

Distributed Discussions in Online Social Networks

Masterarbeit

Florian Müller

Betreuer: Prof. Dr. Max Mühlhäuser

Verantwortlicher Mitarbeiter: Dipl.-Inform. Kai Höver

Darmstadt, September 2013



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Telekooperation
Prof. Dr. Max Mühlhäuser

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, September 2013

(Florian Müller)



Kurzfassung

In der heutigen Zeit verlagert sich unser Leben immer mehr in das World Wide Web. Nicht nur Kommunikation zwischen Freunden, aber auch Diskussionen um Lerninhalte werden auf unterschiedlichsten Plattformen im Web geführt. Gerade Lernplattformen wie Moodle oder soziale Online-Netzwerke sind dazu sehr beliebt geworden. Diese Diskussionen finden allerdings isoliert voneinander statt und Informationen gelangen selten über die Grenzen einer Plattform hinaus. So sind diese mehrfach über das gesamte Web verteilt oder nur an wenigen, versteckten Orten zu finden. Ein Austausch der Daten ist aber durch die unterschiedlichen Datenformate äußerst schwierig.

Aus diesem Grund soll mit dieser Masterarbeit ein System entwickelt werden, dass Beiträge aus verschiedenen Diskussionsquellen untereinander synchronisieren kann. Dazu wird analysiert welche Schritte dazu gemacht werden müssen und welche Komponenten dazu nötig sind. Im Anschluss an diese Analyse wird der hier entwickelte Ansatz der *Social Online Community Connectors* vorgestellt. Auf der Basis des *Resource Description Frameworks* und der Ontologie für *Semantically-Interlinked Online Communities* ist es mit diesem Ansatz möglich, Diskussionen zwischen verschiedenen Plattformen und Datenformaten auszutauschen. Eine Implementierung erfolgt am Ende beispielhaft für die Plattformen Moodle, Canvas, Facebook, Google+ und Youtube.

Abstract

Today, our lives shifts more and more into the World Wide Web. Not only communication between friends, but also discussions about learning content are made in different platforms inside the web. The most popular platforms for these type of discussions are learning management systems like Moodle or online social networks similar to Facebook. However, these discussions are done isolated from each other and the informations are rarely transfered between them. So these information are distributed all over the web multiple times or can only be found on few, different places. Because of the different data formats, the exchange of this information is difficult.

For this reason, a system should be developed that can synchronize discussions of different platforms. A following analysis shows what steps must be taken and what components are needed to achieve this goal. According to this analysis, the developed *Social Online Community Connectors* approach is presented. Based on the *Resource Description Frameworks* and the *Semantically-Interlinked Online Communities* ontology it's possible to exchange discussions between different platforms and data formats. An exemplary implementation of this approach will be done for the platforms Moodle, Canvas, Facebook, Google+ and Youtube.



Inhaltsverzeichnis

1. Einleitung	1
2. Grundlagen	3
2.1. Zugriff auf Webanwendungen	3
2.1.1. Representational State Transfer (REST)	3
2.1.2. Simple Object Access Protocol (SOAP)	4
2.2. Datenintegration	5
2.2.1. Semantic Web	6
2.2.2. Ontologien	9
2.3. Datenaustausch	12
2.3.1. Java Messaging Service	12
2.3.2. Enterprise Integration Patterns (EIP)	14
2.4. Lernplattformen und soziale Online-Netzwerke	16
2.4.1. Moodle	16
2.4.2. Canvas	16
2.4.3. Youtube	17
2.4.4. Facebook	17
2.4.5. Google+	18
2.5. Verwandte Arbeiten und Projekte	18
2.5.1. What happens when Facebook is gone?	18
2.5.2. Reclaim Social	19
3. Analyse	21
3.1. Ein Beispielszenario für die Synchronisation von Beiträgen	21
3.1.1. Datenkonvertierung	21
3.1.2. Datenaustausch	24
3.1.3. Abschluss von Datenkonvertierung und -austausch	24
3.1.4. Privatsphäre	25
3.2. Identifizierung der Komponenten	25
4. Eigener Ansatz: Social Online Community Connectors (SOCC)	27
4.1. Konfiguration	28
4.1.1. SOCC Connector Config Ontologie	29
4.1.2. Services	30
4.1.3. Benutzerdaten	30
4.1.4. Authentifizierung	32
4.1.5. Autorisierung	34
4.2. Design eines Connectors	35
4.2.1. SOCC Context	36

4.2.2. AccessControl	36
4.2.3. ClientManager	38
4.2.4. StructureReader	39
4.2.5. PostReader	40
4.2.6. PostWriter	40
4.3. SOCC-Camel	41
4.3.1. SoccPostPollConsumer	42
4.3.2. SoccPostProducer	42
5. Implementierung und Proof of Concept	45
5.1. Verwendete Programme und Bibliotheken	45
5.2. Implementierung der Connectoren	46
5.2.1. Allgemeine Informationen zum Mapping nach SIOC	46
5.2.2. Allgemeine Probleme und Lösungen bei der Implementierung	47
5.2.3. MoodleConnector	49
5.2.4. CanvasConnector	53
5.2.5. FacebookConnector	58
5.2.6. GooglePlusConnector	64
5.2.7. YoutubeConnector	67
5.3. Implementierung von SOCC-Camel	72
5.4. Proof of Concept	73
5.4.1. Vorbereitungen	74
5.4.2. Ausgangssituation	75
5.4.3. Ablauf der Synchronisation	76
6. Abschlussbetrachtung und Ausblick	81
6.1. Zusammenfassung	81
6.2. Reflektion	82
6.3. Ausblick	83
A. Anhang	85
A.1. SOCC Connector Config Ontologie	85
A.2. SIOC Services Authentication Module	87
A.3. Proof of Concept: Konfigurationsdaten	91
A.4. OAuthTool	94
Literaturverzeichnis	94

Abbildungsverzeichnis

2.1. Einfacher RDF-Graph	8
2.2. Aufbau von SIOC (modifiziert) - Originalquelle: [14]	12
2.3. JMS Programmiermodell - Original Bild: http://docs.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/images/Fig3.1.gif (Zugriff: 2013-09-15) . .	13
3.1. Benutzer erstellt einen Beitrag auf der Webseite A.	21
3.2. Komplexität ohne und mit dem Einsatz eines Zwischenformats - Originalbild: [37]	22
3.3. Daten lesen und in das Zwischenformat konvertieren	23
3.4. Konvertierten des Beitrags in das Format B und schreiben n das soziale Netzwerk B	25
4.1. Übersicht der Komponenten der SOCC	28
4.2. Schema der SOCC Connector Config Ontology	29
4.3. SIOC Services Module	31
4.4. Zusammenhang von Person, UserAccount und Service. Die inversen Eigenschaften sio:account_of und siocs:service_of wurden zu einer besseren Übersicht weggelassen	32
4.5. SIOC Services Authentication Ontology	33
4.6. Basic Access Control Ontologie	35
4.7. PostReader	36
4.9. SOCC Context	36
4.8. UML-Klassendiagramm der Connectoren	37
4.10. AccessControl	38
4.11. ClientManager	38
4.12. StrukturReader	39
4.13. PostReader	40
4.14. UML-Sequenzdiagramm eines PostWriters	41
4.15. Übersicht des SOCC-Camel Moduls als EIP-Diagramm.	42
5.1. Erhalten der Kommentarstruktur beim Schreiben von Beiträgen	48
5.2. Struktur von Moodle und Mapping nach SIOC	51
5.3. Authentifizierungsdaten für Moodle in SIOC	51
5.4. UML Klassendiagramm von CanvasLMS4J	55
5.5. Struktur von Canvas und Mapping nach SIOC	57
5.6. Authentifizierungsdaten für Canvas in SIOC	58
5.7. Struktur von Facebook und Mapping nach SIOC	63
5.8. Authentifizierungsdaten für Facebook in SIOC	63
5.9. Struktur von Google+ und Mapping nach SIOC	67
5.10. Authentifizierungsdaten für Google+ in SIOC	67
5.11. Struktur von Youtube und Mapping nach SIOC	70
5.12. Authentifizierungsdaten für Youtube in SIOC	71

5.13.Proof of Concept: Ausgangssituation	76
5.14.Proof of Concept: Beiträge aus Canvas in Facebook	78
5.15.Proof of Concept: Finales Ergebnis in Canvas	78
A.1. OAuthTool Fenster	94

Tabellenverzeichnis

2.1. Wichtigsten HTTP Operationen mit REST	4
2.2. Einige Beispiel von EIP	15
3.1. Anzahl Konverter bei drei Webseiten	22
4.1. Variablennamen und Ersetzung innerhalb von <code>ccfg:unknownMessageTemplate</code> .	30
5.1. Allgemeine Platzhalter und deren Beschreibung für das Mapping nach SIOC	47
5.2. Format der URIs für Moodle in RDF	52
5.3. Format der URIs für Canvas in RDF	58
5.4. Für SOCC wichtige Facebook Berechtigungen	60
5.5. Format der URIs für Youtube in RDF	68
5.6. Format der URIs für Youtube in RDF	71



1 Einleitung

Durch die Omnipräsenz des Internets im heutigen Alltag haben sich viele Bereiche unseres Lebens sehr verändert. Unter anderem die Art wie wir uns weiterbilden und neue Dinge lernen verlagert sich immer mehr dort hin. Prägend für diesen Umbruch ist der Begriff des „E-Learnings“. Besonders neue Technologien im Zuge des so genannten Web 2.0 wie Blogs, Wikis Diskussionsseiten und sozialer Netzwerke machen es immer leichter neues Wissen zu erwerben und es mit anderen zu teilen. Gerade beim Lernen spielt die Diskussion mit Gleichgesinnten eine wichtige Rolle [15]. Es wurden Studien durchgeführt, die zeigen dass Studenten welche sich an online Diskussionen teilnehmen dazu tendieren bessere Noten zu bekommen, als solche die nicht teilnahmen [13, 31]. Auch für zurückhaltende Studenten ist E-Learning eine Verbesserung, da sie sich so eher an Diskussion beteiligen als zum Beispiel in der Vorlesung oder der Lerngruppe [26].

Qiyun Wang et. al. [38] zeigen in ihrer Studie, dass sich Gruppen im sozialen Netzwerk Facebook für den E-Learning Einsatz als Learning Management System (LMS) gut nutzen lassen. Teilnehmer konnte auf der Gruppenseite durch Kommentare und Chats mit einander diskutieren. Gerade die Organisation der Lerngruppe als auch die Benachrichtigung über Ereignisse funktionierte reibungslos. Jedoch uneingeschränkt konnte Facebook als LMS nicht empfohlen werden. Bemängelt wurde unter anderem die aufwändige Integration von Lernmaterialien „tutor noticed that it was quite troublesome to add teaching materials“ [38, S. 435] und dass es nicht möglich war Diskussionen in einzelne Themen zu unterteilen. Alle Kommentare wurden nur als eine chronologische Liste dargestellt. Seit 2013 ist es aber auch auf Facebook möglich auf Kommentare direkt zu antworten¹ und so sind auch Foren-ähnliche Diskussionen realisierbar. Jedoch ist diese Erweiterung auf „Pages“ beschränkt. Über eine Ausweitung auf Gruppenseiten ist nichts bekannt.

