

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2024

M. Nardone, C. Scarioni, *Pro Spring Security*

https://doi-org.ezproxy.sfpl.org/10.1007/979-8-8688-0035-1_8

8. Open Authorization 2.0 (OAuth 2.0) and Spring Security

Massimo Nardone¹ and Carlo Scarioni²

(1) HELSINKI, Finland

(2) Surbiton, UK

Spring Security is a very extendable and customizable framework. This is primarily because the framework is built using object-oriented principles and design practices so that it is open for extension and closed for modification. In the previous chapter, you saw one of the major extension points in Spring Security—namely, the pluggability of different authentication providers. This chapter covers the popular Open Authorization 2.0 (OAuth 2.0) framework. It shows how to develop login security applications using Spring Boot, Spring Web, and OAuth 2.0 client (security) toward GitHub and Google providers.

An Introduction to OAuth 2.0

OAuth 2.0 is a widely used authorization framework that allows third-party applications to access users' resources without exposing their credentials (such as usernames and passwords). It provides a secure and standardized way for users to grant limited access to their data or services to other applications or services, often called *clients*.

It operates based on tokens, which are short-lived and revocable access credentials. These tokens authenticate and authorize access between the client application, the resource owner (typically a user), and the resource server (where the protected resources are stored).

The following is a high-level overview of how OAuth 2.0 works.

- **Client registration:** The application registers with the OAuth 2.0 authorization server. During this registration, it receives a client identifier and a client secret (a confidential key).
- **User authentication:** When the user wants to grant the client access to their resources, they are redirected to the authorization server for authentication. This step ensures the user's identity and consent.
- **Authorization grant:** After the user is authenticated, they are prompted to grant the client specific permissions to access their resources. This is often done through an authorization code or other types of grants.
- **Token request:** After obtaining the authorization grant, the client requests the authorization server to exchange the grant for an access token.

- **Access token issuance:** If the authorization server validates the request and grants access, it issues an access token to the client.
- **Accessing protected resources:** The client can now use the access token to access the user's protected resources on the resource server. The access token serves as proof of authorization.
- **Token expiration and refresh:** Access tokens typically have a limited lifespan. When they expire, the client can use a refresh token (if provided) to obtain a new access token without requiring the user to re-authenticate.

OAuth 2.0 is widely used for securing APIs, allowing users to grant selective access to their data on platforms like social media, and enabling single sign-on (SSO) across different services.

Please note that OAuth 2.0 is not a protocol for authentication; it's a framework for authorization. For user authentication, OpenID Connect, an identity layer built on top of OAuth 2.0, is often used in conjunction with OAuth 2.0 to provide both authentication and authorization capabilities.

OAuth 2.0 Security

OAuth 2.0 provides a framework for authorization, but it is essential to implement it securely to protect user data and resources. The following describes some key security considerations and best practices when using OAuth 2.0.

- Use **HTTPS**: Always use HTTPS to protect the communication between the client, authorization server, and resource server. This ensures the confidentiality and integrity of data transmitted during the OAuth flow.
- **Client authentication**: Implement proper client authentication. Depending on the OAuth 2.0 flow, clients should authenticate themselves using client credentials or other methods like client certificates.
- **Authorization Code Flow**: Web applications and confidential clients should use this flow, which involves an authorization code exchanged for an access token, reducing the risk of exposing tokens in the browser.
- **Token storage**: Safely store and manage access and refresh tokens on the client side. Avoid storing tokens in insecure locations such as browser cookies, and use secure storage mechanisms.
- **Token validation**: When receiving access tokens from the authorization server, validate them properly. Check the token's signature and expiration date to ensure it's valid.
- **Scope permissions**: Ensure that clients only request the minimum necessary scope of permissions (access rights) from the user. This principle is known as the principle of least privilege.
- **User consent**: Always obtain clear and informed consent from the user before granting access to their data. Users should understand what data the client application can access and for what purpose.
- **Refresh token security**: Protect refresh tokens as they have a longer lifespan. Use secure storage and transmission mechanisms for refresh tokens. Only grant refresh tokens to confidential clients when necessary.

- **Token revocation:** Implement token revocation mechanisms. Allow users to revoke access to their data, invalidate access tokens, and re-fresh tokens when no longer needed.
- **Rate limiting and throttling:** Implement rate limiting and throttling to protect against brute-force and denial-of-service attacks on OAuth endpoints.
- **Cross-site request forgery (CSRF) protection:** Use anti-CSRF tokens or other techniques to protect against CSRF attacks that trick users into making unintended requests.
- **Authorization server security:** Secure the authorization server against common security threats, such as injection attacks, and keep its software and libraries current.
- **Logging and monitoring:** Implement comprehensive logging and monitoring to detect and respond to suspicious activities and security breaches.
- **Token rotation:** Periodically rotate client secrets and access tokens to mitigate the risk of exposure due to unauthorized access or leaks.
- **Security assessments:** Conduct security assessments, code reviews, and penetration testing to identify and address vulnerabilities in your OAuth 2.0 implementation.

