**Spring Security ACL Demo**

This project demonstrates how to use Spring Security's Access Control List (ACL) functionality to implement fine-grained authorization for domain objects.

Overview

Spring Security ACL allows you to control access to individual objects in your application, rather than just at the application level (e.g., by roles). This is useful when you need to define very specific permissions, such as "User A can edit Document 1, but not Document 2."

This demo provides a basic setup to help you understand and experiment with Spring Security ACL.

**Prerequisites**

* Java Development Kit (JDK): Version 21 or higher.
* Maven: Version 3.6.0 or higher.
* Database: A running instance of a database (e.g., MySQL, PostgreSQL, H2). The **application is configured to use H2 by default**.
* Basic understanding of Spring Security: Familiarity with concepts like authentication, authorization, roles, and Spring Security annotations.

**Getting Started**

1. Download this zip file in your <my-app> folder. Unzip and import it into your favourite IDE.
2. Configure the database (Optional):
   * If you want to use a database other than the default H2, update the database configuration in src/main/resources/application.properties. For example, for MySQL:
   * spring.datasource.url=jdbc:mysql://localhost:3306/your\_database\_name
   * spring.datasource.username=your\_mysql\_username
   * spring.datasource.password=your\_mysql\_password
   * spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
   * spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
   * Make sure the database exists and the user has the necessary privileges.
3. Build the project:
4. mvn clean install
5. Run the application: I used the following:  
   "**<** Maven install directory path**>**" **clean install** -Dmaven.test.skip=true

The application will start, and you can access it at   
1. <http://localhost:9192/my_login.html>  
2. Enter user name adminUser/adminPass1 as username /password.   
3. Enter some document name , and document content in the respective text boxes, Hit “Create”.  
You should be able to see some success message.  
4. If you **used another**, already created, user(without create permissions) credentials userReader1/userPass1, you **won’t be allowed** to create the document!  
These users are created in the file **DatabseInitilaizer.java**

**Additional stuff, not necessarily related to this application, but to Spring Security ACL**

**Key Features and How to Test Them**

This demo includes functionality for managing documents with ACLs. Here's how you can test the key features:

1. Authentication:
   * The application uses Spring Security for authentication. You'll need to authenticate before accessing secured resources. See the src/main/resources/data.sql file for default users and roles. Here are a couple of users:
     + userReader1 / password (Has the ROLE\_USER role)
     + adminUser1 / password (Has the ROLE\_ADMIN role)
   * Use a tool like curl or Postman to make HTTP requests. Include the Authorization header with a Basic Auth token. For example, in curl:
   * curl -v -u userReader1:password http://localhost:8080/some/secured/endpoint
2. Document Creation:
   * The POST /cDoc endpoint creates a new document. Only **users with** the ROLE\_ADMIN role can access this endpoint.
   * To test:
   * curl -v -u adminUser1:password \
   * -X POST http://localhost:8080/cDoc \
   * -d "pName=MyDocument&pContent=This is the content"
     + Verify that the response has a 201 Created status.
     + Try the same request with userReader1 and confirm that it works.
     + Try with a user that does not have either role and confirm that you get a 403 Forbidden error.
3. Securing the Document Creation Endpoint:
   * The @PreAuthorize("hasRole('ROLE\_ADMIN') or hasRole('ROLE\_USER')") annotation on the createDocument method in the controller enforces this authorization. This ensures that only users with the specified roles can create documents. This is the core of the ACL demo.

**Important Files and Classes**

* src/main/java/com/example/springsecurityacl/config/SecurityConfig.java: Spring Security configuration, including ACL setup.
* src/main/java/com/example/springsecurityacl/service/DocumentAclService.java: Service class that handles ACL operations for documents. This is where the ACLs are created and managed.
* src/main/java/com/example/springsecurityacl/web/controller/DocumentController.java: Controller for handling document-related requests.
* src/main/resources/application.properties: Application properties, including database configuration.
* src/main/resources/data.sql: Initial data, including users, roles, and initial ACL setup. This file is crucial for setting up your initial testing environment.

**Understanding the ACL Structure**

Spring Security ACL uses these core concepts:

* Object Identity (OI): Uniquely identifies the domain object (e.g., a specific document).
* Security Identity (SID): Represents who is granted permissions (a user or a role).
* Permission: Defines what actions can be performed on the object (e.g., read, write, delete).
* Access Control Entry (ACE): связывает OI, SID и Permission, specifying who can do what to which object.
* Acl: Holds all the ACEs for a given OI.

**Further Exploration**

* Custom Permissions: Explore how to define custom permissions beyond the basic ones (read, write, etc.).
* ACL Inheritance: Learn how ACLs can inherit permissions from parent objects.
* AclService: Dive deeper into the AclService interface and its implementations for managing ACLs.
* Method Security: Investigate how to use Spring Security ACL with method security annotations (@PreAuthorize, @PostAuthorize, @PreFilter, @PostFilter) for fine-grained control.

Enjoy experimenting with Spring Security ACL!

Social Login Integration(s)

Add Github login . Let me know all the steps, and the additional application.properties changes and mylogin.html page changes.

**1. Register Your Application with GitHub**

* Go to [GitHub Developer Settings](https://github.com/settings/developers).
* Click **"Register a new application"** or select an existing one.
* **Application name:** Choose a descriptive name.
* **Homepage URL:** Your application's base URL (e.g., http://localhost:8080 for local development).
* **Authorization callback URL:** This is crucial. For local development, it's typically: http://localhost:8080/login/oauth2/code/github
* In production, this will be your domain: https://your-domain.com/login/oauth2/code/github
* Click **"Register application"**.
* You'll get a **Client ID** and **Client Secret**. Keep these secure!

