Philipp Lämmel, Nikolay Tcholtchev* und Ina Schieferdecker

Integriertes Internet-basiertes System für Authentifizierung und Autorisierung

DOI 10.1515/pik-2015-0002

Abstract: Die Sicherung von Diensten spielt eine zentrale Rolle bei der Entwicklung einer sicherheitskritischen Infrastruktur. In diesem Artikel wird daher eine *Integrierte Komponente zur Sicherung von Cloud Services (ISCS)* präsentiert, die den Zugriff auf Dienste in der Cloud gewährleistet. Das bedeutet, dass die ISCS für Zugriffsaspekte wie Authentifizierung, Autorisierung und Registrierung zuständig ist. 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.

Schlüsselwörter: Authentifizierung; Autorisierung; Open Data

I Einleitung

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. Üblicherweise 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. In diesem Artikel wird eine Komponente vorgestellt, welche OAuth und OpenID kombiniert, um einen sicheren Zugriff für Dienste beziehungsweise Applikationen und Nutzer auf die Cloud zu gewährleisten. Die *Integrierte Komponente zur Sicherung von Cloud Services* wird im weiteren Verlauf durch das Akronym *ISCS* benannt. Ein besonderer Wert wird auf die Zuverlässigkeit der Lösung gelegt, in dem die von Fraunhofer FOKUS entwickelte Bibliothek *Fuzzino* [1], [2], [3] für das Fuzz Testing verwendet wird.

Der Artikel ist wie folgt aufgebaut: Anschließend an die Einleitung werden im nachfolgenden Kapitel Anforderungen an die Integrierte Komponente aufgestellt und erläutert. Im darauffolgenden Kapitel wird die Architektur des Moduls präsentiert, wobei die identifizierten Komponenten sowie die dynamischen Aspekte erläutert und anhand von Sequenzdiagrammen ergänzt werden. Daran anknüpfend wird die Implementierung der Komponente dargestellt, 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. Abschließend wird der aktuelle Forschungsstand zu alternativen Authentifizierungs- und Autorisierungssystemen dargestellt. Dabei wird vor allem die Funktionsweise der Mechanismen fokussiert. Im Fazit werden alle wichtigen Erkenntnisse resümierend zusammengefasst. Ein Ausblick zu noch offenen Fragen schließt den Artikel ab.

Philipp Lämmel: E-Mail: philipp.laemmel@fokus.fraunhofer.de
***Kontaktperson: Nikolay Tcholtchev:** E-Mail: nikolay.tcholtchev@fokus.fraunhofer.de
Ina Schieferdecker: E-Mail: ina.schieferdecker@fokus.fraunhofer.de

II Anforderungsanalyse

In der Anforderungsanalyse werden verschiedene Voraussetzungen erarbeitet, welche die Integrierte Komponente zur Sicherung von Cloud Services (ISCS) erfüllen muss oder sollte. 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 [4]. Eine detailierte Erklärung ist im Appendix angegeben.

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 sollte oder erfüllen 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[1], welche auf die geschützten Services zugreifen 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 sich kein Client 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 nicht mehrere Benutzer 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.

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

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[2] leicht integrierbar sein.

Nicht-Funktionale Anforderungen

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

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 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 ISCS zugreift.

Anforderung 15b Die Komponente sollte benutzerfreundlich für den Endnutzer sein.
Der Endnutzer möchte den Client eines Entwicklers nutzen, wobei es vorkommt, dass er sich 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.

III Architektur

In diesem Kapitel werden die einzelnen Elemente der entwickelten Integrierten Komponente zur Sicherung von Cloud Services (ISCS) aufgezeigt und hinsichtlich ihrer Existenz erläutert. Weiterhin werden die dynamischen Aspekte des Systems visualisiert. Dynamische Aspekte beschreiben die Interaktionen und Informationsflüsse, welche die geforderte Funktionalität innerhalb der Komponente realisieren.

A Identifizierte Komponenten

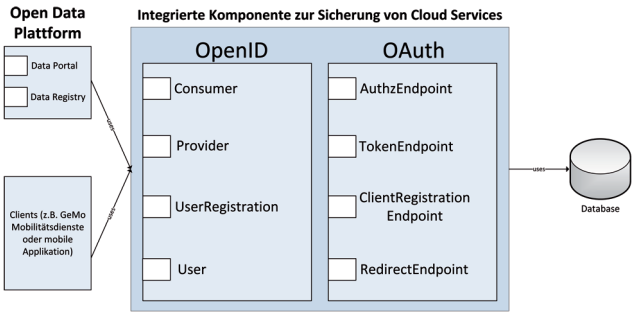

Abbildung 1: Darstellung der Architektur der entwickelten ISCS.

In Abbildung 1 ist die Architektur der integrierten Komponente visualisiert. Dabei sind auf der linken Seite die Open Data Plattform, eine von Fraunhofer FOKUS entwickelte Plattform zur Verwaltung von offenen Daten, und

2 Gängige Systeme sind zum Beispiel CKAN, Liferay et cetera.

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 vorherigen 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 1 sind die Module *UserRegistration* für die Benutzerregistrierung und *ClientRegistrationEndpoint* für die Clientregistrierung[3] 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 1 ist dafür der *TokenEndpoint* innerhalb der OAuth-Komponente vorgesehen. Zusätzlich gibt es einen *RedirectEndpoint*, welcher als Ausweich-Endpoint für Clients ohne 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 1 “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 [5], umgesetzt werden.

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

B 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.