Aber nicht nur soziale Netzwerke sind für Diskussionen innerhalb von E-Learning geeignet, Foren oder Blogs sind ebenfalls sehr beliebte Plattformen. Jedoch ein Problem bei der Nutzung des Internets zum Lernen liegt darin, dass es in der Regel nicht nur eine Plattform genutzt wird, sondern häufig mehrere simultan. Zum Beispiel könnte für einen Kurs ein eigenes Forum im LMS des Veranstalters und nebenbei noch eine Gruppe in Facebook existieren. Teilnehmer, die vorzugsweise nur eine der Plattformen nutzen, erhalten vielleicht von wichtigen Diskussion auf der anderen Plattform keine Kenntnis. Durch diese Inselbildung werden Themen mehrfach behandelt, da Suchfunktionen nur innerhalb der eigenen Plattform suchen und von der Existenz in der Anderen nichts wissen. Eine Integration von zusätzlichen Wissensquellen ist nur schwer möglich und erfolgt immer wieder nur in verbaler Form wie „Schau dir auf der Seite x den Artikel y an“.

Aus diesen Gründen soll in dieser Arbeit ein Ansatz entwickelt werden der es ermöglicht verteilte Diskussionen zusammen zu führen und wiederverwenden zu können. Ein solcher Ansatz muss dazu mehrere Anforderungen erfüllen. Da es sich bei online Plattformen in der Regel um

¹ <https://www.facebook.com/notes/facebook-journalists/improving-conversations-on-facebook-with-replies/578890718789613>

abgeschottete „Datensilos“[4] handelt auf die nur über zum Großteil heterogene Schnittstellen zugegriffen werden kann, ist es hier wichtig eine einheitlich Schnittstelle für den Zugriff auf die gespeicherten Diskussionsdaten zu schaffen. Nicht nur in der Art des Zugriffs unterscheiden sich die einzelnen Plattformen, auch das Format der Daten ist davon Betroffen. Um Diskussionen zwischen den Plattformen überhaupt austauschen zu können ist demzufolge eine Umwandlung in ein gemeinsames Datenformat notwendig, welches erst eine Interoperabilität möglich macht. Als letztes muss noch ein System zur automatischen Synchronisation entwickelt werden wodurch verteilte Diskussionen aktuell gehalten werden können.

Diese Arbeit gliedert sich dazu in folgende Kapitel:

Kapitelbeschreibung

2 Grundlagen

In diesem Kapitel werden einige grundlegende Begriffe und Techniken, die für das weitere Verständnis dieser Arbeit notwendig sind, behandelt. Zuerst wird auf zwei verbreitete Arten von Architekturen für den Zugriff auf Daten im World Wide Web (WWW, oder Web) gezeigt. Darauf folgt eine Einführung in das *Sementic Web* und die Verwendung des *Resource Description Frameworks* (RDF) und Ontologien für die Datenintegration von heterogenen Diskussionsplattformen im Web. Dabei werden die Ontologien des *Friend of a Firend* (FOAF) und des *Semantically-Interlinked Online Communities* (SIOC, ausgesprochen wie „Schock“) Projekts vorgestellt. Im Anschluss wird gezeigt wie das *Java Messaging Service* (JMS) Framework und die *Enterprise Integration Patterns* (EIP) mit dem darauf aufbauenden Apache Camel für den Austausch von Daten über das Versenden und Empfangen von Nachrichten eingesetzt werden kann. Danach folgt eine kurze Beschreibung von fünf Lernplattformen und sozialen Online-Netzwerke, die später in der Implementierung eine Rolle spielen. Abschließend werden noch Projekte und Programme besprochen, die ein ähnliches Ziel wie das in dieser Arbeit vorgestellte System verfolgen.

2.1 Zugriff auf Webanwendungen

Der Zugriff auf Daten von einer Webanwendung ist in den seltensten Fällen durch eine direkte Anbindung an die dahinter liegende Datenbank möglich beziehungsweise gewünscht. Gerade wenn das eigene Geschäft von diesen Daten abhängt, will man nur ungern alles mit allen teilen. Um trotzdem Dritten die Nutzung zu ermöglichen, kann anderen ein Zugriff über eine vordefinierte Programmierschnittstelle (API) gestattet werden. Für Anwendungen und Dienste im Web sind die folgenden zwei Ansätze für die Architektur einer solchen API besonders verbreitet.

2.1.1 Representational State Transfer (REST)

Eine sehr beliebte Architektur für den öffentlichen Zugriff auf Webanwendungen ist der *Representational State Transfer* (REST). REST baut auf dem *Hypertext Transfer Protocol* (HTTP) auf und definiert einige Beschränkungen die eine REST basierter Dienst erfüllen muss.

Die Grundidee besteht darin, dass hinter einer URL eine bestimmte Ressource sich verbirgt, auf die man von außen zugreifen möchte. REST schreibt aber nicht vor in welchen Datenformat diese Ressource übermittelt werden soll, sondern dass das zurückgelieferte Format der Ressource änderbar sein sollte.

„REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the resource.“[19, S. 87]

Dadurch soll einer einfachere Verwendbarkeit mit unterschiedliche Systemen ermöglicht werden. So kann für eine Webanwendung beim aufrufen im Internetbrowser eine HTML-Datei zurück

geliefert werden, die sofort betrachtet werden kann und falls ein Programm darauf zugreifen möchte, wird ein maschinenlesbares Format verwendet. Neben HTML sind auch noch XML und JSON sehr verbreitete Formate. Die Kommunikation wird dabei komplett Zustandslos abgehalten und alle Zustandsinformationen müssen immer mitgeliefert werden. Durch die Zustandslosigkeit skaliert das System viel besser, da Ressourcen sofort wieder frei gegeben werden können und nicht für spätere Anfragen gespeichert werden müssen.

Wie schon beschrieben, nutzen REST basierte Dienste HTTP als Grundlage zur Kommunikation. Die dort definierten Operationen werden mit REST zur Auslieferung und Manipulation der Ressourcen verwendet. Zur Grundausstattung gehören dabei GET, POST, PUT und DELETE (siehe Tabelle 2.1). Die anderen Operationen HEAD, TRACE, OPTIONS und CONNECT sind eher selten anzutreffen (vgl. [19, S. 76]).

Tabelle 2.1.: Wichtigsten HTTP Operationen mit REST

Operation	Beschreibung
GET	Liefert die hinter einer URL liegende Ressource an den Aufrufer zurück.
POST	Dient zum Anlegen einer neuen Ressource. Die URI der neuen Ressource ist beim Aufruf noch unbekannt und wird vom Service bestimmt.
PUT	Wird zum Ändern eine bestehenden Ressource genutzt.
DELETE	Löscht, wie der Name schon sagt, eine Ressource.

2.1.2 Simple Object Access Protocol (SOAP)

Das *Simple Object Access Protocol* (SOAP) ist ein vom W3C standardisiertes Netzwerkprotokoll für den Austausch von Daten zwischen heterogenen Systemen. SOAP schreibt einen bestimmten Aufbau von Nachrichten vor, innerhalb derer die Daten transportiert werden. Als Repräsentation für diese Nachrichten wird hauptsächlich auf XML gesetzt. Bei der Wahl des Transportprotokolls werden dahingegen keine Vorgaben gemacht und es ist frei wählbar. Häufig wird es aber in Verbindung mit HTTP und TCP verwendet.

Listing 2.1: SOAP Nachricht

```
1 <Envelope xmlns="http://www.w3.org/2003/05/soap-envelope">
2   <Header>
3     <!-- header information -->
4   <Header>
5   <Body>
6     <!--body content-->
7   </Body>
8 </Envelope>
```

Eine Nachricht besteht im Grunde aus drei Elementen: den *Envelope*, einen optionalen *Header* und einem *Body* (siehe Listing 2.1). Der *Envelope* fungiert als Briefumschlag für die zu transpor-

tierenden Daten. Innerhalb jedes Envelopes können zusätzliche Meta-Informationen im Header Element gespeichert werden. Die eigentlichen Daten befinden sich im Body Element des Envelopes. Wie der Inhalt von Header und Body auszusehen haben wird von SOAP nicht vorgeschrieben. Dies können weitere XML Elemente oder einfache Zeichenketten sein (vgl. [30]).

Web Services Description Language

Gebräuchlich ist der Einsatz von SOAP für *Remote Procedure Calls* (RPC). Unter RPC versteht man den Aufruf einer Funktion von einem entfernten Dienst und das Zurückliefern einer eventuell vorhandenen Antwort. Welche Funktionen von einem Dienst zur Verfügung stehen wird in einer *Web Services Description Language* (WSDL) Datei beschrieben. Diese WSDL-Datei wird in XML Format verfasst und enthält alle wichtigen Informationen für RPC Aufrufe, die von einem Dienst zur Verfügung gestellt werden. Eine vollständige WSDL-Datei enthält dabei folgende Elemente:

types: Enthält Definitionen von Datentypen die in einer Message eingesetzt werden können. Zur Definition der Datentypen wird das Vokabular von XML Schema¹ eingesetzt.

message: Message-Elemente beschreiben den Aufbau der einzelnen Nachrichten.

portType: Hier wird eine Menge an zur Verfügung stehenden Operationen definiert. Inklusive Eingabe- und Ausgabeparameter. In der WSDL Version 2.0 wurde portType in *interface* umbenannt.

binding beschreibt das Format und den Protokollablauf mehrerer Operationen. Zum Beispiel wie Eingabe- und Ausgabeparameter kodiert werden sollen.

port: Definiert eine Adresse hinter der sich ein Binding befindet. Üblicherweise in Form einer URI. Seit der WSDL Version 2.0 wird statt port der Begriff *endpoint* verwendet.

service Das Service-Element dient zum Zusammenfassen mehrerer Ports zu einem einzigen Dienst.

Wird eine solche WSDL-Datei öffentlich zugänglich gemacht, kann festgestellt werden welche Funktionen ein Dienst anbietet und automatisch APIs für unterschiedliche Systeme generiert werden. Der weitere Datenaustausch erfolgt dann über SOAP-Nachrichten (vgl. [12]).

2.2 Datenintegration

Der Begriff Datenintegration beschreibt das zusammenführen von heterogenen Datenquellen an einen gemeinsamen Ort. Gerade bei der Integration verschiedener, teils kommerzieller Plattformen im Web ist dies ein Problem, da sie zwar ein ähnliches Ziel verfolgen aber sich von anderen abgrenzen wollen. So sind die verwendeten Datenformate in der Regel nicht direkt untereinander austauschbar. Um dies trotzdem zu erreichen, muss ein Mapping (Zuordnung) zwischen den verschiedenen Formaten erfolgen. In dieser Arbeit wird dazu auf RDF und Ontologien wie FOAF und SIOC gesetzt welche im den Folgenden Abschnitten erklärt werden.

¹ <http://www.w3.org/XML/Schema>

2.2.1 Semantic Web

Seit den Anfängen des Webs hat die Masse an abrufbaren Information immer mehr zugenommen, wobei die Vorteile eindeutig in der guten Aktualität und Erreichbarkeit von überall auf Erde liegen. Die Menge an Informationen sind aber auch ein Problem im Web. Da diese überall verteilt sind, ist es schwer für einen einzelnen alles zu einem Thema zu finden. Suchmaschinen wie Google², Yahoo³ oder Microsoft Bing⁴ leisten hier gute Dienste. Doch für Maschinen ist es noch immer nicht einfach die Inhalte von Webseiten zu verstehen, da diese für Menschen gemacht wurden. Auch das Erlernen von neuen Wissen anhand vorhandener Informationen ist in der aktuellen Form des Webs nur schwer realisierbar (vgl. [23]).

