# 31. OAuth 2.0 Login — Advanced Configuration

`HttpSecurity.oauth2Login()` provides a number of configuration options for customizing OAuth 2.0 Login. The main configuration options are grouped into their protocol endpoint counterparts.

For example, `oauth2Login().authorizationEndpoint()` allows configuring the *Authorization Endpoint*, whereas `oauth2Login().tokenEndpoint()` allows configuring the *Token Endpoint*.

The following code shows an example:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.oauth2Login()
				.authorizationEndpoint()
					...
				.redirectionEndpoint()
					...
				.tokenEndpoint()
					...
				.userInfoEndpoint()
					...
	}
}
```

The main goal of the `oauth2Login()` DSL was to closely align with the naming, as defined in the specifications.

The OAuth 2.0 Authorization Framework defines the Protocol Endpoints as follows:

The authorization process utilizes two authorization server endpoints (HTTP resources):

- Authorization Endpoint: Used by the client to obtain authorization from the resource owner via user-agent redirection.
- Token Endpoint: Used by the client to exchange an authorization grant for an access token, typically with client authentication.

As well as one client endpoint:

- Redirection Endpoint: Used by the authorization server to return responses containing authorization credentials to the client via the resource owner user-agent.

The OpenID Connect Core 1.0 specification defines the UserInfo Endpoint as follows:

The UserInfo Endpoint is an OAuth 2.0 Protected Resource that returns claims about the authenticated end-user. To obtain the requested claims about the end-user, the client makes a request to the UserInfo Endpoint by using an access token obtained through OpenID Connect Authentication. These claims are normally represented by a JSON object that contains a collection of name-value pairs for the claims.

The following code shows the complete configuration options available for the `oauth2Login()` DSL:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.oauth2Login()
				.clientRegistrationRepository(this.clientRegistrationR
				.authorizedClientService(this.authorizedClientService(
				.loginPage("/login")
				.authorizationEndpoint()
					.baseUri(this.authorizationRequestBaseUri())
					.authorizationRequestRepository(this.authoriza
					.and()
				.redirectionEndpoint()
					.baseUri(this.authorizationResponseBaseUri())
					.and()
				.tokenEndpoint()
					.accessTokenResponseClient(this.accessTokenRes
					.and()
				.userInfoEndpoint()
					.userAuthoritiesMapper(this.userAuthoritiesMap
					.userService(this.oauth2UserService())
					.oidcUserService(this.oidcUserService())
					.customUserType(GitHubOAuth2User.class, "githu
	}
}
```

The sections to follow go into more detail on each of the configuration options available:



## 31.1 OAuth 2.0 Login Page

By default, the OAuth 2.0 Login Page is auto-generated by the `DefaultLoginPageGeneratingFilter`. The default login page shows each configured OAuth Client with its `ClientRegistration.clientName`

as a link, which is capable of initiating the Authorization Request (or OAuth 2.0 Login).

The link's destination for each OAuth Client defaults to the following:

`OAuth2AuthorizationRequestRedirectFilter.DEFAULT_AUTHORIZATION_REQUEST_BASE_URI` + "/{registrationId}"

The following line shows an example:

```
<a href="/oauth2/authorization/google">Google</a>
```

To override the default login page, configure `oauth2Login().loginPage()` and (optionally) `oauth2Login().authorizationEndpoint().baseUri()`.

The following listing shows an example:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .oauth2Login()
                                .loginPage("/login/oauth2")
                                ...
                                .authorizationEndpoint()
                                        .baseUri("/login/oauth2/authorization")
                                        ....
        }
}
```

**Important**

You need to provide a `@Controller` with a `@RequestMapping("/login/oauth2")` that is capable of rendering the custom login page.

As noted earlier, configuring `oauth2Login().authorizationEndpoint().baseUri()` is optional. However, if you choose to customize it, ensure the link to each OAuth Client matches the `authorizationEndpoint().baseUri()`.
The following line shows an example:

```
<a href="/login/oauth2/authorization/google">Google</a>
```

## 31.2 Authorization Endpoint

### 31.2.1 `AuthorizationRequestRepository`

`AuthorizationRequestRepository` is responsible for the persistence of the `OAuth2AuthorizationRequest` from the time the Authorization Request is initiated to the time the Authorization Response is received (the callback).

The `OAuth2AuthorizationRequest` is used to correlate and validate the Authorization Response.

The default implementation of `AuthorizationRequestRepository` is `HttpSessionOAuth2AuthorizationRequestRepository`, which stores the `OAuth2AuthorizationRequest` in the `HttpSession`.

If you would like to provide a custom implementation of `AuthorizationRequestRepository` that stores the attributes of `OAuth2AuthorizationRequest` in a `Cookie`, configure it as shown in the following example:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .oauth2Login()
                                .authorizationEndpoint()
                                        .authorizationRequestRepository(this.cookieAut
                                        ...
        }

        private AuthorizationRequestRepository<OAuth2AuthorizationRequest> cookieAutho
                return new HttpCookieOAuth2AuthorizationRequestRepository();
        }
}
```

## 31.3 Redirection Endpoint

The Redirection Endpoint is used by the Authorization Server for returning the Authorization Response (which contains the authorization credentials) to the client via the Resource Owner user-agent.