**2. Update application.properties or application.yml**

Add the GitHub client registration details:

**application.yml:**

Code snippet

spring:

security:

oauth2:

client:

registration:

google:

client-id: your-google-client-id

client-secret: your-google-client-secret

scope:

- openid

- email

- profile

github: # GitHub configuration

client-id: your-github-client-id

client-secret: your-github-client-secret

scope:

- read:user

- user:email

**application.properties:**

Properties

spring.security.oauth2.client.registration.google.client-id=your-google-client-id

spring.security.oauth2.client.registration.google.client-secret=your-google-client-secret

spring.security.oauth2.client.registration.google.scope=openid,email,profile

spring.security.oauth2.client.registration.github.client-id=your-github-client-id

spring.security.oauth2.client.registration.github.client-secret=your-github-client-secret

spring.security.oauth2.client.registration.github.scope=read:user,user:email

**3. Modify my\_login.html**

Add a link for GitHub login:

HTML

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Custom Login Page</title>

</head>

<body>

<h1>Welcome! Please Login</h1>

<form action="/login" method="post">

<div>

<label for="username">Username:</label>

<input type="text" id="username" name="username"/>

</div>

<div>

<label for="password">Password:</label>

<input type="password" id="password" name="password"/>

</div>

<div>

<input type="submit" value="Log in"/>

</div>

</form>

<hr/>

<h2>Or login with:</h2>

<div>

<a href="/oauth2/authorization/google">Login with Google</a>

</div>

<div>

<a href="/oauth2/authorization/github">Login with GitHub</a> </div>

<div th:if="${param.error}">

<p style="color: red;">Invalid username, password, or social login attempt.</p>

</div>

<div th:if="${param.logout}">

<p style="color: green;">You have been logged out.</p>

</div>

</body>

</html>

**4. Review Your SecurityConfig.java (Usually No Changes Needed)**

If you've followed the previous steps correctly, your SecurityConfig.java should already be set up to handle multiple OAuth2 providers. The key is the oauth2Login() configuration and your custom oauth2UserService():

Java

// ... (Your other imports and configurations) ...

@Bean

public OAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {

final DefaultOAuth2UserService delegate = new DefaultOAuth2UserService();

return (userRequest) -> {

OAuth2User oauth2User = delegate.loadUser(userRequest);

String email = oauth2User.getAttribute("email"); // Often available for Google, GitHub (if scope requested)

String name = oauth2User.getAttribute("name"); // User's display name

String sub = oauth2User.getName(); // Unique ID from the provider

// --- IMPORTANT: Your ACL/Role Assignment Logic Goes Here ---

Set<GrantedAuthority> authorities = new HashSet<>();

authorities.add(new SimpleGrantedAuthority("ROLE\_USER"));

// Example: Assign ROLE\_ADMIN based on email

if (email != null && email.equals("admin@example.com")) {

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

// ... (Your database lookup or other role assignment logic) ...

return new DefaultOAuth2User( authorities, oauth2User.getAttributes(), userRequest.getClientRegistration().getProviderDetails().getUserInfoEndpoint().getUserNameAttributeName()

);

};

}

// ... (Your other configurations) ...

* **No Changes (Usually):** You typically *don't* need to modify SecurityConfig.java further, as the existing oauth2Login() and oauth2UserService() are designed to handle multiple providers. The userRequest object within oauth2UserService() tells you which provider (Google, GitHub, etc.) is being used.

**5. Test GitHub Login**

1. Run your application.
2. Go to http://localhost:8080/my\_login.html.
3. You should now see a "Login with GitHub" link.
4. Click it, and you'll be redirected to GitHub for authorization.
5. After authorizing your application on GitHub, you'll be redirected back to your application.
6. Verify that you are logged in and that your application receives user details (name, email, etc.) from GitHub.

**Important Considerations:**

* **GitHub Scopes:** The read:user and user:email scopes are common for GitHub login. Adjust them based on the data you need.
* **Email Privacy on GitHub:** Users can keep their email address private on GitHub. If you rely on email for user identification or role assignment, handle cases where the email is not available.
* **Error Handling:** Implement proper error handling for OAuth2 login failures.
* **User Linking:** If you want users to be able to link *both* their Google and GitHub accounts to a *single* application account, you'll need more complex logic in your oauth2UserService() to handle this.
* **Testing:** Thoroughly test the login flow, including cases where the user denies access or if there are network issues.

By following these steps, you should successfully integrate GitHub login into your Spring Boot application alongside Google login.

[=---- Types of “USERs” .STARTs ----=]  
In this myriad of "users", help me understand the precise difference between the following three users:

1.   String sub = oauth2User.getName(); // The user's unique identifier from the IdP

2. "YOUR\_GITHUB\_USERNAME"

3. String githubUsername = (String) oauth2User.getAttributes().get("login");

You've hit upon a crucial point when working with OAuth2 and different identity providers (IdPs)! The "user" can be represented in various ways, and understanding the precise difference between these identifiers is key to correct user management and authorization.