2001 machte Tim Berners-Lee (der Erfinder des Webs) den Vorschlag das Web mit maschinenlesbaren Informationen zu erweitern und so die Verarbeitung mit Computerprogrammen zu vereinfachen (siehe [5]). Die Idee des Semantic Webs wurde geboren. Der Inhalt des Webs wird mit semantischen Information so erweitert, dass Programme zum Beispiel erkennen können das zwei oder mehr Texte sich um das selbe Thema drehen ohne erst den Text analysieren zu müssen. Aber auch die Anzeige von impliziten Wissen wäre so einfacher möglich. Zum Beispiel sucht jemand eine Telefonnummer in Los Angeles, USA und will aus Deutschland anrufen. Anhand von zusätzlichen Informationen über die Länder erkennt das Programm, dass zwischen Los Angeles und Deutschland eine Zeitverschiebung von neun Stunden dazwischen liegt. Es wird also vielleicht den Benutzer darüber informieren erst am Nachmittag anzurufen, weil sein Gesprächspartner sonst noch schlafen würde.

Resource Description Framework

Eine Baustein des Semantic Webs ist das Resource Description Framework. Wie der Name schon suggeriert, dient RDF zur Beschreibung von einzelnen Ressourcen innerhalb des Webs. Nach [27, 28] bestand die Motivation bei der Entwicklung von RDF Information über Ressource in einen offenen Datenmodell zu speichern, so dass diese Daten von Maschinen automatisch verarbeitet, manipulieren und untereinander ausgetauscht werden können. Gleichzeitig sollte es auch einfach von jedem erweiterbar sein. Ressourcen stellen dabei alle Dinge dar, über die man eine Aussage treffen kann.

„RDF is designed to represent information in a minimally constraining, flexible way.“[27, Abschnitt 2.1 „Motivation“]

Das Datenmodell von RDF ist zur effizienten Verarbeitung sehr einfach aufgebaut. Die Grundlage bilden Tripel (Eine Menge von genau 3 Elementen) aus Subjekt, Prädikat und Objekt. Einer oder mehrere solcher Triple zusammen werden als gerichteter RDF-Graph bezeichnet. Subjekt und Objekt stehen über das Prädikat mit einander in Beziehung, wobei die Beziehung immer vom Subjekt zum Objekt geht. Das Prädikat wird auch als Eigenschaft (engl. Property) bezeichnet. Gemeinsam beschreibt das Triple immer eine Aussage über eine oder zwei Ressourcen. Zum Beispiel „Die Dose enthält Kekse“ wäre eine Aussage, dass in der Ressource Dose sich andere

² <https://www.google.com>

³ [yahoo.com](https://www.yahoo.com)

⁴ <http://www.bing.com/>

Ressourcen und zwar Kekse befinden. Die „Dose“ ist dabei das Subjekt, „enthält“ das Prädikat und „Kekse“ das Objekt. Ein Triple ist quasi ein einfacher Satz in der natürlichen Sprache. Für die Belegung von Subjekt, Prädikat und Objekt mit Werten werden in RDF *Uniform Resource Identifier* (URI), *Literale* oder *leere Knoten* (im Englischen *Blank Nodes* genannt) verwendet.

URIs sind eindeutige Bezeichner, die eine beliebige reale oder abstrakte Ressource darstellen und werden wie in RFC 2396⁵ beschrieben formatiert. Relative URIs sollten aber nach [27] aber nach Möglichkeit vermieden werden. URIs bilden eine Verallgemeinerung der im Web gebräuchlichen Uniform Resource Locator (URL). Diese Arbeit verwendet aber auch für URLs einheitlich den Begriff URI.

Literale bestehen aus einfachen Zeichenketten die zum Speichern der Informationen dienen. Zusätzlich können Literale mit der Angabe der verwendeten Sprache `Öbjekt"@de` oder des Datentyps `"42"8sd:integer` erweitert werden. Bei Literalen ist darauf zu achten, dass die Literale `Öbjekt"`, `Öbjekt"@de` und `Öbjekt"8sd:string` auf den ersten Blick zwar den selben Wert beschreiben, aber aus Sicht von RDF nicht die selben sind. Sowohl die angegebene Spracht als auch der Datentyp müssen übereinstimmen, um zwei Literale als gleich anzuerkennen.

Leere Knoten werden als alle Knoten im RDF Graphen beschrieben, welche weder eine URI noch ein Literal sind. Sie dienen häufig dazu, um Subjekte zu beschreiben für die nicht unbedingt eine eigene URI nötig ist. Leere Knoten sind nur innerhalb eines Graphen eindeutig, weshalb sie für Referenzierungen von Ressourcen außerhalb des RDF-Graphen ungeeignet sind.

Doch nicht jede dieser drei Belegungen ist in jedem Teil des Tripels erlaubt. Das Subjekt ist entweder eine URI oder ein leerer Knoten, wobei das Prädikat nur eine URI sein kann. Dahingegen ist es beim Objekt möglich eine URI, einen leeren Knoten oder ein Literal zu verwenden.

Repräsentation von RDF-Graphen

In Laufe der Zeit von RDF wurde verschiedene Möglichkeiten erfunden einen RDF-Graphen darzustellen. In diesem Abschnitt werden drei Formen vorgestellt die auch in dieser Arbeit zur Visualisierung benutzt werden. Zuerst die graphische Darstellung für das Visualisieren von RDF-Daten in Abbildung, dann eine Form mit der Extensible Markup Language (XML)⁶ und als letztes mit der platzsparenden Sprache Turtle.

Graphische Darstellung

Graphisch lässt sich RDF als gerichteter Graph mit Knoten und Kanten darstellen. Ressourcen werden dabei als Ellipsen, Literale als Rechtecke und die Prädikate als gerichtete Kante dazwischen gezeichnet. Ein Beispiel ist in Abbildung 2.1 zu sehen. Es verdeutlicht noch einmal den Aufbau von RDF-Dokumenten als gerichteter Graph.

⁵ <http://www.isi.edu/in-notes/rfc2396.txt>

⁶ <http://www.w3.org/XML>

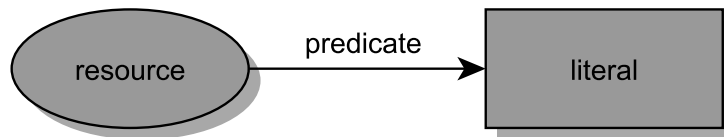


Abbildung 2.1.: Einfacher RDF-Graph

RDF/XML

RDF/XML ist eine verbreitete Form RDF-Dokumente zu beschreiben. Die Basis bildet hierbei die Verwendung von XML. In Listing 2.2 ist ein Beispieldokument in RDF/XML zusehen. Das in Zeile 2 zu sehende `rdf:RDF` ist das Wurzelement für den RDF-Graphen. In diesem Wurzelement werden mit `xmlns:... einige Präfixe für Namensräume definiert, um das Dokument übersichtlicher zu gestalten. Alle Präfixe werden danach mit den angegebenen Namensraum ersetzt. Das Description in Zeile 5 stellt die Beschreibung einer Ressource im RDF-Graphen dar. Die URI der Ressource wird mit dem Attribut rdf:about definiert. Innerhalb des Description-Elements befinden sich die Prädikate. In Zeile 6 steht also, dass die Ressource die Eigenschaft externs:enthaelt hat und auf das Literal Kekse verweist. Wäre das Objekt nicht wie hier ein Literal sondern eine weitere Ressource, könnte man über das Attribut rdf:ressource für das externs:enthaelt URI der Ressource angeben.`

Listing 2.2: RDF/XML Beispiel

```

1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3   xmlns:externs="http://www.example.org/terms#">
4
5   <rdf:Description rdf:about="http://www.example.org/dose">
6     <externs:enthaelt>Kekse</externs:enthaelt>
7   </rdf:Description>
8 </rdf:RDF>

```

Turtle

Turtle⁷ (Ausgeschrieben: *Terse RDF Triple Language*) ist eine weitere Möglichkeit RDF-Graphen darzustellen und ging aus der Sprache N3⁸ (Kurzform für Notation 3) hervor. In Turtle wird das Triple aus Subjekt, Prädikat und Objekt hintereinander geschrieben und zwischen jeden mindestens ein Leerzeichen gelassen. Als Abschluss folgt nach jedem Tripel noch ein Punkt. Der Punkt verdeutlicht noch einmal die Ähnlichkeit mit gesprochenen Sätzen. Listing 2.3 zeigt das Beispiel mit der Keksdose noch einmal in Turtle Notation.

Listing 2.3: Turtle Beispiel

```

1 <http://www.example.org/dose> <http://www.example.org/terms#enthaelt> "
  Kekse" .

```

⁷ <http://www.w3.org/TeamSubmission/turtle/>

⁸ <http://www.w3.org/DesignIssues/Notation3.html>

In Turtle ist darauf zu achten, dass alle URIs immer zwischen spitzen Klammern stehen müssen und Literale werden in Anführungszeichen geschrieben. Da nun einzelne Prädikate beziehungsweise allgemein URIs recht häufig innerhalb eines Graphen auftauchen können, kann es einfacher sein diese abzukürzen. Wie schon in RDF/XML können auch in Turtle Präfixe definiert werden, um Schreibarbeit zu sparen und den RDF-Graphen für Menschen lesbarer zu gestalten.

Listing 2.4: Turtle Präfixe

```
1 @prefix exterm:s: <http://www.example.org/terms#> .  
2 <http://www.example.org/dose> exterm:s:enthaelt "Kekse" .
```

In der ersten Zeile von Listing 2.4 wird durch Einleiten mit dem Schlüsselwort `@prefix` ein neuer Präfix `exterm:s:` für den Namensraum `http://www.example.org/terms#` festgelegt. Dieser Präfix kann nun überall innerhalb des Dokumentes verwendet werden, wobei die spitzen Klammern dann weggelassen werden können.

Listing 2.5: Turtle abkürzende Schreibweise

```
1 @prefix exterm:s: <http://www.example.org/terms#> .  
2 <http://www.example.org/dose> exterm:s:farbe "blau";  
3   exterm:s:enthaelt "Kekse", "Geld" .
```

Listing 2.5 zeigt ein drittes Beispiel, wie redundante Angaben eingespart werden können. Wie man in der zweiten Zeile sehen kann, wird das Triple mit einem Semikolon abschlossen und nicht mit einem Punkt. Durch das Semikolon ist es Möglich das Subjekt mehrfach wieder zu verwenden, wenn sich nur Prädikat und Objekt ändern. So können sich mehrere Eigenschaften einer Ressource platzsparend schreiben lassen ohne das Subjekt immer wieder anzugeben. Ändert sich dahingegen nur das Objekt, können mehrere durch Kommata getrennt hintereinander geschrieben werden. Die dritte Zeile beschreibt zum Beispiel, dass in der Dose nicht nur Kekse sonder auch Geld steckt.

Leere Knoten können dann noch in Turtle durch angeben einer geöffneten eckigen Klammer gefolgt von einer sich Schließenden dargestellt „`[]`“. Zwischen diesen beiden Klammern kann zusätzlich eine Menge von Prädikaten und ihren Objekten stehen, die von Semikolons getrennt werden. Beispiele dazu sind im Anhang A.3 zu sehen. Soll ein leerer Knoten innerhalb eines Graphen referenziert werden, kann er auch als `_:LABEL`, wobei LABEL ein beliebiger Beizeichner ist, geschrieben werden.

