Secure Spring REST With Spring Security and OAuth2

In this post, we are going to demonstrate Spring Security + OAuth2 for securing REST API endpoints on an example Spring Boot project. Clients and user credentials will be stored in a relational database (example configurations prepared for H2 and PostgreSQL database engines). To do it we will have to:

- Configure Spring Security + database.
- Create an Authorization Server.
- Create a Resource Server.
- Get an access token and a refresh token.
- Get a secured Resource using an access token.

To simplify the demonstration, we are going to combine the Authorization Server and Resource Server in the same project. As a grant type, we will use a password (we will use BCrypt to hash our passwords).

Before you start you should familiarize yourself with OAuth2 fundamentals.

Introduction

The OAuth 2.0 specification defines a delegation protocol that is useful for conveying authorization decisions across a network of web-enabled applications and APIs. OAuth is used in a wide variety of applications, including providing mechanisms for user authentication.

OAuth Roles

OAuth specifies four roles:

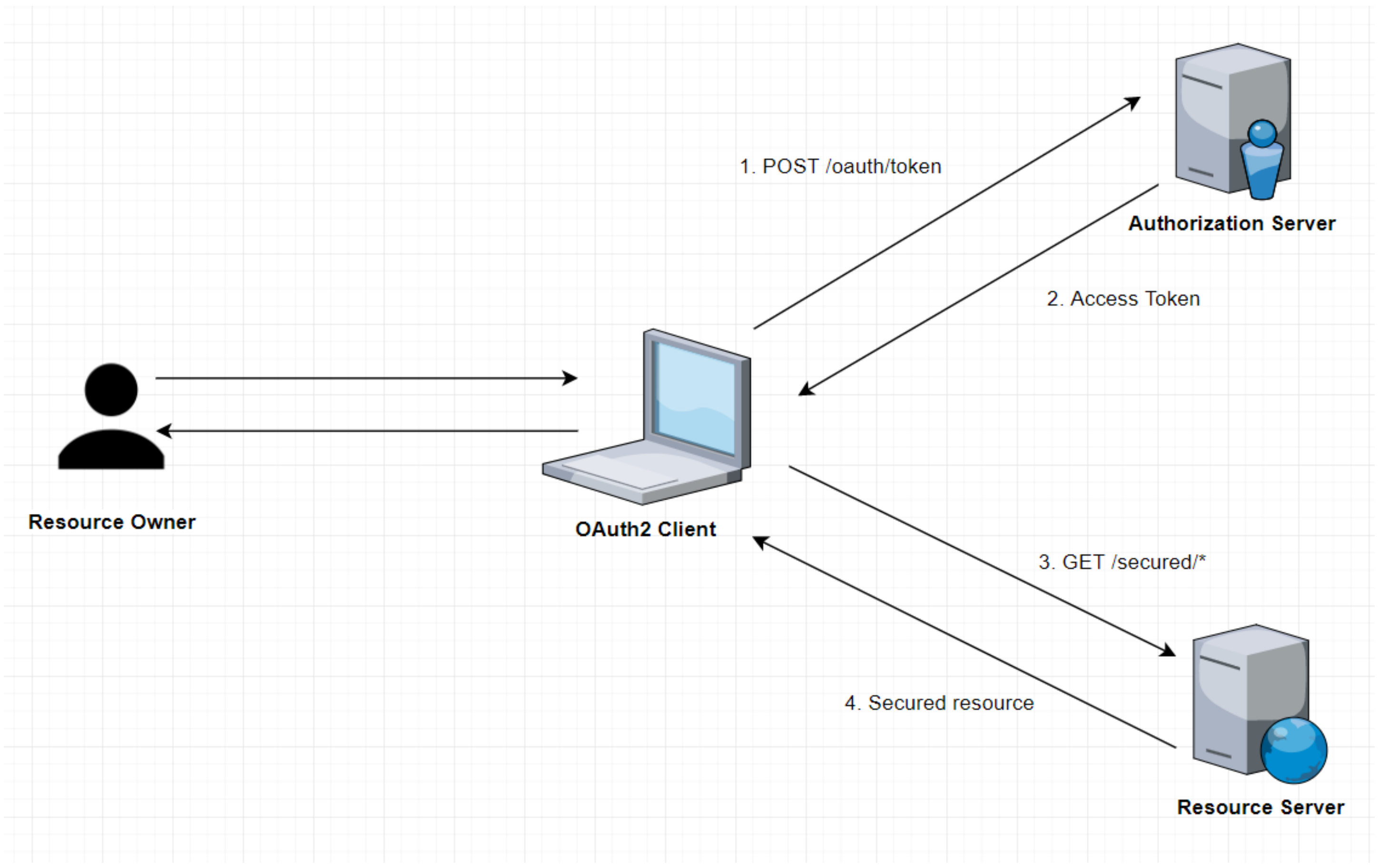
- **Resource owner (the User)** – an entity capable of granting access to a protected resource (for example end-user).
- **Resource server (the API server)** – the server hosting the protected resources, capable of accepting responding to protected resource requests using access tokens.
- **Client** – an application making protected resource requests on behalf of the resource owner and with its authorization.
- **Authorization server** – the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Grant Types

OAuth 2 provides several “grant types” for different use cases. The grant types defined are:

- **Authorization Code**
- **Password**
- **Client credentials**
- **Implicit**

The overall flow of a Password Grant:

