Fachbereich Mathematik und Informatik

Masterarbeit

mit dem Thema

Sicherung von Services in der Cloud auf der Basis der Kombination von OpenID und OAuth

Berlin, 03. September 2014

Betreuer: Dipl-Inf. Nikolay Tcholtchev
Erstgutachter: Prof. Dr. Ina Schieferdecker
Zweitgutachter: Prof. Dr. Claudia Müller-Birn

Philipp Lämmel
Matrikel-Nr. 4285328
philipp.laemmel@fu-berlin.de

Abstract

Die Sicherung von Diensten spielt eine zentrale Rolle bei der Entwicklung einer sicherheitskritischen Infrastruktur. In dieser Masterarbeit wird daher eine *Integrierte Komponente zur Sicherung von Cloud Services (ISCS)* spezifiziert und entwickelt, die den Zugriff auf Dienste in der Cloud gewährleistet. Das bedeutet, die ISCS ist für Zugriffsaspekte wie Authentifizierung, Autorisierung und Registrierung zuständig. Die entwickelte Komponente wird mit Hilfe von Unit Tests, Integrationstests, Fuzz Tests und Performance Tests eingehend geprüft. Weiterhin wird sie im Rahmen des Fraunhofer Projektes Gemeinschaftlich-e-Mobilität: Fahrzeuge, Daten und Infrastruktur (GeMo) eingesetzt, in welchem sie für die Sicherung von Cloud Services im Bereich Elektromobilität verwendet wird. Für die ISCS wird OAuth und OpenID verwendet, welche mit bestehenden Open Source Bibliotheken implementiert werden. OAuth steht für einen Standard, welcher es Diensten und Applikationen erlaubt, über eine vertrauenswürdige Infrastruktur auf sicherheitskritische Ressourcen in der Cloud zuzugreifen. OpenID ist ein populärer Standard aus der Webcommunity, welcher die domainübergreifende Authentifizierung von Portalnutzern ermöglicht. Die Kombination beider Standards erlaubt das Angebot einer vielfältigen Funktionalität, welche die wesentlichen Bereiche der Sicherung von Cloud Services abdeckt. Für das Testen und die Evaluation der Zuverlässigkeit der entwickelten Lösung wird insbesondere die von Fraunhofer FOKUS entwickelte Bibliothek Fuzzino verwendet. Es kann festgestellt werden, dass die Implementierung dem jeweiligen Standard entspricht und hinreichend sicher, stabil und performant ist.

Inhaltsverzeichnis

1 Einleitung und Motivation **1**

2 Projekthintergrund **3**

3 Aktueller Forschungsstand **7**

3.1 Kerberos 7

3.1.1 Komplexität 11

3.1.2 Sicherheitsaspekte 11

3.2 Mozilla Persona 12

3.2.1 Komplexität 16

3.2.2 Sicherheitsaspekte 16

3.3 OpenID 17

3.3.1 Komplexität 21

3.3.2 Sicherheitsaspekte 21

3.4 Security Assertion Markup Language (SAML) 22

3.4.1 Komplexität 25

3.4.2 Sicherheitsaspekte 25

3.5 Shibboleth 26

3.5.1 Komplexität 28

3.5.2 Sicherheitsaspekte 28

3.6 OAuth 29

3.6.1 Komplexität 32

3.6.2 Sicherheitsaspekte 32

4 Anforderungsanalyse **34**

5 Architektur **38**

5.1 Mobilitätsdaten-Cloud 38

5.2 Integrierte Komponente zur Sicherung von Cloud Services . . . 41

5.2.1 Identifizierte Komponenten 41

5.2.2 Dynamische Aspekte 43

6 Implementierung **47**
6.1 Verwendete Technologien 47
6.2 Datenbanktechnologien 48
6.2.1 Autorisierungsdatenbank 48
6.2.2 Benutzerdatenbank 50
6.3 Umsetzung der OpenID-Subkomponente 52
6.3.1 Consumer 52
6.3.2 Provider 53
6.3.3 UserRegistration 55
6.3.4 User 56
6.4 Umsetzung der OAuth-Subkomponente 56
6.4.1 AuthzEndpoint 57
6.4.2 TokenEndpoint 59
6.4.3 ClientRegistrationEndpoint 63
6.4.4 RedirectEndpoint 64

7 Evaluation **65**
7.1 Anforderungen 65
7.2 Unit Tests 66
7.3 Integrationstests 67
7.4 Fuzzing 68
7.4.1 Data Fuzzing 69
7.4.2 Behavioral Fuzzing 70
7.5 Performance Testing 74
7.5.1 Laufzeitmessungen 75
7.5.2 Speicherbedarf 77

8 Fazit **83**

9 Ausblick **87**

1 Einleitung und Motivation

Die Sicherung des Zugriffs auf Dienste in Kommunikationsnetzen ist relevant für deren generelle Sicherheit und Zuverlässigkeit. Dabei spielen Authentifizierung und Autorisierung von Nutzern eine wichtige Rolle. Unter dem Begriff Authentifizierung ist zu verstehen, dass sich eine Person mit einem Beweis eindeutig identifiziert. Traditionellerweise werden zur Authentifizierung Benutzername und Passwort verwendet. Autorisierung beschreibt darüber hinaus den Prozess, ob ein Nutzer auf eine bestimmte Ressource zugreifen darf. Dies verlangt jedoch, dass sich dieser Nutzer vorab erfolgreich authentifiziert hat. In den letzten Jahren wurden immer mehr Applikationen entwickelt, die es dem Nutzer ermöglichen, auch mobil, beispielsweise über das Internet, auf Dienste zuzugreifen. Dabei stellt sich die Frage, ob es sinnvoll ist, Zugangsdaten bei mehreren Programmen zu speichern. OAuth wurde entwickelt, um dieser Notwendigkeit zu entgehen.
Autorisierungsprotokolle wie OAuth stellen einen Prozess zur Verfügung, welcher es ermöglicht, dass keine Zugangsdaten (Nutzername und Passwort) in einer Applikation lokal gespeichert werden müssen. Dadurch wird die Geheimhaltung der Zugangsdaten und somit auch deren Sicherheit erhöht. Aus diesem Grund wird OAuth im Rahmen dieser Masterarbeit als das Protokoll ausgewählt, das im Fraunhofer-Projekt *Gemeinschaftlich-e-Mobilität: Fahrzeuge, Daten und Infrastruktur (GeMo)* genutzt werden soll, um die zur Verfügung stehenden Cloud Services zu sichern. Ziel des Projekts ist unter anderem, die Infrastruktur von Car-Sharing-Diensten so auszubauen, dass die angebotenen Services von jedem Nutzer sicher und mobil genutzt werden können. Die Sicherheit der Nutzung wird mit der Verwendung von OAuth gewährleistet. Eine gemeinsam genutzte Cloud-Infrastrutkur ist die Hauptvoraussetzung für die Realisierung von gemeinschaftlicher Mobilität im Rahmen des GeMo-Projektes durch die verschiedene Applikationen, die unter anderem auf Smartphones oder in On-Board-Units (OBUs) laufen, angeboten werden.
Wie bereits erwähnt, wird zur Autorisierung eine Authentifizierung des Nutzers benötigt. Im Rahmen des Projekts GeMo wurde das dezentrale Authenti-

fizierungssystem OpenID verwendet. Dieses hat den Vorteil, dass sich Nutzer mit einer URL bei unterstützenden Systemen identifizieren können und sich nur einmalig mit einem Benutzernamen und Passwort bei einem OpenID-Provider anmelden müssen.
In dieser Masterarbeit wird eine Komponente im Rahmen des Projekts GeMo entwickelt, welche OAuth und OpenID kombiniert, um einen sicheren Zugriff für Dienste beziehungsweise Applikationen und Nutzer auf die bei GeMo verwendete Cloud zu gewährleisten. Die *Integrierte Komponente zur Sicherung von Cloud Services* wird im Verlauf dieser Arbeit durch das Akronym *ISCS* benannt. Ein besonderer Wert wird auf die Zuverlässigkeit der Lösung gelegt, in dem die von Fraunhofer FOKUS entwickelte Fuzzino Bibliothek [41, 42, 43] für das Fuzz Testing verwendet wird.
Anschließend an die Einleitung wird im nachfolgenden Kapitel der Hintergrund des Projektes GeMo zum besseren Verständnis vorgestellt. Daran anknüpfend wird der aktuelle Forschungsstand zu alternativen Authentifizierungs- und Autorisierungssystemen dargestellt. Dabei werden vor allem die Funktionsweise, Komplexität und Sicherheitsaspekte der Mechanismen fokussiert und vergleichend analysiert. Im Hauptteil dieser Arbeit erfolgt die Dokumentation der Entwicklung der ISCS. Dazu wird im dritten Kapitel eine ausführliche Analyse bezüglich funktionaler und nicht-funktionaler Anforderungen der Komponente erstellt. Daran anknüpfend wird die Architektur des Moduls präsentiert, wobei die identifizierten Komponenten sowie die dynamischen Aspekte erläutert und anhand von Sequenzdiagrammen ergänzt werden. Zentral für den Hauptteil dieser Arbeit ist die Implementierung der Komponente, welche im vierten Kapitel behandelt wird. Zudem wird in diesem Kapitel auf die Subkomponenten, die OAuth und OpenID implementieren, eingegangen. Darauf aufbauend werden im darauffolgenden Kapitel die Testverfahren sowie die aufbereiteten Ergebnisse vorgestellt. Verfahren wie Unit Tests, Integrationstests, Fuzz Tests und Performance Tests werden diskutiert. Im Fazit werden alle wichtigen Erkenntnisse resümierend zusammengefasst. Ein Ausblick zu noch offenen Fragen und dem Entwicklungspotential der ISCS schließt die Arbeit ab.

2 Projekthintergrund

Das Projekt *Gemeinschaftlich-e-Mobilität: Fahrzeuge, Daten und Infrastruktur (GeMo)* ist ein Verbundprojekt, an welchem die Institute *Fraunhofer-Institut für Integrierte Schaltungen (IIS)*, *Fraunhofer-Institut für Verkehrs- und Infrastruktursysteme (IVI)*, *Fraunhofer-Institut für Eingebettete Systeme und Kommunikationstechnik (ESK)*, *Fraunhofer-Institut für Solare Energiesysteme (ISE)*, *Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS)* und *Fraunhofer-Institut für Arbeitswirtschaft und Organisation (IAO)* involviert sind. Die geplante Projektlaufzeit begann mit der Initiierung im Jahr 2011 und dauert voraussichtlich bis November 2014.
Die Studie *Neue Mobilität. Ein Ausblick zur urbanen Mobilität in vier Dekaden (2013)* [39] stellt die Rahmenbedingungen des Verbundprojekts GeMo vor und beschreibt, dass in dessen Kontext folgende drei Entwicklungstrends der Mobilität zusammengeführt werden sollen: Die *Vernetzung* von unterschiedlichen Verkehrsträgern, Nutzern und der Infrastruktur, wodurch eine komfortable Nutzung ermöglicht wird, die weitere Förderung von *Elektromobilität* und die *gemeinschaftliche Nutzung von vorhandenen Ressourcen*, wie beispielsweise der Infrastruktur oder von Rohstoffen, welche somit geschont werden [39].
Weiterhin wird erläutert, dass im Rahmen des Projekts erste technische Lösungen definiert und umgesetzt werden. Folgende Schwerpunkte werden dabei thematisiert:

- Ressource *Fahrzeuge*: Dabei seien insbesondere die OBUs, welche als fahrzeugintegrierte Telematikeinheit dienen, von zentraler Bedeutung. Mit Hilfe dieser können alle Kommunikationsströme zusammenlaufen und somit gemeinschaftlich nutzbar gemacht werden [6, 39].

- Ressource *Infrastruktur*: Hierbei spielt im Rahmen des Projekts GeMo ein induktives Ladesystem eine wichtige Rolle, da dadurch komfortables und schnelles Laden ermöglicht wird [3, 6, 34, 39, 44].

- Ressource *Daten und Dienste*: Wichtig hierbei ist die Mobilitätsdaten-Cloud (MDC), wodurch alle mobilitätsrelevanten Daten über definierte

Schnittstellen aggregiert und bereitgestellt werden. Eine zentrale Rolle werden auf die MDC aufbauende Mobilitätsdienste spielen [6, 39, 45].

Insbesondere die Ressource *Daten und Dienste* ist für diese Masterarbeit von Relevanz und wird daher im Folgenden und auf Grundlage der oben benannten Studie *Neue Mobilität* [39] ausführlicher erläutert.
Die MDC, welche in Abbildung 1 schematisch dargestellt ist, stellt eine offene Datenintegrationsplattform bereit, welche die Erhebung, Aggregation, Bereitstellung und Auswertung mobilitätsrelevanter Daten ermöglicht. Dabei könnten verschiedene Quellen wie Ladestationen, Fahrzeugauslastungen oder ÖPNV-Verbindungsdaten berücksichtigt werden, wodurch eine gemeinschaftliche Nutzung von Mobilitätsressouren realisiert wird. Die MDC bildet somit die Grundlage, um verschiedene Teilsysteme miteinander zu verknüpfen [4, 39, 45, 46].

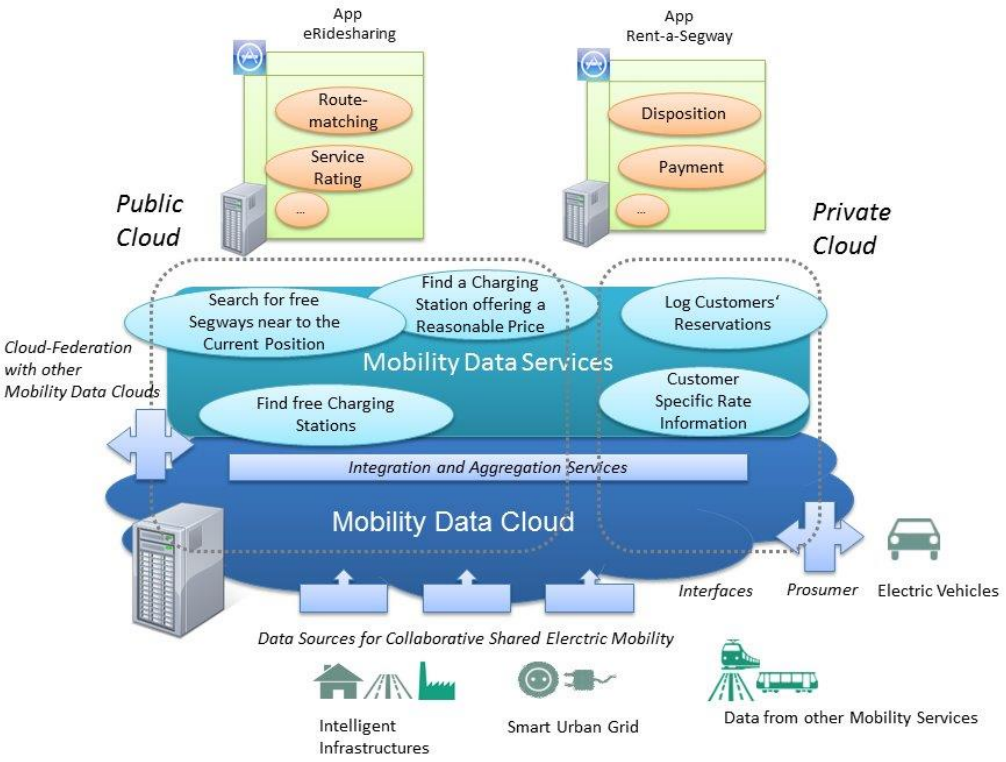


Abbildung 1: Schematische Darstellung der Mobilitätsdaten-Cloud

Mobilitätsrelevante Informationen können mit Hilfe der MDC über definier-

te Schnittstellen als Datenquellen übergreifend zugänglich und nutzbar gemacht werden. Dabei ist zu beachten, dass die Cloud in einen privaten und in einen öffentlichen Bereich unterteilt ist. Im öffentlichen Bereich werden nichtsensitive Mobilitätsdaten und Dienste bereitgestellt, welche für verschiedene Akteure offen zur Weiterverwendung zur Verfügung stehen. Im privaten Bereich werden hingegen Datenschutz-sensitive Daten für spezifische Anwendungen zuverlässig auf Basis von etablierten Sicherheits- und Zugriffsstandards persistent gehalten. Deshalb ist es möglich, dass Dienste sowohl Daten aus dem öffentlichen Bereich als auch aus dem privaten Bereich intelligent integrieren, kombinieren und aufbereiten können, um diese spezifisch nutzbar zu machen [4].

Die MDC funktioniert vor allem über offene Daten, welche in maschinenlesbaren Formaten frei zugänglich gemacht werden. Eine Öffnung der MDC im Kontext der Entwicklung von Diensten wird angestrebt. Die MDC agiert demzufolge als eine Art Plattform-as-a-Service (PaaS), wodurch externen Entwicklern ein effizienter Umgang mit aggregierten Daten und Diensten ermöglicht wird [4, 46].

Im Projekt GeMo wird im Rahmen von Smartphone-Applikationen für die Plattform Android Cloud-basierte Mobilitätsdienste zur Registrierung, Nutzerprofilpersonalisierung, Buchung von Fahrzeugen oder Ladestationen sowie Abrechnung von Fahrten demonstriert werden [5].

Nach Abschluss der Registrierung, wird dem Nutzer ermöglicht, in einem persönlichen Bereich Präferenzen zu hinterlegen und konkrete Reisedetails anzugeben, wodurch die Suche nach einem verfügbaren Fahrzeug eingeleitet wird. Die Fahrzeugauswahl erfolgt dabei mit Hilfe eines speziell an die Bedürfnisse der Elektromobilität adaptierten Dispositionssystems. Es gibt einen vierstufigen Selektionsprozess, wobei nutzerspezifische, zeitliche, räumliche und innovative ladezustandsabhängige Kriterien miteinbezogen werden. Die Reichweite der verfügbaren Fahrzeuge wird unter Berücksichtigung von fahrzeugspezifischen Informationen, wie zum Beispiel Ladestand und Energieverbrauch, für eine gegebene Route mit spezifischen Geländeprofil und aktuellen Wetterbedingungen eingeschätzt [5].

Die durch die MDC bereitgestellten Dienste, welche im privaten Bereich der

Cloud agieren, müssen ausreichend gesichert werden. Dazu wird ein Schutz benötigt, welcher die Autorisierung überprüft. Um einen besseren Überblick über Authentifizerungs- und Autorisierungssysteme zu erhalten, werden im anschließenden Kapitel eine Auswahl solcher Systeme präsentiert.

3 Aktueller Forschungsstand

In diesem Kapitel werden aktuelle Authentifizierungs- und Autoriseriungssysteme vorgestellt. Zu Beginn werden die reinen Authentifizerungssysteme erklärt, während im weiteren Verlauf Mechanismen, welche sowohl authentifizieren als auch autorisieren, erläutert werden. Abschließend wird ein reines Autorisierungssystem präsentiert. Jedes System wird hinsichtlich seiner Technologie und seines Protokollablaufs beschrieben. Daran anknüpfend werden die Komplexität sowie relevante Sicherheitsaspekte diskutiert.

3.1 Kerberos

Kerberos ist ein System, welches aktuell in Version 5 verfügbar ist und ursprünglich am *Massachusetts Institute of Technology (MIT)* entwickelt wurde [37]. Es soll eine sichere Authentifizierung in einem ungesicherten Netzwerk bieten. Kerberos basiert teilweise auf Needham und Schroeders Protokoll zur Authentifizierung [36] und auf Weiterentwicklungen, welche von den Autoren Denning und Sacco vorgeschlagen wurden [26]. Dabei wird im Rahmen von Kerberos eine zentrale Instanz verwendet, welche die Authentifizierung und das Erstellen von Tickets übernimmt.
Für Kerberos Authentifizierungssystem werden drei Parteien verwendet: der Client, der Server und ein dedizierter Kerberos-Server. Mit Hilfe des Clients wird es dem Nutzer ermöglicht, auf einen zugriffsgeschützten Service zuzugreifen, welcher sich auf einem Server befindet. Der Kerberos-Server besteht aus dem Authentication Server, welcher die Authentifizierung übernimmt sowie dem Ticket Granting Service (TGS), welcher demgegenüber Tickets ausstellt und verifiziert. Kerberos realisiert eine gegenseitige Authentifizierung des Clients als auch des Servers. Mit der Authentifizierung des Servers wird die Intention verfolgt, Man-in-the-middle-Angriffe zu verhindern. Darüber hinaus authentifiziert sich der Kerberos-Server gegenüber dem Client sowie gegenüber dem Server, um somit deren Identität zu verifizieren.
Kerberos verwendet Tickets zur Authentifizierung. Zur erfolgreichen Nutzung des Kerberos-Diensts ist die vorige Anmeldung eines Clients beim Kerberos-Server notwendig. Der Client fordert dazu ein Ticket an, wofür sich der Nutzer

vorab und in den meisten Fällen mit einem Passwort anmelden muss. Das erhaltene Ticket führt zur Erzeugung weiterer Tickets, wobei die wiederholte Eingabe eines Passworts jedoch nicht erforderlich ist. Mit Hilfe eines solchen, im späteren Verlauf generierten Tickets, überprüft ein Dienst, ob dem Client der Zugriff gestattet werden kann.
In Abbildung 2 ist der Protokollablauf erkennbar. Da bei Kerberos alle Nachrichten verschlüsselt und entschlüsselt werden, gilt für die Abbildung 2 die Notation: $\{msg\}_K$. Diese bedeutet, dass die Nachricht „msg“ mit dem Schlüssel K verschlüsselt wurde und somit einzig mit eben diesem Schlüssel K dechiffriert werden kann. Bei Kerberos kann sowohl symmetrische als auch public-key Kryptographie verwendet werden. Im Folgenden wird davon ausgegangen, dass symmetrische Kryptographie angewendet wird. Im ersten Schritt des Protokolls startet der Client eine Authentifizierungsanfrage an den Authentication Server (Nachricht „authentication request“). Dieser überprüft die Bekanntheit des Clients, woraufhin der Authentication Server bei einer positiven Rückmeldung ein *Ticket Granting Ticket (TGT)* erstellt , welches mit dem geheimen Schlüssel des TGS verschlüsselt wird (Nachricht „$\{\text{TGT}\}_{TGS-Key}$“). Der Authentication Server ist im Besitz dieses Schlüssels, weil ein Vertrauensverhältnis zwischen den beiden Instanzen besteht. Darüber hinaus wird ein neuer Client/TGS Session Key erzeugt, welcher mit dem Schlüssel chiffriert wird, der auf dem Nutzerpasswort basiert (Nachricht „$\{\text{Client/TGS Session key}\}_{PW-Key}$“). Das TGT enthält unter anderem die Client ID, die Gültigkeitsdauer sowie den Client/TGS Session Key. Nach Erhalt der beiden Nachrichten versucht der Client die verschlüsselte Nachricht mit dem Client/TGS Session Key zu entschlüsseln (Nachricht „decrypt $\{\text{Client/TGS Session key}\}_{PW-Key}$“). Die Authentifizierung des Nutzers ist erfolgreich, wenn der Client diese Nachricht erfolgreich dechiffrieren kann. Die Inkorrektheit des Passwort hätte zur Folge, dass die Generierung eines anderen Schlüssels durch den Client erfolgen würde und somit keine erfolgreiche Entschlüsselung dieser Nachricht möglich wäre. Nach diesem Schritt ist der Client im Besitz des Client/TGS Session Key [37].
Im darauffolgenden Schritt sendet der Client zwei Nachrichten an den TGS - eine Nachricht mit dem TGT und der ID des anzufragenden Service sowie

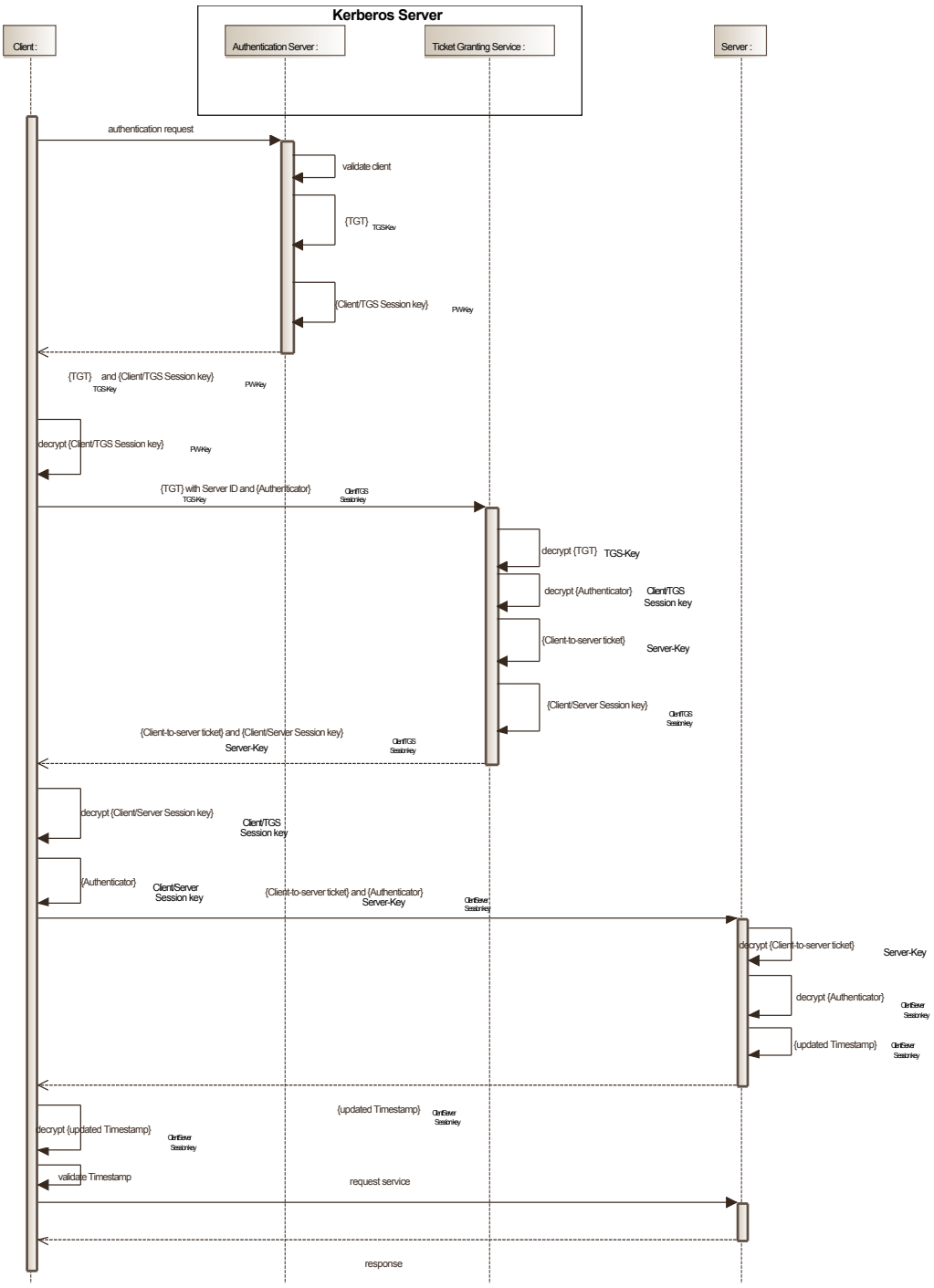


Abbildung 2: Der Protokollfluss von Kerberos

eine weitere mit einem *Authenticator*, welcher die Client ID und einen Zeitstempel beinhaltet. Der Authenticator ist mit Hilfe des Client/TGS Session Key verschlüsselt. Der TGS kann das TGT aus der Nachricht extrahieren und entschlüsseln, da es mit seinem privaten Schlüssel chiffriert wurde (Nachricht „decrypt $\{\text{TGT}\}_{TGS-Key}$“). Dadurch erhält der TGS den Client/TGS Session Key, mit welchem wiederum der Authenticator dechiffriert werden kann. Daraufhin wird eine neue Nachricht mit einem *Client-to-Server Ticket* vom TGS erstellt, welches unter Anderem den Client/Server Session Key, die Client ID und die Gültigkeitsdauer des Tickets enthält. Dieses Ticket wird mit dem geheimen Schlüssel des Dienstes chiffriert (Nachricht „$\{\text{Client-to-Server Ticket}\}_{Server-Key}$“). Eine weitere Nachricht beinhaltet den generierten Client/Server Session Key, welcher mit dem Client/TGS Session Key verschlüsselt ist. Diese beiden Nachrichten werden an den Client zurückgesendet. Abschließend wird es dem Client ermöglicht, aus der erwähnten Nachricht den Client/Server Session Key zu extrahieren und mit diesem einen neuen Authenticator zu generieren (Nachrichten „decrypt $\{\text{Client/Server Session Key}\}_{\text{Client/TGS Session Key}}$“ und „$\{\text{Authenticator}\}_{\text{Client/Server Session Key}}$“). Dieser beinhaltet wiederholt die Client ID und einen Zeitstempel. Sowohl der Authenticator als auch die Nachricht, die das Client-to-Server Ticket beinhaltet, werden an den Server versendet (Nachricht „$\{\text{Client-to-Server ticket}\}_{Server-Key}$ and $\{\text{Authenticator}\}_{\text{Client/Server Session key}}$“). Der Server entschlüsselt das Client-to-Server Ticket mit seinem geheimen Schlüssel und erhält dadurch den Client/Server Session Key. Mit diesem kann der Authenticator wiederum dechiffriert werden und der Service antwortet dem Client mit einer Nachricht, in welcher der Zeitstempel aus der zuvor entschlüsselten Nachricht mit der Zahl eins addiert wurde. Diese Nachricht wird mit dem Client/Server Session Key chiffriert und an den Client gesendet (Nachricht „$\{\text{updated timestamp}\}_{\text{Client/Server Session Key}}$“). Daran anschließend entschlüsselt der Client die Nachricht und überprüft, ob der Zeitstempel ordnungsgemäß aktualisiert wurde. Wenn die Aktualisierung korrekt vorliegt, ist der Client erfolgreich autorisiert und kann daraufhin den Service anfragen.