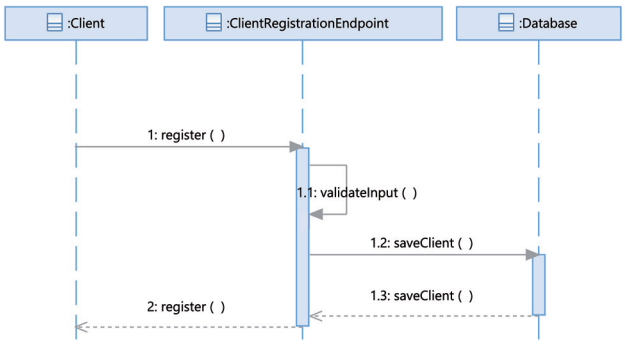

Abbildung 2: Darstellung der Clientregistrierung.

In Abbildung 2 ist die Registrierung eines Clients dargestellt.

Dabei stellt dieser eine Anfrage mit den entsprechenden Parametern an den ClientRegistrationEndpoint, welcher zur OAuth-Teilkomponente gehört. Diese Eingabeparameter werden validiert (Nachricht “validateInput()” in Abbildung 2) und auf Gültigkeit geprüft. Wenn sie gültig sind, werden die Registrierungsinformationen in einer Datenbank gespeichert (Nachricht “saveClient()” in Abbildung 2). 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 diesem Aspekt werden die Anforderungen 1, 2 und 3 erfüllt, welche die wesentlichen Eigenschaften der Registrierung einer Applikation oder eines Dienstes bestimmen.

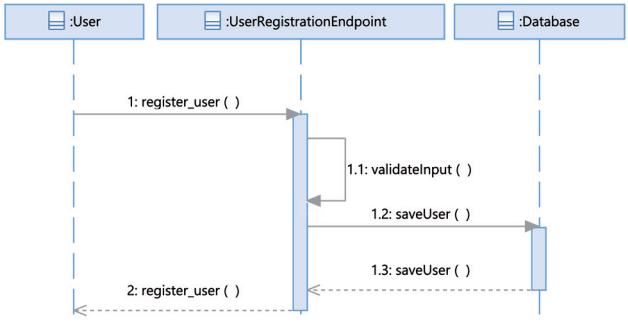

Abbildung 3: Darstellung der Benutzerregistrierung.

In Abbildung 3 ist die Registrierung eines Nutzers verdeutlicht. Dabei stellt dieser eine Anfrage mit den erforderlichen Parametern an den UserRegistration Endpoint, der zur OpenID-Subkomponente gehört. Die Eingabeparameter werden ebenfalls validiert (Nachricht "validateInput()" in Abbildung 3) und auf Gültigkeit geprüft. Die erhaltenen Nutzerinformationen werden in einer Datenbank gespeichert, sofern sie gültig sind (Nachricht "saveUser()" in Abbildung 3). 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.

In der Abbildung 4 sind die typischen Anfragen an die ISCS verdeutlicht. Die Darstellung beginnt mit der Anfrage nach einem Autorisierungscode. Dafür sendet der Client eine Anfrage mit den erforderlichen Parametern an den AuthzEndpoint, welcher zur OAuth-Teilkomponente gehört (siehe Nachricht "requestAuthcode()" in Abbildung 4). Im nächsten Schritt werden die gesendeten Informationen validiert (Nachricht "validateInput()" in Abbildung 4) und erneut auf Gültigkeit getestet. Daran anknüpfend muss der Client vom Nutzer autorisiert werden (Nachricht "askUserForAuthorization()" in Abbildung 4). Nachdem die Autorisierung beendet wurde, wird ein Autorisierungscode ausgestellt, der in der Datenbank gespeichert wird (Nachricht "saveAuthCode()" in Abbildung 4).

Weiterhin ist der Austausch eines zuvor erhaltenen Autorisierungscodes für ein Access Token und ein Refresh Token visualisiert (Nachricht "requestToken()" in Abbildung 4). Dabei wird der Autorisierungscode an den Token-Endpoint, 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 4). 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 "apiCall()" in Abbildung 4). An dieser Stelle lässt der Service die erhaltenen Parameter durch die ISCS validieren (Nachrichten "validateToken()" und "isValidAccessToken()" in Abbildung 4), 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 4 verdeutlicht. Dabei wird die Eingabe erneut validiert und das Refresh Token auf seine Gültigkeit überprüft (Nachrichten "validateInput()" und "isValidRefreshToken()" in Abbildung 4). 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 ISCS das OpenID Protokoll starten und der Nutzer könnte sich mit seinen in der Komponente hinterlegten Identitätsinformationen anmelden, womit Anforderung 11 erfüllt ist.

IV Implementierung der integrierten Komponente rur Sicherung von Cloud Services

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

A Datenbanktechnologien

In diesem Abschnitt werden die Zugriffsmöglichkeiten auf die Datenbanken dargestellt und erläutert.

1) Autorisierungsdatenbank: In Abbildung 5 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 ISCS 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(String)* 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.

