Masterarbeit

im Rahmen des Studiengangs

Technomathematik

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 – Medizintechnik und Technomathematik

Entwicklung eines tokenbasierten Authentifizierungs- und Autorisierungssystems für eine verteilte Dateninfrastruktur

Jülich, den 25. September 2018

Tobias Korf

Diese Arbeit wurde betreut von:
Prof. Dr. rer. nat. Bodo Kraft
Dr. Ralf Kunkel

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Masterarbeit mit dem Thema „Entwicklung eines tokenbasierten Authentifizierungs- und Autorisierungssystems für eine verteilte Dateninfrastruktur“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Name: _____

Jülich, den _____

Unterschrift der Studentin / des Studenten

Diese Arbeit wurde betreut von:
1. Prüfer: Prof. Dr. rer. nat. Bodo Kraft
2. Prüfer: Dr. Ralf Kunkel
Betreuer: Jürgen Sorg

Sie wurde im IBG-3 der Forschungszentrum Jülich GmbH konzipiert.

Fachhochschule Aachen

Thema der Masterarbeit:	Entwicklung eines tokenbasierten Authentifizierungs- und Autorisierungssystems für eine verteilte Dateninfrastruktur
Verfasser:	Tobias Korf

Abstract

In der vorliegenden Arbeit wird ein Konzept und eine Implementierung zur Authentifizierung und Autorisierung von Benutzern und Ressourcen mit Hilfe des OAuth2-Protokolls in einer verteilten Dateninfrastruktur vorgestellt. Die Implementierung erfolgt in der Programmiersprache Java unter Zuhilfenahme des Spring-Frameworks. Das OAuth2-Protokoll wird ausführlich erläutert und die notwendigen Anpassungen an der bestehenden Dateninfrastruktur werden für alle definierten Entitäten des Protokolls ausgeführt und dokumentiert. Es wird darauf eingegangen welche Sicherheitsmaßnahmen sind erforderlich und wie diese am konkreten Beispiel zu implementieren sind.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 OAuth 2 . . . 2
2.2 Spring . . . 5
2.3 SOAP . . . 6
2.4 Samplify . . . 7
2.5 SOS . . . 7
2.6 Data Discovery Portal . . . 8

3 Anforderungsanalyse **9**
3.1 Authorisierungsserver . . . 9
3.2 Resourceservers . . . 10
3.3 Clients . . . 10
3.3.1 Clisos . . . 12
3.3.2 Samplify . . . 13
3.3.3 Data Discovery Portal . . . 14

4 Entwicklung **15**
4.1 Entwicklungsumgebung . . . 16
4.2 Autorisierungsserver . . . 16
4.2.1 Login . . . 24
4.2.2 Index . . . 25
4.2.3 Registrierung . . . 29
4.2.4 Nutzerverwaltung . . . 34
4.2.5 Tokenerzeugung für temporäre Nutzer . . . 35
4.2.6 Statistiken . . . 37
4.2.7 Token Informationsinhalt . . . 40
4.3 SOAP-Dienst . . . 41
4.4 Samplify . . . 44
4.5 SOS . . . 48
4.6 Clisos . . . 53
4.7 Data Discovery Portal . . . 54
4.8 Sicherheit . . . 55

5 Zusammenfassung und Ausblick **57**

6 Glossar **58**
6.1 PostgreSQL & Postgis 58
6.2 Maven 59
6.3 Thymeleaf 59
6.4 HTTP 59
6.5 Captcha 59
6.6 Argon 2 60
6.7 zxcvb 60
6.8 JAX-WS 60
6.9 GWT 61
6.10 Docker 61
6.11 LDAP 61
6.12 DSGVO 61

7 Anhang **62**

Literaturverzeichnis **81**

Abbildungsverzeichnis

2.1 OAuth2 Flow 4
2.2 Aufbau einer SOAP-Nachricht 6
2.3 UML Sequenzdiagramm - Datenabruf SOS 7

3.1 UML Use Case Diagramme - Clisos 12
3.2 UML Use Case Diagramme - Samplify 13
3.3 UML Use Case Diagramme - Data Discovery Portal 14

4.1 Übersicht der einzelnen Komponenten 15
4.2 ER-Diagramm der OAuth2 Datenbank - Listing 7.1-7.6 17
4.4 Ansicht der Login-Seite 25
4.5 LoginController - Klassendiagramm 28
4.6 RegistrationController - Klassendiagramm 34
4.7 Aufbau der Downloads 38
4.8 UML-Sequenzdiagramm Statistikerfassung 38
4.9 Quelle: [9] Abb.1 - Der Aufbau eines SOS 49
4.10 Data Discovery Portal 55

7.1 Ablauf des *authorization-code*-Verfahrens 67
7.2 Ansicht der Index-Seite 72
7.3 Ablauf der Nutzerregistrierung 75

Listingverzeichnis

4.1 XML: Maven-Abhängigkeiten für OAuth 2 16
4.2 application.yml (OAuth2 Datenbank) 17
4.3 DatabaseConfig.java (OAuth2 Datenbankkonfiguration) 17
4.4 application.yml (Hikari Datenbank konfiguration 18
4.5 AuthorizationServerConfiguration.java Konstruktor 18
4.6 AuthorizationServerConfiguration.java ClientDetailService 19
4.7 AuthorizationServerConfiguration.java TokenStore 19
4.8 AuthorizationServerConfiguration.java ApprovalStore 19
4.9 AuthorizationServerConfiguration.java AuthorizationCodeServices . . 19
4.10 AuthorizationServerConfiguration.java ClientDetailService configure . 19
4.11 XML: Maven-Abhängigkeiten für Spring Security 20
4.12 WebSecurityConfiguration.java HttpSecurity 21
4.13 AuthorizationServerConfiguration.java Sicherheitseinstellung 21
4.14 application.yml (ActiveDirectory) 22
4.15 WebSecurityConfiguration.java ActiveDirectory 22
4.16 WebSecurityConfiguration.java OAuth2 Datenbank 22
4.17 User.java Annotationen 23
4.18 LoginController Login Methodenkörper 24
4.19 LoginController Index-Mapping 25
4.20 LoginController Approvals und Token 26
4.21 LoginController Model füllen 26
4.22 LoginController Version 27
4.23 LoginController Bestätigung zurücknehmen 27
4.24 LoginController InitBinder 28
4.25 RegistrationController Register-GET 29
4.26 RegistrationController POST-Methode 30
4.27 RegistrationController Captcha-Überprfüng Teil 1 30
4.28 RegistrationController Captcha-Überprfüng Teil 2 30
4.29 application.yml Email-Konfiguration 31
4.30 EmailService.java 31
4.31 Argon2PasswordEncoder.java 32
4.32 Argon2PasswordEncoder.java - Iterationen berechnen 33
4.33 WebSecurityConfiguration.java PasswordEncoder 33
4.34 RegistrationController - Confirm POST 33
4.35 RegistrationController - edit 35
4.36 RegistrationController - Temporärer Benutzer 35

4.37 RegistrationController - Generierung des Access-Token 36
4.38 TemporaryTokenService 36
4.39 CustomAuthenticationKeyGenerator Temporaryuser 37
4.40 CustomAuthenticationKeyGenerator SHA-256 37
4.41 AuthorizationServerConfiguration.java Endpunkt-Konfiguration . . . 37
4.42 StatisticsController.java Annotationen 39
4.43 StatisticsController.java Security 39
4.44 StatisticsController.java *addDownload*-Methode 40
4.45 CustomAccessTokenConverter.java 41
4.46 SoapOdmServiceFacade.java Generierung des *Principal*-Objekts . . . 41
4.47 SoapOdmServiceFacade.java WebServiceContext 41
4.48 Constants.java 42
4.49 AuthenticationMethod.java Enum 42
4.50 ActiveDirectoryOAuthAuthentication.java getAccount 43
4.51 AbstractAuthentication.java 43
4.52 Constants.java mit OAuth2 44
4.53 SoapOdmServiceFacade.java WebServiceContext 44
4.54 OAuth2Controller.java OAuthClientRequest 46
4.55 OAuth2Controller.java Response-Objekt 46
4.56 Json-Objekt 46
4.57 SoapOdmServiceFacadeOauthWrapper.java 47
4.58 RequestOperator.java Anfragetypen 49
4.59 RequestOperator.java getObservation 50
4.60 RequestOperator.java recieveRequest 51
4.61 GetObservationListener.java Wrapper-Klasse 51
4.62 Benachrichtigung über Download 52
4.63 HttpPostDownload.java 53
4.64 clisos.py *Access Token* Übergabe 53
4.65 sosclient.py *Authorization*-Header 53
4.66 DisclamerDialog.java 54
4.67 SensorObservationService.java 54
4.68 SensorObservationServiceAsync.java 55
4.69 WebSecurityConfiguration.java SSL/TLS erzwingen 56

7.1 SQL: OAUTH_ACCESS_TOKEN 62
7.2 SQL: OAUTH_CLIENT_DETAILS 62
7.3 SQL: OAUTH_REFRESH_TOKEN 62
7.4 SQL: OAUTH_CODE 62
7.5 SQL: OAUTH_APPROVALS 63
7.6 SQL: OAUTH_CLIENT_TOKEN 63
7.7 HTML: login.html 63
7.8 JAVA: TokenMapper.java 65
7.9 HTML: index.html 66
7.10 HTML: index.html Client Verwaltung 70

7.11 HTML: register.html 71
7.12 HTML: confirm.html 73
7.13 SoapOdmServiceFacade.java extractAccountInfoFromSoapHeader . . 74
7.14 SoapOdmServiceFacade.java doGet-Methode 77

1 Einleitung

Immer mehr Webseiten-Betreiber bieten die Möglichkeit Benutzer über bereits bestehende Benutzerkonten großer Anbieter, wie Google oder Facebook, zu authentifizieren. Zum einen entfallen dadurch die Verwaltungskosten und der Aufwand Zugangsdaten zu schützen. Zum anderen etabliert sich dadurch die eindeutige digitale Identität eines Benutzers im Internet. Ein bekanntes Beispiel für die dezentrale Authentifizierung von Nutzern ist das eduroam-Netzwerk, welches schon seit Längerem von vielen in der akademischen Gemeinschaft genutzt wird. In dieses Netzwerk können sich Nutzer aller teilnehmenden Einrichtungen mit den bekannten Zugangsdaten anmelden und haben dadurch direkten Zugriff auf Dienste, die dort angeboten werden (bspw. Internet). Die Authentifizierung wird an den jeweilige Heimatserver delegiert und dort ohne Anlegen neuer Zugangsdaten durchgeführt.
Mit dem OAuth2-Protokoll wurde ein offener Standard entwickelt, der einen allgemeinen implmentierungsunabhängigen Authentifizierungsmechanismus für diese Anwendungsfälle schafft.
Darin werden unter anderem Rollen definiert (Resource Owner, Resource Server, Authorization Server und Clients), die Softwarekomponenten sowie Benutzer einnehmen. Darüber hinaus wird festgelegt, wann und wie diese Rollen Informationen austauschen müssen um eine sichere und zuverlässige Authentifizierung gewährleisten zu können.

In dieser Arbeit wird zum einen gezeigt, wie das OAuth2-Protokoll in einer verteilten Dateninfrastruktur eingesetzt werden kann und wo die Vorteile dabei liegen. Zum anderen wird eine Implementierung mithilfe des Spring-Frameworks vorgeschlagen und detailliert erklärt.

2 Grundlagen

2.1 OAuth 2

OAuth2 (Open Authorization 2) ist offenes Protokoll, welches sichere und standardisierte Autorisierungen über eine API zulässt. OAuth2 definiert vier Rollen:

Resource Owner Eine Entität, welche das Recht hat, Zugriffe auf eine geschützte Ressource zu gewähren. Im Normalfall handelt es sich bei dem *Resource Owner* um einen Benutzer, welchem die Ressourcen gehören und durch einen

Resource Server angeboten werden. Dieser Dienst kann mithilfe eines übertragenen *Access-Token* Zugriffe auf eine Ressource realisieren. Der Token repräsentiert die Zustimmung des *Resource Owners* und berechtigt einen

Client dazu, auf die eine geschützte Ressource zuzugreifen. Ein *Client* ist eine beliebige Applikation bzw. Anwendung. Sie muss nicht zwangsläufig vertrauenswürdig sein und kann aus Fremdquellen stammen. Zuletzt gibt es noch den

Authorization Server, der den ganzen Prozess orchestriert. Er authentifiziert den *Resource Owner* und stellt *Access-Token* für bestimmte Anwendungsbereiche (Scopes) und *Clients* aus.

Es gibt zwei verschiedene Arten von Token. Der

Access-Token dient dazu, auf geschütze Inhalte zuzugreifen. Er wird vom *Authorization Server* ausgestellt und muss in einer Anfrage vom *Client* mitgeschickt werden, um diese zu autorisieren. Über den Scope-Parameter können die Berechtigungen des *Access Token* bei der Erteilung eingeschränkt bzw. definiert werden. Typische Scopes sind *read* oder *read,write*, um zu signalisieren, dass mit dem vorliegenden *Access Token* lesende bzw. lesende und schreibende Aktionen durchgeführt werden dürfen. Daneben gibt es noch den optionalen

Refresh-Token, welcher bei der Erzeugung des *Access Token* mit übertragen werden kann. Normalerweise hat ein *Access Token* nur eine limitierte Gültigkeitsdauer. Wenn diese überschritten wurde, muss der Autorisierungsprozess erneut durchgeführt werden. Mit dem *Refresh Token* kann man einen neuen *Access Token* beim *Authorization Server* generieren lassen. Der *Refresh Token* hat allerdings ebenfalls eine begrenzte Lebensdauer. Diese fällt jedoch meist wesentlich länger aus als die des *Access Token*.

Wir gehen in dem folgenden Beispiel eines Protokollflusses davon aus, dass ein Client auf eine geschützte Ressource zugreifen möchte. Dieser Zugriff wird in sechs Schritten gewährt.

1. Es wird eine Autorisierungsanfrage an den *Ressource Owner* gestellt.
2. Der *Ressource Owner* autorisiert die Anfrage und erteilt somit eine Genemigung.
3. Die Genemigung wird an den *Authorization Server* weitergeleitet.
4. Der *Authorization Server* stellt einen *Access Token* aus.
5. Der Client schickt den *Access Token* an den *Ressource Server*.
6. Der *Resource Server* antwortet mit den geschützten Daten.

Das OAuth2-Protokoll spezifiziert nicht, wie der *Access Token* vom *Resource Server* überprüft wird. In der Praxis stellt entweder der *Authorization Server* eine Schnittstelle bereit, über die der *Access Token* verifiziert werden kann, oder der Token enthält alle notwendigen Daten, welche dann selbständig vom *Resource Server* verifiziert werden. Im ersten Fall erhöht man die Last auf dem *Authorization Server*, da jede Anfrage hier verifiziert werden muss. Eine Möglichkeit, dieser Problematik zu entgegenzuwirken ist es, die Antwort zwischenzuspeichern und somit die Last pro *Client* zu reduzieren. Man gewinnt durch dieses Verfahren die Sicherheit, dass der Token noch gültig ist und in der Zwischenzeit nicht, beispielsweise widerrufen wurde.
Der gezeigte Protokollfluss muss nicht zwingend auf alle Anwendungsfälle zutreffen. Um dieses Problem zu lösen, wurden vier Genehmigungsprozesse, mit der Möglichkeit zur Erweiterung, standardisiert.

authorization code Entspricht dem oben gezeigten Beispiel.

resource owner password credentials Hier hat der *Client* direkten Zugriff auf die Zugangsdaten des *Resource Owners*. Die ersten beiden Schritte aus dem Beispiel entfallen somit. Dieses Verfahren sollte nur dann angewendet werden, wenn es sich bei dem *Client* um eine vertrauenswürdige, first-party[1] Anwendung handelt.

implicit Ist prinzipiell das gleiche Verfahren wie das *authorization code*-Verfahren. Jeder Client bekommt eine *ID* und ein *secret* zugewiesen, damit er sich gegenüber dem Authorization Server authentifizieren kann. Sollte eine Anwendung allerdings beispielsweise Quelloffen sein, so ist auch das *secret* nicht mehr geheim. Aus dem Grund wird in diesem Verfahren auf das *secret* verzichtet.

client credentials Hier nimmt ein *Client* die Rolle des *Resource Owners* ein, welcher die Berechtigung um auf die Ressource zuzugreifen. Daher wird keine Autorisierung durch einen Nutzer benötigt. Die Angabe von Client-*ID* und Client-*Secret* genügt um einen *Access Token* zu erhalten.

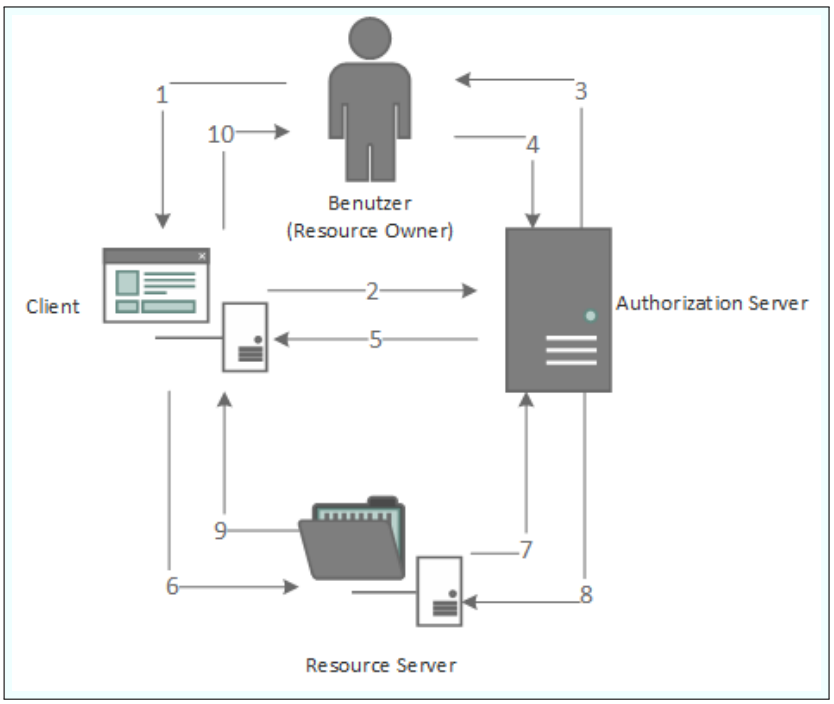


Abbildung 2.1: OAuth2 Flow

[4]
Abbildung 2.1 zeigt den kompletten Ablauf einer OAuth2 geschützten Anfrage.

1. Der Nutzer möchte ein Programm nutzen.
2. Der Client stellt eine *authorization-code*-Anfrage.
3. Der Client leitet den Nutzer weiter an den *Authorization Server* und wird von diesem aufgefordert sich einzuloggen.
4. Dort loggt sich der Nutzer ein und autorisiert die Anfrage.
5. Der *Authorization Server* stellt einen *Access Token* aus und schickt diesen an den Client.
6. Der Client schickt die Anfrage des Nutzers an den *Resource Server* und schickt dabei den *Access Token* mit.
7. Der *Resource Server* schickt den *Access Token* an den *Authorization Server*.
8. Der *Authorization Server* schickt das *Principal*-Objekt und verifiziert somit die Anfrage.

[1] Aus eigener Entwicklung

9. Der *Resource Server* führt die Anfrage aus und antwortet mit den angeforderten Daten.

10. Der Client stellt das Ergebnis der Anfrage dar.

2.2 Spring

Spring ist ein quelloffenes Framework für die Java-Plattform. Es soll das Programmieren vereinfachen und dadurch die Anwendung von guten Programmierpraktiken fördern. [5] Es ist Modular und verhindert dadurch, dass große Mengen unbenutzten Codes in das eigentliche Programm einfließen. Benötigte Komponenten können somit einfach eingebunden oder selber entwickelt werden. Spring MVC bietet beispielsweise Funktionalitäten, wie REST Endpunkte, Vorlagen-Prozessierung und verknüpft diese mit JavaScript über das AngularJS-Framework. Andere weit verbreitete Module sind:

- Spring Core
- Spring Security
- Spring OAuth
- Spring Data
- Spring Cloud
- Spring Web MVC

Das Core-Framework stellt dabei die Basis dar und enthält die Implementierungen von Kernfeatures wie die *dependency-injection* , die Unterstützung von Web-Komponenten für Spring (MVC[2]), Datenbankzugriffe usw.
Web MVC bietet die Ünterstützung für Web Applikationen, welche die Servlet API benutzen und so in Container-Diensten eingesetzt werden können.
Spring Data bietet die Integration der Datenhaltung für das Programm. Es werden viele verschiedene Datenbanktechnologien unterstützt. Das Ziel in diesem Modul liegt darin, den Code rund um die Datenhaltung zu komprimieren und auf das Wichtigste zu beschränken. So existieren beispielsweise viele Interfaces, in denen lediglich Anotationen angegeben werden müssen bzw. Methoden definiert werden müssen, deren Implementierung Spring automatisch über Proxy-Elemente durchführt. So reicht es beispielsweise aus, ein Interface von JPARepository<Objekttyp> erben zu lassen, um eine funktionsfähige CRUD[3]-Schnittstelle zu einer belebigen Datenbank zu erzeugen. Weiterhin kann man durch die Benennung von Methoden SQL-Anfragen abbilden:

Eine Methodedeklaration
Collection<Objekttyp> findAllByCounterGreaterThan(int zahl);
, in dem zuvor erzeugten Interface, sucht alle Einträge heraus, in der die Variable *counter* größer als die übergebene Zahl ist und gibt diese als Liste des definierten Typ zurück.[8]

Dependency-Injection ist ein Kernpunkt von Spring. Hierzu werden drei verschiedene Methoden angeboten.

[2]Model-View-Controller
[3]create,retrieve,update and delete

1. **Field-Injection**

2. **Setter-Injection**

3. **Contructor-Injection**

Field-Injection ist hierbei die einfachste und sauberste Lösung, allerdings sollte auf diese Methode verzichtet werden. Durch eine Annotation einer Klassenvariable wird hier das entsprechende Objekt injiziert. Nach außen werden so allerdings die Abhängigkeiten verborgen und die Initialisierung des Objekts ist außerhalb der Spring-Umgebung nur schwer möglich. Für Test beispielsweise ist dies aber unbedingt nötig. Aus diesem Grund sollte für obligatorische Abhängigkeiten die konstruktobasierte Autoinjizierung verwendet werden und für optionale Abhängigkeiten die setterbasierte Autoinjizierung. [10]

2.3 SOAP

SOAP ist ein Protokoll, welches komplexe Datenobjekte ohne Verluste in Textform abbildet (serialisiert), transportiert und anschließend rekonstruiert. Zur Serialisierung wird normalerweise auf die XML-Repräsentation zurückgegriffen. Ähnlich zu HTTP gibt es ein **Header-** und **Body-Element**, welche durch einen **SOAP-Envelope** umfasst werden. In dem optionalen Header-Element werden Metadaten wie zum Beispiel Authentifizierungsinformationen übertragen. Im obligatorischen *Body-Element* werden die eigentlichen Daten oder ein **SOAP-Fault** Element übertragen, welches einen aufgetretenen Fehler des Aufrufs umschreibt. Neben der Möglichkeit, Daten zu übertragen, ist die Hauptaufgabe von SOAP die RPC[4]-Funktionalität. Mit anderen Worten den entfernten Aufruf von Methoden.

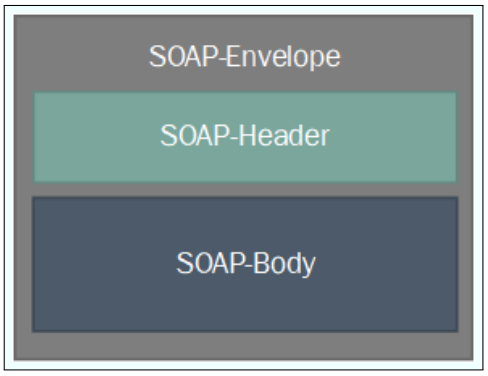


Abbildung 2.2: Aufbau einer SOAP-Nachricht

[4]Remote Procedure Call

2.4 Samplify

Samplify ist eine Desktopanwendung, die zur Pflege von Daten und Kennzeichnung von Proben entwickelt wurde. Sie kommuniziert zu diesem Zweck mit einem SOAP-Dienst, welcher die Schnittstelle zu einer Datenbank darstellt. Bei jeder Anfrage an den Dienst wird der Benutzername und das Passwort des Nutzers mitgeschickt, um dann im Dienst, mithilfe des lokalen Verzeichnisdienst, authentifiziert und lokal autorisiert wird.