Let's break down the three "users" you've listed:

1. **String sub = oauth2User.getName(); (The sub claim / OAuth2User.getName()):**
   * **What it is:** This sub (short for "subject") is a **standard claim** in the OpenID Connect (OIDC) specification. It represents the **unique identifier of the end-user at the Identity Provider (IdP)**. It is guaranteed to be unique for that user *at that specific IdP* and never reassigned.
   * **Origin:** It comes directly from the IdP's user information endpoint (the UserInfoEndpoint) as part of the OAuth2User's attributes. When you call oauth2User.getName(), Spring Security extracts the value of the attribute that the IdP designated as its "name attribute" (usually sub for OIDC-compliant providers like Google, sometimes id or user\_id for others, but Spring handles the mapping).
   * **Purpose:** This is the **most reliable and stable identifier** for a user *from the perspective of the IdP*. It should be your primary key if you are storing OAuth2 users in your own database, as it's immutable and unique.
   * **Example (Google):** For a Google user, oauth2User.getName() will return their unique Google user ID (a long string of numbers). This ID will never change for that Google account.
   * **Example (GitHub):** For a GitHub user, oauth2User.getName() will typically return their GitHub user ID (a numeric ID), not their username.
2. **"YOUR\_GITHUB\_USERNAME" (A Hardcoded Specific Username):**
   * **What it is:** This is a **literal, hardcoded string** that represents a specific, predefined GitHub username that ***you*** have designated as an "**admin**" in your application's logic.
   * **Origin:** It originates from your application's source code, typically as a constant or a configurable property.
   * **Purpose:** It's used for **direct comparison** against a user's ***actual* GitHub** username **retrieved from the IdP**. It's a way to say, "if the logged-in GitHub user's username is exactly this string, then they are an admin."
   * **Limitations:**
     + It's **not dynamic**: You have to change the code to change which GitHub username gets admin access.
     + It's **specific to GitHub**: It's meaningful only when logging in via GitHub.
     + It's **reliant on the username**: Usernames can change on GitHub (though less common for established users). If a user changes their GitHub username, they might lose admin access unless you update this hardcoded value.
3. **String githubUsername = (String) oauth2User.getAttributes().get("login"); (The GitHub login attribute):**
   * **What it is:** This githubUsername is a **specific attribute** provided by GitHub in its user information (UserInfo) response. The login attribute specifically refers to the user's **GitHub username** (e.g., "octocat", "johndoe").
   * **Origin:** It's extracted from the oauth2User.getAttributes() map, which contains all the raw user profile data returned by the GitHub UserInfo endpoint.
   * **Purpose:** It's primarily used for **display purposes** (e.g., "Welcome, octocat!") or for **human-readable identification** within your application. You're also using it to implement your authorization logic (e.g., if login matches "YOUR\_GITHUB\_USERNAME").
   * **Limitations:**
     + It's **provider-specific**: This attribute (login) is unique to GitHub's user info response. Google, for instance, won't have a "login" attribute with the same meaning.
     + It's **mutable**: While rare for established users, a GitHub user *can* change their username (login). If you rely solely on this for persistent identification or critical authorization, be aware that it might change, potentially affecting a user's access. This is why sub (or GitHub's internal numeric id) is generally preferred for stable linking to your internal user records.

**Summary Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Identifier** | **What it is** | **Origin/Source** | **Purpose** | **Key Characteristics** |
| **oauth2User.getName()** (e.g., sub claim) | **Unique ID of the user at the IdP.** | IdP's UserInfo endpoint (mapped by Spring Security). | **Primary key** for linking to your internal user records. Stable and immutable. | Provider-specific, **immutable**, globally unique for that IdP. Usually non-human-readable. |
| **"YOUR\_GITHUB\_USERNAME"** | **Hardcoded string** (specific username). | Your application's code. | Direct comparison for **specific, predefined** authorization. | Hardcoded, **mutable only by code change**, specific to GitHub. |
| **oauth2User.getAttributes().get("login")** | **GitHub username** (login attribute). | GitHub's UserInfo response. | Display name, human-readable identifier, or **dynamic authorization logic**. | Provider-specific, **mutable** (user can change), human-readable. |

[=---- Types of “USERs” .STARTs ----=]

[=------ Github user promotion to Admin STARTs ---=]  
Let's customize.

I wish to have the Github user being provided an admin level role to be able to create a document. What code mofiications do I need to make in SecurityConfig class, in particular?

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.oauth2.client.userinfo.DefaultOAuth2UserService;

import org.springframework.security.oauth2.client.userinfo.OAuth2UserRequest;

import org.springframework.security.oauth2.client.userinfo.OAuth2UserService;

import org.springframework.security.oauth2.core.OAuth2AuthenticationException;

import org.springframework.security.oauth2.core.user.DefaultOAuth2User;

import org.springframework.security.oauth2.core.user.OAuth2User;

import java.util.HashSet;

import java.util.Map;

import java.util.Set;

@Bean

public OAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {

final DefaultOAuth2UserService delegate = new DefaultOAuth2UserService();

return (userRequest) -> {

OAuth2User oauth2User = delegate.loadUser(userRequest);

String email = oauth2User.getAttribute("email");

String name = oauth2User.getAttribute("name");

String sub = oauth2User.getName();

Set<GrantedAuthority> authorities = new HashSet<>();

authorities.add(new SimpleGrantedAuthority("ROLE\_USER"));

// GitHub-specific admin check

if ("github".equals(userRequest.getClientRegistration().getRegistrationId())) {

String githubUsername = (String) oauth2User.getAttributes().get("login"); // Get GitHub username

// Grant admin role to a specific GitHub user

if ("YOUR\_GITHUB\_USERNAME".equals(githubUsername)) { // Replace "YOUR\_GITHUB\_USERNAME" with the actual username

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

}

// Google-specific admin check (keep existing logic)

if ("google".equals(userRequest.getClientRegistration().getRegistrationId())) {

if (email != null && email.equals("admin@example.com")) { // Replace with your actual admin email