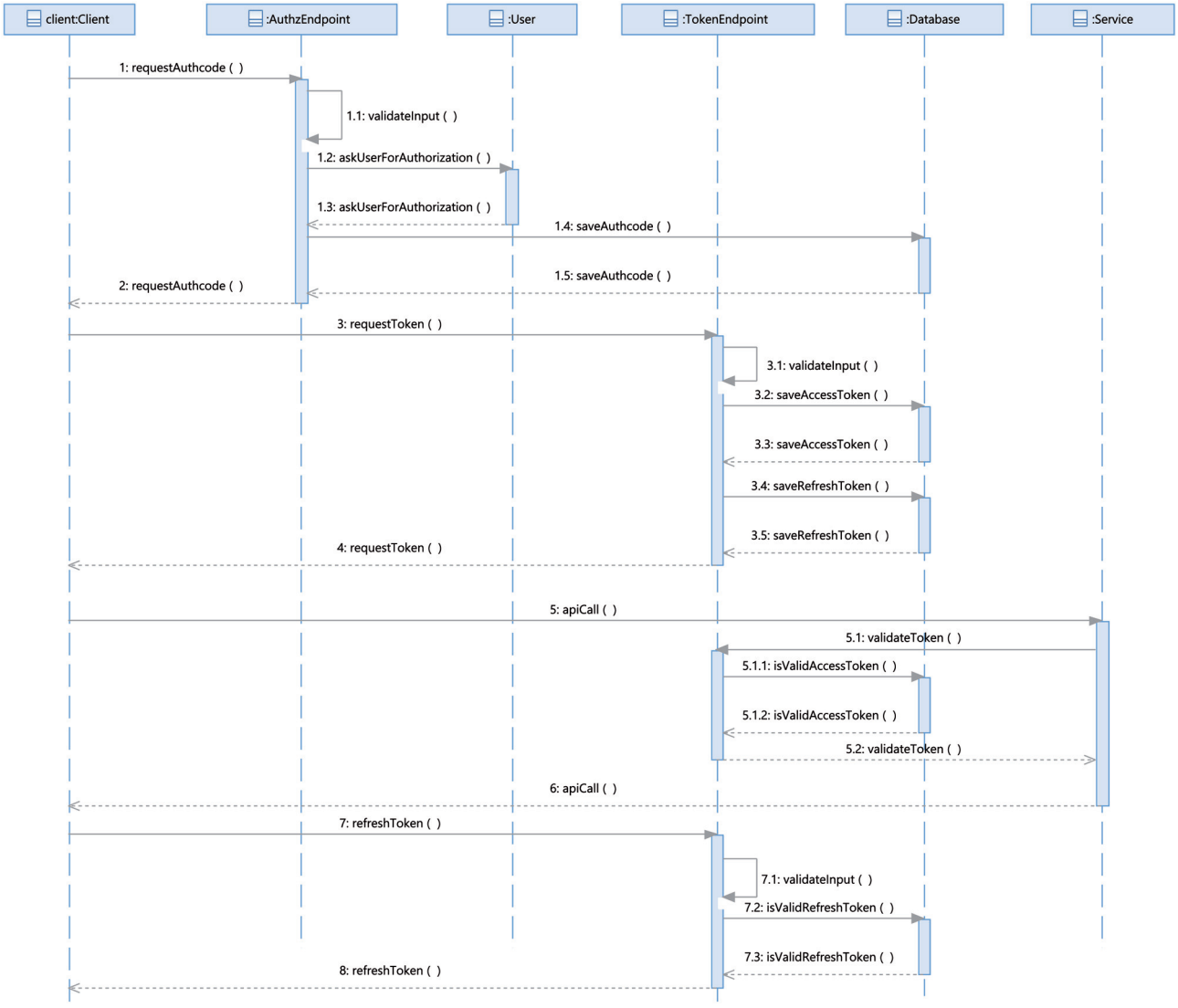

Abbildung 4: Darstellung der typischen Anfragen.

Außerdem sind Methoden für das Speichern der einzelnen Teile (Clients, Autorisierungscode et cetera) der OAuth Spezifikation vorgesehen. Wie in Abbildung 5 erkennbar, gibt es zwei Beispielimplementierungen: *DirectDatabase* und *StorageDatabase*. 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 eines Endpunktes 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. Mit Hilfe einer CDI Konfigurationsdatei (*beans.xml*) wird definiert, welche Implementierung der Schnittstelle *DataAccess* beim Deployment verwendet werden soll. Durch so eine XML Datei wird es sehr einfach, eine eigene Implementierung in die ISCS 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.

2) Benutzerdatenbank: In Abbildung 6 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 6) und aus der Datenbank extrahiert werden können (getUser(String) in Abbildung 6). 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 6). Für die ISCS wurde nur eine Implementierung entwickelt (*UserDAODBImpl*). Anderenfalls würde analog zur Autorisierungsdatenbank in einer CDI Konfigurationsdatei ein Eintrag für die entsprechende Implementierung vorhanden sein. Dadurch ist der Austausch der Implementierung sehr einfach realisierbar.

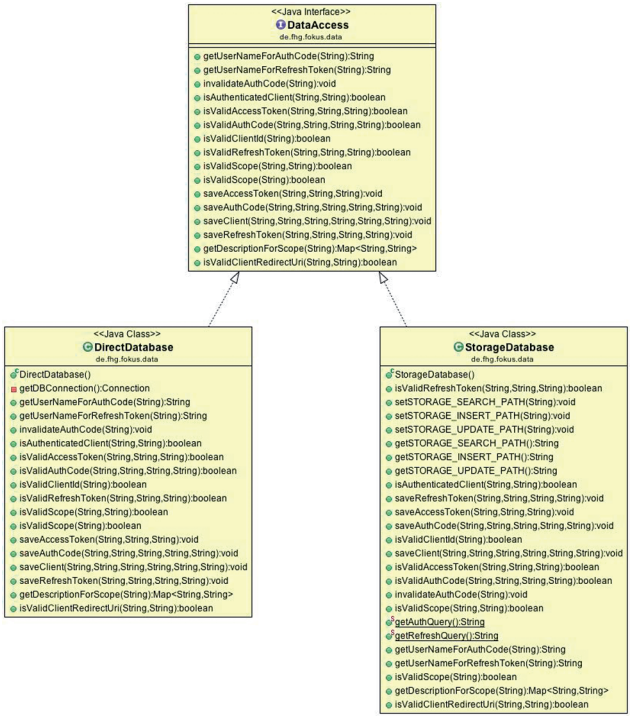

Abbildung 5: Klassendiagramm des Autorisierungsdatenbank Moduls.

Abbildung 6: Klassendiagramm des Benutzerdatenbank Moduls.

