Identity und Access Management leicht gemacht

Mathias Conradt, Auth0

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 wurden *OAuth 2.0* und die *OpenID-Connect-(OIDC)*-Unterstützung rundum erneuert beziehungsweise re-implementiert. Während das *OAuth-2.0*-Protokoll zur Autorisierung oder 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 Benutzer-Authentifizierung dient.

Im Folgenden wollen wir uns anschauen, wie wir Benutzern einer Spring-MVC-Applikation ermöglichen können, sich mittels *OpenID Connect* über einen eigenen *Identity-Provider (IdP)* zu authentifizieren, ohne selbst einen *Autorisierungsserver* implementieren zu müssen. Hierfür nutzen wir Auth0 als IDaaS-Plattform. Technisch wird hier – für den Entwickler transparent – eine Mongo-Datenbank in der Auth0-AWS-Cloud verwendet. Im zweiten Schritt schauen wir uns an, wie wir als Entwickler mühelos weitere Social IDPs (Facebook, Twitter, LinkedIn etc.) ohne viel Mehraufwand föderiert anbinden können.

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 folgt aus *(siehe Abbildung 1)*:

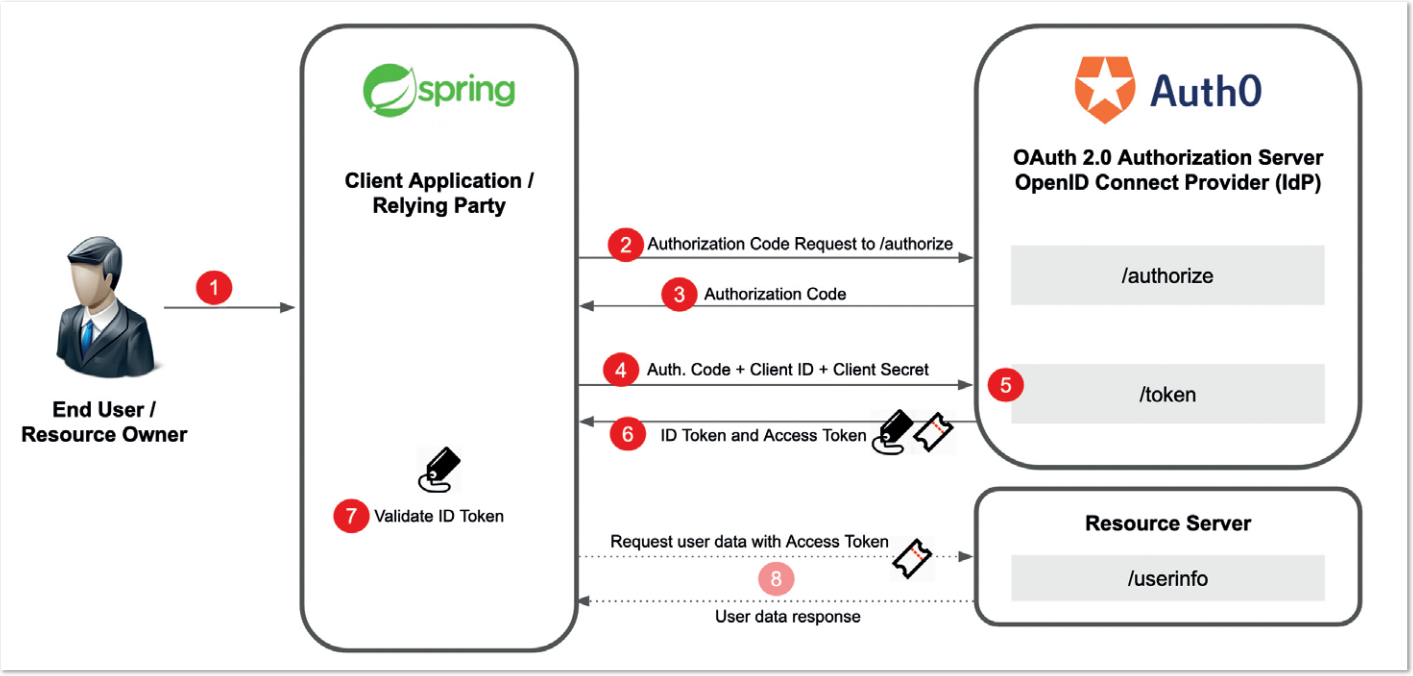

Abbildung 1: OpenID Connect mit Authorization Code Grant [1]