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

}

return new DefaultOAuth2User(

authorities,

oauth2User.getAttributes(),

userRequest.getClientRegistration().getProviderDetails().getUserInfoEndpoint().getUserNameAttributeName()

);

};

}

**Explanation:**

* **userRequest.getClientRegistration().getRegistrationId():** This line retrieves the ID of the OAuth2 provider (e.g., "google", "github"). This allows you to apply different logic based on the provider.
* **oauth2User.getAttributes().get("login"):** For GitHub, the username is typically found in the login attribute. This line retrieves the GitHub username. Note that the attribute name might be different depending on the provider, so always inspect the oauth2User.getAttributes() map to confirm.
* **"YOUR\_GITHUB\_USERNAME".equals(githubUsername):** This is the core of the customization. **You must replace "YOUR\_GITHUB\_USERNAME" with the actual GitHub username** of the user you want to grant admin privileges to. This is a case-sensitive comparison.
* **authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));:** If the GitHub username matches, this line adds the ROLE\_ADMIN authority to the user's set of roles.
* **Google-specific logic:** The existing Google-specific logic (checking the email address) is preserved.

**How to Adapt for Other Criteria:**

* **Different GitHub Attributes:** GitHub provides various user attributes. If you want to use a different attribute (e.g., the user's ID), you'll need to find the correct key in the oauth2User.getAttributes() map. You can log the contents of this map to see what's available: System.out.println(oauth2User.getAttributes());
* **Multiple GitHub Users:** If you need to grant admin to multiple GitHub users, you can use a List or Set to store the usernames and check if the githubUsername is in that collection:

List<String> adminGithubUsers = Arrays.asList("user1", "user2", "user3");

if (adminGithubUsers.contains(githubUsername)) {

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

**Combining Criteria:** You can combine multiple criteria using && (AND) or || (OR) operators. For example, to grant admin to a specific GitHub user *and* a specific Google email:

if (("github".equals(userRequest.getClientRegistration().getRegistrationId()) && "YOUR\_GITHUB\_USERNAME".equals(githubUsername)) ||

("google".equals(userRequest.getClientRegistration().getRegistrationId()) && email != null && "admin@example.com".equals(email))) {

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

Remember to replace the placeholder values ("YOUR\_GITHUB\_USERNAME", "admin@example.com") with your actual values.

Can we add code to retrieve the Authentication token also?

Yes, you can absolutely retrieve the OAuth2 Access Token within your oauth2UserService and potentially store or use it. This token is what your application receives from the OAuth2 provider (Google, GitHub) after a successful authentication, and it's used to access the user's resources on that provider's API (e.g., fetching more user details, accessing their repositories, etc.).

Here's how you can modify your SecurityConfig to retrieve the access token:

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.oauth2.client.userinfo.DefaultOAuth2UserService;

import org.springframework.security.oauth2.client.userinfo.OAuth2UserRequest;

import org.springframework.security.oauth2.client.userinfo.OAuth2UserService;

import org.springframework.security.oauth2.core.OAuth2AuthenticationException;

import org.springframework.security.oauth2.core.user.DefaultOAuth2User;

import org.springframework.security.oauth2.core.user.OAuth2User;

import org.springframework.security.oauth2.core.OAuth2AccessToken; // Import this

import java.util.HashSet;

import java.util.Map;

import java.util.Set;

import org.slf4j.Logger; // For logging

import org.slf4j.LoggerFactory; // For logging

// ... (other imports and class definition)

public class SecurityConfig {

private static final Logger logger = LoggerFactory.getLogger(SecurityConfig.class);

// ... (your existing filterChain method and other beans) ...

@Bean

public OAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {

final DefaultOAuth2UserService delegate = new DefaultOAuth2UserService();

return (userRequest) -> {

OAuth2User oauth2User = delegate.loadUser(userRequest);

// \*\*\* RETRIEVE THE ACCESS TOKEN HERE \*\*\*

OAuth2AccessToken accessToken = userRequest.getAccessToken();

logger.info("Successfully received OAuth2 Access Token for {}: {}",

userRequest.getClientRegistration().getRegistrationId(),

accessToken.getTokenValue());

// You can log other token details if needed:

// logger.info("Token Type: {}", accessToken.getTokenType());

// logger.info("Expires At: {}", accessToken.getExpiresAt());

// logger.info("Scopes: {}", accessToken.getScopes());

String email = oauth2User.getAttribute("email");

String name = oauth2User.getAttribute("name");

String sub = oauth2User.getName(); // The user's unique identifier from the IdP

Set<GrantedAuthority> authorities = new HashSet<>();

authorities.add(new SimpleGrantedAuthority("ROLE\_USER"));

// GitHub-specific admin check

if ("github".equals(userRequest.getClientRegistration().getRegistrationId())) {

String githubUsername = (String) oauth2User.getAttributes().get("login");

if ("YOUR\_GITHUB\_USERNAME".equals(githubUsername)) {

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

}

// Google-specific admin check

if ("google".equals(userRequest.getClientRegistration().getRegistrationId())) {

if (email != null && email.equals("admin@example.com")) {

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

}

// If you need to store the accessToken or other user details for later use

// you can create a custom OAuth2User implementation or extend DefaultOAuth2User

// For now, let's just log it.

return new DefaultOAuth2User(

authorities,

oauth2User.getAttributes(),

userRequest.getClientRegistration().getProviderDetails().getUserInfoEndpoint().getUserNameAttributeName()

);

};

}

}

**Explanation of Changes:**  
 **Import OAuth2AccessToken:**

Java

import org.springframework.security.oauth2.core.OAuth2AccessToken;

 **Retrieve Token from OAuth2UserRequest:** Inside your oauth2UserService lambda, the OAuth2UserRequest object contains the OAuth2AccessToken. You can get it using:

OAuth2AccessToken accessToken = userRequest.getAccessToken();

The OAuth2UserRequest object holds all the information relevant to the user's authentication request, including the client registration, the access token, and the OAuth2AuthorizationExchange.