The security of an OAuth 2.0 implementation depends on a combination of factors like the OAuth flow being used, the specific use case, and the client and authorization server configurations. Therefore, it's crucial to follow best practices, stay informed about security updates, and adapt your OAuth 2.0 implementation to the unique requirements of your application.

Integrating OAuth 2.0 with Spring Security

OAuth 2.0 can be integrated with Spring Security to secure your Java-based web applications, APIs, and microservices. Spring Security provides robust support for implementing OAuth 2.0 authentication and authorization in a Spring-based application.

Here's a basic overview of implementing OAuth 2.0 using Spring Security.

- **Add dependencies:** Ensure that you have the necessary dependencies in your project. Spring Security OAuth 2.0 module is essential for OAuth 2.0 support. You can include it in your pom.xml or build.gradle file.
- **Configuration:** Configure Spring Security to handle OAuth 2.0 by creating a configuration class that extends `AuthorizationServerConfigurerAdapter`. This class should provide information about your OAuth 2.0 authorization server, client credentials, and endpoints. Listing 8-1 shows a configuration Java example.

```
import org.springframework.context.annotation.*;
```

```
import org.springframework.security.oauth2.config.annotation.web.configuration.*;

@Configuration

@EnableAuthorizationServer

public class OAuth2AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

    @Override

    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

        clients.inMemory()

            .withClient("client-id")

            .secret("client-secret")

            .authorizedGrantTypes("authorization_code", "password", "refresh_token")

            .scopes("read", "write")

            .redirectUri("http://localhost:8080/callback");

    }

}
```

Listing 8-1

Configure OAuth 2.0

- **User authentication:** Configure how your application handles user authentication. You can use the default Spring Security mechanisms or integrate with external identity providers.
- **Resource server configuration (optional):** If you're building an OAuth 2.0 resource server (e.g., an API), you must configure Spring

Security to validate access tokens. You can do this by creating a class that extends `ResourceServerConfigurerAdapter`. Listing 8-2 shows a server configuration Java example.

```
import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

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

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

import org.springframework.security.web.SecurityFilterChain;


@Configuration

@EnableWebSecurity


public class SecurityConfiguration {


    @Bean

    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        return http

            .authorizeHttpRequests(auth -> {

                auth.requestMatchers("/api/**").authenticated ();

                auth.anyRequest().authenticated();

            })

            .build();
    }
}
```

```
        })

        .oauth2Login(withDefaults())

        .build();

    }

}
```

Listing 8-2

Configure OAuth 2.0 Resource Server

- **Secure endpoints:** Use Spring Security annotations like `@Secured`, `@PreAuthorize`, or `@PostAuthorize` to secure specific methods or endpoints in your application.
- **User consent and authentication flow:** Implement a user interface for the OAuth 2.0 authentication flow. This includes handling user consent and redirecting users to the OAuth 2.0 authorization endpoint.
- **Token storage and management:** Implement token storage and management, including access tokens, refresh tokens, and their lifecycles. Spring Security OAuth 2.0 provides mechanisms to handle this.
- **Testing and validation:** Thoroughly test your OAuth 2.0 implementation to ensure that the authentication and authorization flows work as expected. You can use tools like Postman or dedicated OAuth 2.0 clients for testing.
- **Logging and monitoring:** Implement logging and monitoring to track security-related events and potential issues.
- **Documentation and error handling:** Provide clear documentation for developers using your OAuth 2.0-protected resources and implement proper error handling to respond to various OAuth 2.0-related errors gracefully.

Remember that OAuth 2.0 implementation can vary depending on your requirements and use cases. Spring Security provides flexibility to adapt OAuth 2.0 to your application's needs while following security best practices.

OAuth 2.0 Login

OAuth 2.0 Login is a secure and standardized way for users to grant permission to third-party applications to access their protected resources or perform actions on their behalf without sharing their login credentials. It is commonly used for single sign-on (SSO) and enabling users to log in to different websites or applications using their existing credentials from a trusted identity provider (IdP).

Here's what is included in OAuth 2.0 Login.

- **User-centric authentication:** OAuth 2.0 Login is user-centric, focusing on the user's consent and authorization. Users grant permission to applications to access specific resources without revealing their usernames and passwords.
- **Role of the user:** When users try to log in to a third-party application, they are redirected to an IdP authentication page. The user authenticates themselves with the IdP, which verifies their identity.
- **Authorization Code Flow:** One of the most common OAuth 2.0 Login flows.
 - The user is redirected to the IdP's login page.
 - After successful authentication, the IdP asks the user for consent to share their data with the requesting application.
 - If the user consents, the IdP generates an authorization code.
 - The application exchanges this authorization code for an access token, which it can use to access the user's resources.
- **Single sign-on (SSO):** OAuth 2.0 Login is often used for SSO scenarios, where users can log in once and then access multiple applications without re-entering their credentials. This reduces the need for users to remember multiple usernames and passwords.
- **Third-party applications:** OAuth 2.0 Login enables third-party applications to request access to a user's resources without having direct access to their credentials. This enhances security by limiting the exposure of sensitive information.
- **Scoped access:** OAuth 2.0 Login allows users to grant specific permissions (scopes) to applications, ensuring that applications only access the data or perform the actions the user allows.
- **Token-based authentication:** OAuth 2.0 Login relies on access tokens, which are short-lived revocable credentials used to authenticate API requests on behalf of the user. These tokens replace the need for the user's username and password in each API call.
- **Security and authorization:** OAuth 2.0 Login provides a robust framework for securing user data and resources. Users have control over which applications have access to their data, and they can revoke access at any time.
- **Widely adopted:** OAuth 2.0 is widely adopted across various platforms, services, and industries. It is used by many well-known identity providers, including Google, Facebook, and Microsoft, making it a common choice for integrating with their services.