1. Der Benutzer ruft die Spring-Applikation im Browser auf.
2. Die Spring-Applikation beziehungsweise Spring Security leitet den Benutzer aufgrund unserer Sicherheitskonfiguration zum Autorisierungsserver/IdP (`/authorize`-Endpunkt) weiter, in unserem 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 mit einer der konfigurierten Login-Optionen (zum Beispiel Benutzername/Passwort) und sieht daraufhin einen Consent-Dialog, der die Berechtigungen (beispielsweise das Auslesen des Benutzerprofils) auflistet, die der Autorisierungsserver/IdP (Auth0) der anfragenden Spring-Applikation gewähren wird.
3. Der Autorisierungs-Server/IdP (Auth0) leitet den User mit einem *„Authorization Code"* zurück an unsere Spring-Applikation.
4. Spring Security sendet den Code an den Autorisierungsserver/IdP. (`/oauth/token`-Endpunkt) zusammen mit Client ID und Client Secret.
5. Der Autorisierungsserver/IdP (Auth0) verifiziert den Code, Client ID und Client Secret.
6. Der Autorisierungsserver/IdP (Auth0) gibt *„ID-Token"* sowie *„Access-Token"* (und optional *„Refresh-Token"*) an die Spring-Applikation zurück.
7. Unsere 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 Autorisierungsservers/IdP abzurufen. Für unsere Demo reicht uns jedoch der *„ID-Token"*.

Eine Anmerkung am Rande: Theoretisch könnten wir für eine reine OpenID-Connect-Authentifizierung, bei der wir lediglich den *„ID-Token"* und keinen *„Access-Token"* verwenden, auch den *„Implicit Grant mit Form Post Response Mode"* *[2]* in Erwägung ziehen, allerdings wird dieser von 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 konfigurieren wir unseren Identity-Provider seitens Auth0, danach erfolgt die Integration in ein Spring-Boot-beziehungsweise Spring-Security-Projekt.

IdP-Konfiguration seitens Auth0

Wir erstellen zunächst auf *https://auth0.com* einen kostenfreien Account sowie Mandanten und registrieren unsere geplante Java-Applikation im Auth0-Dashboard unter dem Punkt *Applications*. Als Technologie-Stack wählen wir *Regular Web App* und *Java Spring MVC* aus. Nach erfolgreichem Anlegen der Client-Applikation und Wechsel zum *Settings*-Tab erhalten wir eine *Client ID* sowie ein *Client Secret*. Dieses benötigen wir später bei der Konfiguration des Auth0-IdP in der Spring-Boot-Applikation. In den Settings konfigurieren wir noch die erlaubten Callback-URLs auf *http://localhost:3000/login/oauth2/code/auth0* sowie die erlaubten Logout-URLs auf *http://localhost:3000*. Optional kann ein Icon vergeben werden. Ich verwende in meinem Beispiel das Java-Logo *(siehe Abbildung 2)*.

Nach den getroffenen Vorbereitungen können wir nun mit der eigentlichen App-Entwicklung beginnen.

Spring-Boot-Projektinitialisierung

Wir erstellen ein Spring-Boot-Projekt mit wesentlichen, für *„OpenID Connect"* relevanten Abhängigkeiten *(siehe Listing 1)*. Das gesamte Beispielprojekt ist auf GitHub verfügbar *[3]*.

Um unsere Applikation vor nicht authentifizierten Zugriffen zu schützen, erstellen wir zunächst eine Konfigurations-Klasse, die von `WebSecurityConfigurerAdapter` *[4]* ableitet. In ihr überschreiben wir die `configure()`-Methode und definieren die Sicherheitsregeln für eingehende HTTP-Requests. In unserem Fall wollen wir für die

