Automatisiertes Access Management mit Keycloak

Jannik Hüls, codecentric AG

Viele Anwendungen nutzen Drittanwendungen für die Authentifizierung und Autorisierung der Endnutzer. Die Integration dieser sicherheitskritischen Dienste ist häufig nicht automatisiert, damit unpraktisch für die Entwickler und zudem fehleranfällig. Dieser Artikel stellt den Standard der „OpenID Connect Dynamic Client Registration" vor und auf dessen Basis ein automatisiertes Access Management, beispielhaft implementiert mit dem Open-Source-Tool Keycloak und dem CI-/CD-Server Jenkins.

Die Automatisierung manueller Schritte hat in der Anwendungsentwicklung mittlerweile breite Akzeptanz gefunden. CI-/CD-Server unterstützen dabei enorm. Entwickler veröffentlichen Code-Änderungen in Versionskontroll-Systemen, CI-/CD-Server übernehmen danach viele aufwändige Tasks. Vom Bauen und Testen bis hin zur Integration und Veröffentlichung der Anwendung sind im optimalen Fall keine manuellen Interaktionen mehr notwendig. Dabei werden die Tasks aus unterschiedlichen Gründen automatisiert. Zunächst

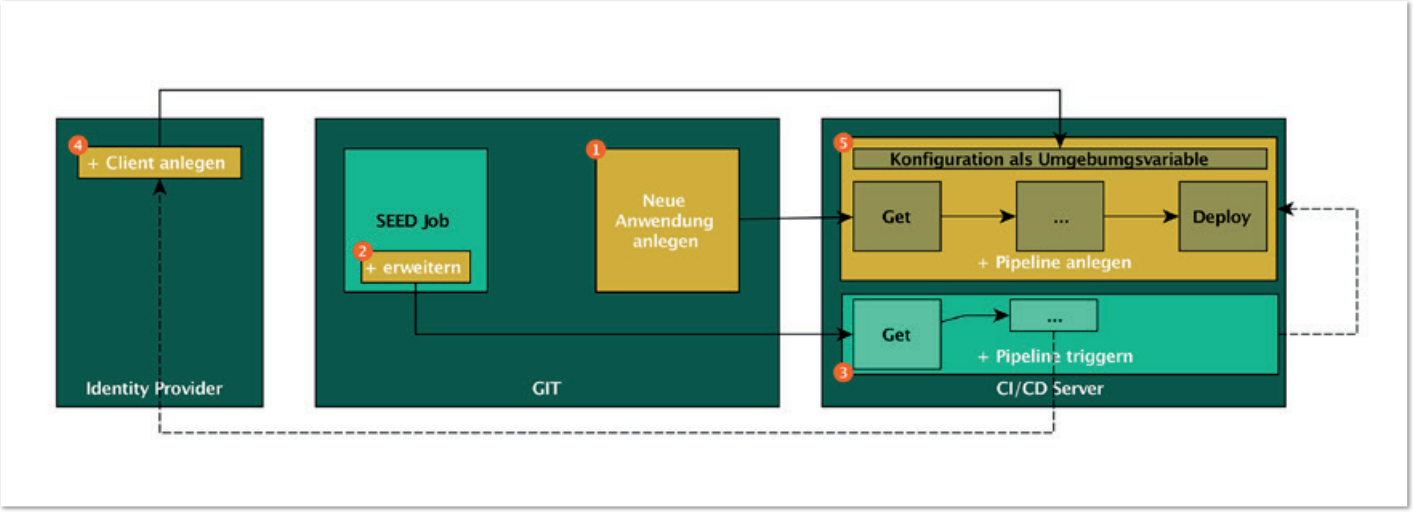

Abbildung 1: Automatische Integration einer neuen Anwendung in CI-/CD-Server

einmal soll dadurch die Geschwindigkeit erhöht werden. Es wird versucht, die Zeit zu minimieren, in der der Entwickler auf Rückmeldung wartet, ob seine Änderung erfolgreich integriert und somit veröffentlicht werden kann. Zudem sind Tasks reproduzierbar gespeichert. Automatisierungs-Artefakte enthalten möglichst alles an Expertenwissen; Tasks können somit wiederholt und ohne Abhängigkeiten zu bestimmten Verantwortlichen ausgeführt werden. Zudem sind manuelle Tasks fehleranfällig und ein hoher Automatisierungsgrad sorgt für sehr robuste Systeme.