OAuth 2.0 Login is implemented using the authorization code grant, as specified in the OAuth 2.0 authorization framework at

<https://datatracker.ietf.org/doc/html/rfc6749#section-4.1>.

Similarly, OpenID Connect Core 1.0 is at

https://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth.

The OAuth 2.0 Login feature lets users log in to the application using their existing account at an OAuth 2.0 provider (such as GitHub) or OpenID Connect 1.0 provider (such as Google). OAuth 2.0 Login can also authenticate toward Facebook, Twitter, and so on.

This example seeks to configure the Spring Authorization Server with a social login provider such as Google and GitHub and authenticate the user with OAuth 2.0 Login, replacing the common form login.

Let's build our authentication and login application using Spring Boot 3.x, Spring Security 6, Spring Web, and OAuth 2.0 Client.

The first step is to create the Spring Boot Maven project using the Spring Initializr, the quickest way to generate Spring Boot projects. You just need to choose the language, build system, and JVM version for your project, and it is automatically generated with all the dependencies needed.

Navigate to <https://start.spring.io/> and use the Spring Initializr web-based Spring project generator to create the Spring Boot Maven project named OAuth2SecurityLogin, as shown in Figure 8-1.

The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', version '3.1.3' is selected. The 'Project Metadata' section has the following values: Group: 'com.apress', Artifact: 'OAuth2SecurityLogin', Name: 'OAuth2SecurityLogin', Description: 'Demo project for Spring Boot Security and OAuth2', and Package name: 'com.apress.OAuth2SecurityLogin'. The 'Packaging' is set to 'jar'. The 'Dependencies' section on the right has checkboxes for 'OAuth2 Client' and 'Spring Web', both of which are checked. A button 'ADD DEPENDENCIES... CTRL + B' is visible.

Figure 8-1 Generate an OAuth 2.0 project using the Initializr web-based Spring project generator

Select a Java 20 Maven project using Spring Boot version 3.1.3, and add the following dependencies: Spring Web and OAuth 2.0 Client.

Add the Thymeleaf Java library as a dependency to the pom.xml file. It is a template engine used to parse and transform the data produced by the application to template files. It acts just like HTML but provides more attributes used to render data.

Fill in all the required information and then click to generate the project. A project .zip file is automatically generated. Download and unzip the file on your machine.

Figure 8-2 shows the project in IntelliJ IDEA 2023.1.2.

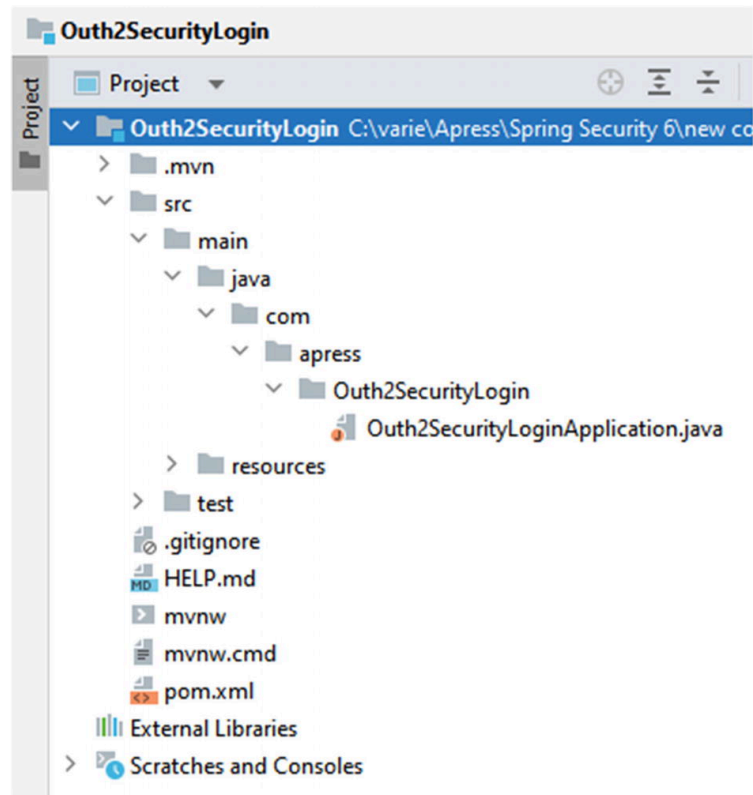


Figure 8-2 Maven project structure

The most important dependencies, which are automatically updated in the `pom.xml` file, are shown in Listing 8-3.

```
<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

</dependency>
```

