#JAVAPRO #SpringBoot #Security #Authentication

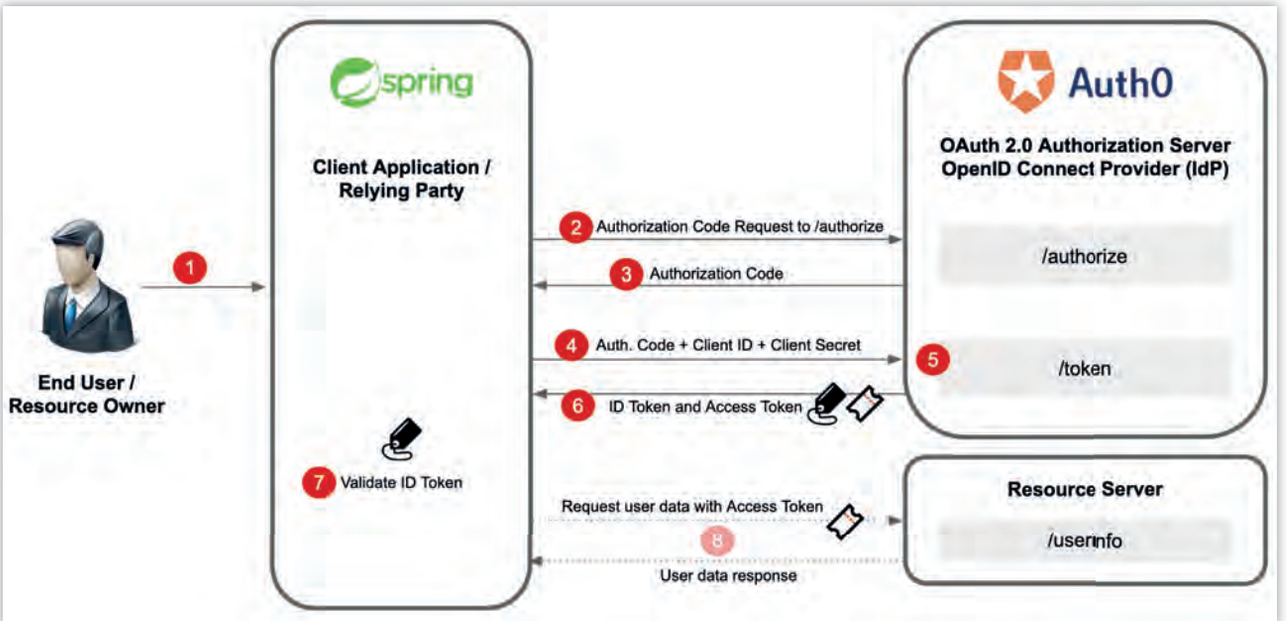
Authentifizierung mit OpenID-Connect

Der Artikel zeigt, wie Benutzern einer Spring-MVC-Applikation ermöglicht werden kann, sich mittels OpenID-Connect über einen eigenen Identity-Provider (IdP) zu authentifizieren, ohne selbst einen Autorisierungsserver implementieren zu müssen. Im zweiten Schritt wird gezeigt, wie Entwickler mühelos weitere Social-IDPs, wie zum Beispiel Facebook, Twitter, LinkedIn etc. ohne viel Mehraufwand föderiert anbinden können.

Spring-Security ist nicht nur ein mächtiges Framework zur Authentifizierung und Autorisierung von Benutzern, sondern auch der De-Facto Standard zur Absicherung von Java-Applikationen. In Verbindung mit einer Identity-as-a-Service-Plattform (IDaaS) lassen sich sicherheitsrelevante Features in kurzer Zeit und mit wenig eigenem Code abbilden. Mit Spring 5.1 wurde der OAuth 2.0 und OpenID-Connect (OIDC)-Unterstützung rundum erneuert bzw. reimplementiert. Während das OAuth 2.0 Protokoll zur Autorisierung bzw. Access-Delegation auf Ressourcen wie APIs oder sonstige Backends dient, handelt es sich bei OpenID-Connect um einen Identity-Layer, der auf OAuth 2.0 aufsetzt und rein der Authentifizierung des Benutzers dient.