Im CI-/CD-Server werden die einzelnen Schritte in einer sogenannten „Pipeline" ausgeführt. Diese enthält meist Zusatz-Informationen zur Ziel-Plattform der Veröffentlichung oder zu Abhängigkeiten von Drittanwendungen. Solche Abhängigkeiten treten beispielsweise auf, wenn die Authentifizierung der zu entwickelnden Anwendung gegen vorhandene Dienste erfolgen soll. Beim Einrichten der Pipeline müssen so vorab sicherheitskritische, authentifizierungsrelevante Informationen bekannt sein. Beim Anlegen eines neuen Projekts werden dabei zwar häufig die einzelnen Schritte der Pipeline automatisiert, die Erstellung der Pipeline ist allerdings meist noch ein manueller Task. Vor allem die Anbindung an Dienste wie die Authentifizierung erschwert die automatisierte Erstellung der Pipeline. Das ist ein Problem, da genau diese extrem sicherheitskritischen Tasks aus den oben genannten Gründen nicht immer wieder von den Entwicklern manuell ausgeführt werden sollten. Nachfolgend wird betrachtet, wie die Nutzung von Authentifizierungsdiensten automatisiert und in die Pipeline integriert werden kann. *Abbildung 1* zeigt dabei das gewünschte Ziel-Szenario.

Eine neue Anwendung wird im Versionskontroll-System angelegt (1). Die Erstellung der CI-/CD-Pipeline erfolgt dabei nicht manuell, sondern durch einen sogenannten „Seed-Job". Dies ist eine Pipeline, die ausschließlich für die Erstellung anderer Pipelines zuständig ist. Dieser Seed-Job wird erweitert (2), die entsprechende Pipeline des Jobs gestartet (3) und auf diese Weise vollautomatisch eine neue Pipeline erzeugt (5). Abhängigkeiten zu sicherheitskritischen Diensten wie dem Identity-Provider werden im Seed-Job programmatisch gelöst (4) und alle notwendigen Informationen stehen automatisch in der neuen Pipeline zur Verfügung. Essenziell bei diesem Vorgehen ist der Standard „OpenID Connect Dynamic Client Registration".

OpenID Connect Dynamic Client Registration

Aus Gründen der Sicherheit, der einfachen Anbindung sowie der gewünschten Unabhängigkeit soll in der neu zu entwickelnden Web-Anwendung für die notwendige Authentifizierung das Standardprotokoll „OpenID Connect" eingesetzt werden. Es handelt sich um eine Identitätsschicht, aufbauend auf dem Protokoll „OAuth 2.0". Mit OpenID Connect lässt sich sowohl die Frage der Authentifizierung („Wer ist der Benutzer?") als auch die Frage der Autorisierung („Was ist dem Benutzer erlaubt zu tun?") klären.

Die Web-Anwendung ist eine „Relying Party" im „OpenID Connect"-Terminus. Jede Authentifizierung erfolgt im Sinne des Endbenutzers, des „Resource Owner". Dies bedeutet, die Relying Party verweist zur Authentifizierung im Standardfall an den Identity-Provider. Der Resource Owner meldet sich dort an und wird nach erfolgreicher Anmeldung erneut zur „Relying Party" umgeleitet.

Benutzerspezifische Informationen werden im Anschluss als JSON Web Token (JWT) enkodiert an die Relying Party, und somit die Web-Anwendung, übermittelt. Diese verifiziert damit die Identität des Resource Owner, kann mit den erhaltenen Token Zugriffsrechte prüfen und auf Basis dieser Informationen Inhalte anzeigen.

Jede Relying Party kommuniziert also zur Authentifizierung mit dem Identity-Provider. Diese Kommunikations-Schnittstelle wird im Iden-

```
POST /connect/register HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: server.example.com

{
  "redirect_uris":
    [
      "https://client.example.org/callback",
      "https://client.example.org/test"
    ],
  "client_name": "My Example",
  "application_type": "web"

}
```

Abbildung 2: HTTP-Post-Nachricht, um einen neuen Client zu registrieren

tity-Provider „Client" genannt. Für jede Relying Party muss, wenn sie einen Identity-Provider verwenden möchte, demnach in genau diesem ein entsprechender Client registriert und konfiguriert werden.

Die Registrierung des Clients kann manuell erfolgen, beispielsweise über eine webbasierte Administrationsoberfläche des Identity-Providers. Dazu ist zur Anlage des Clients der Zugriff auf den Identity-Provider notwendig. Die Konfiguration der Relying Party erfolgt dann mit dem Identity-Provider-URL, der „Client ID" sowie einem „Client Secret".