2.5 SOS

Ein *Sensor Observation Service* ist ein standardisierter Webdienst zum Anfragen, Filtern und Ausliefern von Sensordaten und Sensorbeschreibungen. Er ermöglicht einen einheitlichen Zugriff auf Zeitreihendaten, die von Sensoren gemessen wurden. Der Dienst bietet dabei einen homogenen Zugriff auf heterogene Beobachtungs- und sensorbezogene Metadaten über definierte, einheitliche Schnittstellen und Operationen. Hierbei werden drei obligatorische Kernoperatoren *GetCapabilities*, *DescribeSensor* und *GetObservation* definiert.[11]

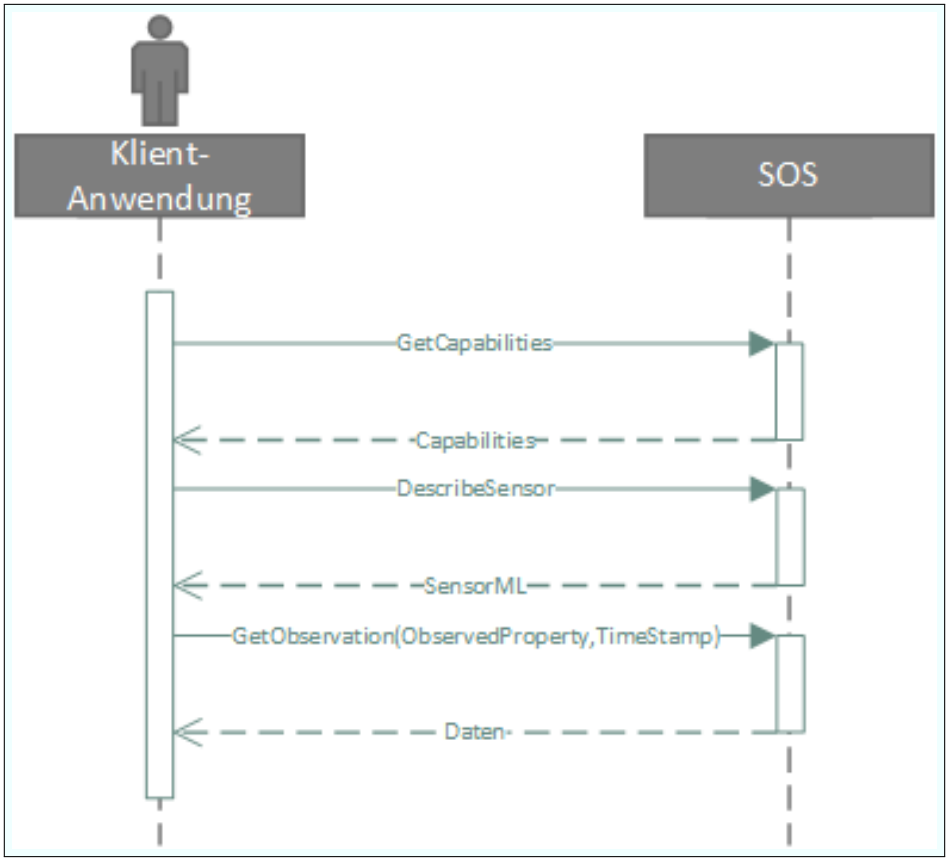


Abbildung 2.3: UML Sequenzdiagramm - Datenabruf SOS

In Abbildung 2.3 wird der Ablauf einer Abfrage von Daten über einen *SOS* gezeigt. Zunächst werden über die Möglichkeiten, welche der Dienst zur Verfügung stellt, durch die *GetCapabilities*-Anfrage abgerufen. In der Antwort werden, unter Anderem, alle verfügbaren Sensoren definiert.
Danach werden die Beschreibungen der gewünschten Sensoren über die *DescribeSensor*-Anfrage abgerufen. Die Antwort hat das *SensorML*-Format.
Mit diesen Informationen kann eine *GetObservation*-Anfrage erzeugt und gestellt werden. Das Ergebnis ist eine Zusammenstellung der gewünschten Datenwerte.

SensorML (Sensor Model Language) ist ein XML-Modell, welches zur Beschreibung von Sensoren benutzt wird. Der Messprozess und die Weiterverarbeitung der Sensordaten kann hier beschrieben werden. Auch andere Metadaten, wie die zeitlichen und räumlichen Gegebenheiten, können hier modelliert werden.[9]

2.6 Data Discovery Portal

Das Daten Discovery Portal (DDP) ist eine auf der Servlet-Spezifikation der Java-EE-Technologie basierte Webapplikation, dass standardisiert heterogene Daten von OGC konformen Sensor Observation Services sowie verschiedene Metadatenkatalogen abruft und in einer einheitlichen Form Benutzern zur Verfügung stellt. Es bildet damit die öffentliche Benutzerschnittstelle zu einer verteilten Dateninfrastruktur, die im Fall des TERENO Projekts mehrere Helmholtzzentren umfasst.

3 Anforderungsanalyse

Die Authentifizierung und Autorisierung aller Software-Komponenten sollen gebündelt an einer Stelle ablaufen.
Eine weitere Anforderung stellt die Erfassung von Downloads und Statistiken über die *SOS*-Dienste des Instituts dar. Die Persistierung soll nun nicht mehr jährlich anhand von Emails geschehen, sondern direkt während des Download-Vorgangs erfolgen.
Zur Realisierung dieses Projekts werden mehrere voneinander getrennte Komponenten benötigt. Wir sprechen also von einem **verteilten System**. Die Kernkomponente stellt der OAuth2-*Authorization Server* dar. Er bekleidet die Schlüsselrolle der Autorisierung. Die Authentifizierung wird von verschiedenen Verzeichnisdiensten übernommen. Alle anderen Komponenten müssen diesem Dienst also „vertrauen“. Folglich muss die komplette Kommunikation über **TLS/SSL** abgesichert werden. Die entsprechenden Zertifikate stellt eine *vertrauenswürdigen Zertifizierungsstelle*[1] aus. Neben dem *Authorization Server* gibt es noch die Rollen des *Client*, *Resource Server* und *User*.

3.1 Authorisierungsserver

Die Implementierung dieser Komponente erfolgt in der Programmiersprache Java 1.8.
Das Spring-Framework wird dabei zu Hilfe genommen.
Dieser Teil soll mehrere Aufgaben übernehmen. Diese Aufgaben sollen durch den *Autorisierungsserver* abgedeckt werden:

Die Authentifizierung der Nutzer soll an den jeweiligen LDAP-Dienst delegiert werden Es gibt mehrere Server mit verschiedenen Nutzern. Der AS soll von allen Nutzern gleichermaßen benutzt werden können.

Nutzer anlegen und verwalten Nutzer sollen sich registrieren, die eigenen Daten einsehen und verwalten, sowie Access-Tokens generieren können.

Das OAuth 2 Protokoll soll verwendet werden Alle Funktionen des OAuth2-Protokolls sollen implementiert werden.

Download-Statistiken sollen gespeichern werden. Sobald Daten über einen der SOS-Dienste heruntergeladen werden, sollen Informationen, wie Datum, Anzahl der Datenwerte, Filter, Verwendungszweck, Name des Nutzers, Email-Adresse des Nutzes und die Messstelle erfasst und in der Datenbank abgelegt werden.

[1] Certificate Authority

Temporäre Nutzer sollen anlegt werden können

Access-Token sollen automatisch generiert und per Mail verschickt werden Wenn ein Nutzer ohne Account Daten herunterladen will, soll ein Access-Token erstellt werden, der an die angegebene Email-Adresse geschickt wird. Hiermit wird sichergestellt, dass die Email-Adresse gültig ist. Der Token soll für eine Woche gültig sein und stellt eine Verknüpfung zwischen dem Download und dem Nutzer her, sodass der Prozess erfasst werden kann.

Die Produktiv-Datenbank soll genutzt werden (PostgreSQL) Die Statistiken und Nutzerdaten sollen in der gleichen Produktivdatenbank abgelegt werden, wie die Datenwerte auch. Diese Anforderung vereinfacht das Verknüpfen der Daten innerhalb der Datenbank. (s. Kapitel 6.1)

Der Nutzer soll einheitlich autorisiert werden Egal, mit welcher Quelle der Nutzer authentifiziert wird, sollen die Rechte einheitlich definiert sein. Somit wird sichergestellt, dass alle Ressourcen-Server mit allen Nutzern kompatibel sind.

Persönlichen Daten sollen einsehbar sein und über ein Webinterface angezeigt werden.

Bestätigte Clients sollen entfernt werden können und

Aktive Access Token sollen dargestellt werden.

Administratoren sollen neue Clients registrieren können.

3.2 Resourceservers

Diese Softwarekomponenten existieren bereits und müssen für die Verwendung des OAuth2 Protokoll angepasst undt erweitert werden. Wie im vorherigen Kapitel wurde ebenfalls Java 1.8 verwendet.
Zunächst sollen alle SOS-Dienste sowie der SOAP-Dienst in das neue System migriert werden. Hierzu muss die Autorisierung und Authentifizierung an den *Authorization Server* ausgelagert werden. Die Antwort muss entsprechend verarbeitet werden.

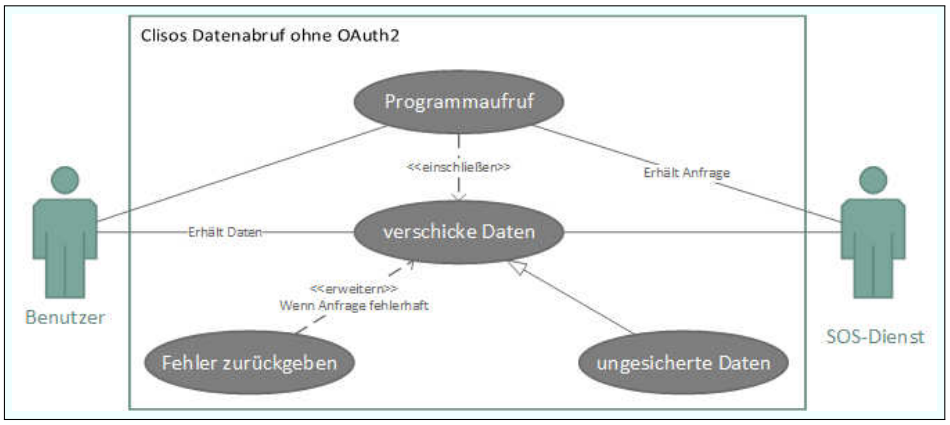
3.3 Clients

Wir beschränken uns zunächst auf drei Klient-Programme. Das **DDP**, **Samplify** und **Clisos**. Die ersten Beiden sind Java-Programme. *Clisos* ist ein in Python geschriebenes Kommandozeilen-Werkzeug, welches mit den *SOS*-Diensten kommuniziert. *Samplify* muss die Möglichkeit haben, einen *Access-Token* über das *password authentication*-Verfahren zu erhalten. Dieser muss gespeichert werden und bei allen Anfragen an den Dienst angegeben werden. Bei der Generierung des Token muss ein Gültigkeitszeitraum übertragen werden, damit das Programm weiß, wann ein neuer

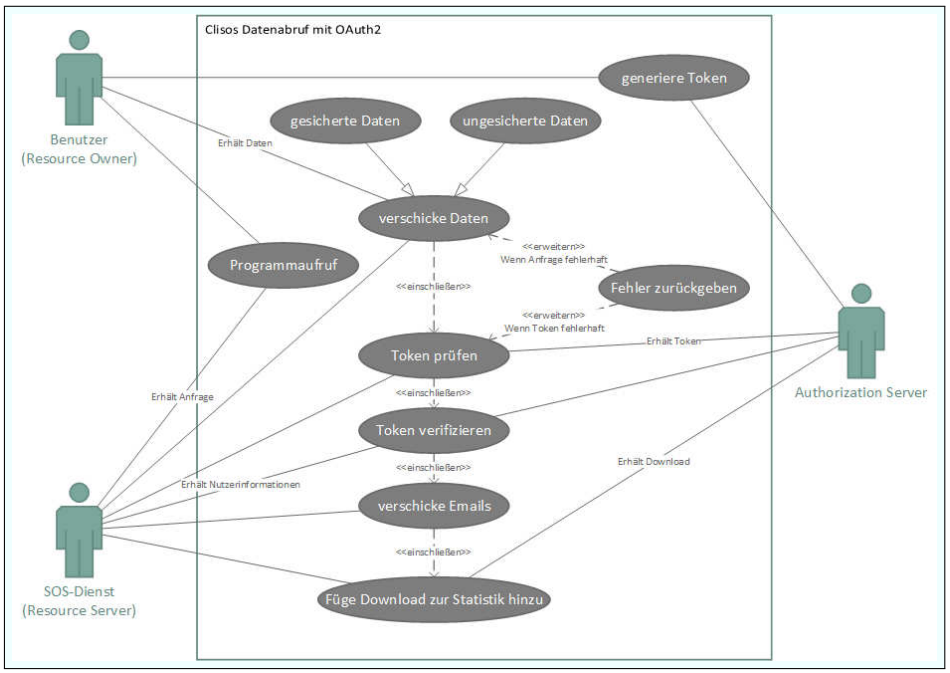
Token benötigt wird. Zusätzlich muss bei jeder Anfrage auf OAuth2-Fehler geprüft und entsprechend auf diese reagiert werden.
Bei *Clisos* muss ein Parameter zur Übergabe des Token hinzugefügt werden, welcher dann für die Anfrage verwendet wird. Zum *DDP* muss eine Funktion hinzugefügt werden, die es dem Nutzer erlaubt, einen Token zu generieren, also die benötigten Daten einzutragen und im Gegenzug einen Token per Email zu erhalten. Dieser Token muss dann an der Stelle, an der vorher die Daten eingetragen wurden, eingegeben werden. Der Token kann in den Cookies für die Gültigkeitsdauer gespeichert werden, damit er nicht jedes Mal eingetragen werden muss.

3.3.1 Clisos

Das jetzige Use-Case Diagramm wird in Abbildung 3.1a dargestellt. Das Ziel besteht darin, CliSos so zu erweitern, dass ein Token angegeben werden kann, welcher mit der Anfrage geschickt wird und so von der Gegenseite geprüft werden kann, ob es sich um einen berechtigten Nutzer handelt oder nicht. Wenn diese Verifizierung erfolgt ist, können ebenfalls geschützte Daten angeboten werden. Dieser Status wird in Abbildung 3.1b gezeigt.



(a) Ohne OAuth2

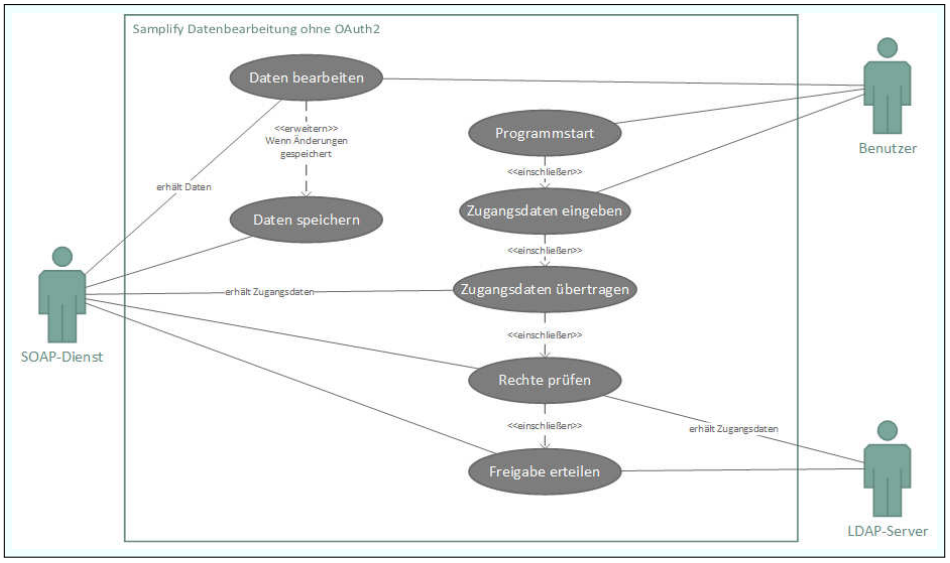


(b) Mit OAuth2

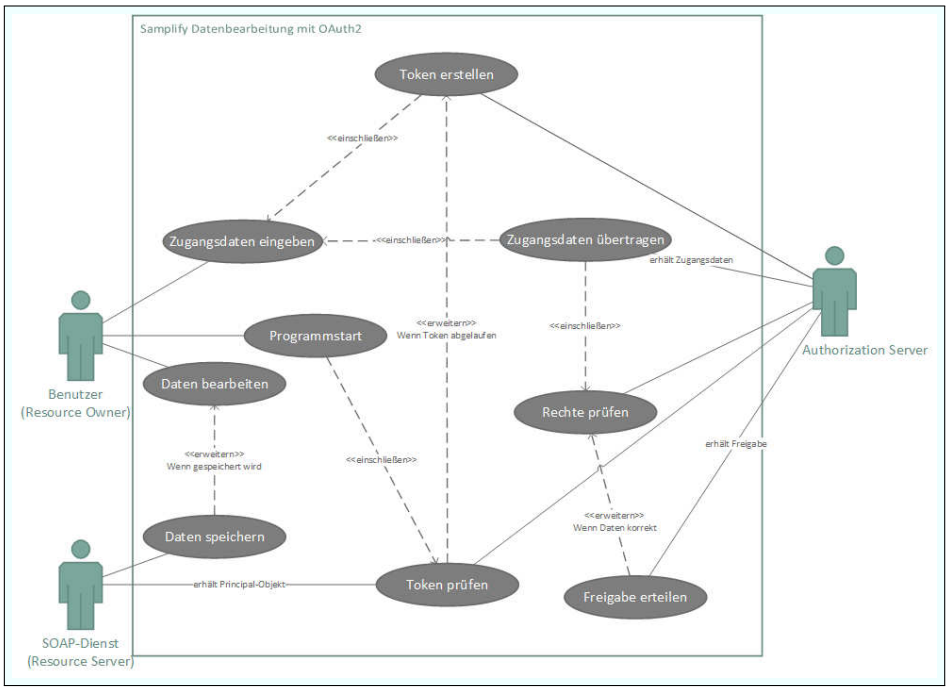
Abbildung 3.1: UML Use Case Diagramme - Clisos

3.3.2 Samplify

Abbildung 3.2a stellt den aktuellen Zustand von Samplify dar. Nach den änderungen soll dieser Klient die Möglchkeit bieten OAuth2 zu verwenden (siehe Abbildung 3.2b).



(a) Ohne OAuth2

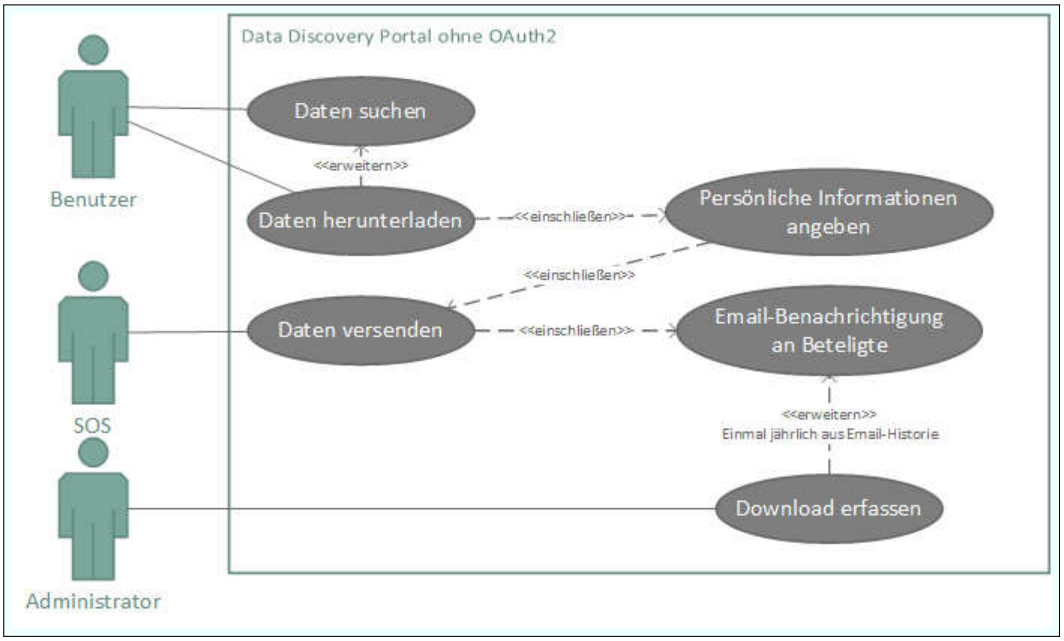


(b) Mit OAuth2

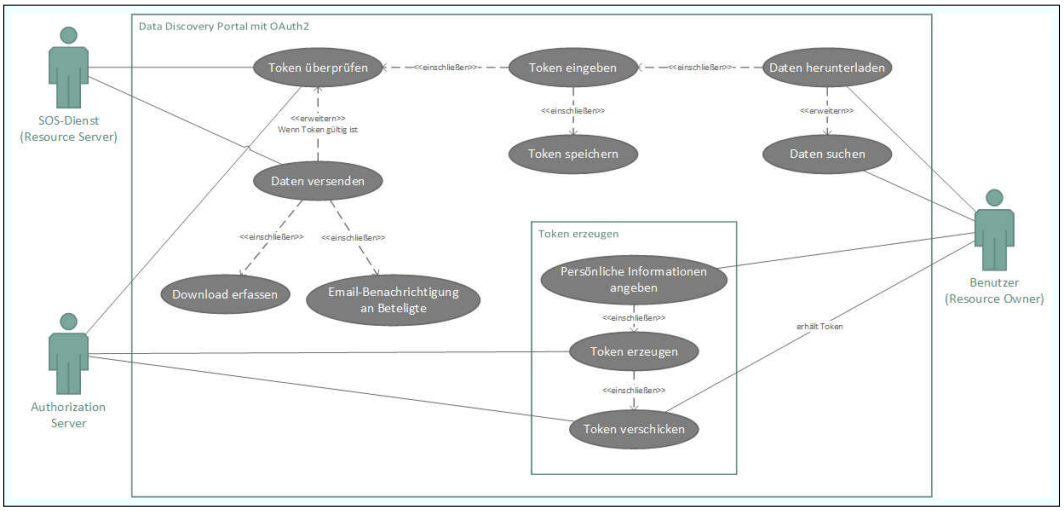
Abbildung 3.2: UML Use Case Diagramme - Samplify

3.3.3 Data Discovery Portal

Abbildung 3.3a stellt den aktuellen Zustand des Data Discovery Portal dar. Nach den änderungen soll dieser Klient die Möglchkeit bieten OAuth2 zu verwenden (siehe Abbildung 3.3b).



(a) Ohne OAuth2



(b) Mit OAuth2

Abbildung 3.3: UML Use Case Diagramme - Data Discovery Portal

4 Entwicklung

Dieses Kapitel beschäftigt sich mit der Implementierung der einzelnen Softwarekomponenten. Grundsätzlich werden alle neuen Softwareprojekte als *Maven*-Projekt angelegt.

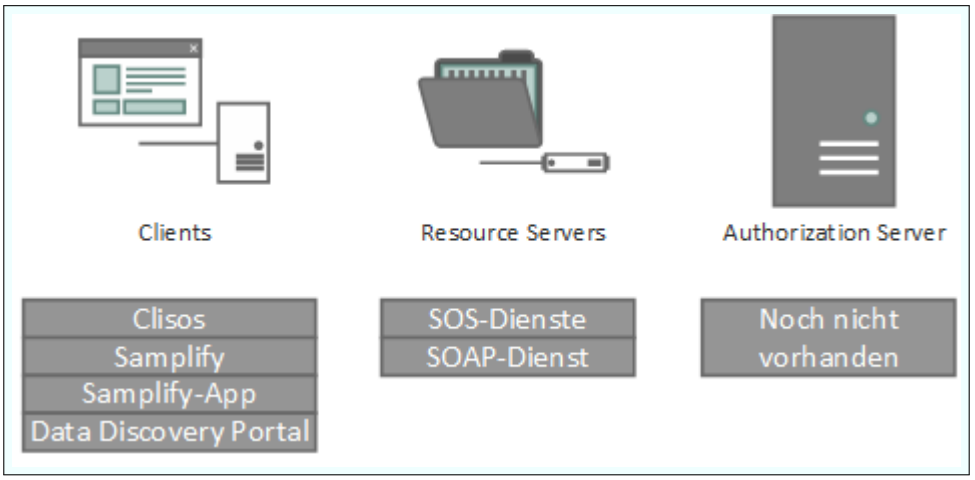


Abbildung 4.1: Übersicht der einzelnen Komponenten

In Abbildung 4.1 sehen wir den Zustand der Komponenten vor diesem Kapitel. Wir werden die Anwendungen **Clisos**, **Samplify** und das **Data Discovery Portal**, sowie den SOAP und SOS-Dienst so erweitern, dass sie den Anforderungen gerecht werden und das OAuth2-Protokoll beherrschen und verwenden. Der *Authorization Server* ist noch nicht implementiert worden und muss von Grund auf geschrieben werden. Die nachfolgenden Unterkapitel beschreiben diesen Prozess detailliert.