3.1.1 Komplexität

Die Komplexität des Protokolls ist als hoch zu bezeichnen, da zahlreiche Nachrichten versendet werden, welche jedes Mal erneut ver- und entschlüsselt werden. In Abhängigkeit vom verwendetem Verschlüsselungsverfahren ist dieses Vorgehen mehr oder weniger komplex. Des Weiteren wird der Client relativ hoch belastet, obwohl seine Aufgabe lediglich aus der Authentifizierung des Nutzers besteht. Darüber hinaus werden HTTP-Anfragen mit den erforderlichen Parametern gestellt, welche entsprechend verifiziert und bearbeitet werden.

3.1.2 Sicherheitsaspekte

Ein wichtiger Sicherheitsaspekt ist die Notwendigkeit der Existenz des Kerberos Servers. Wenn dieser nicht vorhanden ist, kann keine Authentifizierung stattfinden und folglich bricht das darauf beruhende Netzwerk zusammen. Daraus ergibt sich das Problem des Single-point-of-failure. Dieses kann jedoch reduziert werden, indem mehrere Kerberos Server eingesetzt werden, welche den Bedarf an Ressourcen jedoch stark erhöhen.
Des Weiteren hat Kerberos durch die Verwendung von Zeitstempeln strikte Anforderungen an die Synchronisierung der Uhren von Rechnern. Da nicht alle Uhren perfekt laufen, ist die regelmäßige Synchronisierung der Uhren oder die Verwendung eines Protokoll wie *Network Time Protocol (NTP)* notwendig. Anderenfalls können sich temporäre Unterschiede entwickeln, sodass das Protokoll fehlschlagen würde.
Im Falle der Verwendung von symmetrischer Kryptographie besteht die Gefahr der Kompromittierung des Kerberos Servers. Ist dieser Server erfolgreich kompromittiert, kann der Angreifer jeden Nutzer imitieren, da er auf alle geheimen Schlüssel zugreifen kann und folglich auch für jeden Server als ein beliebiger Nutzer ein valides Ticket ausstellen kann.
Abschließend ist anzumerken, dass Kerberos ein Vertrauensverhältnis zwischen Nutzer, Client und Services auf dem Server zu dem Kerberos Server verlangt. Dieses Vertrauensverhältnis basiert unter Anderem auf dem Austausch von geheimen Schlüsseln zwischen Authentication Server und Ticket

Granting Service. Kerberos kann folglich nicht in Szenarios verwendet werden, in welchen sich Nutzer zu Services von unbekannten Clients verbinden wollen, was einen typischen Fall in Cloud-Umgebungen darstellt. Durch das erforderliche Vertrauen zwischen Client und Nutzer wird es schwieriger, Kerberos in verschiedenen Umgebungen (zum Beispiel Test- und Produktivumgebung) zu verwenden, da entweder die strikte Trennung dieser Umgebungen aufgehoben werden oder zusätzliche Clients für jede Umgebung verwendet werden müssten.

3.2 Mozilla Persona

Mozilla Persona ist ursprünglich unter dem Namen BrowserID bekannt und wurde von der Mozilla Foundation entwickelt [11]. Es handelt sich um ein dezentrales Authentifizerungssystem, welche das Anmelden eines Nutzers mit seiner eigenen E-Mail-Adresse auf mehreren Webseiten erlaubt. Die Sicherheit des Systems basiert auf einem asymmetrischen Kryptosystem und digitalen Signaturen. Dabei wird beim Login eine sogenannte *Identity Assertion* erzeugt, welche die E-Mail-Adresse des Nutzers enthält. Diese Assertion wird mit dem privaten Schlüssel des Nutzers signiert und an den Server gesendet, welcher wiederum die Assertion von einem *Identity Provider* verifizieren lässt [20].

In Abbildung 3 ist der Protokollablauf erkennbar. Es ist ersichtlich, dass drei Parteien existieren: der Nutzer, der die Authentifizierung anstrebt, die Relying Party und der Identity Provider. Die Relying Party bezeichnet die Webseite, die den Nutzer mittels Persona authentifizieren will, während der Identity Provider die Authentifizierung des Nutzers übernimmt. Persona verwendet E-Mail-Adressen als Identitäten, weshalb es sich anbieten würde, dass die Rolle des Identity Providers von einem E-Mail-Anbieter übernommen wird [20]. Falls für den entsprechenden Anbieter kein Identity Provider existiert, stellt die Mozilla Foundation auf ihren Servern einen Ausweichprovider zur Verfügung. Als Voraussetzung gilt, dass der Nutzer über eine aktive Identity, sprich E-Mail-Adresse, verfügt, welche er für den Login verwenden kann.

Der erste Schritt im Protokoll ist das *User Certificate Provisioning*. Dabei

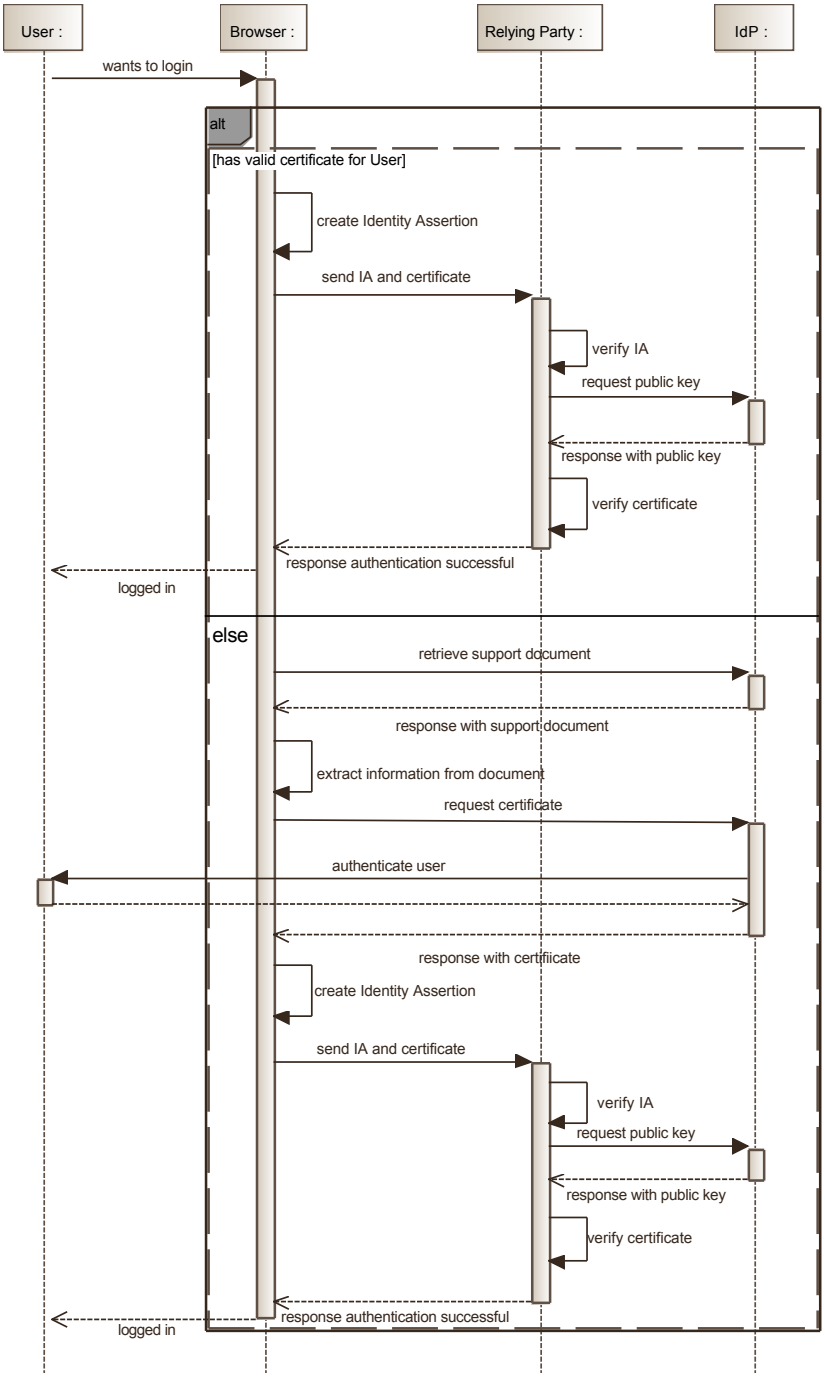


Abbildung 3: Der Protokollfluss von Persona

muss der Benutzer beweisen können, dass er der Eigentümer dieser E-Mail-

Adresse ist. Als Grundlage wird dafür ein kryptographisch signiertes Zertifikat vom Identity Provider erstellt, welches die Eigentümerschaft bestätigt. Dabei verfolgt Persona den Ansatz der *public-key Kryptographie*. Unter einem Zertifikat wird bei diesem Ansatz eine Menge an relevanten Informationen, die mit einem entsprechenden Schlüssel signiert sind, verstanden. Das Zertifikat wird mit Hilfe des privaten Schlüssels des Identity Providers erstellt und beinhaltet die folgenden Informationen:

- die E-Mail-Adresse des Nutzers,
- den öffentlichen Schlüssel des Nutzers für diese Adresse,
- das Erstellungsdatum des Zertifikats,
- das Ablaufdatum des Zertifikats und
- den Domainnamen des Identity Providers [20].

Für jede E-Mail-Adresse des Nutzers erzeugt der Browser sowohl einen privaten als auch öffentlichen Schlüssel, welche nicht browserübergreifend ausgetauscht werden. Folglich muss der Nutzer ein neues Zertifikat erhalten, wenn das veraltete abgelaufen ist. Dies geschieht nach maximal 24 Stunden oder beim Verwenden eines anderen Browsers beziehungsweise Rechners.
Nachdem der Nutzer eine Identität ausgewählt hat, überprüft der Browser, ob ein aktuelles Zertifikat vorhanden ist, wobei eine Assertion Generierung gestartet wird, wenn dies zutrifft. Liegt kein aktuelles Zertifikat vor, versucht der Browser ein Zertifikat vom Provider zu erhalten. Dabei wird ein Supportdokument per SSL beim Identity Provider angefragt (Nachricht „retrieve support document“ in Abbildung 3). Mit Hilfe der dort enthaltenden Informationen, kann der Browser eine Anfrage bezüglich eines Zertifikats an den Provider senden, wobei die E-Mail-Adresse des Nutzers mitgesendet wird. Unter Umständen, beispielsweise wenn keine aktive Sitzung vorhanden ist, wird es notwendig, dass der Nutzer sich beim Identity Provider authentifizieren muss, bevor das *Provisioning* fortfahren kann. Nach der Eingabe des Nutzers erstellt der Provider das gewünschte Zertifikat und sendet es an den Browser zurück [20].

Im zweiten Schritt wird eine Assertion generiert [20]. Das Zertifikat nimmt in diesem Rahmen die Funktion einer nachvollziehbaren Verbindung zwischen E-Mail-Adresse und öffentlichem Schlüssel ein. Darüber hinaus muss der Nutzer jedoch die Verbindung zwischen Zertifikat und privatem Schlüssel beweisen. Dafür erstellt und signiert der Browser ein Dokument, welches als *Identity Assertion* bezeichnet wird (Nachricht „create Identity Assertion“ in Abbildung 3). Dieses beinhaltet die Domain inklusive Schema (im Sinne eines URL-Schemas) und Port (auf dem der Server lauscht) der Relying Party sowie ein Ablaufdatum für die Assertion [20]. Im weiteren Verlauf sendet der Browser sowohl das Dokument als auch das Zertifikat des Nutzers an die Relying Party.
Im letzten Schritt erfolgt die Verifizierung der Assertion. Die Kombination von Zertifikat und *Identity Assertion* ist zur Bestätigung der Identität eines Nutzers ausreichend. Zu Beginn überprüft die Relying Party die Domain und das Ablaufdatum der Assertion. Werden dabei Unstimmigkeiten festgestellt, wird diese Assertion abgewiesen. Dieses Vorgehen verhindert die Wiederverwendung von Assertions. Im nächsten Schritt wird die Signatur der Assertion mit dem öffentlichen Schlüssel des Nutzerzertifikats verifiziert. Passen Schlüssel und Signatur zusammen, ist sichergestellt, dass der aktuelle Nutzer im Besitz des dazugehörigen privaten Schlüssels ist [20]. Abschließend beschafft sich die Relying Party den öffentlichen Schlüssel des Identity Providers und vergleicht diesen mit der Signatur im Zertifikat (Nachricht „verify certificate“ in Abbildung 3). Im Fall der Übereinstimmung ist sichergestellt, dass das Zertifikat von diesem Identity Provider ausgestellt wurde [20].
Zusammenfassend ist festzuhalten, dass zum Abschluss einer erfolgreichen Authentifizierung die Überprüfung des Loginversuchs hinsichtlich der Aktualität, die Sicherstellung der Übereinstimmung des Nutzers mit dem im Zertifikat enthaltenen Nutzer sowie die Überprüfung der Gültigkeit des Zertifikats benötigt werden.

3.2.1 Komplexität

Die Verwendung von public-key Kryptographie und Zertifikaten trägt enorm zur Erhöhung der Komplexität bei. Durch das Session Management wird diese noch gesteigert. Des Weiteren werden HTTP Anfragen gestellt, welche die entsprechenden Parameter enthalten. Diese Parameter werden vom Empfänger extrahiert, validiert und entsprechend bearbeitet.

3.2.2 Sicherheitsaspekte

Speicherung der Identity Assertion
Bei der Nutzung von Persona ist zu beachten, dass die Identity Assertion auch auf dem eigenen Server für Verifikationszwecke abgelegt werden sollte. Anderenfalls wäre es einem bösartigen Benutzer möglich, einen anderen Nutzer der Seite zu imitieren, in dem er eigenen Code in den Browser injiziert. Dies könnte beispielsweise über JavaScript oder mit Hilfe von verschiedenen Plug-Ins, die das Browserverhalten beeinflussen, erfolgen. Da nicht davon auszugehen ist, dass über die vollständige Kontrolle eines Browser verfügt werden kann, ist es folglich notwendig, die Identity Assertion ebenfalls zu speichern und wiederholt zu verifizieren [12].

Überprüfung der SSL Zertifikate
Die Verifizierung des Zertifikats, welches der Server bei der Verwendung von SSL sendet, sollte gegenüber eines vertrauenswürden Rootzertifikats ebenfalls überprüft werden.[12].

Schutz gegen CSRF
Die Verwendung eines geheimen Identifiers stellt eine weit verbreitete Möglichkeit dar, um sich vor CSRF zu schützen. Der Identifier wird dabei zwischen Browser und Server geteilt, was notwendig ist, um eine Loginanfrage zu starten [12].
Abschließend ist anzumerken, dass Persona seit März 2014 in das Eigentum der Community übergeht [23]. Mozilla behebt weiterhin kritische Bugs aber es werden keine neuen Features implementiert [23]. Im After Action

Review werden unter anderem Gründe für die Einstellung der Weiterentwicklung genannt. [19]. Es wird erörtert, dass Persona eine einfache Lösung für die dezentrale Authentifizierung darstellt. Aus Sicht der Mozilla Foundation war der kalte Start bei den Nutzern, Anbietern und den Webseiten das größte Problem. Große Webseiten verfügen über genug Ressourcen, um eigene Lösungen umzusetzen, was zur Nichtbeachtung von Persona führte [19]. Ein weiterer Grund war die Konkurrenz der Marke Persona mit der jeweiligen Marke einer anderen Webseite, die Persona verwenden wollte. Dies trug ebenfalls nicht zur Verbreitung von Persona bei. Abschließend war die Komplexität einiger Features (wie zum Beispiel das Session Management) zu hoch, was den Schwierigkeitsgrad des Verständnisses erhöhte und folglich Nutzer an der Verwendung von Persona hinderte.

3.3 OpenID

OpenID ist ebenfalls ein dezentrales Authentifizierungssystem für webbasierte Dienste. Es funktioniert, indem sich ein Nutzer (der Principal) bei einem OpenID-Provider registriert und daraufhin eine URL als OpenID erhält. Mit dieser ist es möglich, sich auf unterstützenden Webseiten, den Relying Parties, anzumelden, ohne dort erneut den Nutzernamen und das Passwort einzugeben. Es gibt gleichermaßen drei Parteien: den Principal, die Relying Party sowie den OpenID-Provider [16]. Mit Hilfe von Weiterleitungen wird der Identifier, welcher vorab der Relying Party übergeben wurde, an den OpenID-Provider weitergeleitet, bei welchem sich der Nutzer authentifiziert. Daraufhin antwortet der Provider mit einer Assertion, ob die Authentifizerung erfolgreich war oder fehlgeschlagen ist [16].
Ähnlich wie bei Persona kann ein eigener OpenId-Provider erzeugt und verwendet werden.

In Abbildung 4 ist der Protokollfluss von OpenID erkennbar. Es wird dargestellt, wie sich ein Nutzer auf einer Relying Party anmelden möchte, welche OpenID unterstützt. Im ersten Schritt präsentiert der Nutzer seinen Identifier, der, wie bereits erwähnt, eine URL ist, welche die Relying Party zu

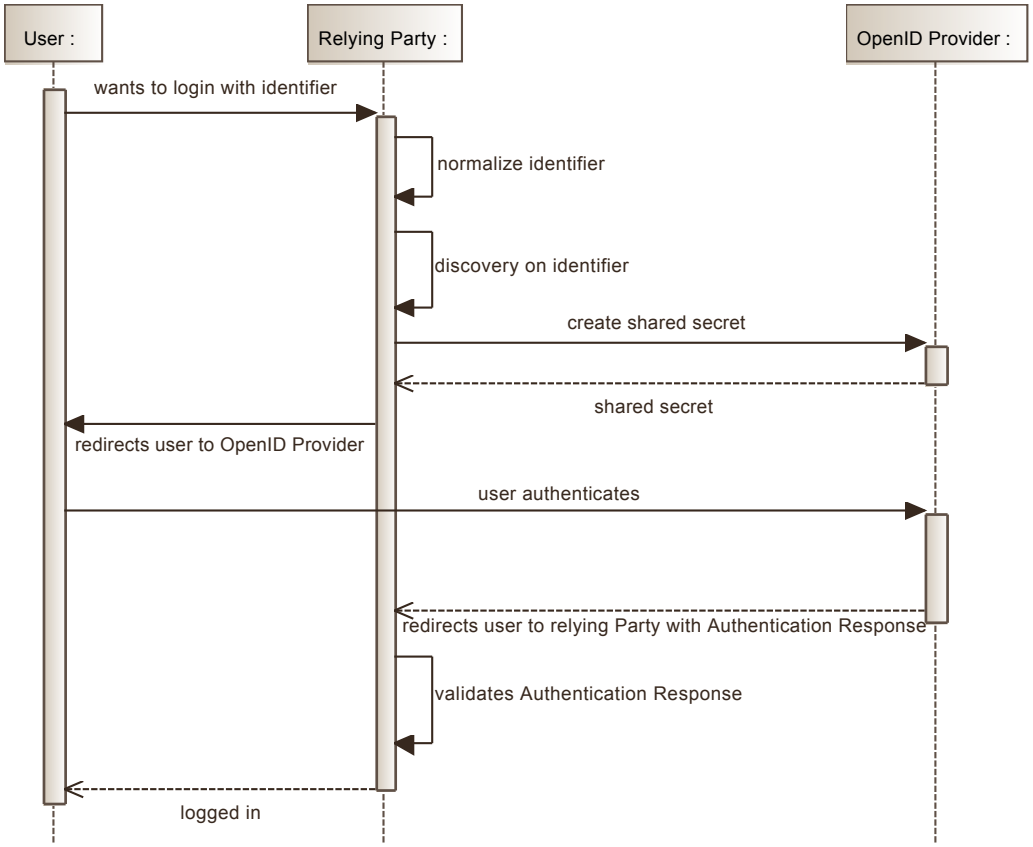


Abbildung 4: Der Protokollfluss von OpenID

Beginn normalisiert (Nachricht „normalize identifier“ in Abbildung 4) [16]. Es wird überprüft, ob die Eingabe mit „xri://“ beginnt. Wenn dieser Fall eintritt, muss die Entfernung des Präfix erfolgen. Daraufhin werden die darauffolgenden Zeichen überprüft. Handelt es sich dabei um ein XRI Global Context Symbol, dann sollte die Eingabe des Nutzers als *extensible resource identifier (XRI)* behandelt werden. Bei XRI handelt es sich um ein Schema und Auflösungsprotokoll, welches unter Anderem kompatibel zu Uniform Resource Identifiers (URIs) ist. Das Protokoll verfolgt das Ziel, eine standardisierte Syntax für abstrakte und strukturierte Identifier zu schaffen, damit diese leicht über verschiedene Domänen hinweg ausgetauscht werden können. Anderenfalls sollte mit der Eingabe als HTTP URL umgegangen werden. Ist kein „http://“ oder „https://“ in der URL vorhanden, muss ein

„http://“ vor die Eingabe gesetzt werden [16]. Eine weitere Überprüfung erfolgt bezüglich des Auftretens eines Fragments (Teil der URL nach dem Symbol „#“) in der URL. Wenn die Prüfung positiv ist, muss das Fragment sowie das Fragmentzeichen „#“ entfernt werden. Weitere Normalisierung findet durch das Folgen von HTTP/HTTPS-Redirects und der Erstellung einer Signatur für die normalisierte URL statt [16].
Nach der Normalisierung des Identifiers, führt die Relying Party eine Discovery durch (Nachricht „discovery on identifier“ in Abbildung 4). Dabei werden drei Fälle unterschieden:

- Wenn es sich bei dem Identifier um ein XRI handelt, erhält man ein *eXtensible Resource Descriptor Sequence (XRDS)* Dokument als Antwort, in welcher alle notwendigen Informationen, insbesondere auch die URL des OpenID Providers, enthalten sind [16].

- Wenn es sich um eine URL handelt, soll als erstes das Yadis Protokoll [35] verwendet werden. Mit Hilfe dieses Protokolls können Informationen über unterstützte Dienste auf Basis der URL-basierten Identität beschrieben und abgerufen werden. Dabei wird versucht, als Resultat ein XRDS Dokument zu erhalten. Dies geschieht entweder über die direkte Antwort oder über eine weitere Anfrage, nachdem der Dokumentort erfahren wurde. Verläuft das Protokoll erfolgreich, erhält die Relying Party als Ergebnis ebenfalls ein XRDS Dokument [16].

- Wenn das Yadis Protokoll nicht funktioniert und keine Serviceelemente, wie beispielsweise die URL, gefunden werden, sollte eine HTML-basierte Discovery gestartet werden [16]. Dabei werden bestimmte Attribute im Header der Antwort erwartet, welche auf die URL des Providers und auf den lokalen Identifier hinweisen.

Nach Abschluss der Discovery-Phase hat die Relying Party alle relevanten Informationen, um eine Anfrage an den Openid Provider Endpoint zu stellen. Die Relying Party und der Provider generieren eine optionale Assoziation im Sinne einer eindeutigen Verbindung zueinander. Dies geschieht mit Hilfe eines gemeinsamen Geheimnisses, welches das Resultat des *Diffie-Hellman Key*

Exchange (D-H) [40] ist. Mit Hilfe des D-H kann ein gemeinsames Geheimnis über einen unsicheren Kommunikationskanal, wie zum Beispiel das Internet, ausgetauscht werden. Dabei erfolgt der Schlüsselaustausch auf prime Restklassengruppen. Der Provider verwendet die oben genannte Assoziation, um seine darauffolgenden Nachrichten zu signieren, welche die Relying Party wiederum mit Hilfe des gemeinsamen Geheimnisses verifizieren kann. Dadurch wird die Notwendigkeit von zusätzlichen Anfragen zur Nachrichtenverifizierung umgangen [16].
Im nächsten Schritt wird der User-Agent des Nutzers an den OpenID Provider mit einer Authentifizierungsanfrage weitergeleitet (Nachricht „redirects user to OpenID Provider“ in Abbildung 4). Der Provider authentifiziert daraufhin den Principal (der anfragende Nutzer). Der Nutzer wird mit einer Zusicherung des Providers bezüglich des Erfolgs oder Misserfolgs der Authentifizierung zurück an die Relying Party geleitet (Nachricht „redirects user to Relying Party with Authentication Response“ in Abbildung 4). Diese Zusicherung wird von der Relying Party abermals überprüft, indem die OpenID Provider URL, die Nonce[1], welche anfangs generiert wurde, sowie die Signatur des Providers verifiziert werden (Nachricht „validates authentication response“ in Abbildung 4) [16]. Infolgedessen erhält der Principal Zugriff auf die Relying Party, ohne sich dort anmelden oder seine Zugriffsdaten preisgeben zu müssen.
Eine wichtige Erweiterung von OpenID ist die *OpenID Attribute Exchange* Erweiterung [15]. Mit Hilfe dieser können verschiedene Endpunkte Informationen über Nutzer austauschen. Beispielsweise kann die Relying Party ergänzend zu der Authentifizierungsanfrage zusätzliche Informationen anfragen. Üblicherweise werden Informationen wie die E-Mail-Adresse, der Name sowie das Geburtsdatum abgefragt. Dabei werden diese Parameter bei der Authentifizierungsanfrage mitgesendet. Der Principal wird daraufhin während der Authentifizierung gefragt, ob er diese Informationen preisgeben möchte. Mit diesen Informationen könnte die Relying Party eigene Nutzer und Identifier anlegen.

[1]Eine Nonce bezeichnet im Kontext von Kryptographie, einen zufällig generierten und eindeutigen Sitzungsschlüssel, welcher nur einmal verwendet wird.

3.3.1 Komplexität

Die Entkoppelung von Relying Party und Provider erhöht sowohl die Modularität als auch die Komplexität, weil beispielsweise stets eine Discovery Phase durchgeführt werden muss. Durch die optionale, aber durchaus wichtige Assoziation zwischen Provider und Relying Party, wird abermals die Komplexität erhöht, da beide Parteien mit Hilfe von Diffie-Hellman Key-Agreement ein gemeinsames Geheimnis erzeugen. Weiterhin handelt es sich um HTTP Anfragen, die Parameter beinhalten, welche von der jeweiligen Partei empfangen, überprüft und entsprechend bearbeitet werden. Diese Anfragen erhöhen die Komplexität jedoch nicht.

3.3.2 Sicherheitsaspekte

Eavesdropping Attacken
Als Eavesdropping Attacken bezeichnet man Angriffe, bei welchen ein Kommunikationskanal abgehört wird, um sicherheitsrelevante Informationen zu erfassen und selber zu verwenden. Die Anfälligkeit des Protokolls gegen Eavesdropping Attacken wird durch die Überprüfung der Nonce am Ende des Protokolls verhindert. In diesem Schritt könnte der Eavesdropper ebenfalls die Zusicherung zur Authentifizierung abfangen und selber verwenden. Der Einsatz von Transport Layer Security beziehungsweise die Überprüfung der Nonce bietet Schutz vor diesem Angriff [16].