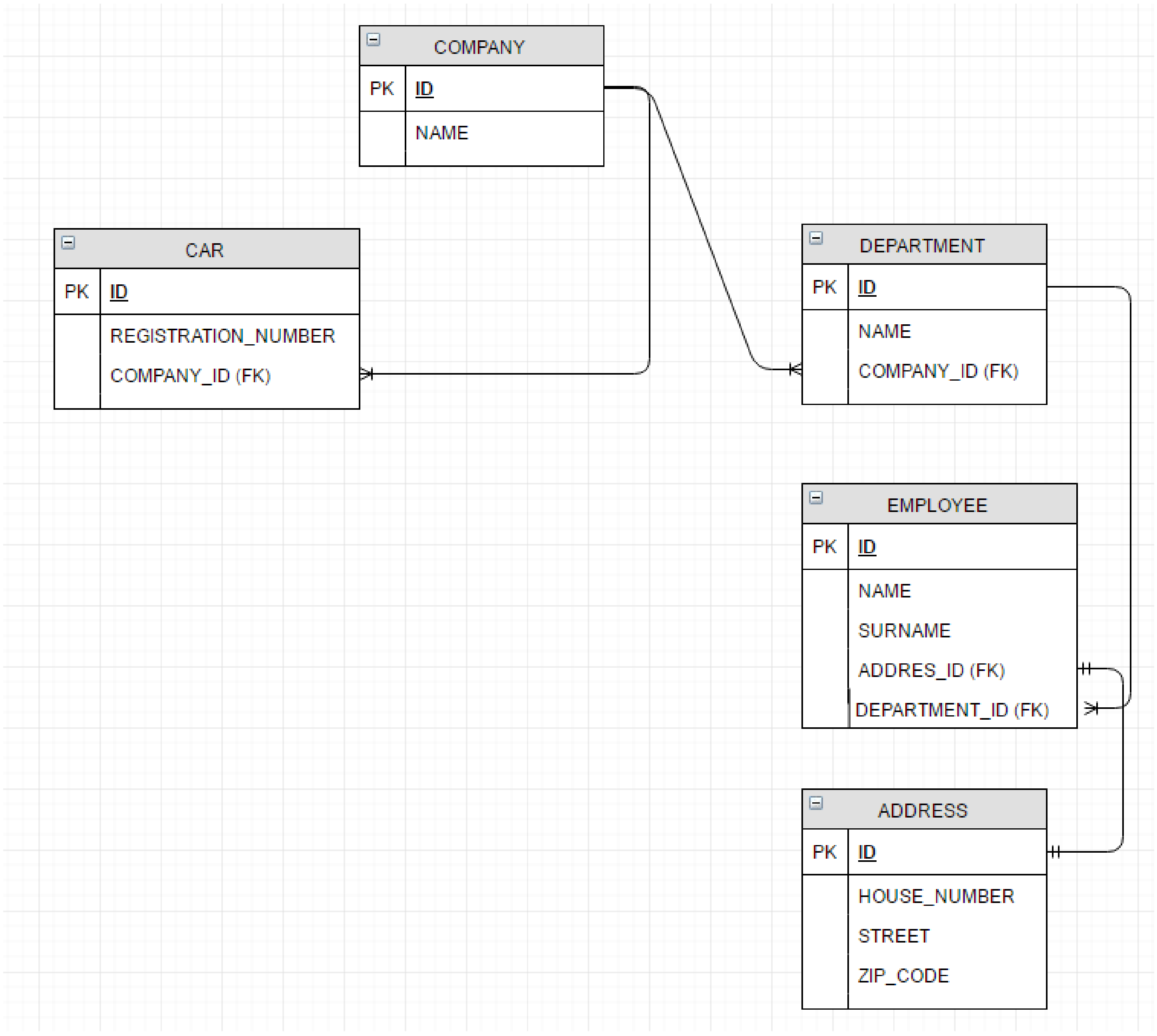


Application

Let's consider the database layer and application layer for our example application.

Business Data

Our main business object is Company:



Based on CRUD operations for Company and Department objects ,we want to define following access rules:

- COMPANY_CREATE
- COMPANY_READ

- COMPANY_UPDATE
- COMPANY_DELETE
- DEPARTMENT_CREATE
- DEPARTMENT_READ
- DEPARTMENT_UPDATE
- DEPARTMENT_DELETE

In addition, we want to create ROLE_COMPANY_READER role.

OAuth2 Client Setup

We need to create the following tables in the database (for internal purposes of OAuth2 implementation):

- OAUTH_CLIENT_DETAILS
- OAUTH_CLIENT_TOKEN
- OAUTH_ACCESS_TOKEN
- OAUTH_REFRESH_TOKEN
- OAUTH_CODE
- OAUTH_APPROVALS

Let’s assume that we want to call a resource server like ‘resource-server-rest-api.’ For this server, we define two clients called:

- spring-security-oauth2-read-client (authorized grant types: read)
- spring-security-oauth2-read-write-client (authorized grant types: read, write)

```
INSERT INTO OAUTH_CLIENT_DETAILS(CLIENT_ID, RESOURCE_IDS, C
 VALUES ('spring-security-oauth2-read-client', 'resource-se
 /*spring-security-oauth2-read-client-password1234*/'$2a$04
 'read', 'password,authorization_code,refresh_token,implici
INSERT INTO OAUTH_CLIENT_DETAILS(CLIENT_ID, RESOURCE_IDS, C
 VALUES ('spring-security-oauth2-read-write-client', 'resou
 /*spring-security-oauth2-read-write-client-password1234*/'
 'read,write', 'password,authorization_code,refresh_token,i
```