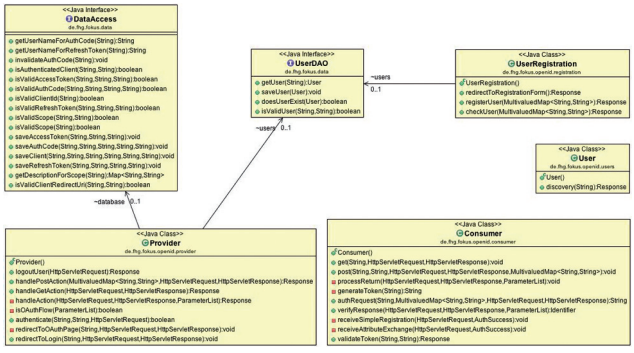

Abbildung 7: Klassendiagramm der OpenID Komponente.

B Umsetzung der OpenID-Subkomponente

Der folgende Abschnitt befasst sich mit der Umsetzung der OpenID-Subkomponente, welche die Authentifizierung eines Endnutzers innerhalb einer verteilten Infrastruktur ermöglicht. In Abbildung 7 ist das Klassendiagram der OpenID-Teilkomponente verdeutlicht. Die Komponente besteht aus den bereits erwähnten Modulen *Provider, Consumer, User* und *UserRegistation*. Dabei greift der *Provider* über definierte Schnittstellen (*DataAccess* und *User DAO*) auf zwei verschiedenen Datenbanken zu, während der *UserRegistration* Dienst über eine wohldefinierte Schnittstelle (*UserDAO*) eine Datenbank anfragt.

1) Consumer: Der *Consumer* ist ein REST Service, welcher in Abbildung 7 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 startet der *Consumer* den Protokollablauf in dem er die erhaltene URL normalisiert. 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 von 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[4] generiert, welche sicherstellt, dass diese Anfrage vom *Consumer* gesendet wurde.

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 und andererseits für die Authentifizierung eines Nutzers während der Autorisierungsanfrage eines Clients verwendet.

Dezentrale Authentifizierung

Die Open Data Plattform beinhaltet ein Datenportal, auf welchem sich potentielle Nutzer Daten anschauen und neue hinzufügen können. 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 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 ausgeführt, 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 sehen. Abermals würde sich der Nutzer authentifizieren, während der *Provider* eine Authentication Response erstellt. Das OpenID Protokoll ware 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 II, die sich auf die leichte Integrierbarkeit des Identitätsmanagements bezieht.

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 7). 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.

Während der Eingabe eines Benutzernamens in das Formular wird der Benutzername 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. Bei den erforderlichen Parametern handelt es sich um einen eindeutigen Benutzernamen sowie Passwort, Vor- und Nachname des Nutzers, 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.

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.

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

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.

C 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 8 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.

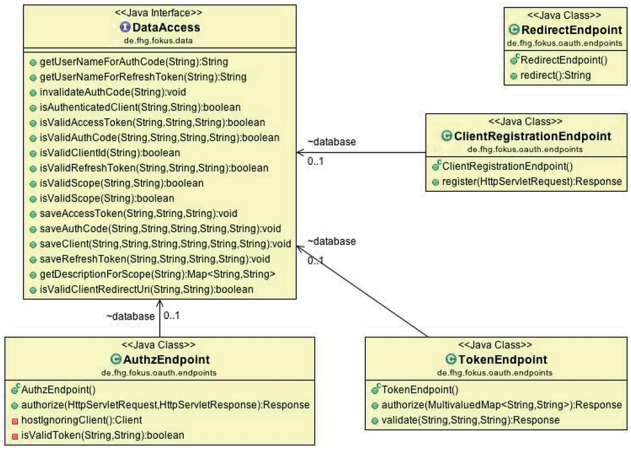

Abbildung 8: Klassendiagramm der OAuth-Komponente.

1) AuthzEndpoint: Der *AuthzEndpoint* ist ein REST Service, welcher in Abbildung 8 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 bestimmten Parametern, welche im *HTTPServletRequest* in Abbildung 8 enthalten sind. 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 [6] 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[5] 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 dargestellt, 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 werden. 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 II realisiert, die sich auf die Autorisierung eines Clients bezieht.

2) TokenEndpoint: Bei dem *TokenEndpoint* handelt es sich um einen REST Service, welcher in Abbildung 8 abgebildet ist. Dieser stellt über einen URL-Pfad die Methode authorize nach außen zur Verfügung.

Der *TokenEndpoint* übernimmt sowohl die Aufgabe des Eintauschs 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.

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

Anfrage mit Autorisierungscode

Bei dieser Anfrage sendet der Client eine HTTP-POST Anfrage mit Parametern, welche vom *TokenEndpoint* validiert werden. Dabei wird, analog zum *AuthzEndpoint*, darauf geachtet, dass die Werte der Parameter gemäß Spezifikation [6] 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 OAuthSub-komponente gestellt wird. 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 II realisiert. Diese Anforderung stellt die Validierung der Parameter sicher.

3) ClientRegistrationEndpoint: In Abbildung 8 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. Bei den erforderlichen Parametern handelt es sich 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 eine eindeutige Client ID sowie ein Client Secret generiert. Außerdem wird daraufhin gespeichert, wann diese Informationen ausgestellt wurden (issued at) und bis wann diese gültig sind. Gleichzeitig werden sowohl die Parameter der Anfrage als auch die zuvor generierten Werte in einer Datenbank persistiert. Als Antwort auf seine Anfrage 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 II erfüllt, welche die Registrierung eines Clients regeln.

4) RedirectEndpoint: Der *RedirectEndpoint* ist ein zusätzlicher REST Service, der in Abbildung 8 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 über die Header und Parameter der HTTP-GET Anfrage iteriert 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.