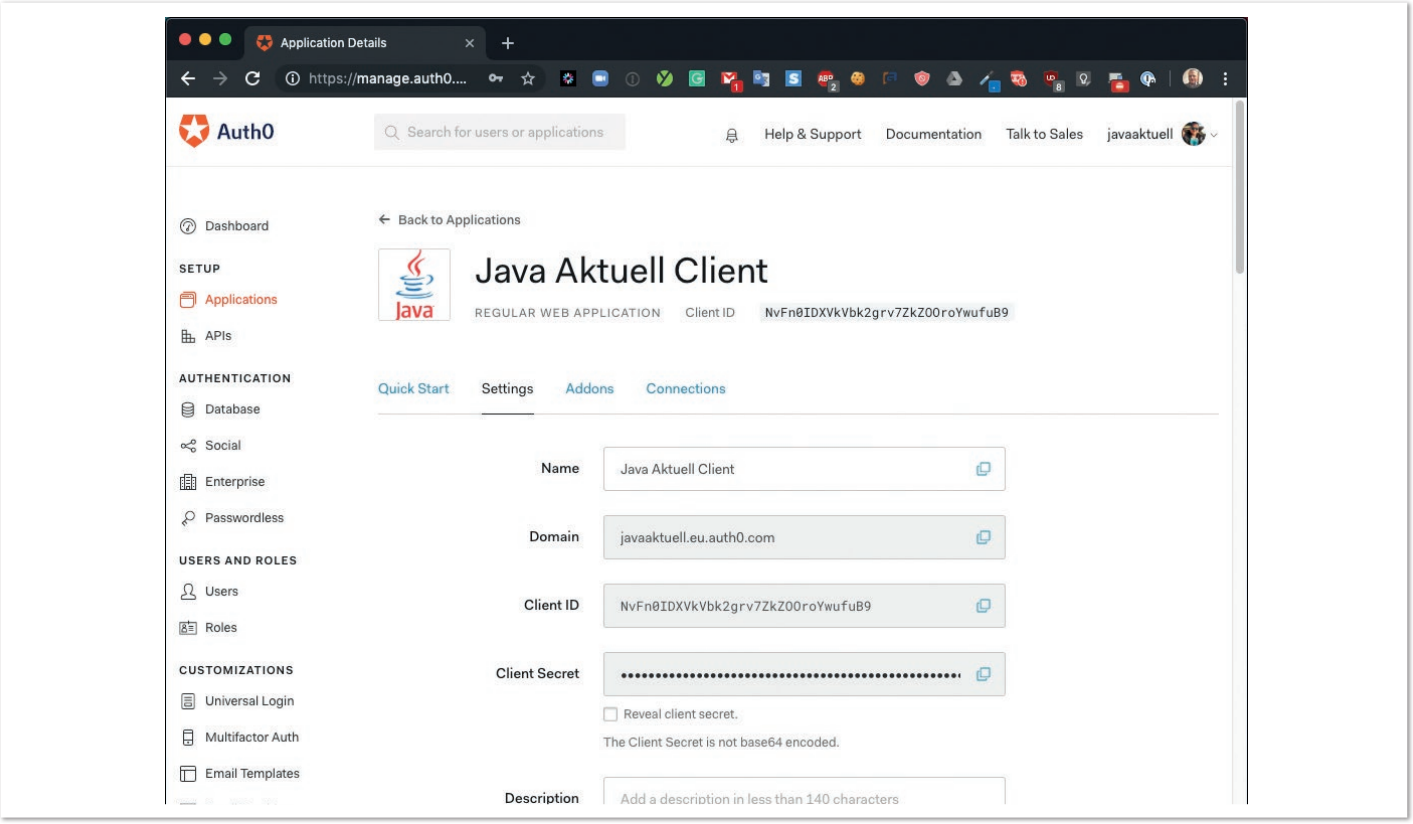

Abbildung 2: Client-Application-Settings im Auth0-Dashboard

gesamte Applikation eine Authentifizierung voraussetzen. Außerdem wollen wir OAuth2-Login aktivieren. Zu guter Letzt definieren wir noch einen eigenen `LogoutHandler`, der den Default-Handler überschreibt *(siehe Listing 2)*.

Der `oauth2Login` holt sich die Konfiguration des zu verwendenden Identity-Providers aus der Applikationskonfiguration `application.yml`. In *Listing 3* definieren wir zunächst Auth0 als Identity-Provider.

Betrachten wir die einzelnen Konfigurationsabschnitte genauer: Die in *Listing 3* gezeigte Konfiguration unterteilt sich in zwei Teile. Der Identity-Provider wird im fett markierten Bereich definiert.

Über die Issuer-URI ist Spring imstande, sich über das als *„OpenID Connect Discovery" [5]* standardisierte Verfahren die notwendigen OAuth-2.0-Endpunkte wie Authorization-Endpoint sowie Token-Endpoint zur Kommunikation zwischen Client-Applikation (Spring) und Autorisierungsserver (Auth0) selbstständig herauszusuchen.

Der obere Teil registriert die zugehörige Client-Applikation *(siehe Listing 4)*. Der `client-name` kann hierbei beliebig gewählt werden; es bietet sich der eigenen Übersicht halber an, den gleichen Namen zu verwenden, wie in Auth0 bei der Registrierung der Applikation gewählt. Als `scope` fragen wir `openid profile email offline_access` an. Hiermit zeigen wir an, dass es sich um einen OpenID

```
<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>
```

Listing 1: Wesentliche Dependencies in der pom.xml

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

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepository;
    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);
        }
    }
}
```

Listing 2: SecurityConfig.java

```
[...]
spring:
  security:
    oauth2:
      client:
        registration:
          auth0:
            client-name: Java Aktuell Client
            client-id: {clientID}
            client-secret: {clientSecret}
            scope:
            - openid
            - profile
            - email
            - offline_access
            authorization-grant-type: authorization_code
            logout-uri: https://javaaktuell.eu.auth0.com/v2/logout?client_id=[...]
        provider:
          auth0:
            issuer-uri: https://javaaktuell.eu.auth0.com/
            user-name-attribute: name
```

Listing 3: Konfiguration des Identity-Providers in der application.yml