Autor:

Mathias Conradt ist Senior-Solutions-Engineer bei Auth0 und beschäftigt sich vorwiegend mit Identity & Access Management Lösungen rund um OAuth und OpenID Connect.



OpenID-Connect mit Authorization-Code-Grant[1] (Abb. 1)

OpenID-Connect und der OAuth 2.0 Authorization-Code-Grant

Konzeptionell sieht der Ablauf einer OpenID-Connect basierten Authentifizierung gemäß dem OAuth 2.0 Authorization-Code-Grant wie in (Abb 1.) aus.

1. Der Benutzer ruft die Spring-Applikation im Browser auf.
2. Die Spring-Applikation bzw. Spring-Security leitet den Benutzer aufgrund der Sicherheitskonfiguration zum Autorisierungs-Server/IdP (`/authorize` Endpunkt) weiter, in diesem Fall `Auth0`. Sofern keine aktive Session beim `IdP` besteht, wird dem Benutzer eine Login-Seite und/oder ein Consent-Dialog angezeigt. Der Benutzer authentifiziert sich mittels einer der konfigurierten Login-Optionen, zum Beispiel `Benutzername / Passwort`. Er sieht daraufhin einen Consent-Dialog der die Berechtigungen, zum Beispiel `Auslesen des Benutzerprofils` auflistet, die der Autorisierungs-Server/IdP `Auth0` der anfragenden Spring-Applikation gewähren wird.
3. Der Autorisierungs-Server/IdP `Auth0` leitet den User mit einem `Authorization-Code` zurück an die Spring-Applikation.
4. Spring-Security sendet den Code an den Autorisierungsserver/IdP (`/oauth/token` Endpunkt) zusammen mit `Client ID` und `Client Secret`.
5. Der Autorisierungs-Server/IdP `Auth0` verifiziert den `Code`, `Client ID` und `Client Secret`.
6. Der Autorisierungs-Server /IdP `Auth0` gibt `ID Token` und `Access Token` und optional `Refresh Token` an die Spring-Applikation zurück.
7. Die Spring-Applikation validiert und dekodiert den `ID Token` und kann dessen Inhalt (Payload) zur Anzeige des Benutzerprofils nutzen.
8. Optional kann die Spring-Applikation den `Access-Token` nutzen, um weitere Benutzerinformationen vom gemäß OpenID-Connect standardisierten UserInfo-Endpunkt des Autorisierungsserver/IdP abzurufen. In diesem Beispiel reicht der `ID Token` allein aus.

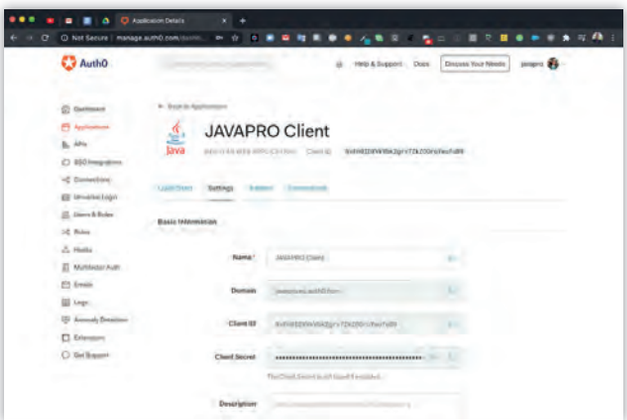
Theoretisch kann für eine reine OpenID-Connect-Authentifizierung, bei der lediglich der ID-Token und kein Access-Token verwendet wird, auch der Implicit-Grant mit Form-Post-Response-Mode[2] in Erwägung gezogen werden, allerdings wird dieser seitens Spring Security 5.1 derzeit nicht unterstützt. Dies würde die generelle Komplexität verringern und das Verwalten von Client-Secrets unnötig machen.

Die Implementierung erfolgt in zwei Schritten. Zuerst wird der Identity-Provider seitens Auth0 konfiguriert, danach erfolgt die Integration in ein Spring-Boot- bzw. Spring-Security-Projekt.