Abschließend kann betont werden, dass alle erstellten Dienste über wohldefinierte Schnittstellen verfügen, wodurch die ISCS für einen Entwickler benutzerfreundlich ist und Anforderung 15a erfüllt wird. Weiterführend wurde Bootstrap [7] verwendet, welches für die Erstellung von optimierten Webseiten für mobile Applikationen konzipiert wurde. Mit dessen Hilfe konnten übersichtliche Formulare entwickelt werden, wodurch auch eine hohe Benutzerfreundlichkeit für den Endnutzer erzielt werden kann und Anforderung 15b berücksichtigt wird.

V 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 ISCS 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 III-B) angeschaut und auf dessen Basis die Tests sowie die Eingabedaten erstellt wurden.

A Anforderungen

Bezugnehmend auf die Anforderungsanalyse aus dem zweiten 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 ausgeführt, 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 Fehler 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.

B Unit Tests

Mit Hilfe von Unit Tests sollen einzelne Komponenten isoliert getestet werden, wodurch deren Funktionalität überprüft wird. Unit Tests geben eine strikte Anforderung vor, welche vom System erfüllt werden muss. Mit Hilfe dieser Tests wird das im Entwicklungszyklus frühe Finden von Fehlern möglich. Zudem kann der entwickelte 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 Ausdrücken 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.

C 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 dieser der Spezifikation entsprechend aufgebaut ist). Im zweiten Schritt des Testfalls versucht der Client den Autorisierungscode gegen ein Access Token und ein Refresh Token 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.

D 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 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 [1], [2]. 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 [1]. Beim Data Fuzzing ist zu beachten, dass durch vollständig zufällig generierte Eingabedaten zwar immer noch Fehler aufgedeckt werden können, dadurch jedoch auch Nachteile auftreten. Bei vollständiger Randomisierung der Eingabedaten kann davon ausgegangen werden, dass sie ungültig sind. Folglich können solche invaliden Daten leicht vom System entdeckt und abgewiesen werden. Sofern einige Constraints für Eingaben korrekt implementiert sind, können die Daten auch dann abgewiesen werden, obwohl ein Teil der Überprüfungen für Eingabedaten fehlerhaft implementiert ist.

Zum Beispiel könnten Teile der Eingabe bei einer ersten Validierung fälschlicherweise als korrekt eingeschätzt werden, wobei bei einer weiteren Überprüfung die gesamte Eingabe als ungültig identifiziert werden könnte. Daher ist bei diesem Ansatz die Wahrscheinlichkeit Fehler zu finden eher gering [1].

Ein weiterer Nachteil entsteht durch nicht vorhandenes Wissen über den Nachrichtenaustausch. Es ist schwierig ohne Wissen über das Protokoll 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 semivalid 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 ISCS untersucht wurde.

1) Data Fuzzing: Beim Data Fuzzing werden, wie bereits vorgestellt, zufällig generierte Werte an das SUT gesendet. Im Rahmen dieses Artikels wurde die von Fraunhofer FOKUS entwickelte Bibliothek *Fuzzino* zur Generierung randomisierter Werte verwendet. Dabei ist zu beachten, dass die von *Fuzzino* generierten Werte nicht vollständig zufällig sind, sondern auch gezielt mit Hilfe der Datenformatbeschreibungen erstellt werden. Der Vorteil der Verwendung von *Fuzzino* [1], [2], [3] 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 beinhal-

ten 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 lasst, werden mit diesem Generator beliebig lange Strings erzeugt, mit welchen nach Schwachstellen, wie beispielsweise Buffer Overflow, gesucht werden können.
- 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 möglich ist, Schwachstellen bei der Formatierung von Strings aufzudecken.
- SQLInjections: Der Generator *SQLInjections* kreiert Strings, mit welchen es möglich wird, Schwachstellen im Code bezüglich SQL Injection aufzuzeigen.

Zusätzlich zu dieser Bibliothek wurde *JUnit 4*[8] verwendet, welches in der Version 4 ein neues Feature mit dem Namen *Parameterized Tests* bekommen hat. 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 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 String-Formatierung oder SQL Injection ermittelt. Außerdem hat das System stets wie erwartet reagiert. Folglich kann geschlussfolgert werden, dass durch diese Testfälle die Anforderungen 12 und 17 aus dem zweiten Kapitel untersucht und erfüllt wurden. Diese geben an, dass das System stabil (Anforderung 12) und sicher (Anforderung 17) funktionieren soll.

2) Behavioral Fuzzing: Wie bereits angemerkt, sind Fuzzer, welche vollständig randomisierte Daten als Eingabe verwenden, weniger mächtig als jene, welche über Protokollwissen verfügen. Beim Behavioral Fuzzing kann daher auf modellbasierte Fuzzer gesetzt werden, 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 [1].

Im Gegensatz zum Data Fuzzing, bei welchem die Generierung ungültiger Eingabedaten stattfindet, werden beim Behavioral Fuzzing ungültige Nachrichtensequenzen generiert [1]. Da die Implementierung des SUT nicht bekannt sein muss, handelt es sich beim Behavorial Fuzzing um einen Ansatz des Black-Box Testens [2]. Mit Hilfe des Behavorial Fuzzings sollen Schwachstellen in der Software aufgedeckt werden, welche nicht auf ungültigen Eingabedaten basieren, sondern 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 Operatoren des Behavioral Fuzzings angewendet werden, um neue Nachrichtensequenzen zu generieren: beispielsweise durch das Verändern einzelner Nachrichten, dem Ändern der Reihenfolge von Nachrichten sowie dem Ändern des Kontrollflusses, der mit Hilfe von Combined Fragments (zum Beispiel alternative oder loop) beschrieben wird. 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 oder
- an eine anderen Stelle in der Sequenz verschoben werden [2].

Es gibt zwei Möglichkeiten mehrere 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 [2].

