

Privacy by Design

| | |
|----------------------|---|
| <i>Firma</i> | Stiwa |
| <i>Partner</i> | Software Competence Center Hagenberg GmbH |
| <i>Dokument Name</i> | Privacy by Design |
| <i>Erstellt von</i> | Benjamin Mayer |
| <i>Version</i> | 2.0 |

| | | |
|----------|---|-----------|
| 1 | <i>Einleitung</i> | 3 |
| 1.1 | Grundlagen | 3 |
| 1.2 | Anforderungen | 4 |
| 2 | <i>Beispiel Scenario</i> | 4 |
| 2.1 | Services | 4 |
| 2.2 | Rollen und Berechtigungen | 5 |
| 2.3 | Architektur | 6 |
| 3 | <i>Rechte- und Benutzerverwaltung (Keycloak)</i> | 7 |
| 3.1 | OAuth2 Clients | 8 |
| 3.2 | Benutzer und Rollen in Keycloak | 8 |
| 3.3 | Dynamische Rechteverwaltung | 9 |
| 3.4 | Custom Login Theme | 10 |
| 4 | <i>Authentication</i> | 10 |
| 4.1 | Spring Authentication | 10 |
| 4.1.1 | Spring Service Authentication mit Keycloak Adapter | 11 |
| 4.1.2 | Spring Service Authentication mit Spring Cloud Security | 12 |
| 4.1.3 | Service zu Service Kommunikation | 14 |
| 4.2 | JavaScript Client Authentication (mit Keycloak Adapter) | 19 |
| 5 | <i>Authorization (Access Control)</i> | 21 |
| 5.1 | Web Security | 21 |
| 5.2 | Method-Level Security | 21 |
| 5.3 | Field-Level Security | 23 |
| 6 | <i>Verschlüsseln von Konfigurationen</i> | 26 |
| 7 | <i>Installationsanleitung</i> | 27 |
| 8 | <i>Verwendungsbeispiel</i> | 28 |
| 8.1 | Method-Level Security | 28 |
| 8.1.1 | Unautorisierter Zugriff | 28 |
| 8.1.2 | Erlaubter Zugriff | 28 |
| 8.1.3 | Verbotener Zugriff | 29 |
| 8.2 | Field-Level Security | 29 |
| 8.3 | Service zu Service Kommunikation | 30 |

1 Einleitung

In diesem Projekt wurde ein Technology Stack zur Absicherung von Microservices geschaffen. Die zu sichernden Microservices benutzen das Spring Framework. In diesem Dokument wird der geschaffene Technology Stack anhand eines Beispiel-Szenarios beschrieben. Der Erklärung des Technology Stacks erfolgt Großteils anhand von Code Beispielen. Das Beispiel-Szenario ist in folgendem Github-Repository zu finden: <https://github.com/benni333/securityexamples>. Begleitdokumente sind im Ordner „documents“ des Github-Repository ersichtlich.

1.1 Grundlagen

(Software) Security: „Engineering software so that it continues to function correctly under malicious attack“

Privacy: „The ability to protect sensitive information about personally identifiable information“

| Security vs Privacy | | |
|---------------------|--|---|
| | Security | Privacy |
| Definition | Protection of people, assets and information from harm or abuse. | Freedom from being observed or disturbed. |
| Relationship | Security defends privacy | Privacy requires security |

Abbildung 1 Security vs Privacy

„**Privacy by design** calls for privacy to be taken into account throughout the whole engineering process.“

“**Authentication** is act of confirming the truth of an attribute of a single piece of data claimed true by an entity“. Who are you?

Authorization (Access Control): “Is the function of specifying access rights/privileges to resources“. What are you allowed to do?

OAuth2 ist ein offener Standard zur Token-basierten Authentifizierung und Autorisierung. Der Standard erlaubt es, Zugriff auf Ressourcen zu gewähren ohne, dass User-Credentials an Client-Applikationen preisgegeben werden.

OpenID Connect ist eine Authentifizierungsschicht, die auf dem Autorisierungsprotokoll

OAuth2 basiert. Diese Schicht erlaubt es Client-Applikationen, die Identität eines Anwenders zu überprüfen und grundlegende Informationen über den Anwender zu erhalten.

1.2 Anforderungen

In diesem Abschnitt werden die Anforderungen an die Security-Schicht beschrieben. Die Authentifizierung an Services soll über einen zentralen Authentifizierungsserver mittels OAuth2 erfolgen. Die Zugriffskontrolle soll auf Basis von Rollen durchgeführt werden. Es ist darauf zu achten, dass beim Kunden Rollen mit unterschiedlichen Berechtigungen ausgestattet werden können. Die Berechtigungen die mit einer Rolle verbunden sind, sollen daher beim Kunden individuell konfiguriert werden können. Die Befugnisse einer Rolle sind ausschlaggebend welche Methoden (oder Endpunkte) aufgerufen werden können (siehe Abschnitt 5.2). Jede Rolle soll nur jene Aktionen durchführen können, die zur Auftragsabwicklung notwendig ist. Weiters soll auf Basis von Berechtigungen oder Rollen die Sicht auf Domänenobjekte eingeschränkt werden können. Benutzer des Systems sollen nur jene Attribute sehen, die für die Auftragsabwicklung notwendig sind (siehe Abschnitt 5.3). Es soll z.B. nur für Benutzer einer bestimmten Rolle möglich sein, einen Datensatz zu einer Person in Verbindung zu bringen. Ein weiterer Punkt der umgesetzt werden soll, ist die Verschlüsselung von Konfigurationen, sodass sensible Attribute des Konfigurationsmanagements (z.B. Zugangsdaten zur Datenbank) verschlüsselt abgelegt werden können.

2 Beispiel Szenario

Zur Umsetzung der im vorherigen Abschnitt beschriebenen Anforderungen wurde eine Technology-Stack definiert. Dieser Technology-Stack wurde in einem Beispiel-Szenario angewendet. Das Ergebnis ist über ein Github-Repository zugänglich (<https://github.com/benni333/securityexamples>). In diesem Abschnitt wird das Beispiel Szenario beschrieben.

2.1 Services

Das Beispiel anhand welchem der Security-Stack gezeigt werden soll besteht aus drei funktionalen Services:

- **LaboratoryService:** Ist ein Service der die Ressource *LaboratoryResult* verwaltet. Dieser Service wurde mit dem Spring Framework entwickelt und bietet zwei HTTP-GET-Endpunkte an („api/laboratory-results“ → zum Laden aller Laborergebnisse, „api/laboratory-results/{id}“ → zum Laden eines Laborergebnis mit bestimmter ID). Dieser Service läuft standardmäßig auf Port 8080. Dieser Service ist im Github-Repository im Ordner "laboratoryservice" verfügbar.
- **PatientService:** Ist ein Service der die Ressource Patient verwaltet. Dieser Service wurde mit dem Spring Framework entwickelt und bietet zwei HTTP-GET-Endpunkte an („api/patients“ → zum Laden aller Patient, „api/patients/{svnr}“ → zum Laden eines Patienten mit bestimmter SVNR). Dieser Service läuft standardmäßig auf Port 8081. Dieser Service ist im Github-Repository im Ordner "patientservice" verfügbar.

- **Webclient:** Ist eine JavaScript Single-Page-Applikation, welche Laborergebnisse und Patienten anzeigt. Der Webclient wurde mit Angular entwickelt und läuft in einem NodeJs Server auf Port 4200. Dieser Service ist im Github-Repository im Ordner "webclient" verfügbar.

Neben den funktionalen Services wurden auch drei Infrastruktur-Services entwickelt:

- **Keycloak:** Ist der zentrale Authentifizierungspunkt des Beispiel-Szenarios. Die Authentifizierung aller anderen Services erfolgt über diesen Service. Keycloak wurde im Beispiel-Szenario so konfiguriert, dass er auf Port 8180 läuft.
- **Config-Server:** Ist der Service, der Konfigurationen für alle springbasierten Services bereitstellt. Dieser Service läuft auf Port 8888 und ist im Github-Repository im Ordner "config" verfügbar. Dieser Service benutzt Konfigurationsattribute, welche ebenfalls im Github-Repository im Ordner "application-config" abgelegt sind.
- **Eureka:** Ist die zentrale Service-Discovery, die gebraucht wird um die anderen Services zu finden. Dieser Service ist spring-basiert und befindet sich im Github-Repository im Ordner „eureka“.

2.2 Rollen und Berechtigungen

Die Zugriffskontrolle soll primär auf Basis von Rollen durchgeführt werden. Im Beispiel-Szenario wurden daher die folgenden Rollen definiert:

- *ROLE_ADMIN*
- *ROLE_DOCTOR*
- *ROLE_ASSISTENT*
- *ROLE_SECRETARY*
- *ROLE_SYSTEM* (diese Rolle wird nicht von Menschen, sondern von Services eingenommen)

Diese Rollen sind ausschlaggebend für die folgenden Befugnisse:

- *READ_LABORATORY_RESULTS:* Diese Befugnis ist notwendig, um die Endpunkte/Methoden des *LaboratorService* aufrufen zu können
- *READ_EXTENDED_LABORATORY_RESULTS:* Das Domänenobjekt Laborergebnis hat sensible Attribute. Diese können nur von Benutzern mit dieser Befugnis gelesen werden.
- *READ_PATIENTS:* Diese Befugnis ist notwendig, um die Endpunkte/Methoden des *PatientService* aufrufen zu können.
- *READ_SENSITIVE_PATIENT_DATA:* Das Domänenobjekt Patient hat sensible Attribute. Diese können nur von Benutzern mit dieser Befugnis gelesen werden.
- *ACCESS_EUREKA:* Diese Befugnis wird benötigt um Endpunkte der Service Discovery aufrufen zu können.
- *ACCESS_CONFIG:* Diese Befugnis wird benötigt um Endpunkte des Konfigurationsservice aufrufen zu können.

Die folgende Tabelle zeigt, welche Befugnisse welchen Rollen zur Verfügung stehen:

| | Admin | Doctor | Assistent | Secretary | System |
|-------------------------------------|--------------|---------------|------------------|------------------|---------------|
| Read Laboratory Results | X | X | X | | |
| Read Ext. Laboratory Results | X | X | | | |
| Read Patients | X | X | X | X | |
| Read Sensitive Patient Data | X | X | | | |
| Access Eureka | X | | | | X |
| Access Config | X | | | | X |

Tabelle 1 Rollen und Berechtigungen

2.3 Architektur

Da die Komplexität in den einzelnen Services gering ist, wird auf das Design der Services nicht näher eingegangen. Damit die Funktionalität der Sicherheitsschicht nicht mehrfach implementiert werden muss, benutzen alle Spring Services (*LaboratoryService*, *PatientService*, *Eureka*, *Config-Server*) eine „Securitylibrary“. Diese wurde auf zwei unterschiedliche Arten realisiert. Beide Arten können über das *pom*-File als Abhängigkeit hinzugefügt werden. Die erste Art (= „securitylibrary“ im Github-Repository) benutzt zur Authentifizierung das von Keycloak zur Verfügung gestellte Framework. Die zweite Art (= „cloudsecuritylibrary“) benutzt die OAuth2 Authentifizierung von Spring („spring-cloud-security-oauth2“), um Abhängigkeiten zu Keycloak zu vermeiden.