Diese manuellen Schritte widersprechen der gewünschten Automatisierung. Beim Anlegen eines neuen Entwicklungsprojekts sind diese Konfigurationen wiederkehrend durchzuführen, was fehleranfällig und zeitraubend ist. Zudem ist Expertenwissen und -unterstützung notwendig. Die Authentifizierung kann erst dann implementiert werden, wenn ein Client im Identity-Provider angelegt wurde oder wenn der Entwickler Zugriff auf den Identity-Provider bekommt, um dies im Self-Service zu erledigen. Beides sind häufig keine gewünschten Szenarien, da solche Abhängigkeiten Zeit rauben und umfangreiche Zugriffe auf extrem sicherheitsrelevante Systeme meist unerwünscht sind.

„OpenID Dynamic Client Registration" ist ein Standard, der es einer Relying Party erlaubt, sich dynamisch beim Identity-Provider zu registrieren. Die Relying Party nutzt dabei ein RESTful-HTTP-API, um die notwendigen Informationen eines Clients zu übermitteln und alle Konfigurationsdaten für die lokale Anwendung zu erhalten – ideal also für die angestrebte Automatisierung und die Integration in die Pipeline. Es wird zwischen einem Registrierungs- und einem Konfigurations-Endpunkt unterschieden. Neue Clients können am Registrierungs-Endpunkt dynamisch angelegt und die Einstellungen existierender Clients am Konfigurations-Endpunkt im Self-Service geändert werden. Um einen neuen Client zu registrieren, wird eine HTTP-Post-Nachricht an den Identity-Provider gesendet. *Abbildung 2* zeigt beispielhaft eine solche Nachricht.

Die mitgesendeten Client-Metadaten konfigurieren den Client im Identity-Provider. Lediglich der Parameter „redirect_uris", also die erlaubten Umleitungen nach erfolgreicher Authentifizierung, ist dabei Pflicht. Der Identity-Provider antwortet bei erfolgreicher Registrierung dann mit einer Client-ID, einem Client Secret sowie allen erstellten Client-Metadaten *(siehe Abbildung 3)*.

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache
Host: server.example.com

{
  "client_id": "s6BhdRkqt3",
  "client_secret": "ZJYCqe3GGRvdrudKyZS0XhGv_Z45DuKhCUk0gk",
  "registration_access_tkoen": "tokenstring",

  ...

  "redirect_uris":
    [
      "https://client.example.org/callback",
      "https://client.example.org/test"
    ],
  "client_name":"My Example"
  "application_type":"web",

  ...