Zu jeder vorgestellten Methode wurde **JUnit-Tests** geschrieben, die automatisch durch Maven bei der Kompilierung durchgeführt werden. Sollte ein Fehler auftreten werden die Entwickler über diesen informiert.

4.1 Entwicklungsumgebung

Folgende Hard- und Software wurde für die Implementierung verwendet:

- i7-6700 CPU
- 16GB RAM
- Ubuntu 18.04
- IntelliJ IDEA
- PyCharm
- Spring 5
- Spring Boot 2
- Thymeleaf
- Java 1.8
- Python 3
- Tomcat 8.5
- Maven 3
- Gitlab
- Postman
- CI
- Gitlab Runner
- Docker
- Wireshark

Die Entwicklung der einzelnen Komponenten wurde in der Entwicklungsumgebung IntelliJ IDEA bzw. pyCharm durchgeführt. Die Versionierung wurde von einem internen **Gitlab**-Server verwaltet. Der *Authorization Server* wird mittels **Continuous Integration** und dem dazugehörigen **Gitlab Runner** kompilliert, getestet und veröffentlicht. Die Veröffentlichung geschieht in einem **Docker-Container**. Das *Data Discovery Portal* und der *SOAP-Dienst* werden in einer **Tomcat** Instanz ausgeführt. *Samplify* und *Clisos* (**Python**) sind Desktopanwendungen. Alle Java-Anwendungen sind **Maven**-Projekte. Der *Authorization Server* wurde mit Hilfe des **Spring** Frameworks implementiert und benutzt unter Anderem **Thymeleaf** zur Erzeugung der einzelnen Webansichten. **Postman** wurde verwendet um HTTP-Anfragen zu verschicken. **Wireshark** wurde dabei zur Analyse der verschickten Daten benutzt.

4.2 Autorisierungsserver

Zunächst beschäftigen wir uns mit der Umsetzung des OAuth2-Protokolls. Spring bietet hier schon eine sehr gute Integration. Es muss lediglich ein Paket importiert werden, um die Implementierung des OAuth2-Protokolls verfügbar zu machen. Da es sich um ein *Maven*-Projekt handelt, reicht ein einfacher Eintrag in der *pom.xml*.

```
1        <dependency>
2            <groupId>org.springframework.cloud</groupId>
3            <artifactId>spring-cloud-starter-oauth2</artifactId>
4        </dependency>
```

Listing 4.1: XML: Maven-Abhängigkeiten für OAuth 2

Darüber hinaus wollen wir die Datensicherungsschicht der OAuth-Implementierung als Postgres-Datenbank realisieren, weswegen wir zusätzlich das Spring JDBC-Starter Paket installieren. (*org.springframework.cloud:spring-starter-jdbc*)
Abbildung 4.2 beschreibt den Aufbau der einzelnen Tabellen, welche in der Spring Implementierung zum Tragen kommen. Die Tabelle *oauth_client_details* hält dabei die Informationen zu den registrierten *Clients*. Die *Authorization-Grant* Token

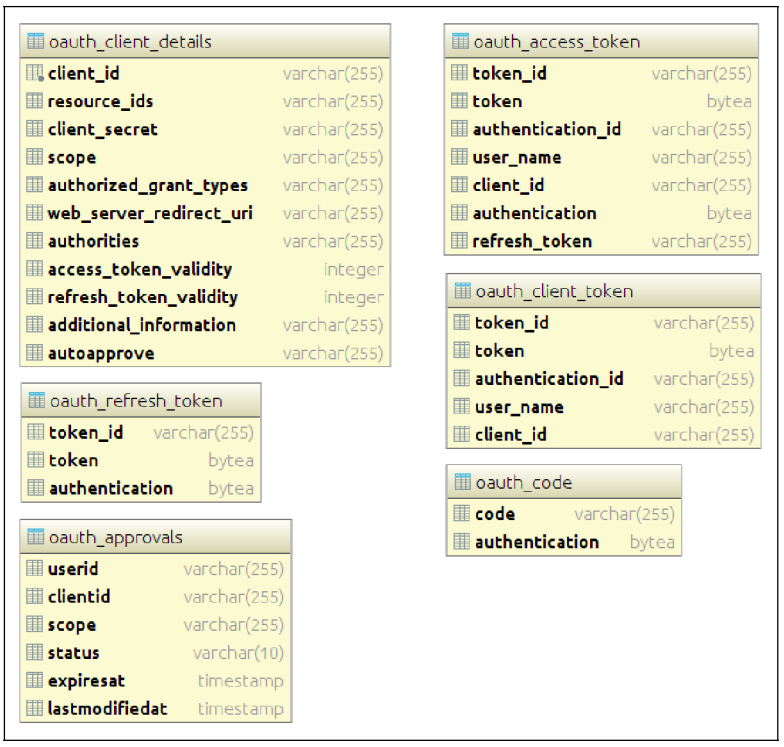


Abbildung 4.2: ER-Diagramm der OAuth2 Datenbank - Listing 7.1-7.6

des *Authorization-Code* Verfahrens werden in der *oauth_client_token*-Tabelle abgelegt und dort solange gespeichert, bis der *Resource Owner* die Anfrage freigegeben hat. Sobald diese Freigabe erfolgt ist, wird dies in der *oauth_approvals*-Tabelle mit einem Eintrag vermerkt. In Zukunft werden alle Anfragen dieser Anwendung automatisch freigegeben, wenn sich an den Konditionen nichts verändert hat. Die eigentlichen *Access* bzw. *Refresh Token* werden in den Tabellen *oauth_access_token* und *oauth_refresh_token* abgelegt.
Wir definieren die Datenbank in der *application.yml*, damit wir im Programm damit arbeiten können.

```yaml
spring:
    oauth:
      datasource:
        jdbc-url: jdbc:postgresql://ibg3db.ibg.kfa-juelich.de/oauth2
        driver-class-name: org.postgresql.Driver
        username: **********
        password: **********
```

Listing 4.2: application.yml (OAuth2 Datenbank)

Sobald der Datenbankzugriff so angegeben ist, kann man diesen in Spring konfigurieren.

```
public class DatabaseConfig {
    @Bean(name = "oauthDataSource")
    @ConfigurationProperties(prefix = "spring.oauth.datasource")
    public DataSource mainDataSource() {
        return DataSourceBuilder.create().type(HikariDataSource.class).
            build();
    }
}
```

Listing 4.3: DatabaseConfig.java (OAuth2 Datenbankkonfiguration)

Über die Annotation *@ConfigurationProperties* kann man den zugehörigen Konfigurationspfad in der *application.yml* angeben. Der Rest wird automatisch von Spring durchgeführt. Ab jetzt steht diese Datenbankverbindung zur Verfügung.
HikariCP ist ein Verbindungs-Pooling-Framework und verwaltet den Aufbau der Verbindungen zu einer Datenbank, wodurch ein effizienter Zugriff gewährleistet wird.

```
spring:
    oauth:
      datasource:
        hikari:
          connection-test-query: SELECT 1 FROM DUAL
          minimum-idle: 1
          maximum-pool-size: 5
        initialization-mode: never
```

Listing 4.4: application.yml (Hikari Datenbank konfiguration

Mit diesen Einstellungen steuert man *HikariCP*. Die Parameter sind selbsterklärend.

Kommen wir zu der eigentlichen Konfiguration des *Authorization Server*.

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfiguration extends
    AuthorizationServerConfigurerAdapter {
    private final DataSource oauthDataSource;
    @Autowired
    public AuthorizationServerConfiguration(@Qualifier("
        authenticationManager") AuthenticationManager
        authenticationManagerBean, DataSource oauthDataSource) {
        this.oauthDataSource = oauthDataSource;
    }
```

Listing 4.5: AuthorizationServerConfiguration.java Konstruktor

In der *AuthorizationServerConfiguration*-Klasse, welche von *AuthorizationServerConfigurerAdapter* erbt, wird zunächst das eben erstellte *DataSource*-Objekt injiziert. Hier versuchen wir, *reflection*-basierte Autoinjizierung (*@Autowired*) zu vermeiden, und

stattdessen konstruktor-basierte oder setter-basierte Injizierungen zu verwenden (siehe Kapitel 2.2). Mit diesem Objekt erzeugen wir einen *ClientDetailService*, welcher Informationen über die OAuth2-Clients bereitstellt. Er verwendet die Datenbank-Tabelle *oauth_client_detail*.

```
1    @Bean
2    public JdbcClientDetailsService clientDetailsService() {
3        return new JdbcClientDetailsService(oauthDataSource);
4    }
```

Listing 4.6: AuthorizationServerConfiguration.java ClientDetailService

Außerdem wird ein *TokenStore*-Objekt benötigt, welches auf die Tabellen oauth_access_token und oauth_refresh_token zugreift und dort die generierten Token speichert.

```
1    @Bean
2    public TokenStore tokenStore() {
3        JdbcTokenStore jdbcTokenStore = new JdbcTokenStore(
             oauthDataSource);
4        return jdbcTokenStore;
5    }
```

Listing 4.7: AuthorizationServerConfiguration.java TokenStore

Zudem wird ein *ApprovalStore* erzeugt, welcher die *Client*-Bestätigungen der Nutzer speichert und somit auf die Tabelle oauth_approvals zugreift.

```
1    @Bean
2    public ApprovalStore approvalStore() {
3        return new JdbcApprovalStore(oauthDataSource);
4    }
```

Listing 4.8: AuthorizationServerConfiguration.java ApprovalStore

Das Objekt vom Typ *AuthorizationCodeServices* ist für die Speicherung des Principal[1]-Objekts zuständig und benötigt Zugriff auf die oauth_code Tabelle.

```
1    @Bean
2    public AuthorizationCodeServices authorizationCodeServices() {
3        return new JdbcAuthorizationCodeServices(oauthDataSource);
4    }
```

Listing 4.9: AuthorizationServerConfiguration.java AuthorizationCodeServices

Zuletzt folgt der *ClientDetailService* und die korrespondierende Tabelle oauth_client_details, welcher durch das Überladen der *configure*-Methode mit dem Parameter *ClientDetailsServiceConfigurer* übergeben wird.

```
1    @Bean
2    public JdbcClientDetailsService clientDetailsService() {
```

[1]Das Objekt, welches den aktuell angemeldeten Nutzer repräsentiert

```
3         return new JdbcClientDetailsService(oauthDataSource);
4     }
5     public void configure(ClientDetailsServiceConfigurer clients)
          throws Exception {
6         clients.withClientDetails(clientDetailsService());
7     }
```

Listing 4.10: AuthorizationServerConfiguration.java ClientDetailService configure

Kombinieren wir nun alle erstellten *Beans*, haben wir einen funktionierenden *Token-Service* (vgl. Listing 4.41), welcher mit den folgenden Endpunkten ausgestattet ist:

/oauth/authorize Dient zur Bearbeitung von *Authorization Requests*. Sobald der *Resource Owner* die Anfrage erhält, gibt er diese über die

/oauth/confirm_access Schnittstelle frei. So erhält der Client ein *Authorization Granted* und kann sich damit an der Schnittstelle

/oauth/token einen *Access-Token* generieren lassen. Sollte ein anderes Verfahren verwendet werden, gehen alle Anfragen direkt an diesen Endpunkt. Die ausgegebenen *Access-Token* können von allen zugelassenen[2] Clients über die

/oauth/check_token Schnittstelle überprüft werden. Diese liefert das zugrunde liegende *Principal*-Objekt.

Abbildung 7.1 zeigt den Ablauf eines *authorization_code*-Verfahren, bei dem alle Endpunkte benutzt werden. Die Endpunkte sind geschützt. Das OAuth2-Protokoll sieht vor, dass ein Client seine *ID* und sein *secret* angeben muss. In Spring werden diese Parameter nicht über HTML-Parameter übergeben, sondern im HTTP *Authorization-Header*. Wenn ein Zugriffsversuch auf die Endpunkte **/oauth/authorize** oder **/oauth/token** erfolgt, werden diese blockiert, wenn diese Angaben fehlen. Um die anderen Schnittstellen benutzerdefiniert zu schützen, importieren wir das *Spring Security*-Paket (*org.springframework.boot :spring-boot-starter-security*).

```
1         <dependency>
2             <groupId>org.springframework.boot</groupId>
3             <artifactId>spring-boot-starter-security</artifactId>
4         </dependency>
```

Listing 4.11: XML: Maven-Abhängigkeiten für Spring Security

Wir prüfen, welche Berechtigungen an welchen Endpunkten nötig sind:

- Ein anonymer Nutzer muss sich authentifizieren können. Dies geschieht über einen Login. Der Endpunkt wird als **/login** definiert. Wir wählen ein HTML-Formular basiertes Anmeldeverfahren. Die POST-URL für den Login-Prozess definieren wir als */login.do*. Es werden zwei Parameter *username* und *password* erwartet. Außerdem ist bei jeder Anfrage ein CSRF-Token mitzuschicken (siehe Kapitel 6.4).

[2] Authority: ROLE_TRUSTED_CLIENT

- Alle anderen Anfragen sollen zunächst eine gültige Anmeldung voraussetzen.
- Ein **Logout** soll über den Endpunkt *logout.do* realisiert werden.

Diese Einstellungen geben wir in der *WebSecurityConfiguration*-Klasse, welche von *WebSecurityConfigurerAdapter* erben muss, an:

```
protected void configure(HttpSecurity http) throws Exception {
    http
            .authorizeRequests()
            .antMatchers("/login", "/logout.do").permitAll()
            .antMatchers("/**").authenticated()
            .and()
            .formLogin()
            .loginProcessingUrl("/login.do")
            .usernameParameter("username")
            .passwordParameter("password")
            .loginPage("/login")
            .and()
            .logout()
            .logoutRequestMatcher(new AntPathRequestMatcher("/
               logout.do"))[...]
}
```

Listing 4.12: WebSecurityConfiguration.java HttpSecurity

Um ausgestellte Access Token verifizieren zu können, muss der **/oauth/check_token** Endpunkt freigegeben werden. Im unkonfigurierten Zustand wird die *denyAll()* Regel verwendet und somit alle Anfragen blockiert.
Dies geschieht in der bereits bekannten *AuthorizationServerConfiguration*-Klasse. Die *configure*-Methode mit dem Parameter vom Typ *AuthorizationServerSecurityConfigurer* wird überschrieben.

```
public void configure(AuthorizationServerSecurityConfigurer
    oauthServer) {
    oauthServer
            .checkTokenAccess("hasAuthority('ROLE_TRUSTED_CLIENT')"
               );
}
```

Listing 4.13: AuthorizationServerConfiguration.java Sicherheitseinstellung

Da wir bei der Tokenverifizierung am **/oauth/check_token**-Enpunkt sensible Daten preisgeben, muss ein Client die Rolle *TRUSTED_CLIENT* haben, um die Tokenverifizierung nutzen zu dürfen.

Spring braucht Informationen über Benutzer, die sich anmelden können sollen. Wir bedienen uns hier dreier Stellen.

1. Ein interner **ActiveDirectory**-Server.
2. Ein interner **LDAP**[3]-Server.
3. Eine **Datenbank** (registrierte Nutzer).

Diese Datenquellen werden über einen *AuthenticationManager* angegeben und als *Bean* in der *WebSecurityConfiguration*-Klasse konfiguriert. Bei Punkt 1 und 2 handelt es sich um zwei Verzeichnisdienste, die über Einträge in der *application.yml* konfiguriert werden.

```
ldap:
  ad:
    context:
      url: ldaps://agro007.icg.kfa-juelich.de:636/dc=icg-iv-w2k,dc=kfa-juelich,
          dc=de
      managerDN: **********@icg-iv-w2k.kfa-juelich.de
      managerPassword: **********
    userSearchBase: cn=Users
    groupSearchBase: cn=Users
    groupSearchFilter: member={0}
    userSearchFilter: sAMAccountName={0}
```

Listing 4.14: application.yml (ActiveDirectory)

```
  protected void configure(AuthenticationManagerBuilder auth) throws
      Exception {
      auth
              .ldapAuthentication()
              .userDetailsContextMapper(userDetailsContextMapper())
              .userSearchFilter(userSearchFilter)
              .userSearchBase(userSearchBase)
              .groupSearchBase(groupSearchBase)
              .groupSearchFilter(groupSearchFilter)
              .authoritiesMapper(ldapAuthoritiesMapper())
              .contextSource()
              .url(url)
              .managerDn(managerDN)
              .managerPassword(managerPassword)
```

Listing 4.15: WebSecurityConfiguration.java ActiveDirectory

Äquivalent gehen wir bei dem LDAP-Server vor. Zuletzt fehlt noch die Datenbank-Authentifizierung.

```
              .and()
```

[3] s. Kapitel 6.11

```
2                .and()
3                .authenticationProvider(createAuthenticationProvider(
                    userService));
4    }
5    private AuthenticationProvider createAuthenticationProvider(
        UserDetailsService service) {
6        DaoAuthenticationProvider provider = new
            DaoAuthenticationProvider();
7        provider.setUserDetailsService(service);
8        provider.setPasswordEncoder(passwordEncoder());
9        provider.setHideUserNotFoundExceptions(true);
10       return provider;
11   }
```

Listing 4.16: WebSecurityConfiguration.java OAuth2 Datenbank

Die injizierte *UserService*-Bean wird von Spring automatisch erstellt, sobald man eine Klasse, in diesem Fall *RegisteredUserService*, erstellt und diese das Interface *UserDetailsService* implementieren lässt. Das *RegisteredUserService*-Objekt enthält einen Parameter vom Typ *UserRepository*, welches wiederum ein DAO[4] darstellt und *JpaRepository<User, Long>* erweitert. *User* ist die Klasse, welche die registrierten Benutzer beinhaltet und wird mit Hilfe eines *@Interface* als *Entity* deklariert. Weiterhin geben wir mit der Annotation[5] *Table* das entsprechende Schema und die Tabelle an, in der die Benutzer abgelegt werden.

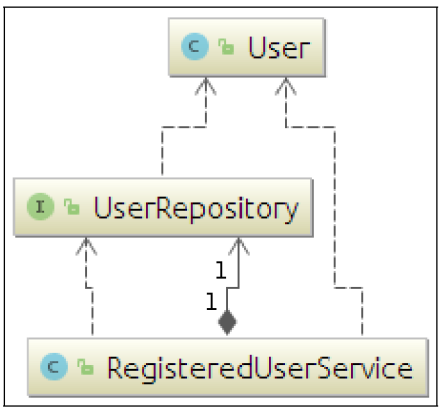
Hinweis: Alle Klassen, die zur Speicherung von Passwörtern verwendet werden, sollten das Interface *CredentialsContainer* implementieren und die sensiblen Zugangsdaten in der definierten Methode verwerfen. Andernfalls kann es dazu kommen, dass diese Informationen (beispielsweise der Hash des Passworts) nach außen verfügbar gemacht werden.

```
1  @Entity
2  @Table(schema = "registration", name = "registered_user")
3  public class User implements UserDetails,CredentialsContainer {
4      @Transient
5      public Collection<? extends GrantedAuthority> getAuthorities() {
6          return Arrays.stream(new String[] {"ROLE_USER", "
               ROLE_REGISTERED_USER"})
7                  .map(SimpleGrantedAuthority::new)
8                  .collect(Collectors.toList());
9      }
```

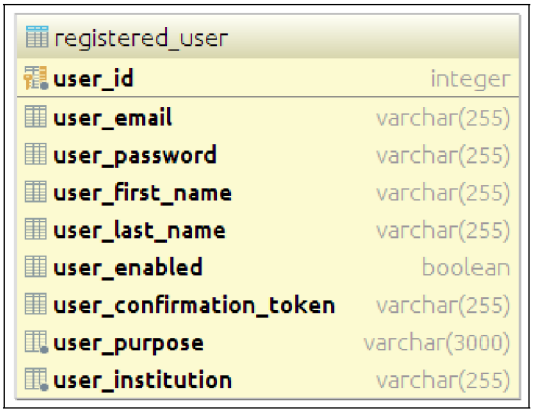
Listing 4.17: User.java Annotationen

[4]Data Access Object (s. Kapitel 2.2)
[5]Java-@interface

Jeder Benutzer dieser Klasse bekommt die Rollen *USER* und *REGISTERED_USER* zugeordnet. Dies stellt die spätere Unterscheidungsmöglichkeit zwischen verschiedenen Nutzertypen und Quellen sicher. *@Transient* sind alle Komponenten der Klasse, welche nicht auf die Datenbank abgebildet werden sollen; so auch diese Methode.



(a) User Klassendiagramm



(b) User ER-Diagramm

[6] [2]

4.2.1 Login

Um die in der Anforderungsanalyse festgestellten Features zu entwickeln, wird die *Spring-Thymeleaf* Erweiterung (s. Kapitel 6.3) verwendet. Der erste Schritt ist es, die *Login*-Seite zu erstellen. Dazu benötigen wir eine mit *@Controller* deklarierte Klasse. Wir kreieren eine Methode und annotieren diese mit *@RequestMapping("/login")* (vgl. Listing 4.12, Zeile 11). Der Methodenkörper beinhaltet lediglich

```
1       return "login";
```

Listing 4.18: LoginController Login Methodenkörper

und verweist somit auf das *Thymeleaf* Template *login.html* im *resource* Verzeichnis. Dieses werden wir als nächstes implementieren. Grundsätzlich handelt es sich um normalen HTML Quelltext, allerdings fügen wir Steuerkommandos[6] hinzu, die vor der Darstellung von Thymeleaf prozessiert bzw. ersetzt werden. Der komplette Quelltext befindet sich in Listing 7.7. Die nachfolgenden Zeilenangaben beziehen sich auf dieses. Wir benutzen zur optischen Verschönerung das *Bootstrap*-Framework. Durch das Artefakt *bootstrap* und der Gruppe *org.webjars* werden die Abhängigkeiten über die *pom.xml* importiert. In der *login.html* wird diese über ein *<style>*-Tag angegeben (s. Zeile 6). Das eigentliche Login-Formular wird in den Zeilen 26-69 beschrieben. Wie in Listing 4.12, Zeile 8 angegeben, schreiben wir ein HTML-Formular mit genau diesem Pfad als Ziel in Zeile 29. Wir benutzen die POST-Methode, da Spring dies so erwartet. Sollten Fehler auftreten und diese übergeben worden sein, so werden diese

[6] Spring Expression Language

durch die Zeilen 30-34 entsprechend dargestellt. Zu dem Formular gehören die Parameter *username*, *password* und *${_csrf.parameterName}*. Letzterer ist dabei als *hidden* deklariert, wird dem Nutzer also nicht angezeigt, und wird von Spring automatisch injiziert um CSRF-Angriffe zu verhindern (s.o.). Das führende $-Symbol gibt an, dass Thymeleaf das nachfolgende Objekt als Parameter behandelt und im Verlauf der Prozessierung durch den eigentlichen Parameterinhalt ersetzt. In den Zeilen 53ff. wird der *Login*-Knopf angelegt, welcher das Formular an die entsprechende Adresse schickt, sobald er aktiviert wird. Zeile 64 verweist auf die *Nutzerregistrierung*. Das Ergebnis ist eine schlichte, aber ansprechende, Login-Seite (s. Abbildung 4.4). Die Formatierung wurde mit Hilfe von *Bootstrap*-Klassen durchgeführt. Abbildung 4.4 zeigt das Ergebnis der bisherigen Arbeit.

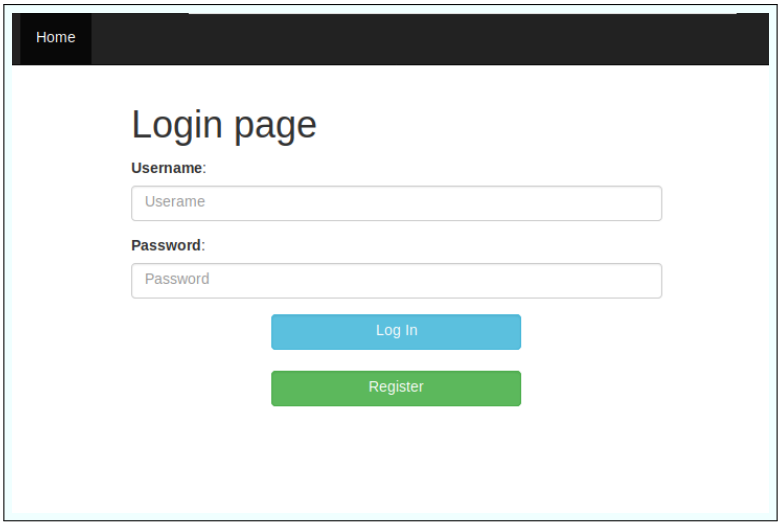


Abbildung 4.4: Ansicht der Login-Seite

4.2.2 Index

Sobald der Login erfolgt ist, liegt ein *Principal*-Objekt vor, und der Benutzer erhält den Status *authenticated*. Diese Informationen kann man nutzen, um beispielsweise verschiedene Ansichten für Nutzergruppen zu erstellen. Zunächst betrachten wir allerdings den Teil, der für Alle gleich ist. Wir benötigen eine neue Methode, welche sich um die Aufrufe von "/" kümmert, also den *Index* der Seite.