2.2.2 Ontologien

Sollen Daten aus verschiedenen Quellen zusammengefügt werden, stellt sich häufig das Problem dass Teile dieser Daten zwar den gleichen Sinn haben, aber aufgrund der Sichtweise des jeweiligen Systems eine andere Bezeichnung besitzen. Dies kann zum Beispiel zu Missverständnissen bei der Verarbeitung führen oder dass Teile eines anderen Systems nicht wiederverwendet werden können (vgl [37]). Einen Ausweg aus diesem Dilemma kann die Verwendung von Ontologien zeigen. Ontologien können allgemein als Wissensbasis [37, 23] bezeichnet werden und liefern eine formale Spezifikation über eine bestimmte Interessensdomäne. Sie beschreibt nicht nur wie das verwendete Vokabular aussieht, sondern legt auch fest welche einheitliche Bedeutung jede Vokabel hat.

Im Bereich des Semantic Webs sind heutzutage zwei Sprachen für die Erstellung von Ontologien verbreitet. Diese sind *RDF Schema* (RDFS)[10] und die darauf aufbauende *Web Ontology Language* (OWL)[32]. Beide Sprachen basieren auf RDF, so können sie zusammen mit jedem System verwendet werden, das RDF versteht. Mit ihnen ist man im Stande abstrakte Klassen von Dingen und deren Eigenschaften zu definieren und diese in eine Vererbungshierarchie einzugliedern. Die Ausprägung einer Klasse wird als Individuum bezeichnet, in dieser Arbeit wird aber Begriff Objekt aus der objektorientierten Programmierung verwendet, um die Überleitung von der theoretischen Beschreibung zu Implementierung einheitlich zu halten. Im Gegensatz zu RDFS können in OWL diverse Einschränkungen definiert werden, wie zum Beispiel, dass eine Eigenschaft nur einmal pro Objekt einer Klasse vorhanden sein darf. Die Anhänge A.1 und A.2 zeigen zwei Ontologien in der Sprache OWL, welche innerhalb dieser Arbeit entwickelt wurden und in Abschnitt 4.1.1 und 4.1.4 beschreiben werden.

Friend of a Friend (FOAF)

Friend of a Friend⁹ ist ein im Jahr 2000 gestartetes Projekt, das versucht Personen innerhalb des Webs, inklusive der Verbindungen zwischen ihnen und anderen, sowie dem was sie machen, in maschinenlesbarer Form abzubilden. FOAF stellt hierzu eine Ontologie [11] (RDF-Präfix `foaf:`) für solche sozialen Netzwerke zur Verfügung. Das Vokabular von FOAF gliedert sich dazu in einen „FOAF Core“ und einen „Social Web“ Bereich. Der Core-Bereich enthält die Klasse `foaf:Agent` für alle Dinge die eine Handlung ausführen können. Also sowohl natürliche Personen, Gruppen oder Organisationen, als auch Computerprogramme oder Maschinen. Für diese gibt es jeweils noch einzelne Unterklassen die von der Klasse `foaf:Agent` erben. Objekten dieser Klassen können eigene Eigenschaften wie einen Namen, ein Alter oder wen sie kennen gegeben werden. Der Social Web Bereich enthält alle Teile die für das Web interessant sind. Das wären zum Beispiel welche E-Mail-Adresse eine Person besitzt, welche Benutzerkonten ihm auf welcher Webseite gehören oder wie die URI seiner Homepage lautet. Das FOAF Projekt sieht sich aber selber nicht als Konkurrenz gegenüber den etablierten sozialen Online-Netzwerken, sondern eher ein Ansatz für einen besser Austausch zwischen den einzelnen Seiten [11, siehe „Abstract“].

Listing 2.6 zeigt ein Beispiel FOAF-Dokument in RDF/XML. Es beschreibt die fiktive Person „Max Mustermann“, mit Vor- und Nachname sowie der Hashwert seiner E-Mail-Adresse (`foaf:mbox_sha1sum`). Diese Person kennt (`foaf:knows`) eine Person mit dem Namen „John Doe“ der ebenso eine E-Mail-Adresse mit den angegebenen Hashwert besitzt. Statt die Eigenschaften von John Doe hier noch einmal vollständig anzugeben, wird über die Eigenschaft `rdfs:seeAlso` auf ein weiteres FOAF-Dokument verwiesen, dass die fehlenden Daten enthält.

Listing 2.6: FOAF Beispiel

```
1 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
3     xmlns:foaf="http://xmlns.com/foaf/0.1/">
4   <foaf:Person>
5     <foaf:name>Max Mustermann</foaf:name>
6     <foaf:firstName>Max</foaf:firstName>
7     <foaf:surname>Mustermann</foaf:surname>
```

⁹ <http://www.foaf-project.org>

```

8      <foaf:mbox_sha1sum>dce4fc922158f8b26fbf0a65ea32bfab58488bd2</
foaf:mbox_sha1sum>
9      <foaf:knows>
10     <foaf:Person>
11       <foaf:name>John Doe</foaf:name>
12       <foaf:mbox_sha1sum>479ea35d3522662b70dc7afd721853485c95db57</
foaf:mbox_sha1sum>
13       <rdfs:seeAlso rdf:resource="http://www.example.org/people/jd/foaf.
rdf"/>
14     </foaf:Person>
15   </foaf:knows>
16 </foaf:Person>
17 </rdf:RDF>

```

Semantically-Interlinked Online Communities (SIOC)

Semantically-Interlinked Online Communities ist ein Projekt, welches von Uldis Bojārs und John Breslin begonnen wurde, um unterschiedliche, webbasierte Diskussionsplattformen (Blog, Forum, Mailinglisten, ...) untereinander verbinden zu können (siehe [14, 8, 7]). Der Kern von SIOC besteht aus einer Ontologie (RDF-Präfix `sioc:`), welche den Inhalt und die Struktur dieser Plattformen in ein maschinenlesbares Format bringen kann und es erlaubt diese auf semantischer Ebene zu verbinden. Auch soll es so möglich sein Daten von einer Plattform zu einer Anderen zu transferieren und so einfacher Inhalte austauschen zu können. Als Basis für SIOC dient RDF, die Ontologie selber wurde in RDFS und OWL designet. Um nicht das Rad neu erfinden zu müssen greift SIOC auf schon bestehende und bewährte Ontologien zurück. Für die Abbildung von Beziehungen zwischen einzelnen Personen wird FOAF und für einige Inhaltliche- und Metadaten (Titel, Inhalt, Erstelldatum, ...) Dublin Core Terms¹⁰ eingesetzt.

Die wichtigsten Klassen von SIOC sind in Abbildung 2.2 in der mittleren Spalte zu sehen. Die Klasse `sioc:Site` ist für die Beschreibung von Webseiten in denen Beiträge innerhalb von Containern verfasst werden. Ein solcher Container ist die Klasse `sioc:Forum` und steht für einen Ort an dem Beiträge geschrieben werden. Enthält ein Forum Diskussionen zu unterschiedlichen Themen, kann es noch einmal in `sioc:Thread` unterteilt werden, welche immer ein Forum als Elternteil haben müssen (`sioc:has_parent`). Beide Klassen leiten sich von der Klasse `sioc:Container`, als einen allgemeinen Ort für Beiträge, ab. Die einzelnen Beiträge werden durch die Klasse `sioc:Post` beziehungsweise von der übergeordneten Klasse `sioc:Item` modelliert. Beiträge gehören in der Regel immer zu einem bestimmten Container oder mindesten zu einer Webseite. Es ist auch möglich Beiträge als Kommentar zu anderen Beiträgen über die Eigenschaft `sioc:has_reply` abzubilden (vgl. [9, S. 203ff]). Jeder Beitrag besitzt mindesten einen Autor der ein Benutzerkonto auf der betreffenden Seite besitzt. Für die Beschreibung eines solchen Benutzerkontos wird die Klasse `sioc:UserAccount` verwendet. Dieses Benutzerkonto kann zusätzlich zu einer Gruppe gehören. Über die Klasse `sioc:Role` kann einem Benutzerkonto eine bestimmte Rolle innerhalb einer Webseite, Forum und so weiter zugeteilt werden. Ein Beispiel dafür wäre die Rolle eines Moderator, der überwacht ob die Regel der Seite in den einzelnen Foren eingehalten werden.

¹⁰ <http://dublincore.org/documents/dcmi-terms>

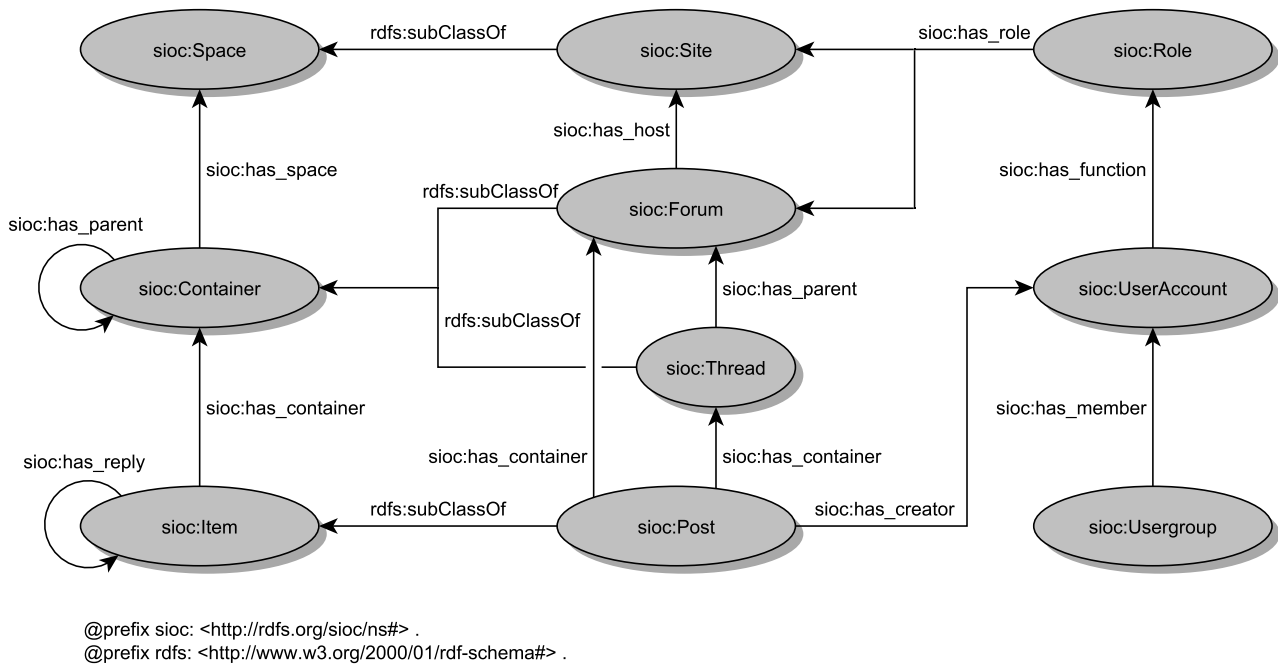


Abbildung 2.2.: Aufbau von SIOC (modifiziert) - Originalquelle: [14]

2.3 Datenaustausch

Sollen zwischen zwei unabhängigen Systemen Daten ausgetauscht werden, hat man im Allgemeinen mehrere Ansätze dieses Problem zu lösen. Beide Programme könnten Dateien mit den Daten an einen festen Ort speichern, die der jeweilig andere lesen kann. Auch ist das Benutzen eines gemeinsamen Datenbankmanagementsystems oder die Verwendung von RPCs denkbar. Ebenso ist der Datenaustausch mit Nachrichten über ein Netzwerk ein sehr flexibler und sicherer Weg. Jeder dieser Ansätze hat seine eigenen Vor- und Nachteile und sollte am Besten nach den Anforderungen für den Verwendungszweck gewählt werden.