**Logging (for verification):** I've added org.slf4j.Logger to log the token. This is a good way to verify that you're successfully retrieving it.

private static final Logger logger = LoggerFactory.getLogger(SecurityConfig.class);

// ...

logger.info("Successfully received OAuth2 Access Token for {}: {}",

userRequest.getClientRegistration().getRegistrationId(),

accessToken.getTokenValue());

**What to do with the Access Token?**

* **Accessing Provider APIs:** This accessToken is primarily used to make calls to the respective OAuth2 provider's APIs on behalf of the user (e.g., fetch more details from Google People API, list GitHub repositories, etc.). You'd typically use a WebClient or similar HTTP client, adding an Authorization: Bearer <token\_value> header to your requests.
* **Storing the Token:**
  + **Temporarily (in memory):** For simple cases, you might just use it within the current request context.
  + **Persistently (database):** For long-lived sessions or if you need to perform background tasks on behalf of the user, you would store this token (and potentially the refresh token, if available) in a database. This usually requires a more advanced user management system where you link the OAuth2 user's details to your internal user ID.
  + **Custom OAuth2User:** For more complex scenarios, you might create a custom implementation of OAuth2User (or extend DefaultOAuth2User) to include the accessToken directly in the Authentication object, making it accessible throughout your application's security context. This is often done by extending DefaultOAuth2User and adding a constructor that accepts the token.

For your immediate need of just retrieving it, the provided code snippet is sufficient. You'll see the token value printed in your application logs upon successful social login.

[=------ Github user promotion to Admin ENDs ---=]  
My logic needs **to create an ACL level permission** of ROLE\_ADMIN , for **any person**, **logged in from Github**. I don't think the code which you gave will enable that. This code seems to give the Spring Framework, the necessary grants\role, but not to the ACL part of the code. If I am right then show me the code modifications which encompasses the ACL level User intitialization, with Admin role.  
Gemini **wasn’t able** to respond this query, so I changed it:  
In your explanation for Github specific Role\_Admin suggestion you wrote:

"Combining Criteria: You can combine multiple criteria using && (AND) or || (OR) operators. For example, to grant admin to a specific GitHub user and a specific Google email:"

Java code:

if (("github".equals(userRequest.getClientRegistration().getRegistrationId()) && "YOUR\_GITHUB\_USERNAME".equals(githubUsername)) ||

    ("google".equals(userRequest.getClientRegistration().getRegistrationId()) && email != null && "admin@example.com".equals(email))) {

    authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

It is in this context **that I want your reply** , **whether this code will enable ACL user creation or not?**

You are absolutely correct in your observation!

The if statement you quoted:

if (("github".equals(userRequest.getClientRegistration().getRegistrationId()) && "YOUR\_GITHUB\_USERNAME".equals(githubUsername)) ||

("google".equals(userRequest.getClientRegistration().getRegistrationId()) && email != null && "admin@example.com".equals(email))) {

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}  
This code snippet **alone will NOT enable ACL user creation or grant ACL-level permissions.**

**Here's why:**

1. **Spring Security ROLE\_ADMIN (GrantedAuthority):** This if block's sole purpose is to add new SimpleGrantedAuthority("ROLE\_ADMIN") to the authorities Set. This means the authenticated user will have the ROLE\_ADMIN role within Spring Security's context. This is what enables @PreAuthorize("hasRole('ROLE\_ADMIN')") checks.
2. **Spring Security ACL (Separate Layer):** Spring Security ACL is a distinct authorization framework that operates on domain objects. It has its own permission system and security identities (Sids). Just because a user has ROLE\_ADMIN as a GrantedAuthority in Spring Security doesn't automatically mean they have corresponding permissions within the ACL tables.

**To enable ACL user creation (i.e., for the user to be able to create documents via ACL permissions), you need the *subsequent logic* that interacts with the MutableAclService.**

This is why, in my previous comprehensive answer, I introduced the isAclAdmin flag and the ACL LEVEL PERMISSION ASSIGNMENT block.

**The complete logic for ACL integration is:**

1. SecurityConfig.java

package com.cybage.config;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity; // If you use @PreAuthorize

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configurers.AuthorizeHttpRequestsConfigurer;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.oauth2.client.userinfo.DefaultOAuth2UserService;

import org.springframework.security.oauth2.client.userinfo.OAuth2UserRequest;

import org.springframework.security.oauth2.client.userinfo.OAuth2UserService;

import org.springframework.security.oauth2.core.OAuth2AccessToken;

import org.springframework.security.oauth2.core.user.DefaultOAuth2User;

import org.springframework.security.oauth2.core.user.OAuth2User;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.acls.domain.BasePermission;

import org.springframework.security.acls.domain.ObjectIdentityImpl;

import org.springframework.security.acls.domain.PrincipalSid;

import org.springframework.security.acls.model.AccessControlEntry;

import org.springframework.security.acls.model.Acl;

import org.springframework.security.acls.model.MutableAcl;

import org.springframework.security.acls.model.MutableAclService;

import org.springframework.security.acls.model.NotFoundException;

import org.springframework.security.acls.model.ObjectIdentity;

import org.springframework.security.acls.model.Permission;

import org.springframework.security.acls.model.Sid;

import java.util.HashSet;

import java.util.LinkedHashMap;

import java.util.List;

import java.util.Map;

import java.util.Set;

import static org.springframework.security.config.Customizer.withDefaults; // For frameOptions