IdP-Konfiguration seitens Auth0

Auf `https://Auth0.com` wird zunächst ein kostenfreier Account sowie Mandanten erstellt. Die geplante Java-Applikation im Auth0-Dashboard wird unter dem Punkt `Applications` registriert. Als Technologie-Stack wird Regular-Web-App und Java-Spring-MVC gewählt. Nach erfolgreichem Anlegen der Client-Applikation und Wechsel zum Settings-Tab erhält man eine Client-ID sowie ein Client-Secret. Dieses wird später für die Konfiguration des Auth0-IdP in der Spring-Applikation benötigt. Es werden noch in den Settings die erlaubten Callback-URLs konfiguriert `http://localhost:3000/login/oauth2/code/Auth0` sowie die erlaubten Logout-URLs auf `http://localhost:3000`. Optional kann ein Icon vergeben werden. In diesem Beispiel wird das Java-Logo verwendet (Abb. 2).

Nun kann mit der eigentlichen App-Entwicklung begonnen werden.



Client-Application-Settings im Auth0-Dashboard (Abb. 2)

Spring-Boot Projektinitialisierung

Es wird ein Spring-Boot-Projekt mit folgenden, wesentlichen für OpenID-Connect relevanten Abhängigkeiten erstellt **(Listing 1)**. Das gesamte Beispielprojekt ist auf GitHub[3] verfügbar.

(Listing 1)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>Spring Security-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>Spring Security-oauth2-jose</artifactId>
</dependency>
```

Um die Anwendung für nicht authentifizierten Zugriffen zu schützen, wird zunächst eine Konfigurationsklasse erstellt, die von **`WebSecuritConfigurerAdapter`** ableitet. In ihr wird die **`configure`** Methode überschrieben und die Sicherheitsregeln für eingehende HTTP-Requests definiert.
Im Beispiel soll für die gesamte Applikation eine Authentifizierung vorausgesetzt sein. Zusätzlich soll OAuth2-Login aktiviert werden. Schließlich wird noch ein eigener **`LogoutHandler`** definiert, der den Default-Handler überschreibt **(Listing 2)**.

(Listing 2)

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepo-
sitory;
    private final LogoutController logoutController;

    public SecurityConfig(LogoutController logoutController) {
        this.logoutController = logoutController;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2Login()
            .and()
            .logout()
            .addLogoutHandler(logoutController);
        }
    }
}
```

Der **`oauth2Login`** holt sich die Konfiguration des zu verwendenden Identity-Provider aus der Applikationskonfiguration **`application.yml`**. Zunächst wird der **`Auth0`** als Identity-Provider definiert **(Listing 3)**.

(Listing 3)

```
[...]
spring:
  security:
    oauth2:
      client:
        registration:
          Auth0:
            client-name: JAVAPRO Client
            client-id: {clientID}
            client-secret: {clientSecret}
            scope:
            - openid
            - profile
            - email
            - offline_access
            authorization-grant-type: authorization_code
            logout-uri: https://javapro.eu.Auth0.com/v2/
logout?client_id=[...]
        provider:
          Auth0:
            issuer-uri: https://javapro.eu.Auth0.com/
            user-name-attribute: name
```

Die in **(Listing 3)** gezeigte Konfiguration unterteilt sich in zwei Teile: es wird zum einen der Identity-Provider definiert **(Listing 4)**.

(Listing 4)

```
provider:
  Auth0:
    issuer-uri: https://javapro.eu.Auth0.com/
```

Über die Issuer-URI ist Spring im Stande, sich über das als OpenID-Connect-Discovery[4] standardisierte Verfahren die notwendigen OAuth 2.0 Endpunkte wie Authorization-Endpoint und Token-Endpoint zur Kommunikation zwischen Client-Applikation (Spring) und Autorisierungs-Server (**Auth0**) selbständig herauszusuchen. Der zweite Teil der **application.yml** (Listing 3) registriert die zugehörige Client-Applikation (Listing 5).