Listing 8-3
Needed Dependencies

The entire generated `pom.xml` file with the added dependencies is shown in Listing 8-4.

```
<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd"
    >

    <modelVersion>4.0.0</modelVersion>

    <parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

        <version>3.1.3</version>

        <relativePath/> <!-- Lookup parent from repository -->

    </parent>

    <groupId>com.apress</groupId>

    <artifactId>OAuth2SecurityLogin</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>OAuth2SecurityLogin</name>

    <description>Demo project for Spring Boot Security and OAuth 2.0</description>
```

```
<properties>

    <java.version>20</java.version>

</properties>

<dependencies>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-oauth2-client</artifactId>

    </dependency>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-web</artifactId>

    </dependency>

    <dependency>

        <groupId>org.thymeleaf.extras</groupId>

        <artifactId>thymeleaf-extras-springsecurity6</artifactId>

        <version>3.1.1.RELEASE</version>

    </dependency>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-test</artifactId>
```

```
        <scope>test</scope>

    </dependency>

</dependencies>

<build>

    <plugins>

        <plugin>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-maven-plugin</artifactId>

        </plugin>

    </plugins>

</build>

</project>
```

Listing 8-4
pom.xml File and Dependencies

Let's build our first Java controller class, `UserController`, in Spring MVC to specify its methods with various annotations of the requests, such as the URLs of the endpoint, the HTTP request method, the path variables, etc.

Listing [8-5](#) shows the `UserController` class.

```
import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

@Controller

public class UserController {

    @GetMapping("/")

    public String homePage() { return "welcome";

    }

    @GetMapping("/welcome")

    public String welcomePage() {

        return "welcome";

    }

    @GetMapping ("/authenticated")

    public String AuthenticatedPage() {

        return "authenticated";
```

```
    }

    @GetMapping ("/logout")

    public String logoutPage() {

        return "redirect:/welcome";

    }

}
```

Listing 8-5

UserController Java Class

The controller Java class redirects to the welcome.html page for the “/” and “/Welcome” and authenticated.html for “/authenticated” URLs. Logout mapping is used when logging out the user from GitHub or Google authentication.

Let’s create two simple HTML pages.

- welcome.html (see Listing [8-6](#)), a simple welcome page permitted to all users to provide the link to the authenticated html page
- authenticated.html (see Listing [8-7](#)), a simple HTML page showing the authenticated (GitHub or Google) username if authenticated

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">

<html lang="en">
```

```
<head>

    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

    <title>Spring Security 6 and OAuth 2.0 Login authentication example!</title>

</head>

<body>


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

        Invalid username and password.

    </div>

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

        You have been logged out.

    </div>


    <h2>Welcome to Spring Security 6 and OAuth 2.0 Login authentication example!</h2>


    <p>Click <a th:href="@{/authenticated}">here</a> to get authenticated to GitHub or Google with OAuth 2.

</body>

</html>
```

Listing 8-6

welcome.html Page

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"

      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity6">

  <head>

    <title>Spring Security 6 and OAuth 2.0 Login authentication example!</title>

  </head>

  <body>

    <h2>Welcome to Spring Security 6 and OAuth 2.0 Login authentication example!</h2>

    <h2 th:inline="text">You are an authenticated user: <span th:remove="tag" sec:authentication="name">thy

    <form th:action="@{/logout}" method="post">

      <input type="submit" value="Logout"/>

    </form>

  </body>
```



```
</html>
```

Listing 8-7

welcome.html Page

You now create the most important Java class of the example, `SpringSecurityConfiguration`, shown in Listing [8-8](#).

```
package com.apress.OAuth2SecurityLogin.configuration;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

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

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

import org.springframework.security.web.SecurityFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration

@EnableWebSecurity

public class SecurityConfiguration {
```

```
@Bean

SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

    return http

        .authorizeHttpRequests(auth -> {

            auth.requestMatchers("/", "welcome").permitAll();

            auth.anyRequest().authenticated();

        })

        .oauth2Login(withDefaults())

        .formLogin(withDefaults())

        .build();

}
```

Listing 8-8

SpringSecurityConfiguration Java Class

The Spring Security Java class does the following.

- Allows all users to access routes “/” or “Welcome”
- Makes any other request (e.g., /authenticated) authenticated via GitHub or Google

- Uses OAuth 2.0 Login method to log the listed providers in the application.properties file (in our case, GitHub and Google)
- Uses the Spring Security 6 FormLogin

Next, let's start configuring GitHub and Google to be accessed via OAuth 2.0 Login.

The first step is configuring the application properties file, as shown in Listing [8-9](#).

```
# GitHub Login

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

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


# Google Login

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

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


# Configure Spring Security Logging

logging.level.org.springframework.security=TRACE
```

Listing 8-9
application.properties Configuration

Adding the lines `..registration.github` and `..registration.google` tells Spring Security that you want to access those social providers via OAuth 2.0.0.