Dieser Abschnitt beschäftigt sich explizit nur mit dem Versenden und Empfangen von Nachrichten, da dies für das in der Einleitung beschriebene Wunschesystem die meisten Vorteile bringt. Dazu wird zuerst das *Java Messageing Service* (JMS) vorgestellt und danach die Enterprise Integration Patterns (EIP) mit Apache Camel.

2.3.1 Java Messaging Service

Das JMS ist eine Sammlung von Schnittstellen für das Erstellen, Senden und Empfangen von Nachrichten zwischen Programmen. JMS erlaubt eine Entwicklung von verteilten Anwendung die nicht nur lose gekoppelt, sondern auch asynchron und zuverlässig arbeiten (vgl. [35]).

Die Grundstruktur einer JMS Anwendung besteht aus einem *JMS Provider*, der die Schnittstellen von JMS implementiert. Dieser JMS Provider wird auch als *Message Oriented Middleware* (MOM) bezeichnet und kümmert sich auch darum, dass Nachrichten zuverlässig verschickt werden. Hinzu kommt ein JMS Client von dem Nachrichten verschickt und empfangen werden und die Nach-

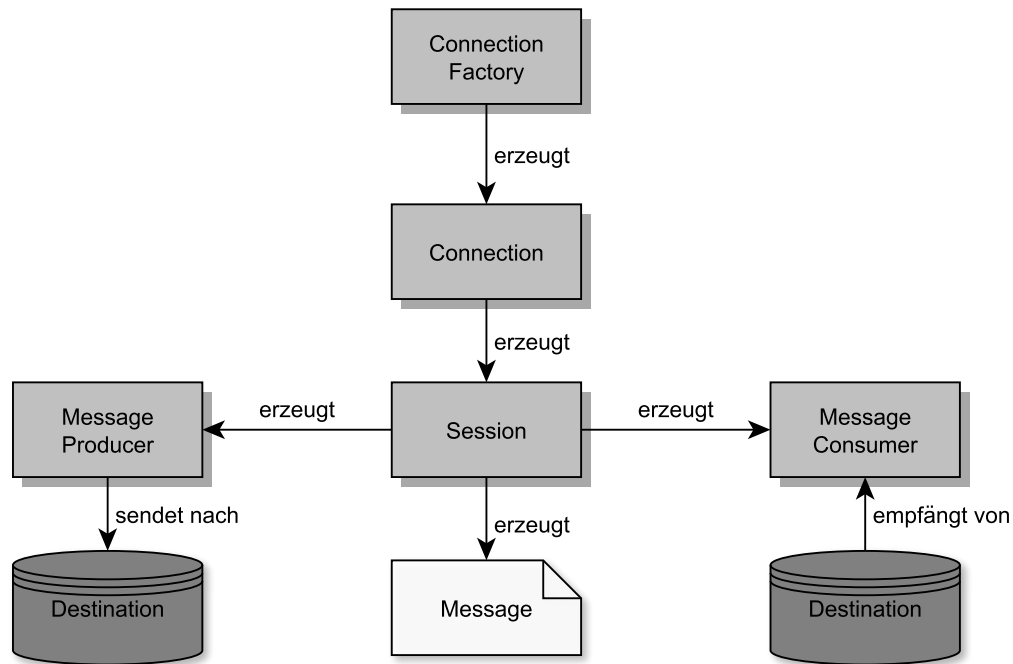


Abbildung 2.3.: JMS Programmiermodell - Original Bild: http://docs.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/images/Fig3.1.gif (Zugriff: 2013-09-15)

richten selber sowie sogenannte *Administered Objects*. Diese Administered Objects bestehen aus vorkonfigurierten *ConnectionFactory*s für das Erstellen von Verbindungen (*Connection*) zwischen Client und Provider und *Destinations* als Sende- und Empfangsendpunkte von Nachrichten. Die Administered Objects können im Client über das Java Naming and Directory Interface (JNDI)¹¹ API abgefragt werden. Alle Clients die nicht die JMS API sondern die implementierte API der MOM direkt verwenden, werden *Native Client* genannt.

JMS unterstützt zwei Verbindungsarten zum Übertragen von Nachrichten: Queue- und Topic-basiert. Als Queue-basiert wird eine Punkt-zu-Punkt Verbindung bezeichnet. Hier werden Nachrichten nur zwischen zwei Clients übertragen und gegebenenfalls in einer Warteschlange zwischengespeichert. Hinter Topic-basiert verbirgt sich ein Publish-Subscribe-Mechanismus bei dem ein Client Nachrichten an eine Topic-Destination schickt und andere Clients sich auf dieses Topic anmelden können, um die dann die Nachrichten des ersten Clients zu empfangen. Ob Queue oder Topic-basiert, wird über die Art der Destination ausgewählt.

Sollen nun Nachrichten von einem Client verschickt beziehungsweise empfangen werden, muss mit einer *ConnectionFactory* eine neue *Connection* zu einer MOM aufgebaut werden. Mit dieser *Connection* wird danach eine *Session* erstellt, das als Kontext zum Senden und Empfangen verwendet wird. Für das versenden von Nachrichten muss mit dem Sessionobjekt ein *MessageProducer* erstellt werden. Für das Empfangen entsprechend einen *MessageConsumer*. Abbildung 2.3 zeigt noch einmal den Zusammenhang aller JMS Komponenten.

Listing 2.7: JMS Beispiel

```
1 Context ctx = new InitialContext();
```

¹¹ JNDI API: <http://www.oracle.com/technetwork/java/index-jsp-137536.html>

```
2 ConnectionFactory connectionFactory = (ConnectionFactory) ctx.lookup("
    ConnectionFactory");
3
4 Connection connection = connectionFactory.createConnection();
5 connection.start();
6
7 Session session = connection.createSession();
8 Destination destination = session.createTopic("topic-test");
9
10 MessageProducer msgProducer = session.createProducer(destination);
11 Message msg = session.createTextMessage("Hallo World!");
12 msgProducer.send(msg);
```

Listing 2.7 zeigt ein kleines Beispiel zum Senden einer Textnachricht mit JMS. Die erste und zweite Zeile zeigt wie ein JNDI Kontext erstellt und nach einer vordefinierten ConnectionFactory mit dem Namen „ConnectionFactory“ gesucht wird. Mit dieser ConnectionFactory wird dann eine neue Connection erstellt und gestartet. Danach folgt in Zeile 7 und 8 das Erstellen einer neuen Session und die Definition eines Topics mit dem Namen „topic-test“ als Destination. In der zehnten Zeile wird dann der MessageProducer zum Senden von Nachrichten und in der Folgezeile die zu sendende Textnachricht erzeugt. Diese wird dann in der letzten Zeile vom MessageProducer an das Topic verschickt.

2.3.2 Enterprise Integration Patterns (EIP)

Bezeichnungen wie „Iterator“, „Factory Method“, „Observer“ oder „Proxy“ hat bestimmt schon jeder Programmierer mindestens einmal gehört. Hierbei handelt es sich um sogenannte Entwurfsmuster für Softwareprogramme. Sie sind Schablonen für Lösungen von Problemen, die in der Entwicklung von Software immer wieder auftreten und sich als hilfreich erwiesen haben. Auch in der Integration von verschiedenen Geschäftsanwendungen in ein System treten solche Muster immer wieder auf. Gregor Hohpe und Bobby Woolf beschreiben in ihrem Buch „Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions“^[24] eine Vielzahl solcher EIP für die Integration mittels MOM. Alle hier aufzuzählen würde jedoch den Rahmen dieser Arbeit sprengen. Aus diesem Grund werden in Tabelle 2.2 fünf Muster inklusive der verwendeten Symbole vorgestellt, die später noch vorkommen werden. Die restlichen Muster sind auf der Webseite der EIP¹² zu finden.



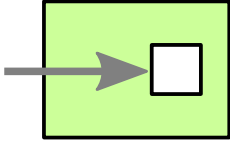
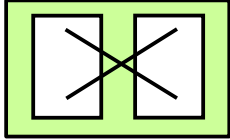
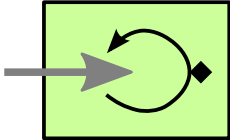
Apache Camel

Apache Camel^[2] (kurz: Camel) ist ein Projekt der Apache Software Foundation (kurz: Apache) für das Routen und Verteilen von Nachrichten zur Integration von Systemen auf Basis von definierten Regeln. Camel stellt dazu eine Java-basierte API für den Einsatz der EIP bereit. Die Regeln für die Routen, die Nachrichten nehmen können, können direkt in Java, aber auch durch das Spring Framework¹³ in XML definiert werden.

¹² <http://www.enterpriseintegrationpatterns.com/toc.html>

¹³ <http://www.springsource.org/>

Tabelle 2.2.: Einige Beispiel von EIP

Icon	Name	Beschreibung
	<i>Message</i>	Über Nachrichten werden Daten zwischen zwei oder mehr Systemen ausgetauscht.
	<i>Channel</i>	Ein Channel beschreibt einen Nachrichtenkanal über den Nachrichten von einem System in ein anderes verschickt werden können.
	<i>Endpoint</i>	Endpoints sind Schnittstellen in einem System, von dem Nachrichten in einen Kanal gesendet oder von dort Empfangen werden.
	<i>Message Translator</i>	Nicht immer liegt eine Nachricht im richtigen Format für ein System vor. Durch Message Translators können diese in das gewünschte Format konvertiert werden.
	<i>Polling Consumer</i>	Manchmal möchte ein Programm Nachrichten nicht verarbeiten wenn sie ankommen, sondern dann wenn es bereit dazu ist. Ein PollingConsumer fragt also erst zu bestimmten Zeitpunkten ab, ob Nachrichten für in vorliegen.

Listing 2.8: Apache Camel Beispiel

```

1 RouteDefinition rd = new RouteDefinition()
2   .from('timer://helloTimer?period=3000')
3   .to('log:helloLog');
4
5 CamelContext camelContext = new DefaultCamelContext();
6 camelContext.addRouteDefinition( rd );
7 camelContext.start();

```

Listing 2.8 zeigt ein Beispiel, wie eine Nachrichtenroute in Java für Camel definiert werden kann. In der ersten Zeile wird über die Klasse `RouteDefinition` eine neue Route erstellt. Über die Methode `from` wird der Endpunkt, vom dem die Route ausgeht, festgelegt. Welcher Endpunkt das genau sein soll kann auf zwei Arten definiert werden. Man übergibt der Methode direkt ein Objekt einer Klasse die das Interface `Endpoint` implementiert oder man macht es, wie hier im Beispiel, über eine URI. Diese URI baut sich auf folgende Weise zusammen: Der Teil bis zum Doppelpunkt, das sogenannte „Schema“, legt die Komponente (engl. Component) fest, von der ein Endpunkt erzeugt werden soll. In diesem Falls ist es eine Timer-Komponente, deren Endpunkt im periodischen Abstand Event-Nachrichten verschickt. Der Rest der URI wird zur Konfiguration an den Endpunkt übergeben. „helloTimer“ steht hier für einen Namen für den Timer und der Parameter „period“ gibt den zeitlichen Abstand zwischen zwei Event-Nachrichten an. Das Ziel der Route wird mit der Methode `to` in Zeile 3 festgelegt. Für das Ziel wird hier eine Log-Komponente

mit dem Namen „helloLog“ festgelegt, die alle reinkommenden Nachrichten protokolliert. In der fünften Zeile wird ein CamelContext Objekt, das für die Verwaltung und Ausführung der Routen verantwortliche ist, erstellt. Die Route wird dann dem CamelContext hinzugefügt und die Ausführung in der letzten Zeile gestartet. Nun wird der Timer alle 3000 Millisekunden eine neue Event-Nachricht erzeugen und an den CamelContext schicken. Dieser leitet die Nachricht dann an das durch die Route definierte Ziel und wird dort von der Log-Komponente ausgegeben.