(Listing 5)

```
registration:
  google:
    client-name: JAVAPRO Client
    client-id: {clientID}
    client-secret: {clientSecret}
    scope:
      - openid
      - profile
      - email
      - offline_access
    authorization-grant-type: authorization_code
    logout-uri: [...]
```

Der **client-name** kann hierbei beliebig gewählt werden. Es bietet sich an für eine bessere Übersicht den gleichen Namen zu verwenden wie in **Auth0** bei der Registrierung der Applikation. Als **scope** wird **openid profile email offline_access** angefragt. Hiermit wird angezeigt, dass es sich um einen OpenID-Connect-Request handelt, der Profildaten des Users zurückerhalten und neben Access- und ID-Token auch ein Refresh-Token erhalten möchte (**offline_access**). Da es sich um eine reguläre Web-Applikation mit Backend handelt, wird der Authorization-Code-Grant von OAuth 2.0 verwendet, zumal Spring-Security aktuell auch nur diesen unterstützt:
authorization-grant-type: authorization_code
Als nächstes wird ein Controller benötigt. Der **HomeController**, der authentifizierte Requests an **http://localhost:3000/** entgegennimmt und das Benutzerprofil - extrahierte Claims des ID-Tokens - darstellt. Dazu nimmt eine Methode **index** die GET-Anfragen am Root-Pfad entgegen. In ihr steht sowohl ein **Model** Objekt zur Verfügung, als auch der erhaltene OAuth2AuthenticationToken, welcher eine **OAuth2-Authentifizierung** repräsentiert und die Token-Information beinhaltet (Listing 6).

(Listing 6)

```
@Controller
public class HomeController {
    public HomeController(OAuth2AuthorizedClientService authorizedClientService) {
        this.authorizedClientService = authorizedClientService;
    }
    @RequestMapping("/")
    public String index(Model model, OAuth2AuthenticationToken
authentication) {
        model.addAttribute("userName", authentication.getName());
        model.addAttribute("clientName", authorizedClient.getCli-
entRegistration().getClientName());
        model.addAttribute("userinfo", authentication.getPrin-
cipal().getAttributes());
        return "index";
    }
}
```

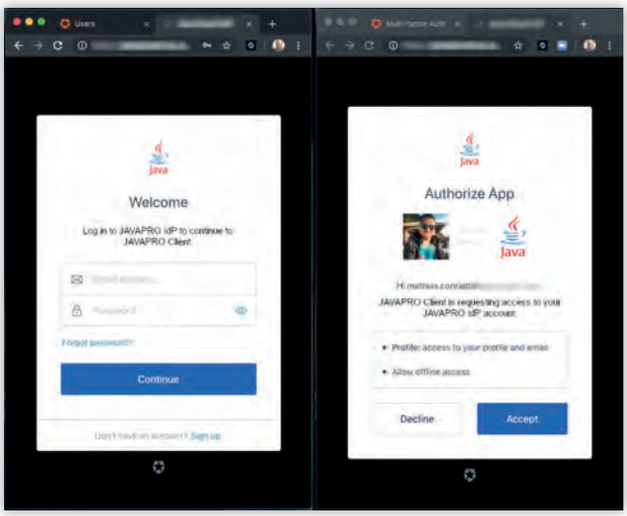
Die Claims aus dem ID-Token, die letztlich die Benutzer-Attribute und somit Benutzerprofil darstellen, werden mittels **authentication.getPrincipal().getAttributes()** erhalten und können diese direkt an die View (**index.html**) unter dem gewählten Attribut-Namen **userInfo** übergeben. Erfreulicherweise übernimmt Spring-Security OAuth- bzw. OpenID-Support automatisch den OAuth 2.0 gemäßen Tausch von Autorisierungscode gegen einen Access- und ID-Token. Auch wird das Dekodieren und Validieren des ID-Tokens, welches vom Identity-Provider als Base64Url-dekodierter und signierter JSON-Web-Token (JWT)[5] zurückkommt, automatisch vorgenommen und nimmt dem Entwickler bereits einiges an manueller Implementierungsarbeit ab. Die zugehörige View, welche Thymeleaf als Template-Library verwendet und an die das Benutzerprofil über das **Model** übergeben wird, ist in (Listing 7) beschrieben. Sie zeigt neben dem Namen der Client-Applikation das Avatar-Foto des Benutzers sowie das Profil an. Das Benutzerfoto wird seitens **Auth0** automatisch vom Gravatar-Dienst auf Basis der bei der Registrierung verwendeten Benutzer-Mail-Adresse aufgesucht, sofern vorhanden.