Aktuell wird im Master-Branch die Keycloak-unabhängige Library benutzt, weil sie dem Spring-Standard entspricht und die Kommunikation zwischen Services einfacher konfiguriert werden kann. Der Wechsel zur alternativen Security-Library ist relativ einfach. Man passt das *pom*-File an, indem die alternative Abhängigkeit angegeben wird und man wechselt das aktive Spring-Profil von *cloudsecurity* zu *keycloak* um vom Konfigurationsserver die entsprechenden Konfigurationen zu laden (am Konfigurationsserver sind für beide Profile die Konfigurationen abgelegt). Alternativ dazu, kann man auch vom Master-Branch in den Keycloak-Branch wechseln, welcher die Keycloak-spezifische Library benutzt.

Abbildung 2 zeigt die Beziehungen zwischen den Services. Alle Services des Beispiel-Szenarios benutzen die zentrale Keycloak-Instanz zur Authentifizierung der Benutzer. Funktionale Abhängigkeiten bestehen zwischen dem *PatientService* und dem *LaboratoryService*, sowie dem Webclient zu den REST-Services. Der *PatientService* benutzt den *LaboratoryService* um alle Laborergebnisse eines Patienten abzufragen. Diese Beziehung wurde eingeführt um zu zeigen wie ein eingehender OAuth-Token bei einer Service-zu-Service-Kommunikation übergeben werden kann (=Token Relay).

Alle Spring-Services erhalten Konfigurationen vom Konfigurationsserver und registrieren sich an der Service-Discovery. Beim Startup eines funktionalen Service (*LaboratoryService*, *PatientService*) wird der Standort des Konfigurationsservers über die Service-Discovery ermittelt. Da auch die Infrastruktur-Services (*Eureka*, *Config-Server*) mit OAuth2 abgesichert sind, müssen Services bei Startup einen OAuth-Token beim Authentifizierungsserver beziehen (siehe Abschnitt 4.1.3).

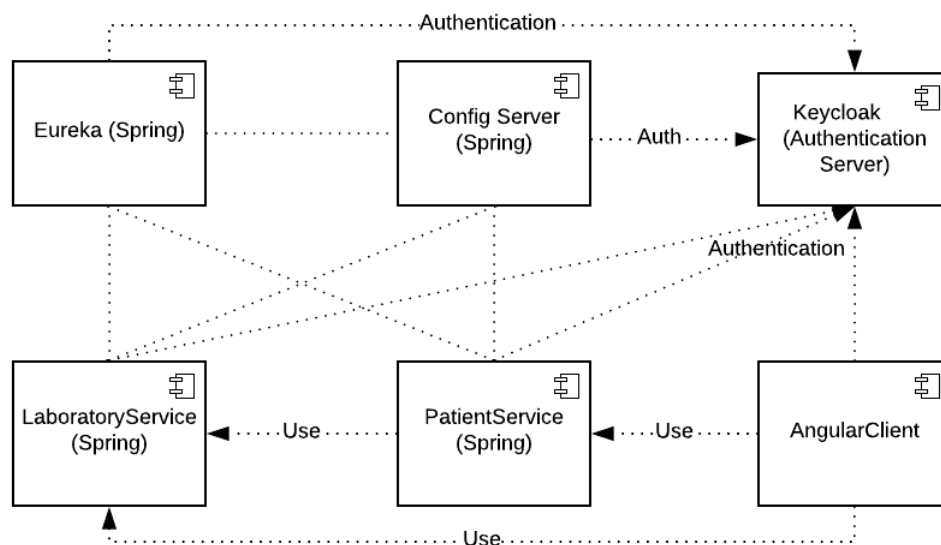


Abbildung 2 Service Beziehungen

3 Rechte- und Benutzerverwaltung (Keycloak)

Im Beispiel-Szenario wird zur Verwaltung der Recht und Benutzer Keycloak verwendet. Keycloak ist ein Identitäts- und Zugriffsmanagementsystem. In diesem Kapitel wird speziell darauf eingegangen wie Keycloak für das Beispiel-Szenario konfiguriert wurde. Im Beispiel-Szenario wird Keycloak verwendet, weil es im Vergleich zur Realisierung eines eigenen Authentifizierungsservers (z.B. mit Spring) erheblich einfacher aufzusetzen ist. Außerdem erlaubt es Keycloak mit geringem Aufwand zusätzliche Benutzerdatenbanken anzubinden. Dadurch kann die Authentifizierung unabhängig von der Benutzerverwaltung beim Kunden durchgeführt werden. Durch die Anbindung von externen Benutzerdatenbanken entsteht auch eine zusätzliche Schicht zwischen Benutzerverwaltung und Services, welche dazu verwendet werden kann, die Befugnisse von Services getrennt zu verwalten.

Zur Installation von Keycloak sei auf die [Dokumentation](#) verwiesen. Im Beispiel-Szenario wurde der Operation-Mode *Standalone* gewählt, welcher für Produktsysteme nicht empfohlen wird. Die Konfigurationen eines Keycloak Servers können einfach importiert und exportiert werden. Die Konfiguration des Keycloak Servers, der im Beispiel-Szenario verwendet wurde, befindet sich im Github-Repository im Ordner "keycloak". Im Beispiel-Szenario läuft Keycloak auf Port 8180. Um den Port in der *Standalone*-Variante zu konfigurieren, muss in der Datei „standalone/configuration/standalone.xml“ der *socket-binding-Eintrag* auf 8180 gesetzt werden.

3.1 OAuth2 Clients

Im OAuth2 Protokoll wird die Rolle Client definiert. Ein Client ist eine Anwendung die auf Benutzer-Accounts zugreifen will. Die Services im Beispiel-Szenario, welche Zugriffe authentifizieren wollen, sind also OAuth2 Clients. In Keycloak, in der Administrations-Konsole, gibt es einen Menüeintrag Clients, über welchen Clients erstellt und konfiguriert werden können.

Im Beispiel-Szenario wurde für alle Services ein Client angelegt. Für jeden Client ist ein Name und ein Access Type zu definieren. Im Beispiel-Szenario wurde für alle Services, ausgenommen des Webclients, der Access Type „confidential“ gewählt. Durch diesen Access Type können die Services bei fehlender Authentifizierung einen Redirect auf die Login-Seite von Keycloak durchführen. Für den Webclient wurde der Access Type „public“ verwendet, da bei einer JavaScript-Anwendung das Client-Secret nicht geheim gehalten werden kann. Der dritte mögliche Access Type ist „bearer-only“. Bei diesem Typ erfolgt die Authentifizierung ausschließlich über einen Token, eine Weiterleitung zur Login-Seite ist nicht möglich. Dieser Access Type ist dann zu wählen, wenn ein Service kein Userinterface anbietet.

3.2 Benutzer und Rollen in Keycloak

Keycloak wird im Beispiel-Szenario sowohl als Identitätsmanagementsystem als auch als Authentifizierungsdienst verwendet. Um Keycloak als Identitätsmanagementsystem zu verwenden wurden die Rollen wie in Abschnitt 2.2 beschrieben in Keycloak erstellt (Rollennamen wurden großgeschrieben, da das in Spring der Standard ist). Weiters wurden für die erstellten Rollen entsprechende Benutzer angelegt (siehe Abschnitt 2.2) und die Rollen zugewiesen.

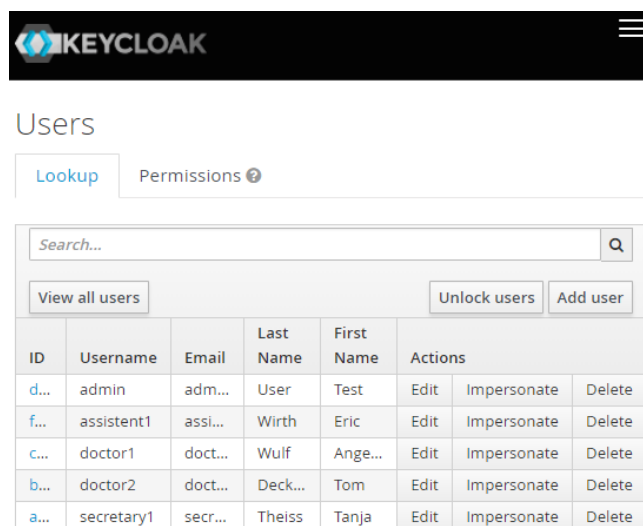


Abbildung 3 Keycloak Benutzer

User Storage Service Provider Interface

Keycloak ist sehr anpassbar und erlaubt es auch eigenen Code zu integrieren, um Anpassungen durchzuführen. Eigener Code kann über Service Provider Interfaces (SPI) integriert werden. Dafür gibt es unterschiedliche SPI-Typen, z.B. *UserStorage*. Um SPIs zu erstellen, müssen Java-Interfaces implementiert werden. Für jeden SPI muss entsprechend des Typ das *ProviderFactory*-Interface überschrieben werden, in welchem

der Provider konfiguriert werden kann. Die *ProviderFactories* müssen in einem Ordner „META-INF/services“ registriert werden. Weiters können mehrere *Provider*-Interfaces überschrieben werden, über welche die Funktionalität gesteuert wird. Um implementierte SPIs in Keycloak zu installieren, wird das gebaute JAR-File in einen Ordner „standalone/deployments“ kopiert und kann anschließend verwendet werden.

Um externe Benutzerdatenbanken in Keycloak zu integrieren gibt es die User Storage SPI. Für die User Storage SPI werden mehrere *Provider*-Interfaces zur Verfügung gestellt, welche je nach Fähigkeit der zu integrierenden Benutzerdatenbank (z.B. Benutzer hinzufügen, Credentials ändern) überschrieben werden können. Zu Testzwecken wurde im Beispiel-Szenario der Storage Provider aus dem folgenden Github-Projekt integriert: <https://github.com/dasniko/keycloak-user-spi-demo>.

3.3 Dynamische Rechteverwaltung

Eine der wesentlichen Anforderungen ist es, dass die Rechte die mit einer Rolle verbunden sind änderbar sind, ohne Services neu installieren zu müssen. Im Beispiel-Szenario wurde für diesen Zweck ein einfacher Ansatz gewählt. Im Beispiel-Szenario wurden für die in Abschnitt 2.2 beschriebenen Befugnisse eigene Rollen erstellt. Um diese Befugnis-Rollen von den tatsächlichen Rollen getrennt verwalten zu können, wurden die Befugnisse als Client-Rollen definiert. Client-Rollen sind nur für einen bestimmten OAuth-Client gültig und können auf Ebene der Clients angelegt werden. Das ermöglicht es die Befugnisse eines Service getrennt von den anderen Services zu verwalten (Anmerkung: die Verwendung von Client-Rollen kompliziert die Authentifizierung im Service. Sind in einem Service sowohl die globalen als auch die Client-Rollen notwendig bedeutet dies mehr Konfigurationsaufwand, siehe Abschnitt 4.1).