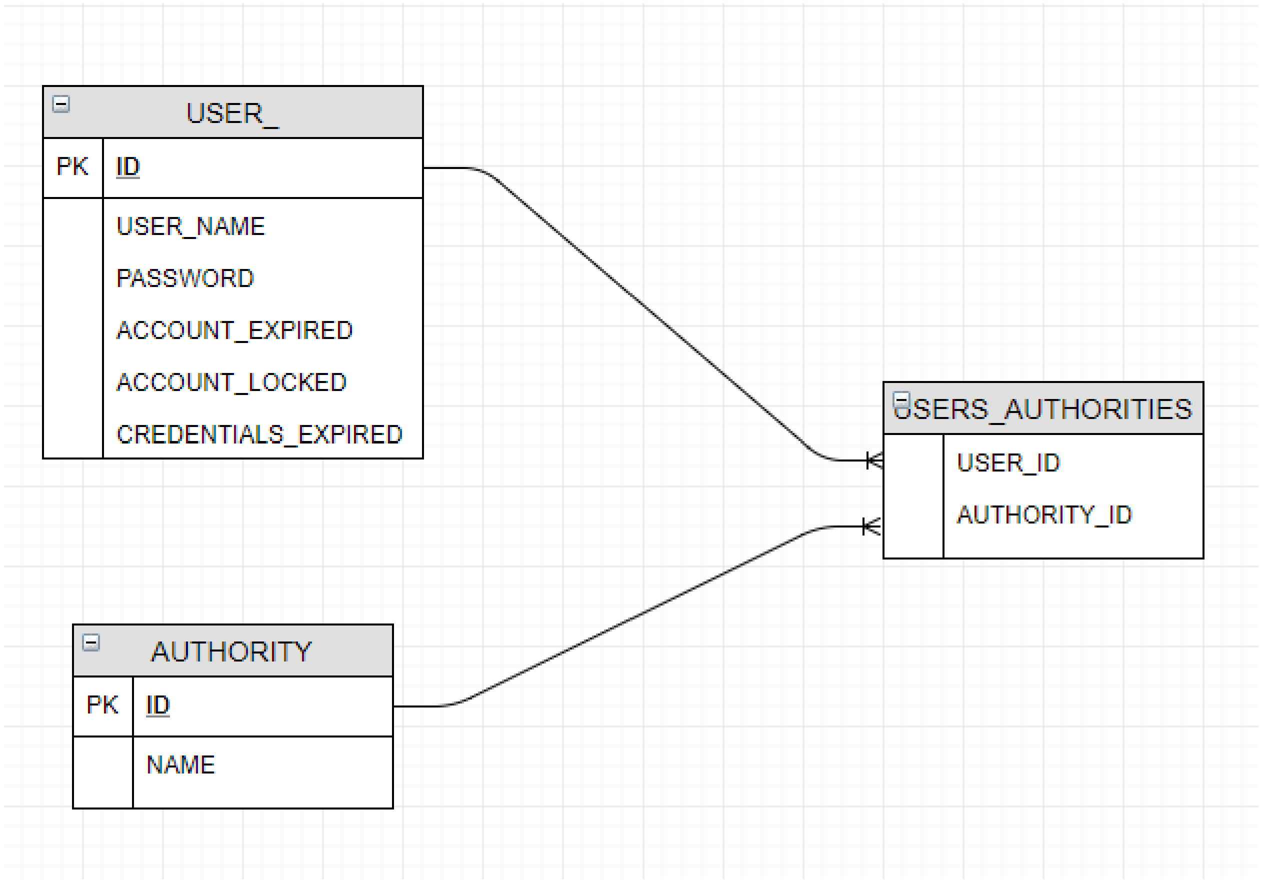
Note that password is hashed with BCrypt (4 rounds).

Authorities and Users Setup

Spring Security comes with two useful interfaces:

- UserDetails - provides core user information.
- GrantedAuthority - represents an authority granted to an Authentication object.

To store authorization data we will define following data model:



Because we want to come with some pre-loaded data, below is the script that will load all authorities:

```
RT INTO AUTHORITY(ID, NAME) VALUES (1, 'COMPANY_CREATE');
RT INTO AUTHORITY(ID, NAME) VALUES (2, 'COMPANY_READ');
RT INTO AUTHORITY(ID, NAME) VALUES (3, 'COMPANY_UPDATE');
RT INTO AUTHORITY(ID, NAME) VALUES (4, 'COMPANY_DELETE');
RT INTO AUTHORITY(ID, NAME) VALUES (5, 'DEPARTMENT_CREATE');
RT INTO AUTHORITY(ID, NAME) VALUES (6, 'DEPARTMENT_READ');
RT INTO AUTHORITY(ID, NAME) VALUES (7, 'DEPARTMENT_UPDATE');
RT INTO AUTHORITY(ID, NAME) VALUES (8, 'DEPARTMENT_DELETE');
```

Here is the script to load all users and assigned authorities:

```
UNT_LOCKED, CREDENTIALS_EXPIRED, ENABLED)
.l83Cd0jNsX6AJUitbgRXGzge4j035ha', FALSE, FALSE, FALSE, TRUE
UNT_LOCKED, CREDENTIALS_EXPIRED, ENABLED)
S4u19LHKW7aCQ0LXXuNlRfjjGKwj5NfKSe', FALSE, FALSE, FALSE, TR
UNT_LOCKED, CREDENTIALS_EXPIRED, ENABLED)
L4FtQH.mhMn7ZAFBYKB3ROz.J24IX8vDAcThsG', FALSE, FALSE, FALSE
UNT_LOCKED, CREDENTIALS_EXPIRED, ENABLED)
Tu/.J25iUCrpGBpyGExA.9yI.IlDRadR6Ea', FALSE, FALSE, FALSE, T
1);
2);
3);
4);
5);
6);
7);
8);
9);
2);
6);
3);
7);
9);
```

Note that the password is hashed with BCrypt (8 rounds).