```
1    @RequestMapping("/")
2    public ModelAndView root(Map<String,Object> model, Principal
         principal){
3        return new ModelAndView ("index",model);
4    }
```

Listing 4.19: LoginController Index-Mapping

Wie man gut erkennen kann, wird das Principal-Object automatisch durch Spring an die Methode übergeben, wenn man dies als Parameter im Methoden-Kopf deklariert. Auch ein *Model*, welches schlussendlich an *thymeleaf* übergeben wird, um dort Daten zur Verfügung zu stellen, wird übergeben. Dem *ModelAndView*-Objekt wird die injizierte *Map<String,Object>* und, ähnlich zu der *login*-Methode (Listing 4.18), das *thymeleaf*-Template als String übergeben. Wir machen den *ClientDetailService* unserer registrierten Nutzer, den *ApprovalStore* und *TokenStore* über setter-basierte *@Autoinjection* als Klassenvariablen verfügbar und nutzen diese um die Informationen zu erhalten, welche wir auf der Übersichtsseite darstellen wollen. Zum einen sind das alle *Clients*, die der Nutzer bestätigt (*Approvals*) hat, und zum andern die generierten *Access-Token*.

```
1       List<Approval> approvals=clientDetailsService.listClientDetails
            ().stream()
2               .map(clientDetails -> approvalStore.getApprovals(
                    principal.getName(),clientDetails.getClientId()))
3               .flatMap(Collection::stream)
4               .collect(Collectors.toList());
5
6
7       List<TokenMapper> tokens=clientDetailsService.listClientDetails
            ().stream()
8               .map(clientDetails -> new TokenMapper(clientDetails,
                    tokenStore.findTokensByClientIdAndUserName(
                    clientDetails.getClientId(),principal.getName())))
9               .collect(Collectors.toList());
```

Listing 4.20: LoginController Approvals und Token

Java-Streams bieten hier eine sehr übersichtliche Lösung an, mehrere Operationen auf einmal durchzuführen. Die Klasse *TokenMapper* (Listing 7.8) übernimmt die Aufgabe der Verknüpfung von einer *Token*-Liste und einem *ClientDetails*-Objekt und soll Lesbarkeit verbessern. Die gesammelten Informationen übergeben wir an *Thymeleaf*, indem wir sie der *model*-Variable hinzufügen, bevor wir das *ModelAndView*-Objekt erstellen und zurückgeben.

```
1       model.put("approvals",approvals);
2       model.put("clientDetails",clientDetailsService.
            listClientDetails());
3       model.put("tokenMappers",tokens);
```

Listing 4.21: LoginController Model füllen

Die folgenden Zeilenangaben beziehen sich auf die *index.html*-Datei bzw. Listing 7.9. Die Datei wurde an den markierten Stellen gekürzt. Die Kommentare *@thymesVar* werden von der Programmierumgebung genutzt um die Autoersetzung zu realisieren. Es erfolgt eine Zuordnung der Variablen zu den entsprechenden Java-Klassen. In den

Zeilen 27ff. iterieren wir über die *Approvals* Liste. Das gleiche wird für die *Token* in den Zeilen 71ff. durchgeführt. Jedes Element wird auf eine neue Zeile in der Tabelle abgebildet. Zusätzlich wird für jedes *Approval*-Element ein *Entfernen*-Icon erstellt, mit dem man den entsprechenden Eintrag löschen kann.
Wenn der Nutzer bzw. das *Principal*-Objekt die Rolle *OAUTH_ADMIN* besitzt, wird die übergebene Variable *clientDetails* verwendet. Im Quelltext wird dies in Listing 7.10, Zeile 1 überprüft und durch die Zeilen 17ff., ebenfalls im Listing 7.10, analog zu den *Approvals* und *Token*, dargestellt. Ein Bearbeiten-Knopf wurde für die Editierung der *Clients* hinzugefügt. Die entsprechenden Ziele aller Formulare werden im Folgenden implementiert.
Um die Version des *Authorization Server* überprüfen zu können, wird ein *<footer>*-Element benutzt (Zeilen 93ff.). Die Variable wird durch den Controller automatisch zur Verfügung gestellt und über die Anotation *@ModelAttribute* deklariert. Die Version wird direkt aus der *pom.xml* ausgelesen. Hierzu wird die Abhängigkeit *maven-model* aus der Gruppe *org.apache.maven* vorausgesetzt.

```
1   @ModelAttribute("version")
2   public String getVersion() {
3       Model model = null;[...]
4           model = reader.read(new FileReader("pom.xml"));
5           return model.getVersion();[...]
6   }
```

Listing 4.22: LoginController Version

Für die Entfernung von *Approvals* muss eine neue Methode erstellt werden, die wir wie folgt definieren:

```
1   @RequestMapping(value="/approval/revoke",method= RequestMethod.
        POST)
2       public String revokeApproval(@Validated Approval approval,
            BindingResult bindingResult ){[...]
3       if( bindingResult.hasErrors())
4       {
5           LOG.error( "There are errors! {}"+ bindingResult );
6           return "error";
7       }
8       approvalStore.revokeApprovals(Collections.singletonList(
            approval));
9       tokenStore.findTokensByClientIdAndUserName(approval.getClientId
            (),approval.getUserId())
10              .forEach(tokenStore::removeAccessToken);
11      return "redirect:/";
12  }
```

Listing 4.23: LoginController Bestätigung zurücknehmen

Wir erwarten eine POST-Anfrage an diesem Endpunkt, in welcher ein *Approval*-Objekt übersendet werden soll. Diese Instanz wir automatisch durch Spring anhand der POST-Parameter erzeugt, geprüft und an die Methode übergeben. Etwaige Fehler werden in dem *BindingResult*-Objekt gespeichert und während der Laufzeit überprüft, notiert und dargestellt. Da in der *Approval*-Klasse ein Datum vorhanden ist, muss ein neuer *Binder* registriert werden, welcher das Format beschreibt, in dem ein Objekt serialisiert wird. Spring erkennt das Datum ohne diese Erweiterung nicht richtig.

```
1    @InitBinder
2    public void bindingPreparation(WebDataBinder binder) {
3        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm
             :ss");
4        CustomDateEditor orderDateEditor = new CustomDateEditor(
             dateFormat, false);
5        binder.registerCustomEditor(Date.class, orderDateEditor);
6    }
```

Listing 4.24: LoginController InitBinder

Um den *LoginController* zu finalisieren, benötigen wir noch einen *Logout*-Endpunkt. Das aktuell *Principal*-Objekt wird benutzt, um den Vorgang durchzuführen. Anschließend wird der Nutzer auf die Login-Seite weitergeleitet und dabei **logout** als GET-Parameter angegeben. Auf diese Weise kann man innerhalb des *login*-Templates feststellen, dass es sich um einen erfolgreichen Logout-Vorgang handelt und die entsprechende Meldung anzeigen. (s. Listing 7.9, Zeile 4 und Listing 7.7, Zeilen 35ff.).

Das Ergebnis der bisherigen Arbeit ist in Abbildung 7.2 zu sehen.

Es ergibt sich das in Abbildung 4.5 zu sehende Klassendiagramm.

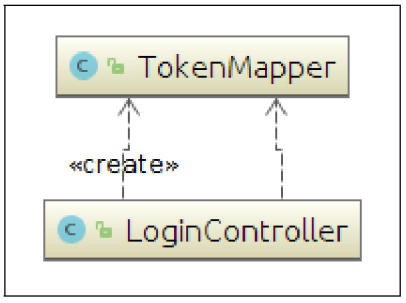


Abbildung 4.5: LoginController - Klassendiagramm

4.2.3 Registrierung

Als nächstes wird die Nutzerregistrierung implementiert. Hierzu gehen wir zunächst genau so vor, wie bei der Login-Umgebung. Um sich registrieren zu können, benötigen wir zuallererst einen zuständigen Endpunkt. Wir erstellen eine neue Klasse *RegistrationController* und annotieren diese, wie bisher, mit *@Controller*. Diese logische Trennung wird durch die Änderung des Zugriffspfads unterstützt und durch das Hinzufügen des *@Interface @RequestMapping("registration")* umgesetzt. Der *Controller* nimmt nun nur noch Anfragen an diesen Pfad und dessen Unterpfade an. Alle *@RequestMapping* Annotationen im Inneren der Klasse verhalten sich relativ zu diesem Pfad.
Wir implementieren eine Methode, die auf das Thymeleaf *register*-Template verweist. Außerdem brauchen wir hier die Unterscheidung zwischen POST und GET-Anfrage, da wir zwei verschiedene Endpunkte benötigen.

Grundsätzlich werden für eine Aktion mindestens zwei Endpunkte bzw. Logiken benötigt. Eine für die Darstellung und eine Andere um die Aktionen auszuführen. Im Normalfall wird, wie oben beschrieben, für die Darstellung eine GET-Anfrage verwendet und für ausführende Aktionen eine POST-Anfrage.

Wir benötigen Zugriff auf die *RegisteredUserService*-Bean, also lassen wir diese von Spring injizieren. Außerdem übergeben wir ein leeres *User*-Objekt, um automatische Überprüfungen mittels Thymeleaf zu ermöglichen.

```
1    @RequestMapping(value = "/register", method = RequestMethod.GET)
2    public ModelAndView showRegistrationPage(ModelAndView modelAndView
         , User user) {
3        modelAndView.addObject("user", user);
4        modelAndView.setViewName("register");
5        return modelAndView;
6    }
```

Listing 4.25: RegistrationController Register-GET

Schauen wir uns das *register.html*-Template an:
Alle folgenden Zeilenangaben beziehen sich auf Listing 7.11.
Analog zu der Fehlerausgabe des *login.html*-Quelltexts erstellen wir einige * *-Tags, die der Ausgabe dienen (Zeilen 8ff.). Im öffnenden *<form>*-Tag geben wir mittels *th:object* das *User*-Objekt an, über welches die Eingaben des Nutzers im weiteren Verlauf verifiziert werden (Zeile 3). Zusätzlich zu der *Spring*-Validierung gibt es noch die *Bootstrap*-Validierung, welche direkt im Browser des Nutzers durchgeführt wird. Hierzu werden die Zeilen 5 und 30 benötigt. Sobald diese Validierung einen Fehler zum Zeitpunkt der Eingabe feststellt, wird dem Nutzer in Echtzeit eine entsprechende Ausgabe angezeigt. Die *Thymeleaf*-Validierung wird über die Zeilen 19ff. gesteuert. Da wir hier eine theoretisch öffentliche Registrierung entwickeln, müssen wir uns gegen den Missbrauch der Seite schützen. Dies wird über so genannte *Captchas* rea-

lisiert. *Google* ist ein bekannter Anbieter für einen solchen Dienst. Wir importieren die Abhängigkeit (Zeile 32) und verändern das Verhalten des *Submit*-Knopfs so, dass er nicht mehr das Formular abschickt. Wir geben ihm die Klasse *g-recaptcha* und setzen die Parameter *data-sitekey* mit dem Wert, der von Google erzeugt wird, und *data-callback* mit der in den Zeilen 34 definierten *onSubmit*-Methode, welche nach der *Captcha*-Überprüfung ausgeführt wird. Das Ergebnis erhält die mit POST annotierte Methode, die wir als nächstes implementieren werden.
Die *Login*-Seite wird in Abbildung 7.3a dargestellt.
Das *@RequestMapping*-Interface wird mit dem gleichen Pfad wie vorher angegeben, allerdings wird dieses Mal auf die POST-Methode reagiert. Das in der *register.html* erstellte Formular schickt die POST-Anfrage an diese Adresse:

```
1    @RequestMapping(value = "/register", method = RequestMethod.POST)
2    ModelAndView processRegistrationForm(ModelAndView modelAndView,
3        @Valid User user,
4        BindingResult bindingResult,
5        HttpServletRequest request,
6        @RequestParam("g-recaptcha-response") String googleResponse) {
```

Listing 4.26: RegistrationController POST-Methode

Wir bekommen ein *User*-Objekt übergeben und den *g-recaptcha-response* String, welcher für die Validierung der Anfrage genutzt wird.

```
1 [...]
2
3        if (!isValidRequest(googleResponse, request.getRemoteAddr())){
4            modelAndView.addObject("errorMessage", "Could not verify
                 Request!");
5            modelAndView.setViewName("register");
6            return modelAndView;
7        }[...]
```

Listing 4.27: RegistrationController Captcha-Überprfüng Teil 1

Sollte ein Fehler aufgetreten sein, wird dieser zurückgegeben bzw. dargestellt und die Anfrage abgebrochen.

Für die Validierung wird eine Anfrage an die in der *application.yml* definierte URL zusammen mit dem von Google vorgegebenen Schlüssel geschickt. Wenn diese Variable mit der IP des Anfragensteller an die Adresse geschickt wird, erhalten wir ein *JSON*-Objekt als Antwort, welches den Status der Validierung angibt. Sollte diese den Wert *success* haben, wird die Registrierung des Nutzers fortgeführt.

```
1    @Value("${google.captcha.v2.url}")
2    String captchaUrl;
3    @Value("${google.captcha.v2.secret}")
4    String captchaSecret;[...]
```

```
5     boolean isValidRequest(String captchaResponse, String remoteip) {
          [...]
6         HttpPost httppost = new HttpPost(captchaUrl);[...]
7         params.add([...]("secret", captchaSecret));
8         params.add([...]("response", captchaResponse));[...]
9        return [...] jsonObject.get("success").getAsBoolean();
10    }
```

Listing 4.28: RegistrationController Captcha-Überprfüng Teil 2

Das übergebene *User*-Objekt wird durch die *Repository* bzw. das *DAO* gespeichert und mit einem generierten *Aktivierungstoken* versehen. Dieser Schlüssel wird zur Bestätigung der angegebenen Email-Adresse verwendet,indem eine Nachricht mit dem entsprechenden Aktivierungsschlüssel an diese geschickt wird. Wenn man den Link erhalten hat und anklickt, wird man auf eine Seite geleitet, die dazu auffordert, ein Passwort zu seinem Konto hinzuzufügen. Um diese Funktionalität umsetzen zu können, benötigen wir ein neues Thymeleaf-Template, zwei Methoden und eine *Email-Service*-Klasse. Ein Objekt der Klasse *JavaMailSender* wird von Spring automatisch als *Bean* generiert, sobald die entsprechenden Parameter in der *applications.yml* angegeben sind.

```
1 spring:
2     mail:
3       host: mail.fz-juelich.de
4       username: **********@fz-juelich.de
5       password: **********
6       port: 587
```

Listing 4.29: application.yml Email-Konfiguration

Durch die Injizierung dieser Instanz ist eine Implementierung des *EmailService* denkbar einfach.

```
1  @Service("emailService")
2  public class EmailService {
3      private JavaMailSender mailSender;
4      @Autowired
5      public EmailService(JavaMailSender mailSender) {
6          this.mailSender = mailSender;
7      }
8      @Async
9      public void sendEmail(SimpleMailMessage email) {
10         mailSender.send(email);
11     }
12 }
```

Listing 4.30: EmailService.java

Wir verwenden das *@Interface @Async* um Spring anzuweisen, diese Methode asynchron, also in einem eigenen Thread, auszuführen. Dies hat den Vorteil, dass mit dem normalen Programmablauf fortgefahren werden kann, ohne dass wir auf das erfolgreiche Versenden der Email warten müssen.
Eine Email wird versendet, indem man ein neues *SimpleMailMessage*-Objekt erstellt, den Empfänger, Betreff, Text und Absender setzt, und anschließend der *sendMail*-Methode übergibt.
In dem *GET* annotierten *confirm*-Endpunkt erhalten wir den vorher generierten Token als Parameter. Wir suchen nach einem Eintrag in der Datenbank mit diesem Token und zeigen das *confirm.html*-Template an. Sollte der Eintrag gefunden worden sein, fügen wir diesen zu dem Model hinzu, andernfalls eine entsprechende Fehlermeldung.

Sicherheit:

Da der Nutzer im folgenden Schritt dazu aufgefordert wird, ein Passwort zu setzen, und dieses abgespeichert wird, ist es wichtig, dies mit besonderer Vorsicht zu tun. Bevor Passwörter abgelegt werden, sollte man sich immer darüber informieren, wie die aktuellen Empfehlungen bzw. *Best Practices* diesbezüglich sind. Zum Zeitpunkt der Implementierung dieses Systems wird dazu geraten[3], *Argon2* (Kap. 6.6) zu verwenden, um Passwörter zu verarbeiten, bevor sie abgelegt werden. So wird es einem Angreifer sehr schwer gemacht Kennwörter zu rekonstruieren, selbst wenn er den Zugriff auf die Datenbank erlangt auf welcher die Passwörter gespeichert werden. Außerdem sollte man darauf achten, dass ein Mindestmaß an Sicherheitskriterien, wie die Passwortlänge, das Verwenden von Sonderzeichen, der Verzicht auf Geburtsdaten und die Email-Adresse im Passwort, etc. angewendet werden, um sicherzustellen, dass das Benutzerkonto nicht durch einen *Brute-Force*[7]-Angriff kompromittiert wird. Mehr dazu im Kapitel 4.8.

Um das Passwort mit Argon2 zu sichern, benötigen wir die *Maven*-Abhängigkeit *de.mkammerer : argon2-jvm*. Für die Messung der Passwortstärke verwenden wir *zxcvbn*. *Argon2* wird als *PasswordEncoder*-Bean durch Spring erzeugt. Die Klasse sieht wie folgt aus:

```
1 public class Argon2PasswordEncoder implements PasswordEncoder {
2     private final static Argon2 argon2 = Argon2Factory.create();
3     private final static int MEM = 65536;
4     private final static int THREADS = 1;
5     private final static int ITERATIONS = 20;
6     public String encode(CharSequence charSequence) {[...]
7             return argon2.hash(ITERATIONS, MEM, THREADS, pass);
8     }
9     public boolean matches(CharSequence charSequence, String s) {[...]
```

[7]Das Ausprobieren aller möglichen Kombinationen

```
10            return argon2.verify(s, pass);
11    }
12 }
```

Listing 4.31: Argon2PasswordEncoder.java

Man kann die mitgelieferte *Argon2Helper*-Klasse dazu verwenden, um die, für das System optimale Anzahl an Iterationen, zu berechnen. Da dieser Vorgang relativ lange dauert und rechenintensiv ist, sollte die Berechnung nur einmal für das entsprechende System durchgeführt werden.

```
1    private final static long TIME = 1500;
2     ITERATIONS = Argon2Helper.findIterations(argon2, TIME, MEM,
         THREADS);
```

Listing 4.32: Argon2PasswordEncoder.java - Iterationen berechnen

Um diesen *PasswordEncoder* nutzen zu können, erstellen wir eine dazugehörige *@Bean* in der *WebSecurityConfiguration*-Klasse.

```
1    @Bean
2    public PasswordEncoder passwordEncoder() {
3        return new Argon2PasswordEncoder();
4    }
```

Listing 4.33: WebSecurityConfiguration.java PasswordEncoder

Zum Zwecke der internen Überprüfung des Benutzers müssen wir diesen *PasswordEncoder*, wie in Listing 4.15, Zeile 8 zu sehen, angeben.
Die *confirm* POST-Methode erhält das Passwort des Nutzers und codiert es mit dem gleichen *PasswordEncoder*-Objekt, wenn die Stärke des gewählten Passworts durch *Zxcvbn* für „gut“ befunden wurde. Der Benutzer wird aktiviert und der Aktivierungsschlüssel wird gelöscht.

```
1 [...]
2        Strength strength = new Zxcvbn().measure(password);
3        if (strength.getScore() < 3) {
4            bindingResult.reject("password");[...]
5        }
6        User user = userService.findByConfirmationToken(token);
7        user.setPassword(passwordEncoder.encode(password));
8        user.setConfirmationToken("");
9        userService.saveUser(user);
10       modelAndView.addObject("successMessage", "Your password has
            been set!");[...]
```

Listing 4.34: RegistrationController - Confirm POST

In der *confirm.html* wird während der Passworteingabe überprüft, wie „gut“ das Passwort ist. Das Ergebnis dieser Prüfung wird dem Nutzer angezeigt. (Listing 7.12, Zeilen

14ff.)

Der komplette Ablauf einer Registrierung wird in Abbildung 7.3 gezeigt.

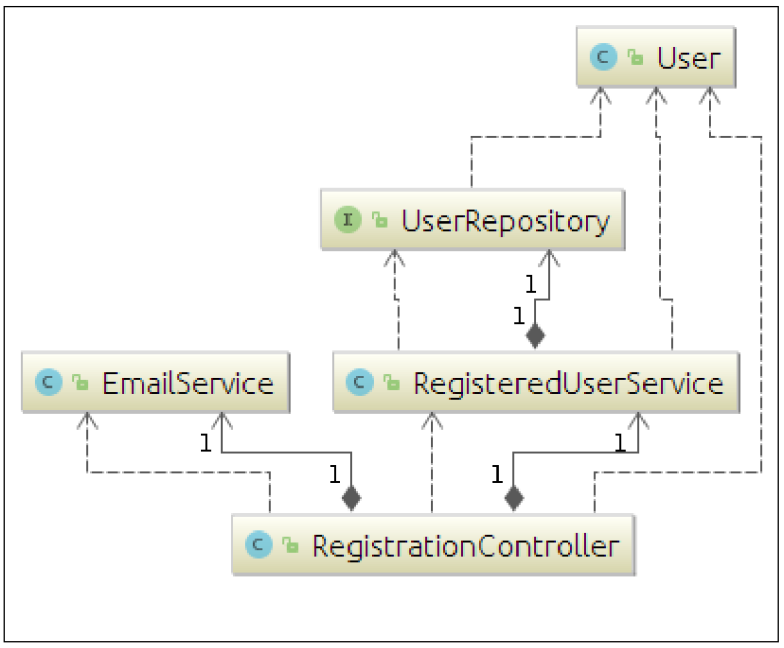


Abbildung 4.6: RegistrationController - Klassendiagramm

Es ergibt sich das in Abbildung 4.6 zu sehende Klassendiagramm. [7]

4.2.4 Nutzerverwaltung

Ein Administrator hat berechtigtes Interesse daran, die registrierten Nutzer sehen, verwalten und verändern zu können. Ebenso sollte dem Benutzer die Möglichkeit gegeben werden, sich seine Informationen anzuzeigen und anpassen zu können. Aus diesem Grund legen wir wieder drei Methoden an. Dieses Mal im *RegistrationController* mit dem Ziel **/edit** bzw. **/overwiev**
Die letzte Methode brauchen wir, um die Übersicht der Daten darzustellen. Wenn man die Rolle **OAUTH_ADMIN** hat, wird eine Tabelle mit allen registrierten Nutzern angezeigt inklusive Löschen-Knopf. Wenn man kein Administrator ist, wird nur das eigene Objekt angezeigt. Über den Bearbeiten-Knopf wird man an die */edit*-Schnittstelle weitergeleitet, an welche die zu bearbeitende User-ID mit übergeben wird. In dieser Ansicht kann man die angezeigten Daten editieren und über den *Speichern*-Knopf mittels POST-Anfrage an den gleichen Endpunkt sichern. Es wird überprüft, ob es sich bei dem *Principal*-Objekt um einen Administrator handelt. Wenn dies der Fall ist, wird das Nutzerobjekt in der Datenbank aktualisiert. Wenn es sich nicht um einen Administrator handelt, wird vorher noch anhand der Email-Adresse überprüft, ob der Nutzer mit dem *Principal*-Objekt übereinstimmt.

```
[...]
else if (principal instanceof InetOrgPerson){
  InetOrgPerson currentUser = (InetOrgPerson) principal;
  boolean isAdmin = currentUser
    .getAuthorities()
    .stream()
    .map(GrantedAuthority::getAuthority)
    .anyMatch( (s) -> s.equals("OAUTH_ADMIN"));

  if (isAdmin){
    userService.saveUser(user);
  }
[...]
```

Listing 4.35: RegistrationController - edit

4.2.5 Tokenerzeugung für temporäre Nutzer

Neben dem normalen OAuth2-Flow benötigen wir für das Datenportal (Kap. 4.7) noch die Möglichkeit, *TemporaryUser*-Objekte anzulegen. Ein Benutzer dieser Klasse ist dafür da, einen Token zu halten, welcher eine Woche lang gültig ist, und die Eingaben des Nutzers (Name, Email-Adresse, Zweck der Datennutzung) speichert. Damit der Prozess automatisch vonstattengeht, brauchen wir eine entsprechende Schnittstelle im Autorisierungsserver. Wir definieren den Endpunkt **/temptoken** im *RegistrationController* mit der POST-Methode.