User-Agents
Vorrangig werden als User-Agents Webbrowser verwendet. Dabei kann es vorkommen, dass diese von Malware oder Spyware befallen sind. Folglich kann der Zusicherung der Authentifizierung nicht mehr vollständig vertraut werden. Da dieses Problem nicht gelöst werden kann, gehen die meisten aktuellen Protokolle von funktionierenden Webbrowsern und Hosts aus [16].
Ebenfalls könnten Cross-site-scripting (XSS) Attacken gegen den OpenID Provider verwendet werden, um denselben Effekt zu erreichen. Im Rahmen eines XSS Angriffs wird ein Scriptfragment (üblicherweise Javascript) in eine dem Endnutzer zu präsentierende HTML Seite injiziert. Dadurch kann

dieses Script, welches durch den Browser ausgeführt wird, unter Umständen den vertrauenswürdigen Kontext des Nutzers, wie beispielsweise Session oder Cookies in Folge einer Anmeldung, ausnutzen, um Zugriff auf geschützte Ressourcen zu erhalten. Nach Fitzpatrick et al. sollten die Provider auf die Nutzung von clientseitig auszuführendem Scripting (in der Regel Javascript) bei der Generierung der HTML Seiten, die dem Endnutzer präsentiert werden, verzichten, sodass das Protokoll auch bei einem Verbot des Scripting durch die User-Agents ablaufen kann. [16].

Denial of Service Attacken
Das Protokoll ist gegen DoS-Attacken anfällig, sofern es eine bösartige Relying Party gibt. Der OpenID Provider hat keine Möglichkeit zur Überprüfung der Authentizität einer Anfrage, da es keine Nachrichten im Protokollfluss diesbezüglich gibt. Folglich kann die bösartige Relying Party zahlreiche Anfragen an Assoziationen, Authentifizierungsanfragen oder Verifizierungen von Signaturen stellen, was unter Umständen zu einem Absturz führen könnte. Die kritischste Situationen findet während der Phase der Assoziation statt, da bei dieser eine diskrete Potenzierung durchgeführt wird [16]. Da die Relying Party sowohl den Modulus als auch den Generator dafür pro Nachricht angeben kann, könnte der Angreifer dadurch den OpenID Provider zwingen, dies real für jede Nachricht durchführen zu müssen. Diese Attacke kann umgangen werden, in dem IP-basierte rate-limiter oder Banntechniken eingesetzt werden.
Weiterführende Sicherheitsaspekte, insbesondere auf das User Interface und die Identifier bezogene, können im offenen Standard für OpenID [16] nachgelesen werden.

3.4 Security Assertion Markup Language (SAML)

Die *Security Assertion Markup Language (SAML)* wurde von dem Security Services Committee von der *Organization for the Advancement of Structured Information Standards (OASIS)* entwickelt [13, 14]. Es handelt sich dabei um ein XML-basiertes Framework, welches sowohl die Authentifizierung als

auch die Autorisierung von Nutzern erlaubt. Durch SAML wird ermöglicht, dass Entitäten verschiedene Assertions über einzelne Identitäten, Attribute und Zugriffsberechtigungen erhalten können.
Grundsätzlich gibt es drei Parteien: einen Principal, einen Identity Provider und einen Service Provider [14]. Der Service Provider stellt den Service zur Verfügung, welcher die zugriffsgeschützte Ressource enthält. Der Identity Provider verwaltet abermals die Nutzeridentitäten, während der Principal den Nutzer darstellt, welcher auf die geschützte Ressource zugreift. Im Gegensatz zu den anderen Systemen ist das Haupteinsatzgebiet von SAML das Web Single-Sign-On [13]. In diesem Use Case fordert der Principal einen Dienst vom Service Provider an. Dieser fragt daraufhin den Identity Provider nach der Identität des Principals und erhält diese Identität als Assertion. Aufgrund dieser Assertion kann der Service Provider entscheiden, ob der Principal zum Zugriff autorisiert ist. Bevor der Identity Provider die Assertion zum Service Provider sendet, muss der Principal authentifiziert werden. Dies geschieht wiederum meist mit Benutzernamen und Passwort.
In Abbildung 5 ist der Protokollfluss von SAML 2.0 erkennbar. Dieser beginnt abermals mit der Anfrage an eine geschützte Ressource vom Service Provider (Nachricht „wants to access resource“). Daraufhin führt der Service Provider eine Überprüfung im Auftrag der Ressource durch und versucht einen validen Sicherheitskontext, das bedeutet, Informationen zur aktuellen Sitzung des Nutzers, zu finden. Bei einer positiven Rückmeldung kann die Ressource an den Principal versendet werden, da der Nutzer in diesem Fall bereits bekannt und authentifiziert ist. Anderenfalls wird der Benutzer an den Identity Provider weitergeleitet, wobei ein *SAMLRequest* Parameter mitgesendet wird. Dieser Parameter beinhaltet ein Base64 encodiertes *<samlp:AuthnRequest>* Element[2]. Außerdem wird ein *RelayState Token* der Anfrage hinzugefügt, welches der Service Provider intern sichert (Nachricht „redirect to IdP with SAMLRequest and RelayState Token“). Im nächsten Schritt extrahiert und verarbeitet der Identity Provider das <samlp:AuthnRequest> Element. Da-

[2]Dieses Element enthält verschiedene Informationen wie etwa die ID oder den Typ der Nachricht und weitere Informationen wie etwa, ob die Authentifizierung forciert werden soll.

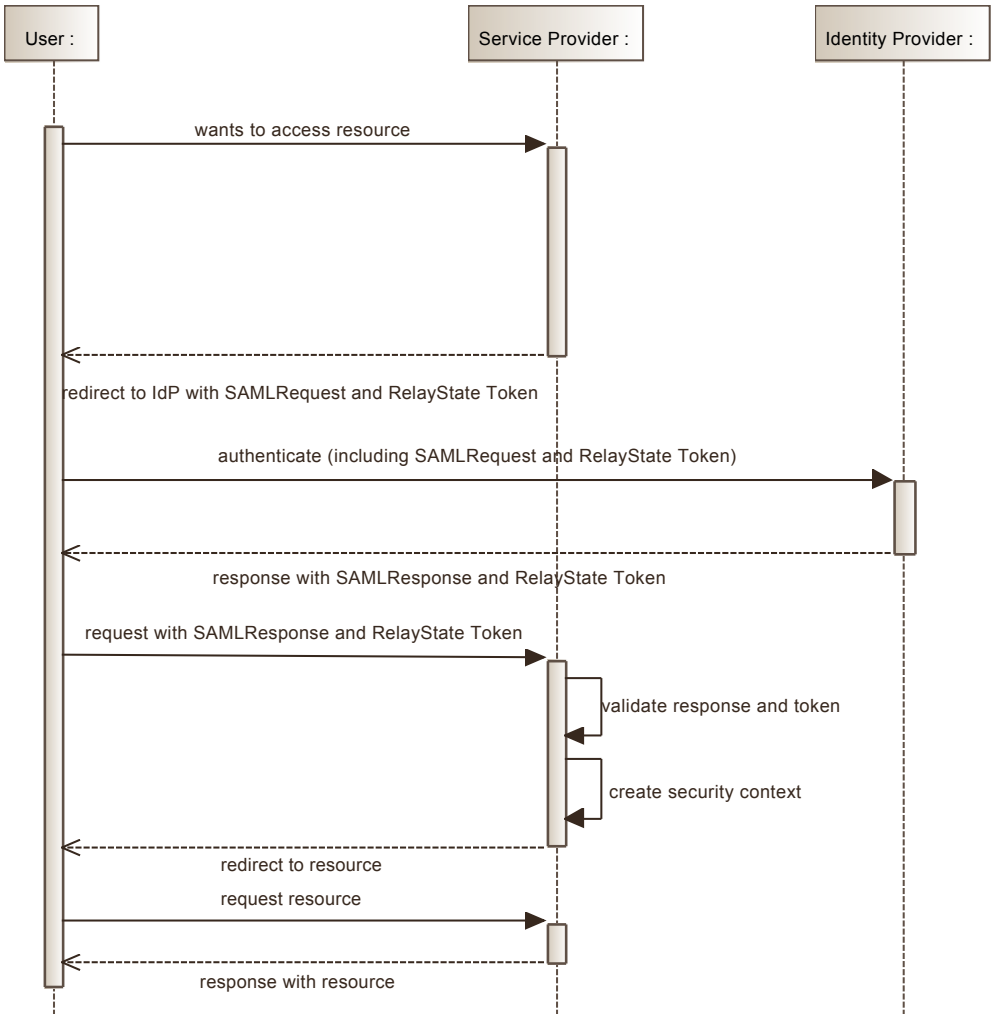


Abbildung 5: Der Protokollfluss von SAML

bei wird abermals überprüft, ob der Nutzer bereits einen aktuellen Sicherheitskontext beim Identity Provider besitzt. Falls dies nicht zutrifft, würde der Principal daraufhin authentifiziert werden. Bei einer erfolgreichen Authentifizierung antwortet der Identity Provider mit einem XHTML Formular, welches einen *SAMLResponse* Parameter, mit einem *<samlp:Response>* Element[3], sowie das zuvor erhaltene RelayState Token beinhaltet. Im Folgenden

[3]Dieses Element enthält verschiedene Informationen wie etwa die ID oder den Typ der Nachricht und weitere Informationen wie beispielsweise eine Assertion, ob die Authentifizierung erfolgreich war. Diese Assertion ist signiert.

startet die Anfrage des Nutzer an den Service Provider, welche das Formular beinhaltet (Nachricht „request with SAMLResponse and RelayState Token“). Daraufhin extrahiert der Service Provider das <samlp:Response> Element sowie das RelayState Token und validiert beides. Im Anschluss daran wird ein Sicherheitskontext erstellt, wobei der Service Provider den Nutzer an die Zielressource weiterleitet. Abschließend führt der Nutzer eine erneute Anfrage auf diese Ressource durch. Da ein valider Sicherheitskontext existiert, kann er darauf zugreifen (Nachrichten „request resource“ und „response with resource“).
Es ist festzustellen, dass keine direkte Kommunikation zwischen den einzelnen Instanzen verläuft, sondern der Nutzer stets als Mediator fungiert. Um eine erhöhte Sicherheit zu erzielen, wäre es auch möglich, dass die Nachrichten per SOAP-Referenzen übertragen werden. Der Nutzer würde dabei als Mediator wegfallen, weshalb er Nachrichten nicht mehr verändern könnte und folglich der bösartige Eingriff in das Protokoll schwieriger wäre.

3.4.1 Komplexität

Die Komplexität ist als weitestgehend gering zu bezeichnen. Es werden standardisierte HTTP Anfragen verwendet, in welchen entsprechende Parameter mitgesendet werden. Diese werden vom Empfänger extrahiert und entsprechend verarbeitet.

3.4.2 Sicherheitsaspekte

Das SAML Protokoll ist anfällig gegen Denial-of-Service Attacken. Die Bearbeitung einer SAML-Anfrage ist eine potentiell aufwändige Operation, da dabei die Nachricht geparst, Datenbank-lookups durchgeführt sowie eine Antwortnachricht erzeugt werden und eventuell Signaturen generiert werden müssen. Folglich kann ein Angreifer zahlreiche Anfragen an das System stellen und es dadurch eventuell zum Absturz bringen. Eine mögliche Schutzmaßnahme ist die Verwendung von Client-Authentifizierung. Dabei ist die vorherige Registrierung des Clients notwendig, sodass infolgedessen nur authentifizierte Clients Anfragen stellen können. Sollten diese Clients auffällig

werden, könnten diese leicht blockiert werden [21].
Während des gesamten Ablaufs besteht die prinzipielle Gefahr von Eavesdropping Angriffen. In Fällen, in welchen Vertraulichkeit verlangt wird, müssen die Anfragen durch die darunter liegenden Kommunikationsprotokollen gesichert werden. Dafür eignet sich entweder die Verwendung von TLS/SSL oder IPSec [21]. Ebenfalls wird durch die Verwendung von TLS/SSL die Möglichkeit von Man-in-the-middle Angriffen stark reduziert. Bei SAML werden die Parameter in den HTTP Anfragen mitgesendet. Zusätzlich zur Verwendung von Verschlüsselung bei der Übertragung, sollten XML Signaturen und XML Verschlüsselung für die einzelnen Nachrichten genutzt werden. Dies würde abermals die Sicherheit erhöhen, da nicht davon ausgegangen werden kann, dass die Implementierung von SSL immer fehlerfrei verläuft [21].
Weitere Sicherheitsaspekte können in „Security and Privacy Considerations for the OASIS Security Markup Language V2.0“ [21] nachgelesen werden.

3.5 Shibboleth

Shibboleth wurde von Internet2/MACE entwickelt und ist ein Authentifizierungs- und Autorisierungssystem für Webservices [22]. Ein Nutzer soll sich einmalig authentifizieren, um somit Dienste verschiedener Anbieter nutzen zu können. Shibboleth basiert auf einer Erweiterung des bereits vorgestellten Standards SAML [22]. Dabei besteht Shibboleth ebenfalls aus drei Parteien: dem Identity Provider, dem Service Provider und einem optionalen Discovery Service [8]. Aufgabe des Identity Providers ist die Verwaltung und Überprüfung der Nutzeridentitäten. Der Service Provider übernimmt hingegen die Sicherung des Webservices. Mit Hilfe des Discovery Services kann die Verbindung zwischen verschiedenen Identity Providern und Service Provider hergestellt werden. Diese Komponenten können unabhängig voneinander installiert werden.
In Abbildung 6 wird sichtbar, dass der Protokollfluss mit dem Zugriff eines Nutzers auf eine geschützte Ressource beginnt. Sofern der Nutzer aktuell über keine aktive Sitzung verfügt, wird er an den Service Provider weitergeleitet (siehe Abbildung 2: Nachricht „wants to access resource“). Dieser star-

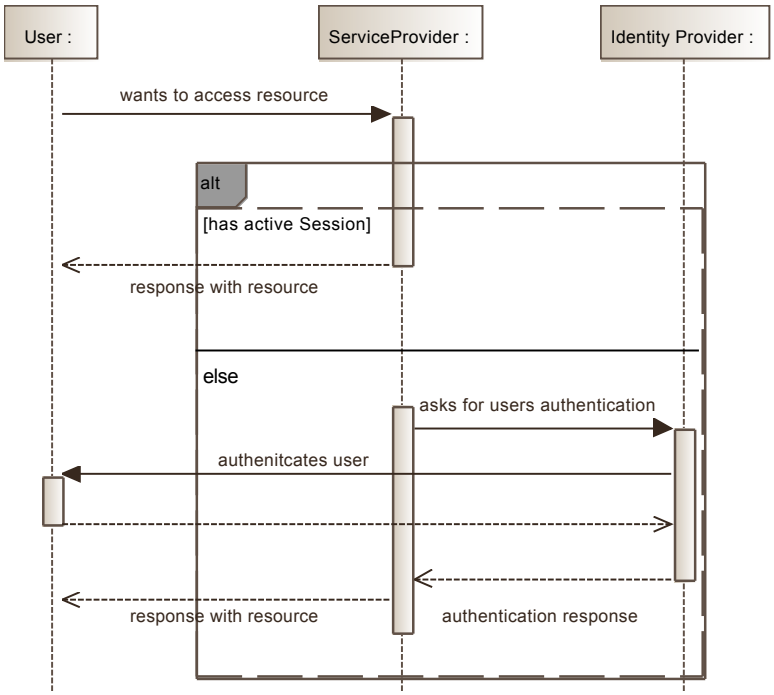


Abbildung 6: Der Protokollfluss von Shibboleth

tet daraufhin den Single-Sign-On (SSO) Prozess mit diesem Nutzer. Dafür wird sowohl die Authentifizerungsanfrage als auch der Nutzer an den Identity Provider gesendet, welcher den Nutzer authentifiziert und eine Authentifizierungsantwort generiert. Diese wird wiederum zusammen mit dem Nutzer an den Service Provider zurückgesendet (siehe Abbildung 2: Nachrichten „asks for users authentication“ bis „authentication response“) . Abschließend wird die Authentifizierungsantwort vom Service Provider überprüft, welche im Falle von Gültigkeit zur Erzeugung einer Sitzung für den Nutzer sowie zu weiteren Information über ihn führt. Ab diesem Schritt kann der Nutzer auf die geschützte Ressource zugreifen, da der Service Provider über die aktive Sitzung die Identität des Nutzers überprüfen kann. Betont sei an dieser Stelle noch einmal, dass der Identity Provider unabhängig vom Service Provider arbeitet (et vice versa) und daher beliebig ausgetauscht werden kann.
Aufgrund der oben beschriebenen Modularität ergibt sich die Frage, wie gewährleistet werden kann, dass der Service Provider den Nutzer an den richtigen Identity Provider sendet. Dies wird dem Nutzer überlassen; er erhält

mit Hilfe des Discovery Providers die Möglichkeit, über ein User Interface den entsprechenden Identity Provider auszuwählen [9]. Dafür gibt es einen eingebunden Discovery Dienst, welcher direkt vom Service Provider für Darstellung der Auswahl der Identity Provider verwendet wird. Weiterhin können die Service Provider diese Aufgabe an einen zentralen Discovery Dienst, welcher auch als eigenständige Applikation vorhanden ist, delegieren. Weitere Nutzerinformationen (beispielsweise die E-Mail-Adresse, spezielle Privilegien et cetera) können über den Identity Provider erhalten werden. Hierbei spielt die Privatsphäre des Nutzers eine zentrale Rolle. Abfragbare Informationen werden in Policies sowohl im Service Provider als auch im Identity Provider festgehalten. Die Policy im Service Provider repräsentiert die Werte, die nur vom Service Provider akzeptiert werden, wohingegen die Policy im Identity Provider die Informationen beinhaltet, welche angefragt werden dürfen [9].

3.5.1 Komplexität

Die funktionale Unabhängigkeit der untereinander agierenden Parteien erhöht die Komplexität des Protokolls, da die einzelnen Parteien vorab bestimmte Informationen, wie beispielsweise die IP-Adresse, austauschen müssen, was mit Hilfe des Discovery Service umgesetzt werden kann. Dazu werden außerdem standardisierte HTTP Anfragen und -Weiterleitungen verwendet.

3.5.2 Sicherheitsaspekte

Die Unabhängigkeit der einzelnen Parteien sowie der Umstand, dass jeder Service Provider verschiedene Identity Provider kontaktieren kann, erhöht nicht nur die Komplexität des Protokolls, sondern führt auch dazu, dass sich die Möglichkeit eines DoS Angriffs auf einzelne Identity Provider steigert. Im Falle, dass kein vernünftiges Load Balancing zwischen den Service Providern und den Identity Provider vorhanden ist, könnte es zu einer Situation kommen, bei der mehrere Service Provider auf denselben Identity Provider zeigen. Bei einem DoS Angriff erhielten die entsprechenden Service Provider jeweils Anfragen und würden all diese an denselben Identity Provider leiten,

sodass dieser ein Vielfaches an Last bearbeiten sollte. Entsprechend besteht die Gefahr der Nichtverfügbarkeit dieses einzelnen Identity Providers, im Falle einer schlechten Verteilung der Anfragen zwischen den Service Providern und den dazugehörigen Identity Providern.
Des Weiteren besteht die Möglichkeit, dass der Service Provider Nutzerinformationen abfragen könnte, obwohl dieser nur den Zugriff des Nutzers auf eine bestimmte Ressource legitimieren sollte. Mit Hilfe von Imitationsverfahren könnte dieser Umstand ausgenutzt werden, sodass Informationen des Nutzers unbefugt weitergegeben werden.

3.6 OAuth

OAuth ist ein offener Standard für Autorisierung, welcher aktuell in Version 2.0 vorhanden ist [28]. Es ist ein weit verbreitetes Protokoll für die Autorisierung und wird von allen gängigen Providern, die eine API anbieten, wie zum Beispiel Google, Twitter oder Facebook, unterstützt. Es werden vier Parteien verwendet: der Resource Owner, der Resource Server, der Client und ein Authorization Server. Der Resource Owner ist der Besitzer der zu schützenden Ressource, welche auf dem Resource Server gespeichert ist. Der Client kann eine Applikation oder ein auf die Ressourcen zugreifender Dienst sein, während der Authorization Server die Zugriffssicherung auf den Resource Server regelt.
OAuth basiert auf Token, welche symbolisieren, ob ein Client für den Zugriff auf die Daten eines Resource Owners autorisiert ist. Weiterhin wird es Nutzern ermöglicht, einem Client Zugriff auf ihre Daten zu gewähren, ohne dass alle Details der Zugangsberechtigungen veröffentlicht werden müssen, die zur Authentifizierung verwendet wurden [28].

In Abbildung 7 wird der Ablauf des OAuth-Protokolls skizziert. Ausgehend von einem Client, beispielsweise dem Browser oder einer Applikation, wird eine Autorisierungsanfrage an den Resource Owner gestellt. Diese kann direkt, wie in der Abbildung 7 mit der Nachricht „(A) Authorization Request“ dargestellt, oder über den Autorisierungsserver als Vermittler erfolgen. Der Re-

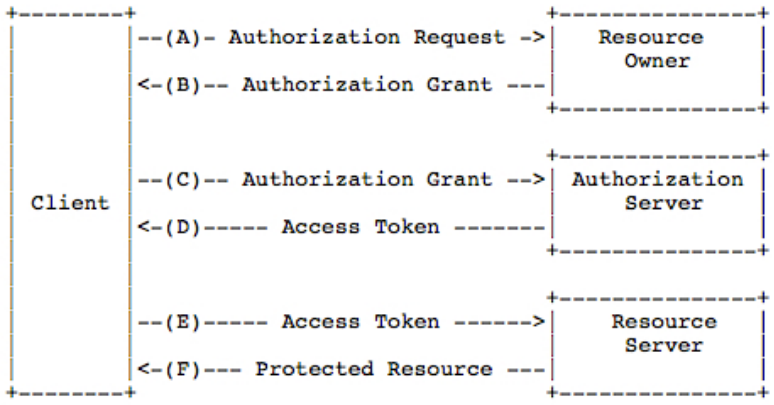


Abbildung 7: abstrakte Darstellung des Protokollflusses [28]

source Owner entscheidet, ob die Anfrage bestätigt oder abgewiesen wird. Davon ausgehend, dass der Resource Owner die Anfrage annimmt, erhält der Client eine Autorisierungsbestätigung, mit welcher dieser den Autorisierungsserver anfragt. Der Autorisierungsserver überprüft die Bestätigung auf Gültigkeit. Wenn die Überprüfung positiv ist, erhält der Client ein Access Token. Im Falle einer ungültigen Bestätigung würde er eine negative Rückmeldung erhalten. Mit dem Access Token kann der Client eine Anfrage an den Ressourcenserver senden. Dieser überprüft wiederholt das Access Token und antwortet bei Gültigkeit mit der zugriffsgeschützten Ressource des Resource Owners [28].

Es gibt verschiedene Möglichkeiten, ein Access Token zu erhalten, wie beispielsweise mit den Zugriffsdaten des Resource Owners, mit den Anmeldeinformationen des Clients, mit einem Autorisierungscode oder über eine implizite Anfrage [28]. Die Zugriffsdaten des Resource Owners sollten nur dann verwendet werden, wenn ein hoher Grad an Vertrauen zwischen Client und Resource Owner herrscht oder wenn es keine andere, vom Autorisierungsserver bereitgestellte Möglichkeit gibt, ein Token zu erhalten. Der Resource Owner würde in diesem Fall seine privaten Informationen veröffentlichen, was ein Sicherheitsrisiko darstellt. Die Anmeldeinformationen des Clients sind dann gut verwendbar, wenn der Client in seinem eigenen Auftrag agiert. Das bedeutet, wenn er gleichzeitig der Resource Owner ist, oder er Zugriff

auf geschützte Ressourcen verlangt, bei welchen die Autorisierung vorab mit dem Autorisierungsserver arrangiert wurde [28]. Der Ansatz mit Hilfe eines Autorisierungscodes ist sehr ähnlich zum impliziten Protokollablauf. Da der Ansatz mit einem Autorisierungscode in dieser Arbeit verfolgt wird, wird dieser im Folgenden detaillierter beschrieben und vergleichend zum impliziten Protokollablauf analysiert.

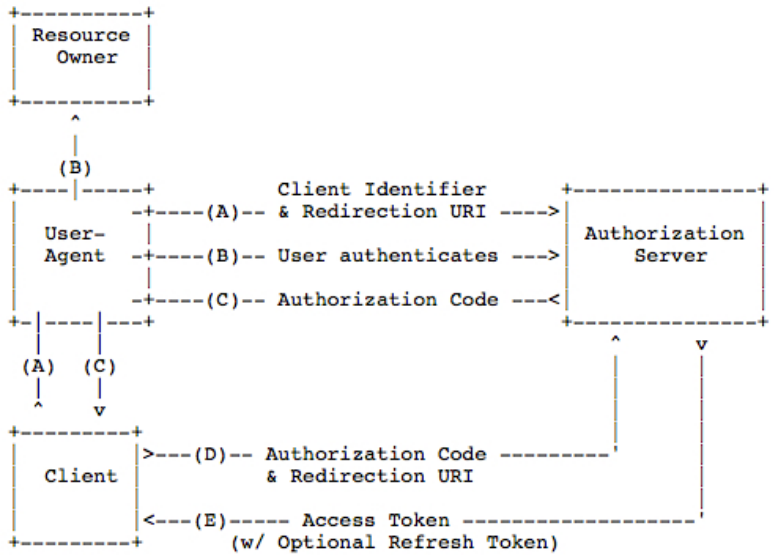


Abbildung 8: Der Protokollfluss von OAuth (Autorisierungscode) [28]

In Abbildung 8 ist erkennbar, dass der Client den Protokollfluss initiiert, indem er den User-Agent des Resource Owners zum Autorisierungsserver weiterleitet. Dieser authentifiziert den Resource Owner und prüft diesen hinsichtlich einer Annahme oder Ablehnung der Anfrage [28]. Sollte die Anfrage bestätigt werden, wird der User-Agent zum Client zurück geleitet. In dieser Rückleitung ist ein Autorisierungscode enthalten (Nachrichten (A) bis (C) in Abbildung 8). Mit Hilfe dieses Codes fordert der Client ein Access Token und ein Refresh Token an. Dafür sendet er eine Anfrage an den Autorisierungsserver, welcher den Autorisierungscode überprüft. Parallel dazu authentifiziert sich der Client beim Autorisierungsserver. Dadurch wird sichergestellt, dass nur dieser Client, für welchen der Code auch ausgestellt wurde, diesen verwenden kann. Wenn der Autorisierungscode gültig ist, er-

hält der Client daraufhin die angeforderten Access und Refresh Token, wobei das Access Token kurzlebig und das Refresh Token langlebig ist. Mit Hilfe des Access Tokens kann auf die geschützte Ressource zugegriffen werden (Nachrichten (D) und (E) in Abbildung 8) [28].
Beim impliziten Ablauf gibt es hingegen keinen Zwischenschritt mit Hilfe des Autorisierungscodes. Das Access Token ist das Resultat der Autorisierungsanfrage, wobei kein Refresh Token ausgestellt wird. Der Client wird ebenfalls nicht gesondert authentifiziert- die Wahrung verschiedener Sicherheitsaspekte wird auf die vorher registrierte Redirection URI delegiert.

3.6.1 Komplexität

Die Komplexität des Protokolls ist als gering zu bezeichnen, da lediglich HTTP Anfragen, welche die erforderlichen Parameter mitsenden, gestellt werden. Diese werden überprüft und es wird zudem sichergestellt, dass nur erlaubte Aktionen durchgeführt werden können.

3.6.2 Sicherheitsaspekte

Das Protokoll der zweiten Version (OAuth 2.0) wurde stark vereinfacht. Ziel war vor allem die Verständlichkeit des Nutzens, insbesondere für den Entwickler. Die Sicherheit des Protokolls wurde vollständig davon entkoppelt und auf die Transportebene übertragen. Das bedeutet, dass das Protokoll ausschließlich unter der Verwendung von *Transport Layer Security (TLS)* als sicher gilt. Anderenfalls ist es anfällig für Man-in-the-middle Attacken, bei welchen alle Codes und Token abgefangen und von Dritten genutzt werden können [28, 33].
Darüber hinaus ist es wichtig, dass der Client seine Anmeldeinformation geheim hält, da sonst eine Imitation des Clients möglich wird, wodurch sich andere bösartige Clients, als dieser ausgeben können. Ebenfalls muss sich der Client gegen *Cross-site request forgery (CSRF)* absichern, insbesondere bei seiner redirection URI [28]. Bei CSRF versucht ein Angreifer die Autorisierungsdaten eines Clients durch eine versteckte Weiterleitung oder einen versteckten Aufruf einer URL für den Zugriff auf geschützte Ressourcen zu

missbrauchen. Weitere Schutzmethoden vor potentiellen Angriffsszenarien können in Hardt (2012) [28] sowie in Lodderstedt, McGloin und Hunt (2013) [33] nachgelesen werden.

