SFS-PROJEKT
OAUTH-2 – WEB-APP

Dokumentation

Wolfgang Lumetsberger

Thomas Herzog

Roman Lumetsberger

1 Beschreibung der Anwendung

Die erstellte Beispielanwendung besteht im Wesentlichen aus einer Spring-Boot-Webanwendung, welche sowohl Facebook als auch IdentityServer-4 zur Benutzerverwaltung verwendet. Dabei werden alle Benutzer über Facebook als normale Benutzer behandelt. Benutzer, die sich über den IdentityServer-4 einloggen, können sowohl Administrator als auch normaler Benutzer sein.

2 Facebook Authentication

Um Facebook als Provider zu verwenden, muss eine Applikation im Facebook-Developer-Portal[1] angelegt werden. Dadurch bekommt man die benötigten Parameter welche dann in der Spring-Boot-Anwendung verwendet werden können.

3 OAuth-2 Authentication Server

Um Benutzerauthentifizierung über OAuth durchzuführen, wird ein Authentication-Server benötigt, welcher die Benutzerdaten bzw. *Claims* zur Verfügung stellt. In diesem Beispiel wurde die *Open Source* Software *IdentityServer4*[2] verwendet.

Der Server kann einfach in eine *.Net-Core* Applikation integriert und dann konfiguriert werden. Die Konfiguration erfolgte in der Beispielanwendung direkt im Code, doch es werden auch viele verschiedene Datenbanken unterstützt.

Konfiguration - Clients

Um externe Benutzer authentifizieren zu können, muss die Anwendung zuerst angelegt und konfiguriert werden. Folgende Einstellungen müssen gesetzt werden:

- ClientId: Eindeutige ID des Clients, um das Profil später zuordnen zu können.
- AllowedGrantTypes: Definiert die verschiedenen OAuth-Methoden zur Authentifizierung. Im Beispiel wird *AuthorizationCode* verwendet.
- ClientSecrets: Zusätzliches Geheimnis, welches der Client mitschicken muss.
- AllowedScopes: *Scopes*, welche für diesen Client zugelassen werden. Diese definieren welche *Claims* der Client später abfragen kann.
- RedirectUris: URL auf die nach erfolgreicher Authentifizierung umgeleitet wird. Diese muss auch der Client mitschicken und zusammenpassen.

Konfiguration - Benutzer

Für die Authentifizierung werden auch Benutzer benötigt, welche im Beispiel direkt im Code angelegt werden. Jeder Benutzer kann verschiedene *Claims* besitzen, welche dann abgefragt werden können. Eine Besonderheit in der Beispielanwendung ist das Zusammenspiel zwischen *.Net IdentityServer* und *Spring-Security*, da hier ein spezielles *Claim* verwendet wird, um die Rolle direkt an die Web-Anwendung zu übergeben. Im folgenden Codeausschnitt sieht man, dass neben dem Namen und der Email-Adresse auch das Property *Authorities* gesetzt wird. Dieses wird dann von *Spring-Security* als Rolle interpretiert.

[1] `developers.facebook.com`
[2] IdentityServer4 s. `https://github.com/IdentityServer/IdentityServer4`

```
new TestUser {
        SubjectId = "5BE86359-073C-434B-AD2D-A3932222DABE",
        Username = "test",
        Password = "test",
        Claims = new List<Claim> {
                        new Claim(JwtClaimTypes.Name, "test user"),
                        new Claim(JwtClaimTypes.Email, "test@sfs.com"),
                        //Spring security role
                        new Claim("authorities", "ROLE_USER"),

        }
```

4 OAuth Spring Boot Demo Anwendung

Es wurde eine Spring-Boot-Anwendung implementiert, die *OAuth*-Authentifizierungen über *Facebook* und über IdentityServer-4, einen *Authorization Server* in *.Net-Core* verwendet. Es werden auch andere *Open Source Authorization Server* angeboten wie z.B. *Keycloak* von der *Apache Software Foundation*, welcher in Java und AngularJS implementiert ist.