```
@RequestMapping(value = "/temptoken", method = RequestMethod.POST)
@ResponseBody
public String getTokenForDDP(
   HttpServletRequest request,
   @RequestParam("email") String email,
   @RequestParam("acceptConditions") boolean accept,
   @RequestParam("purpose") String purpose,
   @RequestParam("institution") String institution,
   @RequestParam("first_name") String firstName,
   @RequestParam("last_name") String lastName,
   @RequestParam(name = "submit", required = false) String submit,
   @RequestParam(value = "g-recaptcha-response") String gResponse) {
```

Listing 4.36: RegistrationController - Temporärer Benutzer

@ResponseBody veranlasst Spring dazu, den Rückgabewert direkt als HTML-Body zu verwenden und nicht, wie normalerweise, diesen als Template-Namen zu interpretieren.

Die Eingaben des Nutzers werden als Parameter erwartet. Zusätzlich erwarten wir

eine Captcha-Überprüfung und, dass der Nutzer die Allgemeinen Bedingungen akzeptiert hat. Die Verifizierung des Captcha wird, wie bereits gezeigt, durchgeführt. Wenn die Anfrage so verifiziert ist, wird der Nutzer angelegt und es wird ein neuer *Access-Token* generiert.

```
1        String accessToken = temporaryTokenService.
             generateOAuth2AccessToken(temporaryUser, Collections.
             singletonList("read"));
```

Listing 4.37: RegistrationController - Generierung des Access-Token

Da das OAuth2 Protokoll nicht vorsieht, dass auf diesem Weg ein Token generiert wird, müssen wir diesen Schritt manuell durchführen. Dazu benötigen wir einen eigenen *TokenService*. Wir greifen dabei auf die bereits erzeugte *DefaultTokenService*-Bean zurück.

```
1  @Service
2  public class TemporaryTokenService {
3      private final DefaultTokenServices defaultTokenServices;
4      public String generateOAuth2AccessToken(TemporaryUser user, List<
           String> scopes) {[...]
5          responseTypes.add("code");[...]
6
7          List<GrantedAuthority> authorities = user.getAuthorities()
8              .stream()
9              .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
10             .collect(Collectors.toList());[...]
11
12         OAuth2Authentication auth = new OAuth2Authentication(
               oauth2Request, authenticationToken);
13         OAuth2AccessToken token = defaultTokenServices.
               createAccessToken(auth);
14
15         return token.getValue();
16     }
```

Listing 4.38: TemporaryTokenService

Der so generierte Token wird mit Hilfe des *EmailService*-Objekts an die angegebene Email-Adresse geschickt.

Hinweis:
An dieser Stelle tritt ein Problem auf: Die Spring OAuth2 Implementierung erzeugt immer den gleichen Token, solange der alte noch aktiv ist. Wenn ein Benutzer also einen neuen Token für beispielsweise einen anderen Zweck haben möchte, so bekommt er den gleichen Token zugeschickt, den er beim vorherigen Mal erhalten hat. Dieser behält die alten Informationen und Gültigkeitsdauer. Um dieses Problem zu adressieren,

müssen wir einen eigenen *AuthenticationKeyGenerator* schreiben, welcher das gleichnamige Interface implementieren muss. Der Token wird generiert, indem ein Hash über verschiedene Einträge einer *Map* gebildet wird. Dieser Map (values) müssen wir lediglich einen *Primärschlüssel* der Klasse *TemporaryUser* hinzufügen, in diesem Fall die *User ID*. Wir erweitern die **extractKey**-Methode also um einen Fall:

```
1       if (authentication.getPrincipal() instanceof TemporaryUser){
2           TemporaryUser temporaryUser = (TemporaryUser)
                authentication.getPrincipal();
3           values.put(TEMP_USER_ID,temporaryUser.getId()+"");
4       }
```

Listing 4.39: CustomAuthenticationKeyGenerator Temporaryuser

Ebenfalls passen wir die **generateKey**-Methode an und verwenden einen etwas zeitgemäßeren Hash-Algorithmus (SHA-256).

```
1           digest = MessageDigest.getInstance("SHA-256");
```

Listing 4.40: CustomAuthenticationKeyGenerator SHA-256

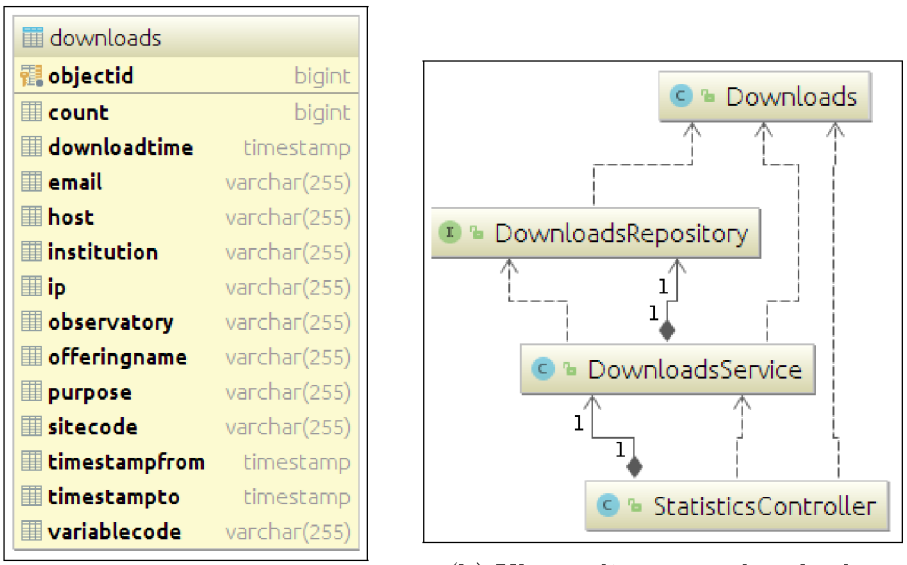
Sobald die Klasse fertiggestellt ist, kann sie in der *AuthorizationServerConfiguration*-Klasse angegeben werden.

```
1   public void configure(AuthorizationServerEndpointsConfigurer
        endpoints) {
2       endpoints
3               .approvalStore(approvalStore())
4               .authorizationCodeServices(authorizationCodeServices())
5               .tokenStore(tokenStore())
6               .authenticationManager(authenticationManagerBean)
7       .accessTokenConverter(new CustomAccessTokenConverter());
8   }
```

Listing 4.41: AuthorizationServerConfiguration.java Endpunkt-Konfiguration

4.2.6 Statistiken

Eine weitere wichtige Anforderung ist das Erfassen von Download-Statistiken. Wie schon in der Motivation beschrieben, ist es eine der ausschlaggebenden Punkte, die zu der Idee dieser Ausarbeitung geführt haben. Um diese Statistiken erheben zu können, stellen wir eine weitere Schnittstelle bereit. Für die Datenhaltung benötigen wir ein weiteres *DAO*, eine *Repository* und einen *Service* (Abbildung 4.7).
Da diese Funktion in Zukunft noch erweitert werden kann, verwenden wir einen eigenen *Controller*, welcher auf den Unterpfad */stats* hören soll. Der Statistik-Bereich wird mit in diesem Autorisierungsserver verwirklicht, jedoch handelt es sich in der Realität um ein eigenständiges Modul. Da die beiden Module allerdings auf die gleiche Datenquelle zugreifen, macht es Sinn, diese in einem Projekt zu implementieren. Wir bieten



(a) ER-Diagramm *downloads*

(b) Klassendiagramm *downloads*

Abbildung 4.7: Aufbau der Downloads

diesen Bereich also als *Resouce Server* an und benutzen somit das OAuth2-Protokoll, um die Nutzer an diesem Pfad zu autorisieren.
Abbildung 4.8 zeigt ein UML-Sequenzdiagramm, welches den Vorgang anschaulich darstellt.

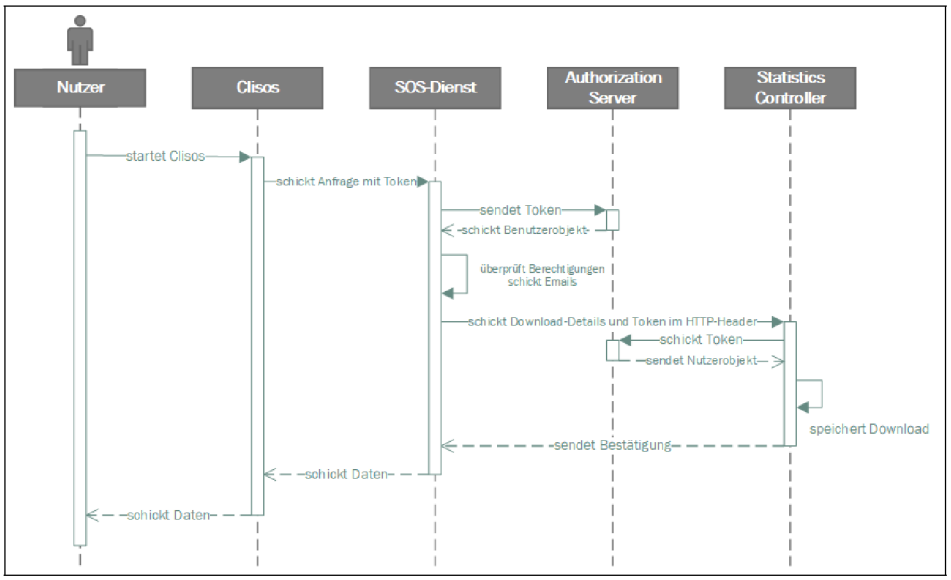


Abbildung 4.8: UML-Sequenzdiagramm Statistikerfassung

Die *StatisticsController*-Klasse muss von *ResourceServerConfigurerAdapter* erben und

wird mit den *Annotationen* aus Listing 4.42 versehen.

```
@RestController
@RequestMapping("/stats")
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class StatisticsController extends
    ResourceServerConfigurerAdapter {[...]
```

Listing 4.42: StatisticsController.java Annotationen

@RestController Dieses *@Interface* vereint @Controller und @ResponseBody. Somit werden alle Rückgabewerte von Methoden direkt als HTML-Antwort interpretiert.

@RequestMapping Wie bereits bekannt, geben wir hiermit den Pfad an, auf den dieser *Controller* reagiert.

@EnableResourceServer Mit dieser Annotation wird eine neue *WebSecurity*-Bean erzeugt, welche für diesen Bereich zuständig ist und Diesen mit Hilfe der *Access Token* sichert.

@EnableGlobalMethodSecurity Wird benötigt, um jeder Methode bestimmte Berechtigungen zuweisen zu können, die der Benutzer aufweisen muss, um sie zu benutzen.

Da ein neuer *WebSecurity*-Bean erstellt wurde, konfigurieren wir diese Einstellungen hier.

```
    public void configure(HttpSecurity http) throws Exception {
        http
          .requestMatchers().antMatchers("/stats/**")
          .and()
          .authorizeRequests()
          .antMatchers("/stats/**")
          .authenticated()
          .antMatchers("/stats/**")
          .access("hasIpAddress('127.0.0.1/32') or hasIpAddress
              ('134.94.0.0/16')")
          .and()
          .csrf().ignoringAntMatchers("/stats/**");

    }
```

Listing 4.43: StatisticsController.java Security

Zunächst geben wir den Pfad an, auf den die Konfiguration reagieren soll (Zeilen 3-5). Danach lassen wir in dem gleichen Pfad alle ankommenden Anfragen autorisieren (Zeile 6). Die Autorisierung erfolgt hier über OAuth2. Da wir genau wissen, welche Adressen in Zukunft auf diesen Endpunkt zugreifen, können wir den Zugriff auf diese IP-Adressen beschränken (Zeile 9). Als Letztes müssen wir den Gebrauch von *CSRF*-Token abschalten, da wir keine *GET*-Methode definieren, die diesen Token vorher überträgt (Zeile 11).
Wir erstellen eine Methode, die sich um das Hinzufügen neuer Downloads kümmert.

```
1    @PreAuthorize("!#oauth2.client and #oauth2.clientHasRole('
        ROLE_SOS_ACCESS')")
2    @RequestMapping(value = "/addDownload",method = RequestMethod.POST
        )
3    public void addDownload(Principal user, @Validated @RequestBody
        Downloads download) {
4
5        download.setDownloadtime(new Date());
6        downloadsService.saveDownload(download);
7
8    }
```

Listing 4.44: StatisticsController.java *addDownload*-Methode

Mit *@PreAuthorize* kann man Autorisierungen vor Aufruf der Methode prüfen. Wir beschränken den Zugriff auf *User*, die mit einem *Client*, welcher mindestens die Rolle *SOS_ACCESS* besitzt, auf die Ressource zugreifen wollen. Die SOS-Dienste schicken den Token der Benutzer einfach weiter an diesen Statistik-Controller und sind somit als diese autorisiert. Die Statistiken werden im *POST*-Body übertragen. Spring verpackt die Parameter dann automatisch in ein neues *Downloads*-Objekt. Mit diesen Informationen wird der Download, nachdem die aktuelle Zeit eingetragen wurde (Listing 4.44, Zeile 5), in der Datenbank abgelegt (Listing 4.44, Zeile 6).

4.2.7 Token Informationsinhalt

Der Rückgabewert des **/oauth/check_token**-Endpunkts wird über einen *AccessTokenConverter* gesteuert. Dieser kann in der *configure*-Methode der *AuthorizationServerConfiguration*-Klasse angegeben werden (Listing 4.41, Zeile 7). Da wir mehr Informationen brauchen, als dieser Endpunkt normalerweise zur Verfügung stellt, müssen wir diese Konfiguration durchführen. Wir erstellen eine Klasse *CustomAccessTokenConverter*, die von *DefaultAccessTokenConverter* erbt. Wir müssen lediglich die *convertAccessToken*-Methode überschreiben. Zunächst führen wir die super-Methode aus und arbeiten mit deren Rückgabewert weiter, da wir die aktuell verfügbaren Informationen nur erweitern möchten.

Hinweis: LDAP-Nutzer werden normalerweise auf Objekte der Klasse *User*, welche keinerlei Zusatzinformationen enthalten, abgebildet. Um die anderen Informationen

sichtbar zu machen, müssen wir das *Principal*-Objekt durch Hinzufügen der Zeile 4 in Listing 4.15 zum Typ *InetOrgPerson* konvertieren. Die Angegebene Methode erzeugt ein neues Objekt vom Typ *InetOrgPersonContextMapper*.

Der *TokenConverter* überprüft nun, ob das *Principal*-Objekt vom Typ *InetOrgPerson*, *TemporaryUser* oder *User* ist, und fügt die jeweilig verfügbaren Informationen zu der *Map* hinzu, welche wir anschließend zurückgegeben.

```
1  if (principal instanceof User){
2    User user = (User) principal;
3    castedResult.put("mail", user.getEmail());
4    castedResult.put("first_name",user.getFirstName());
5    castedResult.put("last_name",user.getLastName());
6    castedResult.put("purpose", user.getPurpose());
7    castedResult.put("institution",user.getInstitution());
8  }
```

Listing 4.45: CustomAccessTokenConverter.java

Bei der Bearbeitung von Anfragen werden diese Informationen dann in die Antwort eingeschlossen.

4.3 SOAP-Dienst

Der SOAP-Dienst ist die Schnittstelle für **Samplify** und die **Samplify App** und stellt die Klasse *SoapOdmServiceFacade* bereit, welche als *@WebService* annotiert ist. Die veröffentlichten Methoden wurden über die *@WebMethod* Annotation gekennzeichnet. **JAX-WS** veröffentlicht mit diesen Angaben automatisch die **WSDL**-Datei und die zugehörigen Schnittstellen. Bei jedem Methodenaufruf bzw. bei jeder Anfrage wird zunächst das zu Spring äquivalente *Principal*-Objekt generiert. Im Fall des SOAP-Servers ist es vom Typ **SampleManagementAccount**, welcher von der Klasse **AuthentificationAccount** erbt und Teil des *IbgJavaLibrary*-Pakets ist.

```
1        SampleManagementAccount account = this.authenticate();
```

Listing 4.46: SoapOdmServiceFacade.java Generierung des *Principal*-Objekts

Die *authenticate*-Methode ruft eine weitere Routine, deren Aufgabe es ist den *Authorization Header* aus der HTTP-Anfrage zu extrahieren und daraus ein *SampleManagementAccount*-Objekt zu erstellen, auf. Diese Methode heißt **extractAccountInfoFromSoapHeader**.
Die finale Version dieser Methode wird in Listing 7.13 dargestellt. Zunächst wird der Nachrichtenkontext durch autoinjizierte Klassenvariable **wsctx** vom Typ *WebServiceContect* abgerufen.

```
1    @Resource
2    private WebServiceContext wsctx;
```

Listing 4.47: SoapOdmServiceFacade.java WebServiceContext

Dieser Kontext hält die Parameter der Anfrage, die zur Zeit bearbeitet wird. Der HTTP-Header wird extrahiert und eine Liste mit allen *Authorization*-Header Elementen wird aufgebaut. Das erste Element wird verwendet um, den *Base64* kodierten String zu dekodieren und an dem Doppelpunkt aufzuteilen. Der erste Teil des resultierenden String-Array ist der Benutzername und der zweite Teil ist das dazugehörige Passwort.
In der *Constants*-Klasse werden die meisten Konstanten und statischen Variablen gehalten. So auch das aktuell verwendete *AbstractAuthentication*-Objekt, welches die eigentliche Authentifizierung durchführt.

```
this.authentication = AbstractAuthentication.factory([...]);
```

Listing 4.48: Constants.java

Dieses Objekt wird im Konstruktor der *Constants*-Klasse über eine *Factory*-Methode erstellt und benutzt den lokalen LDAP-Server des Instituts, um die Benutzer zu authentifizieren und bildet die dem Nutzer zugeordneten *ActiveDirectory*-Gruppen auf das *Principal*-Objekt ab.

Wir müssen also die abstrakte Klasse *AbstractAuthentication* um eine Implementierung erweitern, diese über die *factory*-Methode anbieten und vorher in der *extractAccountInfoFromSoapHeader*-Methode der *SoapOdmServiceFactory*-Klasse überprüfen, ob es sich um einen normalen *Authentication*-Header handelt oder ein *Bearer*-Token übertragen wurde. Dies ist nötig, da wir wollten, dass der Dienst rückwärtskompatibel bleibt.

Die Klasse *AbstractAuthentication* befindet sich in dem Maven-Projekt *org.fzj.ibg3 : IbgJavaLibrary*. Um die Auswahl der jeweiligen Authentifizierungsmethode schöner zu gestalten, erstellen wir eine *Enum*-Klasse mit den entsprechenden Elementen.

```
public enum AuthenticationMethod {
    BASIC,
    LDAP,
    ACTIVE_DIRECTORY,
    ACTIVE_DIRECTORY_WITH_OAUTH
}
```

Listing 4.49: AuthenticationMethod.java Enum

Das letzte Element ist dabei die Klasse, welche wir gleich implementieren werden. Wir machen zunächst eine Kopie der *ActiveDirectoryAuthentication*-Klasse, da dies die einzige Authentifizierungs-Methode ist, die OAuth2-Unterstützung erhalten soll. Wenn die Migration im gesamten System abgeschlossen ist, soll eine reine *OAUTH*-Authentifizierung umgesetzt werden. Die alten Verfahren werden dann sukzessive abgeschafft.
Die Klasse *ActiveDirectoryOAuthAuthentication* erbt von *ActiveDirectoryAuthentication* und wird gleich initialisiert. Die **getAccount**-Methode wird allerdings überladen.

```
1    public AuthentifcationAccount getAccount(String json) throws
         NamingException, IbgException {
2        if (!json.trim().startsWith("{")) {
3           return super.getAccount(json);
4        }
5        JSONObject user = new JSONObject(json);
6        AuthentifcationAccount result = this.accountFactory.
             createAccount();
7        result.setUsername(user.getString("user_name"));
8        for (int i = 0 ; i < authorities.length() ; i++){
9            Object authority = authorities.get(i);
10           if (authority instanceof String){
11               result.addGroup((String) authority);
12           }
13       }
14
15       return result;
16   }
```

Listing 4.50: ActiveDirectoryOAuthAuthentication.java getAccount

Ein *JSON*-Objekt beginnt per Definition mit einer geschwungenen Klammer ({). Ein Nutzername kann nicht mit einem Sonderzeichen beginnen. Somit können wir an diesem Merkmal unterscheiden, ob es sich um eine normale *ActiveDirectory*-Authentifizierung oder um eine *OAuth2*-Authentifizierung handelt. Im ersten Fall wird die *super*-Methode aufgerufen und somit nach dem alten Schema verfahren. Wenn der zweite Fall eintritt, werden die Informationen aus den übergebenen *JSON*-Objekt extrahiert und in das *AuthentificationAccount*-Objekt übertragen. Der *factory*-Methode wird ein **switch**-Block hinzugefügt und die einzelnen Elemente des *Enum* werden implementiert.

```
1  AbstractAuthentication _this = null;
2  [...]
3  public static AbstractAuthentication factory(AuthenticationMethod
       authMethod,[...]) {
4    if (_this == null) {
5       switch (authMethod) {
6           [...]
7           case ACTIVE_DIRECTORY:
8               _this = new ActiveDirectoryAuthentication([...]);
9               break;
10          case ACTIVE_DIRECTORY_WITH_OAUTH:
11              _this = new ActiveDirectoryOAuthAuthentication([...]);
12      }
13   }
14   return _this;
```

```
15 }
```

Listing 4.51: AbstractAuthentication.java

Wir veröffentlichen diese Version des Projekts über den institutsinternen Repository-Server und machen die Änderungen so in unserem SOAP-Server durch eine Versionsanpassen in der *pom.xml* verfügbar.
Wir passen die Erstellung des *AbstractAuthentification*-Objekts durch das Hinzufügen des gewählten Enum-Elements an.

```
1 this.authentication = AbstractAuthentication.factory(
     AuthenticationMethod.ACTIVE_DIRECTORY_WITH_OAUTH,[...]);
```

Listing 4.52: Constants.java mit OAuth2

Die *extractAccountInfoFromSoapHeader*-Methode wird um die Unterscheidung zwischen *Basic* und *Bearer Authentication* erweitert. Dies wird mit einem *if-else if*-Block erreicht.

```
1            if (authString.startsWith("Basic")) {[...]
2            } else if (authString.startsWith("Bearer")) {
3                String encoded = authString.trim().substring(6).trim();
4                String jsonString = doGet(encoded);
5                JSONObject userObject = new JSONObject(jsonString);
                     [...]
6            }
```

Listing 4.53: SoapOdmServiceFacade.java WebServiceContext

Das JSON-Objekt enthält die gleichen Informationen wie das LDAP-Objekt, da wir den Inhalt des Token im *Authorization Server* angepasst haben (Kapitel 4.2.7). Das Objekt kann dementsprechend initialisiert werden.

Die letzte zu klärende Frage betrifft den Inhalt der **doGet**-Methode (Listing 7.14). Hier wird der übergebene *Access Token* an die richtige Schnittstelle des *Authorization Server* geschickt. Diese Anfrage enthält die *ID* und das *secret* des für den SOAP-Dienst generierten *Client*. Mit diesen Informationen erhält der Dienst das hinter dem Token stehende *Principal*-Objekt im JSON Format. Um die Anzahl der Anfragen an den *Authorization Server* zu minimieren wird ein lokaler Zwischenspeicher verwendet. Das Ergebnis der Tokenverifizierung wird für einen Tag gespeichert. Da es sich um einen komplett internen Prozess handelt, kann davon ausgegangen werden, dass in diesem Zeitraum kein Token widerrufen wird.
Die Änderungen am SOAP-Dienst sind somit abgeschlossen.

4.4 Samplify

Auch in **Samplify** wollen wir sicherstellen, dass die Rückwärtskompatibilität gegeben ist. Bisher wurde ein Popup bei der Initialisierung des Programms geöffnet, welches

nach dem Benutzernamen und Passwort des Nutzers gefragt hat. Da wir diese beiden Angaben aufgrund der Kompatibilität auch weiterhin benötigen, bietet sich hier das *resource owner password credentials* Verfahren an. Die abgefragten Zugangsdaten wurden bisher im Arbeitsspeicher abgelegt und bei jeder Anfrage an den Dienst in den HTTP Authorization-Header geschrieben. Auch in diesem Projekt gibt es eine *Constants*-Klasse. Deren Variablen werden ebenfalls aus einer Konfigurationsdatei bei Programmstart geladen. Um unterscheiden zu können, welches Autorisierungsverfahren verwendet werden soll, fügen wir eine Variable zu der Konfigurationsdatei hinzu. Wenn der Wert **true** ist, wird überprüft, ob die anderen OAuth2 Parameter vorhanden sind.

OAUTH_ENABLED Gibt an, ob das OAuth2-Verfahren verwendet werden soll.

OAUTH_ENDPOINT Für das verwendete Autorisierungsverfahren wird der Token-Endpunkt benötigt und mit diesem Parameter angegeben.

ACCESS_TOKEN Hält den früher erzeugten *Access Token*.

ACCESS_TOKEN_EXPIRES_AT Wenn ein *Access Token* gespeichert ist, wird mit diesem Parameter der Zeitpunkt des Ablaufs in Millisekunden angegeben.

CLIENT_ID Enthält die *Client-ID*. Dieser Parameter ist optional, da der Standartwert im Programm steht. Falls sich dieser Wert ändert, muss man nur diesen Parameter setzen, anstatt den Code neu zu kompilieren.

CLIENT_SECRET Siehe CLIENT_ID.

Zunächst erstellen wir eine neue Klasse, die für die Kontrolle der OAuth2-Funktionalität zuständig ist. Wir nennen diese Klasse **OAuth2Controller**.
Der Einstiegspunkt des Programms ist die *Constants*-Klasse. Hier wird durch **JAX-WS** zunächst ein Objekt erzeugt, welches das Interface **SoapOdmServiceFacade** implementiert. Dieses Interface gibt alle Methoden des SOAP-Webdienstes vor und implementiert zusätzlich das *BindingProvider*-Interface, welches dafür genutzt werden kann, die HTTP-Header aller SOAP-Anfragen zu steuern. Diese Einstellungen werden in der **setSoapSecurity**-Methode vorgenommen. Bevor der Nutzer nach den Zugangsdaten gefragt wird, fügen wir einen *if-else*-Block hinzu, welcher überprüft, ob *OAUTH_ENABLED true* ist. Wenn das der Fall ist, erstellen wir einen neuen *OAuth2Controller* und übergeben dem Konstruktor das *BindingProvider*-Objekt und die *Constants*-Instanz. Wenn der Wert *false* ist, wird das Programm wie vorher weitergeführt.
Es wird geprüft, ob bereits ein *Access Token* vorhanden ist. Wenn nicht, wird über die Methode *getAccessTokenViaPasswordAuth* versucht, ein *Access Token* zu erzeugen. Um das zu erreichen, benötigen wir die Zugangsdaten des Nutzers. Wir implementieren eine Methode in der *Constants*-Klasse, welche uns erlaubt, das Login-Popup zu

öffnen, die Daten abzufragen und ein *String*-Array mit dem entsprechenden Benutzernamen und Passwort zurück gibt. Diese zwei Parameter werden an die *obtainAccessToken*-Methode übergeben. Dort wird ein *OAuthClientRequest*-Objekt erzeugt und mit den Parametern der Anfrage gefüllt.