Um die Rollen mit den Befugnis-Rollen zu verbinden, bietet Keycloak das Konzept der „Composite Roles“. Dieses Konzept erlaubt es, dass Rollen vererbt werden. Z.B. kann man die Rolle „Doctor“ als „Composite Role“ definieren und bestimmen, dass die Rolle „Doctor“ von der Rolle „Assistent“ erbt. Das würde heißen, dass alle Personen, welchen die Rolle „Doctor“ zugewiesen ist, auch die Rolle „Assistent“ besitzen. Dieses Konzept kann man auch benutzen um Client-Rollen zu vererben. Z.B. kann man bestimmen, dass die Rolle „Assistent“ die Befugnis „Read_Laboratory_Results“ erbt.

Eine Alternative um eine dynamische Rechteverwaltung umzusetzen ist die Verwendung einer Access Control List (ACL). Bei einer ACL definiert man welche Identitäten welche Rechte auch welches Objekt haben. Dieser Ansatz ermöglicht eine sehr feingranulare Rechteverwaltung, ist aber im Vergleich mit dem zuvor beschriebenen Ansatz erheblich komplexer. Eine ACL könnte man sowohl auf Basis von Spring (siehe <https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#domain-acls>) als auch mit Keycloak (siehe https://www.keycloak.org/docs/latest/authorization_services/index.html) realisieren.

3.4 Custom Login Theme

Keycloak bietet die Möglichkeit das Aussehen des Userinterfaces an das Unternehmen anzupassen. Es gibt mehrere Möglichkeiten Anpassungen vorzunehmen: Login-Seite, Administrator Konsole, Benutzerprofilverwaltung. Im Beispiel-Szenario wurde diesbezüglich das Login Theme angepasst. Die Themes von Keycloak werden in einem Ordner „themes“ verwaltet. In diesem Ordner befinden sich auch die vorinstallierten Default-Themes. Themes bestehen aus HTML-Templates, Images, Stylesheets, Script und Properties. Bei der Erstellung eines neuen Themes wird empfohlen ein bestehendes Theme zu erweitern. Wenn man ein Theme erweitert können individuell Ressourcen (Template, Styles) überschrieben werden. Um ein neues Theme zu erstellen wird ein neuer Ordner innerhalb des „themes“ Ordner erstellt. Der Name dieses Ordners wird zum Namen des Themes. Für jeden Typ (z.B. Login-Seite, Administrator Konsole) wird innerhalb des erstellten Ordners ein weiterer Ordner erstellt. Die Basis jedes Themes ist ein Datei mit dem Namen „theme.properties“. In dieser Datei wird das Theme konfiguriert. Es wird definiert welches Theme erweitert wird und welche Styles etc. angewendet werden. Im Beispiel-Szenario wurde das Standard-Keycloak-Theme erweitert und Anpassungen wurden rein über CSS und Bilder durchgeführt. Das erweiterte Theme befindet sich ebenfalls im Github-Repository im Ordner „keycloak/themes“. Nachdem ein neues Theme erstellt wurde, kann dieses in der Administrator Konsole angewendet werden. Themes können Global für alle Clients, oder individuell pro Client konfiguriert werden.

4 Authentication

In diesem Kapitel wird gezeigt, wie die einzelnen Services die Authentifizierung mit Keycloak durchführen.

4.1 Spring Authentication

Da die meisten Services des Beispiel-Szenarios auf dem Spring-Framework basieren, wird in diesem Abschnitt auf die Möglichkeiten eingegangen, wie der Zugriff auf Spring Services mittels Keycloak authentifiziert werden kann. Die Verwendung von Spring Security (<https://spring.io/projects/spring-security>) ist de-facto der Standard bei der Sicherung von Spring Applikationen. Bei der Verwendung von Spring Security wird die Authentifizierung von der Zugriffskontrolle getrennt. Als Ergebnis der Authentifizierung in Spring wird ein Authentifikationsobjekt an den Thread gebunden. Auf dieses Objekt kann, wie in Code Beispiel 1 gezeigt, zugegriffen werden. Zur Authentifizierung mit Spring gibt es bereits einige vordefinierte Authentication Manager, z.B. Basic Auth, OAuth2 die sich entsprechend in die Spring-Security-Filter-Kette hängen und die Authentifikation an den Thread binden. Teil dieses Authentifikationsobjekt sind auch „Authorities“. Diese „Authorities“ werden zur Abbildung der Befugnisse verwendet. Rollen sind eine spezielle Form von „Authorities“ die mit dem String „ROLE_“ beginnen. Im Folgenden wird gezeigt wie in Spring die Authentifizierung mit Keycloak und dem OAuth2 Protokoll durchgeführt werden kann.

```

Authentication auth = SecurityContextHolder.getContext().getAuthentication();
auth = {
    authenticated: true,
    authorities: [„ROLE_USER“]
    principal:{
        username: „user“
        password: „password“
    }
}

```

Code Beispiel 1 Security Context

4.1.1 Spring Service Authentication mit Keycloak Adapter

Keycloak stellt eigene Client-Adapter zur Verfügung, um Anwendungen mit unterschiedlichen Technologien zu sichern. Für Spring Security und für Spring Boot gibt es eigene Keycloak Client Adapter. In diesem Abschnitt wird gezeigt, wie die Authentifizierung in Spring mit diesen Adaptern durchgeführt werden kann.

Zunächst sei darauf hingewiesen, dass es für unterschiedliche Spring Boot Versionen, unterschiedliche Client Adapter gibt (org.keycloak.keycloak-spring-boot-starter vs. org.keycloak.keycloak-spring-boot-2-starter). Da die Services im Beispiel-Szenario Spring Boot in Version 2 benutzen wird auch der entsprechende Keycloak Adapter benutzt.

Die Konfiguration der Authentifizierung mit dem Keycloak Adapter ist im Github-Repository im Ordner „securitylibrary“ ersichtlich. Zur Konfiguration des Keycloak Adapters im Beispiel-Szenario werden die Konfigurationen wie in Code Beispiel 2 benötigt. Diese Konfiguration wird im Beispiel-Szenario vom Konfigurationsserver geladen. Für weitere Konfigurationsmöglichkeiten siehe [Link](#).

```

keycloak.auth-server-url = http://localhost:8180/auth/
keycloak.realm = stiwa
keycloak.resource = laboratory-service
keycloak.principal-attribute: preferred_username
keycloak.ssl-required = external
keycloak.credentials.secret = aa854057-f09e-413a-8a63-488f51a69d0f
keycloak.use-resource-role-mappings = true #important to use client roles

```

Code Beispiel 2 Keycloak Adapter Konfiguration

Neben den Konfigurationen muss auch eine Java-Konfiguration vom Typ *KeycloakWebSecurityConfigurerAdapter* erstellt werden um die Authentifizierung mittels Keycloak Adapter durchführen zu können (siehe Code Beispiel 3). In dieser Klasse kann mittels *AuthortyMapper* konfiguriert werden wie die Keycloak-Rollen in Spring abgebildet werden sollen. Weiteres kann die *SessionStrategy* definiert werden und es sollte bei Verwendung von Spring Boot der *KeycloakSpringBootConfigResolver* als Bean zur Verfügung gestellt werden, der dafür sorgt, dass die bereitgestellten Konfigurationen verwendet werden. Durch diese beschriebenen Konfigurationsmaßnahmen werden alle nötigen Authentifikationsinformationen inklusive der Rollen an den Thread gebunden.

Besonders hervorgehoben bei der Konfiguration des Keycloak Adapters sei noch der *CustomKeycloakAuthenticationProvider*. Dieser wurde erstellt, um sowohl die globalen als auch die Client-Rollen aus Keycloak in die Authentifikation zu laden. Mit der Konfiguration „use-resource-role-mappings“ kann definiert werden, ob die Client-Rollen oder die globalen Rollen verwendet werden soll. Durch die Implementierung eines

eigenen Authentication Provider können beide Rollen-Typen geladen werden.

```
@Configuration
@ComponentScan(basePackageClasses = KeycloakSecurityComponents.class)
public class KeycloakConfig extends KeycloakWebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

        SimpleAuthorityMapper grantedAuthorityMapper = new SimpleAuthorityMapper();
        grantedAuthorityMapper.setPrefix("");
        grantedAuthorityMapper.setConvertToUpperCase(true);

        KeycloakAuthenticationProvider keycloakAuthenticationProvider = new
            CustomKeycloakAuthenticationProvider();//keycloakAuthenticationProvider();

        keycloakAuthenticationProvider
            .setGrantedAuthoritiesMapper(grantedAuthorityMapper);

        auth.authenticationProvider(keycloakAuthenticationProvider);
    }

    @Bean
    public KeycloakSpringBootConfigResolver keycloakConfigResolver() {
        return new KeycloakSpringBootConfigResolver();
    }

    @Bean
    @Override
    protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
        return new NullAuthenticatedSessionStrategy();//stateless
    }
}
```

Code Beispiel 3 Keycloak Java Konfiguration

4.1.2 Spring Service Authentication mit Spring Cloud Security

Alternativ kann statt der Verwendung des Keycloak Client Adapters auch mit Bibliotheken des Springframeworks gearbeitet werden. Dafür gibt es laut meinem Wissensstand zwei Alternativen: „spring-security-oauth2“ oder „spring-cloud-starter-oauth2“. Der Vorteil der ersten Variante ist, dass mehrere Authentication-Server angebunden werden können und dass bereits einige Authentication-Server vorkonfiguriert sind (z.B. Github, Facebook). Der Vorteil der Cloud-Variante ist, dass er besser in eine Microservice-Infrastruktur integriert werden kann, da unter anderem Funktionen zur Verfügung gestellt werden, um Token bei ausgehenden Request weiterzugeben. Im Beispiel-Szenario wird daher die Cloud-Variante verwendet. Die Konfiguration der Authentifizierung mit der Spring Cloud OAuth2 Bibliothek ist im Github-Repository im Ordner „cloudsecuritylibrary“ ersichtlich.