OAuth 2.0 Login leverages the Authorization Code Grant. Therefore, the authorization credential is the authorization code.

The default Authorization Response `baseUri` (redirection endpoint) is `/login/oauth2/code/*`, which is defined in `OAuth2LoginAuthenticationFilter.DEFAULT_FILTER_PROCESSES_URI`.

If you would like to customize the Authorization Response `baseUri`, configure it as shown in the following example:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .oauth2Login()
                                .redirectionEndpoint()
                                        .baseUri("/login/oauth2/callback/*")
                                        ....
        }
}
```

**Important**

You also need to ensure the `ClientRegistration.redirectUriTemplate` matches the custom Authorization Response `baseUri`.
The following listing shows an example:

```
return CommonOAuth2Provider.GOOGLE.getBuilder("google")
        .clientId("google-client-id")
        .clientSecret("google-client-secret")
        .redirectUriTemplate("{baseUrl}/login/oauth2/callback/{registrationId}")
        .build();
```

# 31.4 Token Endpoint

## 31.4.1 OAuth2AccessTokenResponseClient

`OAuth2AccessTokenResponseClient` is responsible for exchanging an authorization grant credential for an access token credential at the Authorization Server's Token Endpoint.

The default implementation of `OAuth2AccessTokenResponseClient` is `NimbusAuthorizationCodeTokenResponseClient`, which exchanges an authorization code for an access token at the Token Endpoint.

`NimbusAuthorizationCodeTokenResponseClient` uses the Nimbus OAuth 2.0 SDK internally.

If you would like to provide a custom implementation of `OAuth2AccessTokenResponseClient` that uses Spring Framework 5 reactive `WebClient` for initiating requests to the Token Endpoint, configure it as shown in the following example:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.oauth2Login()
				.tokenEndpoint()
					.accessTokenResponseClient(this.accessTokenRes
					...
	}

	private OAuth2AccessTokenResponseClient<OAuth2AuthorizationCodeGrantRequest> a
		return new SpringWebClientAuthorizationCodeTokenResponseClient();
	}
}
```

## 31.5 UserInfo Endpoint

The UserInfo Endpoint includes a number of configuration options, as described in the following sub-sections:

- Section 31.5.1, “Mapping User Authorities”
- Section 31.5.2, “Configuring a Custom OAuth2User”
- Section 31.5.3, “OAuth 2.0 UserService”
- Section 31.5.4, “OpenID Connect 1.0 UserService”

### 31.5.1 Mapping User Authorities

After the user successfully authenticates with the OAuth 2.0 Provider, the `OAuth2User.getAuthorities()` (or `OidcUser.getAuthorities()`) may be mapped to a new set of `GrantedAuthority` instances, which will be supplied to `OAuth2AuthenticationToken` when completing the authentication.

`OAuth2AuthenticationToken.getAuthorities()` is used for authorizing requests, such as in `hasRole('USER')` or `hasRole('ADMIN')`.

There are a couple of options to choose from when mapping user authorities:

- Using a `GrantedAuthoritiesMapper`
- Delegation-based strategy with `OAuth2UserService`

#### Using a `GrantedAuthoritiesMapper`

Provide an implementation of `GrantedAuthoritiesMapper` and configure it as shown in the following example:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.oauth2Login()
				.userInfoEndpoint()
					.userAuthoritiesMapper(this.userAuthoritiesMap
					...
	}

	private GrantedAuthoritiesMapper userAuthoritiesMapper() {
		return (authorities) -> {
			Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

			authorities.forEach(authority -> {
				if (OidcUserAuthority.class.isInstance(authority)) {
					OidcUserAuthority oidcUserAuthority = (OidcUse

					OidcIdToken idToken = oidcUserAuthority.getIdT
					OidcUserInfo userInfo = oidcUserAuthority.getU

					// Map the claims found in idToken and/or user
					// to one or more GrantedAuthority's and add i

				} else if (OAuth2UserAuthority.class.isInstance(author
					OAuth2UserAuthority oauth2UserAuthority = (OAu

					Map<String, Object> userAttributes = oauth2Use

					// Map the attributes found in userAttributes
					// to one or more GrantedAuthority's and add i

				}
			});

			return mappedAuthorities;
		};
	}
}
```

Alternatively, you may register a `GrantedAuthoritiesMapper` `@Bean` to have it automatically applied to the configuration, as shown in the following example:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.oauth2Login();
```

```
	}

	@Bean
	public GrantedAuthoritiesMapper userAuthoritiesMapper() {
		...
	}
}
```

### Delegation-based strategy with `OAuth2UserService`

This strategy is advanced compared to using a `GrantedAuthoritiesMapper`, however, it's also more flexible as it gives you access to the `OAuth2UserRequest` and `OAuth2User` (when using an OAuth 2.0 UserService) or `OidcUserRequest` and `OidcUser` (when using an OpenID Connect 1.0 UserService).