```
1            OAuthClientRequest request = OAuthClientRequest
2                    .tokenLocation(Constants.OAUTH_ENDPOINT)
3                    .setGrantType(GrantType.PASSWORD)
4                    .setScope("read,write")
5                    .setUsername(userName)
6                    .setPassword(password);
```

Listing 4.54: OAuth2Controller.java OAuthClientRequest

Da Spring die Client-*ID* und das Client-*secret* im HTTP *Authorization-Header* erwartet, wird dieser angegeben. Wenn die Anfrage erfolgreich war, erhalten wir das in Listing 4.55 zu sehende Objekt als Antwort.

```
1            OAuthJSONAccessTokenResponse oAuthJSONAccessTokenResponse =
                 oAuthClient.accessToken(request);
```

Listing 4.55: OAuth2Controller.java Response-Objekt

Es hält den generierten *Access Token*, den Token Typ (Bearer), die Lebenszeit in Sekunden und das Scope.

```
1 {
2     "access_token":"5d0ccfff-5dbd-4475-8d82-e6f5aaa3f744",
3     "token_type":"bearer",
4     "expires_in":7197,
5     "scope":"read"
6 }
```

Listing 4.56: Json-Objekt

Wenn kein Fehler geworfen wurde, wird dieses Objekt zurückgegeben und anschließend benutzt, um den Token zu speichern und den Zeitpunkt, an dem der Token abläuft, zu berechnen und abzulegen. Auf diese Weise weiß das Programm, wann es einen neuen *Access Token* generieren muss.
Sollte jedoch ein Fehler auftreten (Passwort falsch eingegeben, etc.), wird die Methode rekursiv aufgerufen, bis die Authentifizierung entweder funktioniert hat oder 5 Versuche überschritten wurden. Sollte das Programm zu diesem Zeitpunkt noch in der Initialisierungsphase sein, so wird es geschlossen. Wenn man sich bereits erfolgreich eingeloggt hat, kann man die Instanz nicht einfach schließen, da aktuelle Änderungen eventuell noch nicht gespeichert wurden. Es muss also unterschieden werden, ob das Programm komplett initialisiert ist oder nicht. Dies erreichen wir über ein Flag, welches bei der erfolgreichen Anmeldung gesetzt wird. Wenn während des Programmgebrauchs ein neuer *Access Token* vonnöten ist, wird das gleiche Login-Popup geöffnet. Hier wird im Fehlerfall allerdings eine neue *RuntimeException* geworfen und

der Fehler-Zähler wird zurückgesetzt. Die *RuntimeException* sorgt im Gegensatz zu der normalen *Exception* dafür, dass eine Fehlermeldung angezeigt und geloggt wird, der Programmablauf danach jedoch nicht weiter beeinflusst wird. Ein sehr ähnliches Problem taucht auf, wenn man in dem Login-Popup auf den Knopf *Abbrechen* klickt. Vorher wurde das Programm einfach beendet. Aus dem gleichen Problem, wie oben kann man nun nicht mehr so verfahren. Ein weiteres Problem stellt hier der mit einem *Button* verknüpfte *ActionListener* dar. Da dies ein *Interface* ist und in der entsprechenden Methodendeklaration keine *Exception* geworfen wird, kann man hier nicht den gleichen Lösungsansatz verwenden wie zuvor. Außerdem muss man im weiteren Verlauf unterscheiden, ob es sich um einen Abbruch der Eingabe oder um einen anderen Fehler handelt. Wenn die Passworteingabe abgebrochen wird, soll logischerweise kein neues Login-Fenster geöffnet werden. Im Falle eines anderen Fehler, sehr wohl. Wir benötigen im Passwortdialog also ein Flag, welches den Wunsch des Nutzers nach einem Abbruch signalisiert. Das Login-Fenster wird dann geschlossen und eine *LoginAbortedException* wird erzeugt und geworfen. In dem *OAuth2Controller* wird dieser Fehler dann explizit gefangen, was den Vorgang beendet. Wenn es sich um einen anderen Fehler handelt, wird der Vorgang ebenfalls bis zu 5 mal wiederholt und anschließend abgebrochen, wenn er nicht erfolgreich war. Damit erkannt werden kann, dass ein Token seine Gültigkeit verloren hat, muss er regelmäßig auf Aktualität überprüft werden. Da der *Access Token* nur bei der Verwendung von Methoden des SOAP-Dienstes verwendet wird, macht es Sinn, diese Überprüfung vor jedem Aufruf einer solchen Methode auszuführen. Um dieses Verhalten zu implementieren, schreiben wir eine Wrapper-Klasse für das *SoapOdmServiceFacade*-Objekt. Wir entwickeln die Klasse *SoapOdmServiceFacadeOauthWrapper*, die *SoapOdmServiceFacade* implementiert und ein Objekt des gleichen Interface im Konstruktor erwartet. Alle Methoden der Schnittstellendefinition werden implementiert und in jeder Methode wird die gleichnamige Routine des übergebenen Objekts aufgerufen bzw. deren Rückgabewert, wenn vorhanden, zurückgegeben. Vor diesem Aufruf wird allerdings noch die *checkTokenExpiry*-Methode aufgerufen.

```
1   private void checkTokenExpiry() throws IbgException {
2       if (System.currentTimeMillis() > Constants.
            ACCESS_TOKEN_EXPIRES_AT){
3           try {
4               OAuth2Controller.reinit();
5           } catch (LoginAbortedException e) {
6               throw new IbgException("Login aborted!",null);
7           }
8       }
9   }
```

Listing 4.57: SoapOdmServiceFacadeOauthWrapper.java

Auch in dieser Methode muss beachtet werden, dass der Nutzer die Eingabe möglicherweise abbrechen möchte. Die *Exception* wird aus diesem Grund auch hier weitergeleitet. Wenn der *Access Token* seine Gültigkeit verloren hat, wird die *reinit*-Methode

der *OAuth2Controller*-Klasse aufgerufen. Diese stößt den gleichen Prozess der Tokenerzeugung an, wie bei der Erstinitialisierung, jedoch wird das Programm bei einem Fehler nicht beendet (s.o.).

4.5 SOS

Das SOS-Projekt besteht aus mehreren Maven-Modulen.

52n-sos-coding Ist für die Konvertierung verschiedener XML-Schemata zuständig.

52n-sos-core Enthält die Kernkomponenten des Services

52n-sos-dao-postgis Ermöglicht den Zugriff auf die *Postgres*-Datenbank mit *postgis* Erweiterung.

52n-sos-ogc Enthält die Spezifikationen.

52n-sos-service Enthält den Webdienst.

Wie man sehen kann, ist das Projekt sehr modular aufgebaut. Man kann es durch das Hinzufügen von Abhängigkeiten um einzelne Komponenten erweitern. Die losen Kopplungen werden durch *Dependency Injection* erreicht (so auch bei Spring). Wir können die Veränderungen auf das *core*-Modul beschränken, ohne Abhängigkeiten außerhalb des Moduls zu erzeugen. Einstiegspunkte für HTTP-Anfragen sind die Methoden **doGet** und **doPost**, die mit je einem Parameter vom Typ *HttpServletRequest* und *HttpServletResponse* aufgerufen werden.
Wir verwenden nur vier der definierten SOS Anfrage-Typen:

1. getCapabilities
2. describeSensor
3. getObservation
4. getFeatureOfInterest

Hierbei stellt nur der Typ *GetObservationRequest* eventuell beschränkte Daten zur Verfügung. Die jeweiligen Beschränkungen beziehen sich auf bestimmte Nutzergruppen, welche in *SensorML* Dateien beschrieben werden. Diese Dateien kann man über die *describeSensor*-Anfrage erhalten. Die Anfragen werden normalerweise als HTTP POST gestellt, weswegen wir uns zunächst auf die *doPost*-Methode als Einstiegspunkt konzentrieren können.
Als Erstes gehen wir wieder davon aus, dass im HTTP-Header ein *Authorization*-Element zu finden ist. Wir iterieren durch die einzelnen Einträge und suchen dabei nach diesem Element. Sollten keine Autorisierungsinformationen gefunden werden, kann man davon ausgehen, dass es sich um eine Anfrage vom Typ 1,2 oder 4 handelt und somit keine Authentifizierung nötig ist. Aus diesem Grund wird mit der

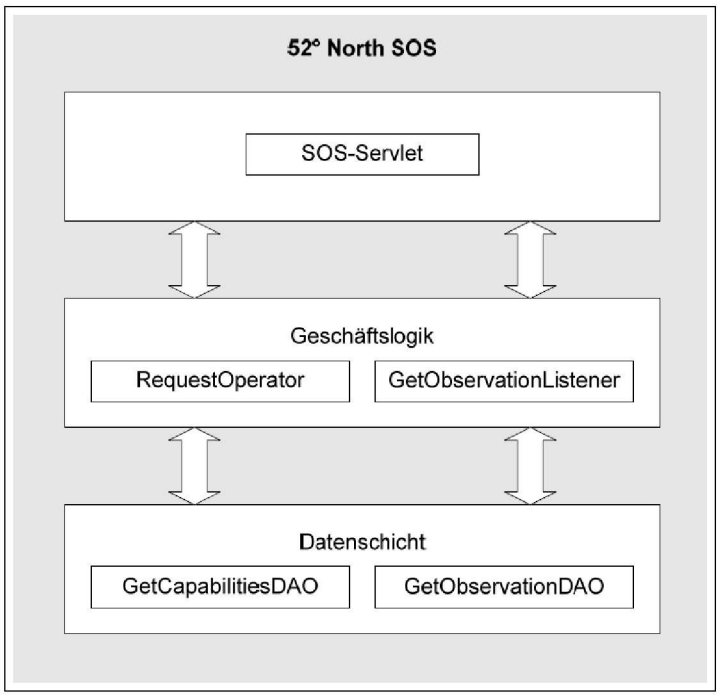


Abbildung 4.9: Quelle: [9] Abb.1 - Der Aufbau eines SOS

Programmausführung normal fortgefahren. Wenn ein jedoch Authorization-Header gefunden wurde, fragen wir das entsprechende *Principal*-Objekt über die bekannte Schnittstelle des *Authorization Server* ab. Wenn bei der Authentifizierung ein Fehler auftritt, wird die Verbindung direkt über das *HttpServletResponse*-Objekt mit dem HTTP Fehlercode 401 beendet.

Die Antwort auf die Anfrage wird über ein Objekt vom Typ *RequestOperator* erstellt, welches über die Methode *doPostOperation* dynamisch entscheidet, von welchem Typ die Anfrage ist und die entsprechende Antwort bestimmt. Das generierte *Principal*-Objekt vom Typ *JSONObject* wird bei jedem Methodenaufruf weitergegeben, um es in den Untermethoden verfügbar zu machen. Wir implementieren alle Methoden so, dass der ursprüngliche Methodenkopf weiterhin verfügbar bleibt, und die neu erstellte Methode mit einem *null Principal*-Objekt aufgerufen wird. So wird die Rückwärtskompatibilität gesichert. Die Anfrage wird in der *doPostOperation*-Methode von einem Objekt des Typs *IHttpPostRequestDecoder* dekodiert und in ein entsprechendes Anfrageobjekt geparst. Es wird geprüft, um welchen Typ es sich handelt, indem alle Möglichkeiten nacheinander überprüft werden.

```
1 if (request instanceof SosGetCapabilitiesRequest) {
2   requestListener = reqListener.get(Operations.getCapabilities.
        toString());
3 }
4 else if (request instanceof SosDescribeSensorRequest) {
5   requestListener = reqListener.get(Operations.describeSensor.
```

```
      toString());[...]
```

Listing 4.58: RequestOperator.java Anfragetypen

Der so ausgewählte *RequestListener* wird an eine weitere Methode (*getISosResponse*) übergeben, deren Rückgabewert zurückgegeben wird.
Von der aufrufenden Methode wird ein *requestListener* für den entsprechenden Anfragetyp übergeben. Wenn es sich um den zu sichernden Typ *SosGetObservationRequest* handelt, brauchen wir noch die dazugehörigen *describeSensor*-Dokumente, welche durch eine Anfrage vom Typ *SosDescribeSensorRequest* geliefert werden, um die passende Sensorbeschreibung abfragen zu können. Um nun auf diese Beschreibungen zugreifen zu können, müssen wir diesen *RequestListenerHelper* zusätzlich zu dem regulären *requestListener*, im Falle der *SosGetObservationRequest*-Anfrage, übergeben.

```
1 else if (request instanceof SosGetObservationRequest) {
2   requestListener = reqListener.get(Operations.getObservation.
       toString());
3   requestListenerHelper = reqListener.get(Operations.describeSensor.
       toString());
4 }
```

Listing 4.59: RequestOperator.java getObservation

Wir befinden uns nun in der *getISosResponse*-Methode.
Hier implementieren wir eine Methode (*checkForPermissions*), welche die Überprüfung der Rechte übernehmen soll. Sie bekommt das *Principal*-Objekt, das *Request*-Objekt und das zuvor erstellte *requestListenerHelper*-Objekt übergeben. Wenn dieses den Wert *null* hat, kann die Überprüfung beendet werden, da es sich nicht um eine *SosFetObservationsRequest*-Anfrage handelt und somit nicht gesichert werden muss. Andernfalls wird überprüft, ob das Nutzer-Objekt vorhanden ist. Wenn nicht, wird eine Antwort vom Typ *OwsExceptionReport* erstellt. Diese Klasse implementiert ebenfalls das *ISosResponse*-Interface und kann somit einfach zurückgegeben werden. Der Fehler wird dem Nutzer dann auf der Clientseite präsentiert. Für jedes Element der zu überprüfenden *Procedure*-Liste wird die *describeSensor*-Beschreibung mit Hilfe des *requestListenerHelper*-Objekts erzeugt. Sollte dabei ein Fehler auftreten, wird, wie zuvor, eine entsprechende Fehlermeldung zurückgegeben. So wird eine *Map<String, SensorDocument>* aufgebaut, welche die benötigten Daten hält. Sollte dieser Vorgang erfolgreich sein, wird eine Liste mit Gruppen des *Principal*-Objekts, eine Liste mit allen benötigten Gruppen und eine Liste mit den erforderlichen Berechtigungsstufen erzeugt.

Berechtigungsstufen: Datensätze haben verschiedene Freigabestufen:

U Unrestricted

R Restricted

S Secret

TS Top-Secret

Ab der Stufe *R* ist einem Datum mindestens einer bestimmten Gruppe zugeordnet. Der Nutzer, welcher das betreffende Datum herunterladen möchte, muss Mitglied alle dieser Gruppen sein. Anonyme Nutzer erhalten somit nur Zugriff auf Datensätze mit der Stufe U.

Wenn eine dieser Stufen über **R** liegt, wird geprüft, ob der Nutzer Mitglied aller Gruppen dieser *Procedure* ist. Wenn alle Elemente auf diese Weise prozessiert wurden und kein Fehler aufgetreten ist, gibt die Methode den Wert *null* zurück, was die erfolgreiche Autorisierung signalisiert. Ein Rückgabewert vom Typ *ISosResponse* würde ja einen Fehler signalisieren. Die Antwort wird durch das *requestListener*-Objekt erzeugt und implementiert das Interface *ISosResponse*, wobei die konkrete Implementierung vom Typ *ObservationResponse* ist. *ObservationResponse* enthält die XML-Antwort als reinen Text. Die entsprechenden Java-Objekte wurde verworfen, da sie normalerweise an dieser Stelle nicht mehr benötigt werden. In diesen Objekten befinden sich allerdings die Email-Adressen aller Beteiligten, sodass wir ein Interesse daran haben, diese Java-Elemente zu erhalten, um keine Suche im XML-String durchführen zu müssen.

```
1          response = requestListener.receiveRequest(request);
```

Listing 4.60: RequestOperator.java recieveRequest

Die benötigte Klasse ist *SosObservationCollection*. Also schreiben wir eine Wrapper-Klasse, welche von *ObservationResponse* erbt und das gewünschte Objekt über eine Setter-Methode übergeben bekommt. Die Interfaces bleiben durch dieses Vorgehen weiterhin implementiert. Durch einen Typ-Cast erhält man jedoch Zugriff auf die zugrundeliegenden Implementierungen der Interface-Objekte. Man muss das gewünschte Objekt natürlich vorher noch in diese Wrapperklasse „verpacken“.

```
1  ObservationResponseWithCollection oRWC = new
       ObservationResponseWithCollection([...]);
2  oRWC.setSosObservationCollection(obsCollection);
3  return oRWC;
```

Listing 4.61: GetObservationListener.java Wrapper-Klasse

Als nächstes wird die Methode *sendMailsToResponsibles* mit dem vorher erhaltenen *SOSObservationCollection*-Objekt, dem *Principal* und der eigentlichen *Anfrage* aufgerufen. Über einen Java-Stream werden für alle Prozeduren die jeweiligen *SensorML*-Beschreibungan aus der Datenbank geholt, die entsprechenden Email-Adressen extrahiert und eine Liste daraus erstellt. Mit dem *Principal* und *Request*-Objekt wird ein *MailSerice* erstellt, welchem die Liste aller Adressaten übergeben wird. Aus dem *User* werden die Parameter

- **user_name**
- **mail** und
- **institution**
- **purpose**

entnommen und aus der Anfrage die Parameter

- **phenomenon**
- **samples**
- **procedures**
- **offering**,
- **filter**

welche dann an alle angegebenen Adressaten verschickt wird.

```
The user Tobias Korf
of institution IBG-3
with email address t.korf@fz-juelich.de
has downloaded data
from Juelich_A_LANUV with following parameters:

Intended purpose: testing
Begin: 2018-03-27 11:41:36
End: 2018-04-28 11:41:36
Offering: Public
Station(s): Juelich_A_LANUV
Phenomenon(s): SurfaceWaterLevel
Sample filter: none
```

Listing 4.62: Benachrichtigung über Download

Hinweis:
An dieser Stelle ist es notwendig, den **/oauth/check_token**-Endpunkt des *authorization server* so anzupassen, dass die oben genannten Elemente auch enthalten sind. Ein LDAP-Benutzer hat ohne Änderungen beispielsweise keine Email-Adresse (siehe Kapitel 4.2.7).

Als letzter Schritt muss die Erfassung des Downloads in der Datenbank implementiert werden. Aus diesem Grund haben wir den **/stats/addDownload**-Endpunkt in Kapitel 4.2.6 implementiert. Die gleichen Informationen aus den Emails werden verwendet, um ein *Downloads*-Objekt mit den erforderlichen Informationen zu erstellen. Wir wollen dieses Objekt in das JSON-Format serialisieren und nutzen dafür die *Gson*-Bibliothek von Google. Alle Zeitangaben müssen mit der Annotation *@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ssZ")* versehen worden sein, um die Form vorzugeben, mit der diese serialisiert werden sollen. Das Gleiche haben wir auch im *Authorization Server* gemacht. Wir erstellen eine Hilfsklasse *HttpPostDownload* mit der statischen Methode *addDownload* , welche den *Access Token* des Nutzers und das erstellte *Downloads*-Objekt erhält. Es wird eine POST-Anfrage an die Schnittstelle

vorbereitet. Der *Header*-Eintrag **Content-type** muss auf den Wert **application/json** gesetzt werden und dem *Authorization*-Element wird der Token hinzugefügt. Die *Download*-Instanz wird über ein *ObjectMapper*-Objekt serialisiert und als *Body*-Element der HTTP-Anfrage angehängt. Die Anfrage wird durchgeführt und der *Status-Code* 200 wird erwartet. Wenn dies der Fall ist, werden die Daten an den Nutzer verschickt, im Fehlerfall wird er entsprechend benachrichtigt und erhält keinen Zugriff.

```
1     HttpPost httpPost = new HttpPost(SOS.STATS_DOWNLOAD_ENDPOINT);
2     String downloadJsonString = mapper.writeValueAsString(download);
3     httpPost.setEntity(new StringEntity(downloadJsonString));
4     httpPost.setHeader("Content-type", "application/json");
5     httpPost.setHeader("Authorization", bearerToken);
```

Listing 4.63: HttpPostDownload.java

4.6 Clisos

Clisos ist ein Kommandozeilenwerkzeug[8] für die Verwendung von SOS-Diensten (cliSOS). Es wurde in Python geschrieben. Es erzeugt HTTP-Anfragen, die von dem jeweils gewählten Dienst verarbeitet werden. Das Ergebnis wird aufgearbeitet und für Skripte optimiert ausgegeben. Wir erweitern das Programm zunächst so, dass ein *Access Token* mit über die Kommandozeile angegeben werden kann, und speichern die übergebene Zeichenkette als lokalen Parameter. Um die Eingabe zu *parsen* wird das *getopt*-Paket verwendet, welches eine Verarbeitung von Parametern bei Programmstart ähnlich zu Der gestaltet, die das *Bash*-Programm unter Linux verwendet.

```
1    elif o == "-B":
2        bearerToken = a
```

Listing 4.64: clisos.py *Access Token* Übergabe

Der *Access Token* wird dem Konstruktor der Anwendung übergeben und dort als Klassenvariable abgelegt. In der Methode *getObservation* wird die Durchführung eines *getObservationRequest* vorbereitet. Die eigentliche HTTP-Anfrage wird in der Klasse *GetObservation* gestellt, genauer genommen in der Super-Klasse *SosRequest*. Der Token wird auch hier dem Konstruktor übergeben. Die Methode *getResponse* wird aufgerufen, um eine Antwort des SOS-Dientes zu erhalten. Wir müssen hier überprüfen, ob ein *Access Token* vorhanden ist und diesen dann in den *Authorization*-Header schreiben.

```
1        if self._bearerToken is None:
2            headers = {"Content-type":"application/xml"}
3        else:
```

[8]Command-line interface

```
4            headers = {"Content-type":"application/xml", "
                 Authorization":"Bearer "+self._bearerToken}
5        con.request("POST", self._url, self.getRequest(), headers)
```

Listing 4.65: sosclient.py *Authorization*-Header

4.7 Data Discovery Portal

Das *Data Discovery Portal* ist in mehrere Maven-Pakete mit der *groupId org.fzj.ibg3* unterteilt.

ibg.gwt.library Enthält die GWT-Deklarationen, die im HauptProjekt implementiert werden.

mapviewer Enthält die Logik zur Darstellung der Karten.

IbgJavaLibrary Enthält mehrere wiederverwendbare Java Implementierungen. (vgl. Kapitel 4.3)

An den ersten beiden Paketen werden wir Änderungen vornehmen.
Durch die Auswahl einer Messstation auf der dargestellten Karte wird ein Dialog geöffnet, welcher es ermöglicht die Verschiedenen Parameter der Suche anzupassen. Wenn Daten zu den angegebenen Filtern heruntergeladen werden sollen kann man den Prozess durch die Angabe seiner Daten im *DisclamerDialog* anstoßen, welcher sich in dem **Mapviewer**-Projekt befindet. Wir müssen einen Knopf hinzufügen, welcher es ermöglicht einen *Access Token* zu generieren. Außerdem benötigen wir ein Eingabefeld für diesen Token, welcher als Cookie gespeichert werden soll. Dies erstellen wir mit der *GWT*-Klasse **TextBox**. Wenn ein Token eingegeben wurde, wird dieser in die Cookies geschrieben, wenn der entsprechende *EventListener* ausgeführt wird.