Auch bei der Verwendung der Spring-Variante müssen einige Konfigurationen vorgenommen werden. Die notwendigen Konfigurationen im Beispiel-Szenario sind in Code Beispiel 4 ersichtlich. Neben den Konfigurationen wurden zur Authentifizierung auch noch weitere Maßnahmen im Code getroffen. Zum einen wurde die Annotation *EnableOAuth2Sso* an eine Konfiguration Klasse angehängt. Diese Annotation ermöglicht es, dass Browser sich nur einmalig einloggen müssen und dass bei nicht-authentifizierten Zugriff auf die Login-Seite weitergeleitet wird. Diese Annotation ist nur notwendig, wenn es sich um einen Service handelt, der ein Userinterface anbietet. Neben Single-Sign-On über den Browser wurde auch die Token-Authentifizierung aktiviert um Service-zu-Service-Kommunikation zu ermöglichen. Zur Aktivierung der Token-Authentifizierung gibt es die Annotation *EnableResourceServer*. Zur Token-Authentifizierung wurde auch eine Konfigurationsklasse erstellt um bei Nicht-Authentifizierten Zugriff auf den Single-Sign-On Filter zu verweisen (siehe Code Beispiel 5).

```

auth-url: http://localhost:8180/auth/realms/stiwa/protocol/openid-connect/
security:
  oauth2:
    client:
      clientId: laboratory-service
      clientSecret: aa854057-f09e-413a-8a63-488f51a69d0f
      accessTokenUri: ${auth-url}/token
      userAuthorizationUri: ${auth-url}/auth
      scope: openid, profile, roles
    resource:
      user-info-uri: ${auth-url}/userinfo

```

Code Beispiel 4 Spring OAuth Konfiguration

```

@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(ResourceServerSecurityConfigurer resources) throws Exception {

        @Override
        public void configure(HttpSecurity http) throws Exception {
            http
                .requestMatcher(BearerTokenRequestMatcher.build())
                .authorizeRequests()
                    .anyRequest().permitAll().and();
        }

        public static class BearerTokenRequestMatcher implements RequestMatcher {
            @Override
            public boolean matches(HttpServletRequest request) {
                return Optional.ofNullable(request.getHeader("Authorization"))
                    .map(value -> value.startsWith("Bearer"))
                    .orElse(false);
            }
            public static BearerTokenRequestMatcher build() {
                return new BearerTokenRequestMatcher();
            }
        }
    }
}

```

Code Beispiel 5 Spring OAuth Java Konfiguration

Bei Verwendung des Keycloak Adapters wurden die Rollen, welche in Keycloak definiert wurden, automatisch in dem Authentifikationsobjekt abgelegt. Bei Verwendung der Spring Bibliothek ist das ohne entsprechende Einstellung in Keycloak nicht möglich. Um die Rollen aus Keycloak für Spring verständlich zu machen, wurde in Keycloak ein *Client Scope* angelegt. Scopes sind ein Teil der OpenID Spezifikation und bestimmen welche Informationen in den ID-token bzw. in die *Userinfo* kodiert werden. Spring benutzt automatisch das Attribut „authorities“ der *Userinfo* um die Authorities (=Rollen) zu extrahieren. Daher wurde im Beispiel-Szenario ein Client-Scope mit dem Namen „authorities“ angelegt, der die Client-Rollen in das Authorities-Attribut kodiert. Die globalen Rollen aus Keycloak werden in ein Attribut „roles“ kodiert. Die Client-Rollen im Authorities-Attribut werden durch diese Konfiguration in Keycloak automatisch als Rollen erkannt und entsprechend verarbeitet. Um auch die globalen Rollen nutzen zu können wurde ein Bean vom Type *AuthoritiesExtractor* erstellt (siehe Code Beispiel 6), der dafür sorgt, dass sowohl Client-Rollen als auch globale Rollen in die Authentifikation übernommen werden.

```

@Component
public class CustomAuthoritiesExtractor implements AuthoritiesExtractor{

    @Override
    public List<GrantedAuthority> extractAuthorities(Map<String, Object> map) {
        List<String> roles = new ArrayList<>();
        if(map.containsKey("roles")) {
            roles.addAll((Collection<? extends String>) map.get("roles"));
        }
        if(map.containsKey("authorities")) {
            roles.addAll((Collection<? extends String>) map.get("authorities"));
        }

        SimpleAuthorityMapper mapper = new SimpleAuthorityMapper();
        mapper.setConvertToUpperCase(true);
        mapper.setPrefix("");

        Collection<? extends GrantedAuthority> authorities =
            AuthorityUtils.createAuthorityList(roles.toArray(new String[0]));

        return new ArrayList<GrantedAuthority>(mapper.mapAuthorities(authorities));
    }
}

```

Code Beispiel 6 AuthoritiesExtractor

4.1.3 Service zu Service Kommunikation

Eine wichtige Eigenschaft von Microservices ist es, dass Services miteinander kommunizieren. Da alle Services eine Authentifizierung erfordern, muss bei einer Kommunikation zwischen Services ein OAuth-Token übergeben werden. Man kann dabei zwischen zwei Szenarien unterscheiden. Beim ersten Szenario arbeitet der Client-seitige Service selbst eine Anfrage ab und benötigt dabei einen anderen Service. In diesem Fall soll der Token des ursprünglichen Request weitergegeben werden. Beim zweiten Szenario löst der Client-seitige Service unabhängig einen Request aus. Im Folgenden wird beschrieben, wie mit diesen Szenarien im Beispiel umgegangen wird.

Weitergabe von Access Token (Token Relay)

Soll bei einer Service-zu-Service Kommunikation ein Token eines ursprünglichen Request weitergegeben werden, unterscheidet sich der Ansatz basierend auf der verwendeten Bibliothek.

Keycloak Adapter

Wird der Keycloak Adapter benutzt und soll ein Token eines Requests weitergegeben werden, steht eine Klasse *KeycloakRestTemplate* zur Verfügung. Diese Klasse erbt von *RestTemplate* (=Klasse um HTTP-Aufrufe durchzuführen). Ein Objekt dieser Klasse kann sehr einfach mit folgendem Code instanziiert werden.

```

@Bean
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public RestTemplate keycloakRestTemplate(KeycloakClientRequestFactory
    keycloakClientRequestFactory, LoadBalancerInterceptor interceptor) {
    KeycloakRestTemplate result =
        new KeycloakRestTemplate(keycloakClientRequestFactory);
    result.getInterceptors().add(interceptor);
    return result;
}

```

Code Beispiel 7 KeycloakRestTemplate

Im Beispiel-Szenario wird das *KeycloakRestTemplate* als Bean mit dem Scope Prototype definiert, damit bei jeder Verwendung eine neue Instanz dieser Klasse erstellt wird. Weiters wird im Beispiel-Szenario *Eureka* verwendet und es muss daher ein Interceptor gesetzt werden, um den Service-Namen in einen Host umwandeln zu können und das clientbasierte Loadbalancing durchzuführen. Wird das *KeycloakRestTemplate* wie in

Code Beispiel 7 erstellt, kann es wie ein normales *RestTemplate* verwendet werden und der Token des ursprünglichen Request wird übergeben.

Im Beispiel-Szenario wird zur Kommunikation zwischen den Services ein *Feign*-Client benutzt. Damit auch bei der Verwendung von *Feign*-Clients die Authentication-Token übergeben werden, muss dies entsprechend wie in Code Beispiel 8 ersichtlich, konfiguriert werden. Es wird für die *Feign*-Clients ein Request-Interceptor gesetzt, welcher im HTTP-Header den Access-Token hinzufügt. Um den Token des ursprünglichen Requests zu ermitteln wurde eine Komponente *AccessTokenResolver* realisiert (siehe Code Beispiel 9). Die Klasse *KeycloakFeignConfiguration* wurde mit der Annotation *Configuration* versehen. Dadurch wird diese Konfiguration für alle *Feign*-Clients angewendet. Alternative zur Konfiguration aller *Feign*-Clients, können auch einzelne Clients konfiguriert werden (siehe Code Beispiel 10).

```
@Configuration //Apply this configuration to all feign clients
public class KeycloakFeignConfiguration{

    public static final String AUTHORIZATION_HEADER = "Authorization";

    @Autowired
    AccessTokenResolver accessTokenResolver;

    @Bean
    public RequestInterceptor keycloakInterceptor() {
        return new RequestInterceptor() {
            @Override
            public void apply(RequestTemplate template) {
                String token = accessTokenResolver.getAccessTokenOfUser();
                if(token!=null) {
                    template.header(AUTHORIZATION_HEADER, "Bearer " + token);
                }
            }
        };
    }
}
```

Code Beispiel 8 Feign Konfiguration

```
protected String getAccessTokenOfUser() {
    KeycloakSecurityContext context = getKeycloakSecurityContext();
    if(context!=null) {
        return context.getTokenString();
    }
    return null;
}

protected KeycloakSecurityContext getKeycloakSecurityContext() {
    Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();
    KeycloakAuthenticationToken token = (KeycloakAuthenticationToken) authentication;
    KeycloakSecurityContext context = token.getAccount().getKeycloakSecurityContext();

    return context;
}
```

Code Beispiel 9 Authentication Token ermitteln

```
@FeignClient(name = "laboratory-service", configuration =
KeycloakFeignConfiguration.class)
```

Code Beispiel 10 Anwendung Feign Konfiguration

Spring Cloud Security

Zur Konfiguration eines *RestTemplate*, mit der Spring Cloud Security Bibliothek steht eine Klasse *OAuth2RestTemplate* zur Verfügung. Diese Klasse kann ähnlich dem *KeycloakRestTemplate* konfiguriert werden (siehe Code Beispiel 11). Im Vergleich zum *RestTemplate* des Keycloak Adapters bietet das *OAuth2RestTemplate* den Vorteil, dass Accesstoken aktualisiert werden, wenn Sie abgelaufen sind.

```
@Bean
public OAuth2RestTemplate restTemplate(OAuth2ProtectedResourceDetails resource,
    OAuth2ClientContext context, LoadBalancerInterceptor interceptor) {
    OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource, context);
    restTemplate.getInterceptors().add(interceptor);
    return restTemplate;
}
```

Code Beispiel 11 OAuth2RestTemplate

Auch die Konfiguration von *Feign*-Clients mit dem Spring Cloud Security Framework ist ähnlich der Konfiguration mit Keycloak. Um den aktuellen Token zu erheben, wird allerdings nicht auf den Security-Kontext zugegriffen, sondern das *OAuth2RestTemplate* verwendet, welches eine Methode besitzt die den Access Token zurückgibt. Auch dabei besteht der Vorteil, dass abgelaufene Token aktualisiert werden.

Service Authentication

In einigen Fällen ist es notwendig, dass Services miteinander kommunizieren ohne, dass diese Kommunikation von einer vorherigen Anfrage ausgelöst wurde. Es kann in diesem Fall kein Token weitergegeben werden, sondern ein Service muss sich selbst bei einem anderen Service authentifizieren. Dies ist im Beispiel-Szenario der Fall, wenn Services Konfigurationen vom Konfigurationsserver laden und bei der Kommunikation mit der Service-Discovery. Es sind aber auch andere Szenarien denkbar, bei der diese Form der Authentifizierung notwendig ist (z.B., wenn Services periodisch Aufgaben übernehmen oder Events aus einer Queue verarbeiten). Damit Services sich selbst Authentifizieren können, müssen entsprechende User und Rollen für Services vergeben werden. Im Beispiel-Szenario wurde zu diesem Zweck in Keycloak die Funktion Service Accounts für alle Clients aktiviert. Dies ermöglicht es, dass Services mit Client-ID und Client-Secret einen Access-Token von Keycloak beziehen können. Für Services, die sich auf diese Weise authentifizieren, wurde die Rolle System definiert. In diesem Abschnitt wird gezeigt wie diese Form der Authentifizierung sowohl mit dem Keycloak Adapter als auch mit Spring Cloud Security gelöst werden kann.