@Configuration

@EnableWebSecurity

@EnableMethodSecurity // Enable @PreAuthorize, @PostAuthorize, @Secured, etc.

public class SecurityConfig {

private static final Logger logger = LoggerFactory.getLogger(SecurityConfig.class);

// Autowire MutableAclService for ACL manipulation

@Autowired

private MutableAclService aclService;

// --- Centralized Authorization Rules ---

// Using LinkedHashMap to maintain order of request matchers, which is important.

private static final Map<String, String> AUTH\_RULES = new LinkedHashMap<>();

static {

// Publicly accessible paths (permitAll())

AUTH\_RULES.put("/index.html", "permitAll");

AUTH\_RULES.put("/static/\*\*", "permitAll");

AUTH\_RULES.put("/my\_login.html", "permitAll");

AUTH\_RULES.put("/login", "permitAll"); // For traditional form login POST

AUTH\_RULES.put("/oauth2/\*\*", "permitAll"); // OAuth2 authorization endpoints

AUTH\_RULES.put("/login/oauth2/code/\*\*", "permitAll"); // OAuth2 redirect/callback URIs

AUTH\_RULES.put("/x-documents/user", "permitAll"); // Example public API

AUTH\_RULES.put("/create-document.html", "permitAll"); // Assuming the form is public, but submission may be protected

AUTH\_RULES.put("/document-list.html", "permitAll");

AUTH\_RULES.put("/document-detail.html", "permitAll");

AUTH\_RULES.put("/permission-management.html", "permitAll");

AUTH\_RULES.put("/css/\*\*", "permitAll");

AUTH\_RULES.put("/js/\*\*", "permitAll");

AUTH\_RULES.put("/api/public/\*\*", "permitAll"); // Public API endpoints

AUTH\_RULES.put("/api/auth/\*\*", "permitAll"); // Authentication related APIs (e.g., login, register)

// H2 Console needs specific configuration AND permitAll() for its path

// Disable CSRF and frameOptions for H2 console

AUTH\_RULES.put("/h2-console/\*\*", "permitAll");

// Paths requiring authentication (authenticated())

AUTH\_RULES.put("/x-documents/cDoc", "authenticated"); // Example protected API for document creation

// Add more authenticated or role-specific paths here

// AUTH\_RULES.put("/admin-dashboard/\*\*", "hasRole('ROLE\_ADMIN')");

}

/\*\*

\* Configures the security filter chain.

\* This is the main configuration point for Spring Security.

\*/

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

logger.debug("Configuring Spring Security Filter Chain...");

// Disable CSRF for H2 console, as it's a development tool and conflicts with iframe usage

http.csrf(csrf -> csrf

.ignoringRequestMatchers("/h2-console/\*\*")); // Ignore CSRF for H2 console

// H2 Console needs frameOptions disabled to render in an iframe

http.headers(headers -> headers

.frameOptions(frameOptions -> frameOptions.sameOrigin())); // Allow embedding from same origin for H2 console

// Configure authorization rules

http.authorizeHttpRequests(auth -> {

logger.debug("Applying authorization rules from AUTH\_RULES map.");

AUTH\_RULES.forEach((pattern, access) -> {

if ("permitAll".equals(access)) {

auth.requestMatchers(pattern).permitAll();

logger.debug("Path '{}' set to permitAll().", pattern);

} else if ("authenticated".equals(access)) {

auth.requestMatchers(pattern).authenticated();

logger.debug("Path '{}' set to authenticated().", pattern);

} else if (access.startsWith("hasRole(")) { // Example for role-based access

auth.requestMatchers(pattern).hasRole(access.substring(access.indexOf("'") + 1, access.lastIndexOf("'")));

logger.debug("Path '{}' set to {}.", pattern, access);

}

// Extend with more custom authorization rules (e.g., hasAuthority, access)

});

// Any other request not matched by the above rules must be authenticated.

// THIS MUST BE THE LAST RULE IN THE AUTHORIZE\_HTTP\_REQUESTS BLOCK.

auth.anyRequest().authenticated();

logger.debug("Any other request requires authentication.");

});

// Configure traditional form login

http.formLogin(formLogin ->

formLogin

.loginPage("/my\_login.html") // Specify your custom login page

.defaultSuccessUrl("/create-document.html", true) // Redirect after successful login

.failureUrl("/my\_login.html?error") // Redirect on failed login

.permitAll() // Allow access to login-related URLs

);

// Configure OAuth2 Login (Google, GitHub, etc.)

http.oauth2Login(oauth2Login ->

oauth2Login

.loginPage("/my\_login.html") // Use the same custom login page

.defaultSuccessUrl("/create-document.html", true) // Redirect after successful OAuth2 login

.failureUrl("/my\_login.html?error") // Redirect on OAuth2 login failure

.userInfoEndpoint(userInfo ->

userInfo.userService(this.oauth2UserService()) // Inject our custom OAuth2UserService bean

)

);

// Configure logout

http.logout(logout ->

logout

.logoutUrl("/logout") // URL to trigger logout

.logoutSuccessUrl("/my\_login.html?logout") // Redirect after successful logout

.permitAll()

);

// Build and return the HttpSecurity object

return http.build();

}

/\*\*

\* Configures the custom OAuth2UserService to handle user details after successful authentication

\* from social providers (Google, GitHub). This is where roles and ACL permissions are assigned.