Spring Security kann vollständig transparent von der eigentlichen Anwendung konfiguriert werden. Die geschützten Ressourcen können gebündelt in einer Konfigurationsklasse definiert werden, ohne dass die Implementierungen der eigentlichen Anwendung geändert muss. Als Benutzerrepräsentation sollte eine eigene Klasse implementiert werden, die die *Security*-spezifische Implementierung von der Anwendung abstrahiert, damit kann die gewählte Authentifizierungsmethode im Hintergrund ausgetauscht werden.

Listing 1: Spring application.properties

```
# Logging setup
#logging.level.root=DEBUG
logging.level.org.springframework.security=DEBUG

# thymeleaf setup
spring.thymeleaf.cache=false
spring.thymeleaf.check-template-location=true
spring.thymeleaf.prefix=classpath:templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML5
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.content-type=text/html

facebook.client.clientId=1749583525361910
facebook.client.clientSecret=6195e3b92e6032a33fafcb75a1682945
facebook.client.accessTokenUri=https://graph.facebook.com/oauth/access_token
facebook.client.userAuthorizationUri=https://www.facebook.com/dialog/oauth
facebook.client.tokenName=oauth_token
facebook.client.authenticationScheme=query
facebook.client.clientAuthenticationScheme=form
facebook.resource.userInfoUri=https://graph.facebook.com/me

# remote setting
identityserver.client.clientId=sfsclient
identityserver.client.clientSecret=sfsclient
identityserver.client.accessTokenUri=https://sfsidentityserver.azurewebsites.net/connect/token
identityserver.client.userAuthorizationUri=https://sfsidentityserver.azurewebsites.net/connect/authorize
identityserver.client.tokenName=oauth_token
identityserver.client.scope=openid profile role
```

```
identityserver.client.authenticationScheme=query
identityserver.client.clientAuthenticationScheme=form
identityserver.resource.userInfoUri=https://sfsidentityserver.azurewebsites.net/connect/userinfo
```

In der Datei *application.properties* werden die verwendeten *OAuth-Authentication-Server* konfiguriert, jedoch nicht die geschützten Ressourcen. Diese werden in einer Spring-Konfigurationsklasse definiert.

Diese Methode ist Teil der programmatischen Konfiguration von *Spring Security* der Klasse *SecurityConfiguration* und konfiguriert die *HttpSecurity* Instanz.

```
/**
* Configure the htt security for the spring application
*
* @param http the HttpSecurity object to configure
* @throws Exception if an error occurs during the configuration
*/
@Override
protected void configure(HttpSecurity http) throws Exception {
        // Enable OAuth for all resources
        http.antMatcher("/**").authorizeRequests()
        // Exclude open resources such as css, js and open views
        .antMatchers("/", "/dist/**", "/vendor/**", "/login**", "/landing")
                .permitAll()
        // protect listing view for roles 'ADMIN, USER' only
        .antMatchers("/admin/list").hasAnyRole("ADMIN", "USER")
                .anyRequest().authenticated()
        // protect all admin views for role 'ADMIN' only
        .antMatchers("/admin/create", "/admin/delete")
                .hasRole("ADMIN").anyRequest().authenticated()
        // Redirect to login page if error
        .and().exceptionHandling()
                .authenticationEntryPoint(new LoginUrlAuthenticationEntryPoint("/"))
        // Redirect to login page if successfully logged out
        .and().logout().logoutSuccessUrl("/").permitAll()
        // Enable cross site request forgery support
        .and().csrf().csrfTokenRepository(CookieCsrfTokenRepository
                .withHttpOnlyFalse())
        // Add the OAuth filter before the BasicAuthentication filter
        .and().addFilterBefore(ssoFilter(), BasicAuthenticationFilter.class);
}
```

Mit dieser Methode wird der *OAuth-Servlet-Filter* registriert und wird als erster Filter aktiviert *(ordinal=-100)*.

```
    /**
* Register OAuth filter
*
* @param filter the filter to append
* @return the filter registration
*/
@Bean
public FilterRegistrationBean
        oauth2ClientFilterRegistration(OAuth2ClientContextFilter filter) {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(filter);
        registration.setOrder(-100);
        return registration;
}
```