Keycloak Adapter

Bei Verwendung des Keycloak Adapters gibt es (zumindest laut meinem Wissensstand) keine Funktion um mit einer Client-ID und einem Client-Secret einen Token von Keycloak zu beziehen. Diese Funktionalität wurde daher im Beispiel-Szenario selbst entwickelt, indem die Access-Token-Uri von Keycloak mit den Credentials des Clients aufgerufen wird (siehe Code Beispiel 12). Das Vorgehen hat den Nachteil, dass bei jeder Authentifizierung die Credentials an Keycloak gesendet werden um einen Access Token zu erhalten.


```

@Value("${auth.client-id}")
private String clientId;
@Value("${auth.client-secret}")
private String clientSecret;
@Value("${auth.access-token-uri}")
private String authEndpoint;

protected String getAccessTokenOfService() {
    RestTemplate restTemplate = new RestTemplate();

    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

    MultiValueMap<String, String> map = new LinkedMultiValueMap<String, String>();
    map.add("grant_type", "client_credentials");

    HttpEntity<MultiValueMap<String, String>> request = new
    HttpEntity<MultiValueMap<String, String>>(map, headers);

    restTemplate.getInterceptors().add(new BasicAuthorizationInterceptor(clientId,
    clientSecret));

    ResponseEntity<Map> response = restTemplate.postForEntity(authEndpoint, request,
    Map.class);
    return response.getBody().get("access_token").toString();
}

```

Code Beispiel 12 Keycloak Adapter Service Access Token

Spring Cloud Security

Bei Verwendung von Spring Cloud Security besteht die Möglichkeit ein *OAuth2RestTemplate* auf Basis von Client-ID und Client-Secret zu erstellen (siehe Code Beispiel 13). Von diesem Objekt kann anschließend, ein Access-Token abgefragt werden. Dieser Ansatz bietet den Vorteil, dass Client-ID und Client-Secret nicht jedes Mal an Keycloak gesendet werden, da das *OAuth2RestTemplate*, den Access-Token bei Bedarf automatisch erneuert.

```

@Bean
public OAuth2RestTemplate serviceRestTemplate() {
    return new OAuth2RestTemplate(oAuthDetails());
}

@Bean
@ConfigurationProperties("auth")
protected ClientCredentialsResourceDetails oAuthDetails() {
    return new ClientCredentialsResourceDetails();
}

```

Code Beispiel 13 Spring Cloud Security Service Access Token

Authentifizierung am Konfigurationsserver

Im Beispiel-Szenario wird die Kommunikation der Services mit dem Konfigurationsserver ebenfalls mit OAuth2 authentifiziert. Die Services müssen sich daher wie oben beschrieben bei Keycloak authentifizieren und bei Request zum Konfigurationsserver den Token als Header übermitteln. Um dieses Verhalten zu konfigurieren muss ein Bean vom Typ *ConfigServicePropertySourceLocator* erstellt werden, bei dem ein *RestTemplate* mit entsprechendem Header definiert werden kann (siehe Code Beispiel 14). Bei Verwendung von Spring Cloud Security kann das *OAuth2RestTemplate* verwendet werden. Bei Verwendung des Keycloak Adapters wurde im Beispiel-Szenario ein eigenes

RestTemplate erstellt, welches den Token zum Header hinzufügt. Damit das notwendige Bean während des Bootstrap Prozesses von Spring zur Verfügung steht, muss die Klasse, in der das Bean definiert wird, in den Spring-*Factories* registriert werden (siehe Datei „META-INF/spring.factories“)

```
@Autowired
private ConfigurableEnvironment environment;

@Autowired
@Qualifier("serviceRestTemplate")
OAuth2RestTemplate restTemplate;

@Bean
@Primary
public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
    ConfigClientProperties clientProperties = configClientProperties();
    ConfigServicePropertySourceLocator configServicePropertySourceLocator = new
    ConfigServicePropertySourceLocator(clientProperties);
    configServicePropertySourceLocator.setRestTemplate(restTemplate);
    return configServicePropertySourceLocator;
}

@Bean
public ConfigClientProperties configClientProperties() {
    ConfigClientProperties client = new ConfigClientProperties(this.environment);
    client.setEnabled(false);
    return client;
}
```

Code Beispiel 14 Authentifizierung am Kommunikationsserver

Authentifizierung an der Service Discovery

Die Kommunikation von Services mit der Service-Discovery wird ebenfalls mit OAuth2 authentifiziert. Die Authentifizierung an der Service Discovery funktioniert ähnlich der Authentifizierung am Konfigurationsserver. Für diesen Zweck muss ein Bean vom Typ *DiscoveryClientOptionalArgs* erstellt werden (siehe Code Beispiel 15). Auch die Klasse die dieses Bean definiert muss im Beispiel-Szenario in den Spring-*Factories* registriert werden, da über die Service-Discovery der Host des Konfigurationsservers aufgelöst wird.

```
@Autowired
@Qualifier("serviceRestTemplate")
OAuth2RestTemplate restTemplate;

@Bean
DiscoveryClientOptionalArgs discoveryClientArgs() {

    Collection<ClientFilter> filters = new ArrayList<>();
    filters.add(new ClientFilter() {

        @Override
        public ClientResponse handle(ClientRequest cr) throws ClientHandlerException {
            String token = restTemplate.getAccessToken().getValue();
            if(token!=null) {
                cr.getHeaders().add(HttpHeaders.AUTHORIZATION, "Bearer " + token);
            }
            return getNext().handle(cr);
        }
    });

    DiscoveryClientOptionalArgs args = new DiscoveryClientOptionalArgs();
    args.setAdditionalFilters(filters);
    return args;
}
```

Code Beispiel 15 Authentifizierung Service Discovery

4.2 JavaScript Client Authentication (mit Keycloak Adapter)

Im Beispiel-Szenario wurde auch ein JavaScript Client realisiert, welcher die Funktionen des Laboratory- und Patient Service nutzt. Zur Umsetzung dieses JavaScript Clients wurde Angular (in Version 7) verwendet. Zur Authentifizierung von Usern wurde der JavaScript Adapter von Keycloak verwendet. Zur Verwendung dieses Adapters siehe https://www.keycloak.org/docs/latest/securing_apps/index.html#_javascript_adapter. Der Source Code dieses JavaScript Clients befindet sich im Ordner "webclient" im Github-Repository. Zur Erstellung dieses Angular-Projektes wurde die Angular CLI (siehe <https://cli.angular.io/>) verwendet.

In jeder Instanz des Keycloak-Servers befindet sich auch der entsprechende JavaScript-Adapter. Es wird empfohlen diesen Adapter zu benutzen, da dadurch auch nach Updates des Keycloak-Servers, der JavaScript-Adapter in einer kompatiblen Version verwendet wird. Der Import dieses Adapters erfolgt daher wie in Code Beispiel 16 abgebildet. Zur Konfiguration dieses Adapters im Beispiel-Szenario siehe Ordner "webclient\angular-client\src\app\keycloak" im Github-Repository und Code Beispiel 17. Der JavaScript Adapter von Keycloak bietet eine Methode `Keycloak`, die ein Objekt liefert, welches die Funktionalitäten zur Authentifizierung zur Verfügung stellt. Zur Ausführung dieser Methode wird eine JSON-Konfigurationsdatei benötigt, die als Parameter übergeben wird. Dieses JSON-File kann vom Keycloak Server heruntergeladen werden (Tab "Installation" in Keycloak Client). Für den Angular-Client wurde in Keycloak ein eigener Client angelegt. Für diesen Client muss der Access Type "public" verwendet werden, da in einer Webapplikation das Client Secret nicht geheim gehalten werden kann.

```
<script src="http://localhost:8180/auth/js/keycloak.js"></script>
```

Code Beispiel 16 JavaScript Adapter importieren

Durch die Ausführung der Keycloak-Methode entsteht ein Keycloak-Objekt. Dieses Objekt bietet eine Methode `init`, welche bei Start einer Webapplikation ausgeführt werden soll. Bei Ausführung dieser Methode kann zwischen zwei `onLoad`-Strategien gewählt werden: "login-required", "check-sso". Bei Wahl von "login-required" wird bei Start der Applikation geprüft, ob im Browser Authentifizierungsinformationen vorhanden sind. Ist dies nicht der Fall wird auf die Login-Page weitergeleitet. Bei Wahl dieser Strategie kann die Webapplikation nicht anonym verwendet werden. Bei Wahl von "check-sso" wird bei Start der Applikation die Authentifizierung überprüft. Bei dieser Strategie kann die Applikation auch anonym verwendet werden.

```
init(): Promise<any> {
  return new Promise((resolve, reject) => {
    const keycloakAuth = Keycloak('assets/keycloak/keycloak.json');
    keycloakAuth.init({onLoad: 'check-sso'}).then(() => {
      AuthService.auth.loggedIn = keycloakAuth.authenticated;
      AuthService.auth.keycloak = keycloakAuth;
      AuthService.auth.logoutUrl = keycloakAuth.authServerUrl +
        '/realms/stiwa/protocol/openid-connect/logout?redirect_uri='+document.baseURI;
      resolve();
    }).catch(() => {
      reject();
    });
  });
}
```

Code Beispiel 17 Konfiguration Javascript Adapter

Nach Initialisierung bietet das Keycloak-Objekt mehrere Methoden zur Authentifizierung und Autorisierung von Benutzern an. Es gibt beispielsweise Methoden um einen Token zu aktualisieren, um das Benutzerprofil zu laden (siehe Code Beispiel 18) oder um zu prüfen ob ein Benutzer eine bestimmte Rolle hat (siehe Code Beispiel 19).

```
getUserProfile(): Promise<any> {
  return new Promise<any>((resolve, reject) => {
    if (!AuthService.auth || !AuthService.auth.keycloak ||
        !AuthService.auth.keycloak.authenticated) {
      reject();
    }
    else if (AuthService.auth.userProfile) {
      resolve(AuthService.auth.userProfile);
    }
    else {
      AuthService.auth.keycloak.loadUserProfile().success(user => {
        AuthService.auth.userProfile = user;
        resolve(AuthService.auth.userProfile);
      }).error(err => {
        reject(err);
      });
    }
  });
}
```

Code Beispiel 18 Benutzerprofil laden Javascript Adapter

```
hasRole(role: String): boolean {
  if (!AuthService.auth || !AuthService.auth.keycloak) {
    return false;
  }
  if (AuthService.auth.keycloak.hasRealmRole(role)) {
    return true;
  }
  return AuthService.auth.keycloak.hasResourceRole(role);
}
```

Code Beispiel 19 Rollenprüfung Javascript Adapter

Der JavaScript Adapter soll natürlich auch dazu verwendet werden, um bei Kommunikation mit den REST-Services einen Access-Token zu übergeben. Zu diesem Zweck wurde ein *HttpInterceptor* erstellt, welcher den Access-Token des aktuellen Nutzers zu jedem HTTP-Header hinzufügt (siehe Code Beispiel 20). Dieser Interceptor benutzt eine Methode *getToken*, welche einen aktuellen Token zurückgibt, indem alle 5 Minuten der Token aktualisiert wird (siehe Code Beispiel 21).

```
@Injectable()
export class TokenInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const tokenPromise: Promise<string> = this.auth.getToken();
    const tokenObservable: Observable<string> = from(tokenPromise);

    return tokenObservable.pipe(
      map(authToken => {
        req = req.clone({setHeaders: {'Authorization': 'Bearer ' + authToken}});
      }),
      concatMap(request => {
        return next.handle(req);
      })
    );
  }
}
```

Code Beispiel 20 HttpInterceptor JavaScript Adapter

```
getToken(): Promise<string> {
  return new Promise<string>((resolve, reject) => {
    if (AuthService.auth.keycloak.token) {
      AuthService.auth.keycloak.updateToken(5).success((refreshed) => {
        resolve(<string>AuthService.auth.keycloak.token);
      }).error(() => {
        reject('Failed to refresh token');
      });
    }
    else {
      AuthService.auth.keycloak.login();
    }
  });
}
```

```

        reject('Not logged in');
    }
});
}

```

Code Beispiel 21 Token Resolver JavaScript Adapter

5 Authorization (Access Control)

Im vorherigen Kapitel wurde beschrieben wie die Identität des Aufrufers eines Service in Spring festgestellt und sichergestellt werden kann. Als Ergebnis der Authentifizierung in Spring werden die Identitätsinformationen an den Thread gebunden. Diese Informationen können anschließend im Service genutzt werden um zu Entscheiden ob ein Servicenutzer eine Aktion durchführen darf bzw. welche Informationen ein Nutzer sehen darf. In diesem Kapitel wird beschrieben, wie die Zugriffskontrolle mit Spring durchgeführt werden kann.