Für Camel existieren schon eine Menge vorgefertigter Komponenten¹⁴ die ein großes Spektrum an Anwendungsmöglichkeiten abdeckt. Es gibt Komponenten für die Anbindung an HTTP-Webserver, JMS-Provider, E-Mail-Server, RSS/Atom-Feeds und viele mehr.

2.4 Lernplattformen und soziale Online-Netzwerke

An dieser Stelle sollen kurz einige Lernplattformen und soziale Online-Netzwerke vorgestellt, die im späteren Verlauf dieser Arbeit für die Implementierung verwendet wurden. Im Einzelnen waren dies *Moodle*, *Canvas*, *Facebook*, *Google+* und *Youtube*, da sie einen guten Schnitt von den Plattformen bilden, die heutzutage sowohl im Bereich E-Learning als auch von der breiten Masse verwendet werden.

2.4.1 Moodle

Moodle¹⁵ ist ein weit verbreitetes Open Source Online LMS. Die Hauptaufgabe liegt im Verwalten von Online-Kursen im Bereich E-Learning. Hierzu bietet Moodle von Haus aus eine große Menge an Funktionen für die Verwaltung des Kurses und die Kommunikation zwischen Lehrenden und Lernenden. Es bietet die Möglichkeit Aufgaben an die Teilnehmer zu verteilen, Fragebögen zu erstellen, zusätzlichen Kursmaterialien bereitzustellen und den Lernerfolg durch Benotung und Feedback zu kontrollieren. Funktionen für die Unterstützung des kollaborativen Lernens sind ebenfalls vorhanden. Teilnehmer können Lerngruppen bilden, sich über persönliche Nachrichten austauschen, gemeinsam an Wikis arbeiten oder in Foren diskutieren.

Moodle wurde in der Programmiersprache PHP geschrieben und unterstützt die Datenbanken MySQL, PostgreSQL, MSSQL und Oracle. Die Installation von weiteren Funktionalitäten ist durch von Dritten geschriebenen Erweiterungen möglich. Seit Version 2.0 können für Moodle auch Webservices installiert werden. So können auch externe Anwendungen auf interne Funktionen und Daten zugreifen.

2.4.2 Canvas

Das von der Firma Instructure¹⁶ entwickelte *Canvas* ist ein recht neues, unter Open Source Lizenz gestelltes LMS. Vom Funktionsumfang ist es Moodle nicht unähnlich. Es existiert eine Verwaltung einzelner Kurse. Innerhalb dieser Kurse können in einem Forum Diskussionen geführt

¹⁴ <http://camel.apache.org/components.html>

¹⁵ <https://moodle.org/>

¹⁶ <https://www.instructure.com>

und Lernmieteralien hoch- und heruntergeladen werden. Verteilung von Aufgaben, deren Benotung und ein Benachrichtigungssystem existiert ebenfalls. Canvas erlaubt auch das Einbinden von externen Diensten zum kollaborativen Lernen und Arbeiten wie Google Docs¹⁷ oder der Webkonferenz Anwendung BigBlueButton¹⁸.

Canvas wird mittels des Webframeworks *Ruby on Rails*¹⁹ entwickelt. Das Aussehen ist etwas moderner, als das von Moodle und es wird sehr stark auf die neuesten Webtechnologien wie HTML5 CSS3 und JQuery gesetzt. Eine Erweiterung der Funktionalität von Canvas ist durch das einbinden von Programmen möglich, die den *Learning Tools Interoperability*[™] (LTI) Standard erfüllen. Einige solcher Programme finden sich auf der Webseite <https://www.edu-apps.org>. Unter anderem Programme zum Suchen und Einbinden von Youtube Videos, Wikipedia Artikeln, GitHub Gists²⁰ und vielen weiteren.

2.4.3 Youtube

Youtube²¹ gehört wohl heute zu den beliebtesten Anlaufpunkten im Internet, wenn es um das Thema Videos geht. Monatlich nutzen über 1 Milliarde Nutzer die Seite und pro Minute werden 100 Stunden neuer Videos hochgeladen (siehe [39]). Doch nicht das komplette Videomaterial besteht aus Katzen, Musik oder Unfallvideos. Ein Teil der Benutzer, die eigene Videos hochladen, wollen anderen Dinge beibringen, weil es in ihr Interessengebiet passt oder früher selber Probleme damit hatten. Einer erklärt die Logarithmengesetze, ein anderer wie man Feuer ohne Feuerzeug macht und eine ganz andere gibt Schönheitstipps. Youtube ist also auch im E-Learning Bereich gut einsetzbar. Lehrende können eigene Videos hochladen, von anderen interessante Videos in Playlisten zusammenfassen und die Lernenden können über Kommentare Fragen zum Inhalt stellen.

2.4.4 Facebook

Das soziale Online-Netzwerk Facebook²² kann mit rund 699 Millionen aktiven Benutzern täglich zu den aktuell beliebtesten Vertretern seiner Art bezeichnet werden (siehe [17]). Facebook erlaubt es, wie alle sozialen Online-Netzwerke, bekannte Personen in Freundeslisten zusammenzufassen und mit ihnen private Nachrichten auszutauschen. Beiträge wie Texte, Fotos oder Videos können auf einer Art Pinnwand, der „Wall“, öffentlich oder nur mit Freunden geteilt werden. Benutzer mit gemeinsamen Interessen können dazu eigene Gruppen bilden und dort auf einer eigenen Wall Beiträge veröffentlichen oder die anderer kommentieren. Wie in der Einleitung schon erklärt zeigt Qiyun Wang et. al. [38] das sich Facebook, wenn auch mit Einschränkungen, wunderbar zur Verwaltung und Nutzung durch Lernkurse und Lerngruppen eignet. Die gleiche Erfahrung teilte Anthony Fontana [20, 18], der Facebook als Alternative zum bestehenden System der Bowling Green State University in Ohio, USA verwendete.

¹⁷ <https://drive.google.com>

¹⁸ <http://www.bigbluebutton.org>

¹⁹ <http://rubyonrails.org>

²⁰ <https://gist.github.com>

²¹ <https://www.youtube.com>

²² <https://www.facebook.com/>

2.4.5 Google+

Google+²³ ist ein 2011 von Google gestartetes soziales Online-Netzwerk. Seit Anfang 2013 ist Google+, von der Anzahl der aktiven Benutzer her gesehen, auf Platz 2 hinter Marktführer Facebook (siehe [36]). Vom Funktionsumfang sind sich beide sehr ähnlich. Auf Google+ können andere Benutzer in sogenannten „Circles“ sortiert werden, welche den auf Facebook genutzten Freundeslisten entspricht. Jeder Benutzer hat einen eigenen „Stream“ in dem er Beiträge öffentlich oder nur für ein oder mehrere Circles verfassen kann. Das Gründen von Gruppen für bestimmte Interessensbereiche ist auch in Google+ möglich und werden dort als „Communities“ bezeichnet. Eines der interessantesten Funktionen von Google+ dürfte die Einführung von „Google Hangout“ sein. Hier können Benutzer neben Chats auch Videokonferenzen mit bis zu zehn anderen abhalten, ohne einen externen Service wie Skype²⁴ zu nutzen. Diese Funktion wäre gut für den Einsatz in E-Learning nutzbar. Ein Tutor könnte so in kleiner Runde Fragestunden oder Gruppen Treffen abhalten.

2.5 Verwandte Arbeiten und Projekte

Leider gibt es so gut wie kein Projekt, dass sich direkt mit den Austausch von Diskussionsbeiträgen zwischen unterschiedlichen Plattformen befasst. Ein dieser der doch existierenden Projekte wurde weiter oben mit SIOC in Verbindung mit FOAF schon vorgestellt. Nichtsdestotrotz existieren Arbeiten und Projekte die sich damit befassen, wie jeder seine Daten, die er irgendwann und irgendwo in einen sozialen Online-Netzwerk hinterlassen hat, wieder zurück unter seine Kontrolle bringt. Von diesem Punkt ist es nicht mehr weit, diese Daten wieder in eine andere Plattform zu übertragen, falls dies gewünscht ist.

2.5.1 What happens when Facebook is gone?

Frank McCown und Michael L. Nelson beschreiben in ihrem Bericht „What happens when Facebook is gone?“[29], wie Möglichkeiten aussehen können, die unsere Daten von sozialen Online-Netzwerken (hier im speziellen Fall von Facebook) für uns und die Nachwelt archivieren können. Zum Beispiel, wenn eine Person einen großen Teil seines persönlichen Lebens auf Facebook verbringt und plötzlich stirbt. Wie sollen seine Angehörigen an nicht öffentliche Texte, Bilder, Videos heran kommen, wenn sie in der Regel keinen Zugriff auf das Benutzerkonto haben, da der Verstorbene so etwas nicht vorhersehen konnte. Oder wenn ein Benutzer mit seinen Daten in ein anderes soziales Online-Netzwerk umziehen will, sei dies bei Facebook (zum damaligen Zeitpunkt) nur schwer möglich.

„It is also likely he was not prepared to die at such a young age, and much of his personal life, which lies in the digital “cloud“, may never be accessible to his loved ones“ [29, S. 251]

Zum Anlegen eines solchen Archivs wurden mehrere Ansätze vorgestellt. Die einfachste Ansatz wäre die E-Mail-Benachrichtigung zu aktivieren und alle neuen Beiträge in einem E-Mail-Postfach

²³ <https://plus.google.com>

²⁴ <http://www.skype.com/>

zu sichern. So können aber nur alle neuen Beiträge erfasst werden. Alte bleiben weiterhin in Facebook. Eine sehr aufwändige Möglichkeit wäre es Bildschirmfotos von den Beiträgen zu machen und diese durch ein Texterkennungsprogramm laufen zu lassen. Die dadurch erzeugten Dateien können dann in einer Datenbank gespeichert werden. Heutige Internetbrowser bieten es zusätzlich zum Anzeigen von Webseite auch das Herunterladen selbiger an. Dabei wird die HTML-Datei inklusive aller darin enthaltenen weiteren Dateien wie Bilder, Videos und CSS-Dateien gespeichert. Die so archivierte Seite hat dann im beschränkten Umfang genau das gleiche Aussehen und Verhalten wie die original Seite. Ebenfalls wäre eine Nutzung der von Facebook bereitgestellten API für Anwendungen eine Überlegung wert. 2009 war diese API noch sehr eingeschränkt. Gerade der Zugriff auf Beiträge und private Nachrichten war nicht möglich (siehe [29, S. 253, Table 1]). Für die Implementierung eines Beispiel Programms wurde ein fünfter Ansatz gewählt. Über einen sogenannten Webcrawler oder eine Erweiterung für den Browser werden relevante Seiten automatisch heruntergeladen und in einen Archiv abgelegt. Dynamische Inhalte sollen kein Problem darstellen, da Seite erst heruntergeladen wird, wenn alle Aufrufe dynamischer Funktionen abgeschlossen ist. Die archivierten Dateien können dann mittels Datamining Techniken verarbeitet und als Atom/RSS Feed²⁵ bereitgestellt werden.