Im weiteren Verlauf könnten Combined Fragments verändert werden. Dabei muss 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 [2]. 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 ange-

fügt wird. Dadurch würde eine ungültige Sequenz entstehen [2]. Diese Operatoren konnten in dieser Arbeit jedoch nicht angewendet werden, weil die dadurch gefuzzten Modelle (Testfälle aus UML Sequenzdiagrammen) weiterhin gültig sind.

In einem nächsten Schritt wurde ein typischer Protokollablauf modelliert: Es findet eine Clientregistrierung stattfindet, der Client wird autorisiert und fragt daraufhin mit dem Autorisierungscode nach Token an. Mit diesen Token werden beispielhafte Serviceaufrufe realisiert. Darüber hinaus wird das Access Token mit Hilfe eines Refresh Tokens erneuert. 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 gewählt, 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 mit den bereits dargestellten 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.

Zusammenfassend ist zu sagen, dass mit Behavioral Fuzzing keine Implementierungsfehler gefunden werden konnten, wodurch das Vertrauen in die Stabilität des System erhöht wird. Durch diese Testfälle werden die Anforderungen 12 und 17 erfüllt, welche, wie bereits erwähnt, die Stabilität und Sicherheit der ISCS beinhalten.

E Performance Tests

Abschließend wurden Performance Tests durchgeführt, um zu untersuchen, ob die Anforderungen 13 und 14 aus dem zweiten 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.

1) Laufzeitmessungen: Zu Beginn wurden für die Laufzeitmessungen einige Integrationstests, welche unter anderem bereits beim Data Fuzzing Anwendung fanden, ausgewählt.

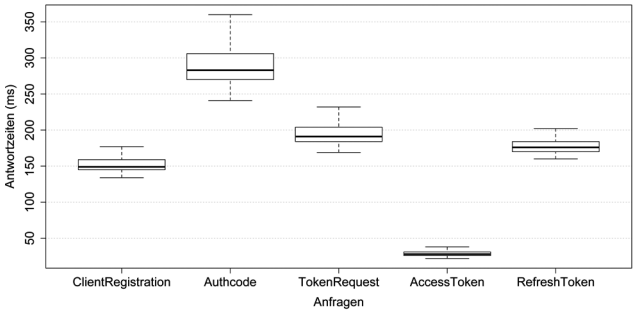

Fig. 9: Laufzeitmessungen bei gültigen Anfragen an die ISCS.

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 9 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 9 Eintrag Authcode) liegen 75 Prozent der Messungen bei einem Wert kleiner als 300 ms. Das ist ein sehr gutes Ergebnis, da dieser Fall bereits die Authentifizierung per OpenID beinhaltet. Wie bereits im Kapitel Implementierung ausgeführt, 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 darauffolgenden Durchläufen geringer als der erste Wert aus, wobei 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 9 Eintrag TokenRequest). 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 nur unwesentlich 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 [9]. Bei einer Antwortdauer von bis zu 1 Sekunde wird der Gedankenfluss des Nutzers nicht unterbrochen, sodass es nicht notwendig wird, zusätzliches Feedback zu versenden [9]. Jedoch würde der Nutzer eine Verzögerung bemerken. Es kann daher betont werden, dass die ermittelten Werte auch im echten Anwendungsfall größtenteils kleiner als 300 ms sind. Dies entspricht einem sehr guten Ergebnis, sodass die ISCS als performant angesehen werden kann und Anforderung 13 erfüllt ist.

2) Speicherbedarf: In diesem Kapitel wurde der Speicherbedarf der ISCS 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 Heapgroßen 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 10, 11 und 12 dargestellt, 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 korrekt, was darauf hinweist, dass die Komponente zum einen nicht negativ durch die gefuzzten Werte beeinflusst wird, was die Stabilität des Systems erhöht. Zum anderen kann angenommen werden, dass keine auf Implementierungsschwächen basierenden Memory Leaks vorhanden sind. Wie im vorangegangenen 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 Schließen der Verbindungen an die Datenbank.

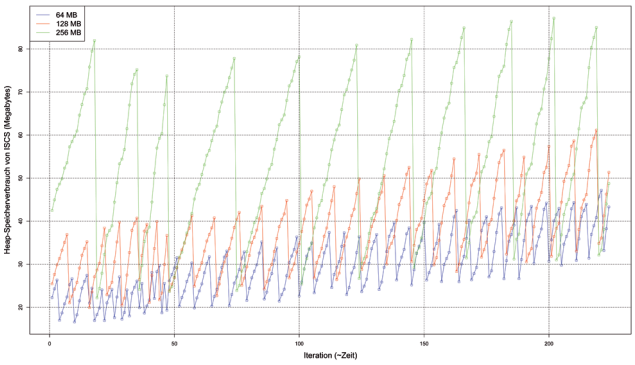

Abbildung 10: Clientregistrierung.

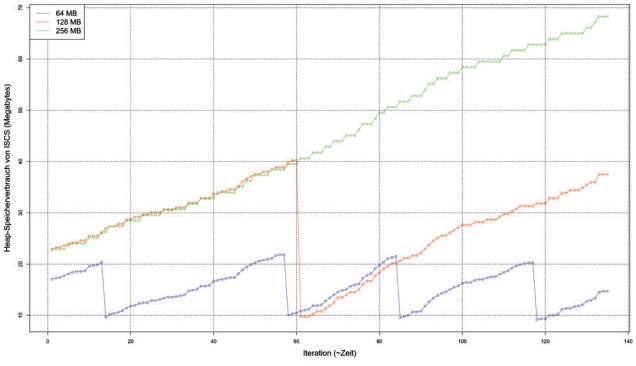

Abbildung. 11: Autorisierungscodeanfrage.