5.1 Web Security

Eine Möglichkeit den Zugriff auf Funktionen einzuschränken ist Web Security. Bei dieser Art der Zugriffskontrolle erfolgt die Absicherung direkt auf Ebene von HTTP-Endpunkten. Mit unterschiedlichsten Matcher-Methoden können dabei Endpunkt identifiziert und abgesichert werden. Standardmäßig ist Spring Security so konfiguriert, dass alle Nutzer authentifiziert sein müssen um HTTP-Endpunkte abfragen zu können (daher kein anonymer Zugriff möglich ist). Bei Verwendung von Web Security wird bei notwendiger aber fehlender Authentifizierung ein 401 (Unauthorized) Response erstellt. Bei fehlender Berechtigung (z.B. fehlender Rolle) wird ein 403 (Forbidden) Response erstellt.

Das Code Beispiel 22 zeigt wie diese Form der Absicherung beispielsweise konfiguriert werden kann. Bei diesem Beispiel können alle Endpunkte deren Uri mit „/api/users/“ beginnt nur von Nutzern mit der Rolle „ADMIN“ aufgerufen werden. Weiters wird definiert, dass alle anderen GET-Endpunkte für alle Nutzer (auch anonym) aufrufbar sind und dass für POST-Endpunkte eine Authentifizierung notwendig ist (unabhängig von den Rollen). Im Beispiel-Szenario wurde die Web-Security so konfiguriert, dass auf alle Endpunkte auch anonym zugegriffen werden kann.

```

@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().antMatchers("/api/users/**").hasAnyRole("ADMIN")
            and().authorizeRequests().antMatchers(HttpMethod.GET).permitAll()
            and().authorizeRequests().antMatchers(HttpMethod.POST).fullyAuthenticated();
    }
}

```

Code Beispiel 22 Web Security Konfiguration

5.2 Method-Level Security

Im Beispiel-Szenario wird alternativ zu Web-Security die Absicherung mit Method-Level Security durchgeführt. Web-Security und Method-Level Security können und werden aber

auch häufig gemeinsam eingesetzt. Bei der Method-Level Security erfolgt die Absicherung auf Ebene von Methoden. Es kann mittels Annotationen definiert werden, ob eine Methodenaufruf zulässig ist oder nicht. Die Method-Level Security wird häufig auf Service-Ebene angewendet (Klassen die mit `@Service` annotiert sind). Ein Vorteil der Method-Level Security im Vergleich zu Web-Security ist, dass auch Methodenaufrufe innerhalb eines Services abgesichert sind. Dies ist zum Beispiel dann der Fall, wenn eine Service-Methode eine andere Service-Methode aufruft.

Um Method-Level Security benutzen zu können, muss diese mit der Annotation *EnableGlobalMethodSecurity* aktiviert werden (siehe Code Beispiel 27). Es gibt drei unterschiedliche Arten von Annotationen die getrennt aktiviert und mit denen Methoden gesichert werden können: *jsr250*, *secured*, *prepost*. Code Beispiel 23 zeigt wie diese unterschiedlichen Annotationen eingesetzt werden können um eine Methode abzusichern. Die Methode des Beispiels kann nur von Benutzern aufgerufen werden, die entweder die Rolle User oder die Rolle Admin besitzen. Im Beispiel-Szenario werden die „prepost“-Annotationen verwendet, da diese mehr Möglichkeiten bieten (die anderen Annotationen sind in Code Beispiel 23 auskommentiert).

```
public interface UserService {
    // @Secured({ "ROLE_USER", "ROLE_ADMIN" }) // Secured
    // @RolesAllowed({ "ROLE_USER", "ROLE_ADMIN" }) // JSR-250
    @PreAuthorize("hasRole('ROLE_USER') or hasRole('ROLE_ADMIN')") // PrePost
    User getUserProfile (String username);
}
```

Code Beispiel 23 Beispiel Method-Level Security Annotationen

Durch die Verwendung von „prepost“-Annotationen können auch Prüfungen auf Basis der Methoden-Parameter durchgeführt werden. Code Beispiel 24 zeigt, wie geprüft werden kann ob ein Parameter „username“ mit dem Benutzernamen des authentifizierten Benutzers übereinstimmt. Es ist auch möglich die Sicherheitsabfrage nach Ausführung der Methode durchzuführen und dabei auf den Rückgabewert zuzugreifen (siehe Code Beispiel 25).

```
@PreAuthorize("#username == authentication.principal.username or
    hasRole('ROLE_ADMIN')")
User getUserProfile (String username);
Code Beispiel 24 Method-Level Security und Methoden Parameter
```

```
@PostAuthorize("returnObject.username == authentication.principal.username or
    hasRole('ROLE_ADMIN')")
User getUserProfile (String username);
Code Beispiel 25 Method-Level Security Post Authorize
```

Sollte die Prüfung von Parametern und Rückgabewerten wie oben beschrieben nicht ausreichen, können auch eigene Sicherheitsabfragen definiert werden. Dafür steht in Spring das Konzept des *PermissionEvaluators* zur Verfügung. Code Beispiel 26 zeigt wie ein *PermissionEvaluator* erstellt und Code Beispiel 27 zeigt wie dieser zur Anwendung gebracht werden kann. Durch den *PermissionEvaluator* des Code Beispiels soll überprüft werden, ob der gefragte Benutzer in derselben Abteilung arbeitet wie der authentifizierte Benutzer (siehe Code Beispiel 28)

```
@Component
public class CustomPermissionEvaluator implements PermissionEvaluator {
    @Override
    public boolean hasPermission(Authentication auth, Object targetDomainObject, Object
    permission) {
        if("SAME_DEPARTMENT".equals(permission.toString)){
            //do check (i.e. call repository, other service)
        }
        return false;
    }
}
```

```
}
}
```

Code Beispiel 26 Permission Evaluator

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityMethodConfig extends GlobalMethodSecurityConfiguration {

    @Autowired
    CustomPermissionEvaluator customPermissionEvaluator;

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        DefaultMethodSecurityExpressionHandler handler = new
        DefaultMethodSecurityExpressionHandler();
        handler.setPermissionEvaluator(customPermissionEvaluator);
        return handler;
    }
}
```

Code Beispiel 27 Konfiguration Permission Evaluator

```
@PreAuthorize("hasPermission(#username, 'SAME_DEPARTMENT')")
User getUserProfile (String username);
```

Code Beispiel 28 Verwendung Permission Evaluator

5.3 Field-Level Security

Im Beispiel-Szenario ist es nicht ausreichend, dass der Zugriff auf Methoden und Endpunkte bei fehlender Berechtigung untersagt wird. Es ist weiters auch notwendig, dass Benutzern eines Service nur jene Informationen angezeigt werden, die für den Nutzer notwendig sind. Das heißt auch, dass bei einer REST-Schnittstelle der Rückgabewert eines Endpunkts nur jene Felder beinhalten darf, die für den Benutzer notwendig sind. Nach längerer Recherche bezüglich dieses Problems konnte kein standardisierter Ansatz gefunden werden. Es gibt aber grundsätzlich einige Möglichkeiten um dieses Problem zu lösen.

Zunächst muss die Grundsatzentscheidung getroffen werden, auf welcher Ebene die Objekte auf Basis von Rollen und Berechtigungen gefiltert werden sollen. Es besteht die Möglichkeit dies bereits auf Datenebene durchzuführen. Das hätte den Vorteil, dass in der Geschäftslogik bereits keine Informationen vorhanden sind die dem authentifizierten Benutzer nicht angezeigt werden sollen. Die Durchführung auf Datenebene wäre die sicherste Variante, da die Objekte zum frühesten möglichen Zeitpunkt gefiltert werden. Dieses Vorgehen könnte aber auch zu Problemen führen, wenn z.B. ein Attribut zur Verarbeitung in der Geschäftslogik benötigt wird, dieses Attribut aber dem Benutzer nicht angezeigt werden darf. Weiters könnten die Objekte auch auf Service Ebene gefiltert werden. Dadurch wären in der Geschäftslogik alle Informationen verfügbar. Eine andere Alternative wäre es die Objekte zum Zeitpunkt der Serialisierung zu filtern. Das bedeutet, dass die Funktionen, die Objekte in JSON oder XML abbilden, entsprechend instrumentiert werden. In einer Microservice Architektur kann davon ausgegangen werden, dass auch die Filterung zum Zeitpunkt der Serialisierung ausreichend ist, da nur über APIs mit einem Microservice kommuniziert wird.

Im Folgenden werden kurz einige Möglichkeiten beschrieben um dieses Problem zu

lösen:

- **Spring Data + Security:** Bei der Verwendung von Spring Data können Security Ausdrücke, wie in Abschnitt 5.2 beschrieben (*hasRole*, *isFullyAuthenticated*, *hasPermission*), verwendet werden. Dies ermöglicht es Objekte bereits auf Datenebene zu sichern und nur die Daten von der Datenbank zu laden die der authentifizierte Benutzer sehen darf. Die Verwendung von Security Ausdrücken ist allerdings nur möglich, wenn eigene Datenbankabfragen definiert werden. Wird mit den Standard Datenbanktransaktionen gearbeitet ist das nicht möglich. Ein Beispiel dafür ist in Code Beispiel 29 ersichtlich.