4 Anforderungsanalyse

In der Anforderungsanalyse werden verschiedene Voraussetzungen erarbeitet, welche die entwickelte Komponente erfüllen muss oder soll. Diese Voraussetzungen sind prozessual entstanden und wurden oft verändert, angepasst und verfeinert. Die in diesem Abschnitt identifizierten Anforderungen werden mit Hilfe von RFC 2119 gemäß ihrer Anforderungslevel identifiziert [25]. Dabei wurden die Begriffe MUST (MUSS), MUST NOT (DARF NICHT), SHOULD (SOLLTE), SHOULD NOT (SOLLTE NICHT) und MAY (KANN) von Bradner im Original definiert. In dieser Arbeit wird die entsprechende deutsche Übersetzung für die Begriffe verwendet. Unter dem Begriff MUSS werden daher Anforderungen definiert, welche erfüllt werden müssen. DARF NICHT sind hingegen negative Anforderungen, welche sinngemäß nicht realisiert werden sollen. Anforderungen welche unter den Begriffen SOLLTE und SOLLTE NICHT erfasst werden, entsprechen konkreten Empfehlungen. Unter dem Begriff KANN sind demgegenüber fakultative Aufgaben formuliert, deren Erledigung optional ist.
Die Anforderungsanalyse wird in funktionale und nicht-funktionale Anforderungen gegliedert. Funktionale Anforderungen sind Voraussetzungen, welche bestimmen, was für eine Funktionalität beziehungsweise welche Features das System erfüllen soll oder muss. Beispiele für solche Voraussetzungen sind die Art und Weise der Realisierung bestimmter Dienste, die Verarbeitung von Werten sowie das Aussehen von Ausgaben des Systems. Nicht-Funktionale Anforderungen stellen dahingegen zusätzliche Anforderungen dar, welche sich unter Anderem auf Qualitätsaspekte beziehen können. Einige Beispiele dafür sind die Sicherheit, die Performanz oder die Stabilität des Systems. Ebenfalls fallen Erwartungen an die Benutzbarkeit des Systems unter den Aspekt der nicht-funktionalen Anforderungen.

Funktionale Anforderungen

Anforderung 1 Clients[4], welche auf die geschützten Services zugreifen

[4]Mit Clients werden Services oder Applikationen bezeichnet, die das im Rahmen dieser Arbeit entwickelte System nutzen.

möchten, MÜSSEN sich registrieren können. Die Registrierung MUSS über eine externe Schnittstelle möglich sein und SOLLTE ebenfalls über ein Formular erfüllbar sein.
Clients sind ein integraler Bestandteil und müssen folglich unterstützt werden.

Anforderung 2 Die bei der Registrierung gewonnenen Informationen über den Client MÜSSEN persistiert werden.
Dieser Schritt ist im weiteren Verlauf für die Identifizierung und Authentifizierung des Clients notwendig.

Anforderung 3 An Anforderung 1 anschließend MUSS die Komponente sicherstellen, dass kein Client sich mehrfach registrieren kann.
Dadurch wird das Sicherheitsrisiko der Nachahmung eines Clients unterbunden beziehungsweise reduziert.

Anforderung 4 Endnutzer MÜSSEN sich registrieren können. Sofern kein externer OpenID Provider verwendet wird, muss diese Komponente die Registrierung übernehmen, damit Nutzer fortan über eine OpenID verfügen. Die Registrierung SOLLTE über ein Formular erfolgen können.
Auch hier gilt, dass Nutzer ein wichtiger Bestandteil sind und daher nicht davon ausgegangen werden kann, dass diese extern gehalten werden.

Anforderung 5 Analog zu Anforderung 2 MÜSSEN die Informationen des Endnutzers persistiert werden
Dies ist wiederholt für die Identifizierung und Authentifizierung des Endnutzers erforderlich.

Anforderung 6 Parallel zu Anforderung 3 MUSS die Komponente sicherstellen, dass sich mehrere Benutzer nicht unter dem selben Nutzernamen registrieren können.
Dadurch soll an dieser Stelle wiederholt das Sicherheitsrisiko der Imitation eines Nutzers reduziert werden.

Anforderung 7 Die entwickelte Komponente MUSS den Client vor der Verwendung der Services vom Nutzer autorisieren.
Dies stellt sicher, dass kein Client auf die Daten eines Nutzers (die in den Services hinterlegt sind) zugreifen kann.

Anforderung 8 Aufbauend auf Anforderung 7 MUSS der Nutzer authentifiziert werden, bevor die Autorisierung eines Clients abgeschlossen werden kann.
Dies stellt sicher, dass kein Angreifer einen Client autorisieren kann, ohne dass der Nutzer vorher gefragt wurde.

Anforderung 9 Aufbauend auf Anforderungen 7 und 8 MUSS der Client authentifiziert werden, bevor dieser eine endgültige Zugriffsberechtigung erhält.
Dadurch wird sichergestellt, dass kein Dritter einen Client nachahmen und in dessen Namen Zugriff erhalten kann.

Anforderung 10 Die Komponente MUSS bei jeder Anfrage überprüfen, ob die notwendigen Informationen bezüglich der Autorisierung vom Client vorhanden und insbesondere auch valide sind.

Anforderung 11 Das Identitätsmanagement der Komponente SOLLTE in gängigen Systemen[5] leicht integrierbar sein.

Nicht-Funktionale Anforderungen
Anforderung 12 Die Komponente MUSS stabil laufen.
Mit *Stabilität* bezeichnet man die Fähigkeit des Systems, auf verlässliche Art und Weise Aufträge bei erhöhter Anzahl und auch von invaliden Anfragen zu bearbeiten. Dadurch wird ein reibungsloser Ablauf sichergestellt. Läuft die Komponente nicht stabil, kann es vorkommen, dass der Zugriff auf die geschützten Services nicht mehr möglich ist.

[5]zum Beispiel Comprehensive Knowledge Archive Network (CKAN), Liferay et cetera

Anforderung 13 Die Komponente SOLLTE performant sein.
Das bedeutet, dass die Antwortzeiten in einem üblichen Rahmen liegen sollten. Dabei muss beachtet werden, bei welchen Anfragen auf eine zusätzliche Datenbank zugegriffen werden muss, da dies die Antwortzeiten erhöhen könnte.

Anforderung 14 Der Ressourcenbedarf SOLLTE gering sein.
Die gemeinsame Nutzung der vorhandenen Eigenschaften eines Systems von mehreren Komponenten würde durch einen geringen Bedarf an Ressourcen ermöglicht werden.

Anforderung 15a Die Integrierte Komponente zur Sicherung von Cloud Services SOLLTE benutzerfreundlich für den Entwickler sein, das heißt es müssen wohldefinierte und leicht bedienbare Schnittstellen angeboten werden.
Der Entwickler stellt sicher, dass sein Client den Anforderungen entspricht und ordnungsgemäß auf die Komponente zugreift.

Anforderung 15b Die Komponente SOLLTE benutzerfreundlich für den Endnutzer sein.
Dieser möchte den Client eines Entwicklers nutzen, wobei es vorkommt, dass sich dieser authentifizieren muss. Dies sollte möglichst einfach und komfortabel umgesetzt werden können.

Anforderung 16 Die Komponente SOLLTE leicht installierbar und in bestehende Services integrierbar sein.
Durch die Erfüllung dieser Anforderung würde sich die Möglichkeit der Weiternutzung erhöhen.

Anforderung 17 Die Komponente SOLLTE sicher bezüglich der Vertraulichkeit und Datenintegrität sein, wobei eine hundertprozentige Sicherheit nicht gewährleistet werden kann.

5 Architektur

In diesem Kapitel wird sowohl die Architektur der Mobilitätsdaten-Cloud (MDC) als auch der integrierten Komponente zu Sicherung von Cloud Services (ISCS) vorgestellt.

5.1 Mobilitätsdaten-Cloud

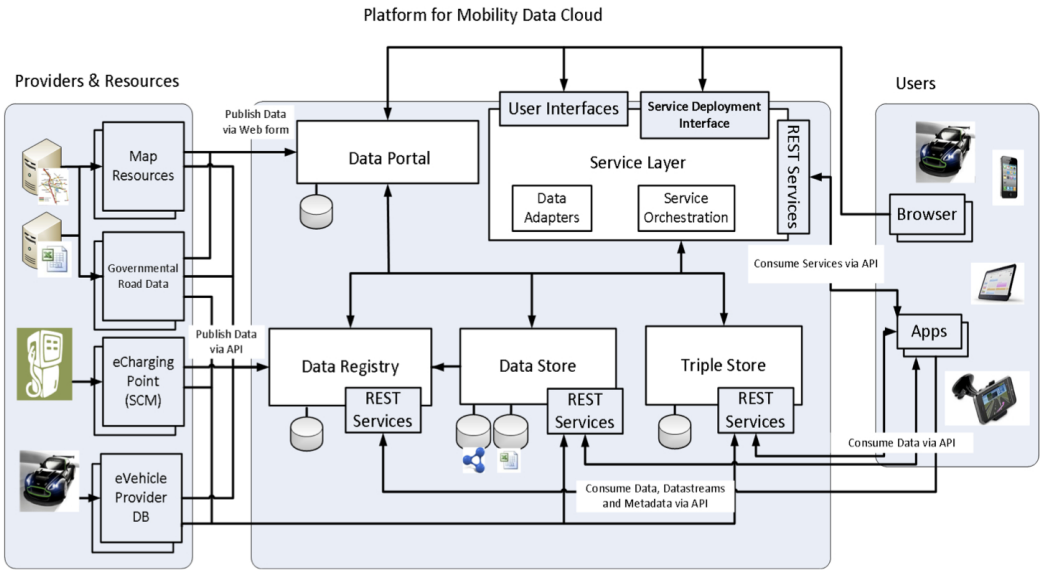


Abbildung 9: Darstellung der Architektur der Mobilitätsdaten-Cloud [45]

Generelle Architektur

In Abbildung 9 ist die Architektur der MDC zu sehen. Die MDC verfolgt die Intention, verschiedene mobilitätsrelevante Daten über eine zentralisierte Plattform anzubieten. Auf der linken Seite sind verschiedene Datenquellen, wie beispielsweise Kartenressourcen („Map Resources“) oder Regierungsdaten („Governmental Road Data“) dargestellt, welche in die Datencloud integriert werden. Diese verschiedenen Arten von Daten werden über die MDC angeboten, sodass diese von diversen Applikationen und Services verwendet werden können. Diese Anwendungen sind auf der rechten Seite der Abbildung 9 verdeutlicht. Eine beispielhafte Anwendung könnte eine eingebettete Applikation in einer On-Board-Unit (OBU) sein oder eine Webanwendung, welche

über den Browser verwendet werden kann. Die Plattform der MDC ist im Zentrum der Abbildung 9 dargestellt. Diese besteht aus den Komponenten:

1. Data Portal
2. Data Registry
3. Data Store
4. Triple Store
5. Service Layer [45].

Die Komponenten 1, 2 und 3 bilden die Open Data[6] Plattform von Fraunhofer FOKUS [32]. Die Open Data Plattform ist in dem Projekt OpenCities [32] entstanden, wobei die MDC als Erweiterung auf diese Open Source Lösung angesehen werden kann. Im Folgenden werden die einzelnen Module der MDC beschrieben.

1) Data Portal
Das Datenportal ist eine der Komponenten, die auch in der Open Data Plattform vorhanden ist. Die Aufgabe des Portals besteht in der Präsentation der Daten der MDC an potentielle Nutzer, wie beispielsweise App-Entwickler. Zusätzlich bietet es die Möglichkeit, neue Daten und Datenquellen in die MDC hinzuzufügen. Dabei ist zu erwähnen, dass die MDC sowohl die Daten innerhalb ihrer Architektur als auch Verweise auf externe Quellen halten kann. Darüber hinaus verfügt das Portal über Social Media Fatures für Open Data Nutzer, wie zum Beispiel die Kommentarfunktion oder die Möglichkeit der Diskussion über ein Forum [32, 45].

[6]Daten werden als offen bezeichnet, wenn sie von jedem genutzt und verteilt werden dürfen, wobei die einzige Voraussetzung besteht, dass die Anforderungen des Urhebers berücksichtigt werden müssen, beispielsweise bei der Attributierung oder der Weitergabe (share-alike) [2].

2) Data Registry
Das Datenregister ist für die Katalogisierung und Verwaltung von Datenquellen innerhalb der MDC zuständig. Es hält somit die Anzahl an Metadaten-Einträgen, welche die Daten beschreiben, die über die MDC verfügbar sind. Dies betrifft sowohl Daten, welche extern verlinkt sind, als auch jene, welche innerhalb der Architektur gehalten werden [45].

3) Data Store
Der Data Store ist eine Komponente, welche die Speicherung beliebiger Daten ermöglicht. Dafür wird eine REST Schnittstelle auf die darunter liegende Datenbank bereitgestellt. Die Komponente kann entweder über eine NoSQL Datenbank [29] oder über die offene REST Schnittstelle auf die darunter liegende Datenbank realisiert werden. Diese Schnittstelle kann von Anwendungen und Diensten verwendet werden, um das Format für den Datenaustausch dynamisch zu definieren [45]. Eine Implementierung dieser Komponente wurde im Rahmen von *Eine webbasierte Infrastruktur zur Speicherung und Abfrage von strukturierten offenen Daten* [27] umgesetzt.

4) Triple Store
Der Triple Store ist eine weitere Komponente, welche ebenfalls die Möglichkeit bietet, Daten innerhalb der MDC zu speichern. Die Komponente erlaubt auf Ontologien basierende Linked Data zu speichern und über eine REST Schnittstelle nach außen zur Verfügung zu stellen (beispielsweise um SPARQL Anfragen an einen SPARQL Endpunkt zu stellen) [45].

5) Service Layer
Mit Hilfe des Service Layers können Mobilitätsdienste und -anwendungen auf Basis der Daten, welche in der MDC verfügbar sind, deployed werden. Dafür wurde eine Service Deployment Schnittstelle entwickelt, welche zuverlässigen Entwicklern das Hinzufügen neuer sowie die Aktualisierung bestehender Dienste erlaubt [45]. Dadurch ist eine Öffnung der Plattform im Sinne von Platform-as-a-Service (PaaS) realisiert [45]. Die oben erwähnten Dienste können die Daten von externen Datenquellen verwenden, welche durch

das Datenregister katalogisiert wurden. Dabei muss beachtet werden, dass eventuelle Adapter geschrieben werden müssen, welche die Daten aus ihrem Rohformat in das gewünschte Format transformieren. Ebenfalls können die Dienste spezifische Nutzerschnittstellen anbieten, beispielsweise bei webbasierten mobilen Applikationen, welche auch über das HTTP Protokoll als Teil des Service Layers anfragbar sind. Darüber hinaus bieten die Dienste eine REST Schnittstelle, über welche verschiedene JSON oder XML basierte APIs angefragt werden können. Die Dienste können außerdem miteinander kombiniert werden, sodass dadurch komplexere Services entstehen. Dies könnte beispielsweise mit Hilfe des Services Orchestration Moduls realisiert werden. Dabei werden die Dienste miteinander verkettet und ihre Ein- und Ausgaben werden jeweils an den nächsten Dienst weitergeleitet.

5.2 Integrierte Komponente zur Sicherung von Cloud Services

In diesem Kapitel werden die einzelnen Elemente der entwickelten ISCS aufgezeigt und hinsichtlich ihrer Existenz erläutert. Weiterhin werden die dynamische Aspekte des Systems visualisiert. Dynamische Aspekte beschreiben die Interaktionen und Informationsflüsse, welche die geforderte Funktionalität innerhalb der Komponente realisieren.

5.2.1 Identifizierte Komponenten

In Abbildung 10 ist die Architektur der integrierten Komponente visualisiert. Dabei sind auf der linken Seite die Open Data Plattform, auf welcher die MDC basiert, und die Clients dargestellt. Im Zentrum ist die Komponente mit ihren einzelnen Modulen abgebildet, während auf der rechten Seite eine Datenbank zu sehen ist, auf welche die Komponente zugreift.
Aus den im vierten Kapitel herausgearbeiteten Anforderungen 1, 3, 4 und 6 wird ersichtlich, dass es jeweils eine Komponente für die Registrierung von Benutzern und Clients geben muss. Um eine möglichst modulare und wartbare Lösung zu erstellen, müssen diese beiden Komponente voneinander unabhängig und getrennt arbeiten. In Abbildung 10 sind die Module *UserRe-*

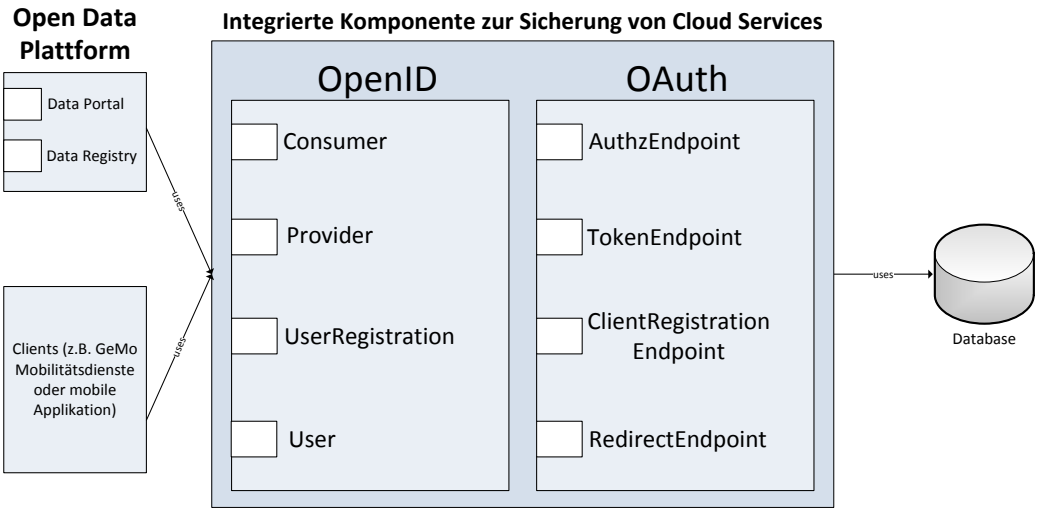


Abbildung 10: Darstellung der Architektur der entwickelten ISCS

gistration für die Benutzerregistrierung und *ClientRegistrationEndpoint* für die Clientregistrierung[7] vorgesehen. Ebenfalls wird die Unabhängigkeit der beiden Komponenten in der Abbildung verdeutlicht.
Bezugnehmend auf Anforderung 7 muss eine Komponente existieren, welche die Autorisierung der Clients übernimmt. Dafür ist der *AuthzEndpoint* vorgesehen. Da diese Autorisierung nur über die vorherige Authentifizierung des Nutzers erfolgen kann, muss dafür ebenfalls eine Komponente bereitgestellt werden. Für die Authentifizierung soll OpenID, was ein Community Standard ist, verwendet werden, weshalb es die beiden Module *Consumer* und *Provider* gibt, welche die Authentifizierung übernehmen. Diese Module sollten ebenfalls unabhängig von den bereits vorher genannten Komponenten sein. Darüber hinaus existiert ein *User* Modul, welches auf eine Anfrage hin ein XRDS Dokument als Antwort liefert, womit der *Provider* identifiziert werden kann.
Weiterhin muss ein Modul vorhanden sein, welches die Ausstellung und Validierung der Zugriffsberechtigungen verwaltet. In Abbildung 10 ist dafür der *TokenEndpoint* innerhalb der OAuth-Komponente vorgesehen. Zusätzlich gibt es einen *RedirectEndpoint*, welcher als Ausweich-Endpoint für Clients oh-

[7]Mit Client wird, wie durchgängig im Dokument erklärt, eine Applikation oder ein Dienst bezeichnet, der auf die ISCS zugreift.

ne eigene Redirect URI angesehen werden kann. Damit wird sichergestellt, dass die Daten auch an den anfragenden Client zurückgesendet werden.
Des Weiteren geben die Anforderung 2 und 5 an, dass alle Daten persistiert werden sollen. Daher muss es eine Datenbank (siehe Abbildung 10 „Database“) geben, auf welche möglichst unabhängig zugegriffen werden kann. Außerdem sollte der Austausch der darunter liegende Datenbank und der darauf aufbauenden Implementierung leicht möglich sein. Das kann mit Hilfe eines Persistenzframeworks, wie beispielsweise Hibernate [7], umgesetzt werden.

5.2.2 Dynamische Aspekte

In diesem Abschnitt werden die dynamischen Aspekte des Systems vorgestellt, was bedeutet, dass die Interaktionen, Aktionen und Informationsflüsse, welche die Funktionalität der Komponente bereitstellen, erklärt werden. Dabei werden die einzelnen Module der integrierten Komponente verdeutlicht, welche bei der jeweilige Aktion involviert sind.

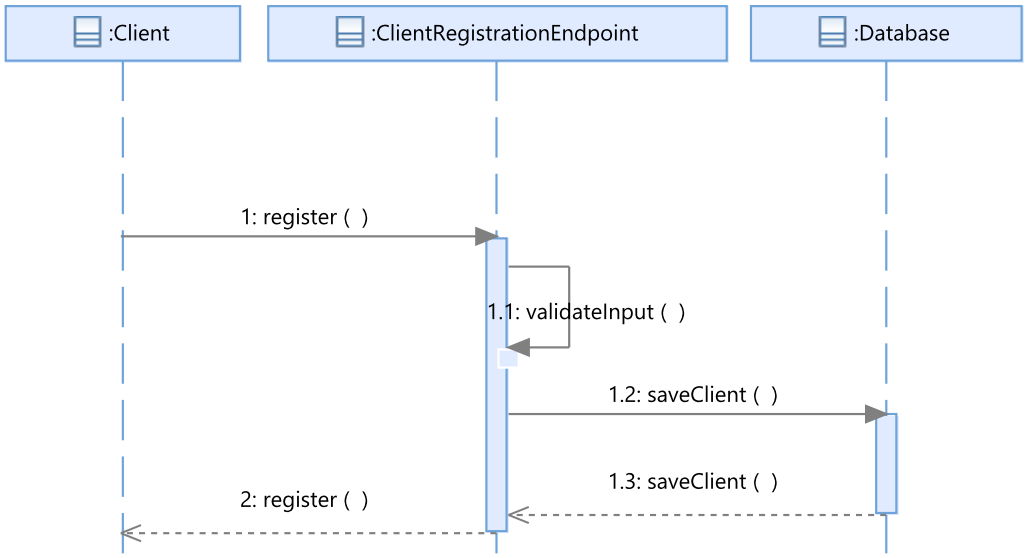


Abbildung 11: Darstellung der Clientregistrierung

In Abbildung 11 ist die Registrierung eines Clients dargestellt. Dabei stellt dieser eine Anfrage an den ClientRegistrationEndpoint, welcher zur OAuth-Teilkomponente gehört, mit den entsprechenden Parametern. Diese Einga-

beparameter werden validiert (Nachricht „validateInput()“ in Abbildung 11) und auf Gültigkeit geprüft. Wenn sie gültig sind, werden die Registrierungsinformationen in einer Datenbank gespeichert (Nachricht „saveClient()“ in Abbildung 11). Als Antwort erhält der Client ein neues Paar Client ID und Client Secret sowie ein Ausstellungsdatum und die Information, bis wann diese Zugangsdaten gültig sind. Mit diesen Aspekt werden die Anforderungen 1, 2 und 3 erfüllt, welche die wesentlichen Eigenschaften der Registrierung einer Applikation oder eines Dienstes bestimmen.

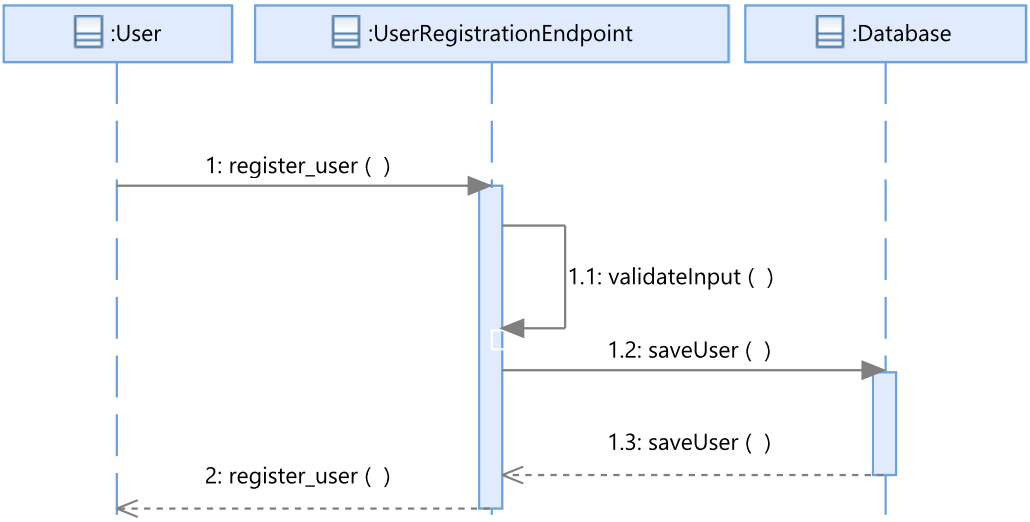


Abbildung 12: Darstellung der Benutzerregistrierung

In Abbildung 12 ist die Registrierung eines Nutzers verdeutlicht. Dabei stellt dieser eine Anfrage an den UserRegistration Endpunkt, der zur OpenID-Subkomponente gehört, mit den erforderlichen Parametern. Die Eingabeparameter werden ebenfalls validiert (Nachricht „validateInput()“ in Abbildung 12) und auf Gültigkeit geprüft. Die erhaltenen Nutzerinformationen werden in eine Datenbank gespeichert, sofern sie gültig sind (Nachricht „saveUser()“ in Abbildung 12). Bei erfolgreicher Registrierung erhält der Nutzer eine positive Rückmeldung. Dieser dynamische Aspekt stellt die Erfüllung der Anforderungen 4, 5 und 6 sicher, die bestimmend für die Registrierung eines Nutzers sind.