```
registration:
  auth0:
    client-name: Java Aktuell Client
    client-id: {clientID}
    client-secret: {clientSecret}
    scope:
      - openid
      - profile
      - email
      - offline_access
    authorization-grant-type: authorization_code
    logout-uri: [...]
```

Listing 4: Registrierung der zugehörigen Client-Applikation

Connect Request handelt, wir die Profildaten des Users zurückerhalten möchten und dass wir neben Access- und ID-Token auch ein Refresh-Token erhalten möchten (`offline_access`). Da es sich um eine reguläre Webapplikation mit Backend handelt, verwenden wir hier den Authorization Code Grant von OAuth 2.0 – zumal Spring Security aktuell auch nur diesen unterstützt: `authorization-grant-type: authorization_code`.

Als Nächstes benötigen wir einen Controller, wir nennen ihn `HomeController`, der authentifizierte Requests an *http://localhost:3000/*

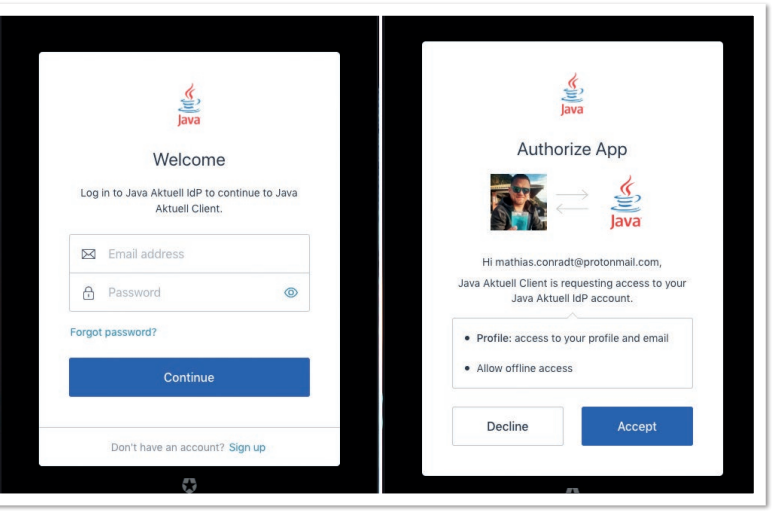

Abbildung 3: Login- und Registrierungs-Seite des Identity-Providers

entgegennimmt und das Benutzerprofil (extrahierte Claims des ID-Tokens) darstellt. Hierfür lassen wir eine Methode `index()` die GET-Anfragen am Root-Pfad entgegennehmen. In ihr haben wir sowohl ein `Model`-Objekt als auch den erhaltenen `OAuth2AuthenticationToken` *[6]*, der eine OAuth2-Authentifizierung repräsentiert und die Token-Information beinhaltet *(siehe Listing 5)*.

Die Claims aus dem ID-Token, die letztlich die Benutzerattribute und somit das Benutzerprofil darstellen, erhalten wir mittels `authentication.getPrincipal().getAttributes()`; wir können diese direkt an die View (index.html) unter dem von uns gewählten Attribut-Namen `userInfo` übergeben. Wie wir hier sehen, müssen wir uns um den OAuth-2.0-gemäßen Tausch von Autorisierungscode gegen einen Access- und ID-Token nicht kümmern, dies übernimmt Spring Security's OAuth- beziehungsweise OpenID-Support automatisch. Auch wird das Dekodieren und Validieren des ID-Tokens,

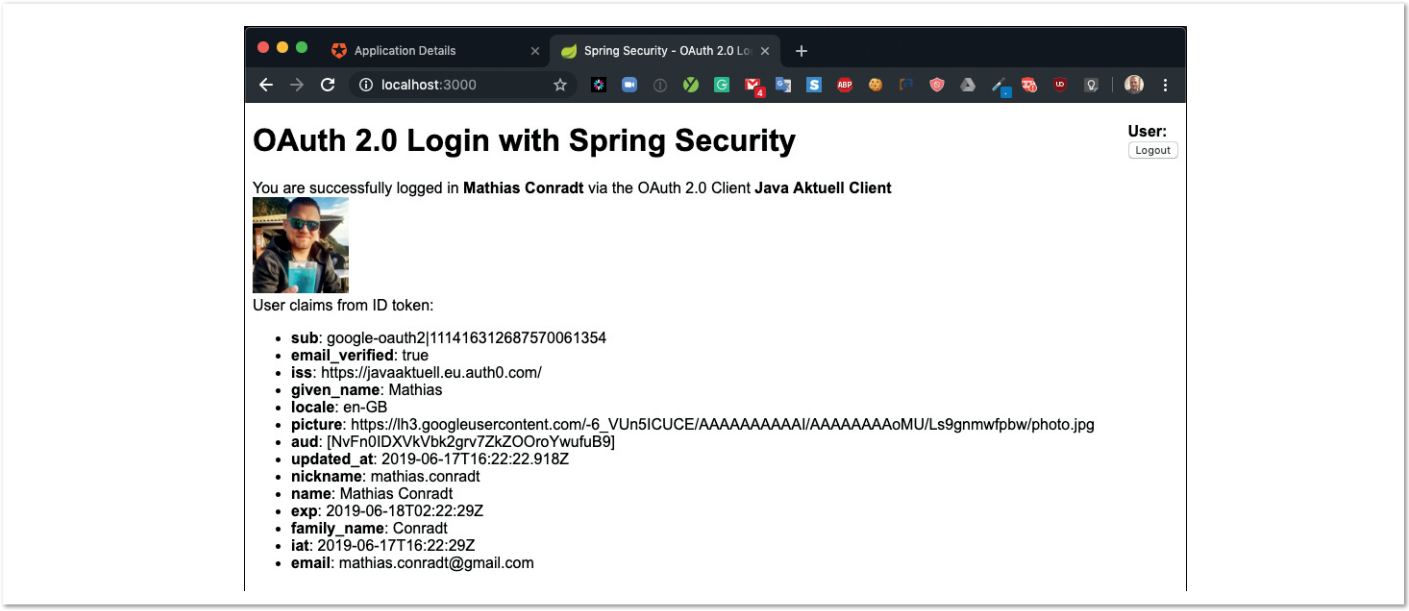

Abbildung 4: Benutzerprofil

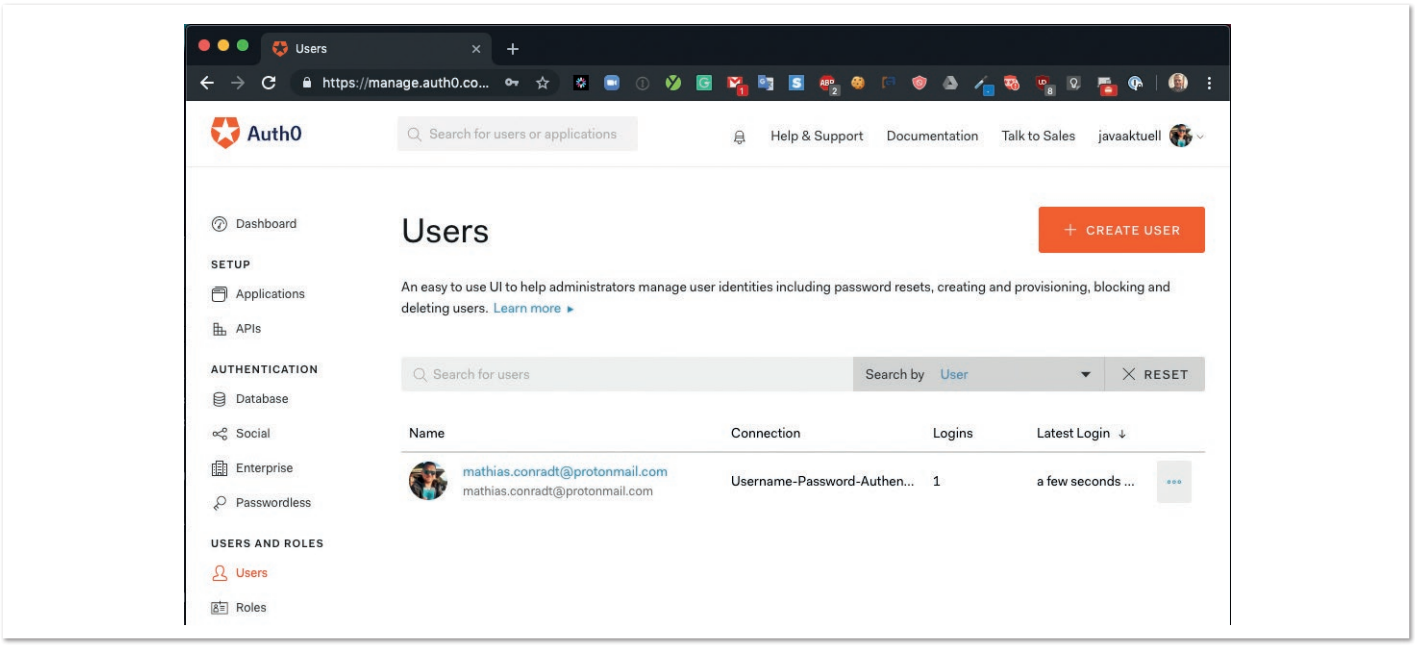

Abbildung 5: Benutzerübersicht im Auth0-Dashboard

der vom Identity-Provider als Base64Url-dekodierter und signierter *JWT (JSON Web Token)* *[7]* zurückkommt, automatisch vorgenommen und nimmt dem Entwickler bereits einiges an manueller Implementierungsarbeit ab.