```
@Query("select m from Message m where m.id = ?#{ principal?.id }")
Page<Message> findInbox(Pageable pageable);
Code Beispiel 29 Spring Data + Security
```

- **Jackson Field Security** ist ein [Github](#)-Projekt, mit welchen Domänen Objekte mit Security Annotationen versehen werden können. Diese Annotationen werden bei Serialisierung des Objekts berücksichtigt und nur die Felder serialisiert, für welche eine Berechtigung besteht. Die Verwendung dieser Bibliothek ist vom Serialisierungsframework Jackson abhängig. Da die Beteiligung an diesem Github-Projekt gering ist, ist unklar ob es für einen produktiven Einsatz geeignet ist.
- **Data Transfer Objects (DTO):** In vielen Projekten werden DTOs eingesetzt um Implementierungsdetails von Domänen Objekten zu verschleiern. Um Domänen Objekte in DTOs zu verwandeln und umgekehrt wird eine Funktion benötigt, die diese Aufgabe übernimmt. Häufig wird für diesen Zweck ein Framework eingesetzt. Ein Beispiel dafür ist [ModelMapper](#). Bei Verwendung von *ModelMapper* können individuelle Mapping-Regeln definiert werden, die auch dazu verwendet werden können um Objekte zu filtern.
- **Aspect-Oriented-Programming:** Es wäre auch möglich einen Aspekt zu definieren, der in einer bestimmten Schicht in der Applikation (z.B. alle Service-Methoden) die Domänen Objekte filtert.

JSON View zum Filtern von Domänen Objekten

Im Beispiel-Szenario hat man sich dafür entschieden, die Domänen Objekte bei der Serialisierung zu sichern. Es wird dazu das Konzept der JSON-Views des Jackson Serialisierungsframeworks verwendet, dass es erlaubt unterschiedliche Sichten auf ein Domänen Objekt zu definieren. Die JSON-Views werden verwendet, weil es ein standardisiertes Konzept eines bewährten Frameworks ist.

Bei Verwendung von JSON-Views können einzelne Attribute von Entitäten mit einer View Annotiert werden. Bei Serialisierung ist anschließend die View zu definieren, die angewendet werden soll. Während der Serialisierung werden dann nur jene Attribute serialisiert die zur definierten View gehören. Dieses Konzept wurde im Beispiel-Szenario für eine Entität *LaboratoryResult* angewendet (siehe Code Beispiel 30). Diese Entität besteht aus einer ID und value A-D. Die ID, valueA und valueB sind für alle Nutzer sichtbar. ValueC gehört zu einer View Extended und valueD gehört zu einer View Admin. Die Views in Jackson können hierarchische gegliedert werden, da immer nur eine View zur Anwendung kommen kann. Von dieser Möglichkeit wurde im Beispiel-Szenario Gebrauch gemacht und eine Menge von Views erstellt die miteinander in hierarchischer Beziehung stehen (siehe Code Beispiel 31). Um Views in hierarchische Beziehung zu stellen wird mit Vererbung gearbeitet. Neben den vordefinierten Views können in

Services auch zusätzliche Views erstellt werden, die auch von den vordefinierten Views erben können. Im Beispiel-Szenario erbt die Admin-View von der Extended-View, wodurch alle Attribute der Extended-View auch in der Admin-View sichtbar ist. Damit die Attribute eines Domänen Objekts, welche nicht mit einer View annotiert sind, ebenfalls serialisiert werden, kann folgende Konfiguration gesetzt werden:

spring.jackson.mapper.default_view_inclusion: true

```
@SecuredEntityViewPolicy(value = "MedicalViewPolicy")
public class LaboratoryResult {
    private Long id;
    private Long valueA;
    private Long valueB;
    @JsonView(Views.Extended.class)
    private String valueC;
    @JsonView(Views.Admin.class)
    private Boolean valueD;
    private Long patientSvnr;
}
```

Code Beispiel 30 JSON View LaboratoryResult

```
public class Views {
    public interface Anonymous {}
    public interface Simple extends Anonymous {}
    public interface Extended extends Simple {}
    public interface Detail extends Extended {}
    public interface Admin extends Detail {}
}
```

Code Beispiel 31 Hierarchische Views

Damit die definierten JSON-Views auch zur Anwendung kommen, muss eine Konfiguration wie in Code Beispiel 32 definiert werden. In dieser Konfiguration wird ein Standard-Mapping von Rolle auf View realisiert, so dass die Rolle Admin die View Admin, die Rolle Doctor die View Detail, usw. bekommt.

```
@ControllerAdvice
public class JsonViewConfiguration extends AbstractMappingJacksonResponseBodyAdvice {

    @Override
    protected void beforeBodyWriteInternal(MappingJacksonValue bodyContainer, MediaType
contentType, MethodParameter returnType, ServerHttpRequest request, ServerHttpResponse response){

        Class<?> viewClass = Views.Anonymous.class;

        Collection<? extends GrantedAuthority> authorities =
SecurityContextHolder.getContext().getAuthentication().getAuthorities();

        if (authorities.stream().anyMatch(o -> o.getAuthority().equals("ROLE_ADMIN"))) {
            viewClass = Views.Admin.class;
        } else if (authorities.stream().anyMatch(o -> o.getAuthority().equals("ROLE_DOCTOR"))) {
            viewClass = Views.Detail.class;
        } else if (authorities.stream().anyMatch(o -> o.getAuthority().equals("ROLE_ASSISTENT"))) {
            viewClass = Views.Extended.class;
        } else if (authorities.stream().anyMatch(o -> o.getAuthority().equals("ROLE_SECRETARY"))) {
            viewClass = Views.Simple.class;
        }
        bodyContainer.setSerializationView(viewClass);
    }
}
```

Code Beispiel 32 JSON View Konfiguration

Neben dem Standard-Mapping von Rolle zu View wurde im Beispiel Szenario auch ein Alternative geschaffen um auf Basis des authentifizierten Benutzers und des Domänen Objekts die entsprechende View zu berechnen. Es wurde dafür eine abstrakte Klasse definiert, die in den einzelnen Services erweitert werden und als Bean zur Verfügung

gestellt werden muss. Im Laboratory-Service des Beispiel-Szenarios sieht diese Konfiguration wie in Code Beispiel 33 aus. Es muss eine Methode `getJsonView` überschrieben werden, die als Parameter eine View-Policy, das zu serialisierende Objekt und die aktuellen Authentifizierungsinformationen erhält und die entsprechende View berechnen soll. Die für ein Domänen-Objekt anzuwendende View-Policy kann, bei der Entitäts-Klasse über eine eigene Annotation `SecuredEntityViewPolicy` definiert werden (siehe Code Beispiel 30). Die Definition dieses `JsonViewCalculators` ist optional. Sollte kein `JsonViewCalculator` definiert sein oder wird für ein Domän-Objekt dadurch keine View berechnet wird das Standard-Mapping benutzt.

```
@Component
public class CustomJsonViewCalculator extends JsonViewCalculator{

    @Override
    protected Class<?> getJsonView(String viewPolicy, Object responseValue,
    Authentication auth) {
        switch(viewPolicy) {

            case "MedicalViewPolicy":
                if(AuthorityChecker.hasAuthority(auth, "ROLE_ADMIN")) {
                    return Views.Admin.class;
                }
                if(AuthorityChecker.hasAuthority(auth,
"READ_EXTENDED_LABORATORY_RESULTS")) {
                    return Views.Extended.class;
                }
                return Views.Simple.class;
            }
        return null;
    }
}
```

Code Beispiel 33 JSON View Berechnung

Es besteht weiteres die Möglichkeit, dass eine View manuell im Controller eines Service gesetzt wird. Wird von dieser Möglichkeit Gebrauch gemacht, kommt der `JsonViewCalculator` und das Standard-Mapping nicht zum Einsatz.

6 Verschlüsseln von Konfigurationen

Eine weitere Anforderung, die im Beispiel-Szenario adressiert werden sollte, ist die Verschlüsselung von Konfigurationseinträgen. Der Spring Cloud Konfigurationsserver bietet dafür eine sehr einfache Möglichkeit. Voraussetzung um die Verschlüsselung von Spring Cloud Config zu benutzen ist „Java Cryptography Extension Unlimited Strength“, welches in der JVM installiert sein muss. Wird Docker verwendet, gibt es bereits vordefinierte Images.

Spring Cloud Config erlaubt sowohl die asymmetrische als auch die symmetrische Verschlüsselung von Konfigurationseinträgen. Da in der Dokumentation asymmetrische Verschlüsselung empfohlen wird, wird diese auch im Beispiel-Szenario verwendet. Dafür wurde ein Keystore mit dem Java-Keytool erstellt und wie in Code Beispiel 34 konfiguriert. Nach dieser Konfiguration bietet der Konfigurationsserver Endpunkte zum ver- (/encrypt) und entschlüsseln (/decrypt) an. Im Beispiel-Szenario wurde der Encrypt-Endpunkt benutzt um einen Konfigurationseintrag zu verschlüsseln. Der verschlüsselte Konfigurationseintrag kann anschließend im Konfigurationsserver verwendet werden indem dem verschlüsselten Eintrag „{cipher}“ vorangestellt wird (z.B. secret={cipher}AgBxrhudWswM9+qKhvDhHpDdPn4f1HN). Werden nun die Konfigurationseinträge vom Konfigurationsserver geladen werden die verschlüsselten

Einträge automatisch entschlüsselt. Die Services die vom Konfigurationsserver ihre Konfigurationen erhalten, müssen daher die Entschlüsselung nicht selbst vornehmen.

```
encrypt.keyStore.location=config-server.jks
encrypt.keyStore.password=y3WF9XbNFMH7Fapn
encrypt.keyStore.alias=config-server-key
encrypt.keyStore.secret=baKrS8X9saxuCwms
Code Beispiel 34 Keystore Konfiguration
```

Das Verschlüsselung im Beispiel-Szenario wurde auf Basis der folgenden Tutorials entwickelt:

- <https://www.baeldung.com/spring-cloud-configuration>
- http://cloud.spring.io/spring-cloud-config/2.0.x/single/spring-cloud-config.html#_encryption_and_decryption

7 Installationsanleitung

- I. **Starten von Keycloak:** Im Beispiel-Szenario wird die Standalone Variante von Keycloak verwendet. Diese kann von der Keycloak-Website heruntergeladen werden. Im Beispiel-Szenario läuft Keycloak auf Port 8180. Um dies zu konfigurieren muss bei der Einstellung "socket.binding.port" in der Datei "/standalone/configuration/standalone.xml" 8180 eingetragen werden. Weiteres wurden die Einstellungen des Beispiel-Szenarios exportiert. Diese Einstellungen können über den Menüeintrag "Import" importiert werden. Im Beispiel-Szenario wurde die Version 4.5.0 verwendet. Bei Verwendung von anderen Versionen kann es beim Import zu Problemen kommen.
- II. **User anlegen:** Da beim Export der Keycloak-Konfigurationen, die User nicht exportiert werden, müssen diese manuell angelegt werden. Es sollte für jede Benutzer-Rolle (Admin, Doctor, Assistent, Secretary) ein User angelegt werden. Die Rollen sind bereits im Export vorhanden und müssen nicht erneut erstellt werden. Durch die Zuordnung der Benutzer-Rollen sollten die User auch die entsprechenden Berechtigungen erben.
- III. **Config-Server starten:** Der Konfigurationsserver ist der Spring Service, der als erstes gestartet werden muss, da die anderen Services die Konfigurationen von diesem Service beziehen. Dieser Service befindet sich im Verzeichnis "config" im Github-Repository. Zum Starten dieses Service kann der Befehl "mvn spring-boot:run" benutzt werden. Dazu muss Maven installiert sein.
- IV. **Service Discovery starten:** Dieser Service befindet sich im Verzeichnis "eureka". Zum Starten dieses Service kann erneut der Befehl "mvn spring-boot:run" benutzt werden.
- V. **LaboratoryService starten:** Dieser Service befindet sich im Verzeichnis "laboratoryservice". Zum Starten dieses Service kann erneut der Befehl "mvn spring-boot:run" benutzt werden.
- VI. **PatientService starten:** Dieser Service befindet sich im Verzeichnis "patientservice". Zum Starten dieses Service kann erneut der Befehl "mvn spring-

boot:run" benutzt werden.

- VII.** Webclient starten: Um den Webclient lokal zu starten muss NodeJs und NPM installiert sein. Für den Webclient müssen die notwendigen Abhängigkeiten heruntergeladen sein. Zum Herunterladen der Abhängigkeiten wird im Verzeichnis "webclient\angular-client" der Befehl "npm install" ausgeführt. Anschließend kann die Applikation mit dem Befehl "ng-serve --open" gestartet werden.

8 Verwendungsbeispiel

In diesem Abschnitt wird ein Verwendungsbeispiel für das Beispiel-Szenario beschrieben, anhand welchem die umgesetzten Konzepte verdeutlicht werden sollen. Voraussetzung für dieses Anwendungsbeispiel ist, dass alle Services laufen und Keycloak entsprechend konfiguriert ist. Im Verwendungsbeispiel wird der HTTP-Client Postman (<https://www.getpostman.com/>) verwendet um mit den Spring-Services zu interagieren.

8.1 Method-Level Security

Mit Method-Level Security kann, wie bereits beschrieben, der Aufruf einer Methode untersagt werden. Um dieses Konzept zu testen benutzen wir den Laboratory-Service der im Beispiel-Szenario auf Port 8080 läuft. Dieser Laboratory-Service bietet einen HTTP-GET Endpunkt mit der URI „api/laboratory-results“ an um alle Laborergebnisse zu laden. Der Zugriff auf diesen Endpunkt ist nicht mit Websecurity (siehe Abschnitt 5.1) abgesichert und erlaubt daher zunächst auch Anonymen Zugriff. Durch den Aufruf dieses Endpunktes wird eine Service-Methode aufgerufen die wie in Code Beispiel 35 annotiert ist. Für den Aufruf dieser Methode ist die Berechtigung „READ_LABORATORY_RESULT“ notwendig.