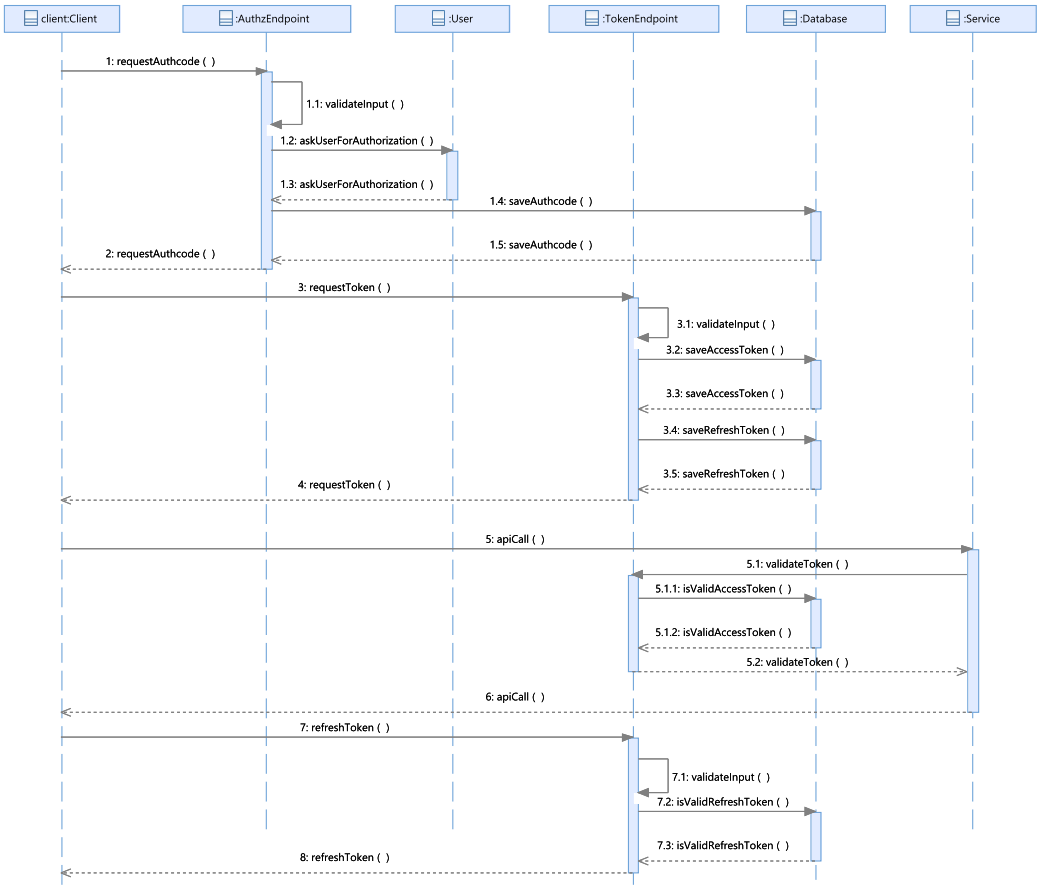


Abbildung 13: Darstellung der typischen Anfragen

In der Abbildung 13 sind die typischen Anfragen an die ISCS verdeutlicht. Die Darstellung beginnt mit der Anfrage nach einem Autorisierungscode. Dafür sendet der Client eine Anfrage an den AuthzEndpoint, welcher zur OAuth-Teilkomponente gehört, mit den erforderlichen Parametern (siehe Nachricht „requestAuthcode()“ in Abbildung 13). Im nächsten Schritt werden die gesendeten Informationen validiert (Nachricht „validateInput()“ in Abbildung 13) und erneut auf Gültigkeit getestet. Daran anknüpfend muss der Client vom Nutzer autorisiert werden (Nachricht „askUserForAuthorization()“ in Abbildung 13). Nachdem die Autorisierung beendet wurde, wird ein Autorisierungscode ausgestellt, der in der Datenbank gespeichert wird (Nachricht „saveAuthCode()“ in Abbildung 13).
Weiterhin ist der Austausch eines zuvor erhaltenen Autorisierungscodes für

ein Access Token und ein Refresh Token visualisiert (Nachricht „requestToken()“ in Abbildung 13). Dabei wird der Autorisierungscode an den TokenEndpoint, der zu der OAuth-Subkomponente gehört, gesendet und bei gültiger Authentifizierung des Clients gegen die oben genannten Token ausgetauscht. Die generierten Token werden zusätzlich in der Datenbank persistiert (Nachrichten „saveAccessToken()“ und „saveRefreshToken()“ in Abbildung 13). Somit werden die Anforderungen 7, 8 und 9 realisiert, welche die Autorisierung des Clients gemäß der Spezifikation von OAuth sicherstellen.
Daraufhin ist eine Anfrage an einen zugriffsgeschützten Service abgebildet (Nachricht „api_call()“ in Abbildung 13). An dieser Stelle lässt der Service die erhaltenen Parameter durch die ISCS validieren (Nachrichten „validateToken()“ und „isValidAccessToken()“ in Abbildung 13), womit Anforderung 10 erfüllt wird.
Access Token weisen eine begrenzte Gültigkeit auf, weshalb sie regelmäßig mit Hilfe von dem zuvor erhaltenen Refresh Token aktualisiert werden müssen. Dieser Vorgang wird in der Nachricht „refreshToken()“ in Abbildung 13 verdeutlicht. Dabei wird die Eingabe erneut validiert und das Refresh Token auf seine Gültigkeit überprüft (Nachrichten „validateInput()“ und „isValidRefreshToken()“ in Abbildung 13).
Abschließend kann man sich bei der Open Data Plattform mittels OpenID einloggen. Dies ist ohne Aufwand möglich, da diese Plattform auf Liferay basiert, welches die Authentifizierung per OpenID standardmäßig mitliefert. Dafür würde die integrierte Komponente das OpenID Protokoll starten und der Nutzer könnte sich mit seinen in der Komponente hinterlegten Identitätsinformationen anmelden, womit Anforderung 11 erfüllt ist.

6 Implementierung

In diesem Kapitel wird auf die Implementierung der integrierten Komponente zur Sicherung von Cloud Services eingegangen. Dabei werden insbesondere die beiden Teilkomponenten OpenID und OAuth sowie der Zugang zu den benötigten Datenbanken erläutert.

6.1 Verwendete Technologien

Das Framework *RESTEasy* wurde verwendet, da die Webdienste unter dem Programmier-Paradigma REST entwickelt werden sollen. Des Weiteren wurde für die Entwicklung der OpenID Komponente die Open Source Bibliothek *openid4java* verwendet, während für OAuth die Open Source Bibliothek *Apache Oltu* genutzt wurde. Durch die Verwendung von Open Source Lösungen wird die Open Source Stellung der integrierten Komponente vereinfacht.
Ebenfalls wurde für das Bildmanagement auf *Apache Maven* gesetzt. Maven basiert auf dem Konzept eines *Project Object Model (POM)*, mit welchem zentral der Bau, das Reporting und die Dokumentation des Projektes verwaltet werden können. Weiterhin wird es durch die Verwendung von Maven leicht möglich, dass dieses Projekt anderweitig installiert und verwendet werden kann. Damit wird die nicht-funktionale Anforderung 16 aus Kapitel 4 gelöst.
Des Weiteren wurde auf das JavaEE Feature *Contexts and Dependency Injection (CDI)* zurückgegriffen. Mit Hilfe von CDI ist es möglich, Komponenten in Applikationen auf eine typsichere Art und Weise zu injizieren. Das bedeutet, dass diese zu einem existierenden Programmgerüst hinzugefügt werden können. Dies beinhaltet ebenfalls die Möglichkeit während des Deployments zu wählen, welche Implementierung einer bestimmten Schnittstelle genutzt werden soll. Da im Kontext des Projekts GeMo *Apache Tomcat* als Webserver und Servletcontainer verwendet wird und dieser kein vollständiger JavaEE Server ist, musste zusätzlich *Weld* verwendet werden. Dies ist die Referenzimplementierung von CDI, wodurch Tomcat um die Möglichkeit von CDI erweitert wird. Darüber hinaus wurde für das Logging *log4j* von Apache verwendet.

6.2 Datenbanktechnologien

In diesem Abschnitt werden die Möglichkeiten für die Zugriffe auf die Datenbanken dargestellt und erläutert.

6.2.1 Autorisierungsdatenbank

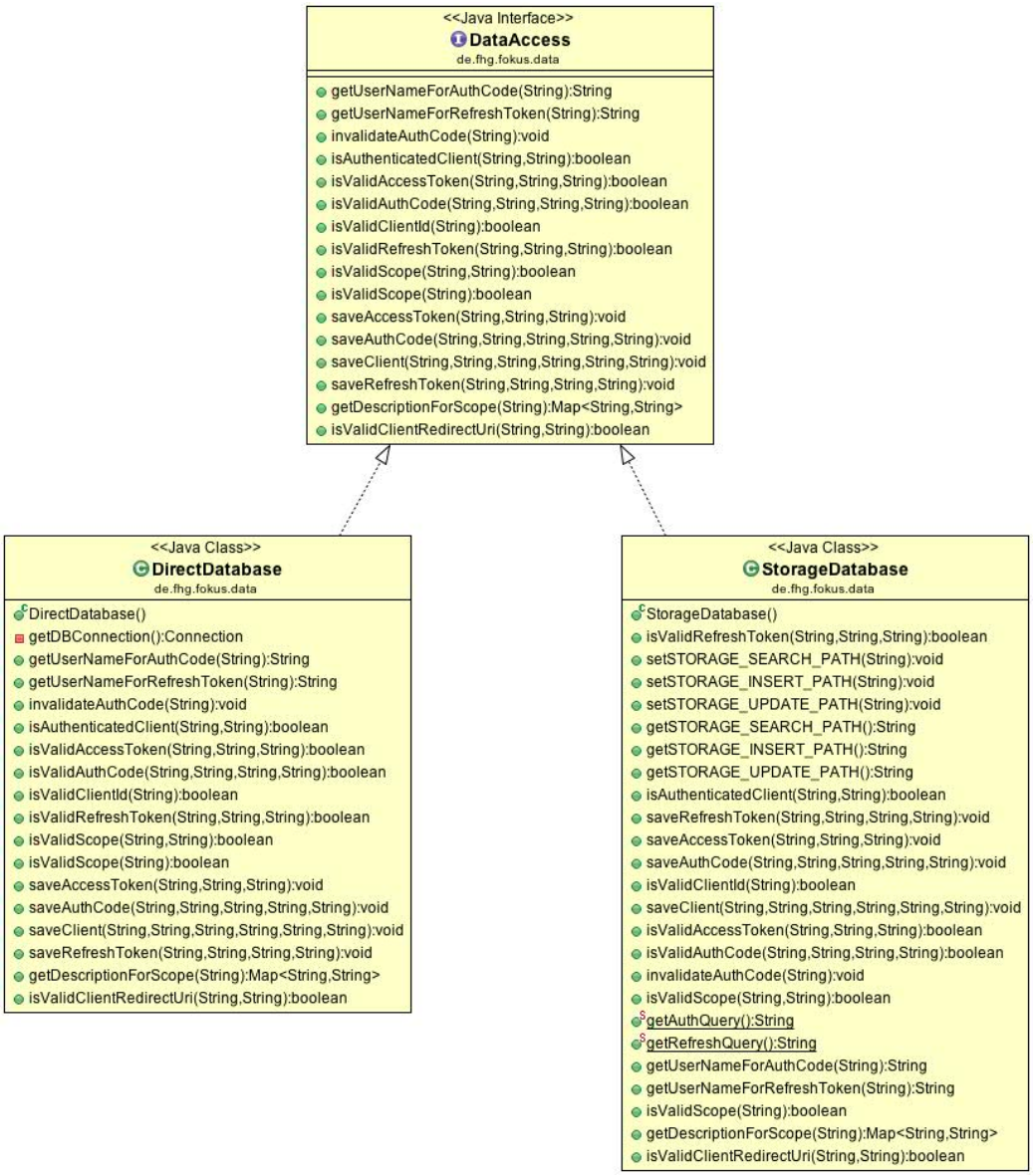


Abbildung 14: Klassendiagramm des Autorisierungsdatenbank Moduls

In Abbildung 14 ist das Klassendiagramm des Autorisierungsdatenbank-Moduls abgebildet. Dieses ist für die direkte Interaktion mit der Datenbank zuständig, welche Autorisierungsdaten enthält. Dabei ist erkennbar, dass es neben der definierten Schnittstelle *DataAccess* zwei Implementierungen dieser Schnittstelle gibt. Es ist zu beachten, dass von der darunter liegenden Datenbank abstrahiert wird und ausschließlich Methoden für den Zugriff definiert sind. Dies entkoppelt die Komponenten voneinander, sodass die darunter liegende Datenbank beliebig ausgetauscht werden kann. Dadurch ist die Entwicklung einer eigenen Implementierung leicht möglich. Durch die Implementierung der Schnittstelle ist die Funktionalität der integrierten Komponente gesichert.
Die Schnittstelle definiert verschiedene Methoden, welche im Kontext von OAuth benötigt werden. Beispielsweise müssen verschiedene Parameter validiert werden, was unter Anderem mit der Methode *isValidClientID()* erfolgen kann. Dabei definiert die Schnittstelle, *was* gemacht werden soll. In diesem Beispiel wäre die Aufgabe der Methode einen boolean bezüglich der Gültigkeit der eingegebenen Client ID als Antwort zurückzugeben. Die Schnittstelle definiert jedoch nicht *wie* dies gemacht werden soll.
Außerdem sind Methoden für das Speichern der einzelnen Teile (Clients, Autorisierungscode et cetera) der OAuth Spezifikation vorgesehen. Wie in Abbildung 14 erkennbar, gibt es zwei Beispielimplementierungen - *DirectDatabase* und *StorageDatabase*. Letztere verwendet die Komponente *Data Store* aus der Mobilitätsdaten-Cloud. Die Klasse *DirectDatabase* wurde zum einen als Proof of Concept von CDI und zum anderen mit der Intention der gesteigerten Performanz implementiert. Dabei wird direkt auf die Datenbank zugegriffen, wodurch die Anfragen der *StorageDatabase* an die REST Schnittstelle des *Data Stores* eingespart werden. Da zwei Implementierungen vorhanden sind, muss geklärt werden, wie die REST Services von der zu verwendenden Implementierung erfahren. Die Dienste wissen nur, dass ihnen ein *DataAccess* Objekt injiziert wird. In Listing 1 ist eine beispielhafte CDI Konfigurationsdatei (*beans.xml*) abgebildet, in welcher definiert ist, welche Implementierung der Schnittstelle *DataAccess* beim Deployment verwendet werden soll. Durch so eine XML Datei wird es sehr einfach, eine eigene Imple-