Let's now look at generating the OAuth 2.0 IDs and secret keys for Google and GitHub.

To use Google's OAuth 2.0 authentication method for login, you must set up a project in the Google API console to obtain OAuth 2.0 credentials (ID and secret) to be added to the application.properties.

Let's follow these steps to generate the OAuth 2.0 ID and secret key for Google.

1. Create a Google OAuth consent project and then link the consent to it. Let's visit the Google Cloud APIs & Services console to create the project and the consent (Figures [8-3](#) and [8-4](#)) at <https://console.cloud.google.com/projectselector2/apis/credentials/consent>

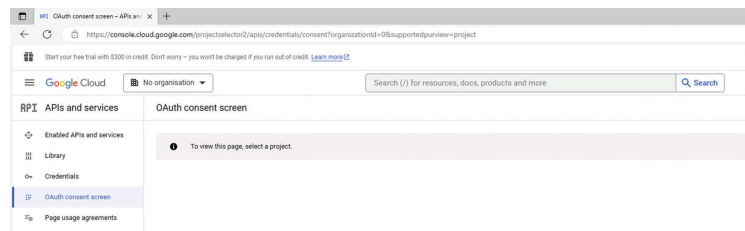


Figure 8-3 The Google Cloud console

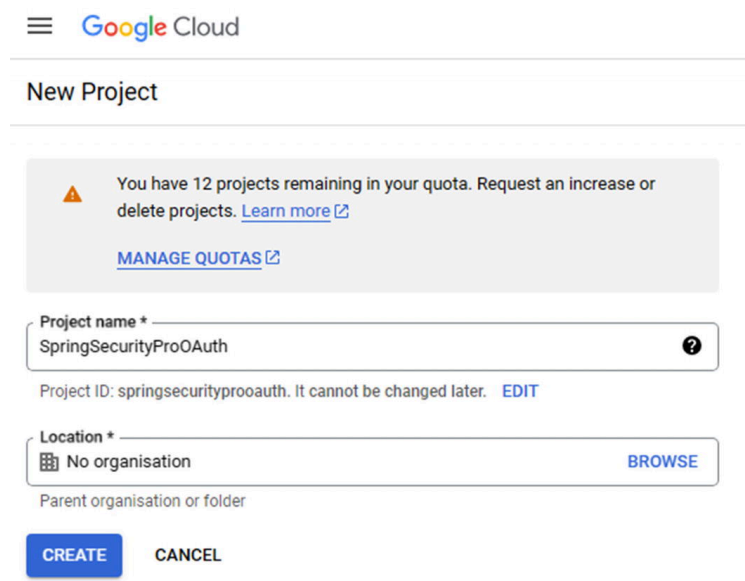


Figure 8-4 The Google Cloud new project page

2. Create a new consent associated with the project, as shown in Figure [8-5](#).

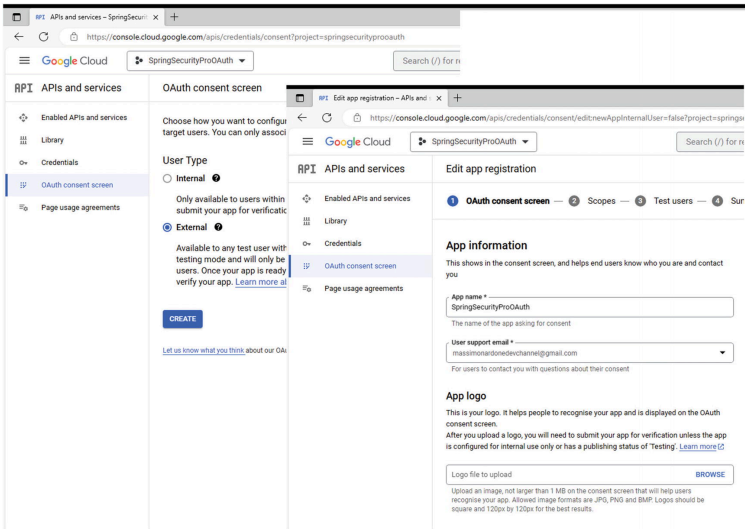


Figure 8-5 The Google Cloud OAuth consent

- 3.Next, go to the Credentials section and select “Create OAuth client ID”.
Select “Web application” as the application type and enter an application name.
Add the following as an authorized redirect URI, as shown in Figure 8-6.
`http://localhost:8080/login/oauth2/code/google`
- 4.Click the Create button to obtain your client_id and client_secret for the application.properties file, as shown in Figure 8-6.

APIs and services

Enabled APIs and services

Library

Credentials

OAuth consent screen

Page usage agreements

Create OAuth client ID

A client ID is used to identify a single app to Google's OAuth servers. If your app runs on multiple platforms, each will need its own client ID. See [Setting up OAuth 2.0](#) for more information. [Learn more](#) about OAuth client types.