Application Layer

The test application is developed in Spring boot + Hibernate + Flyway with an exposed REST API. To demonstrate data company operations, the following endpoints were created:

```
@RestController
@RequestMapping("/secured/company")
public class CompanyController {
    @Autowired
    private CompanyService companyService;
    @RequestMapping(method = RequestMethod.GET, produces = 
    @ResponseStatus(value = HttpStatus.OK)
    public @ResponseBody
    List<Company> getAll() {
        return companyService.getAll();
    }
    @RequestMapping(value = "/{id}", method = RequestMethod
    @ResponseStatus(value = HttpStatus.OK)
    public @ResponseBody
    Company get(@PathVariable Long id) {
        return companyService.get(id);
    }
    @RequestMapping(value = "/filter", method = RequestMeth
    @ResponseStatus(value = HttpStatus.OK)
    public @ResponseBody
    Company get(@RequestParam String name) {
        return companyService.get(name);
    }
    @RequestMapping(method = RequestMethod.POST, produces =
    @ResponseStatus(value = HttpStatus.OK)
    public ResponseEntity<?> create(@RequestBody Company co
        companyService.create(company);
        HttpHeaders headers = new HttpHeaders();
        ControllerLinkBuilder linkBuilder = linkTo(methodOn
        headers.setLocation(linkBuilder.toUri());
        return new ResponseEntity<>(headers, HttpStatus.CRE
    }
    @RequestMapping(method = RequestMethod.PUT, produces = 
    @ResponseStatus(value = HttpStatus.OK)
    public void update(@RequestBody Company company) {
        companyService.update(company);
    }
    @RequestMapping(value = "/{id}", method = RequestMethod
    @ResponseStatus(value = HttpStatus.OK)
    public void delete(@PathVariable Long id) {
        companyService.delete(id);
    }
}
```

PasswordEncoders

Since we are going to use different encryptions for OAuth2 client and user, we will define separate password encoders for encryption:

- OAuth2 client password – BCrypt (4 rounds)
- User password - BCrypt (8 rounds)

```
@Configuration
public class Encoders {
    @Bean
    public PasswordEncoder oauthClientPasswordEncoder() {
        return new BCryptPasswordEncoder(4);
    }
    @Bean
    public PasswordEncoder userPasswordEncoder() {
        return new BCryptPasswordEncoder(8);
    }
}
```

Spring Security Configuration

Provide UserDetailsService

Because we want to get users and authorities from the database, we need to tell Spring Security how to get this data. To do it we have to provide an implementation of the UserDetailsService interface:

```
@Service
public class UserDetailsServiceImpl implements UserDetailsS
    @Autowired
    private UserRepository userRepository;
    @Override
    @Transactional(readOnly = true)
    public UserDetails loadUserByUsername(String username) 
        User user = userRepository.findByUsername(username)
        if (user != null) {
            return user;
        }
        throw new UsernameNotFoundException(username);
    }
}
```

To separate the service and repository layers we will create UserRepository with JPA Repository:

```
@Repository
public interface UserRepository extends JpaRepository<User,
    @Query("SELECT DISTINCT user FROM User user " +
            "INNER JOIN FETCH user.authorities AS authoriti
            "WHERE user.username = :username")
    User findByUsername(@Param("username") String username)
}
```

Setup Spring Security

The @EnableWebSecurity annotation and WebSecurityConfigurerAdapter work together to provide security to the application. The @Order annotation is used to specify which WebSecurityConfigurerAdapter should be considered first.

```
@Configuration
@EnableWebSecurity
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
@Import(Encoders.class)
public class ServerSecurityConfig extends WebSecurityConfig
    @Autowired
    private UserDetailsService userDetailsService;
    @Autowired
    private PasswordEncoder userPasswordEncoder;
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean(
        return super.authenticationManagerBean();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder a
        auth.userDetailsService(userDetailsService).passwor
    }
}
```