The `OAuth2UserRequest` (and `OidcUserRequest`) provides you access to the associated `OAuth2AccessToken`, which is very useful in the cases where the *delegator* needs to fetch authority information from a protected resource before it can map the custom authorities for the user.

The following example shows how to implement and configure a delegation-based strategy using an OpenID Connect 1.0 UserService:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.oauth2Login()
				.userInfoEndpoint()
					.oidcUserService(this.oidcUserService())
					...
	}

	private OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
		final OidcUserService delegate = new OidcUserService();

		return (userRequest) -> {
			// Delegate to the default implementation for loading a user
			OidcUser oidcUser = delegate.loadUser(userRequest);

			OAuth2AccessToken accessToken = userRequest.getAccessToken();
			Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

			// TODO
			// 1) Fetch the authority information from the protected resou
			// 2) Map the authority information to one or more GrantedAuth

			// 3) Create a copy of oidcUser but use the mappedAuthorities
			oidcUser = new DefaultOidcUser(mappedAuthorities, oidcUser.get
```

```
                        return oidcUser;
                };
        }
}
```

### 31.5.2 Configuring a Custom OAuth2User

`CustomUserTypesOAuth2UserService` is an implementation of an `OAuth2UserService` that provides support for custom `OAuth2User` types.

If the default implementation (`DefaultOAuth2User`) does not suit your needs, you can define your own implementation of `OAuth2User`.

The following code demonstrates how you would register a custom `OAuth2User` type for GitHub:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .oauth2Login()
                                .userInfoEndpoint()
                                        .customUserType(GitHubOAuth2User.class, "githu
                                        ...
        }
}
```

The following code shows an example of a custom `OAuth2User` type for GitHub:

```
public class GitHubOAuth2User implements OAuth2User {
        private List<GrantedAuthority> authorities =
                AuthorityUtils.createAuthorityList("ROLE_USER");
        private Map<String, Object> attributes;
        private String id;
        private String name;
        private String login;
        private String email;

        @Override
        public Collection<? extends GrantedAuthority> getAuthorities() {
                return this.authorities;
        }

        @Override
        public Map<String, Object> getAttributes() {
                if (this.attributes == null) {
                        this.attributes = new HashMap<>();
                        this.attributes.put("id", this.getId());
```

```
                        this.attributes.put("name", this.getName());
                        this.attributes.put("login", this.getLogin());
                        this.attributes.put("email", this.getEmail());
                }
                return attributes;
        }

        public String getId() {
                return this.id;
        }

        public void setId(String id) {
                this.id = id;
        }

        @Override
        public String getName() {
                return this.name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getLogin() {
                return this.login;
        }

        public void setLogin(String login) {
                this.login = login;
        }

        public String getEmail() {
                return this.email;
        }

        public void setEmail(String email) {
                this.email = email;
        }
}
```

`id`, `name`, `login`, and `email` are attributes returned in GitHub's UserInfo Response. For detailed information returned from the UserInfo Endpoint, see the API documentation for "Get the authenticated user".

### 31.5.3 OAuth 2.0 UserService

`DefaultOAuth2UserService` is an implementation of an `OAuth2UserService` that supports standard OAuth 2.0 Provider's.

`OAuth2UserService` obtains the user attributes of the end-user (the resource owner) from the UserInfo Endpoint (by using the access token granted to the client during the authorization flow) and returns an `AuthenticatedPrincipal` in the form of an `OAuth2User`.

If the default implementation does not suit your needs, you can define your own implementation of `OAuth2UserService` for standard OAuth 2.0 Provider's.

The following configuration demonstrates how to configure a custom `OAuth2UserService`:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.oauth2Login()
				.userInfoEndpoint()
					.userService(this.oauth2UserService())
					...
	}

	private OAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {
		return new CustomOAuth2UserService();
	}
}
```

### 31.5.4 OpenID Connect 1.0 UserService

`OidcUserService` is an implementation of an `OAuth2UserService` that supports OpenID Connect 1.0 Provider's.

`OAuth2UserService` is responsible for obtaining the user attributes of the end user (the resource owner) from the UserInfo Endpoint (by using the access token granted to the client during the authorization flow) and return an `AuthenticatedPrincipal` in the form of an `OidcUser`.

If the default implementation does not suit your needs, you can define your own implementation of `OAuth2UserService` for OpenID Connect 1.0 Provider's.

The following configuration demonstrates how to configure a custom OpenID Connect 1.0 `OAuth2UserService`:

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {
```

```
        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .oauth2Login()
                                .userInfoEndpoint()
                                        .oidcUserService(this.oidcUserService())
                                        ...
        }

        private OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
                return new CustomOidcUserService();
        }
}
```

Prev Up Next

30. LDAP Authentication Home 32. JSP Tag Libraries