Die zugehörige View, die Thymeleaf als Template Library verwendet und an die wir das Benutzerprofil über das `Model` übergeben, sieht wie in *Listing 6* gezeigt aus. Sie zeigt neben dem Namen der Client-Applikation das Avatar-Foto des Benutzers sowie dessen Profil an. Das Benutzerfoto wird seitens Auth0 automatisch vom Gravatar-Dienst auf Basis der bei der Registrierung verwendeten Benutzer-E-Mail-Adresse aufgesucht, sofern vorhanden.

Wir starten die Client-Applikation mit `mvn spring-boot:run` und rufen die URL *http://localhost:3000* im Browser auf. Da wir aktuell noch nicht authentifiziert sind, werden wir sofort zum konfigurierten Identity-Provider, Auth0, weitergeleitet und aufgefordert, uns einzuloggen beziehungsweise gegebenenfalls einen Account zu erstellen, sofern wir noch keinen haben. Außerdem erscheint ein Consent-Screen, über den wir einwilligen, dass unsere Spring-Applikation Zugriff auf unser Profil bei Auth0 erhalten darf *(siehe Abbildung 3)*.

Nach erfolgreicher Authentifizierung wird das Benutzerprofil, konkret die sogenannten „Claims" des ID-Tokens, dargestellt *(siehe Abbildung 4)*.

Bezüglich des Logout haben wir als Entwickler die Möglichkeit zu entscheiden, ob wir beim Klicken des Logout-Buttons lediglich die Session unserer Applikation beenden oder den Benutzer auch am IdP ausloggen wollen. Ist Letzteres der Fall, müssen wir einen eigenen `LogoutController` erstellen und in unserer `SecurityConfig` registrieren, der den entsprechenden Logout-Endpunkt des IdP, in unserem Fall Auth0, aufruft. Der IdP-seitige Logout-Endpunkt wird aus unserer `application.yml` ausgelesen und ist als `logout-uri` definiert: `https://javaaktuell.eu.auth0.com/v2/logout?client_id={clientID}&returnTo=http://localhost:3000` *(siehe Listing 7)*.

Andernfalls, sofern es uns ausreicht, beim Logout lediglich die aktuelle Session unserer eigenen Applikation zu beenden, nicht jedoch die des IdP, könnten wir ansonsten auf den `LogoutController` sowie die `logout()` und `addLogoutHandler()` in unserer `SecurityConfig`-Konfigurationsklasse verzichten.

Auth0 Dashboard, Social Login und MFA-Konfiguration

Wenn wir ins Auth0-Dashboard schauen, sehen wir unter dem Punkt `Users & Roles` nun auch den soeben registrierten Benutzer *(siehe Abbildung 5)*.

Um nun neben der Auth0-Datenbank weitere föderierte Social-Identity-Provider wie Google und Facebook zu integrieren sowie beispielsweise Multi-Factor-Authentication (MFA) zu aktivieren, können wir dies rein über die Konfiguration innerhalb von Auth0 vornehmen, ohne dabei unsere Java-Applikation anfassen zu müssen *(siehe Abbildungen 6 und 7)*.