```
1        private TextBox textOAuthToken;[...]
2        this.textOAuthToken = new TextBox();
3        JTools.setCookieValue(OAUTH_TOKEN_COOKIE_NAME, textOAuthToken.
             getText());
```

Listing 4.66: DisclamerDialog.java

Auërdem müssen wir der *SosObservationView*-Klasse eine *CallBack*-Klasse hinzufügen, die sich um die Behandlung der Tokengenerierung kümmert, wenn der *Request Token*-Knopf gedrückt wurde. Sie ruft die CreateOAuthToken-Methode der *ServiceFacade* auf. Diese Methoden, also die eigentliche Methode und die dazugehörige *CallBack*-Methode, müssen wir den entsprechenden GWT-Interfaces im Projekt *ibg.gwt.library* hinzufügen, damit sie sichtbar werden.

```
1 String createOAuthToken(GetObservationArguments args) [...]
```

Listing 4.67: SensorObservationService.java

```
1 void createOAuthToken(GetObservationArguments args, AsyncCallback<
    String> callBack) [...]
```

Listing 4.68: SensorObservationServiceAsync.java

-

Implementiert wird die neu hinzugefügte Methode in dem eigentlichen *Data Discovery Portal*-Projekt. Die *createOAuthToken*-Methode schickt eine GET-Anfrage mit den übergebenen Werten an den *Authorization Server*, welcher den Token generiert und per Email verschickt.
Sollte im Mapviewer ein Token eingetragen sein und der Nutzer drückt auf den *Ok*-Knopf, so wird wie vorher verfahren, nur dass in der Anfrage an den *SOS* der Token mit im HTTP-Header verschickt wird.

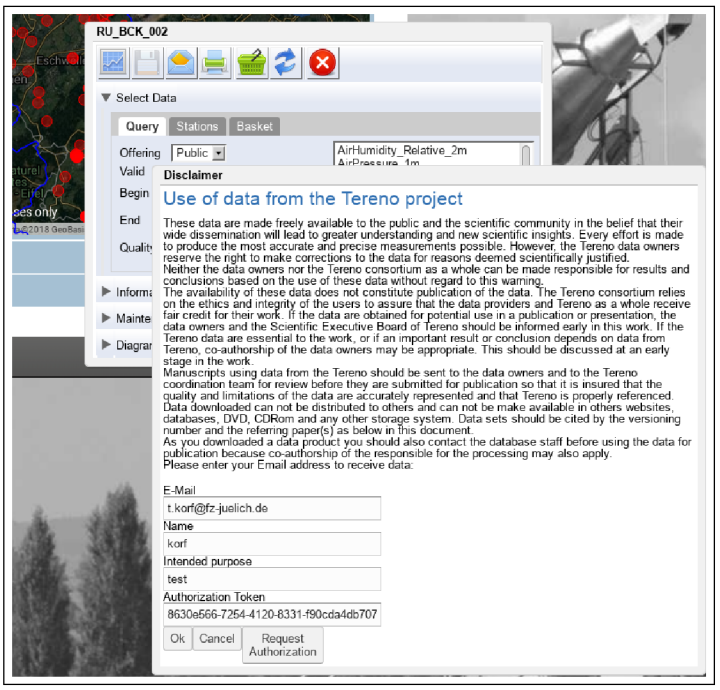


Abbildung 4.10: Data Discovery Portal

4.8 Sicherheit

Bei einem solchen Projekt sind viele Aspekte der Sicherheit zu beachten. Kritische Punkte sind hier:

- Die Veröffentlichung der Dienste
- Die Verarbeitung von Nutzerdaten
- Die Speicherung von Passwörtern

- Schwachstellen in Fremdsoftware

Aus diesen Gründen sollte man darauf achten, dass sämtlicher Datenverkehr über SSL/TLS geschützt wird. *Access Token* und *Passwörter* werden nur durch dieses Verfahren bei der Übertragung geschützt.
In Spring wird dies über die Konfiguration der *WebSecurity*-Klasse realisiert.

```
1  protected void configure(HttpSecurity http) throws Exception {
2      http[...]
3              .requiresChannel()
4          .antMatchers("/**").requiresSecure();
5  }
```

Listing 4.69: WebSecurityConfiguration.java SSL/TLS erzwingen

Bei der Speicherung von Passwörtern sollte man immer darauf achten ein aktuelles Verfahren dafür zu verwenden. Das Passwort muss mindestens gehasht und mit eine Salt versehen worden sein. Zur Zeit wird empfohlen **Argon 2** für die Verarbeitung von Kennwörter zu verwenden (Kapitel 6.6).
Auch HTTP-Anfragen muss man schützen. Zum einen muss sichergestellt werden, dass die Benutzerregistrierung gegen Missbrauch geschützt wird und kein Roboter Nutzer anlegen kann (**Captcha** Kapitel 6.5) und zu Anderen muss sich gegen bekannte Angriffe, wie **CSRF** geschützt werden (Kapitel 6.4). Auch die Daten der Nutzer müssen geschützt werden. Hierzu müssen die Richtlinien der **DSGVO** eingehalten werden.

5 Zusammenfassung und Ausblick

In dieser Masterarbeit wurde gezeigt, wie ein Autorisierungs- und Authentifizierungssystem auf Spring-Basis für eine verteilte Dateninfrastruktur in Java entwickelt wird. Außerdem wurde detailliert aufgezeigt, welche Komponenten das OAuth2 Protokoll besitzt, und welchen Softwarekomponenten diese entsprechen. Die notwendigen Änderungen wurden dokumentiert und für jedes Projekt, das geändert wurde, anschaulich erklärt. Die Implementierung der Softwarekomponente des im OAuth2-Protokoll definierten *Authorization Servers* wurde komplett durchgeführt und konfiguriert, wobei aufgetretene Komplikationen gezeigt und eine Möglichkeit zur Lösung gegeben wurde.

In Zukunft wird an der Umsetzung des OAuth2-Protokolls für alle Softwarekomponenten in der Dateninfrastruktur gearbeitet. Alle Klient-Programme und Dienste sollen durch das in dieser Arbeit implementierte System authentifiziert und autorisiert werden. Außerdem wird die Nutzung der Zugriffsteuerung durch die Verwendung verschiedener Rollen und deren Geltungsbereiche in Zukunft wesentlich granularer gestaltet. Auch die Erweiterung der Erfassung von verschiedenen Statistiken kann in dieses Projekt integriert werden. So ist es beispielsweise möglich, neben den Download-Statistiken auch andere Statistiken, wie beispielsweise das Nutzerverhalten, zu sammeln. Eine Auswertung der gesammelten Daten oder eine Schnittstelle, die diese zugänglich macht, wäre ebenfalls eine sinnvolle Erweiterung. Da der Autorisierungsserver in einem Docker-Container ausgeführt wird und somit nur limitierte Ressourcen zur Verfügung stehen, wird er ausgelagert und außerhalb einer virtuellen Umgebung betrieben um die Leistung zu verbessern.

6 Glossar

6.1 PostgreSQL & Postgis

PostgreSQL ist ein quelloffenes und kostenloses Relationales Datenbank Management System (RDBMS), welches den SQL-Standard weites gehend implementiert.
Es ist um Datentypen bzw. Funktionen erweiterbar. Eine dieser Erweiterungen lautet **PostGIS**. Dabei handelt es sich um eine Sammlung von Datentypen und Funktionen die es ermöglichen geographische Objekte abzubilden und zu verarbeiten.
Außerdem ist es **ACID**-konform. Eine **Transaktion** ist eine Folge von Operationen, die entweder ganz oder gar nicht ausgeführt werden. ACID beschreibt hierbei die englischen Begriffe

Atomicity *Atomarität bzw. Abgeschlossenheit*
Änderungen werden entweder ganz oder gar nicht ausgeführt (s.o.).

Consistency *Konsistenz*
Die Datenbank behält nach der Ausführung einer **Transaktion** die Konsistenz.

Isolation *Isolation bzw. Abgrenzung* und
Die Beeinflussung verschiedener Transaktionen wird ausgeschlossen, indem Daten gesperrt werden.

Durability *Dauerhaftigkeit*
Nach der Ausführung einer Transaktion müssen deren Änderungen dauerhaft gespeichert werden. Insbesondere Ausfälle des Hauptspeichers sind hier zu beachtende Größen.

.
Datenbanken können logisch durch **Schemata** unterteilt werden. Ein *Schema* kann man mit einem Verzeichnis eines Dateibaums vergleichen. Die globale Umgebung, also Benutzer, Passwörter und Berechtigungen, bleiben erhalten. Anfragen innerhalb einer Datenbank sind, auch über mehrere Schamata hinweg, kombinierbar (im Gegensatz zu verschiedenen Datenbanken). In einem *Schema* können mehrere **Tabellen** enthalten sein, welche die eigentlichen Daten halten. *Tabellen* sind in **Spalten** eingeteilt, welche einen bestimmten Datentypen haben. Eine **Zeile** der Tabelle entspricht einem gespeicherten Datensatz.

6.2 Maven

Maven ist ein Build-Management- und Software-Verständnis-Werkzeug, welches von der Apache Software Foundation entwickelt wird. Es basiert auf Java und ist zur standardisierten Erstellung und Verwaltung von Software, die ebenfalls in Java geschrieben wird, ausgelegt. Maven gibt Strukturen vor, die konsequent einzuhalten sind um einen möglichst einfachen und genormten Prozess während der Entwicklung von Software zu gewährleisten.

6.3 Thymeleaf

Thymeleaf ist eine serverseitige *Template Engine* und verarbeitet bestimmte Befehle und Strukturen in einer HTML-Seite. So kann der Inhalt einer sonst statischen Seite dynamisch erzeugt werden.

6.4 HTTP

Das **Hypertext Transfer Protocol** (HTTP) ist ein zustandsloses Protokoll zur Übertragung von Dateien in Netzwerken. Es stützt sich auf verschiedene Anfragemethoden wobei die gebräuchlichsten *GET* und *POST* sind.
Eine Anfrage besteht aus zwei Teilen, dem *Header* und dem *Body*. Im *Header* werden Metadaten zu der Verbindung gespeichert. Dies beinhaltet den Pfad, die Methode und Informationen über den Inhalt. Das Body-Element beinhaltet die eigentlichen Daten (zumeist HTML, XML oder andere Text-Inhalte). Auf eine HTTP-Anfrage erfolgt eine HTTP-Antwort, die den gleichen Aufbau besitzt. Zusätzlich wird jedoch noch eine Statuszeile zurückgegeben, die einen HTTP-Code enthält, wodurch man den Erfolg einer Anfrage ablesen kann.

CSFR-Angriff:
Um sicherzustellen, dass eine HTTP-Antwort nur dann verarbeitet wird, wenn vorher auch eine entsprechende Aktion, die diese Antwort erwartet, durchgeführt wurde, greift man in der Praxis auf **Cross-Site-Request-Forgery**-Token zurück.
Eine bösartige Seite kann beispielsweise eine HTTP-Anfrage auf einen anderen Server einleiten, wodurch Daten im Namen des Nutzers verändert werden können. Um dies zu verhindern wird bei jeder Anfrage des Nutzers ein *csrf*-Token mitgeschickt, welcher in der Antwort wieder erwartet wird. Ein Angreifer kann diesen Token nicht wissen bzw. in die bösartige Anfrage einschließen, was diesen Angriff verhindert.

6.5 Captcha

Ein **Captcha** wird dazu benutzt um festzustellen, ob es sich bei der aufrufenden Entität um eine Person oder um ein Programm handelt. Diese Überprüfung wird

häufig durchgeführt indem ein Bild mit abstrakten Formen und Figuren dargestellt wird in welchem sich eine Zeichenkette verbirgt die in ein Feld eingegeben werden muss. Ein Mensch kann diese Zeichenkette im Normalfall leicht erkennen, wohingegen ein Programm Schwierigkeiten durch diese zufälligen Formen und Figuren bei der Erkennung hat. Andere Möglichkeiten bieten Audio-Dateien, die durch Störungen beeinflusst werden und ebenfalls Zeichenketten zur Bestätigung wiedergeben. Der Nutzer muss die erfasste Zeichenkette eingeben. *Google* hat zudem eine automatisierte Technik (*reCAPTCHA*) entwickelt, die automatisch anhand geheimer Parameter überprüfen sollen, ob es sich um eine Person handelt oder nicht. Nur im Zweifelsfall wird der Nutzer noch dazu aufgefordert ein Puzzle zu lösen, Bilder zu sortieren oder Objekte anzuordnen um die Anfrage freizugeben.

6.6 Argon 2

Argon2 ist eine kryptographische Funktion, die zur Ableitung von Passwörtern verwendet wird und die *Password Hashing Competition* im Jahr 2015 gewann. Ziel dabei ist es, das Hashing eines Passwort sehr zeit und rechenintensiv umzusetzen um den Umkehrprozess umso schwerer zu gestalten.
Für die Initialisierung werden folgende Parameter benötigt:

Passwort Das eigentliche Passwort

Salt Der Teil, welcher dem Passwort angehängt wird, um verschiedene Hash-Werte, selbst bei dem gleichen Passwort, zu erzeugen.

Threads Die Anzahl der Threads, die verwendet werden sollen.

Speichergröße Die zu verwendende Speichermenge.

Iterationen Die Anzahl der durchzuführenden Iterationen.

6.7 zxcvb

Zxcvbn ist ein Schätzer für die Stärke und Komplexität von Passwörtern. Die Dokumentation befindet sich hier [12].

6.8 JAX-WS

JAX-WS (*Java API for XML Web Services*) ist eine Java-API für die Erstellung von Webdiensten.

6.9 GWT

GWT (*Google Web Toolkit*) ist ein von Google entwickeltes Framework für die Erstellung von Webanwendungen in der Programmiersprache Java. Die Anwendung wird in Java entwickelt. Dabei werden zwei Komponenten implementiert. Der Server und Client. Während der Server als Container nativ von der JRE ausgeführt wird, erstellt der GWT-Compiler ein Cross-Kompilat aus der Client-Komponente in JavaScript, welches nativ im Internetbrowser ausgeführt wird. Hierbei kommen RPCs zum Einsatz. Die Kommunikation zwischen Client und Server erfolgt asynchron. So muss für jeden RPC eine entsprechende Call-back-Methode implementiert werden. Diese Methode wird vom Server aufgerufen, sobald er die Antwort senden kann. RPC-Aufrufe haben somit immer den Rückgabewert **void**.

6.10 Docker

Docker ist eine offene Plattform, deren Aufgabe es ist, Anwendungen in virtuellen Containern auszuführen. Jede Anwendung erhält dabei eine eigene virtuelle Umgebung und kann somit von anderen Anwendungen isoliert werden. *Docker* ist für alle gängigen Systeme verfügbar und läuft nativ auf Linux-basierten Betriebssystemen. Unter Windows wird im Hintergrund eine virtuelle Linux-Maschine gestartet, in welcher die einzelnen Container ausgeführt werden.

6.11 LDAP

LDAP (*Lightweight Directory Access Protocol*) ist ein Protokoll, welches den Zugriff auf einen Verzeichnisdienst steuert. Ein solcher Dienst besteht meist aus einer hierarchischen Datenbank und kann entsprechend abgefragt und bearbeitet werden. Microsoft-**ActiveDirectory** ist ein Beispiel für einen solchen Verzeichnisdienst. Es werden Informationen über Benutzer und Computer einer Domäne gespeichert. Der Dienst kann dazu benutzt werden um Nutzer mit dem Benutzernamen und Passwort zu authentifizieren. Ein Nutzer ist einer oder mehreren Gruppen zugeordnet diese kann ebenfalls abgefragt werden, wodurch der Nutzer auch gleichzeitig autorisiert werden kann.

6.12 DSGVO

Die **Datenschutz-Grundverordnung** ist eine von der EU beschlossene Verordnung, die die Verarbeitung von personenbezogenen Daten regelt. [1]
Da wir in dieser Arbeit solche Daten verarbeiten, müssen die Richtlinien der *DSGVO* eingehalten werden.
Auch andere Richtlinien bezüglich der Veröffentlichung von Diensten, wie die **Impressumspflicht**, müssen beachtet werden.

7 Anhang

```sql
CREATE TABLE IF NOT EXISTS oauth_access_token (
  token_id VARCHAR(255),
  token BYTEA,
  authentication_id VARCHAR(255),
  user_name VARCHAR(255),
  client_id VARCHAR(255),
  authentication BYTEA,
  refresh_token VARCHAR(255)
);
```

Listing 7.1: SQL: OAUTH_ACCESS_TOKEN

```sql
CREATE TABLE IF NOT EXISTS oauth_client_details (
  client_id varchar(255) NOT NULL,
  resource_ids varchar(255) DEFAULT NULL,
  client_secret varchar(255) DEFAULT NULL,
  scope varchar(255) DEFAULT NULL,
  authorized_grant_types varchar(255) DEFAULT NULL,
  web_server_redirect_uri varchar(255) DEFAULT NULL,
  authorities varchar(255) DEFAULT NULL,
  access_token_validity integer DEFAULT NULL,
  refresh_token_validity integer DEFAULT NULL,
  additional_information varchar(255) DEFAULT NULL,
  autoapprove varchar(255) DEFAULT NULL
);
```

Listing 7.2: SQL: OAUTH_CLIENT_DETAILS

```sql
CREATE TABLE IF NOT EXISTS oauth_refresh_token(
  token_id VARCHAR(255),
  token BYTEA,
  authentication BYTEA
);
```

Listing 7.3: SQL: OAUTH_REFRESH_TOKEN

```sql
CREATE TABLE IF NOT EXISTS oauth_code (
  code VARCHAR(255), authentication BYTEA
);
```

Listing 7.4: SQL: OAUTH_CODE

```sql
CREATE TABLE IF NOT EXISTS oauth_approvals (
    userId VARCHAR(255),
    clientId VARCHAR(255),
    scope VARCHAR(255),
    status VARCHAR(10),
    expiresAt TIMESTAMP,
    lastModifiedAt TIMESTAMP
);
```

Listing 7.5: SQL: OAUTH_APPROVALS

```sql
CREATE TABLE IF NOT EXISTS oauth_client_token (
  token_id VARCHAR(255),
  token BYTEA,
  authentication_id VARCHAR(255),
  user_name VARCHAR(255),
  client_id VARCHAR(255)
);
```

Listing 7.6: SQL: OAUTH_CLIENT_TOKEN

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.
    thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8
        "/>
    <title>OAuth Server Login</title>
    <link rel="stylesheet" th:href="@{webjars/bootstrap/3.3.7-1/css/
        bootstrap.min.css}"/>


</head>
<body>
<div th:remove="tag">
    <header>
        <nav class="navbar navbar-inverse navbar-static-top">
            <div class="container">

                <div id="navbar" class="collapse navbar-collapse">
                    <ul class="nav navbar-nav">
                        <li class="active">
                            <a th:href="@{/}">Home</a>
                        </li>
                    </ul>
                </div>
```

```
23              </div>
24          </nav>
25      </header>
26      <div class="row">
27          <div class="col-xl-4 col-xl-offset-8 col-md-6 col-md-offset-3
                col-sm-8 col-sm-offset-2">
28              <h1>Login page</h1>
29              <form th:action="login.do" method="post">
30                  <div th:if="${param.error}">
31                      <div class="alert alert-danger">
32                          Invalid username or password.
33                      </div>
34                  </div>
35                  <div th:if="${param.logout}">
36                      <div class="alert alert-info">
37                          You have been logged out.
38                      </div>
39                  </div>
40                  <div class="form-group">
41                      <label for="username">Username</label>:
42                      <input type="text" id="username" class="form-
                            control" name="username" placeholder="Username"/
                            >
43                  </div>
44                  <div class="form-group">
45                      <label for="password">Password</label>:
46                      <input type="password" id="password" class="form-
                            control" name="password" placeholder="Password"/
                            >
47
48                  </div>
49                  <div class="form-group">
50                      <div class="row">
51                          <div class="col-sm-6 col-sm-offset-3">
52                              <input type="hidden" th:name="${_csrf.
                                    parameterName}" th:value="${_csrf.token}"
                                    />
53                              <input type="submit"
54                                     name="login-submit"
55                                     id="login-submit"
56                                     class="form-control btn btn-info"
57                                     value="Log In">
58
59
```

```
                        </div>

                        <div class="col-sm-6 col-sm-offset-3">
                            <br>
                            <a class="form-control btn btn-success" th:
                               href="@{/registration/register}">Register
                               </a>
                        </div>
                    </div>
                </div>
            </form>
        </div>
    </div>


</div>
</body>
</html>
```

Listing 7.7: HTML: login.html

```
package org.fzj.ibg3.oauth2.authentication.utils;

import org.springframework.security.oauth2.common.OAuth2AccessToken;
import org.springframework.security.oauth2.provider.ClientDetails;

import java.util.Collection;

public class TokenMapper {

    private ClientDetails clientDetails;
    private Collection<OAuth2AccessToken> oAuth2AccessTokens;


    public TokenMapper(ClientDetails clientDetails, Collection<
        OAuth2AccessToken> oAuth2AccessTokens) {
        this.oAuth2AccessTokens = oAuth2AccessTokens;
        this.clientDetails = clientDetails;
    }




    public ClientDetails getClientDetails() {
        return clientDetails;
```

```
    }

    public void setClientDetails(ClientDetails clientDetails) {
        this.clientDetails = clientDetails;
    }


    public Collection<OAuth2AccessToken> getoAuth2AccessTokens() {
        return oAuth2AccessTokens;
    }

    public void setoAuth2AccessTokens(Collection<OAuth2AccessToken>
        oAuth2AccessTokens) {
        this.oAuth2AccessTokens = oAuth2AccessTokens;
    }



}
```

Listing 7.8: JAVA: TokenMapper.java

```
<!DOCTYPE html>[...]
        <div class="col-md-12">
            <span sec:authorize="isAuthenticated()" style="display:
                inline-block;">
                    <a th:href="@{/logout}">Sign Out</a> [...]
                </span>
            </br>
            <span sec:authorize="isAuthenticated() and hasRole('
                REGISTERED_USER')" style="display: inline-block;">
                    <a th:href="@{/registration/edit}">Edit Profile</a>
                </span>
        </div>[...]
    <div class="row">
        <div class="col-md-6 col-md-offset-3">
            <h2>Approvals</h2>

            <p>
                If you revoke the approval for one scope of a client
                    all tokens for that client will be removed as well.
            </p>
            <table class="table table-bordered">
                <tr>
                    <th>Client</th>
```