Analog zu den Laufzeitmessungen wurde in einem zweiten Schritt ein Testfall ausgewählt, welcher einen gültigen Aufruf an die Komponente darstellt. Dieser Testfall wurde jeweils 400-mal mit verschiedenen Heapgroßen 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 13 dargestellt. Es ist erneut erkennbar, dass die einzelnen Messungen dem Verlauf einer typischen Java

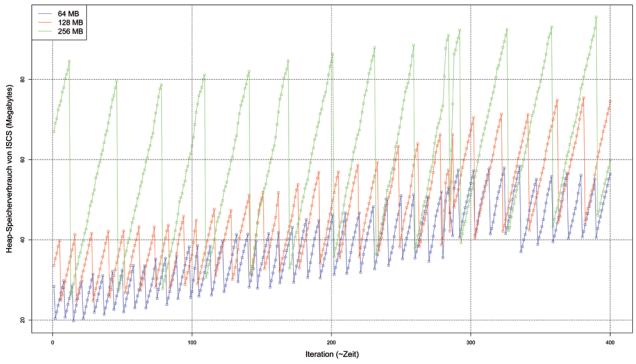

Abbildung 12: Anfrage mit Refresh Token.

Anwendung folgen. Ebenfalls ist zu sehen, dass der Garbage Collector umso ofter aufgerufen werden muss, je geringer die Heapgroße 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 Untersuchungen wird die Anforderung 14 aus dem zweiten Kapitel erfüllt.

Insgesamt wurden 5776 Testfälle entwickelt und generiert.

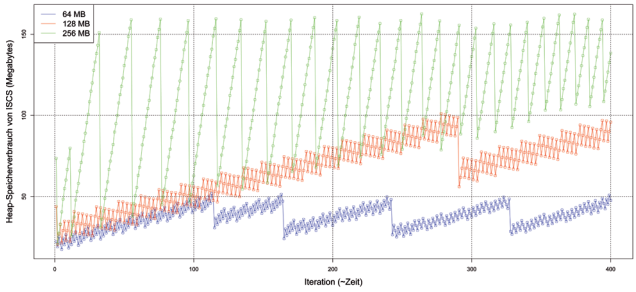

Abbildung 13: Speicherbedarf bei gültigen Anfragen an die Komponente mit verschiedenen Heapgroßen.

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 ISCS 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 konnten.

VI Related work

In diesem Kapitel werden aktuelle Authentifizierungs- und Autorisierungssysteme vorgestellt. Zu Beginn werden die reinen Authentifizierungssysteme 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 beschrieben.

A Kerberos

Kerberos ist ein System, welches aktuell in Version 5 verfügbar ist und ursprünglich am *Massachusetts Institute of Technology (MIT)* entwickelt wurde [10]. Es soll eine sichere Authentifizierung in einem ungesicherten Netzwerk bieten. Kerberos basiert teilweise auf Needham und Schroeders Protokoll zur Authentifizierung [11] und auf Weiterentwicklungen, welche von den Autoren Denning und Sacco vorgeschlagen wurden [12]. Dabei wird im Rahmen von Kerberos eine zentrale Instanz verwendet, welche die Authentifizierung und das Erstellen von Tickets übernimmt.

Für das 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.

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 nicht erforderlich ist. Mit Hilfe eines sol-

chen, im späteren Verlauf generierten Tickets, überprüft ein Dienst, ob dem Client der Zugriff gestattet werden kann.

B Mozilla Persona

Mozilla Persona ist ursprünglich unter dem Namen BrowserID bekannt und wurde von der Mozilla Foundation entwickelt [13]. Es handelt sich um ein dezentrales Authentifizierungssystem, welches 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 lasst [14].

C 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 drei Parteien: den Principal, die Relying Party sowie den OpenID Provider [15]. 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 Authentifizierung erfolgreich war oder fehlgeschlagen ist [15]. Ähnlich wie bei Persona kann ein eigener OpenID Provider erzeugt und verwendet werden.

D Security Assertion Markup Language

Die *SAML* wurde von dem Security Services Committee von der *Organization for the Advancement of Structured Information Standards (OASIS)* entwickelt [16], [17]. 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. Grundsatzlich gibt es drei Parteien: einen Principal, einen Identity Provider und einen Service Provider [17]. 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 [16]. 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.

E Shibboleth

Shibboleth wurde von Internet2/MACE entwickelt und ist ein Authentifizierungs- und Autorisierungssystem für Webservices [18]. Es wird angestrebt, dass sich ein Nutzer einmalig authentifizieren soll, um somit Dienste verschiedener Anbieter nutzen zu können. Shibboleth basiert auf einer Erweiterung des bereits vorgestellten Standards SAML [18]. Dabei besteht Shibboleth ebenfalls aus drei Parteien: dem Identity Provider, dem Service Provider und einem optionalen Discovery Service [19]. 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 Providern hergestellt werden. Diese Komponenten können unabhängig voneinander installiert werden.

F OAuth

OAuth ist ein offener Standard für Autorisierung, welcher aktuell in Version 2.0 vorhanden ist [6]. 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 [6].

VII Fazit und Ausblick

In diesem Artikel wurde eine Integrierte Komponente zur Sicherung von Cloud Services (ISCS) präsentiert. Die Komponente basiert auf dem Standard OAuth und dem Community Standard OpenID. Sie wurde konstruiert, um einen sicheren Zugriff für Dienste beziehungsweise Applikationen und deren Nutzer auf die Cloud zu gewährleisten.

Im Hauptteil dieses Artikels wurde die Entwicklung der ISCS dokumentiert. Dazu wurden im ersten Schritt verschiedene Anforderungen, im Sinne von zu erfüllenden Voraussetzungen, an die Komponente formuliert. Weiterhin wurde die Architektur der ISCS erläutert, wobei die identifizierten Komponenten und dynamische Aspekte (Interaktionen und Informationsflüsse) vorgestellt und hinsichtlich der zu erfüllenden Anforderungen diskutiert wurden. Im Fokus des Hauptteils des Artikels steht die Implementierung der ISCS. 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 Integrationstests, Fuzz Tests sowie Performance Tests durchgeführt wurden. Bei den Fuzz Tests wurde 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 sowie der Speicherbedarf von typischen Anfragen analysiert. Daran anknüpfend wurde der aktuelle Forschungsstand skizziert, wobei mehrere Authentifizierungs- und Autorisierungssysteme hinsichtlich ihrer Funktionsweise, Komplexität und Sicherheitsaspekte vorgestellt wurden.

