined via application properties which is ok for demo purposes but definetly not for production use.

Therefore we introduce a new persistent User entity which has the following attributes:

- firstname
- lastname
- email
- password

To actually use the new *User* entity with spring security this entity class has to implement the predefined interface *UserDetails* as well. Please use the attribute *email* as *username*.

A corresponding repository interface and a data initializer component (just like the one for products) have to be implemented as well.

Finally spring security must be aware to use the persistent users now instead of the one defined in the properties. To achieve this you have to implement the interface *UserDetailsService* as well to provide an operation loading a user for authentication.

```
@Service
public class StandardUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    public StandardUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String userName) throws
UsernameNotFoundException {
        User user = userRepository.findOneByEmail(userName);
        if (user == null) {
            throw new UsernameNotFoundException("No user found for " + userName);
        } else {
            return user;
        }
    }
}
```

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    private final UserDetailsService userDetailsService;

    @Autowired
    public WebSecurityConfiguration(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService);
    }
}
```

3.3. Encrypt the passwords

In the last step we want to change the storage of cleartext passwords for the users in the database. If any attacker gets hold of our database then all passwords are leaked. Therefore a production system must **ALWAYS** store any sensible data like passwords, credit card numbers etc. encrypted using a proven cryptographic algorithm.

```
public interface PasswordEncoder {

    String encode(CharSequence rawPassword);

    boolean matches(CharSequence rawPassword, String encodedPassword);

}
```

Lucklily spring security already provides safe implementations of a PasswordEncoder:

- Pbkdf2PasswordEncoder
- BCryptPasswordEncoder
- SCryptPasswordEncoder

To support authentication using encrypted passwords the current web security configuration has to be extended by the *PasswordEncoder* to be used.

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    private final UserDetailsService userDetailsService;

    @Autowired
    public WebSecurityConfiguration(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

The passwords have to be stored in encrypted form as well, so don't forget to inject and use a *PasswordEncoder* instance in your user data initializer.

You may use the h2 console to have a look into the in-memory database to see that the user passwords are really stored as encrypted values now. See Spring Boot Reference Docs on how to use and configure this.