mentierung in die integrierte Komponente zur Sicherung von Cloud Services hinzuzufügen. Dadurch wird die Wartbarkeit der Komponente stark erhöht, da zum einen eine hohe Modularität gegeben ist und zum anderen eigene Implementierungen leicht hinzugefügt werden können.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
      <alternatives>
        <!--<class>de.fhg.fokus.data.StorageDatabase</class>-->
        <class>de.fhg.fokus.data.DirectDatabase</class>
    </alternatives>
</beans>
```

Listing 1: Beispiel einer CDI Konfigurationsdatei (beans.xml)

6.2.2 Benutzerdatenbank

In Abbildung 15 ist das Klassendiagramm des Benutzerdatenbank-Moduls visualisiert. Dieses ist ähnlich zum Modul der Autorisierungsdatenbank aufgebaut. Abermals wurde eine Schnittstelle erstellt (*UserDAO*), welche Methoden darüber definiert, *was* von der Implementierung erwartet wird. In dieser Schnittstelle müssen Benutzer registriert („saveUser(User)“ in Abbildung 15) und aus der Datenbank extrahiert werden können („getUser(String)“ in Abbildung 15). Ebenfalls beschreibt die Schnittstelle Methoden bezüglich der Überprüfung der Existenz eines Nutzers oder der Beschreibung eines validen Nutzers („isValidUser(String,String)“ und „doesUserExist(User)“ in Abbildung 15). Für die integrierte Komponente zur Sicherung von Cloud Services wurde nur eine Implementierung entwickelt (*UserDAODBImpl*). Anderenfalls würde analog zur Autorisierungsdatenbank in der CDI Konfigurationsdatei aus Listing 1 ein Eintrag für die entsprechende Implementierung vorhanden sein. Dadurch ist der Austausch der Implementierung sehr einfach möglich. Ebenfalls abgebildet ist die Klasse *User*, welche einen GeMo-Nutzer darstellt. Es ist anzumerken, dass die Schnittstelle auf diesem Nutzer basiert. Das be-

deutet, es wird davon ausgegangen, dass alle Attribute des Nutzers vorhanden sind. Dadurch wird die Integration einer neuen Art von Benutzer in die Komponente jedoch erschwert. Flexibilität würde man durch sehr erhöhten Implementierungsaufwand erhalten, weshalb darauf verzichtet wurde. In einer zukünftigen Entwicklungsphase soll dies jedoch berücksichtigt werden.

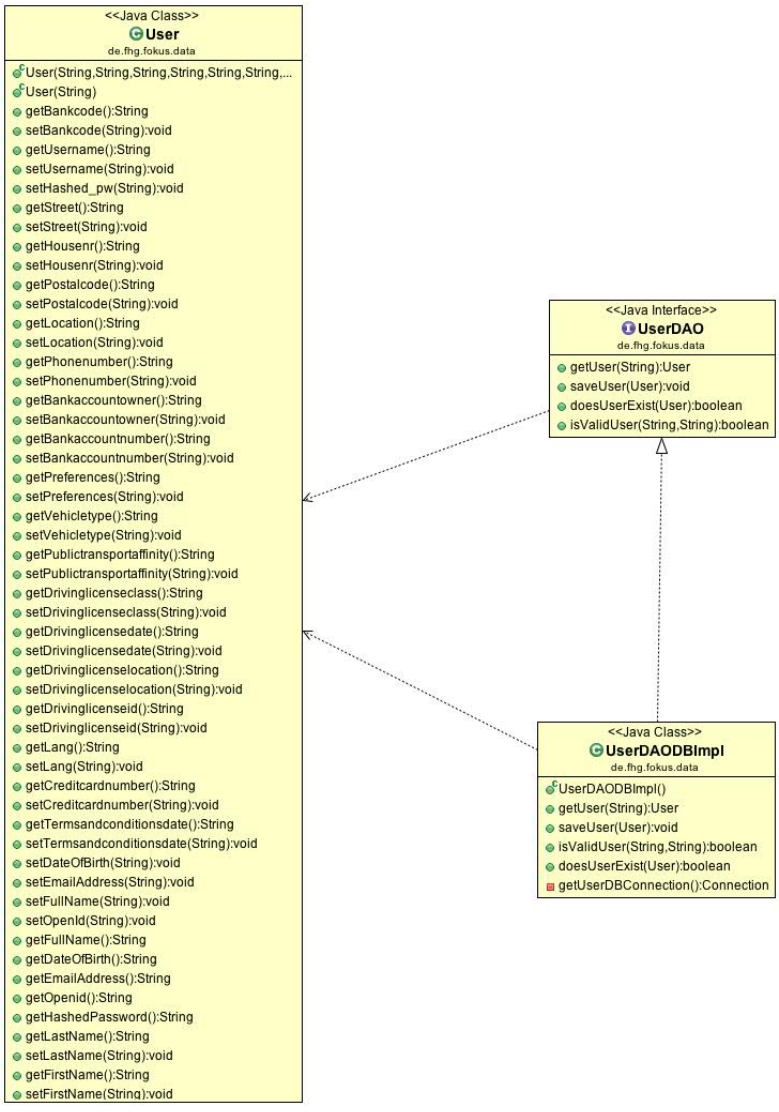


Abbildung 15: Klassendiagramm des Benutzerdatenbank Moduls

6.3 Umsetzung der OpenID-Subkomponente

Der folgende Abschnitt befasst sich mit der Umsetzung der OpenID-Subkomponente, welche die Authentifizierung eines Endnutzer innerhalb einer verteilten Infrastruktur ermöglicht.
In Abbildung 16 ist das Klassendiagramm der OpenID-Teilkomponente verdeutlicht. Die Komponente besteht aus den bereits erwähnten Modulen *Provider*, *Consumer*, *User* und *UserRegistration*. Dabei greift der *Provider* über definierte Schnittstellen (*DataAccess* und *UserDAO*) auf zwei verschiedene Datenbanken zu, während der *UserRegistration* Dienst über eine wohldefinierte Schnittstelle (*UserDAO*) eine Datenbank anfragt.

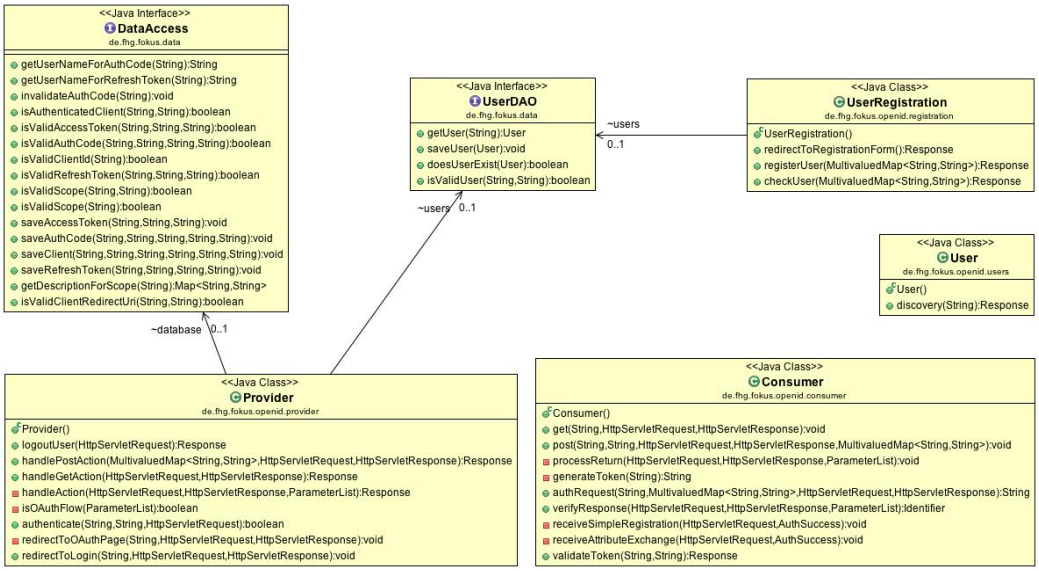


Abbildung 16: Klassendiagramm der OpenID Komponente

6.3.1 Consumer

Der *Consumer* ist ein REST Service, welcher in Abbildung 16 visualisiert ist. Dieser Dienst stellt nach außen die Methoden „get“ und „post“ zur Verfügung. Mit diesen Methoden sollen die entsprechenden HTTP-Anfragen bearbeitet werden.
Der *Consumer* übernimmt die Aufgabe der Initiierung des OpenID Protokollflusses. Dabei wird der Endnutzer auf eine statische Webseite geleitet,

auf welcher er seine OpenID eingeben kann. Daraufhin startete der *Consumer* den Protokollablauf in dem er die erhaltene URL normalisiert (siehe dazu auch Kapitel 3.3). Außerdem wird eine Assoziation mit dem entdeckten *Provider* generiert und eine Authentifizierung des Nutzers eingeleitet. Im weiteren Verlauf wird der Nutzer mit einer signierten Authentication Response an den *Consumer* zurück geleitet, welche vom diesem validiert wird.
Bei der Autorisierung eines Clients spielt der *Consumer* eine zentrale Rolle. Dieser wird zur Authentifizierung des Nutzers verwendet, damit die Autorisierung des Clients erfolgen kann. Bei der Autorisierung wird zusätzlich, bei der Rückkehr des Nutzers, eine Nonce[8] generiert, welche sicherstellt, dass diese Anfrage vom *Consumer* gesendet wurde.

6.3.2 Provider

Der *Provider* ist ebenfalls ein REST Service, welcher die Methoden „handleGetAction“ und „handlePostAction“ für die jeweilige HTTP Anfrage zur Verfügung stellt. Der Provider übernimmt die Authentifizierung des Endnutzers, indem dieser Zugriff auf die Nutzerdatenbank hat. Die Authentifizierung wird einerseits für die dezentrale Authentifizierung eines Nutzers auf der MDC verwendet und andererseits für die Authentifizierung eines Nutzers während der Autorisierungsanfrage eines Clients.

Dezentrale Authentifizierung

Die MDC basiert, wie bereits erwähnt, auf der Open Data Plattform, welche ein Datenportal beinhaltet. Auf diesem können sich potentielle Nutzer Daten anschauen und neue hinzufügen. Für das Hinzufügen neuer Datensätze muss der Endnutzer vorher authentifiziert werden. Dafür kann zusätzlich zur internen Authentifizierung und Datenhaltung der Nutzer, die dezentrale Authentifizierung per OpenID verwendet werden. Dafür gibt der Nutzer, welcher sich zuvor beim *UserRegistration* Service registriert hat, seine OpenID an und startet daraufhin das Protokoll. Im weiteren Verlauf wird der Nutzer an den Provider weitergeleitet, um ein Anmeldeformular (siehe Abbildung

[8]Eine Nonce bezeichnet im Kontext von Kryptographie, einen zufällig generierten und eindeutigen Sitzungsschlüssel, welcher nur einmal verwendet wird.

17) zu erhalten. Bei einer positiven Authentifizierung erstellt der *Provider* eine Authentication Response und sendet diese an die anfragende Relying Party zurück, welche die Authentication Response validiert und diesen Nutzer mit dieser OpenID assoziiert. Fortan kann sich der Nutzer immer über seine OpenID authentifizieren und das Datenportal der Open Data Plattform im vollen Umfang nutzen.

Authentifizierung während einer Autorisierungsanfrage

Wie im Abschnitt *Consumer* bereits erwähnt, wird OpenID während der Autorisierungsanfrage verwendet. Dabei kommt der entwickelte *Consumer* zum Einsatz, welcher den Nutzer an den *Provider* weiterleitet. Der Nutzer würde daraufhin ein anders Formular (siehe Abbildung 20 auf Seite 60) sehen. Abermals würde sich der Nutzer authentifizieren, während der *Provider* eine Authentication Response erstellt. Das OpenID Protokoll wäre somit für den *Provider* beendet.

Mit Hilfe des *Consumers* und des *Providers* kann die Anforderung 8 erfüllt werden, welche die Authentifizierung des Nutzers regelt. Darüber hinaus erfüllt der *Provider* die Anforderung 11 aus Kapitel 4, die sich auf die leichte Integrierbarkeit des Identitätsmanagements bezieht.

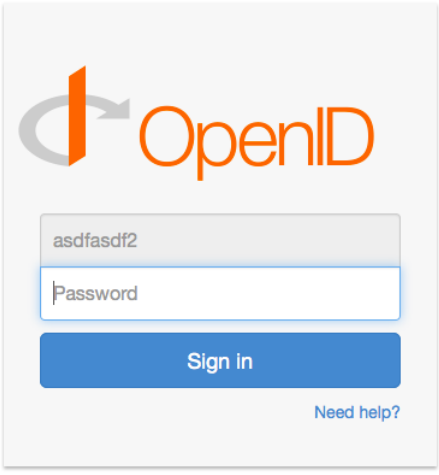


Abbildung 17: Screenshot des Login Formulars

6.3.3 UserRegistration

Bei dem Modul *UserRegistration* handelt es sich um einen REST Service, welcher die Aufgabe der Registrierung von Nutzern übernimmt. Dabei stellt dieser Dienst über verschiedene Pfade die Methoden „register“, „redirectToRegistrationForm“ und „checkUser“ zur Verfügung (siehe Abbildung 16). Es existieren zwei Möglichkeiten der Registrierung eines Benutzers: Zum einen direkt per API über die „register“ Methode oder zum anderen mit Hilfe eines Webformulars über die „redirectToRegistrationForm“ Schnittstelle. Im Folgenden wird von der Registrierung mit Hilfe des Formular ausgegangen.

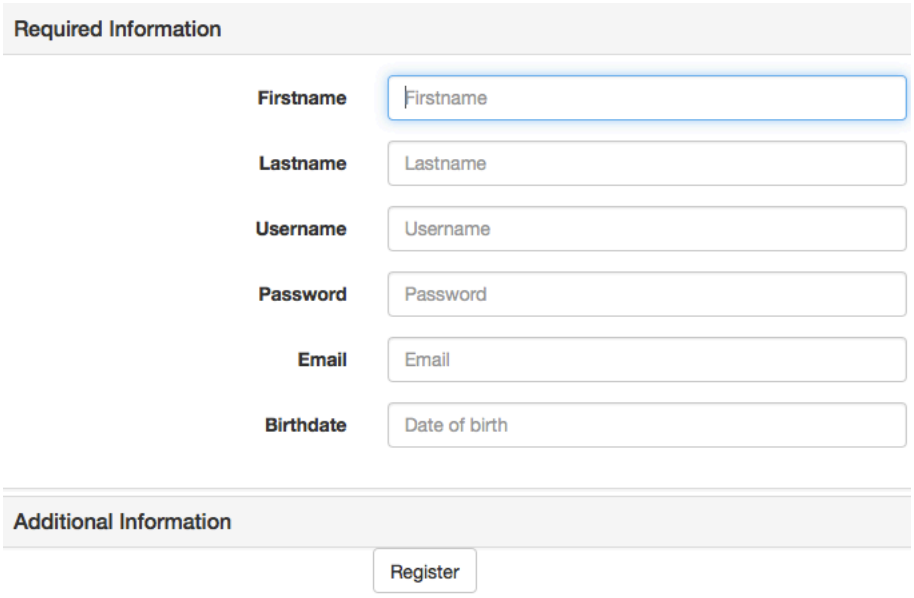


Abbildung 18: Screenshot des Formulars für die Benutzerregistrierung

Der Nutzer erhält das Formular aus Abbildung 18. Während der Eingabe eines Benutzernamens wird dieser validiert und dahingehend überprüft, ob er schon vergeben ist. Dafür wird die Methode „checkUser“ aufgerufen. Beim Absenden des Formulars werden die Parameter per HTTP-POST an die Methode „register“ gesendet, welche daraufhin die Registrierung des Nutzers übernimmt. Die erforderlichen Parameter sind in Abbildung 12 auf Seite 44

dargestellt. Es handelt sich um einen eindeutigen Benutzernamen und Passwort, Vor- und Nachname des Nutzers sowie eine E-Mail Adresse und das Geburtsdatum.
Wiederholt wird überprüft, ob der gewünschte Nutzername gültig und noch nicht vergeben ist, da diese Methode, wie bereits erwähnt, direkt angesprochen werden kann. Wenn die Prüfung auf Gültigkeit erfolgreich ist, wird der Nutzer registriert und erhält eine positive Nachricht als Antwort. Anderenfalls würde der Nutzer eine Fehlermeldung erhalten und müsste die Registrierung erneut starten.
Mit diesem Modul werden die Anforderungen 4, 5 und 6 erfüllt, welche sich auf die Registrierung eines Endnutzers beziehen.

6.3.4 User

Beim Dienst *User* handelt es sich um einen simplen REST Service. Dieser Dienst stellt über einen variablen URL-Pfad die Methode „discovery“ zur Verfügung, wobei der Pfad dem Benutzernamen eines Nutzers entspricht. Bei einer HTTP-GET Anfrage an diesen Dienst, antwortet er immer mit einem XRDS Dokument, in welchem Informationen über den *Provider*, wie beispielsweise die unterstützte OpenID Version oder die URL des *Providers*, gesichert sind.

6.4 Umsetzung der OAuth-Subkomponente

Der folgende Abschnitt beschäftigt sich mit der Implementierung der OAuth-Subkomponente von ISCS und den dazugehörigen Endpoints. Die OAuth-Teilkomponente ist für die Autorisierung und Authentifizierung von Clients (Services und Applikationen) zuständig.
In Abbildung 19 ist das Klassendiagramm der OAuth-Komponente erkennbar. Es wird deutlich, dass wiederholt die vier Module *AuthzEndpoint*, *TokenEndpoint*, *ClientRegistrationEndpoint* und *RedirectEndpoint* existieren. Dabei greifen drei der vier Module über eine definierte Schnittstelle (*DataAccess*) auf die Datenbank zu, während der *RedirectEndpoint* keinen Zugriff benötigt. Im Folgenden werden die einzelnen Module genauer erläutert.

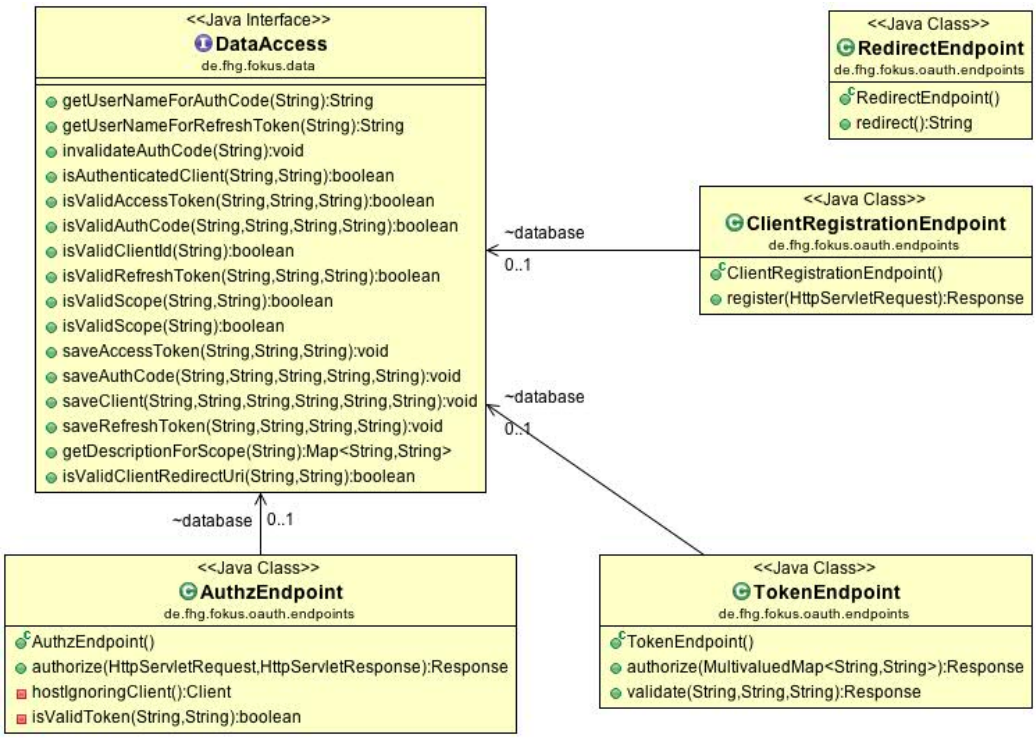


Abbildung 19: Klassendiagramm der OAuth-Komponente

6.4.1 AuthzEndpoint

Der *AuthzEndpoint* ist ein REST Service, welcher in Abbildung 19 zu sehen ist. Er stellt die Methode „authorize“ über einen URL-Pfad nach außen zur Verfügung.
Die Aufgabe des *AuthzEndpoint* ist die Annahme und entsprechende Bearbeitung von Autorisierungsanfragen der Clients. Dabei senden die Clients eine HTTP-GET Anfrage mit folgenden Parametern, welche im *HTTPServletRequest* in Abbildung 19 enthalten sind:

- client_id
  - die ID, welche der Client bei der Registrierung erhalten hat,
- redirect_uri
  - die Redirect URI, welche der Client bei der Registrierung mit angegeben hat und wohin der Nutzer weitergeleitet wird,

- response_type
  - der gewünschte Antworttyp,
  - dieser Wert ist auf „code“ gesetzt, was bedeutet, dass ein Autorisierungscode angefragt wird,
- scope
  - der Scope, welcher der Client vom Nutzer autorisiert haben will,
  - es handelt sich dabei um einen durch Leerzeichen getrennten String,
  - diese Werte sind vorher definiert worden,
- state
  - ein beliebiger Wert, welcher zwischen den Anfragen aufrecht gehalten werden muss,
  - diese Angabe ist optional,
  - dieser Wert soll Cross-Site-Request-Forgery Angriffe verhindern.

Im nächsten Schritt validiert der *AuthzEndpoint* die erhaltenen Parameter. Dabei wird insbesondere darauf geachtet, dass die Parameter gültige Werte enthalten, welche vorab in der Spezifikation [28] definiert wurden. Zudem wird überprüft, ob die Client ID gültig ist und ob die Redirect URI zu dieser Client ID gehört. Dadurch soll sichergestellt werden, dass nur bereits registrierte Clients Autorisierungsanfragen stellen können.
Daran anknüpfend startet der Endpunkt die Autorisierung des Clients. Da der Client vom Nutzer[9] autorisiert wird, muss der Nutzer vorab mit Hilfe von OpenID authentifiziert werden. Dies wird umgesetzt, indem der Nutzer an den *Consumer* weitergeleitet wird. Bei diesem Dienst gibt der Nutzer, wie bereits erwähnt, seine OpenID ein und startet damit das OpenID Protokoll. Im weiteren Verlauf erhält der Nutzer ein Formular, in welchem die ursprünglich gesendeten Parameter des Clients noch einmal visualisiert

[9]Der Nutzer bezeichnet einen Endnutzer einer (mobilen) Applikation beziehungsweise eines Diensts, welcher über OAuth autorisiert wird.

werden (siehe Abbildung 20). Außerdem ist erkennbar, dass der Nutzer die Möglichkeit hat, die Anfrage zu akzeptieren oder abzulehnen. Im positiven Fall würde der *Provider* den Endnutzer zuerst an den *Consumer* weiterleiten, welcher den OpenID Protokollfluss mit einer Validierung der erhaltenen Informationen beendet. Anschließend würde der *Consumer* den Benutzer an den *AuthzEndpoint* weiterleiten, wobei der *Consumer* eine generierte Nonce mitsendet. Somit kann der *AuthzEndpoint* sicherstellen, dass diese Nachricht vom *Consumer* ist. Infolgedessen wird zwar die Kopplung der Komponenten erhöht, jedoch hat dies auch eine Steigerung der Sicherheit zur Folge, weshalb eine Abschätzung zu Gunsten der Sicherheit stattfand. Abschließend validiert der *AuthzEndpoint* die Nonce. Daraufhin würde er, mit dem Wissen einer gültigen Autorisierung, einen Autorisierungscode für den Client ausstellen, welchen er an den Client JSON encodiert zurücksenden kann. Im negativen Fall würde der Client eine JSON encodierte Fehlermeldung erhalten, welche dem Client mitteilt, dass der Nutzer die Anfrage abgelehnt hat. Mit diesem Modul wird die Anforderung 7 aus Kapitel 4 realisiert, die sich auf die Autorisierung eines Clients bezieht.

6.4.2 TokenEndpoint

Bei dem *TokenEndpoint* handelt es sich um einen REST Service, welcher in Abbildung 19 abgebildet ist. Dieser stellt über einen URL-Pfad die Methode „authorize“ nach außen zur Verfügung.
Der *TokenEndpoint* übernimmt sowohl die Aufgabe des Eintausches eines Autorisierungscodes gegen ein Access und Refresh Token, die Erneuerung eines Access Tokens als auch die Validierung der Parameter, die an die zugriffsgeschützten Dienste gesendet werden.

Anfrage mit Autorisierungscode

Bei dieser Anfrage sendet der Client eine HTTP-POST Anfrage mit den folgenden Parametern:

- client_id
  - die ID, welche der Client bei der Registrierung erhalten hat,

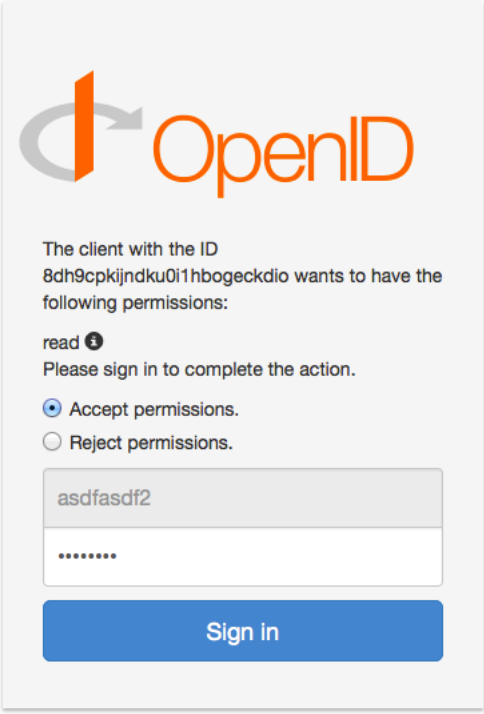


Abbildung 20: Screenshot des Formulars

- client_secret
  - ein zufällig generierter String, welchen der Client bei der Registrierung erhalten hat,
  - zur Authentifizierung des Client,
- redirect_uri
  - die Redirect URI, welche der Client bei der Registrierung mit angegeben hat und wohin der Nutzer weitergeleitet wird,
- grant_type
  - der gewünschte Gewährungstyp,
  - dieser Wert ist auf „authorization_code“ gesetzt, was bedeutet, dass der Client bereits einen Autorisierungscode erhalten hat und verwenden will,

- code
  - der Autorisierungscode, welcher dem Client im vorherigen Schritt ausgestellt wurde,
  - dieser Code ist einmalig gültig,
- scope
  - der Scope, welcher dem Client vom Nutzer autorisiert wurde,
  - dabei kann es sich auch um eine Teilmenge vom vorher autorisierten Scope handeln.

Die erhaltenen Parameter werden vom *TokenEndpoint* validiert. Dabei wird, analog zum *AuthzEndpoint*, darauf geachtet, dass die Werte der Parameter gemäß Spezifikation [28] aufgebaut sind. Darüber hinaus wird überprüft, ob die Client ID gültig ist, ob die Redirect URI zu dieser ID gehört und ob der Client mit Hilfe des Client Secrets authentifiziert ist. Dadurch wird sichergestellt, dass nur registrierte und authentifizierte Clients Token erhalten können.
Im Fall einer positiven Validierung der Parameter wird überprüft, ob der Autorisierungscode an den anfragenden Client ausgestellt und noch nicht eingelöst wurde. Wenn dies zutrifft, wird daraufhin der Autorisierungscode für ungültig erklärt und ein Refresh Token sowie ein Access Token werden generiert. Diese Token werden mit Scope und Benutzernamen des Endnutzers, welcher den Client autorisiert hat, an den Client JSON encodiert zurückgesendet. Mit dieser Teilaufgabe des Moduls wird die Anforderung 9 erfüllt, welche die Authentifizierung des Clients während der Autorisierung regelt.

Erneuerung eines Access Tokens
Diese Aufgabe verläuft ähnlich wie die vorherige, wobei eine HTTP-Anfrage an den *TokenEndpoint* der OAuth-Subkomponente gestellt wird. Folgende Parameter werden dabei gesendet:

- client_id

  - die ID, welche der Client bei der Registrierung erhalten hat,

- client_secret
  - ein zufällig generierter String, welchen der Client bei der Registrierung erhalten hat,
  - zur Authentifizierung des Clients,

- redirect_uri
  - die Redirect URI, welche der Client bei der Registrierung mit gesendet hat und wohin der Nutzer weitergeleitet wird,

- refresh_token
  - das Refresh Token, welches beim Eintausch eines Autorisierungscodes erhalten wurde,
  - dieses Token hat eine variable Gültigkeitsdauer, wobei es in dieser Masterarbeit für 6 Monate gültig ist,

- grant_type
  - der gewünschte Gewährungstyp,
  - dieser Wert ist auf „refresh_token“ gesetzt,

- scope
  - der Scope, welcher dem Client vom Nutzer autorisiert wurde,
  - dabei kann es sich auch um eine Teilmenge vom vorher autorisierten Scope handeln.

Der *TokenEndpoint* validiert erneut die Eingabewerte und überprüft zum einen, ob dieses Refresh Token an den anfragenden Client ausgestellt wurde und zum anderen, ob das Token gültig ist. Wenn das Resultat der Überprüfung positiv ist, generiert der Endpunkt ein neues Access Token, welches er an den Client mit Scope (beispielsweise „read write“) und Benutzernamen

des Endnutzers, welcher den Client autorisiert hat, JSON encodiert zurücksendet.

Validierung der Parameter

Wie aus den vorherigen Aufgaben dieses Moduls erkennbar, erhält der Client am Ende ein Token, welches mit einem bestimmten Scope und Benutzernamen assoziiert ist. Mit diesen Informationen kann der Client die zugriffsgeschützten Dienste anfragen. Dabei werden diese Parameter als Header in der eigentlichen Anfrage an den Dienst mitgesendet. Bevor der Dienst die Anfrage des Clients bearbeitet, wird überprüft, ob die Parameter gültig sind. Dafür stellt der Dienst eine Anfrage an den *TokenEndpoint*. In dieser Anfrage sind die Header Parameter enthalten, welche vom Endpunkt extrahiert und validiert werden. Es wird überprüft, ob das Access Token noch gültig ist und zu diesem Benutzer und Scope passt. Mit dieser Aufgabe wird die Anforderung 10 aus Kapitel 4 realisiert. Diese Anforderung stellt die Validierung der Parameter sicher.

6.4.3 ClientRegistrationEndpoint

In Abbildung 19 ist unter Anderem auch das Modul *ClientRegistrationEndpoint* dargestellt. Es handelt sich dabei um einen REST Service, welcher über einen URL-Pfad die Methode „register“ nach außen verfügbar macht.
Wie bereits erwähnt, übernimmt dieses Modul die Aufgabe der Clientregistrierung. Ein Client kann sich per API mittels HTTP-POST Anfrage registrieren. Dabei sendet der Client die Parameter JSON encodiert, sodass diese vom REST Service verarbeitet werden können. Die erforderlichen Parameter sind in Abbildung 11 auf Seite 43 dargestellt. Es handelt bei diesen um einen eindeutigen Clientnamen sowie eine Redirect URI (diese Informationen identifizieren einen Client), eine Beschreibung, über die Tätigkeit des Clients und eine Client URL, unter welcher weitere Informationen über den Client liegen.
Bei einer gültigen Anfrage (HTTP-POST mit JSON encodierten Parametern), validiert der ClientRegistrationEndpoint die Parameter. Dabei wird

sichergestellt, dass dieser Client noch nicht registriert ist und die Redirect URI sowie die Client URL gültig sind. Wenn die Parameter noch nicht in der Datenbank vorhanden sind, wird daraufhin eine eindeutige Client ID sowie ein Client Secret generiert. Außerdem wird daraufhin gespeichert, wann diese Informationen ausgestellt („issued_at“) wurden und bis wann diese gültig sind. Aktuell verfällt keine Registrierung eines Clients, da dies unnötig im Kontext des Projekts GeMo ist. Daher ist der Wert von „expires_in“ immer auf -1 gesetzt. Gleichzeitig werden sowohl die Parameter der Anfrage als auch die zuvor generierten Werte in einer Datenbank persistiert. Als Antwort auf seine Frage erhält der Client die generierten Daten wie die Client ID, das Client Secret, das Ausstellungsdatum und das Ablaufdatum JSON encodiert. Mit diesem Schritt ist die Registrierung eines Clients abgeschlossen.
Mit diesem Modul sind die Anforderungen 1, 2 und 3 aus Kapitel 4 erfüllt, welche die Registrierung eines Clients regeln.

6.4.4 RedirectEndpoint

Der *RedirectEndpoint* ist ein zusätzlicher REST Service, der in Abbildung 19 dargestellt ist. Der Dienst stellt über einen URL-Pfad die Methode „redirect“ zur Verfügung. Die einzige Anforderung an den Dienst besteht in der Weiterleitung der erhaltenen Informationen. Dabei werden die Header und Parameter der HTTP-GET Anfrage durchiteriert und in ein neues JSON Objekt hinzugefügt. Dieses wird an den Anfragenden weitergeleitet. Dadurch wird sichergestellt, dass Clients, die über keinen eigenen *RedirectEndpoint* verfügen, dennoch die angefragten Informationen erhalten.

7 Evaluation

In diesem Kapitel wird die *Integrierte Komponente zur Sicherung von Cloud Services (ISCS)* evaluiert. Dabei wird insbesondere auf das dynamische Testen der entwickelten Lösung mit Hilfe von Unit Tests, Integrationstests und Fuzz Tests eingegangen. Abschließend wird mit verschiedenen Messungen die Performanz der Komponente untersucht und visualisiert.
Durch dynamisches Testen kann das Verhalten des Systems untersucht werden. Die Software wird dabei ausgeführt, weil Eingabedaten an das System gesendet werden. Anschließend wird die Ausgabe des Systems auf Korrektheit überprüft. Unit Tests testen einzelne Subkomponenten isoliert auf deren Funktionalität, während diese im Gegensatz dazu bei Integrationstests gruppiert und gemeinsam überprüft werden. Fuzz Testing beschreibt eine Möglichkeit des Sicherheitstestens, wobei invalide Daten an das System gesendet werden und unter anderem die fehlerfreie Ausführung des Systems untersucht wird.
Dabei wurden die Unit Tests sowie die Integrationstests nach dem Prinzip aufgebaut, dass sich der Standard und die Sequenzdiagramme aus dem Kapitel dynamische Aspekte (Kapitel 5.2.2) angeschaut wurden und auf dessen Basis die Tests sowie die Eingabedaten erstellt wurden.

7.1 Anforderungen

Bezugnehmend auf die Anforderungsanalyse aus dem vierten Kapitel soll die Komponente stabil laufen (Anforderung 12), performant sein (Anforderung 13) und einen geringen Ressourcenbedarf aufweisen (Anforderung 14). Darüber hinaus soll die Lösung hinreichend sicher sein (Anforderung 17). Wie bereits erwähnt, ist dabei zu beachten, dass es keine hundertprozentige Sicherheit geben kann und daher nur von hinreichender Sicherheit gesprochen werden darf, was bedeutet, dass die Sicherheitsaspekte der Komponente zwar ausführlich überprüft wurden, jedoch immer noch unentdeckte Fehler im Code vorhanden sein könnten.