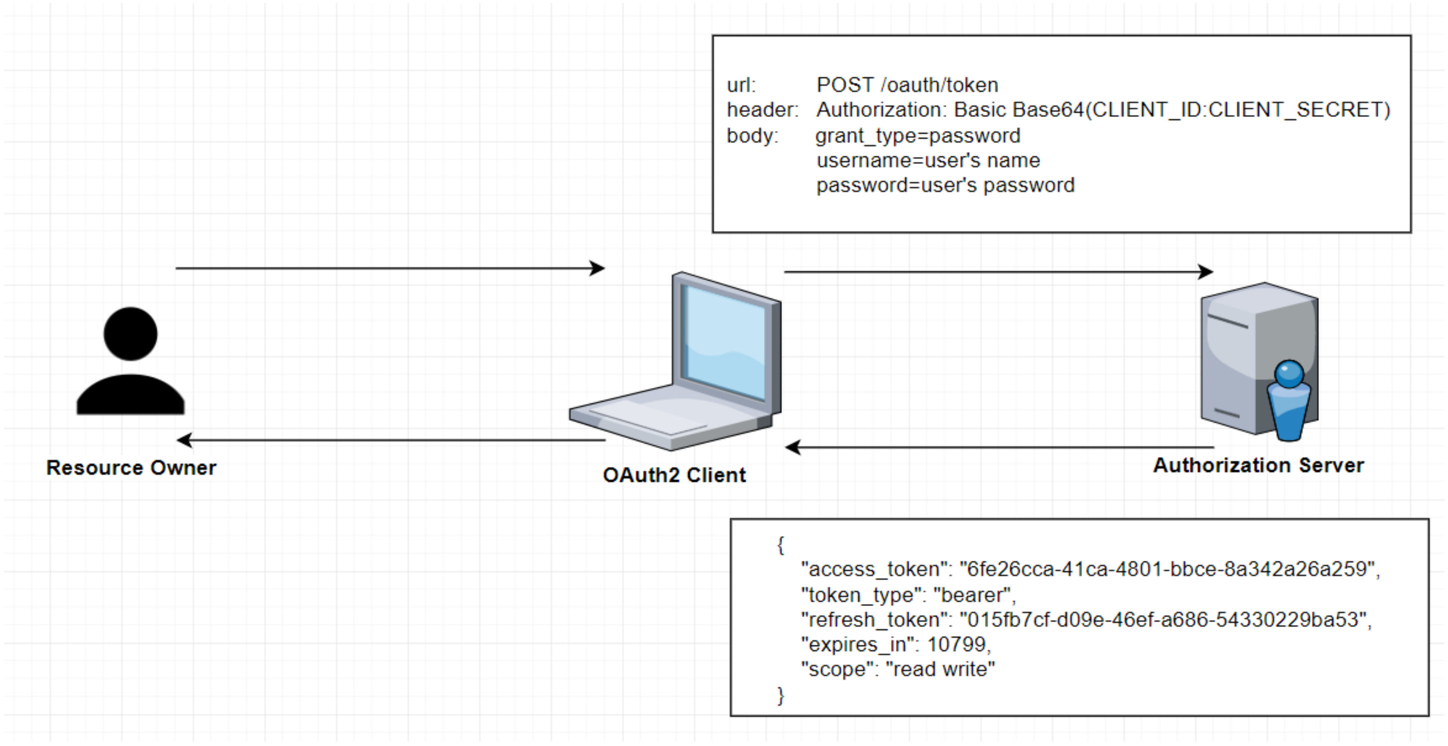
OAuth2 Configuration

First of all, we have to implement the following components:

- Authorization Server
- Resource Server

Authorization Server

The authorization server is responsible for the verification of user identity and providing the tokens.



```
url:      POST /oauth/token
header:   Authorization: Basic Base64(CLIENT_ID:CLIENT_SECRET)
body:     grant_type=password
          username=user's name
          password=user's password
```

```
{
  "access_token": "6fe26cca-41ca-4801-bbce-8a342a26a259",
  "token_type": "bearer",
  "refresh_token": "015fb7cf-d09e-46ef-a686-54330229ba53",
  "expires_in": 10799,
  "scope": "read write"
}
```

Spring Security handles the Authentication and Spring Security OAuth2 handles the Authorization. To configure and enable the OAuth 2.0 Authorization Server we have to use @EnableAuthorizationServer annotation.

```
@Configuration
@EnableAuthorizationServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
@Import(ServerSecurityConfig.class)
public class AuthServerOAuth2Config extends AuthorizationSe
```

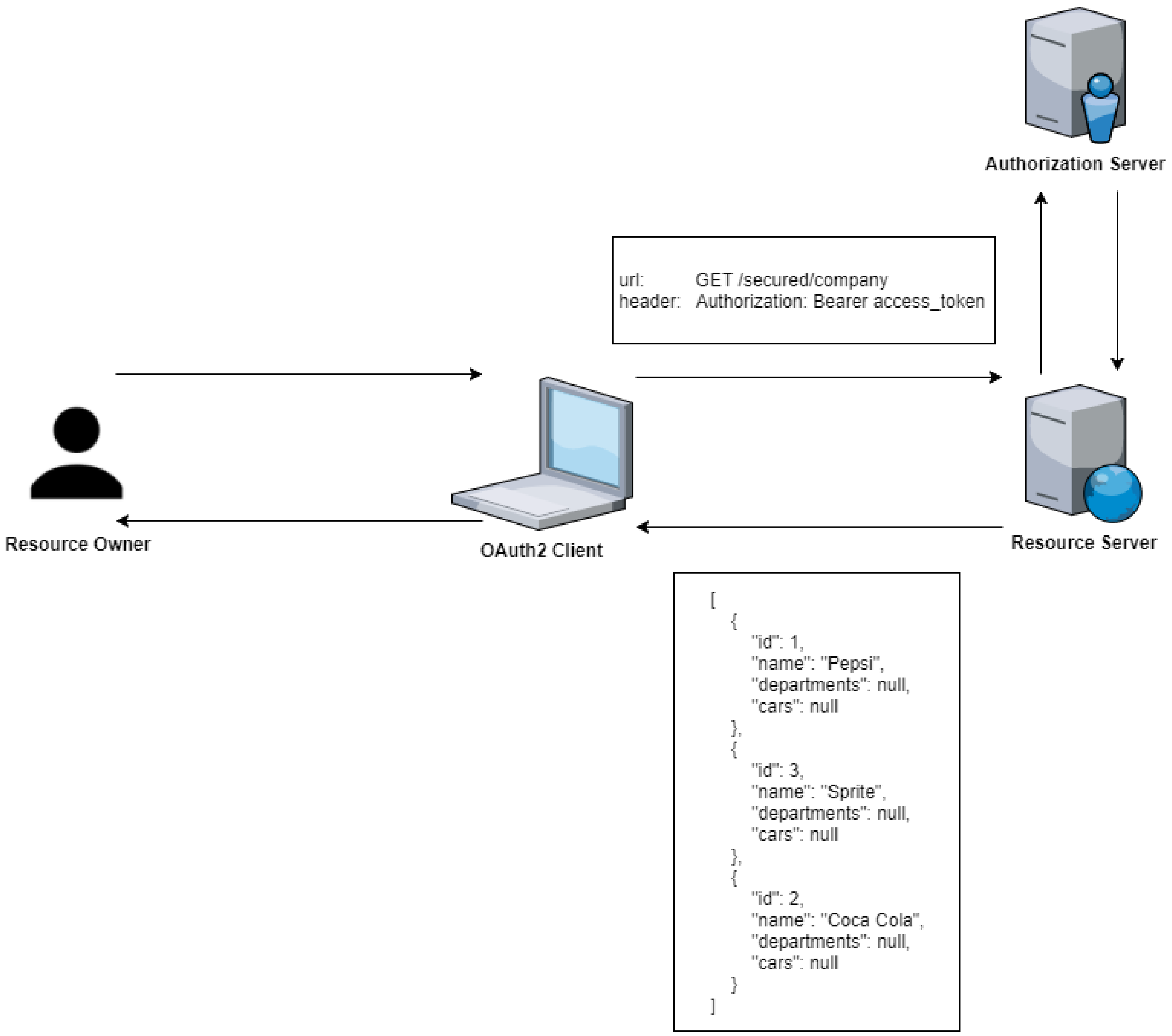
```
    @Autowired
    @Qualifier("dataSource")
    private DataSource dataSource;
    @Autowired
    private AuthenticationManager authenticationManager;
    @Autowired
    private UserDetailsService userDetailsService;
    @Autowired
    private PasswordEncoder oauthClientPasswordEncoder;
    @Bean
    public TokenStore tokenStore() {
        return new JdbcTokenStore(dataSource);
    }
    @Bean
    public OAuth2AccessDeniedHandler oauthAccessDeniedHandl
        return new OAuth2AccessDeniedHandler();
    }
    @Override
    public void configure(AuthorizationServerSecurityConfig
        oauthServer.tokenKeyAccess("permitAll()").checkToke
    }
    @Override
    public void configure(ClientDetailsServiceConfigurer cl
        clients.jdbc(dataSource);
    }
    @Override
    public void configure(AuthorizationServerEndpointsConfi
        endpoints.tokenStore(tokenStore()).authenticationMa
    }
}
```