```
@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.getClientRegistration().getClientName());
        model.addAttribute("userinfo", authentication.getPrincipal().getAttributes());
        return "index";
    }
}
```

Listing 5: HomeController.java

```
[...]
<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>
[...]
```

Listing 6: index.html

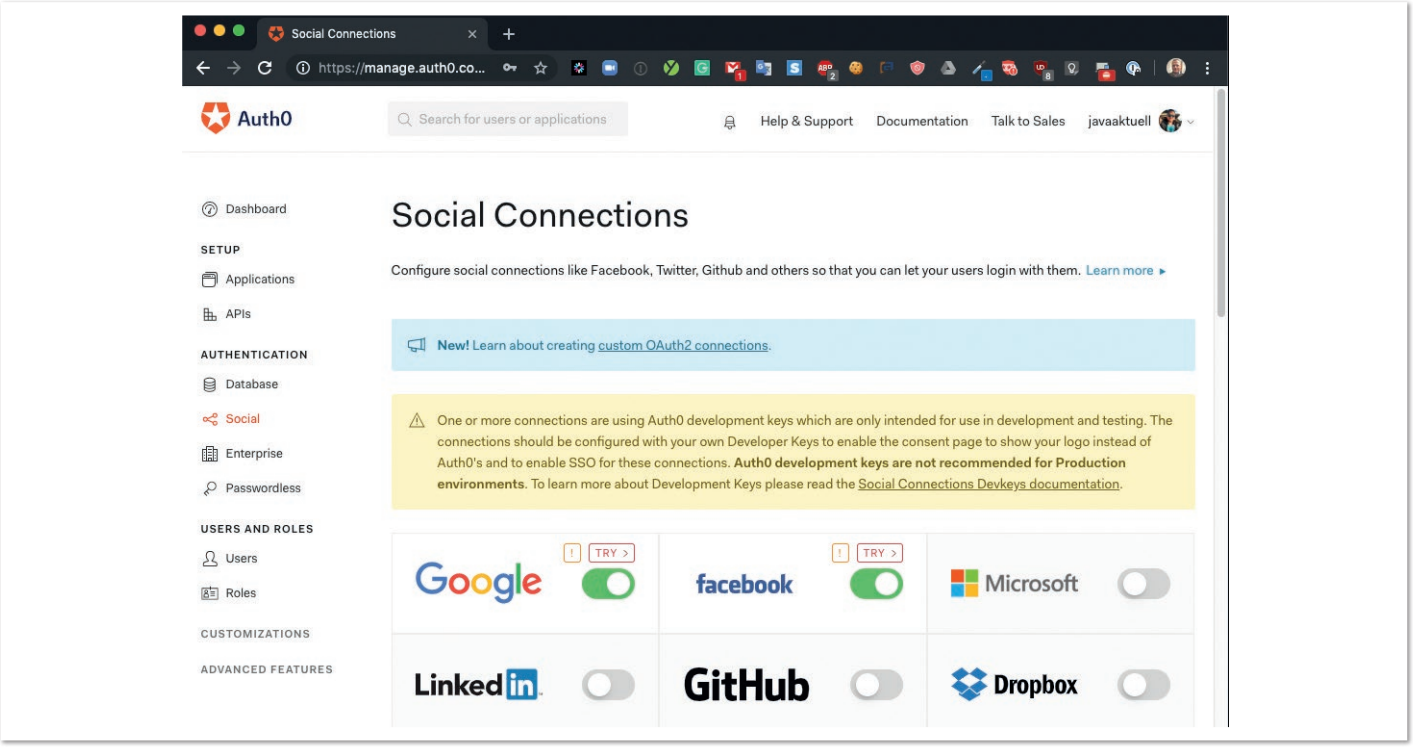

Abbildung 6: Aktivierung weiterer föderierter Social-Identity-Provider