Die folgende Methode konfiguriert einen *CompositeFilter*, der mehrere Filter-Instanzen verwaltet und als einen registriert wird. In unserer Implementierung werden die OAuth-Authentifizierung über *Facebook* und unseren *Identity Server* unterstützt.

```
   /**
* Helper method for configuring the BasicAuthentication filter instance.
*
* @return the configured filter instance
*/
private Filter ssoFilter() {
        CompositeFilter filter = new CompositeFilter();
        List<Filter> filters = new ArrayList<>(2);

        // Configure the OAuth filter for facebook authentication
        OAuth2ClientAuthenticationProcessingFilter facebookFilter =
                new OAuth2ClientAuthenticationProcessingFilter("/login/facebook");
        OAuth2RestTemplate facebookTemplate =
                new OAuth2RestTemplate(facebook(), oauth2ClientContext);
        facebookFilter.setRestTemplate(facebookTemplate);
        facebookFilter.setTokenServices(
                new UserInfoTokenServices(facebookResource().getUserInfoUri(),
                                          facebook().getClientId()));
        facebookFilter.setAuthenticationSuccessHandler(
                new SimpleUrlAuthenticationSuccessHandler("/home"));
        filters.add(facebookFilter);

        // Configure the OAuth filter for identity server authentication
        OAuth2ClientAuthenticationProcessingFilter identityServerFilter =
                new OAuth2ClientAuthenticationProcessingFilter("/login/identityserver");
        OAuth2RestTemplate identityServerTemplate =
                new OAuth2RestTemplate(identityserver(), oauth2ClientContext);
        identityServerFilter.setRestTemplate(identityServerTemplate);
        identityServerFilter.setTokenServices(
                new UserInfoTokenServices(identityserverResource().getUserInfoUri(),
                                          identityserver().getClientId()));
        identityServerFilter.setAuthenticationSuccessHandler(
                new SimpleUrlAuthenticationSuccessHandler("/home"));
        filters.add(identityServerFilter);

        // Set configured filters on the composite filter
        filter.setFilters(filters);

        return filter;
}
```

Die folgenden Methoden erstellen die Java-Repräsentation der in der *application.properties* definierten Properties *facebook.** und *identityserver.**. In diese Objekte werden die gesetzten *Properties* der *application.properties* Datei sowie die programmatischen Konfigurationen zusammengeführt.

```
    /**
* @return the configured facebook client
*/
@Bean
@ConfigurationProperties("facebook.client")
public AuthorizationCodeResourceDetails facebook() {
        return new AuthorizationCodeResourceDetails();
}

/**
* @return the configured facebook resource
*/
@Bean
@ConfigurationProperties("facebook.resource")
public ResourceServerProperties facebookResource() {
        return new ResourceServerProperties();
}

/**
* @return the configured identity server client
*/
@Bean
@ConfigurationProperties("identityserver.client")
public AuthorizationCodeResourceDetails identityserver() {
        return new AuthorizationCodeResourceDetails();
}

/**
* @return the configured identity server resource
*/
@Bean
@ConfigurationProperties("identityserver.resource")
public ResourceServerProperties identityserverResource() {
        return new ResourceServerProperties();
}
```

Mit dieser Konfiguration ist die *Security* der implementierten *Spring-Boot*-Anwendung konfiguriert und die Ressourcen der Anwendung sind gemäß der Konfiguration geschützt.

Die Klasse *UserContext* repräsentiert einen Benutzer, der mit der Anwendung interagiert. Dadurch werden die *Spring-Security*-Spezifika von der Anwendung abstrahiert. Die *Views* und die *Controller* arbeiten mit dem *UserContext*-Objekt und nicht mit dem *Authentication* oder *Principal*-Objekten, die mehr Informationen zur Verfügung stellen, als meiner Meinung nach für die Anwendung zugänglich sein sollte.