}
```

Abbildung 3: HTTP-Response nach erfolgreicher Registrierung eines Clients

Client-ID und Secret werden in der Applikation verwendet, um die Authentifizierung mit dem Identity-Provider durchführen zu können. Das „Registration Access Token" ist zudem von Interesse, da es für die Authentifizierung bei Anfragen an den Konfigurationsendpunkt verwendet werden muss. Auffällig ist, dass keine Authentifizierung bei der Erstellung eines Clients notwendig ist. Die Spezifikation sieht explizit vor, dass auch Anfragen ohne Authentifizierung möglich sein sollen. Dem Identity-Provider obliegt es dann, diese durch gesonderte Mechanismen wie Beschränkungen in der maximalen Anzahl von Anfragen und vertraute Domänen abzusichern.

Mit der Dynamic Client Registration bietet OpenID Connect einen Standard, der für die Nutzung in der Automatisierungspipeline wie gemacht ist. Am Beispiel des Identity-Providers Keycloak und des CI-/CD-Servers Jenkins wird nun gezeigt, wie eine neue Pipeline erstellt und die entsprechende Anwendung automatisch in Keycloak registriert wird.

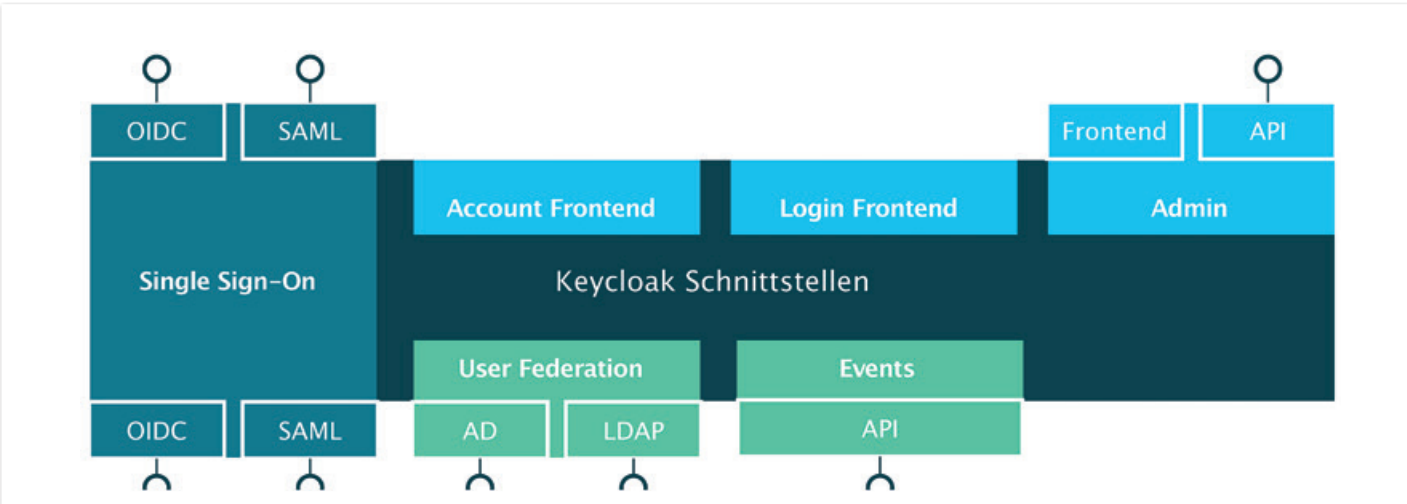

Abbildung 4: Keycloak-Schnittstellen für Anwendung und Administration

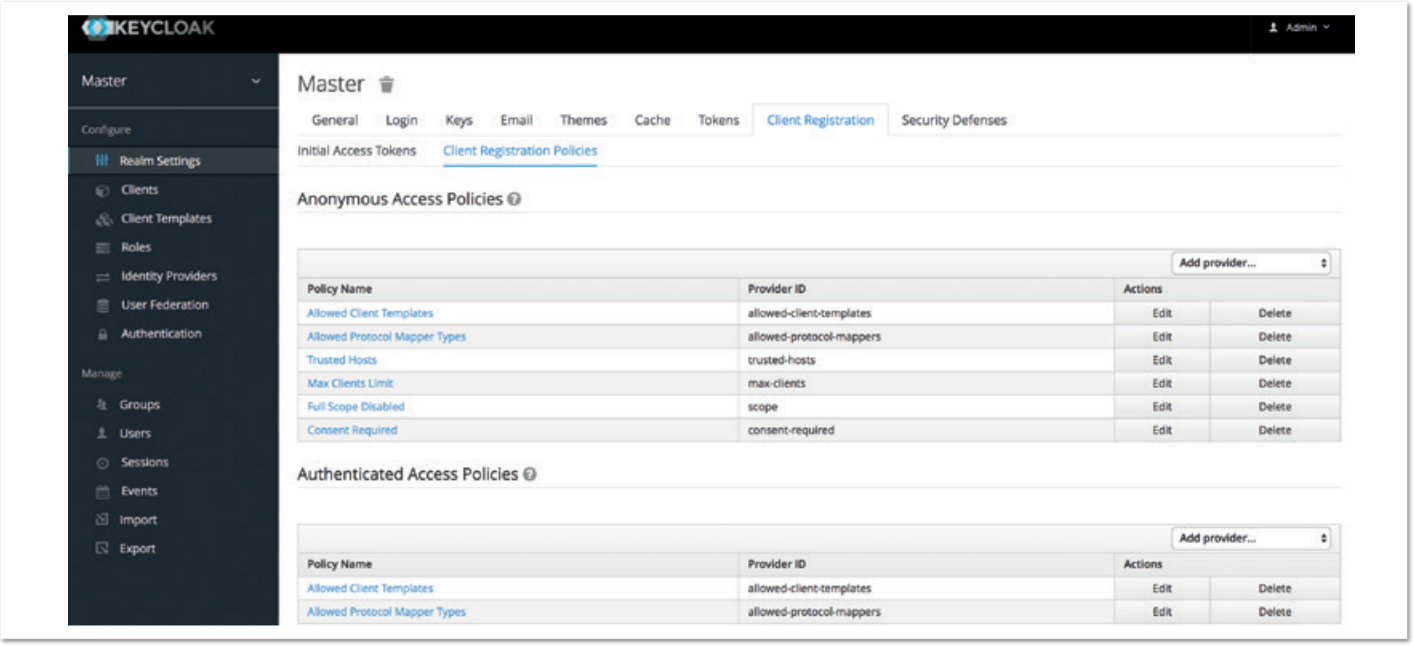

Abbildung 5: Keycloak-Client-Registrierungs-Policies

Dynamische Client-Registrierung in der Praxis

Keycloak *(siehe „http://www.keycloak.org")* ist ein Open-Source-Identity-Provider aus dem Hause Red Hat. Ehrlicherweise nur einer von vielen Identity-Providern, mit dem der Autor bereits in vielen Projekten gute Erfahrungen machen konnte. Keycloak bedient sich sehr ausgiebig am Red-Hat-Stack. Standardmäßig wird WildFly als Application-Server eingesetzt und dessen Clustering- und Hochverfügbarkeits-Funktionen genutzt. Viele Adapter, unter anderem für Tomcat, Spring Boot oder Spring Security, machen die Integration für Entwickler extrem komfortabel. Keycloak setzt auf Standards für