Some important points. We:

- Defined the TokenStore bean to let Spring know to use the database for token operations.
- Override the configure methods to use the custom UserDetailsService implementation, AuthenticationManager bean, and OAuth2 client's password encoder.
- Defined handler bean for authentication issues.
- Enabled two endpoints for checking tokens (/oauth/check_token and /oauth/token_key) by overriding the configure (AuthorizationServerSecurityConfigureroauthServer) method.

Resource Server

A Resource Server serves resources that are protected by the OAuth2 token.



```
url:      GET /secured/company
header:   Authorization: Bearer access_token
```

```
[
    {
        "id": 1,
        "name": "Pepsi",
        "departments": null,
        "cars": null
    },
    {
        "id": 3,
        "name": "Sprite",
        "departments": null,
        "cars": null
    },
    {
        "id": 2,
        "name": "Coca Cola",
        "departments": null,
        "cars": null
    }
]
```

Spring OAuth2 provides an authentication filter that handles protection. The @EnableResourceServer annotation enables a Spring Security filter that authenticates requests via an incoming OAuth2 token.

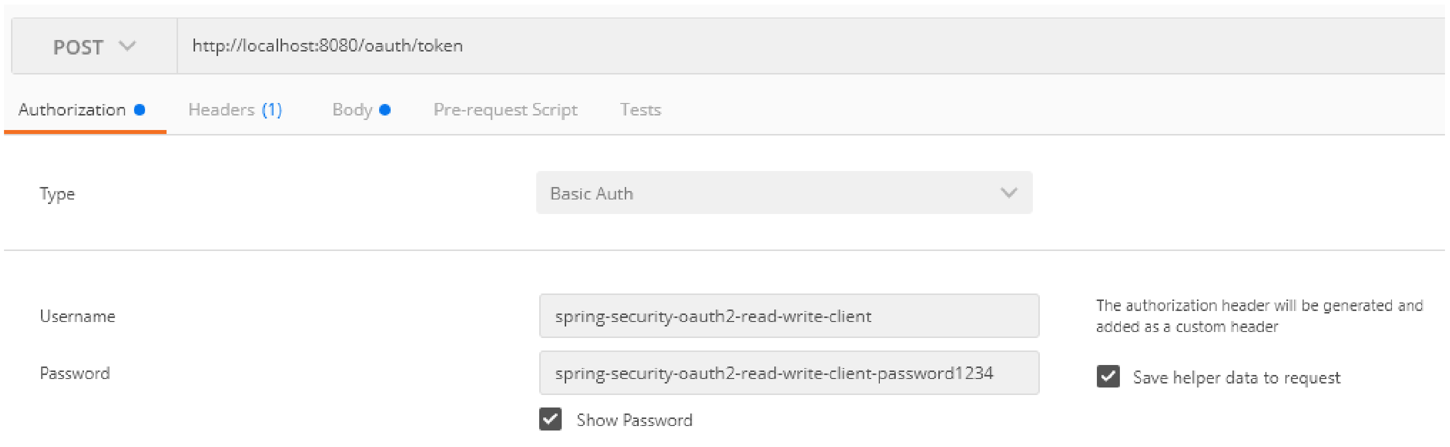
```
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceSe
    private static final String RESOURCE_ID = "resource-ser
    private static final String SECURED_READ_SCOPE = "#oaut
    private static final String SECURED_WRITE_SCOPE = "#oau
    private static final String SECURED_PATTERN = "/secured
    @Override
    public void configure(ResourceServerSecurityConfigurer 
        resources.resourceId(RESOURCE_ID);
    }
    @Override
    public void configure(HttpSecurity http) throws Excepti
        http.requestMatchers()
                .antMatchers(SECURED_PATTERN).and().authori
                .antMatchers(HttpMethod.POST, SECURED_PATTE
                .anyRequest().access(SECURED_READ_SCOPE);
    }
}
```