(Listing 7)

```
[...]
<h1>OAuth 2.0 Login with Spring Security</h1>
<div>
  You are successfully logged in <span style="font-weight:bold"
th:text="${userName}"></span>
  via the OAuth 2.0 Client <span style="font-weight:bold"
th:text="${clientName}"></span>
</div>
<div><img th:src="${userinfo.get('picture')}" height="100"
width="100"/></div>
<div>
  User info from ID token:
  <ul>
    <li th:each="attribute : ${userinfo}">
      <span style="font-weight:bold" th:text="${attribute.
key}"/>: <span th:text="${attribute.value}"></span>
    </li>
    </ul>
</div>
[...]
```

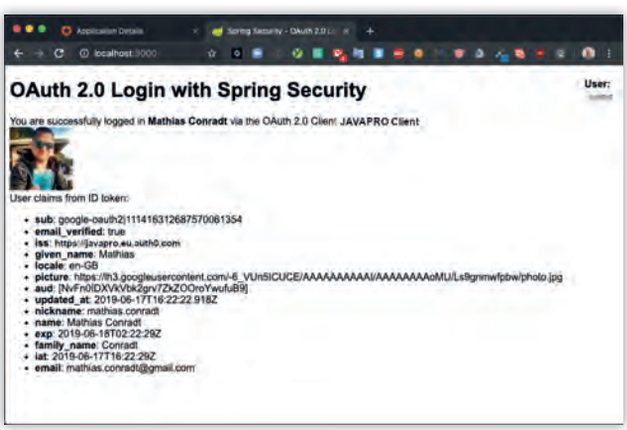
Die Client-Applikation wird mittels **mvn spring-boot:run** gestartet und die URL **http://localhost:3000** wird im Browser

aufgerufen. Da noch keine Authentifizierung vorliegt, wird man sofort zum konfigurierten Identity-Provider **Auth0** weitergeleitet und aufgefordert sich einzuloggen bzw. einen Account zu erstellen, sofern dieser noch nicht besteht. Außerdem erscheint ein Consent-Screen, mit der die Einwilligung eingeholt wird, dass die Spring-Anwendung Zugriff auf das Profil bei **Auth0** erhalten darf (Abb. 3).



Login- und Registrierungsseite des Identity Providers. (Abb. 3)

Nach erfolgreicher Authentifizierung wird das Benutzerprofil, konkret die sogenannten Claims des ID-Tokens, dargestellt (Abb. 4).



Benutzerprofil. (Abb. 4)

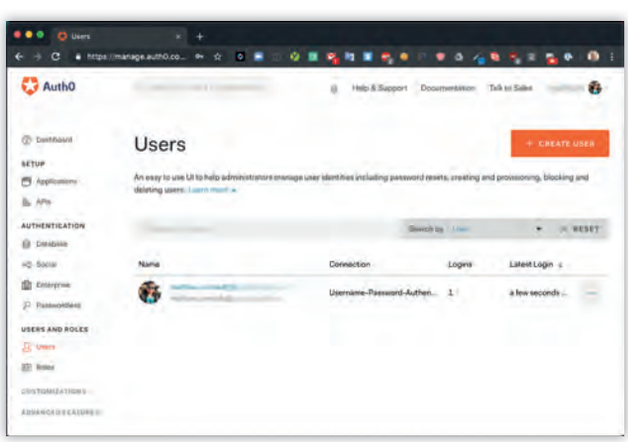
Entwickler haben in Bezug auf den Logout die Möglichkeit zu entscheiden, ob beim Klicken des Logout-Buttons lediglich die Session der Anwendung beendet wird oder der Benutzer auch am IdP ausgeloggt werden soll. Soll der Benutzer auch am IdP ausgeloggt werden, so muss ein eigener **LogoutController** erstellt werden, der in der **SecurityConfig** registriert wird. Der **LogoutController** ruft den entsprechenden Logout-Endpunkt des IdP auf, in diesem Beispiel **Auth0**. Der IdP-seitige Logout-Endpunkt wird aus der **application.yml** ausgelesen und ist als **logout-uri** definiert (Listing 8).