```
@Controller
public class LogoutController extends SecurityContextLogoutHandler {

    private final ClientRegistrationRepository clientRegistrationRepository;
    private Logger logger = LoggerFactory.getLogger(LogoutController.class);

    @Value("${spring.security.oauth2.client.registration.auth0.logout-uri}")
    private String logoutUrl;

    public LogoutController(ClientRegistrationRepository clientRegistrationRepository) {
        this.clientRegistrationRepository = clientRegistrationRepository;
    }

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

Listing 7: LogoutController.java

Wenn wir uns nun aus unserer Webapplikation unter *http://localhost:3000* ausloggen und erneut einzuloggen versuchen, stellen wir fest, dass wir nun sowohl zwei weitere Buttons für Facebook und Google als Authentifizierungs-Möglichkeit haben; des Weiteren werden wir nach dem Login aufgefordert, unsere Multi-Factor-Authentication (MFA) über das Scannen eines QR-Codes zu initialisieren *(siehe Abbildung 8)*.

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 an Implementierungsarbeit ab, was die Details des OAuth2-Protokolls angeht.

Statt weitere Social-Identity-Provider direkt in der Java-Applikation zu konfigurieren, haben wir 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 minimiert doppelten Aufwand. Dasselbe gilt ebenso für die Bereitstellung von MFA. Ein netter Seiteneffekt ist, dass wir so ebenfalls ein Single Sign-on über alle Applikationen hinweg bereitstellen können.