```
1 : def kcRealm = <realmstring>
2 : def kcUrl = <identity provider url>
3 : def kcDcrUrl = kcUrl + "/realms/" + kcRealm + "/clients-registrations/openid-connect"
4 : def registerClient(String urlString, String queryString) {
5 :  def url = new URL(urlString)
6 :  def connection = url.openConnection()
7 :  connection.setRequestMethod("POST")
8 :  connection.doOutput = true
9 :  connection.setRequestProperty("Content-Type", "application/json")
10 :  def writer = new OutputStreamWriter(connection.outputStream)
11 :  writer.write(queryString)
12 :  writer.flush()
13 :  writer.close()
14 :  connection.connect()
15 :  new groovy.json.JsonSlurper().parseText(connection.content.text)
16 : }
17 : job('Webanwendung mit Authentifizierung') {
18 :  def metadata = '{"redirect_uris":["https://client.example.org/callback","https://client.example.org/
callback2"],"client_name": "My Example"}'
19 :  def response = registerClient(kcDcrUrl, metadata)
20 :  environmentVariables {
21 :    env('CLIENT_ID', response.client_id)
22 :    env('REALM', kcRealm)
23 :    env('IDENTITY_PROVIDER', kcUrl)
24 :    env('CLIENT_SECRET', response.client_secret)
25 :    env('REGISTRATION_ACCESS_TOKEN', response.registration_access_token)
26 :   }
27 :  scm {
28 :    git ('https://github.com/jannikhuels/kc-jenkins-dcr')
29 :  }
30 :  triggers {
31 :   scm('*/15 * * * *')
32 :  }
33 :  steps {
34 :    maven("build")
35 :   …
36 :    maven("package")
37 : }
38 : }
```

Listing 1

die Anbindung von Identity-Providern sowie User Federation und ist dabei sehr breit aufgestellt. Neben OpenID Connect und SAML werden standardmäßig auch LDAP und Active Directory unterstützt. Zudem ist Keycloak auf Erweiterbarkeit ausgelegt. Sogenannte „Service Provider Interfaces" (SPIs) sind für alle wesentlichen Funktionalitäten von Keycloak implementiert und schaffen der Erweiterbarkeit extrem großzügige und komfortable Möglichkeiten.

Abbildung 4 zeigt die Schnittstellen von Keycloak. Nicht nur die Automatisierung der Client-Registrierung lässt sich über die standardisierten OIDC-Schnittstellen durchführen. Das RESTful-Admin-API ermöglicht noch viele weitere Möglichkeiten. Von einer automatischen Konfiguration neuer Keycloak-Instanzen bis hin zu einer Self-Service-User-Registrierung werden damit sehr viele Möglichkeiten geboten. Die automatische Client-Registrierung ist für die Anwendungsentwicklung allerdings der Startpunkt; im Folgenden wird erläutert, wie diese in Keycloak verwendet werden kann.

Keycloak bietet vier verschiedene Provider für die Client-Registrierung: (1) default, (2) install, (3) openid-connect und (4) saml2-entity-descriptor. Die ersten beiden implementieren eine Keycloak-spezifische Client-Beschreibung in JSON. Provider drei und vier implementieren die jeweilige standardkonforme Client-Registrierung. An allen vier Schnittstellen können sowohl ein Bearer- als auch ein Initial-Access-Token für die Authentifizierung verwendet werden. Zudem ist eine Registrierung ohne vorherige Authentifizierung möglich. Sicherheit wird dabei auf Basis von konfigurierbaren Richtlinien hergestellt. Diese sogenannten „Policies" sind initial schon sehr umfangreich und zudem über das entsprechende Serial Peripheral Interface (SPI) ierweiterbar. *Abbildung 5* zeigt das User-Interface für die Konfiguration der dynamischen Client-Registrierung in Keycloak.