\*/

@Bean

public OAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {

// Default service to load user attributes from the OAuth2 provider

final DefaultOAuth2UserService delegate = new DefaultOAuth2UserService();

// Return a lambda that acts as our custom user service

return (userRequest) -> {

// Load the OAuth2User from the provider

OAuth2User oauth2User = delegate.loadUser(userRequest);

logger.debug("OAuth2 User loaded for provider: {}", userRequest.getClientRegistration().getRegistrationId());

logger.debug("User attributes: {}", oauth2User.getAttributes());

// Retrieve the Access Token

OAuth2AccessToken accessToken = userRequest.getAccessToken();

logger.info("Access Token for {}: {}",

userRequest.getClientRegistration().getRegistrationId(),

accessToken.getTokenValue());

// You can log other token details like expiration, scopes if needed

// Extract common attributes

String email = oauth2User.getAttribute("email");

String name = oauth2User.getAttribute("name"); // Often the 'sub' or unique ID from IdP

// Initialize authorities with a default role

Set<GrantedAuthority> authorities = new HashSet<>();

authorities.add(new SimpleGrantedAuthority("ROLE\_USER"));

logger.debug("Default role ROLE\_USER assigned.");

// Flag to determine if ACL admin permissions should be processed

boolean isAclAdmin = false;

// --- Determine Spring Security Roles based on Provider and User Attributes ---

if ("github".equals(userRequest.getClientRegistration().getRegistrationId())) {

String githubUsername = (String) oauth2User.getAttributes().get("login");

logger.debug("GitHub Username: {}", githubUsername);

// Example: Grant ROLE\_ADMIN to a specific GitHub username

if ("YOUR\_GITHUB\_USERNAME\_FOR\_ADMIN".equals(githubUsername)) { // <<< REPLACE THIS

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

logger.info("Granted Spring Security ROLE\_ADMIN to GitHub user: {}", githubUsername);

isAclAdmin = true; // Mark for ACL admin processing

}

} else if ("google".equals(userRequest.getClientRegistration().getRegistrationId())) {

logger.debug("Google User Email: {}", email);

// Example: Grant ROLE\_ADMIN to a specific Google email

if (email != null && "admin@example.com".equals(email)) { // <<< REPLACE THIS

authorities.add(new SimpleGrantedAuthority("ROLE\_ADMIN"));

logger.info("Granted Spring Security ROLE\_ADMIN to Google user: {}", email);

isAclAdmin = true; // Mark for ACL admin processing

}

}

// Add more provider-specific role assignments here if needed

// --- ACL LEVEL PERMISSION ASSIGNMENT ---

// This block will only execute if the user was determined to be an ACL admin

if (isAclAdmin) {

// The PrincipalSid uses the unique identifier from the IdP (oauth2User.getName() maps to 'sub').

// This is the stable and recommended identifier for ACLs for OAuth2 users.

Sid userSid = new PrincipalSid(oauth2User.getName());

logger.debug("ACL User SID: {}", userSid.getPrincipal());

// Define the Object Identity for the Document class.

// This means permissions are granted for operations on objects of this type.

// >>> IMPORTANT: Replace com.cybage.entity.Document.class with your actual Document entity class path. <<<

ObjectIdentity oid = new ObjectIdentityImpl(com.cybage.entity.Document.class);

logger.debug("ACL Object Identity: {}", oid.getIdentifier());

MutableAcl acl;

try {

// Try to retrieve an existing ACL for the Document class

acl = (MutableAcl) aclService.readAclById(oid);

logger.debug("Existing ACL found for Document class.");

} catch (NotFoundException e) {

// If no ACL exists for the Document class, create a new one

acl = aclService.createAcl(oid);

logger.info("New ACL created for Document class.");

}

// Define the permission to grant.

// BasePermission.CREATE allows the user to create new instances of Document.

// BasePermission.ADMINISTRATION would grant broader administrative rights on the Document class.

Permission permissionToGrant = BasePermission.CREATE;

logger.debug("Permission to check/grant: {}", permissionToGrant.getPattern());

// Check if the permission already exists for this SID to avoid duplicates

boolean permissionAlreadyExists = false;

List<AccessControlEntry> entries = acl.getEntries();

for (AccessControlEntry entry : entries) {

if (entry.getSid().equals(userSid) && entry.getPermission().equals(permissionToGrant)) {

permissionAlreadyExists = true;

break;

}

}

if (!permissionAlreadyExists) {

// Insert the Access Control Entry (ACE): grant permission to the SID

acl.insertAce(acl.getEntries().size(), permissionToGrant, userSid, true);

// Update the ACL to persist changes to the database

aclService.updateAcl(acl);

logger.info("ACL: Successfully granted {} permission on Document class to user {}.",

permissionToGrant.getPattern(), oauth2User.getName());

} else {

logger.info("ACL: User {} already has {} permission on Document class. No change needed.",

oauth2User.getName(), permissionToGrant.getPattern());

}

}

// Return a DefaultOAuth2User with the determined authorities and original attributes

// This object will be used to populate the SecurityContext

return new DefaultOAuth2User(

authorities,

oauth2User.getAttributes(),

userRequest.getClientRegistration().getProviderDetails().getUserInfoEndpoint().getUserNameAttributeName()

);

};

}

/\*\*

\* Configures the password encoder for traditional form login (e.g., for hashing passwords).