2.5.2 Reclaim Social

Ein Problem bei der heutigen Landschaft an sozialen Online-Netzwerken ist die Tatsache, dass man nicht nur ein einziges für alles Benutzt, sondern mehrere Gleichzeitig. Einige davon fast täglich andere werden immer weniger genutzt und irgendwann vergessen. Zugleich ist alles was man dort geschrieben oder hochgeladen hat auf den Server der Betreiber „gefangen“. Genau aus diesem Grund haben Sascha Lobo und Felix Schwenzel auf der Netzkonferenz re:publica²⁶ 2013 ihr Projekt „Reclaim Social“ [34] vorgestellt.

Ziel dieses Projektes ist es von einen selber erzeugten sozialen Daten aus allen möglichen Quellen auf seinen eigenen Blog zu spiegeln und so einen zentrale Anlaufstelle für seine eigenen Inhalte schaffen. Reclaim Social baut dazu auf der weit verbreiteten Blogsoftware „WordPress“²⁷ und der dafür vorhandenen Erweiterung „FeedWordPress“²⁸ auf. Diese Kombination ermöglicht alle Internetseiten, welche einen RSS/Atom-Feed anbieten, in die Datenbank von WordPress zu überführen. Das Problem hierbei besteht darin, dass einige sehr beliebte Internetseiten solche RSS/Atom-Feeds nicht anbieten (Facebook, Google+) oder eingestellt haben (<https://twitter.com>). Für einige solcher Seiten wurden „proxy-scripte“ [34, „Tech Specs Details“] implementiert, welche für diese einen RSS Feed emulieren. Zugleich können in den Feeds enthaltende Medien, wie Bilder und Videos (bisher nur als Referenz), heruntergeladen und in WordPress gespeichert werden. So ist es möglich alle gespiegelten Daten einfach zu durchsuchen oder nach bestimmten Kriterien zu filtern. Zusätzlich können alle Freunde, welche auch Reclaim Social einsetzen, in einen Kontaktliste eingetragen und so auch deren Inhalte eingebunden werden.

²⁵ <http://www.rssboard.org/rss-specification>

²⁶ <http://re-publica.de/>

²⁷ <http://wordpress.org/>

²⁸ <http://feedwordpress.radgeek.com/>

Aktuell befindet sich dieses Projekt noch im Alphastadium und die Installation ist relativ kompliziert. Es ist aber geplant eine eigene Erweiterung für WordPress zu schreiben „The goal is to build just one Reclaim Social-plugin for any wordpress user“[34, „How Does It Work“]

3 Analyse

Bevor nun ein System entwickelt werden kann, das die Synchronisation von Diskussionen auf Lernplattformen und sozialen Online-Netzwerken ermöglicht, muss erst eine Analyse der nötigen Komponenten durchgeführt werden. Dabei soll überprüft werden welche davon noch zu entwickeln und welche schon vorhanden sind. Diese Analyse wird anhand eines Beispielszenarios geschehen, dass die Schritte zeigt wie ein Beitrag von einer Webseite A in eine zweite Webseite B synchronisiert werden kann. Der umgekehrte Weg erfolgt dazu analog und wird deshalb weggelassen.

3.1 Ein Beispielszenario für die Synchronisation von Beiträgen

Der Anfang wird damit gemacht, dass ein Beitrag zum Synchronisieren vorhanden sein muss (Diese Schritte sind nicht Teil des fertigen Systems). Zum Beispiel geht ein Student in das Forum seiner Veranstaltung auf der Webseite A und in die Diskussion zur aktuellen Übung, weil er dazu eine Frage hat. Er schreibt also einen neuen Beitrag in das Eingabefeld und schickt es ab. Dieser wird dann an die Webseite A geschickt, im dortigen Format (Format A) in eine Datenbank oder Ähnliches geschrieben und als neuer Beitrag in der Diskussion angezeigt (siehe Abbildung 3.1). Da eine andere Webseite B das Format von Webseite A in der Regel nicht versteht, muss in einen nächsten Schritt der Beitrag aus der Datenbank gelesen und auf irgendeine Art in das Format von Webseite B (Format B) konvertiert werden.

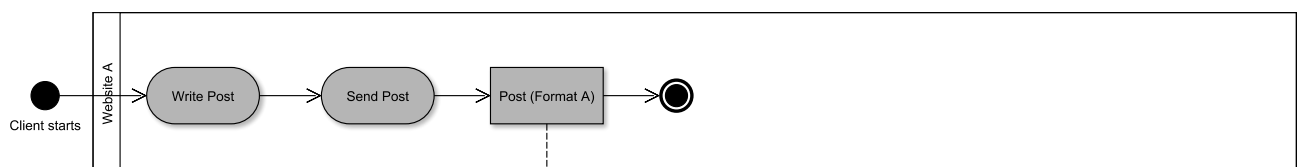


Abbildung 3.1.: Benutzer erstellt einen Beitrag auf der Webseite A.

3.1.1 Datenkonvertierung

Abbildung 3.3 zeigt den Ablauf, wie Beiträge von der Webseite A gelesen und konvertiert werden. Dazu müssen zuerst die Daten über eine öffentliche API vom Server der Webseite A heruntergeladen werden. Da im Allgemeinen nicht automatisch bekannt ist, wann ein neuer Beitrag vorhanden ist, muss der Server in zeitlichen Abständen abgefragt (*Polling* genannt) und die zurückgelieferten Daten nach neuen Beiträgen durchsucht werden. Sind neue Beiträge gefunden worden, können diese nicht direkt an die Webseite B geschickt werden, da sich diese in der Regel im verwendeten Datenformat unterscheiden. Diese müssen zuvor untereinander konvertiert werden.

Die einfachste Möglichkeit wäre die Daten von Webseite A, die in Format A vorliegen, in das Format von Webseite B zu konvertieren. Bei zwei Formaten ist dies noch sehr einfach. Es müsste lediglich ein Konverter von Format A nach Format B und einer in die umgekehrte Richtung implementiert werden. Für den Fall, dass eine weitere Webseite C unterstützt werden soll, würde sich die Anzahl an notwendigen Konvertern , wie Tabelle 3.1 zeigt, auf Sechs erhöhen.

Tabelle 3.1.: Anzahl Konverter bei drei Webseiten

		Nach		
		Webseite A	Webseite B	Webseite C
Von	Webseite A	-	×	×
	Webseite B	×	-	×
	Webseite C	×	×	-

Nimmt man an n_{ws} sei eine beliebige Anzahl Webseiten. Dann entspricht die Anzahl der notwendiger Konverter der Formel $n_{k1} = n_{ws} * (n_{ws} - 1)$, da für jede Webseite ein Konverter in alle anderen erzeugt werden muss. Sollen nur zwei oder drei Webseiten unterstützt werden ist der Aufwand noch sehr überschaubar. Bei mehr Webseiten kann dies aber sehr Aufwendig werden.

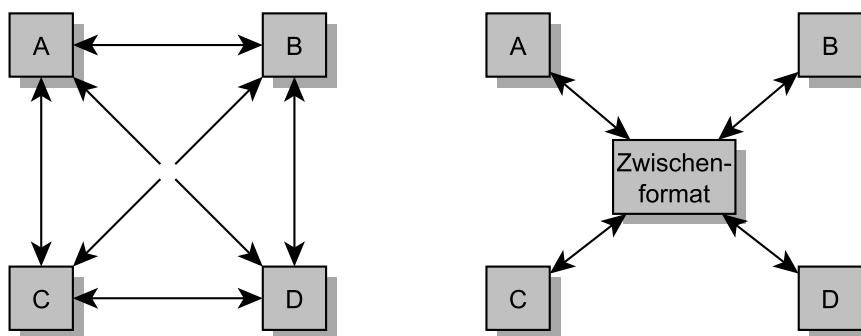


Abbildung 3.2.: Komplexität ohne und mit dem Einsatz eines Zwischenformats - Originalbild: [37]

Eine elegantere Methode für die Lösung dieses Problems, welche die Anzahl zu implementierender Konverter in Grenzen halten kann, wäre die Einführung eines Zwischenformates (auch Inter-Lingua, in Abbildung 3.3 als „IM Format“ bezeichnet) (siehe [37, S. 9]). Geht man davon aus, dass die Daten aller Webseiten nur in dieses Zwischenformat geschrieben und aus diesem gelesen werden müssen, würde sich der Aufwand auf maximal zwei Konverter je Webseite reduzieren. Für eine beliebige Anzahl Webseiten wären also $n_{k2} = n_{ws} * 2$ Konverter nötig. Nachteile hätte dieser Ansatz nur für $n_{ws} = 2$ und $n_{ws} = 3$, da in diesen Fällen mehr beziehungsweise gleich viele Konverter gegenüber der ersten Methode erforderlich wären. Erhöht man die Anzahl Webseiten jedoch nur geringfügig, sinkt die Menge an Konvertern sichtbar. Für $n_{ws} = 4$ wären es $n_{k2} = 8$ statt $n_{k1} = 12$ (siehe Abbildung 3.2) und für $n_{ws} = 5$ ergibt sich $n_{k2} = 10$ statt $n_{k1} = 20$ Konvertern. Gleichzeitig können so syntaktische Unterschiede in den einzelnen Formaten angeglichen werden, was sie leichter handhab- und wiederverwendbar macht.

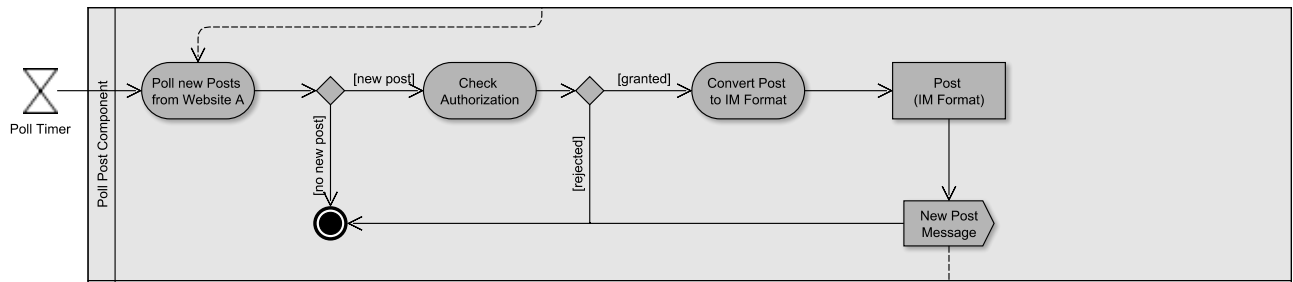


Abbildung 3.3.: Daten lesen und in das Zwischenformat konvertieren

Wahl des Zwischenformats für die Datenkonvertierung

Ein passendes Zwischenformat wurde im Kapitel „2“ mit SIOC beziehungsweise die Kombination aus SIOC und FOAF bereits vorgestellt. Damit ist es nicht nur möglich einzelne Beiträge plattformunabhängig zu speichern, sondern es kann auch die Struktur der Plattform als auch die Daten über die Benutzer mit eingebunden werden. Auch können Maschinen aus den RDF-Daten einfach Wissen für neue, hilfreiche Informationen ableiten. Doch warum sollte man auf RDF und Ontologien setzen anstatt etwas eigenes zu Entwickeln, wie zum Beispiel auf Basis des weitverbreiteten und vielseitigen XML Formats?