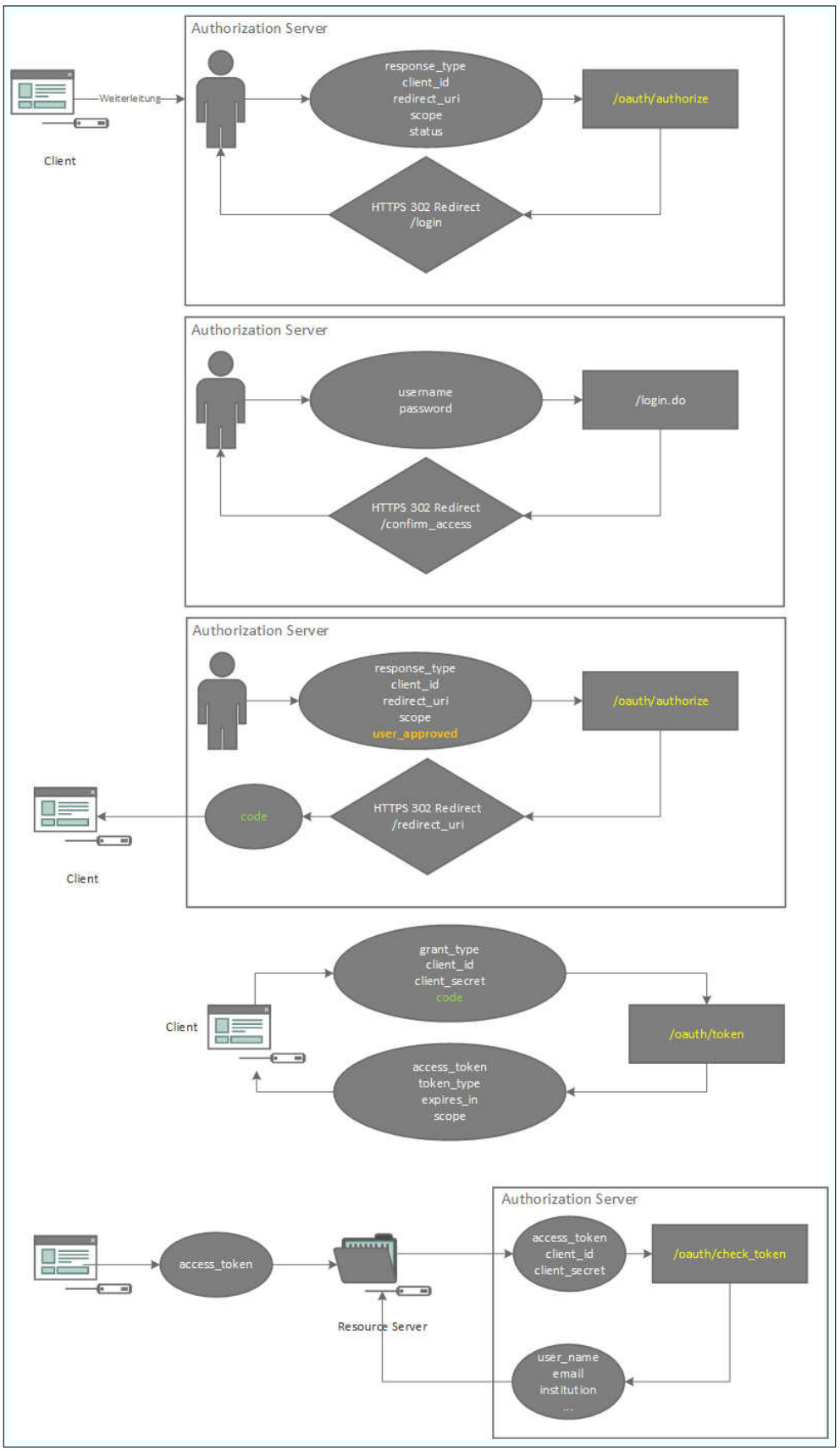


Abbildung 7.1: Ablauf des *authorization-code*-Verfahrens

```
21                  <th>Scope</th>
22                  <th>Expires at</th>
23                  <th>Actions</th>
24                  <th>Status</th>
25              </tr>
26              <!--/*@thymesVar id="approvals" type="java.util.List"*/
                    -->
27              <tr th:each="approval : ${approvals}"> [...]
28                  <!--/*@thymesVar id="approval" type="org.
                        springframework.security.oauth2.provider.
                        approval.Approval"*/-->[...]
29                  <td th:text="${approval.getClientId()}"></td>
30                  <td th:text="${approval.getScope()}">scope</td>
31                  <td th:text="${#dates.format(approval.expiresAt, '
                        dd-MM-yyyy HH:mm:ss')}">expires_at</td>
32                  <td th:text="${approval.getScope()}">scope</td>
33                  <td th:text="${approval.getStatus()}">status</td>
34                  <td>
35                      <form th:action="@{/approval/revoke}" method="
                            POST">
36                          <input type="hidden" th:name="${_csrf.
                                parameterName}" th:value="${_csrf.token}"
                                />
37                          <input type="hidden" name="userId" th:value=
                                "${approval.userId}"/>
38                          <input type="hidden" name="clientId" th:
                                value="${approval.getClientId()}"/>
39                          <input type="hidden" name="scope" th:value="
                                ${approval.getScope()}"/>
40                          <input type="hidden" name="expiresAt"
41                                 th:value="${#dates.format(approval.
                                     expiresAt, 'yyyy-MM-dd HH:mm:ss')}
                                     "/>
42                          <input type="hidden" name="status" th:value=
                                "${approval.getStatus()}"/>
43                          <input type="hidden" name="lastUpdatedAt"
44                                 th:value="${#dates.format(approval.
                                     lastUpdatedAt, 'yyyy-MM-dd HH:mm:
                                     ss')}"/>
45
46                          <button type="submit" class="btn btn-default
                                 btn-xs">
47                              <span class="glyphicon glyphicon-trash">
                                    </span>
```

```
48                          </button>
49                      </form>
50
51                  </td>
52              </tr>
53          </table>
54      </div>
55
56
57      <div class="col-md-10 col-md-offset-1">
58          <h2>Token</h2>
59
60
61          <table class="table table-bordered">
62              <tr>
63                  <th>Client</th>
64                  <th>Scope</th>
65                  <th>Expires at</th>
66                  <th>Type</th>
67                  <th>Expired</th>
68                  <th>Value</th>
69              </tr>
70              <!--/*@thymesVar id="tokenMappers" type="java.util.List
                    "*/-->
71              <div th:each="tokenMapper : ${tokenMappers}">
                    [...][...]
72
73
74                  <!--/*@thymesVar id="tokenMapper" type="org.fzj.
                        ibg3.oauth2.authentication.utils.TokenMapper"*/
                        -->
75                  <tr th:each="token : ${tokenMapper.
                        getoAuth2AccessTokens()}">
76
77                      <!--/*@thymesVar id="token" type="org.
                            springframework.security.oauth2.common"*/-->
78                      <td th:text="${tokenMapper.getClientDetails().
                            getClientId()}"></td>
79                      <td th:text="${token.getScope()}"></td>
80                      <td th:text="${#dates.format(token.
                            getExpiration(), 'dd-MM-yyyy HH:mm:ss')}"></
                            td>
81
82                      <td th:text="${token.getTokenType()}"></td>
```

```
                    <td th:text="${token.isExpired()}"></td>
                    <td th:text="${token.getValue()}"></td>

                </tr>

            </div>
        </table>
    </div>
    </div>[...]
<footer>
    <div class="row"> [...]
        <div class="col-2 col-xs-offset-10" th:text="'Version: '+${
            version}"></div>[...]
    </div>[...]
</footer>
</body>
</html>
```

Listing 7.9: HTML: index.html

```
    <div class="row" sec:authorize="hasRole('OAUTH_ADMIN')">

        <div class="col-md-10 col-md-offset-1">

            <h2>Clients</h2>
            <table class="table table-bordered">
                <tr>
                    <th>Client ID</th>
                    <th>Resource IDs</th>
                    <th>Scopes</th>
                    <th>Grant Types</th>
                    <th>Roles</th>
                    <th>Actions</th>
                </tr>

                <!--/*@thymesVar id="clientDetails" type="java.util.
                    List"*/-->
                <tr th:each="client : ${clientDetails}">
                    <!--/*@thymesVar id="client" type="org.
                        springframework.security.oauth2.provider.
                        ClientDetails"*/-->
                    <td th:text="${client.getClientId()}"></td>
                    <td th:text="${client.getResourceIds()}"></td>
                    <td th:text="${client.getScope()}"></td>
                    <td th:text="${client.getAuthorizedGrantTypes()}"><
```

```
                    /td>
23                  <td th:text="${client.getAuthorities()}"></td>
24                  <td>
25                      <a class="btn btn-default btn-xs" th:href="@{/
                            clients/form?client=__${client.clientId}__}"
                            >
26                          <span class="glyphicon glyphicon-edit"></
                                span>
27                      </a>
28                      <a class="btn btn-default btn-xs" th:href="@{/
                            clients/__${client.clientId}__/delete}">
29                          <span class="glyphicon glyphicon-trash"></
                                span>
30                      </a>
31                  </td>
32              </tr>
33              <tr>
34                  <td colspan="6">
35                      <a class="btn btn-default btn-xs pull-right" th:
                            href="@{/clients/form}">
36                          <span class="glyphicon glyphicon-plus"></
                                span>
37                      </a>
38                  </td>
39              </tr>
40          </table>
41      </div>
42  </div>[...]
```

Listing 7.10: HTML: index.html Client Verwaltung

```
1 [...]
2
3       <form id="register_form" autocomplete="off" action="#" th:
            action="@{/registration/register}" [...]
4           th:object="${user}" method="post" class="m-t" role="form"
5           data-toggle="validator">[...]
6
7
8           <div th:if="${errorMessage}" class="alert alert-danger"
                [...]
9               role="alert" th:text=${errorMessage}></div>[...]
10
11          <div th:if="${confirmationMessage}" class="alert alert-
                success"
```

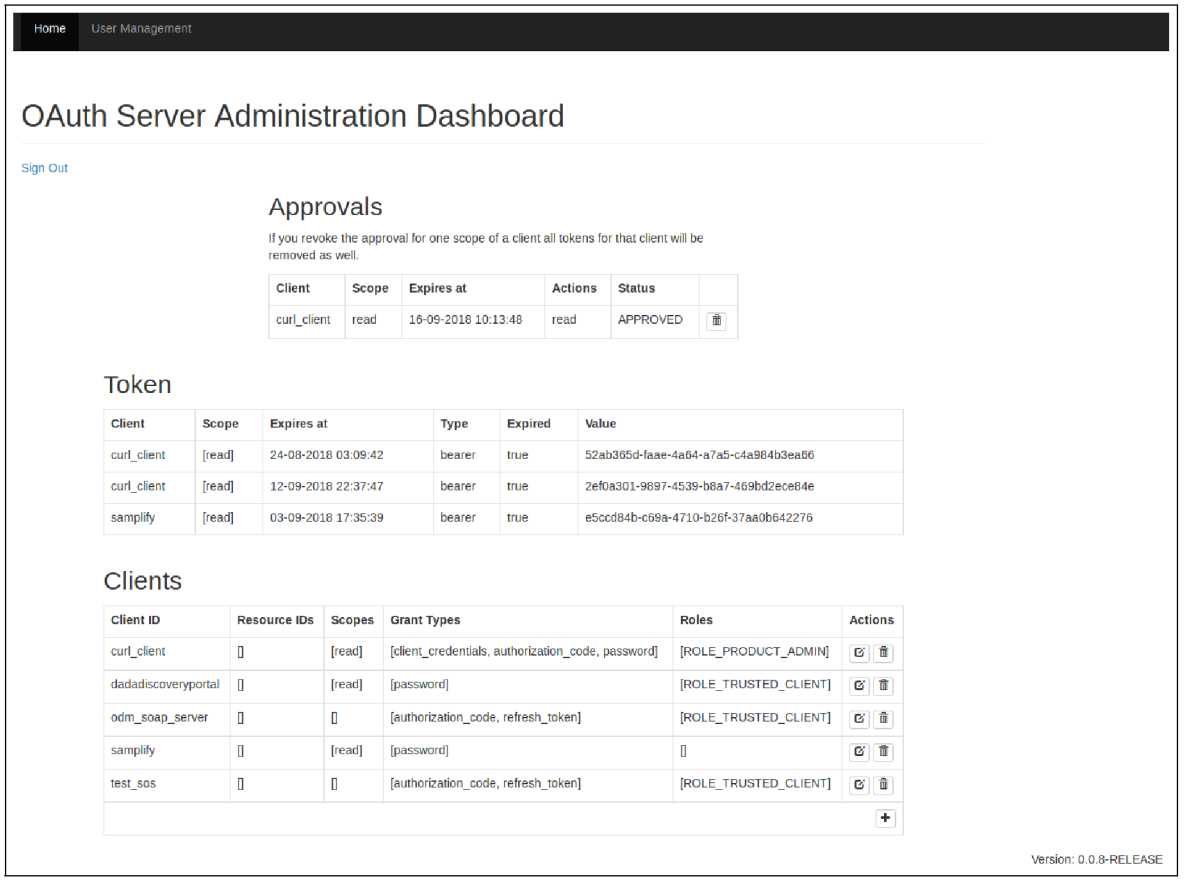


Abbildung 7.2: Ansicht der Index-Seite

```
                role="alert" th:text=${confirmationMessage}></div>

            <div th:if="${alreadyRegisteredMessage}"
                 class="alert alert-danger" role="alert"
                 th:text="${alreadyRegisteredMessage}"></div>


            <div th:if="${#fields.hasErrors('firstName')}" [...]
                 th:errors="*{firstName}"
                 class="validation-message alert alert-danger" role="
                    alert"></div>[...][...]
            <button
                    class="g-recaptcha btn btn-primary block full-width
                        m-b"
                    data-sitekey="6
                        LcDFF8UAAAAALroXFhdZcZACvcmT4_KjaRCb8FF"
                    data-callback="onSubmit">
                Register
            </button>

        </form>[...]
<script src="//cdnjs.cloudflare.com/ajax/libs/1000hz-bootstrap-
    validator/0.11.9/validator.min.js"></script>[...][...]

<script src='https://www.google.com/recaptcha/api.js'></script>
    [...][...]

<script>[...]
    function onSubmit(token) {[...]
        document.getElementById("register_form").submit();[...]
    }
</script>[...]
```

Listing 7.11: HTML: register.html

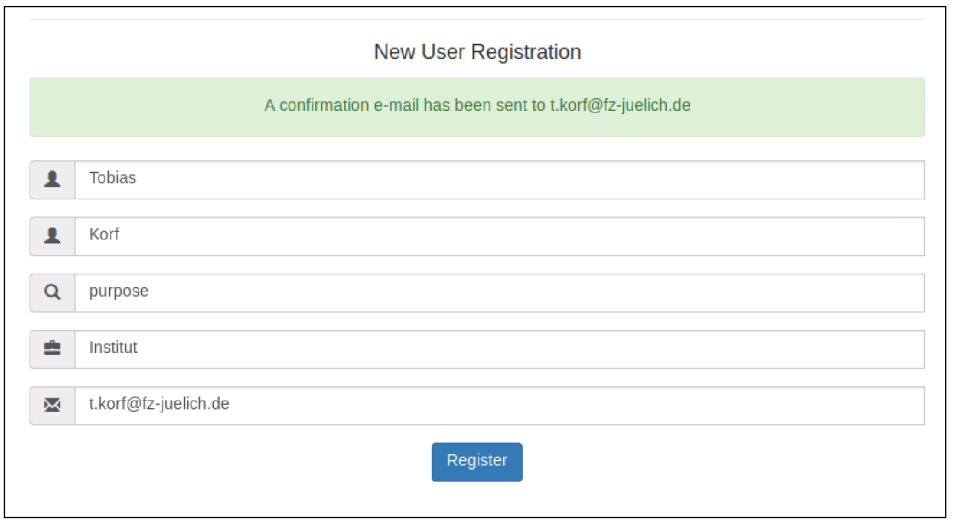
```
[...]

        <form th:if="!${invalidToken}" class="m-t" id="passwordForm"
            role="form" action="#"[...]
            <input type="hidden" name="token" th:value=${
                confirmationToken} >[...][...]
                <input name="password" type="password" id="password"
                       placeholder="Password" class="form-control"
                           required />[...]
                <input type="password" class="form-control" id="signup-
```

```
                    password-confirm" placeholder="Confirm Password"
                    name="ConfirmPassword" data-fv-notempty="true"
8                       data-fv-notempty-message="Please confirm password
                            "
9                       data-fv-identical="true"
10                      data-fv-identical-field="password"
11                      data-fv-identical-message="Both passwords must be
                            identical" />[...]
12 <script src="//cdnjs.cloudflare.com/ajax/libs/zxcvbn/4.4.2/zxcvbn.js"
       ></script>
13 <script th:src="@{/js/fv.min.js}"></script>
14 <script>[...]
15     $(document).ready(function() {[...]
16         $('#passwordForm').formValidation({[...]
17             framework: 'bootstrap',[...]
18             icon: {[...]
19                 valid: 'glyphicon glyphicon-ok',
20                 invalid: 'glyphicon glyphicon-remove',[...]
21                                 var result = zxcvbn(password),
22                                     score = result.score,
23                                     message = result.feedback.warning   '
                                        The password is weak';[...][...]
24
25                                 // Update the progress bar width and add
                                        alert class[...]
26                                 var $bar = $('#strengthBar');[...]
27                                 switch (score) {[...]
28                                     case 0:
29                                         $bar.attr('class', 'progress-bar
                                                progress-bar-danger')[...]
30                                          .css('width', '1%');
31                                         break;[...]
32                                 if (score < 3) {
33                                     return {
34                                         valid: false,
35                                         message: message
36                                     }
37                                 }[...]
```

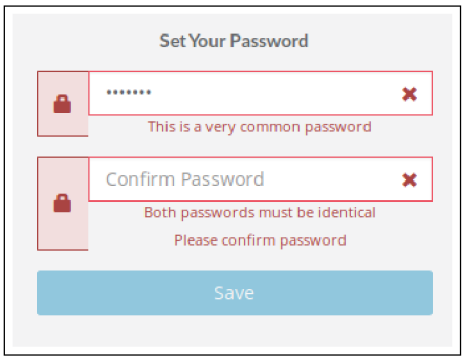
Listing 7.12: HTML: confirm.html

```
1     private SampleManagementAccount extractAccountInfoFromSoapHeader()
2             throws IbgException, NamingException {
3         MessageContext mctx = wsctx.getMessageContext();
4
```

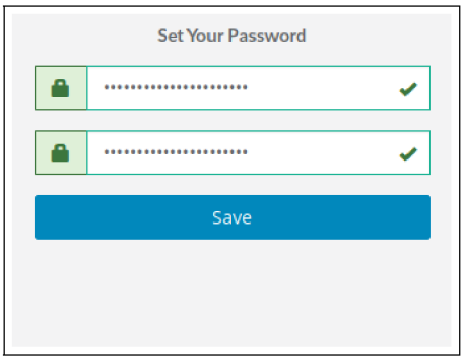


(a) Eingabe der Daten

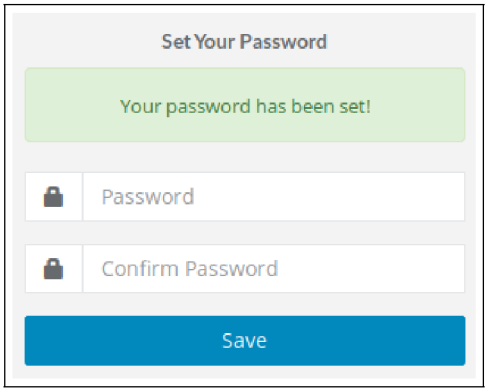
(b) Erhalt der Aktivierungs-Email



(c) Eingabe des Passworts (schwaches Passwort)



(d) Eingabe des Passworts (gutes Passwort)



(e) Bestätigung der Registrierung

Abbildung 7.3: Ablauf der Nutzerregistrierung

```
5       List<String> s = new ArrayList<>();
6
7       Object o = mctx.get(MessageContext.HTTP_REQUEST_HEADERS);
8
9
10      if (o instanceof Map) {
11          Object objectList = ((Map) o).get("Authorization");
12          if (objectList instanceof List) {
13              ((List<?>) objectList)
14                      .stream()
15                      .filter(in -> in instanceof String)
16                      .map(in -> (String) in)
17                      .forEach(s::add);
18          } else throw new IbgException("Cannot find Authentication-
                Header");
19      } else throw new IbgException("Cannot find Authentication-
            Header");
20      if (s.size() > 0) {
21          String authString = s.get(0);
22
23          if (authString.startsWith("Basic")) {
24
25              String encoded = authString.trim().substring(5).trim();
26              try {
27                  String decoded = new String(Base64.decodeBase64(
                        encoded),
28                          "UTF-8");
29                  String[] up = decoded.split(":");
30                  SampleManagementAccount account = (
                        SampleManagementAccount) Constants
31                          .getInstance().getAuthentication().getAccount
                                (up[0]);
32                  account.setPassword(up[1]);
33                  account.setSampleDataWritePermissions(account
34                          .isMemberOf(Constants.getInstance()
35                                  .getLdapSampleDataWriterGroups()));
36                  account.setSampleMetadataWritePermissions(account
37                          .isMemberOf(Constants.getInstance()
38                                  .getLdapSampleMetadataWriterGroups())
                                    );
39                  account.setManager(account.isMemberOf(Constants.
                        getInstance()
40                          .getLdapSampleManagerGroups()));
41                  return account;
```

```
42                } catch (UnsupportedEncodingException e) {
43                    throw new IbgException(
44                            "exception while decode base64
                                authentication info", e);
45                }
46            } else if (authString.startsWith("Bearer")) {
47                String encoded = authString.trim().substring(6).trim();
48                String jsonString = doGet(encoded);
49                JSONObject userObject = new JSONObject(jsonString);
50                if (jsonString == null) throw new IbgException("Could
                      not authenticate!");
51                if (userObject.has("active") && !userObject.getBoolean(
                      "active")) throw new IbgException("User is not
                      active!");
52
53                SampleManagementAccount account = (
                      SampleManagementAccount) Constants
54                        .getInstance().getAuthentication().getAccount(
                              jsonString);
55
56                account.setSampleDataWritePermissions(account
57                        .isMemberOf(Constants.getInstance()
58                                .getLdapSampleDataWriterGroups()));
59                account.setSampleMetadataWritePermissions(account
60                        .isMemberOf(Constants.getInstance()
61                                .getLdapSampleMetadataWriterGroups()));
62                account.setManager(account.isMemberOf(Constants.
                      getInstance()
63                        .getLdapSampleManagerGroups()));
64                return account;
65
66
67
68
69
70            }
71        }
72        throw new IbgException("missing user and password");
73    }
```

Listing 7.13: SoapOdmServiceFacade.java extractAccountInfoFromSoapHeader

```
1 protected String doGet(String token){
2
3   Element cachedObject = oauthCache.get(token);
```

```
4
5   if (cachedObject != null ){
6      log.info("cached token found!");
7      return (String) cachedObject.getObjectValue();
8   }
9
10  CredentialsProvider credsProvider = new BasicCredentialsProvider();
11  credsProvider.setCredentials(
12     new AuthScope(Constants.OAUTH_HOST, 443),
13     new UsernamePasswordCredentials(Constants.OAUTH_CLIENT,
          Constants.OAUTH_CLIENT_SECRET));
14
15  URIBuilder builder = null;
16  try {
17     builder = new URIBuilder(Constants.
          AUTHORIZATION_SERVER_CHECK_TOKEN);
18  } catch (URISyntaxException e) {
19     e.printStackTrace();
20     return null;
21  }
22
23  String content;
24  StatusLine statusLine;
25
26  builder.setParameter("token", token);
27
28  try (CloseableHttpClient httpclient = HttpClients.custom()
29     .setDefaultCredentialsProvider(credsProvider)
30     .build()) {
31     HttpGet httpget = new HttpGet(builder.build());
32
33
34     log.info("Executing request " + httpget.getRequestLine());
35     try (CloseableHttpResponse response = httpclient.execute(httpget
          )) {
36     log.info("----------------------------------------");
37     statusLine= response.getStatusLine();
38     log.info(statusLine);
39     content = EntityUtils.toString(response.getEntity());
40     log.info(content);
41     }
42  } catch (IOException | URISyntaxException e){
43     log.error(e);
44     return null;
```

```
45    }
46
47
48    if (statusLine == null || statusLine.getStatusCode() != 200) return
          null;
49
50    Element oauth = new Element(token,content);
51    oauthCache.put(oauth);
52
53    return content;
54
55 }
```

Listing 7.14: SoapOdmServiceFacade.java doGet-Methode

Literaturverzeichnis

[1] *Datenschutz-Grundverordnung DSGVO*. https://dsgvo-gesetz.de/, Abruf: 10.09.2018

[2] *OAuth 2 Developers Guide*. https://projects.spring.io/spring-security-oauth/docs/oauth2.html, Abruf: 10.09.2018

[3] *Password Hashing Competition*. https://password-hashing.net. Version: 2015, Abruf: 10.09.2018

[4] *OAuth*. https://de.wikipedia.org/wiki/OAuth. Version: 2018, Abruf: 10.09.2018

[5] *Spring (Framework)*. https://de.wikipedia.org/wiki/Spring_(Framework). Version: 2018, Abruf: 10.09.2018

[6] Akourti, Ahmed: Oauth 2 Centralized Authorization with Spring Boot 2.0.2 and Spring Security 5 and JDBC token store. In: *medium.com* (2018). https://medium.com/@akourtim.ahmed/oauth-2-centralized-authorization-with-spring-boot-2-0-2-and-spring-security-5-and-jdbc-token-store-8dbc063bd5d4, Abruf: 10.09.2018

[7] Boroumand, Amir: User Account Registration feature with Spring Boot. In: *www.codebyamir.com* (2017). https://www.codebyamir.com/blog/user-account-registration-with-spring-boot, Abruf: 10.09.2018

[8] Oliveira, Claudio Eduardo d.: *Spring 5.0 by example : grasp the fundamentals of Spring 5.0 to build modern, robust, and scalable Java applications*. Birmingham, UK : Packt Publishing, 2018. – ISBN 9781788628020

[9] Oliver Meyer, Arne B.: Bereitstellung und Visualisierung hydrologischer Zeitreihen mit Hilfe standardisierter Webdienste. (2008). http://www.arne-broering.de/Bereitstellung%20und%20Visualisierung%20hydrologischer%20Zeitreihen%20mit%20Hilfe%20standardisierter%20Webdienste.pdf, Abruf: 10.09.2018

[10] Ruzicka, Vojtech: Field Dependency Injection Considered Harmful. In: *www.vojtechruzicka.com* (2016). https://www.vojtechruzicka.com/field-dependency-injection-considered-harmful/, Abruf: 10.09.2018

[11] Sorg, Jürgen ; Kunkel, Ralf: Conception and Implementation of an OGC-Compliant Sensor Observation Service for a Standardized Access to Raster Data. In: *http://www.mdpi.com* (2015). http://www.mdpi.com/2220-9964/4/3/1076, Abruf: 10.09.2018

[12] Wheeler, Daniel L.: *zxcvbn: Low-Budget Password Strength Estimation.* https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler, Abruf: 10.09.2018