Hinsichtlich der Komplexität der ISCS 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 angestrebt. Dies konnte durch die Verwendung von OpenID realisiert werden.

Ein weiteres zentrales Ergebnis ist, dass in diesem Artikel eine neuartige Lösung konstruiert wurde, die es in diesem Rahmen noch nicht gibt, da bei den meisten bisherigen Implementierungen die Authentifizierung vernachlässigt wird. Die ISCS ist hinsichtlich Authentifizierung und Autorisierung vollständig entwickelt und kann dabei leicht deployed sowie in bestehende Services und 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 genützten Applikationsserver, beispielsweise *Tomcat* [20] oder *Glassfish* [21], angepasst und ein Filter hinzugefügt wird, welcher die Anfrage an den Dienst abfängt und eine Überprüfung der gesendeten Parameter durch die ISCS veranlasst. Die ISCS kann daher auch im Nachhinein noch in Projekte integriert werden und dort die Zugriffssicherheit auf Dienste gewährleisten.

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 die Sicherheit und Stabilität prüfen, da *Fuzzino* 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 Testings (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 möglichst geringen Overhead bei Anfragen an Dienste erzeugen sollte, sodass 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* [9] handelt es sich dabei um einen sehr guten Wert. Ebenfalls konnte festgestellt werden, dass der Ressourcenbedarf der Software in einem geringen Rahmen liegt.

Die integrierte Komponente wurde in Java entwickelt und basiert auf den Open Source Bibliotheken *openid4java* [22] und *Apache Oltu* [23]. 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.

References

1 M. Schneider, J. Großmann, I. Schieferdecker, and A. Pietschker, "Online model-based behavioral fuzzing," *Software Testing Verification and Validation Workshop, IEEE International Conference on*, vol. 0, pp. 469–475, 2013.

2 M. Schneider, J. Großmann, N. Tcholtchev, I. Schieferdecker, and A. Pietschker, "Behavioral fuzzing operators for uml sequence diagrams," in *System Analysis and Modeling: Theory and Practice*, ser. Lecture Notes in Computer Science, A. Haugen, R. Reed, and R. Gotzhein, Eds. Springer Berlin Heidelberg, 2013, vol. 7744, pp. 88–104. [Online]. Available: http://dx.doi.org/10.1007/978-3-64236757-1_6

3 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.

4 S. Bradner, "Key words for use in rfcs to indicate requirement levels," United States, 1997.

5 "Hibernate," http://hibernate.org/, stand: July 21, 2015.

6 D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749 (Proposed Standard), Internet Engineering Task Force, Oct. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6749.txt

7 "Bootstrap," http://getbootstrap.com/, stand: July 21, 2015.

8 "JUnit," http://junit.org/, stand: July 21, 2015.

9 J. Nielsen, *Usability engineering*. Elsevier, 1994.

10 C. Neuman, T. Yu, S. Hartman, and K. Raeburn, "The Kerberos Network Authentication Service (V5)," RFC 4120 (Proposed Standard), Internet Engineering Task Force, Jul. 2005, updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806. [Online]. Available: http://www.ietf.org/rfc/rfc4120.txt

11 R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, Dec. 1978. [Online]. Available: http://doi.acm.org/10.1145/359657.359659

12 D. E. Denning and G. M. Sacco, "Timestamps in key distribution protocols," *Communications of the ACM*, vol. 24, no. 8, pp. 533–536, 1981.

13 "Mozilla Persona," https://developer.mozilla.org/en-US/Persona, stand: July 21, 2015.

14 "Persona Protocol Overview," https://developer.mozilla.org/enUS/Persona/Protocol_Overview, stand: July 21, 2015.

15 "OpenID Authentication 2.0 – Final," http://openid.net/specs/openidauthentication-2_0.html, stand: July 21, 2015.

16 "OASIS Security Services (SAML) TC," https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security, stand: July 21, 2015.

17 "Online Community for the Security Assertion Markup Language (SAML) OASIS Standard," http://www.saml.xml.org, stand: July 21, 2015.

18 "Shibboleth," http://shibboleth.net/, stand: July 21, 2015.

19 "How Shibboleth Works: Basic Concepts," http://shibboleth.net/about/basic.html, stand: July 21, 2015.

20 "Apache Tomcat," http://tomcat.apache.org/, stand: July 21, 2015.

21 "GlassFish," https://glassfish.java.net/, stand: July 21, 2015.

22 "OpenID4Java," https://code.google.com/p/openid4java/, stand: July 21, 2015.

23 "Apache Oltu," https://oltu.apache.org/, stand: July 21, 2015.

Appendix

Es 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 dürfen. Anforderungen welche unter den Begriffen SOLLTE und SOLLTE NICHT erfasst werden, entsprechen konkreten Empfehlungen. Unter dem Begriff KANN sind demgegenüber fakultative Aufgaben formuliert, deren Umsetzung optional ist.

Philipp Lämmel: Fraunhofer-Institut für Offene Kommunikationssysteme FOKUS, Berlin, Deutschland

Nikolay Tcholtchev: Fraunhofer-Institut für Offene Kommunikationssysteme FOKUS, Berlin, Deutschland

Ina Schieferdecker: Fraunhofer-Institut für Offene Kommunikationssysteme FOKUS, Berlin, Deutschland