Application type *
Web application

Name *
OAuth Security Web Client

The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

The domains of the URIs you add below will be automatically added to your [OAuth consent screen](#) as [authorised domains](#).

Authorised JavaScript origins

For use with requests from a browser

+ ADD URI

Authorised redirect URIs

For use with requests from a web server

URIs 1 *
<http://localhost:8080/login/oauth2/code/google>

+ ADD URI

Note: It may take five minutes to a few hours for settings to take effect

CREATE CANCEL

Figure 8-6 The Google Cloud Credentials page

Figure 8-7 shows how `client_id` and `client_secret` are used in the `application.properties` file.

OAuth client created

The client ID and secret can always be accessed from Credentials in APIs & Services

OAuth access is restricted to the [test users](#) listed on your [OAuth consent screen](#)

Client ID	740114053442-ekgdruqacm6cvk3gf715oiu45on0fqns.apps.googleusercontent.com
Client secret	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
Creation date	13 September 2023 at 15:50:25 GMT+3
Status	Enabled

DOWNLOAD JSON

OK

Figure 8-7 The Google `client_id` and `client_secret`

Copy the generated client ID and secret to our example application.properties.

```
# Google Login
```

```
spring.security.oauth2.client.registration.google.client-id= 740114053442-ekgdruqacm6cvk3gf715oiu45on0f
```

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

The application is ready to be tested.

Run the application and visit <http://localhost:8080/welcome>. You see the `welcome.html` page shown in Figure 8-8.

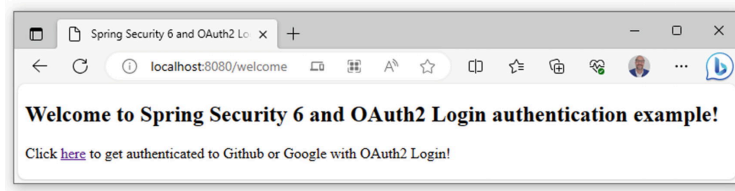


Figure 8-8 The welcome.html page

Click the “here” link to access the `authenticated.html` page, which automatically redirects to the Spring login web page. This provides the list (added in `application.properties`) of the social providers we are trying to access via the OAuth 2.0 Login authentication method, as shown in Figure 8-9.

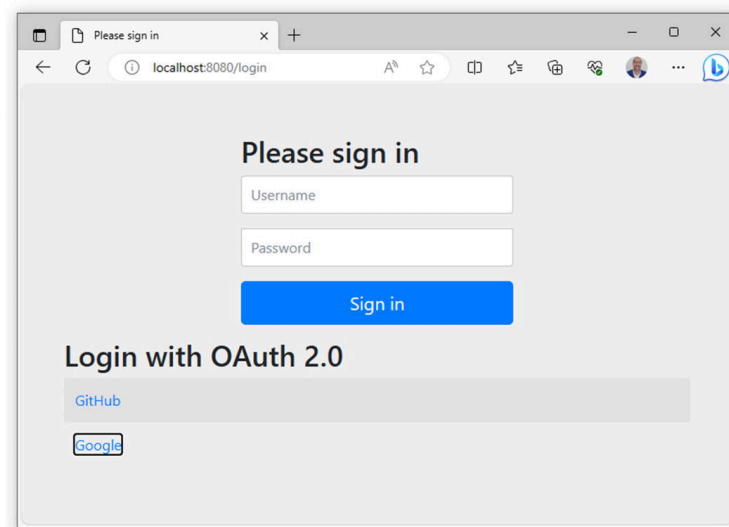


Figure 8-9 The login page

Click the Google link to be redirected to Google for authentication.

Next, authenticate (see Figure 8-10) with your Google account credentials. You see the Consent screen, which asks you to allow or deny access to the OAuth Client you created earlier. Click the Allow button to authorize the OAuth Client to access your email address and basic profile information.