(Listing 8)

```
https://javapro.eu.Auth0.com/v2/logout?client_id={clientID}&re-
turnTo=http://localhost:3000.

@Controller
public class LogoutController extends SecurityContextLogout-
Handler {

    private final ClientRegistrationRepository clientRegistrati-
onRepository;
    private Logger logger = LoggerFactory.getLogger(LogoutCont-
roller.class);

    @Value("${spring.security.oauth2.client.registration.Auth0.
logout-uri}")
    private String logoutUrl;

    public LogoutController(ClientRegistrationRepository client-
RegistrationRepository) {
        this.clientRegistrationRepository = clientRegistrationRe-
pository;
    }

    @Override
    public void logout(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Authentication authen-
tication) {
        super.logout(httpServletRequest, httpServletResponse,
authentication);
        try {
            httpServletResponse.sendRedirect(logoutUrl);
        } catch (IOException ioe) {
            logger.error("Error logging out.", ioe);
        }
    }
}
```

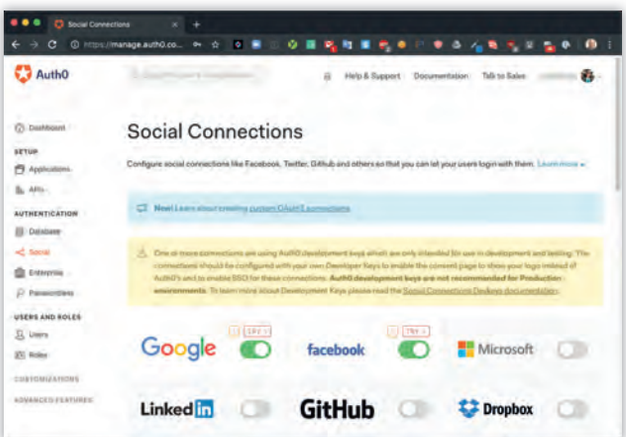
Entscheidet sich der Entwickler dazu, beim Logout lediglich die aktuelle Session der Anwendung zu beenden, nicht jedoch die des IdP, kann auf den **LogoutController** sowie die Methoden **logout** und **addLogoutHandler** in der **SecurityConfig** Konfigurationsklasse verzichtet werden.

Auth0-Dashboard, Social-Login und MFA-Konfiguration

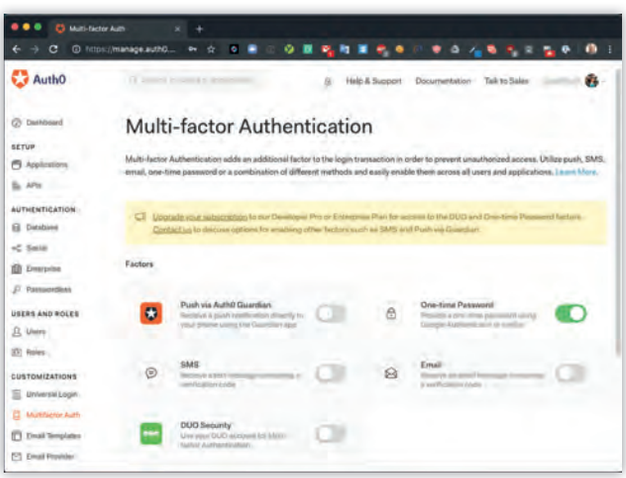


Benutzerübersicht im Auth0-Dashboard (Abb. 5)

Im **Auth0** Dashboard kann nun unter dem Punkt **Users & Roles** auch der soeben registrierte Benutzer gefunden werden (Abb. 5). Um jetzt neben der **Auth0** Datenbank weitere föderierte Social-Identity-Provider wie Google und Facebook zu integrieren (Abb. 6) sowie beispielsweise Multi-Factor-Authentication (MFA) zu aktivieren (Abb. 7), kann dies rein über die Konfiguration innerhalb von **Auth0** vorgenommen werden, ohne dabei die Java-Applikation anfassen zu müssen.