Die oben genannten Anforderungen sollen mit Hilfe von Testfällen überprüft werden. Beim Testen muss davon ausgegangen werden, dass nicht jeder Feh-

ler im Programm gefunden werden kann. Dies ist zu berücksichtigen, weil theoretisch nicht alle Ausführungspfade des Programms geprüft werden können. Aus diesem Grund und weil auch Fehler bei der Integration entstehen können, werden zusätzlich zu Unit Tests auch Integrations- und Fuzz Tests durchgeführt, welche solche Fehler identifizieren sollen. Bei den Integrationstests wird der Ansatz des *Bottom Up Testing* verfolgt. Dabei werden zuerst die unteren und darauf aufbauend die höheren Ebenen der Architektur getestet. Dieser Ansatz erleichtert das Aufdecken von Fehlern im Code, weshalb die Anforderungen an Stabilität und Sicherheit besser erfüllt werden können.

7.2 Unit Tests

Mit Hilfe von Unit Tests sollen einzelne Komponenten isoliert getestet werden, wodurch deren Funktionalität überprüft wird. Unit Tests weisen eine strikte Anforderung auf, welche vom System erfüllt werden muss. Mit Hilfe dieser Tests wird das im Entwicklungszyklus frühe Finden von Fehlern möglich. Zudem kann der entwickelten Code später korrigiert und anschließend mit Hilfe der Tests erneut hinsichtlich des Erhalts der Funktionalität überprüft werden. Ein weiterer Vorteil von Unit Tests ist die dadurch vereinfachte Integration. Die Ungewissheit über einzelne Einheiten kann durch Unit Tests verringert werden, weil die Testfälle genau spezifizieren, welche Aufgaben die Einheiten erfüllen sollen.
Eine Vielzahl von Testfällen wurde beispielsweise für die Überprüfung von validen Eingaben, insbesondere bei den in der Spezifikation von OAuth beschriebenen Parametern, geschrieben. Dabei wurde die Syntax der OAuth Eingaben, welche in der angereicherten Backus-Naur-Form (ABNF) notiert ist, mit Hilfe von regulären Ausdrucken repräsentiert. Die Tests senden sowohl valide als auch invalide Eingaben an die Einheiten und überprüfen, ob der jeweilige reguläre Ausdruck korrekt ist und somit die Eingaben entsprechend der Spezifikation validiert werden. Weitere Unit Tests wurden für die einzelnen Methoden der Datenbank erstellt, welche sicherstellen, dass die jeweilige Funktionalität korrekt realisiert ist. Dies ist wichtig, da somit die korrekte Speicherung der Daten sowie deren Extraktion aus der Datenbank

gesichert ist.

7.3 Integrationstests

Integrationstests sind eine Reihe von Einzeltests, welche verschiedene, voneinander abhängige Komponenten in Verbindung bringen, um ihre Funktionalität als gesamte Einheit zu testen. Die einzelnen Komponenten können bereits vorab durch Unit Tests überprüft werden.
Ein typischer Testfall bezieht sich auf einen der Services, beispielsweise den *ClientRegistrationEndpoint*, und kombiniert diesen mit der Datenbank. Dadurch können komplexere Tests generiert werden. In dem Beispiel der Registrierung eines Clients wird eine Anfrage an den Service generiert, welcher daraufhin die Parameter extrahiert und validiert. Abschließend speichert dieser die Informationen in der Datenbank. Mit diesem Test kann festgestellt werden, dass die gewünschte Funktionalität auch nach der Integration vorhanden ist.
Ein weiterer Testfall ist die Überprüfung des gesamten Ablaufs des OAuth Protokolls. Dieser Test wird erläutert, da er die gesamten Einheiten umspannt und somit bei der erfolgreichen Durchführung ein Indiz für das korrekte Verhalten der ISCS ist. Für diesen Test wird ein Client registriert, welcher die Autorisierungsanfrage startet und die erforderlichen Parameter mitsendet. Während der Authentifizierung wird die Eingabe des Nutzers vom System übernommen, sodass ein automatisiertes Ausführen ermöglicht wird. Daraufhin erhält der Client einen Autorisierungscode, welcher im Testfall überprüft wird (beispielsweise ob er der Spezifikation entsprechend aufgebaut ist). Im zweiten Schritt des Testfalls versucht der Client den Autorisierungscode gegen die Token (Access Token und Refresh Token, siehe Kapitel 3.6) einzutauschen. Nachdem der Client die Token erhalten hat, wird abermals überprüft, ob das System korrekt bezüglich der Spezifikation arbeitet. Mit Hilfe dieses Integrationstests ist der korrekte Ablauf des OAuth Protokolls sowie die Authentifizierung mit OpenID abgesichert.

7.4 Fuzzing

Fuzz Testing oder Fuzzing bezeichnet einen Ansatz beim Sicherheitstesten, bei welchem invalide Eingaben - im Fall von Data Fuzzing handelt es sich dabei um Daten und im Rahmen von Behavioral Fuzzing um Interaktionen - an das *System Under Test (SUT)* gesendet werden. Dabei sollen Fehler in der Implementierung des SUT gefunden werden, welche dazu führen, dass die Eingaben verarbeitet und nicht abgelehnt werden oder das System in einen instabilen Zustand gerät [41, 42]. Dadurch können Schwachstellen in der Implementierung identifiziert werden, welche dafür genutzt werden könnten, um das SUT zum Absturz zu bringen oder das Verhalten des Systems in einer unbeabsichtigten Weise zu verändern. Da das SUT mit invaliden Eingaben angefragt wird, ist Fuzzing eine Möglichkeit zum Testen der Robustheit des Systems [41].
Beim Data Fuzzing ist zu beachten, dass durch vollständig zufällig generierte Eingabedaten zwar immer noch Fehler aufgedeckt werden können, jedoch dadurch auch Nachteile auftreten. Einerseits kann bei vollständiger Randomisierung der Eingabedaten davon ausgegangen werden, dass sie ungültig sind. Folglich können solche invaliden Daten leicht vom System entdeckt und abgewiesen werden. Dieser Vorgang kann auch funktionieren, obwohl ein Teil der Validierung fehlschlägt, sofern andere invalide Daten gefunden werden - zum Beispiel könnten Teile der Eingabe bei einer ersten Validierung fälschlicherweise als korrekt eingeschätzt werden, wobei bei einer weiteren Überprüfung die gesamte Eingabe als invalide identifiziert werden könnte. Daher ist bei diesem Ansatz die Wahrscheinlichkeit Fehler zu finden eher gering [41].
Ein weiterer Nachteil entsteht durch nicht vorhandenes Wissen über das Protokoll. Es ist schwierig ohne Wissen über das Protokolls Fehler zu finden, welche in komplexen Interaktionen mit dem SUT versteckt sind. Aus diesem Grund integrieren bessere Fuzzer Wissen über das zu verwendende Protokoll. Dadurch werden nur teilweise invalide Daten generiert. Diese Eingabedaten werden als semi-valid bezeichnet, da die meisten Teile gültig sind, während nur ein geringer Teil ungültig ist.
Im weiteren Verlauf wird auf die zwei Teilbereiche des Fuzzings eingegangen,

mit welchen die integrierte Komponente untersucht wurde.

7.4.1 Data Fuzzing

Beim Data Fuzzing werden, wie bereits vorgestellt, zufällig generierte Werte an das SUT gesendet. Im Rahmen dieser Masterarbeit wurde die von Fraunhofer FOKUS entwickelte Bibliothek Fuzzino zur Generierung randomisierter Werte verwendet. Der Vorteil der Verwendung von Fuzzino [41, 42, 43] besteht sowohl in der guten Bedienbarkeit der Bibliothek als auch in deren leichter Integrierbarkeit. Sie ist in Java geschrieben und kann ohne Umstände in bestehende Projekte als Bibliothek hinzugefügt werden. Darüber hinaus wird angegeben, welche Art von Werten sich generieren lassen. Eine Auswahl dessen sind:

- BadLongStrings: Dieser Generator erstellt lange Strings mit bis zu 20000 Zeichen. Zusätzlich beinhalten diese Strings besondere Zeichen, wie beispielsweise das NULL Byte.
- BadStrings: Dieser Generator erzeugt Strings, welche eine besondere Bedeutung bei spezifischen Betriebssystemen haben (beispielsweise „COM1:“).
- LongStrings: Wie der Name erahnen lässt, werden mit diesem Generator beliebig lange Strings erzeugt, mit welchen nach Schwachstellen, wie beispielsweise Buffer Overflow, gesucht werden kann.
- AllBadStrings: Dies ist eine Kombination aus den drei zuvor genannten Generatoren.
- BadHostnames: Dieser Generator erstellt ungültige Netzwerk-Computernamen, beispielsweise zu lange Hostnames.
- FormatStrings: Dieser Generator erzeugt Strings, mit welchen es unter Umständen möglich ist, Schwachstellen bei der Formatierung von Strings aufzudecken.

- SQLInjections: Der Generator *SQLInjections* kreiert Strings, mit welchen es unter Umständen möglich wird, Schwachstellen im Code bezüglich SQL Injection aufzuzeigen.

Zusätzlich zu dieser Bibliothek wurde *JUnit 4* verwendet. *JUnit* hat dabei in der Version 4 ein neues Feature namens *Parameterized Tests* bekommen. Mit Hilfe von *Parameterized Tests* kann der gleiche Testfall mehrmals mit jeweils verschiedenen Eingaben hintereinander durchgeführt werden. Die Kombination aus der Verwendung von *Fuzzino* und *JUnit 4* führt zu übersichtlichen und starken Fuzz Tests. Dabei wurde auf zuvor geschriebene Integrationstests zurückgegriffen. Diese wurden dahingehend überarbeitet, dass sie ihre Werte über die Schnittstelle von JUnit 4 empfangen. Die Werte wurden mit Hilfe von Fuzzino generiert und über die Schnittstelle dem jeweiligen Testfall übergeben.
Durch die Ausführung dieser Testfälle wurden keine Schwachstellen bezüglich der Formatierung oder SQLInjection ermittelt. Außerdem hat das System stets wie erwartet reagiert. Folglich kann geschlussfolgert werden, dass durch diese Testfälle die Anforderungen 12 und 17 aus dem vierten Kapitel untersucht und erfüllt wurden. Diese geben an, dass das System stabil (Anfoderung 12) und sicher (Anforderung 17) funktionieren soll.

7.4.2 Behavioral Fuzzing

Im vorherigen Abschnitt wurde bereits erwähnt, dass Fuzzer, welche vollständig randomisierte Daten als Eingabe verwenden, weniger mächtig sind als jene, welche über Protokollwissen verfügen. Beim Behavioral Fuzzing wird daher auf modellbasierte Fuzzer gesetzt, da diese durch das Modell des SUT immer Protokollwissen besitzen. Aus diesem Grund können diese komplexere Fehler finden, da verzweigtere Interaktionen mit dem SUT vollzogen werden können [41].
Im Gegensatz zum Data Fuzzing, bei welchem die Generierung ungültiger Eingabedaten stattfindet, werden beim Behavioral Fuzzing ungültige Nachrichtensequenzen generiert [41]. Da die Implementierung des SUT nicht bekannt sein muss, handelt es sich beim Behavorial Fuzzing um einen Ansatz

des Black-box Testens [42]. Mit Hilfe des Behavorial Fuzzings sollen Schwachstellen in der Software aufgedeckt werden, welche nicht nur auf ungültigen Eingabedaten basieren, sondern auch auf ungültigen Nachrichtensequenzen, welche akzeptiert und verarbeitet werden oder das System zum Absturz bringen können.
Beim Behavioral Fuzzing wird an dieser Stelle von einem gültigen Testfall ausgegangen, welcher als UML Sequenzdiagramm modelliert ist, aus welchem im weiteren Verlauf durch Fuzzing viele verschiedene Sequenzdiagramme generiert werden. Aus diesen werden anschließend neue Testfälle generiert. Es können verschiedene Methoden des Behavioral Fuzzings angewendet werden, um neue Nachrichtensequenzen zu generieren: beispielsweise durch das Verändern einzelner Nachrichten, dem Ändern der Reihenfolge von Nachrichten sowie das Ändern von Combined Fragments (zum Beispiel „alternative“ oder „loop“). Im Folgenden werden Methoden erläutert, welche bei der Evaluation der ISCS angewendet wurden.
Ungültige Nachrichtensequenzen können durch die Veränderung individueller Nachrichten generiert werden. Dabei kann eine einzelne Nachricht

- wiederholt werden, sodass sie zweimal existiert,
- entfernt werden, sodass diese im neuen Modell nicht mehr vorhanden ist,
- an eine anderen Stelle in der Sequenz verschoben werden [42].

Es gibt zwei Möglichkeiten zwei oder mehr Nachrichten zu fuzzen: Zum einen werden diese Nachrichten miteinander vertauscht. Dadurch entsteht eine invalide Sequenz. Zum anderen können mehr als zwei Nachrichten ausgewählt und zufällig permutiert werden. Aufgrund der Zufälligkeit ist dieser Ansatz weniger mächtig als der zuvor genannte [42].
Im weiteren Verlauf könnten Combined Fragments verändert werden. Dabei müsste unterschieden werden, ob das Fragment als Ganzes betrachtet wird oder ob die einzelnen Interaktionsoperanden miteinbezogen werden. Wird das Combined Fragment als Ganzes betrachtet, kann es, analog zu den Nachrichten, entfernt, wiederholt oder hinzugefügt werden [42]. Betrachtet man

dahingegen die einzelnen Interaktionsoperanden können zusätzliche Veränderungen durchgeführt werden. Diese könnten beispielsweise insofern miteinander kombiniert werden, als dass der erste Interaktionsoperand an den zweiten angefügt wird. Dadurch würde eine ungültige Sequenz entstehen [42]. Diese Methode konnte in dieser Arbeit jedoch nicht angewendet werden, weil die dadurch gefuzzten Modelle (Testfälle aus UML Sequenzdiagrammen) weiterhin gültig wären.

In den Abbildung 21, 22 und 23 ist das verwendete Modell für das Beva-

Abbildung 21: Modell für das Behavioral Fuzzing (1)

hioral Fuzzing abgebildet. Dabei ist ein typischer Protokollablauf modelliert worden. Es ist erkennbar, dass eine Clientregistrierung stattfindet, der Client autorisiert wird und daraufhin mit dem Autorisierungscode nach Token anfragt (Abbildung 21). Mit diesen Token werden beispielhafte Serviceaufrufe realisiert (Abbildung 22). Darüber hinaus wird das Access Token mit Hilfe eines Refresh Tokens erneuert (Abbildung 23).
Basierend auf diesem Modell wurden 46 einzigartige Testfälle generiert, welche die verschiedenen Ausführungspfade des Modells durchgehen und anhand der Nachrichten Testfälle generieren. Die Anzahl an Testfällen wurde so ge-

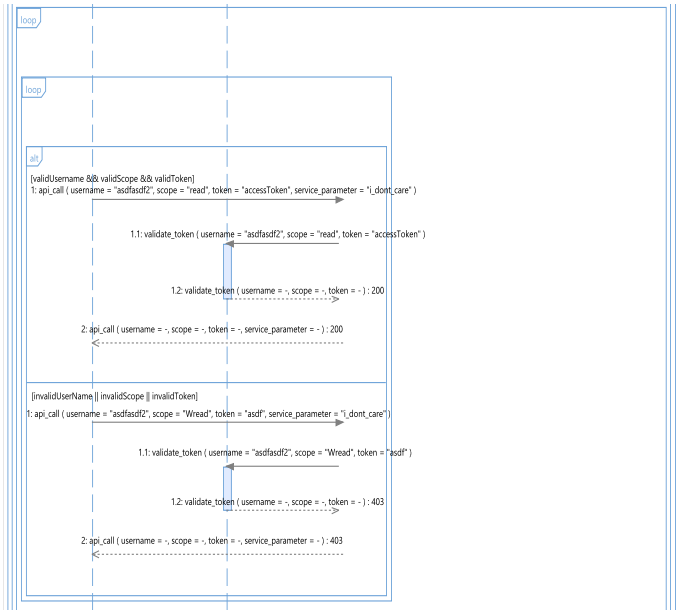


Abbildung 22: Modell für das Behavioral Fuzzing (2)

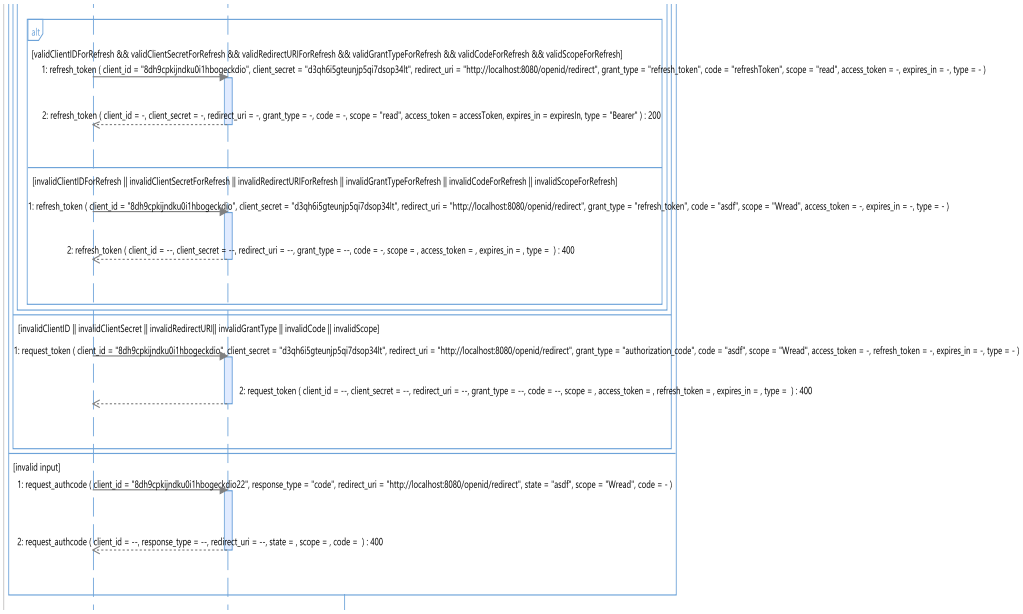


Abbildung 23: Modell für das Behavioral Fuzzing (3)

wählt, da bei den loop Fragmenten eine Anzahl von 0 bis maximal drei Wiederholungen durchgeführt werden können. Die Testfälle wurden gesondert betrachtet und angewendet. Dabei wurden an den entsprechenden Stellen

Assertions in den Testfall hinzugefügt, wodurch das erwartete Verhalten des Systems kontrolliert wurde. Die Testfälle haben einzelne Aspekte validiert und sichergestellt, dass das System entsprechend des Modells funktioniert. Im weiteren Verlauf wurde das Modell aus den Abbildung 21, 22 und 23 mit den bereits erwähnten Veränderungen gefuzzt, sodass aus diesem Sequenzdiagramm 84 neue, gefuzzte Sequenzdiagramme entstanden sind.
Aus diesen 84 Sequenzdiagrammen wurden abermals Testfälle generiert - es ergaben sich insgesamt 1145 einzigartige Testfälle. Da in diesen Testfällen veränderte Nachrichtensequenzen generiert wurden, mussten die Assertions vorab aus dem Testfallgenerator entfernt werden, da diese keinen Hinweis auf die Validität der Rückgaben des Systems geben. Daher wurde auf *valid case instrumentation* zurückgegriffen, was zur Bestimmung des Testfallergebnisses dient. Bei diesem Vorgehen wurden alle Assertions aus den Testfällen entfernt, beziehungsweise wurde der Generator so angepasst, dass die Assertions nicht mehr vorhanden sind und der gefuzzte Testfall ausgeführt wird. Im Anschluss an diesen gefuzzten Testfall wurde ein funktionaler Testfall ausgeführt, welcher sicherstellt, dass das System zum einen noch erreichbar ist und zum anderen kein unerwartetes Verhalten aufweist (valid case instrumentation).
Zusammenfassend ist zu sagen, dass mit Behavioral Fuzzing zwar keine Implementierungsfehler gefunden werden konnten, sich jedoch die Wahrscheinlichkeit bezüglich der Stabilität des Systems enorm steigern konnte. Durch diese Testfälle werden die Anforderungen 12 und 17 erfüllt, welche, wie bereits erwähnt, die Stabilität und Sicherheit der ISCS beinhalten.

7.5 Performance Testing

Abschließend wurden Performance Tests durchgeführt, um zu untersuchen, ob die Anforderungen 13 und 14 aus dem vierten Kapitel ebenfalls erfüllt wurden. Diese beziehen sich auf die Performanz (Anforderung 13) und den Ressourcenbedarf (Anforderung 14) der entwickelten ISCS. Verschiedene Integrationstests wurden ausgewählt und dahingehend erweitert, dass sie zum einen die Laufzeit und zum anderen den Speicherbedarf messen.

7.5.1 Laufzeitmessungen

Zu Beginn wurden einige Integrationstests, welche unter anderem für das Data Fuzzing verwendet wurden, ausgewählt. Die Ergebnisse der Laufzeit sind in Abbildung 24 dargestellt. Es sind vier typische Szenarios zu sehen, welche regelmäßig von der ISCS bearbeitet werden müssen. Die Ergebnisse sind als Boxplot visualisiert. Bei der Anfrage mit einem Access Token wurden 120 Sample erstellt, während bei der Anfrage mit einem Refresh Token 220 und bei den verbleibenden Anfragen jeweils 400 Testdaten generiert wurden. Mit einer Anzahl von stets mehr als 100 Testdaten ist die Stichprobe ausreichend groß.
In den Grafiken ist erkennbar, dass der Median der Antwortzeiten bei jeder einzelnen Anfrage kleiner als 30 ms ist. Ebenfalls sind 75% der Daten kleiner als 32 ms, sodass die Werte als sehr gut bezeichnet werden können. Jedoch ist zu beachten, dass beim Data Fuzzing vorrangig invalide Eingabedaten generiert werden, sodass die ISCS in diesen Tests nicht bei jedem Durchlauf Datenbankanfragen starten muss. Dennoch geben diese Tests einen ersten Hinweis auf die Performanz des Systems. Zudem wurde mit diesen Tests das System hinsichtlich von Verhaltensänderungen überprüft. Es ist festzustellen, dass das System auch beim Data Fuzzing weiterhin korrekt und performant arbeitet.

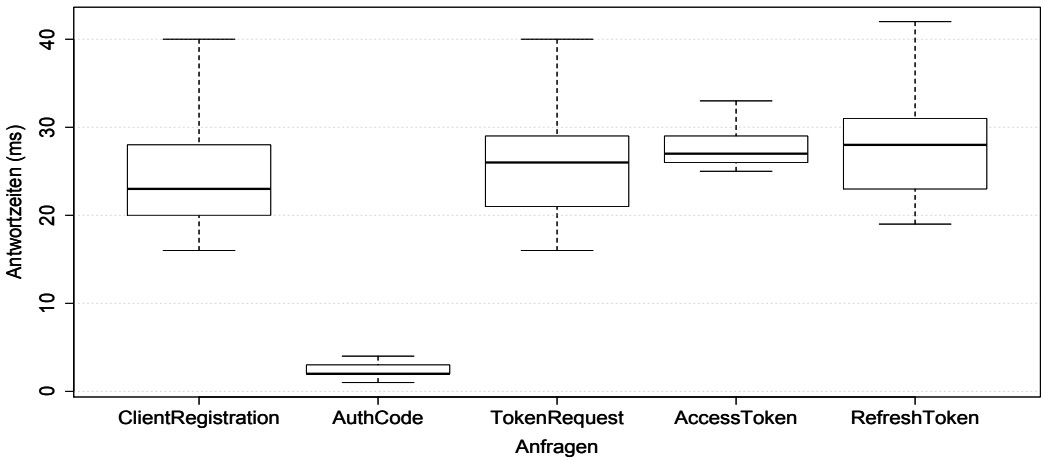


Abbildung 24: Laufzeitmessungen bei typischen Anfragen an die Komponente

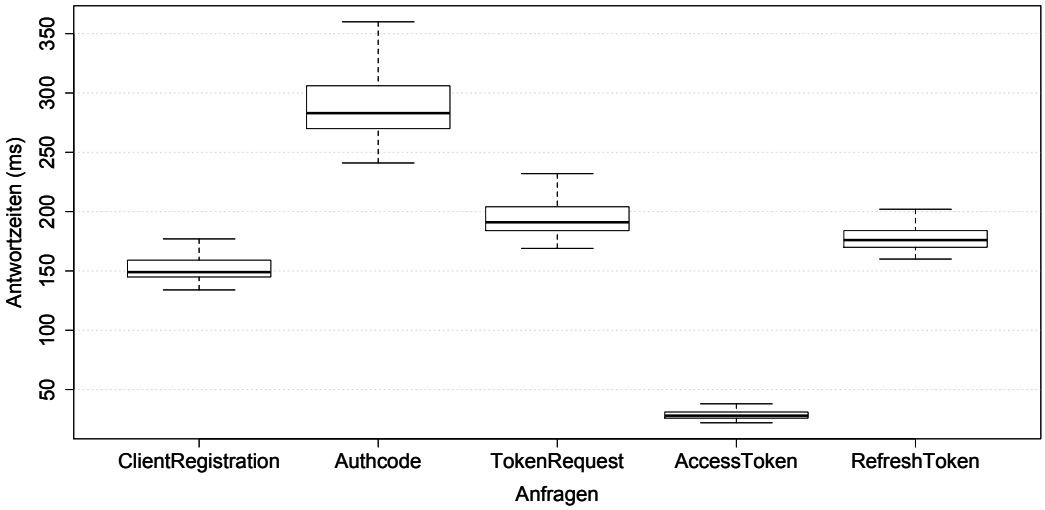


Abbildung 25: Laufzeitmessungen bei gültigen Anfragen an die Komponente

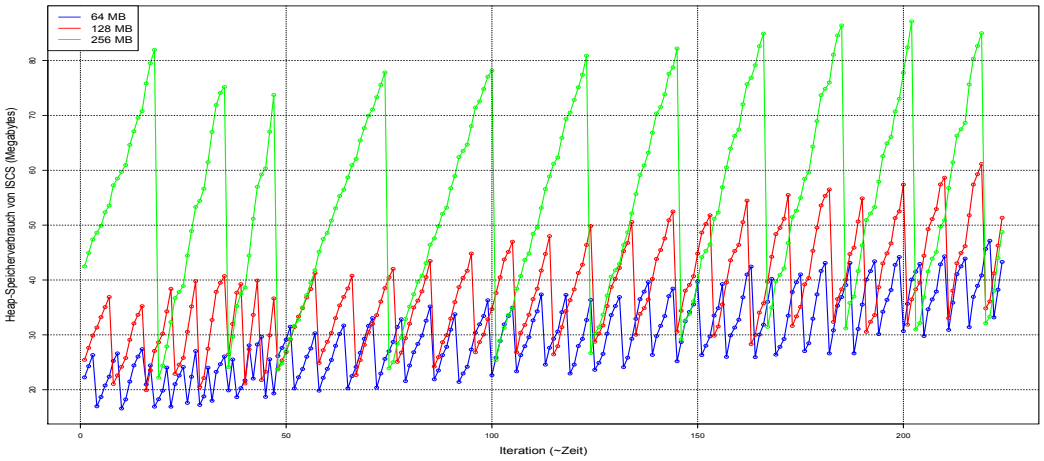
Im zweiten Schritt wurde ein Testfall, welcher den kompletten Ablauf des OAuth Protokolls mit OpenID darstellt, dahingehend erweitert, dass mit diesem die Laufzeiten gemessen werden konnten. Dieser Testfall wurde 400-mal ausgeführt, wobei dessen Ergebnisse in Abbildung 25 visualisiert sind. Es ist erkennbar, dass sich die Laufzeiten erhöht haben, was mit dem Vorhandensein von Datenbankanfragen erklärt werden kann. Im Fall einer Autorisierungsanfrage (Abbildung 25 Eintrag „Authcode“) liegen 75 Prozent der Messungen bei einem Wert kleiner als 300 ms, was ein sehr gutes Ergebnis ist, da dieser Fall bereits die Authentifizierung per OpenID beinhaltet. Wie bereits im Kapitel Implementierung erwähnt, assoziieren sich Consumer und Provider, sodass sie im weiteren Verlauf Signaturen erstellen können. Der Consumer wurde dahingehend optimiert, dass dieser einen Cache von Assoziationen halten kann. Dies verhindert das stetige Erstellen von Assoziationen bei jeder neuen Anfrage. Infolgedessen fallen die Messungen in den darauf folgenden Durchläufen geringer als der erste Wert aus, weil dieser den Maximalwert repräsentiert. Darüber hinaus ist anzumerken, dass der Testfall den Nutzer imitiert, indem er immer zustimmt. Im realen Anwendungsfall würde der Nutzer mehr Zeit benötigen, sodass die ermittelten Werte bereits optimal sind. Zusätzlich zu der Authentifizierung des Nutzers müssen bei dieser Autorisierungsanfrage die meisten Datenbankanfragen gestellt werden. Dadurch

kann erklärt werden, weshalb diese Anfrage die meiste Zeit benötigt.
Weiterhin ist die darauf folgende Anfrage bezüglich des Erhalts von Token mit dem Autorisierungscode zu sehen (Abbildung 25 Eintrag „Token-Request“). Dabei ist zu erkennen, dass der Median bei 177 ms liegt. Dieser Wert ist im Vergleich zur Autorisierungsanfrage geringer, da der Nutzer an dieser Stelle nicht zusätzlich per OpenID authentifiziert werden muss. Der Client authentifiziert sich hierbei über mitgesendete Parameter, was den Aufwand nicht erhöht. Der Wert von 177 ms als Antwortzeit entspricht erneut einem sehr guten Ergebnis. Im Vergleich zu den anderen Anfragen wird die Validierung eines Access Tokens sehr schnell ausgeführt. Keine Anfrage an diesen Endpunkt hat länger als 50 ms benötigt. Es wurde darauf geachtet, dass diese Abfrage möglichst schnell zu einer Entscheidung kommt, damit die Anfrage an den eigentlichen Dienst nicht wesentlich verzögert wird. Mit einem Median von 28 ms ist dieser Wert optimal.
Nach dem Autor Jakob Nielsen sollte der Wert ungefähr zwischen 0.1 und 1 Sekunde liegen, damit der Nutzer denkt, dass das System unverzüglich antwortet [38]. Bei einer Antwortdauer von bis zu 1 Sekunde wird der Gedankenfluss des Nutzers nicht unterbrochen, sodass es nicht notwendig wird, zusätzliches Feedback zu versenden [38]. Jedoch würde der Nutzer eine Verzögerung bemerken. Es kann daher betont werden, dass die ermittelten Werte auch im echten Anwendungsfall im großen Teil kleiner als 300 ms sind. Dies entspricht einem sehr guten Ergebnis, sodass die ISCS als performant angesehen werden kann und Anforderung 13 erfüllt ist.

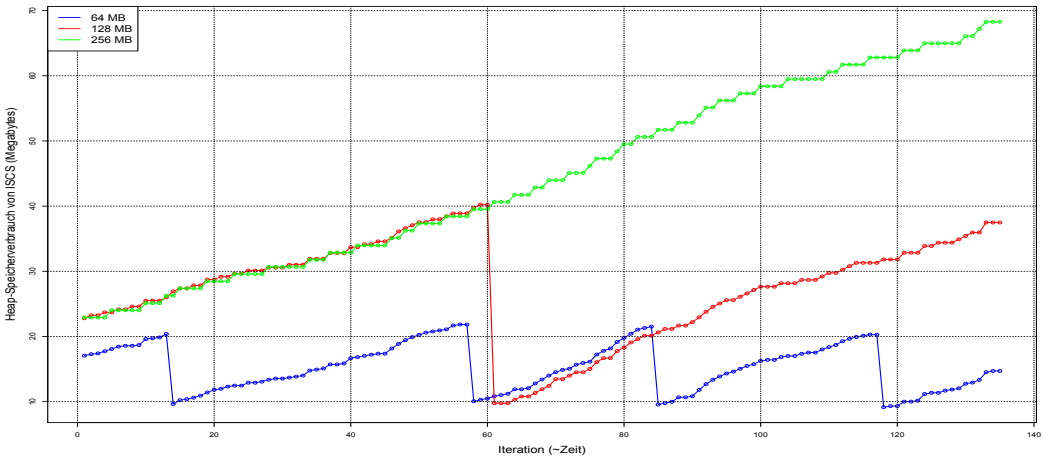
7.5.2 Speicherbedarf

In diesem Kapitel wurde der Speicherbedarf der integrierten Komponenten untersucht. Dabei wurden die Testfälle zur Laufzeitmessung dahingehend angepasst, dass sie den aktuellen Speicherverbrauch bezüglich des Heaps der Komponente ausgeben. Die Java Virtuale Machine wurde mit unterschiedlichen Heap-Größen gestartet, wobei der Heap zwischen 64 MB, 128 MB und 256 MB variiert. Abermals wurden im ersten Schritt Testfälle mit Data Fuzzing verwendet. Die Ergebnisse sind in den Abbildungen 26 und 27 dar-

gestellt, wobei der typische Speicherbedarf einer Java Anwendung dargestellt ist. Der Speicherbedarf wächst bis zu einem bestimmten Punkt, bevor der Garbage Collector aufgerufen wird und Speicher wieder freigibt.
Wichtig ist die Untersuchung des Speicherbedarfs mit der Kombination von Fuzzing. Würde es zu einer Veränderung des Systemverhaltens durch die gefuzzten Werte kommen, könnte dadurch unter Umständen ein Memory Leak entstehen. Dies würde in den Abbildungen deutlich auffallen. Jedoch verlaufen alle Graphen in den Abbildungen korrekt, was darauf hinweist, dass die Komponente zum einen nicht negativ durch die gefuzzeten Werte beeinflusst wird, was die Stabilität des Systems erhöht. Zum anderen kann angenommen werden, dass keine auf Implementierungsschwächen basierende Memory Leaks vorhanden sind. Wie im vorangegangen Abschnitt erwähnt, sind die Daten, welche in diesen Testfällen gesendet werden, immer mit mindestens einem Parameter invalide. Dadurch werden nicht in jedem Fall Datenbankanfragen gestellt. Jedoch erhöht auch dies die Gefahr des Auftretens eines Memory Leaks, zum Beispiel beim nicht ordnungsgemäßen Schließens der Verbindungen an die Datenbank.



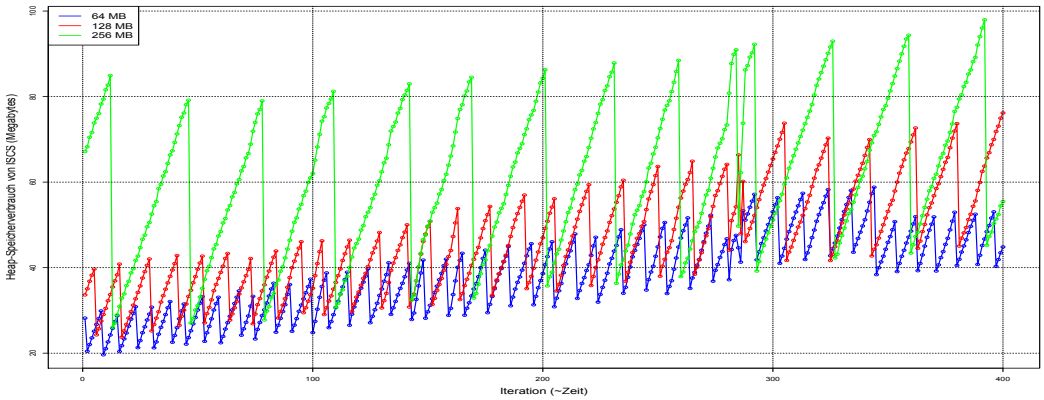
(a) Clientregistrierung



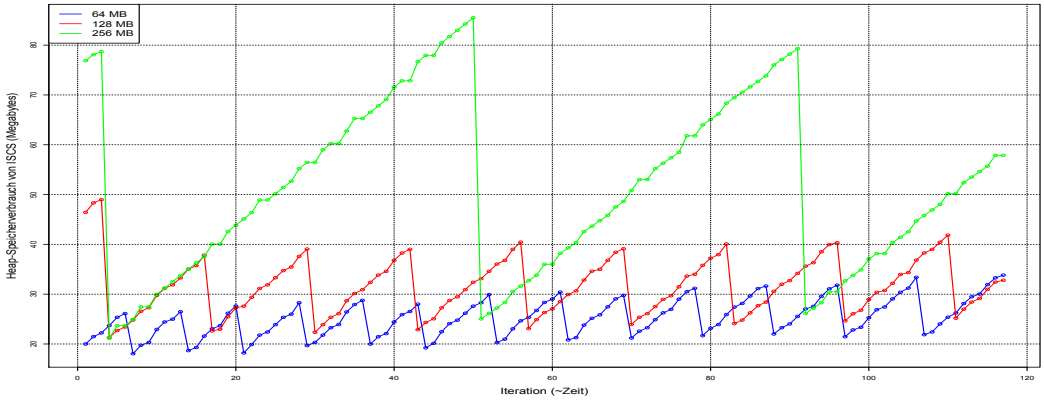
(b) Autorisierungscodeanfrage

Abbildung 26: Speicherbedarf bei typischen Anfragen an die Komponente mit verschiedenen Heapgrößen (1)

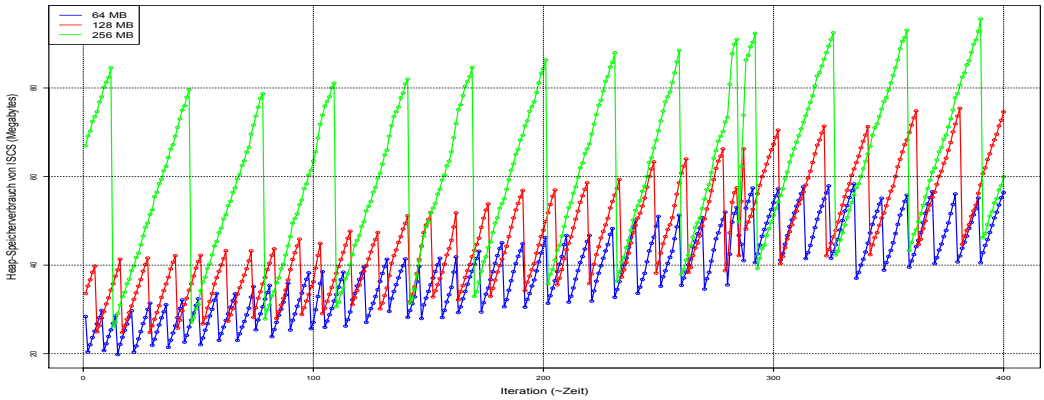
Analog zu den Laufzeitmessungen wurde daher in einem zweiten Schritt ein Testfall ausgewählt, welcher einen gültigen Aufruf an die Komponente darstellt. Dieser Testfall wurde jeweils 400-mal mit verschiedenen Heapgrößen ausgeführt, wobei der Speicherbedarf jeweils am Ende eines einzelnen Durchlaufs (Clientregistrierung, Authcode, Tokenrequest, Accesstokenvalidation und Refresh Token wurden ausgeführt) gemessen wurde. Die Ergebnisse sind in Abbildung 28 dargestellt.