```
@PreAuthorize("hasAuthority('READ_LABORATORY_RESULT')")
Collection<LaboratoryResult> getAllResults(Long patientSvnr);
Code Beispiel 35 Verwendungsbeispiel Method-Level Security
```

8.1.1 Unautorisierter Zugriff

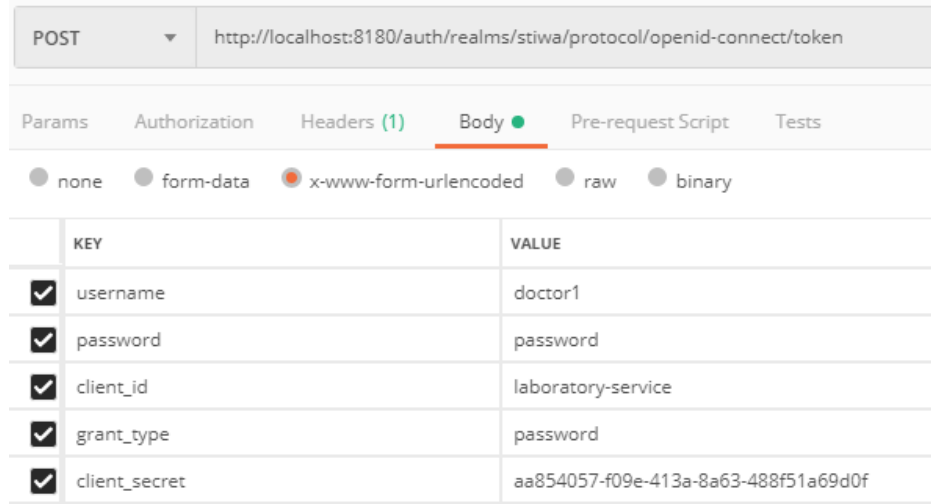
Um den unautorisierten Zugriff auf den Endpunkt des Laboratory-Service zum Laden aller Laborergebnisse zu Testen wird die URL „http://localhost:8080/api/laboratory-results“ mit Postman aufgerufen, ohne dass ein Authentication Token übergeben wird. Als Ergebnis dieses Aufrufs erhält man einen Response mit Status Code 401 (Unauthorized), da durch Aufruf der annotierten Service-Methode eine entsprechende Exception geworfen wird.

Wir die gleiche URL über den Browser aufgerufen, ohne dass der Browser gültige Authentifizierungsinformationen enthält wir der Browser zur Login-Seite von Keycloak weitergeleitet.

8.1.2 Erlaubter Zugriff

Um den Zugriff auf diesen Endpunkt zu authentifizieren muss ein Access-Token übergeben werden. Um einen gültigen Access-Token zu erhalten, kann Keycloak wie in Abbildung 4 gezeigt aufgerufen werden. In Abbildung 4 wird ein Access-Token für einen Benutzer mit der Rolle Doktor geladen. Durch diese Rolle wird auch die Berechtigung „READ_LABORATORY_RESULT“ geerbt, die notwendig ist um den beschriebenen

Endpunkt aufzurufen. Dieser Access Token wird beim Aufruf des Endpunktes als Bearer Token übergeben. Um diesen Request mit Postman durchzuführen gibt es einen Tab Authorization und in diesem Tab kann der Typ Bearer Token gewählt werden. Wird der Request mit dem beschriebenen Access Token durchgeführt erhält man einen Response mit Status Code 200 (OK) mit Body wie in Code Beispiel 36 ersichtlich.



| | KEY | VALUE |
|-------------------------------------|---------------|--------------------------------------|
| <input checked="" type="checkbox"/> | username | doctor1 |
| <input checked="" type="checkbox"/> | password | password |
| <input checked="" type="checkbox"/> | client_id | laboratory-service |
| <input checked="" type="checkbox"/> | grant_type | password |
| <input checked="" type="checkbox"/> | client_secret | aa854057-f09e-413a-8a63-488f51a69d0f |

Abbildung 4 Laden eines Access Token

```
[
  {
    "id": 1,
    "valueA": 123,
    "valueB": 456,
    "valueC": "Test1",
    "patientSvnr": 123401011990
  },
  {
    "id": 2,
    "valueA": 321,
    "valueB": 654,
    "valueC": "Test2",
    "patientSvnr": 123401011990
  },
  ...
]
```

Code Beispiel 36 Method-Level Security OK-Response

8.1.3 Verbotener Zugriff

Beim vorherigen Aufruf des Endpunktes wurde ein Access Token für einen Benutzer mit der Rolle Doktor verwendet. Da die Rolle Doktor die notwendige Berechtigung für diesen Endpunkt erbt war der Aufruf zulässig. Wird derselbe Aufruf mit einem Access Token für einen Benutzer mit der Rolle Sekretär ausgeführt, erhält man einen Response mit Status Code 403 (Forbidden), da die Rolle Sekretär die entsprechende Berechtigung nicht erbt.

8.2 Field-Level Security

Mit der Field-Level Security kann, wie bereits beschrieben, die Sicht auf ein Domänen-Objekt basierend auf Rollen und Berechtigungen eingeschränkt werden. Um dies zu

Testen wird erneut der Endpunkt des Laboratory-Service zum Laden aller Laborergebnisse verwendet. Das Domänen Objekt ist wie in Code Beispiel 30 annotiert. Die Attribute „valueC“ und „valueD“ sind nur für Benutzer mit bestimmten Rollen und Berechtigungen sichtbar. Welche Benutzer diese Attribute sehen dürfen wird mit einer View-Policy beschrieben (siehe Code Beispiel 33). „valueD“ darf nur von Benutzern mit einer Rolle Admin gelesen werden und „valueC“ nur von Benutzern mit der Berechtigung „READ_EXTENDED_LABORATORY_RESULT“. Wird nun der beschriebene Endpunkt mit einem Access Token für einen Benutzer mit der Rolle Assistent ausgeführt erhält man einen Response mit Status Code 200, weil Assistenten für diesen Endpunkt berechtigt sind. Die View auf das Domänen-Objekt ist allerdings eingeschränkt, weil Assistenten nicht die Berechtigung haben das Attribute „valueC“ zu sehen (vergleiche Code Beispiel 36 mit Code Beispiel 37).

```
[
  {
    "id": 1,
    "valueA": 123,
    "valueB": 456,
    "patientSvnr": 123401011990
  },
  {
    "id": 2,
    "valueA": 321,
    "valueB": 654,
    "patientSvnr": 123401011990
  },
  ...
]
```

Code Beispiel 37 Fiel-Level Security Response

8.3 Service zu Service Kommunikation

Zum Testen der Service zu Service Kommunikation wird der HTTP-GET Endpunkt des Patient-Service, um einen Patienten zu laden, aufgerufen, da dieser alle Laborergebnisse des Patienten vom Laboratory-Service lädt. Diese Service zu Service Kommunikation wurde so definiert, dass der eingehende Access Token des Patient-Service an den Laboratory-Service weitergegeben wird. Code Beispiele 38-40 zeigen die Ergebnisse des Aufrufs von `http://localhost:8081/api/patients/123401011990` mit unterschiedlichen Benutzerrollen. In Code Beispiel 38 ist das Attribute „laboratoryResult“ leer, da die Rolle Sekretär nicht die Berechtigung hat Laborergebnisse zu laden. In Code Beispiel 39 ist das Attribute „laboratoryResult“ befüllt, die Werte „valueC“ und „valueD“ sind nicht enthalten weil Assistenten nicht die Berechtigung für diese Attribute haben. In Code Beispiel 40 ist zusätzlich „valueC“ der Laborergebnisse und die Blutgruppe des Patient im Response ersichtlich.

```
{
  "firstName": "Marcel",
  "lastName": "Lange",
  "tel": "0676 640 57 77",
  "svnr": 123401011990,
  "address": "Gartenweg 39, 4212 Albingdorf",
  "gender": "Male",
  "laboratoryResults": null
}
```

Code Beispiel 38 Patient Service Response Rolle Sekretär

```
{
  "firstName": "Marcel",
  "lastName": "Lange",
  "tel": "0676 640 57 77",
  "svnr": 123401011990,
```

```

"address": "Gartenweg 39, 4212 Albingdorf",
"gender": "Male",
"laboratoryResults": [
  {
    "id": 1,
    "valueA": 123,
    "valueB": 456,
    "patientSvnr": 123401011990
  },
  {
    "id": 2,
    "valueA": 321,
    "valueB": 654,
    "patientSvnr": 123401011990
  }
]
}

```

Code Beispiel 39 Patient Service Response Rolle Assistent

```

{
  "firstName": "Marcel",
  "lastName": "Lange",
  "tel": "0676 640 57 77",
  "svnr": 123401011990,
  "address": "Gartenweg 39, 4212 Albingdorf",
  "bloodGroup": "A+",
  "gender": "Male",
  "laboratoryResults": [
    {
      "id": 1,
      "valueA": 123,
      "valueB": 456,
      "valueC": "Test1",
      "patientSvnr": 123401011990
    },
    {
      "id": 2,
      "valueA": 321,
      "valueB": 654,
      "valueC": "Test2",
      "patientSvnr": 123401011990
    }
  ]
}

```

Code Beispiel 40 Patient Service Response Rolle Doktor