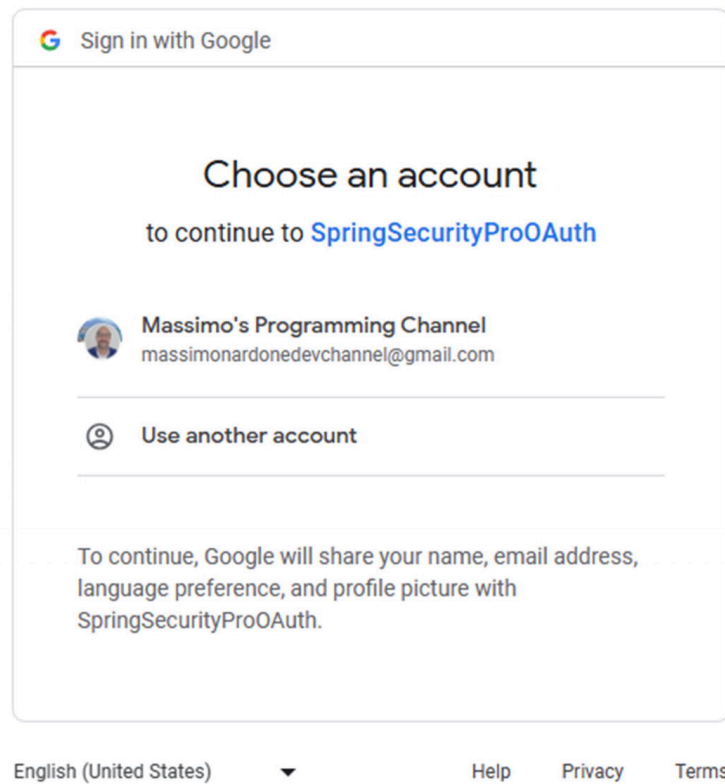


Figure 8-10 The Google selecting account page

Finally, the OAuth Client retrieves your email address and all the basic profile information from the UserInfo endpoint you configured in Google and establishes an authenticated session.

If the Google client credential configured in the Spring application matches the Google OAuth configured ID and secret, then the user is authenticated, showing the unique username, as shown in Figure 8-11; otherwise, a message error is displayed.

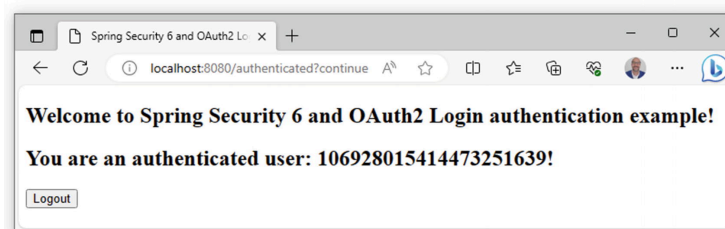


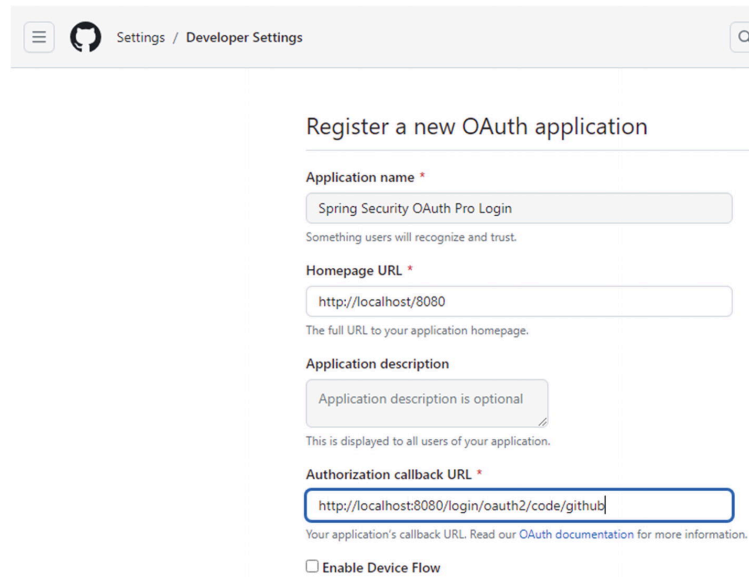
Figure 8-11 Google user is authenticated

Let's configure GitHub as an OAuth 2.0 Login provider.

1. Create a new OAuth app by going to your GitHub account settings and navigating to the Developer settings at

<https://github.com/settings/profile>.

2. Go to the OAuth Apps section and click New OAuth App. Complete the required fields, as shown in Figure 8-12. The authorization callback URL is `http://localhost:8080/login/oauth2/code/github`.

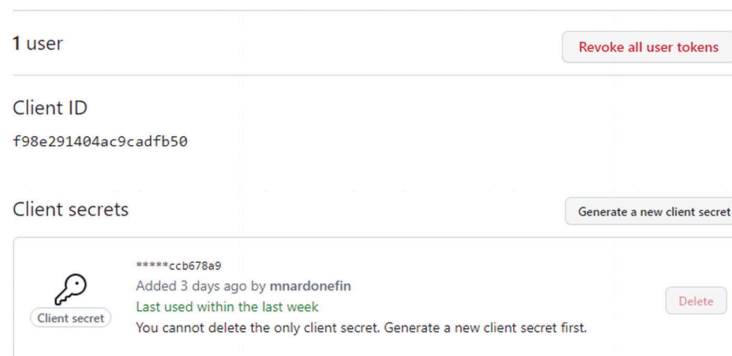


The screenshot shows the 'Register a new OAuth application' form in the GitHub Settings / Developer Settings section. The form includes the following fields and options:

- Application name ***: A text input field containing 'Spring Security OAuth Pro Login'. Below it, a note says 'Something users will recognize and trust.'
- Homepage URL ***: A text input field containing 'http://localhost:8080'. Below it, a note says 'The full URL to your application homepage.'
- Application description**: A text input field containing 'Application description is optional'. Below it, a note says 'This is displayed to all users of your application.'
- Authorization callback URL ***: A text input field containing 'http://localhost:8080/login/oauth2/code/github'. Below it, a note says 'Your application's callback URL. Read our OAuth documentation for more information.'
- Enable Device Flow**: An unchecked checkbox.