Keycloak implementiert die OIDC Dynamic Client Registration konform mit dem Standard. Um die Unabhängigkeit zu bewahren, soll ein Standardprotokoll eingesetzt und daher in der Pipeline diese Schnittstelle verwendet werden. Dabei wird die Policy-basierte Authentifizierung genutzt.

Für unsere Web-Anwendung soll nun eine Pipeline entstehen. Dies muss für Anwendungsentwickler möglichst komfortabel sein und daher weitestgehend automatisiert ablaufen. Als CI-/CD-Server wird Jenkins eingesetzt. Die Erstellung der Pipeline erfolgt in Jenkins über einen Seed-Job. *Listing 1* zeigt das Groovy-Script des entsprechenden Jobs.

In den Zeilen 1 bis 3 werden die Identity-Provider-spezifischen Variablen gesetzt. Dies muss nicht notwendigerweise Keycloak sein, ist es jedoch in diesem Beispiel. Bei abweichenden Identity-Providern würde sich die URL der Zeile 3 ändern. Die Funktion „registerClient" (Zeilen 4 bis 16) setzt einen HTTP-Post-Request zur Registrierung eines neuen Clients ab. Dieser entspricht dem in *Abbildung 2* dargestellten Request. Dieser Seed-Job erstellt genau eine Pipeline. Die anwendungsspezifischen Metadaten (Zeile 18) beschreiben die Konfiguration des Clients. Der Client wird einmalig im Job angelegt (Zeile 19), die Daten der Antwort (analog zu *Abbildung 3)* werden als Umgebungsvariablen der Pipeline gespeichert (Zeilen 20 bis 26). Diese Umgebungsvariablen können dann in den einzelnen Tasks der Pipeline verwendet werden.

Die Tasks der Pipeline sind hier nicht von Relevanz, wurden aber der Vollständigkeit halber hinzugefügt. In der Pipeline wird ein Git-Repository alle 15 Minuten gepollt und auf Änderungen überprüft. Die Web-Anwendung ist ein Java-Projekt, folglich entsprechen die Steps der Pipeline „maven"-Tasks (Zeilen 33 bis 37). Bei Ausführung dieses Jobs wird demnach im Jenkins eine Pipeline erzeugt, in der alle Konfigurationsdaten des Identity-Providers hinterlegt sind. Diese können in der Anwendung dann für die Konfiguration der Authentifizierung verwendet werden.

Fazit

Das Tool Keycloak bietet durch den modularen Aufbau und die vielen Schnittstellen umfangreiche Automatisierungsmöglichkeiten. In diesem Post wurde beispielhaft die initiale Registrierung neuer Anwendungen auf Basis der „OpenID Connect Client Registration" betrachtet. Dies ist der erste Schritt für eine sehr komfortable Erstellung neuer Anwendungs-Pipelines, aber nur einer von wenigen zu einer vollständigen Self-Service-Implementierung. Viele Dinge lassen sich in der Implementierung deutlich optimieren. Beispielsweise sollte eine sichere Verwaltung der Token implementiert werden. Für die Idee dieses Artikels ist dies nicht wichtig, für eine produktionsreife Implementierung allerdings umso mehr. Dieser Beitrag soll dazu anregen, vor allem komplexe und sicherheitsrelevante Integrationen zu automatisieren und die notwendigen manuellen Schritte auf ein Minimum zu reduzieren.

Jannik Hüls
jannik.huels@codecentric.de

Jannik Hüls unterstützt seit dem Jahr 2016 das codecentric Team in Münster. Er beschäftigt sich mit der technischen Umsetzung von IT-Architekturen und der Integration von Anwendungskomponenten, aktuell vor allem mit den Möglichkeiten der Serverless-Architektur. Zudem berät er Kunden im Identity- und Access-Management.