Es ist erneut erkennbar, dass die einzelnen Messungen dem Verlauf einer typi-



(a) Tokenanfrage



(b) Validierung eines Access Token



(c) Anfrage mit Refresh Token

Abbildung 27: Speicherbedarf bei typischen Anfragen an die Komponente mit verschiedenen Heapgrößen (2)

schen Java Anwendung folgen. Ebenfalls ist zu sehen, dass der Garbage Collector umso öfter aufgerufen werden muss, je geringer die Heapgröße ist. Das führt zu Performanceeinbußen, weil der Garbage Collector das Programm unterbricht und erst seine Aufgabe zu Ende führt, bevor das Programm weiterläuft. Darüber hinaus wird Speicher kontinuierlich frei gegeben, womit sich die Vermutung bestätigt, dass auch bei den Datenbankverbindungen keine Memory Leaks vorhanden sind.
Der Ressourcenbedarf der ISCS ist als ausreichend gering zu bezeichnen. Solange Speicher vorhanden ist, wird dieser genutzt, jedoch nie in einem exzessiven Rahmen, was auf Memory Leaks hinweisen könnte. Daher ist davon auszugehen, dass die entwickelte Lösung frei von Memory Leaks oder sonstigen ressourcenerhöhenden Codefragmenten ist. Mit diesen Untersuchen wird die Anforderung 14 aus dem vierten Kapitel erfüllt.

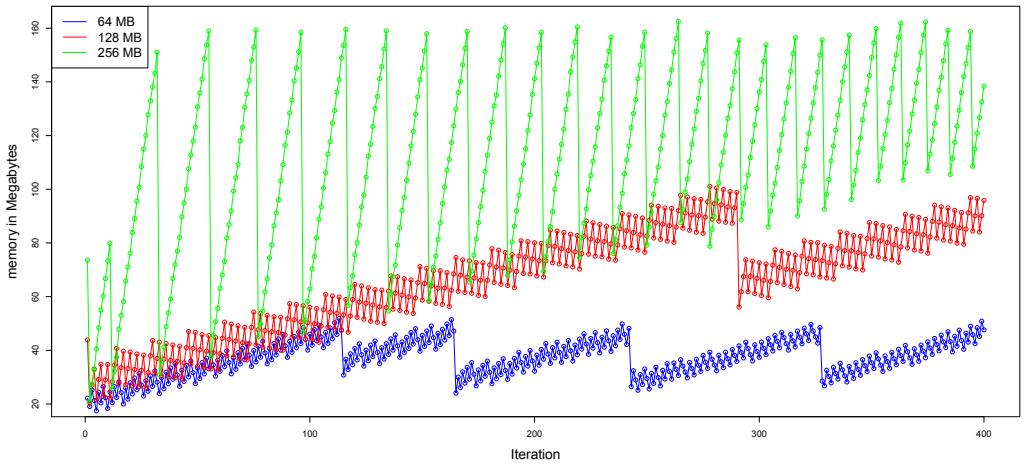


Abbildung 28: Speicherbedarf bei gültigen Anfragen an die Komponente mit verschiedenen Heapgrößen

Insgesamt wurden 5776 Testfälle entwickelt und generiert. Dies beinhaltet die Durchführung von Unit Tests, Integrationstests, Fuzz Tests und Performance Tests. Es konnte festgestellt werden, dass die in diesem Kapitel genannten Anforderungen an die Stabilität, die Performanz und den Ressourcenbedarf erfüllt sind. Darüber hinaus kann davon ausgegangen werden, dass die Komponente hinreichend sicher ist. Jedoch ist abermals zu erwähnen, dass es

keine hundertprozentige Sicherheit gibt, sodass davon auszugehen ist, dass eventuell noch Fehler im Code vorhanden sind, die mit der Vielzahl von Tests nicht aufgedeckt werden können.

8 Fazit

In dieser Masterarbeit wurde eine integrierte Komponente zur Sicherung von Cloud Services entwickelt. Die Komponente basiert auf dem Standard OAuth und dem Community Standard OpenID. Sie wurde im Rahmen des Projekts GeMo konstruiert, um einen sicheren Zugriff für Dienste beziehungsweise Applikationen und Nutzer auf die bei GeMo verwendete Cloud zu gewährleisten. Zu Beginn der Arbeit wurde, zum besseren Verständnis und zur Einbettung des Themas, der Kontext des Projekts GeMo eingehend beleuchtet. Daran anknüpfend wurde der aktuelle Forschungsstand skizziert, wobei mehrere Authentifizierungs- und Autorisierungssysteme hinsichtlich ihrer Funktionsweise, Komplexität und Sicherheitsaspekte vorgestellt und verglichen wurden. Im Hauptteil dieser Arbeit wurde die Entwicklung der integrierten Komponente zur Sicherung von Services in der Cloud dokumentiert. Dazu wurden im ersten Schritt verschiedene Anforderungen, im Sinne von zu erfüllenden Voraussetzungen, an die Komponente formuliert. Weiterhin wurde die Architektur der Mobilitätsdaten-Cloud (MDC) und der entwickelten Komponente erläutert, wobei die identifizierten Komponenten und dynamische Aspekte (Interaktionen und Informationsflüsse) vorgestellt und hinsichtlich der zu erfüllenden Anforderungen diskutiert wurden. Im Fokus des Hauptteils der Arbeit steht die Implementierung der Komponente. In diesem Kapitel wurden insbesondere die Teilkomponenten OAuth und OpenID sowie der Zugang zu benötigten Datenbanken eingehend erörtert. Abschließend wurde die erarbeitete Lösung hinsichtlich ihrer Stabilität, Sicherheit und Performanz evaluiert, indem neben Unit Tests auch Integrationstest, Fuzz Tests sowie Performance Tests durchgeführt wurden. Bei den Fuzz Tests wurden sowohl Data Fuzzing als auch Behavioral Fuzzing angewendet, wobei beim Data Fuzzing die von Fraunhofer FOKUS entwickelte Java Bibliothek *Fuzzino* verwendet wurde. Für die Untersuchung der Performanz wurden die Laufzeiten von typischen Anfragen sowie der Speicherbedarf analysiert.

Hinsichtlich der Komplexität der integrierten Komponente zur Sicherung von Cloud Services kann festgestellt werden, dass die Lösung auf Basis von OpenID und OAuth relativ wenig komplex ist, jedoch eine hohe Funktionalität

aufweist. Die beiden Teilkomponenten wurden unter anderem ausgewählt, da es sich bei diesen um Standards, beziehungsweise im Fall von OpenID, um einen Community Standard handelt, welche sehr häufig eingesetzt werden. Weiterhin wurde eine leichte Integrierbarkeit des Identitätsmanagements angefordert. Dies konnte durch die Verwendung von OpenID realisiert werden, da die Komponente somit ohne zusätzlichen Aufwand in die MDC, welche auf der Open Data Plattform von Fraunhofer FOKUS [32] aufbaut, integriert werden konnte.
Ein weiteres zentrales Ergebnis ist, dass in dieser Arbeit eine neuartige Lösung konstruiert wurde, die es in diesem Rahmen noch nicht gibt, da bei den meisten bisherigen Implementierungen die Authentifizierung vernachlässigt wird. Die Komponente ist hinsichtlich Authentifizierung und Autorisierung vollständig entwickelt und kann dabei leicht deployed sowie in bestehende Services sowie in Service Umgebungen integriert werden. Konkret funktioniert die Integration der Software in bestehende Services, indem die Konfiguration (web.xml) des Services in Bezug auf den genutzten Applikationsserver, beispielsweise Tomcat oder Goldfish, angepasst und ein Filter hinzugefügt wird, welcher die Anfrage an den Dienst abfängt und eine Überprüfung der gesendeten Parameter durch die ISCS veranlasst. Die Komponente kann daher auch im Nachhinein noch in Projekte integriert werden und dort die Zugriffssicherheit auf die Dienste gewährleisten. Auf diese Weise ist es auch im Projekt GeMo umgesetzt worden, da bei jenem die meisten Dienste bereits existierten, als die Entwicklung der Komponente begann. Die einfache Integration der Software konnte daher innerhalb des Projekts überprüft werden. Folglich sollte dies auch in anderen Projekten möglich sein.
Außerdem beinhaltet die Komponente ein eigenes Identitätsmanagement, welches für die MDC, welche auf der von Fraunhofer FOKUS entwickelten Open Data Plattform basiert, genutzt werden kann. An diesem ist bisher noch nachteilig, dass das Modell eines Nutzers aktuell noch zu sehr auf einem typischen GeMo Nutzer basiert und insofern angepasst werden müsste, als dass der Nutzer generischer wird. Dies soll, wie bereits im Kapitel Implementierung erwähnt, in einer weiteren Entwicklungsphase umgesetzt werden.
Zusätzlich zur leichten Integrierbarkeit der Komponente wurde der Fokus bei

der Entwicklung auf die Stabilität und Sicherheit der Lösung gelegt. Durch die Verwendung der Bibliothek *Fuzzino* war es leicht möglich, starke Testfälle zu erstellen, welche jene Sicherheit und Stabilität prüfen, da diese bereits verschiedene Generatoren, beispielsweise für Formatierungsfehler bei Strings oder Strings mit SQLInjections, bereitstellt. Die Stabilität und Zuverlässigkeit der Lösung standen im Fokus, weil bei Instabilitäten der ISCS die zugriffsgeschützten Dienste nicht mehr sicher angesprochen werden können. Daher wurde die Software während und nach der Entwicklung ausgiebig getestet. Es wurden insgesamt 5776 Testfälle, darunter Unit Tests, Integrationstests, Fuzz Tests und Performance Tests, erstellt und generiert. Mit diesen Tests konnte festgestellt werden, dass die Komponente gemäß der Spezifikation der jeweiligen Protokolle funktioniert. Ebenfalls wurde Fuzz Testing, sowohl als Data Fuzzing als auch Behavioral Fuzzing, als gute Möglichkeit des Sicherheitstestens verwendet. Mit Hilfe dieser Evaluationen hat sich die Gewissheit an Stabilität und Sicherheit enorm erhöht. In einem weiteren Schritt könnte man noch die beiden Teilbereiche des Fuzz Testing, Data und Behavioral Fuzzing, kombinieren, um weitere Sicherheitstests durchzuführen. Dies wird in einer zukünftigen Entwicklungsphase umgesetzt.
Abschließend wurde Wert auf die Performanz der Lösung gelegt, weil diese einen möglichsten geringen Overhead bei Anfragen an Diensten erzeugen sollte, sodass dadurch die Antwortzeit der Dienste nicht wesentlich beeinträchtigt wird. Daher wurden im Rahmen von Performance Tests die Laufzeiten von typischen Anfragen und der Speicherbedarf geprüft. Es kann betont werden, dass die Antworten auf fast alle Anfragen in weniger als 300 ms gesendet wurden. Bezugnehmend auf die aufgestellten Kriterien von Jakob Nielsen in *Usability Engineering* [38] handelt es sich dabei um einen sehr guten Wert. Ebenfalls konnte festgestellt werden, dass der Ressourcenbedarf der Software in einem geringem Rahmen liegt.
Die Integrierte Komponente zur Sicherung von Cloud Services wurde in Java entwickelt und basiert auf den Open Source Bibliotheken *openid4java* [18] und *Apache Oltu* [1]. Im nächsten Schritt kann daher angestrebt werden, dass die im Rahmen dieser Arbeit entwickelten ISCS als Open Source für die Community zur Verfügung gestellt wird. Die Verwendung von Open Source

Bibliotheken hat unter anderem den Vorteil, dass Lizenzprobleme reduziert werden.

9 Ausblick

Die integrierte Komponente zur Sicherung von Cloud Services (ISCS) weist großes Potential auf, wobei dennoch Verbesserungen vorgenommen werden können. Auf beide Aspekte soll im Folgenden knapp eingegangen werden.
Die Durchführung einer zweiten Entwicklungsphase bietet die Möglichkeit, das eigene Identitätsmanagement, welches für die Mobilitätsdaten-Cloud genutzt wird, dahingehend zu verbessern, dass das Modell des Nutzers noch generischer werden könnte. In dieser Arbeit erinnert das Nutzermodell an einen typischen GeMo-Nutzer, was nicht der Anwendungsrealität bei einer Integration mit beliebigen Cloud Diensten entsprechen muss. Weiterhin kann die Software durch allgemeines Refactoring in dieser zweiten Phase optimiert werden. Bezüglich der Sicherheit der ISCS könnte zudem ein erweitertes Behavioral Fuzzing stattfinden. Beispielsweise könnte eine Selektion der generierten Testfälle erfolgen, welche mit Hilfe von Data Fuzzing kombiniert sowie mit Assertions und valid case instrumentations ergänzt werden. Durch dieses Vorgehen würde man mehr Gewissheit über die Sicherheit und Stabilität des Systems erlangen.
An dieser Stelle ist zu erwähnen, dass OpenID Connect [17] im Februar 2014 finalisiert wurde. OpenID Connect ist eine neue Version von OpenID, die eine einfache Identitätsschicht zum OAuth-Protokoll hinzufügt. Die ISCS könnte hinsichtlich der Implementierung dieses Standards erweitert werden. Zu Beginn dieser Arbeit befand sich OpenID Connect noch in der Entstehungsphase, sodass die hier verwendete Implementierung zwar ähnlich jedoch nicht identisch zu diesem Standard ist. Die Implementierung beider Standards hätte den Vorteil, dass eine duale Lösung entsteht, welche sowohl OpenID Authentication als auch OpenID Connect umsetzt. Dennoch muss die entwickelte Komponente gegenüber einer Lösung mit OpenID Connect kritisch abgegrenzt werden. Ein wichtiger Vorteil der ISCS ist die leichte Integrierbarkeit in die Mobilitätsdaten-Cloud, was durch die Existenz eines eigenen Providers und Consumers gewährleistet wird. OpenID Connect weist diese Kompatibilität hingegen nicht auf, sodass diese Lösung die Stärke der Integrierbarkeit verliert. Nichtsdestotrotz ist die Integrierbarkeit der ISCS in andere Projek-

te noch zu überprüfen, wobei auch dies zukünftig geschehen könnte, zum Beispiel im Rahmen von ModelBus [10], das für eine Webservice basierende Lösung von Fraunhofer FOKUS zum Austausch und zur Verarbeitung von Modellartefakten, im Bezug auf modellgetriebene Systementwicklungsprozesse, steht. Dem gegenüberzustellen ist, dass die ISCS aktuell wesentlich ausführlicher und vielseitiger getestet wurde als bestehende Implementierungen von OpenID Connect. Vor allem das zusätzliche Sicherheitstesten durch die Bibliothek *Fuzzino* ist als positiv zu benennen, da andere Lösungen oft nur mit Hilfe von Unit Tests und Integrationstests überprüft werden.
Weiterhin steckt das Potential der ISCS in deren möglicher Open Source Stellung. Mit diesem Vorhaben wird die Intention verfolgt, der Community eine leicht zu verwendende Lösung anzubieten, die auch anderweitig genutzt werden kann. Außerdem ist mit der Möglichkeit der öffentlichen Inspektion des Codes die Chance verknüpft, Schwachstellen aufzudecken und ohne viel Zeitverlust beheben zu können.
Der erste Schritt bei der konkreten Umsetzung der Open Source Stellung ist, dass der Code noch einmal überarbeitet und kritisch geprüft werden müsste. Daran anknüpfend müsste eine Lizenz gewählt werden, welche kompatibel zu den Lizenzen der verwendeten Bibliotheken ist. Infolgedessen müssten zudem alle externen Abhängigkeiten aktualisiert und aufgezählt werden, wobei darauf zu achten ist, dass die Software keine Einbußen hinsichtlich ihrer Funktionalität erhält und die erarbeiteten Anforderungen weiterhin erfüllt werden. Die Verbreitung der ISCS in der Community wird dann erfolgreich, wenn eine Dokumentation und Beispiele zugänglich gemacht werden. Dabei ist zu beachten, dass der Code gut kommentiert ist und ein Installationsguide entwickelt wird. Der letzte Umsetzungsschritt besteht darin, dass ein Hoster für die Open Source Stellung gefunden wird. Beispielsweise ist auf GitHub[10] ein Fraunhofer FOKUS Repository vorhanden, zu welchem die entwickelte Komponente hinzugefügt werden könnte.
In der vorliegenden Masterarbeit konnte nachgewiesen werden, dass eine Lösung entwickelt werden kann, die einen sicheren Zugriff auf Dienste, Applikationen oder andere Services ermöglicht, was einen zentralen Aspekt im

[10]URL: `https://github.com/` Stand: 24. August 2014

Zusammenhang mit Kommunikationsnetzwerken darstellt. Die entwickelte Komponente weist großes Einsatzpotential auf, da sie auch in anderen als den hier vorgestellten Kontexten genutzt und noch weiterentwickelt werden kann.

Abbildungsverzeichnis

1 Schematische Darstellung der Mobilitätsdaten-Cloud 4
2 Der Protokollfluss von Kerberos 9
3 Der Protokollfluss von Persona 13
4 Der Protokollfluss von OpenID 18
5 Der Protokollfluss von SAML 24
6 Der Protokollfluss von Shibboleth 27
7 abstrakte Darstellung des Protokollflusses [28] 30
8 Der Protokollfluss von OAuth (Autorisierungscode) [28] 31
9 Darstellung der Architektur der Mobilitätsdaten-Cloud [45] . . 38
10 Darstellung der Architektur der entwickelten ISCS 42
11 Darstellung der Clientregistrierung 43
12 Darstellung der Benutzerregistrierung 44
13 Darstellung der typischen Anfragen 45
14 Klassendiagramm des Autorisierungsdatenbank Moduls 48
15 Klassendiagramm des Benutzerdatenbank Moduls 51
16 Klassendiagramm der OpenID Komponente 52
17 Screenshot des Login Formulars 54
18 Screenshot des Formulars für die Benutzerregistrierung 55
19 Klassendiagramm der OAuth-Komponente 57
20 Screenshot des Formulars 60
21 Modell für das Behavioral Fuzzing (1) 72
22 Modell für das Behavioral Fuzzing (2) 73
23 Modell für das Behavioral Fuzzing (3) 73
24 Laufzeitmessungen bei typischen Anfragen an die Komponente 75
25 Laufzeitmessungen bei gültigen Anfragen an die Komponente . 76
26 Speicherbedarf bei typischen Anfragen an die Komponente mit verschiedenen Heapgrößen (1) 79
27 Speicherbedarf bei typischen Anfragen an die Komponente mit verschiedenen Heapgrößen (2) 80
28 Speicherbedarf bei gültigen Anfragen an die Komponente mit verschiedenen Heapgrößen 81

Listings

1 Beispiel einer CDI Konfigurationsdatei (beans.xml) 50

Abkürzungsverzeichnis

ABNF angereicherte Backus-Naur-Form

CDI Contexts and Dependency Injection

CKAN Comprehensive Knowledge Archive Network

ESK Fraunhofer-Institut für Eingebettete Systeme und Kommunikationstechnik

FOKUS Fraunhofer-Institut für Offene Kommunikationssysteme

GeMo Gemeinschaftlich-e-Mobilität: Fahrzeuge, Daten und Infrastruktur

IAO Fraunhofer-Institut für Arbeitswirtschaft und Organisation

IIS Fraunhofer-Institut für Integrierte Schaltungen

ISE Fraunhofer-Institut für Solare Energiesysteme

IVI Fraunhofer-Institut für Verkehrs- und Infrastruktursysteme

MDC Mobilitätsdaten-Cloud

OBU On-Board-Unit

PaaS Platform-as-a-Service

POM Project Object Model

SAML Security Assertion Markup Language

SUT System Under Test

TGS Ticket Granting Service

TGT Ticket Granting Ticket

XSS Cross-site-scripting

Literatur

[1] Apache Oltu. https://oltu.apache.org/. Stand: 24. August 2014.

[2] Definition of Open Data. http://opendefinition.org/. Stand: 24. August 2014.

[3] GeMo - Hocheffizientes induktives Ladesystem für Elektrofahrzeuge entwickelt. http://www.ise.fraunhofer.de/de/presse-und-medien/presseinformationen/presseinformationen-2013/es-geht-auch-ohne-kabel. Stand: 24. August 2014.

[4] GeMo Mobilitätsdaten-Cloud. http://www.gemo.fraunhofer.de/de/technologien/mobilitaetsdatencloud.html. Stand: 24. August 2014.

[5] GeMo Mobilitätsdienste. http://www.gemo.fraunhofer.de/de/technologien/mobilitaetsdienste.html. Stand: 24. August 2014.

[6] GeMo Projektbeschreibung. http://www.gemo.fraunhofer.de/de/Projektbeschreibung.html. Stand: 24. August 2014.

[7] Hibernate. http://hibernate.org/. Stand: 24. August 2014.

[8] How Shibboleth Works: Basic Concepts. http://shibboleth.net/about/basic.html. Stand: 24. August 2014.

[9] How Shibboleth Works: Identity Provider Discovery, User Attributes and Metadata. http://shibboleth.net/about/intermediate.html. Stand: 24. August 2014.

[10] ModelBus. http://www.modelbus.org/. Stand: 24. August 2014.

[11] Mozilla Persona. https://developer.mozilla.org/en-US/Persona. Stand: 24. August 2014.

[12] Mozilla Persona - Security considerations. https://developer.mozilla.org/en-US/Persona/Security_Considerations. Stand: 24. August 2014.

[13] OASIS Security Services (SAML) TC. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security. Stand: 24. August 2014.

[14] Online Community for the Security Assertion Markup Language (SAML) OASIS Standard. http://www.saml.xml.org. Stand: 24. August 2014.

[15] Openid attribute exchange 1.0 - final. http://openid.net/specs/openid-attribute-exchange-1_0.html. Stand: 24. August 2014.

[16] OpenID Authentication 2.0 - Final. http://openid.net/specs/openid-authentication-2_0.html. Stand: 24. August 2014.

[17] OpenID Connect. http://openid.net/connect/. Stand: 24. August 2014.

[18] OpenID4Java. https://code.google.com/p/openid4java/. Stand: 24. August 2014.

[19] Persona After Action Report. https://wiki.mozilla.org/Identity/Persona_AAR. Stand: 24. August 2014.

[20] Persona Protocol Overview. https://developer.mozilla.org/en-US/Persona/Protocol_Overview. Stand: 24. August 2014.

[21] Security and Privacy Considerations for the OASIS Security Markup Language V2.0. http://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf. Stand: 24. August 2014.

[22] Shibboleth. http://shibboleth.net/. Stand: 24. August 2014.

[23] Transitioning Persona to Community Ownership. http://identity.mozilla.com/post/78873831485/transitioning-persona-to-community-ownership. Stand: 24. August 2014.

[24] L. H. Astrand and T. Yu. Deprecate DES, RC4-HMAC-EXP, and Other Weak Cryptographic Algorithms in Kerberos. RFC 6649 (Best Current Practice), July 2012.

[25] S. Bradner. Key words for use in rfcs to indicate requirement levels, 1997.

[26] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. Communications of the ACM, 24(8):533–536, 1981.

[27] B. Dittwald. Eine webbasierte infrastruktur zur speicherung und abfrage von strukturierten offenen daten. Master's thesis, Freie Universität Berlin, July 2012.

[28] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), Oct. 2012.

[29] M. Indrawan-Santiago. Database research: Are we at a crossroad? reflection on nosql. In Network-Based Information Systems (NBiS), 2012 15th International Conference on, pages 45–51, Sept 2012.

[30] M. Jones and D. Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750 (Proposed Standard), Oct. 2012.

[31] S. Josefsson. Extended Kerberos Version 5 Key Distribution Center (KDC) Exchanges over TCP. RFC 5021 (Proposed Standard), Aug. 2007.

[32] E. Lapi, N. Tcholtchev, L. Bassbouss, F. Marienfeld, and I. Schieferdecker. Identification and utilization of components for a linked open data platform. 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops, 0:112–115, 2012.

[33] T. Lodderstedt, M. McGloin, and P. Hunt. Oauth 2.0 threat model and security considerations. 2013.

[34] M. Mierau, S. Fey, R. Kohrs, and C. Wittwer. Communication Solutions for a Cloud-Based Charging Management System for a Fleet of Shared-Use Electric Vehicles. EVS27 International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium 1, 2013.

[35] J. Miller. Yadis specification. Notes, 6:3, 2006.

[36] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. Commun. ACM, 21(12):993–999, Dec. 1978.

[37] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), July 2005. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806.

[38] J. Nielsen. Usability engineering. Elsevier, 1994.

[39] D. Noeren, S. Reichert, S. Tönjes, and T. Ernst. Neue Mobilität. Ein Ausblick zur urbanen Mobilität in vier Dekaden. 2013.

[40] E. Rescorla. Diffie-hellman key agreement method. 1999.

[41] M. Schneider, J. Großmann, I. Schieferdecker, and A. Pietschker. Online model-based behavioral fuzzing. Software Testing Verification and Validation Workshop, IEEE International Conference on, 0:469–475, 2013.

[42] M. Schneider, J. Großmann, N. Tcholtchev, I. Schieferdecker, and A. Pietschker. Behavioral fuzzing operators for uml sequence diagrams. In Ã. Haugen, R. Reed, and R. Gotzhein, editors, System Analysis and Modeling: Theory and Practice, volume 7744 of Lecture Notes in Computer Science, pages 88–104. Springer Berlin Heidelberg, 2013.

[43] M. Schneider, N. Tcholtchev, J. Großmann, A. Pietschker, A.-G. V. Feudjio, and I. Schieferdecker. Model based behavioral fuzzing for the banking domain. In MBT User Conference, 2012.

[44] H. Streng and J. V. Abeelen. Cloud-controlled charging. 2013.

[45] N. Tcholtchev, B. Dittwald, T. Scheel, B. I. Zilci, D. Schmidt, P. Lämmel, J. Jacobsen, and I. Schieferdecker. The Concept of a Mobility Data Cloud: Design, Implementation and Trials. In IEEE MidArch - 8th IEEE International Workshop on Middleware Architecture in the Internet co-located with COMPSAC 14, 2014.

[46] N. Tcholtchev, L. Farid, F. Marienfeld, I. Schieferdecker, B. Dittwald, and E. Lapi. On the interplay of open data, cloud services and network providers towards electric mobility in smart cities. In Local Computer Networks Workshops (LCN Workshops), 2012 IEEE 37th Conference on, pages 860–867, Oct 2012.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass die vorliegende Masterarbeit von niemand anderem als meiner Person verfasst wurde. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder Ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Philipp Lämmel
Berlin, den 03. September 2014