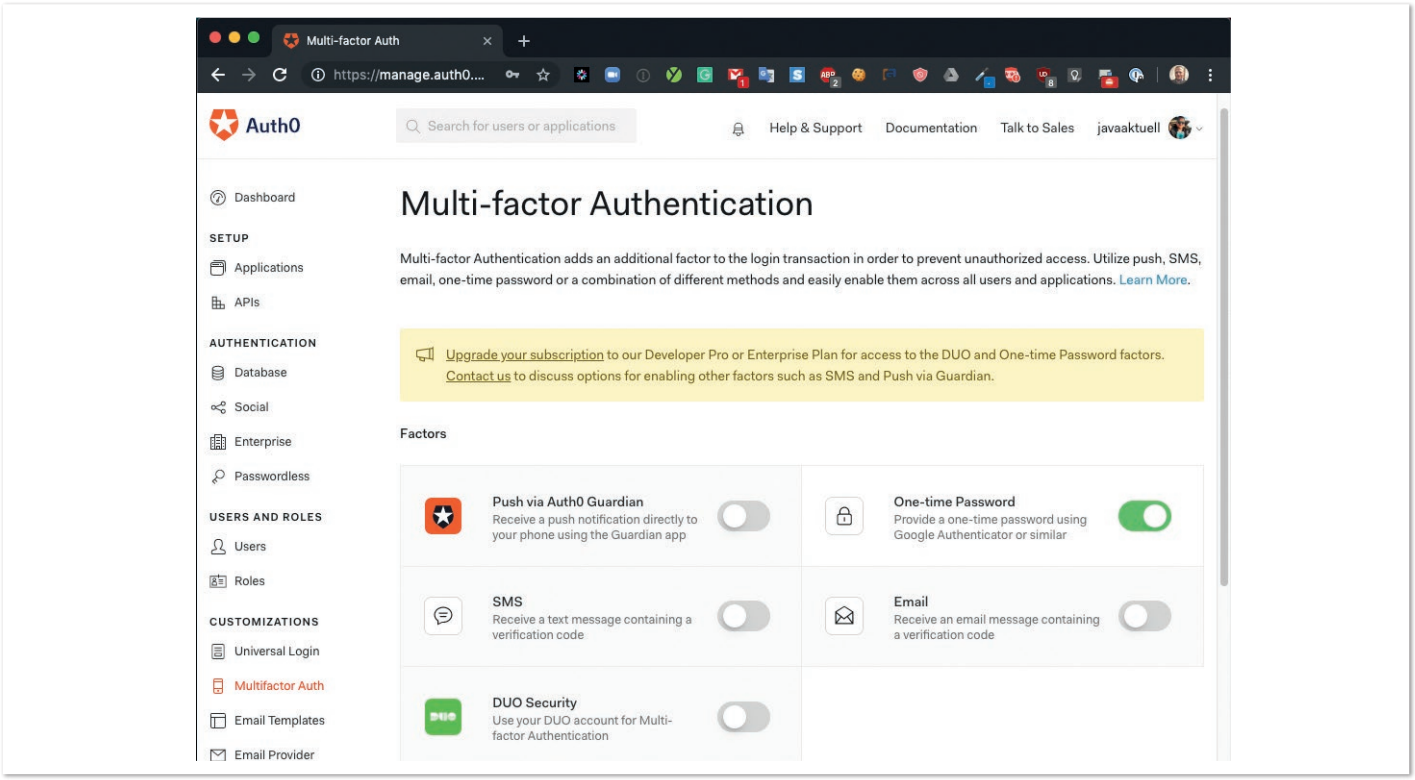

Abbildung 7: Aktivierung von MFA mittels Google Authenticator

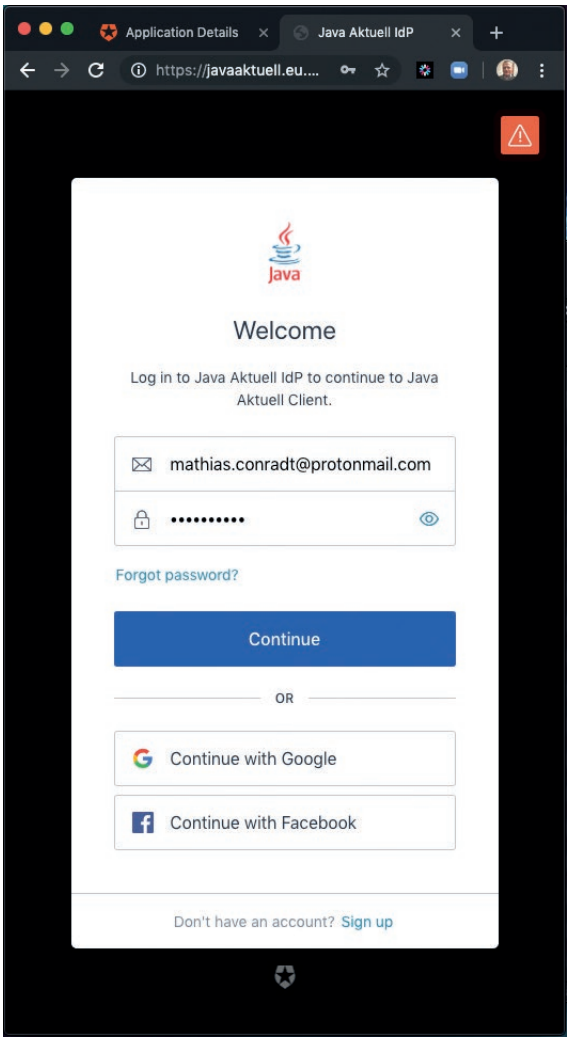

Abbildung 8: Login-Widget mit aktivierten Social Connections

Quellen

[1] *https://auth0.com/docs/flows/concepts/auth-code*

[2] *https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html*

[3] *https://github.com/mathiasconradt/java-aktuell*

[4] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html*

[5] *https://openid.net/specs/openid-connect-discovery-1_0.html*

[6] *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/oauth2/client/authentication/OAuth2AuthenticationToken.html*

[7] *https://jwt.io/*

Mathias Conradt

mathias.conradt@auth0.com

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