The configure(HttpSecurity http) method configures the access rules and request matchers (path) for protected resources using the HttpSecurity class. We secure the URL paths /secured/*. It’s worth noting that to invoke any POST method request, the ‘write’ scope is needed.

Let’s check if our authentication endpoint is working – invoke:

```
curl -X POST \
  http://localhost:8080/oauth/token \
  -H 'authorization: Basic c3ByaW5nLXNlY3VyaXR5LW9hdXRoMi1y
  -F grant_type=password \
  -F username=admin \
  -F password=admin1234 \
  -F client_id=spring-security-oauth2-read-write-client
```

Below are screenshots from Postman:



POST http://localhost:8080/oauth/token

Authorization ● | Headers (1) | Body ● | Pre-request Script | Tests

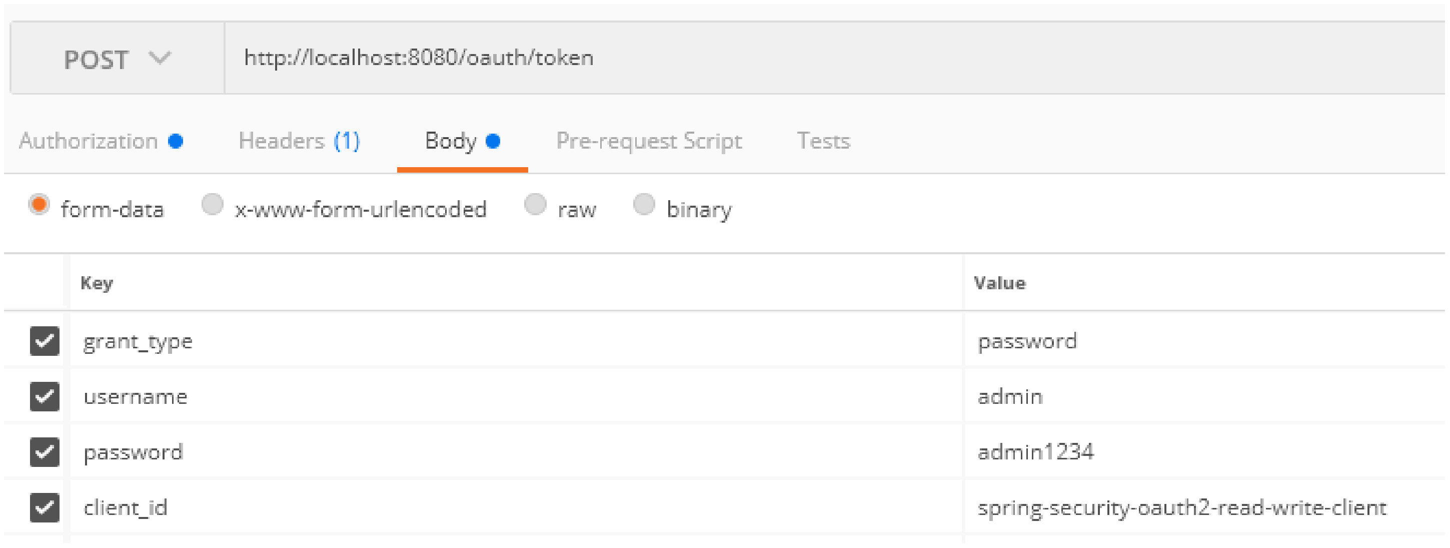
Type	Basic Auth
Username	spring-security-oauth2-read-write-client
Password	spring-security-oauth2-read-write-client-password1234

✓ Show Password

The authorization header will be generated and added as a custom header

✓ Save helper data to request

and:



POST http://localhost:8080/oauth/token

Authorization ● | Headers (1) | Body ● | Pre-request Script | Tests

[✓ form-data] x-www-form-urlencoded raw binary

	Key	Value
✓	grant_type	password
✓	username	admin
✓	password	admin1234
✓	client_id	spring-security-oauth2-read-write-client

You should get a response similar to the following:

```
{
    "access_token": "e6631caa-bcf9-433c-8e54-3511fa55816d",
    "token_type": "bearer",
    "refresh_token": "015fb7cf-d09e-46ef-a686-54330229ba53"
    "expires_in": 9472,
    "scope": "read write"
}
```