\*/

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

}

2. **AclInitializer.java (A Separate Class)**

This class is important if you want to set up base ACL permissions for roles (like ROLE\_ADMIN) on certain domain object types (like Document.class) when your application starts. This ensures that the ROLE\_ADMIN role itself has permissions, separate from individual users.

package com.cybage.config; // Adjust package as needed

import com.cybage.entity.Document; // <<< IMPORTANT: Adjust path to your Document entity class

import org.springframework.boot.CommandLineRunner;

import org.springframework.security.acls.domain.BasePermission;

import org.springframework.security.acls.domain.GrantedAuthoritySid; // For roles as SIDs

import org.springframework.security.acls.domain.ObjectIdentityImpl;

import org.springframework.security.acls.model.\*;

import org.springframework.stereotype.Component;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

/\*\*

\* CommandLineRunner to initialize base ACL permissions on application startup.

\* Ensures that roles (like ROLE\_ADMIN) have necessary permissions on domain object types.

\* This complements the user-specific ACLs set in OAuth2UserService.

\*/

@Component

public class **AclInitializer** implements CommandLineRunner {

private static final Logger logger = LoggerFactory.getLogger(AclInitializer.class);

private final MutableAclService mutableAclService;

public AclInitializer(MutableAclService mutableAclService) {

this.mutableAclService = mutableAclService;

}

@Override

public void run(String... args) throws Exception {

logger.info("Starting ACL initialization for core permissions...");

// 1. Define the Object Identity for the Document class

// This targets the permissions at the type level.

// >>> IMPORTANT: Replace com.cybage.entity.Document.class with your actual Document entity class path. <<<

ObjectIdentity documentClassOi = new ObjectIdentityImpl(Document.class);

// 2. Define the SID for the ROLE\_ADMIN.

// This is a GrantedAuthoritySid because it refers to a Spring Security role.

Sid adminRoleSid = new GrantedAuthoritySid("ROLE\_ADMIN");

try { // Try to retrieve the ACL for the Document class (type-level ACL)

Acl acl;

try {

acl = mutableAclService.readAclById(documentClassOi);

logger.info("ACL found for Document class (type level).");

} catch (NotFoundException e) {

// If ACL not found for the class, create a new one

acl = mutableAclService.createAcl(documentClassOi);

logger.info("Created new ACL for Document class (type level).");

}

// --- Grant CREATE permission to ROLE\_ADMIN on the Document class ---

// This means any user with ROLE\_ADMIN can create instances of Document.

// This is crucial for methods like @PreAuthorize("hasPermission(#document, 'create')") where #document is null (new object).

Permission createPermission = BasePermission.CREATE;

boolean createPermissionExists = false;

for (AccessControlEntry entry : acl.getEntries()) {

if (entry.getSid().equals(adminRoleSid) && entry.getPermission().equals(createPermission)) {

createPermissionExists = true;

logger.debug("ROLE\_ADMIN already has CREATE permission on Document class.");

break;

}

}

if (!createPermissionExists) {

((MutableAcl) acl).insertAce(acl.getEntries().size(), createPermission, adminRoleSid, true);

logger.info("Granted CREATE permission to ROLE\_ADMIN on Document class (type level).");

}

// --- Optionally, grant ADMINISTRATION permission to ROLE\_ADMIN on the Document class ---

// This allows ROLE\_ADMIN to manage ACLs for individual Document instances if needed.

// If you only want them to CREATE, you might not need this.

Permission administrationPermission = BasePermission.ADMINISTRATION;

boolean adminPermissionExists = false;

for (AccessControlEntry entry : acl.getEntries()) {

if (entry.getSid().equals(adminRoleSid) && entry.getPermission().equals(administrationPermission)) {

adminPermissionExists = true;

logger.debug("ROLE\_ADMIN already has ADMINISTRATION permission on Document class.");

break;

}

}

if (!adminPermissionExists) {

((MutableAcl) acl).insertAce(acl.getEntries().size(), administrationPermission, adminRoleSid, true);

logger.info("Granted ADMINISTRATION permission to ROLE\_ADMIN on Document class (type level).");

}

// Save any changes to the ACL

mutableAclService.updateAcl((MutableAcl) acl);

logger.info("ACL initialization for core permissions completed.");

} catch (Exception e) {

logger.error("Error during ACL initialization for Document class:", e);

// Consider more robust error handling or immediate application shutdown if critical ACLs fail to initialize

}

}

}

**Remember to:**

1. **Replace Placeholders:**
   * "YOUR\_GITHUB\_USERNAME\_FOR\_ADMIN" in SecurityConfig.java
   * "admin@example.com" in SecurityConfig.java
   * com.cybage.entity.Document.class in **both** SecurityConfig.java and AclInitializer.java with the *actual fully qualified path* to your Document entity class.
2. **Ensure ACL Configuration:** This code *assumes* you have Spring Security ACL already set up in your project (e.g., in a separate ACL configuration class), including the database schema (e.g., ACL\_SID, ACL\_CLASS, etc.) and the necessary MutableAclService bean being correctly exposed.
3. **Clean and Rebuild:** Always perform a clean and rebuild of your project after these significant code changes.
4. **Test Thoroughly:**
   * Log in with your designated GitHub admin account.
   * Verify you can create a document (assuming your @PreAuthorize is hasPermission(null, 'com.cybage.entity.Document', 'create') or similar).
   * Log in with a non-admin GitHub/Google account.
   * Verify they cannot create a document.
   * Test traditional form login for both admin and non-admin users if you still use it.

Sources