Die Klasse *UserContext* könnte noch zusätzlich benutzer- und anwendungsspezifische Informationen enthalten, die unabhängig von der *Security* sind.

```
/**
* An object of this class represents a user interacting with the application.
*
* @author Thomas Herzog <t.herzog@curecomp.com>
* @since 01/13/17
*/
public class UserContext {

        public enum Role {
                ADMIN,
                USER,
                ANONYMOUS;
        }

        private final boolean logged;
        private final Role role;
        private final String name;

        public UserContext(boolean logged,
                           Role role,
                           String name) {
                this.logged = logged;
                this.role = role;
                this.name = name;
        }

        public boolean isAdmin() {
                return (isLogged()) && (Role.ADMIN.equals(role));
        }

        public boolean isLogged() {
                return logged;
        }

        public Role getRole() {
                return role;
        }

        public String getName() {
                return name;
        }
}
```

Die Klasse *SecurityUtil* ist für die Erstellung des *UserContext* aus einem *Authentication*-Objekt verantwortlich. Sollte ein Benutzer nicht eingeloggt sein, so wird er als anonymer Benutzer angesehen. Sollte ein Benutzer mehrere Rollen zugewiesen haben, dann wird die Rolle mit den höchsten Privilegien bevorzugt.

```
/**
* @author Thomas Herzog <t.herzog@curecomp.com>
* @since 01/22/17
*/
public class SecurityUtil {

/**
* Create a UserContext object for the given authentication.
*
* @param auth the authentication holding the authentication information
* @return the create user context object
*/
public static final UserContext createUserCtx(final Authentication auth) {
        // If not authentication is available
        // then the user is always considered to be anonymous
        UserContext.Role role = UserContext.Role.ANONYMOUS;
        String name = UserContext.Role.ANONYMOUS.name();

        if (auth != null) {
                name = ((String) auth.getPrincipal());

                for (GrantedAuthority ga : auth.getAuthorities()) {
                        switch (ga.getAuthority()) {
                                case "ROLE_ADMIN":
                                        role = UserContext.Role.ADMIN;
                                        break;
                                case "ROLE_USER":
                                        role = UserContext.Role.USER;
                                        break;
                                default:
                                        role = UserContext.Role.ANONYMOUS;
                                }
                        // Cannot have more privileges, therefore can break here
                        if (UserContext.Role.ADMIN.equals(role)) {
                                break;
                        }
                }
        }

        return new UserContext(!UserContext.Role.ANONYMOUS.equals(role), role, name);
        }
}
```

Die folgende abstrakte Klasse repräsentiert die Basisklasse aller *Controller* und stellt eine *@ModelAttribute* annotierte Methode zur Verfügung, die den *UserContext* für die *Views* bereitstellt, der aus dem *Authentication*-Objekt erstellt wird. Somit können alle *Views* denselben Namen verwenden und es gibt nur eine Stelle, wo der *UserContext* erstellt wird. Mit Diesem Ansatz bleiben die *Controller* befreit von *Security*-spezifischen *Code*.

```
/**
* The base controller which holds the common resources which each controller depends on.
*
* @author Thomas Herzog <t.herzog@curecomp.com>
* @since 01/23/17
*/
public abstract class AbstractController {

        @ModelAttribute("utx")
        public UserContext getUserContext() {
                return SecurityUtil.createUserCtx(
                        SecurityContextHolder.getContext().getAuthentication());
        }
}
```

Die folgende Auszüge aus den *HTML*-Seiten zeigt die Verwendung des *UserContext*.

Der *Logoutbutton* ist nur sichtbar wenn ein Benutzer eingeloggt ist (ADMIN, USER).

```
...
<form th:action="@{/logout}" method="post" th:if="${utx.logged}">
        <input class="btn btn-nav" type="submit" value="Logout"/>
</form>
...
```

Der *Delete Button* ist nur sichtbar wenn ein Benutzer eingeloggt ist und der Rolle AMIN zugewiesen ist.

```
...
<form th:method="post" th:action="@{/admin/delete}" th:if="${utx.admin}">
        <input type="hidden" id="name" th:name="name" th:value="${project.name}"/>
        <button type="submit" name="submit" class="btn btn-primary">Delete</button>
</form>
...
```

Die *Controller* erfordern keine Modifikation bezüglich der *Security*, da die Konfiguration in der programmatischen Konfiguration der Klasse *SecurityConfiguration* erfolgt und daher die *Controller* abstrahiert von der *Security* sind.