Access Rules Configuration

We decided to secure access to the Company and Department objects on the service layer. We have to use the @PreAuthorize annotation.

```
@Service
public class CompanyServiceImpl implements CompanyService {
    @Autowired
    private CompanyRepository companyRepository;
    @Override
    @Transactional(readOnly = true)
    @PreAuthorize("hasAuthority('COMPANY_READ') and hasAuth
    public Company get(Long id) {
        return companyRepository.find(id);
    }
    @Override
    @Transactional(readOnly = true)
    @PreAuthorize("hasAuthority('COMPANY_READ') and hasAuth
    public Company get(String name) {
        return companyRepository.find(name);
    }
    @Override
    @Transactional(readOnly = true)
    @PreAuthorize("hasRole('COMPANY_READER')")
    public List<Company> getAll() {
        return companyRepository.findAll();
    }
    @Override
    @Transactional
    @PreAuthorize("hasAuthority('COMPANY_CREATE')")
    public void create(Company company) {
        companyRepository.create(company);
    }
    @Override
    @Transactional
    @PreAuthorize("hasAuthority('COMPANY_UPDATE')")
    public Company update(Company company) {
        return companyRepository.update(company);
    }
    @Override
    @Transactional
    @PreAuthorize("hasAuthority('COMPANY_DELETE')")
    public void delete(Long id) {
        companyRepository.delete(id);
    }
    @Override
    @Transactional
    @PreAuthorize("hasAuthority('COMPANY_DELETE')")
    public void delete(Company company) {
        companyRepository.delete(company);
    }
}
```

Let’s test if our endpoint is working fine:

```
-X GET \
p://localhost:8080/secured/company/ \
'authorization: Bearer e6631caa-bcf9-433c-8e54-3511fa55816d'
```

Let’s see what will happen if we authorize with it ‘spring-security-oauth2-read-client’ – this client has only the read scope defined.

```
curl -X POST \
  http://localhost:8080/oauth/token \
  -H 'authorization: Basic c3ByaW5nLXNlY3VyaXR5LW9hdXRoMi1y
  -F grant_type=password \
  -F username=admin \
  -F password=admin1234 \
  -F client_id=spring-security-oauth2-read-client
```

Then for the below request:

```
curl -X POST \
  http://localhost:8080/secured/company \
  -H 'authorization: Bearer f789c758-81a0-4754-8a4d-cbf6eea
  -H 'content-type: application/json' \
  -d '{
    "name": "TestCompany",
    "departments": null,
    "cars": null
}'
```

We are getting the following error:

```
{
    "error": "insufficient_scope",
    "error_description": "Insufficient scope for this resou
    "scope": "write"
}
```

Summary

In this blog post, we showed OAuth2 authentication with Spring. Access rights were defined straightforward – by establishing a direct connection between User and Authorities. To enhance this example we can add an additional entity – Role – to improve the structure of the access rights.

The source code for the above listings can be found in this GitHub project.

Written on February 23, 2018

12 Comments **adamzareba-blog** 1 **Login**

Favorite 2 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Julius Spencer • 4 years ago

Thank you for this, it has helped me understand the integration and I've gotten it pretty much all going.

1 ^ | ⌄ • Reply • Share ›

Adam Zaręba Mod ➦ Julius Spencer • 4 years ago
Thank you. I'm happy to hear that :)
^ | ⌄ • Reply • Share ›

Abdullah Al Qayum • 8 months ago
Greetings. Two Questions :
1. I want to use login form and Social Login using Oauth2 . Is this possible to complete the task without enabling own authorization server and resource server.
2. when using login in form I want to use JWT .. I want to use only spring boot library oauth2.client
Any suggestions
^ | ⌄ • Reply • Share ›

togaurav • a year ago
Thanks for the clear explanation, can you please also add the logout behavior.
^ | ⌄ • Reply • Share ›

Rajeev • 2 years ago • edited
Nice Article!.
But 1 doubt.. How we are able to access <http: localhost:8080="" oauth="" token=""> as we dont see it in code ??
^ | ⌄ • Reply • Share ›

Mohit Darmwal • 2 years ago
@Adam Zaręba amazing article ...thanks ...i was finally able to run it with latest Spring boot 2.2.X..
please keep your readers updated if you include Role layer in this project in future..
^ | ⌄ • Reply • Share ›

Ameeth • 3 years ago
Hi Adam,
Thank you for the tutorial it helped me to understand OAuth and implement with my project.
I have one doubt if in my existing application the User password which is stored in DB is encrypted by using MD4 and here we have used BCrypt. Is it possible to implement same OAuth project by using MD5 encryption?
^ | ⌄ • Reply • Share ›

Ameeth ➦ Ameeth • 3 years ago

I have implemented the same with MD5.
Thank you.

Reply • Share ›

Adam Zaręba Mod ➦ Ameeth • 3 years ago

Hi @Ameeth,
Thank you, I am glad to hear that.
Regarding your question - first of all, you should consider to do not use MD5, since it is less secure algorithm than BCrypt. But if you really want to use MD5 you have to (please remember these are draft codes):
1. Create own implementation that reuses

```
org.springframework.security.authentication.encoding.Md5Pass
```

class, like:

```
public class CustomMD5PasswordEncoder implements
PasswordEncoder {

@Override
public String encode(CharSequence rawPassword) {
return new
org.springframework.security.authentication.encoding.Md5Pass
.encodePassword(rawPassword.toString(), "salt");
```

see more

Reply • Share ›

Ameeth ➦ Adam Zaręba • 3 years ago

Hi @Adam Zaręba ,

Thank you for reply.

Adam I'm trying to create custom login page and not using spring's default login page.
This custom login page is for end user from where user will enter user name and password.
I think I have to edit
ResourceServerConfiguration...
@Override
public void configure(HttpSecurity http) throws Exception {
http.requestMatchers()
.antMatchers(SECURED_PATTERN).and().authoriz
.antMatchers(HttpMethod.POST,
SECURED_PATTERN).access(SECURED_WRITE_
.anyRequest().access(SECURED_READ_SCOPE).
.loginPage("/login");
}

I was trying with above code without success.

Thanks,
Ameeth

Reply • Share ›

Cargo me • 3 years ago

most clear demo ever with regards to this... even a bit out of date.