Figure 8-12 Create a new GitHub OAuth app

3. Click register the application to receive a client_id and client_secret to add to the application.properties file, as shown in Figure 8-13.



The screenshot shows the '1 user' section of the GitHub OAuth app details page. It includes the following information:

- Client ID**: f98e291404ac9cadfb50
- Client secrets**: A section with a 'Generate a new client secret' button. Below it, a card shows a client secret: *****cc678a9. The card also includes the text 'Added 3 days ago by mnardonefin', 'Last used within the last week', and a 'Delete' button. A message at the bottom states: 'You cannot delete the only client secret. Generate a new client secret first.'

Figure 8-13 New GitHub OAuth app generates ID and secret key

Copy the ID and secret key into the application.properties file in our application.

```
# GitHub Login
```

```
spring.security.oauth2.client.registration.github.client-id=f98e291404ac9cadfb50
```

```
spring.security.oauth2.client.registration.github.client-secret=e291201b5f8f3e368bc7380ecf0e3534ccb678a
```

Rerun the application and select GitHub as the login method, as shown in Figure 8-14.

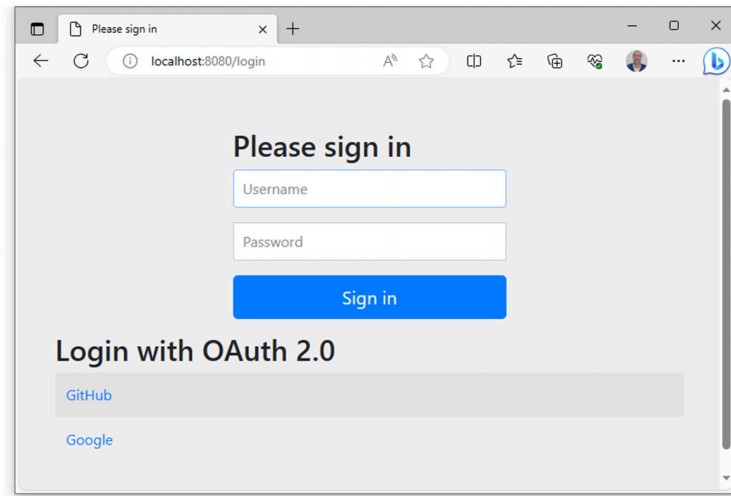


Figure 8-14 The GitHub account selecting the web page

If the user has a registered ID and secret key match, you can access the authenticated page, which shows the unique username (see Figure 8-15).

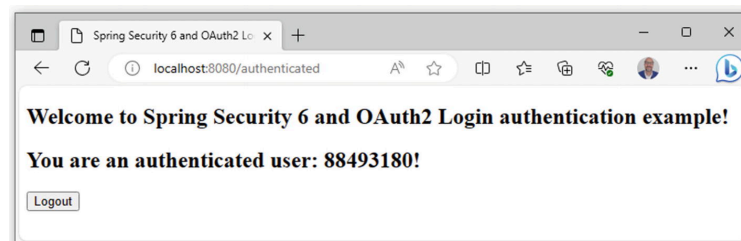


Figure 8-15 GitHub user is authenticated

This completes the demonstration on configuring OAuth 2.0 Login for Google and GitHub. Please note that this example is only configured in the application.properties file with some values like ID and secret key. Many other Spring Boot property mappings values can be added as OAuth Client properties to the ClientRegistration properties, including the following.

- `spring.security.oauth2.client.registration.[registrationId]`
- `spring.security.oauth2.client.registration.[registrationId].client-id`
- `spring.security.oauth2.client.registration.[registrationId].client-secret`
- `spring.security.oauth2.client.registration.[registrationId].client-authentication-method`
- `spring.security.oauth2.client.registration.[registrationId].authorization-grant-type`

- `spring.security.oauth2.client.registration.
[registrationId].redirect-uri`
- `spring.security.oauth2.client.registration.
[registrationId].scope`
- `spring.security.oauth2.client.registration.
[registrationId].client-name`
- `spring.security.oauth2.client.provider.
[providerId].authorization-uri`
- `spring.security.oauth2.client.provider.
[providerId].token-uri`
- `spring.security.oauth2.client.provider.[providerId].jwk-
set-uri`
- `spring.security.oauth2.client.provider.
[providerId].issuer-uri`
- `spring.security.oauth2.client.provider.
[providerId].user-info-uri`
- `spring.security.oauth2.client.provider.
[providerId].user-info-authentication-method`
- `spring.security.oauth2.client.provider.
[providerId].user-name-attribute`

Summary

This chapter demonstrated how Spring Security can be a very extendable and customizable framework as it is built using object-oriented principles and design practices so that it is open for extension and closed for modification. You learned how to use the Open Authorization 2.0 (OAuth 2.0) framework and how to develop login security applications using Spring Boot, Spring Web, and OAuth 2.0 client (security) to authenticate toward GitHub and Google providers.