Aktivierung weiterer föderierter Social-Identity-Provider (Abb. 6)

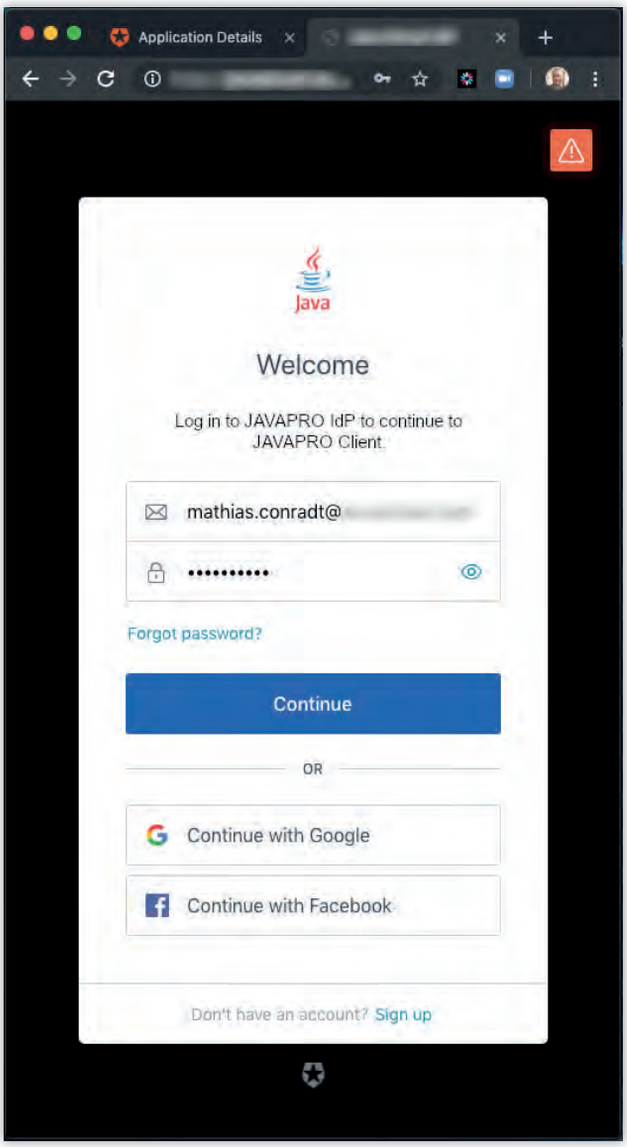


Aktivierung von MFA mittels Google-Authenticator (Abb. 7)

Loggt man sich nun aus der Web-Applikation unter **http://localhost:3000** aus und versucht sich erneut einzuloggen, stellt man fest, dass nun sowohl zwei weitere Buttons für Facebook und Google als Möglichkeit zur Authentifizierung zur Verfügung stehen (Abb. 8). Auch die Aufforderung nach dem Login erscheint, die Multi-Factor-Authenfication (MFA) über das Scannen eines QR-Codes zu initialisieren.

Fazit:

Ein OAuth 2.0 bzw. OpenID-Connect-basierter Login ist in Spring Security 5.1 mit sehr wenig Zeilen Code und minimaler Konfiguration möglich. Das Framework nimmt dem Entwickler einiges



Login-Widget mit aktivierten Social-Connections (Abb. 8)

an Implementierungsarbeit ab, was die Details des OAuth2-Protokolls angeht. Statt weitere Social-Identity-Provider direkt in der Java-Applikation zu konfigurieren, werden diese stattdessen mittels **Auth0** als Broker föderiert integriert. Dies hat den Vorteil, dass die Konfiguration an zentraler Stelle vorgenommen werden kann. Bei der Entwicklung weiterer Client-Applikationen skaliert dies hervorragend und vermeidet doppelten Aufwand. Dies gilt auch für die Bereitstellung von MFA. Als netter Seiteneffekt kann so ebenfalls ein Single-Sign-On über alle Applikationen hinweg bereitgestellt werden.

Quellen:

1 http://bit.ly/grant-a1
2 http://bit.ly/oauth-m2
3 http://bit.ly/github-3
4 http://bit.ly/openid-4
5 https://jwt.io/