Hier gibt es mehrere Vorteile, warum man RDF einer mit XML-Schema definierten XML-Datei vorziehen sollte. XML hat das Problem, dass die selbe Aussage auf verschiedene Art repräsentiert werden kann. Listing 3.1 zeigt ein Beispiel für die Repräsentation der Aussage „Max Hiwi ist der Autor des Beitrags mit der ID 42“ in XML.

Listing 3.1: XML: Unterschiedliche Syntax, gleiche Semantik

```

1  <post>
2    <id>42</id>
3    <author>Max Hiwi</author>
4  </post>
5
6  <post id="42">
7    <author>Max Hiwi</author>
8  </post>
9
10 <post id="42" author="Max Hiwi" />
  
```

Alle drei XML-Elemente beschreiben die selbe Aussage, aber auf unterschiedliche Art und Weise. Für einen Menschen haben diese drei Schreibweisen die selbe Bedeutung, für eine Maschine ist dies aber nicht mehr ganz so offensichtlich. Auch kann nicht die Bedeutung der einzelnen Element und Tags von Maschinen erfasst werden. Dies muss der Programmierer erledigen. Zusätzlich erschweren diese Unterschiedlichen Strukturen das ausführen von Abfragen auf diesen Daten, da ein Mapping zwischen einer logischen Abfrage und den Daten nicht eindeutig erfolgen kann (vgl. [33, s. 41]).

Außerdem ist nie verkehrt auf bereits bestehende Formate zurückzugreifen, das sich diese oftmals schon bewährt und mit der Zeit gewachsen sind. Eine Neuentwicklung ist immer mit hohen Zeitaufwand und Fehlerrisiko verbunden.

3.1.2 Datenaustausch

Da das Eintreffen neuer Beiträge nicht vorhersagbar ist, ist es angebracht beim Synchronisieren das Lesen und Schreiben zeitlich zu entkoppeln. Eine der weit verbreitetsten Techniken dazu ist das Versenden von Nachrichten über einen Nachrichtenkanal den die schreibende Komponente nach neuen Beiträgen abhört. Diesen Kanal können mehrere Gleichzeitig abhören und stellen so eine gute Flexibilität und Erweiterbarkeit sicher.

Wahl der Technik für den Datenaustausch

Dieses Verhalten lässt sich mit JMS und Camel gut implementieren. Doch welche von den zwei Technologien sollte man nutzen? JMS hat den Vorteil, dass das versenden von Nachrichten ohne großen Aufwand möglich ist, da die meiste Arbeit von der MOM übernommen wird und man selber nur die Nachrichten zusammensetzen und verarbeiten muss. Mit JMS können auch so hilfreiche Funktionen wie „Durable Subscriber“ eingesetzt werden. Falls diese Funktion aktiviert ist und ein Client kurzzeitig keine Nachrichten empfangen kann (Er ist zu ausgelastet oder die Verbindung ist abgebrochen), so speichert die MOM alle Nachrichten für diesen Client und schickt sie ihn erst, wenn er wieder verbunden ist. Da sich JMS nur um das Versenden und Empfangen kümmert, muss alles was darüber hinaus geht selbst implementiert oder nach zusätzlichen Bibliotheken gesucht werden.

Mit Camel ist es genauso wie mit JMS mögliche Nachrichten von einem System in ein anderes zu schicken. Mit Camel kann dafür im Gegensatz zu JMS die Route der Nachricht frei definiert und diverse Komponenten erweitert werden. So wäre es ohne Probleme möglich einen Filter für unerwünschte Worte in die Route einzubauen ohne etwas an den anderen Komponenten zu ändern. Camel hat aber das Problem, dass es ohne die richtigen Zusatzkomponenten Nachrichten nur innerhalb des selben Programms verschicken kann. Für das Verschicken über die Grenzen des Programms und des Systems hinaus müssen Komponenten in die Route eingebaut werden, die eine Netzwerkkommunikation anbieten.

Die beste Wahl wäre eine Mischung aus JMS und Camel. So kann das entstehende System den Nachrichtenstrom durch unterschiedliche Routen erweitern und umbiegen und durch JMS ist die Kommunikation über ein Netzwerk möglich und es stehen Funktionen wie der „Durable Subscriber“ zur Verfügung.

3.1.3 Abschluss von Datenkonvertierung und -austausch

Empfängt nun eine Komponente, die für das Schreiben zuständig ist, eine Nachricht von einem neuen Beitrag der Webseite A, ist der Ablauf ähnlich wie beim Lesen nur in umgekehrter Reihenfolge (Siehe Abbildung 3.4 links, oberer Ablauf). Der neue Beitrag wird vor dem Schreiben aus dem Zwischenformat erst in das Format B konvertiert. Da bei einer Synchronisation von Vorteil wäre, wenn der synchronisierte Beitrag so aussehen würde, als hätte ihn der original Autor geschrieben, muss das System Zugriff auf die Benutzerkonten haben und diese in einer Datenbank verwalten. Dort kann dann nach dem passenden Benutzerkonto des Autors für die Webseite gesucht und dieses dann zum Schreiben verwendet werden. Steht ein solches Benutzerkonto

nicht zur Verfügung, ist das Ausweichen auf ein vorgegebenes Benutzerkonto hilfreich. Sind alle Voraussetzungen gegeben, kann der Beitrag mit dem ausgewählten Benutzerkonto auf die Webseite B geschrieben werden.

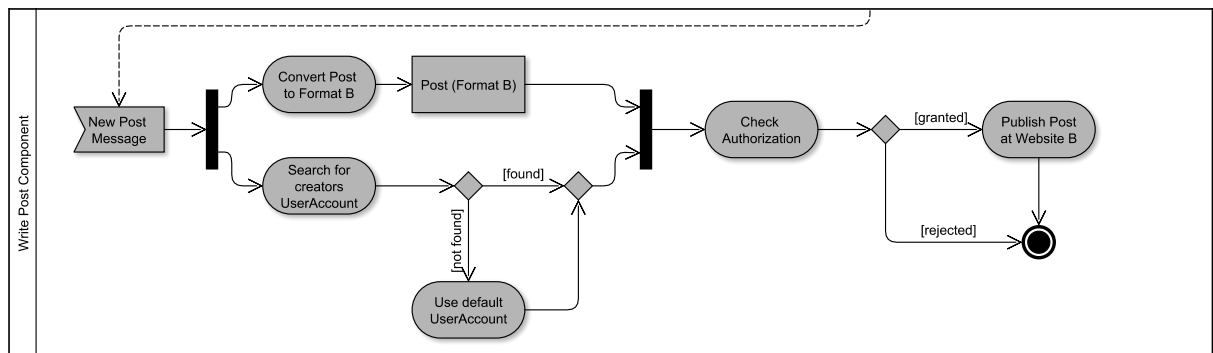


Abbildung 3.4.: Konvertierten des Beitrags in das Format B und schreiben n das soziale Netzwerk B

3.1.4 Privatsphäre

Wie in Abbildung 3.3 und 3.4 des Ablaufdiagramms zu sehen ist, wird jeweils beim Lesen und Schreiben nach einer Autorisation gefragt. Bei der automatischen Sammlung von nutzergenerierte Inhalten stellt sich immer die Frage wie es mit der Wahrung der Privatsphäre aussieht. Nicht jeder möchte, dass vielleicht sensible Informationen von ihnen weitergegeben werden oder ein Programm in seinen Namen Beiträge schreibt. Aus diesem Grund ist die Einführung eines Mechanismus sinnvoll mit dem ein Benutzer das Lesen seiner Beiträge und Schreiben in seinen Namen erlaubt, auf bestimmte Orte beschränkt oder verbietet (vgl. [6, S. 7]).

3.2 Identifizierung der Komponenten

Anhand dieser Schritte im Beispielszenario können sich eine Komponenten ablesen lassen, die das zu entwickelnde System enthalten muss. Gleichzeitig wurden einige Techniken ausgewählt auf die das System aufbauen kann, um nicht das Rad komplett neu zu erfinden:

- Eine Komponente muss eine einheitliche Schnittstelle bereitstellen mit der die Daten eine beliebigen Webseite durch eine Öffentliche API gelesen, in das SIOC-Format konvertiert und von diesem wieder in eine andere Webseite geschrieben werden können.
- Die Beiträge im SIOC-Format sollen über Camel und JMS zwischen den Schnittstellen ausgetauscht werden.
- Um stellvertretend für einen Benutzer schreiben zu können, muss es möglich sein nach dem Konto eines Benutzer, das zu einer bestimmten Webseite 'gehört, zu suchen und für den Zugriff über die API zu nutzen.
- Um die Privatsphäre der Benutzer zu gewährleisten, ist ein Mechanismus zum Festlegen von Zugriffsrechten sinnvoll.



4 Eigener Ansatz: Social Online Community Connectors (SOCC)

Aufbauend auf den in Kapitel 3 identifizierten Komponenten und Wahl der für ein System zur Synchronisation von Beiträgen passenden Techniken, soll nun der als *Social Online Community Connectors* (SOCC) benannter Ansatz vorgestellt werden. Den Aufbau von SOCC ist in Abbildung 4.1 zu sehen. Ein *Connector* von SOCC dient dabei als Schnittstelle zwischen den Datenformat der Plattform für die er implementiert wurde und dem SIOC-Format. SOCC stützt sich dabei auf die Prinzipien für die Verbindung von Daten im Web von Tim Berners-Lee (siehe [3]):

- „Use URIs as names for things“: Ressourcen wie Seiten, Foren, Threads und Beiträge sollten immer durch eine URI benannt werden.
- „Use HTTP URIs so that people can look up those names“: Die verwendeten URIs sollten einer Adresse im Web entsprechen, um auf die dahinter liegenden Daten zugreifen zu können.
- „When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)“: Die Daten sollten in einen Standard für das Semantic Web formatiert werden. Wie schon erwähnt setzt SOCC dazu auf RDF und darauf aufbauenden Ontologien wie FOAF und SIOC. Dadurch können zum Beispiel andere Anwendungen mit Anfragen in SPARQL¹ nach Beiträgen suchen.
- „Include links to other URIs. so that they can discover more things.“: Nicht nur die Struktur eine Diskussion kann über Links rekonstruiert werden, auch der Gewinn zusätzlicher Informationen ist so möglich. Beiträge können auf Lernmaterialien wie Folien oder Videos verweisen oder über das FOAF-Profil eines Autor können weitere Beiträge von ihm gefunden werden.

URIs sind also das wichtigste Element mit denen SOCC arbeitet. Soll zum Beispiel eine Diskussion synchronisiert werden, muss einem Connector die URI übergeben werden, hinter der sich die Daten befinden. Ein Connector ist nur für eine einzige Plattform zuständig. Aber eine Plattform kann über mehrere Connectoren angesprochen werden.

Intern besteht ein Connector aus drei Komponenten die zum Einen für das Lesen (*PostReader*) und Schreiben (*PostWriter*) von Beiträgen verwendet werden, zum Anderen aus einen *StructureReader* der für das Auslesen der Struktur der Plattform verantwortlich ist. Eine genaue Beschreibung dieser Komponenten folgt in Abschnitt 4.2.

Jeder Connector hat Zugriff auf eine als *Triplestore* bezeichnete Datenbank in der RDF-Triples gespeichert und mit SPARQL abgefragt werden können. Der Connector benutzt diesen Triplesore als Speicher in dem seine Konfigurationsdaten lagern, aber auch für zusätzliche Daten die er von Außerhalb benötigt. Er wird aber auch benutzt um Daten zu speichern, die während des Betriebes anfallen, da sie so irgendwann wieder verwendet werden können ohne sie erneut aus der von der

¹ <http://www.w3.org/TR/sparql11-overview>

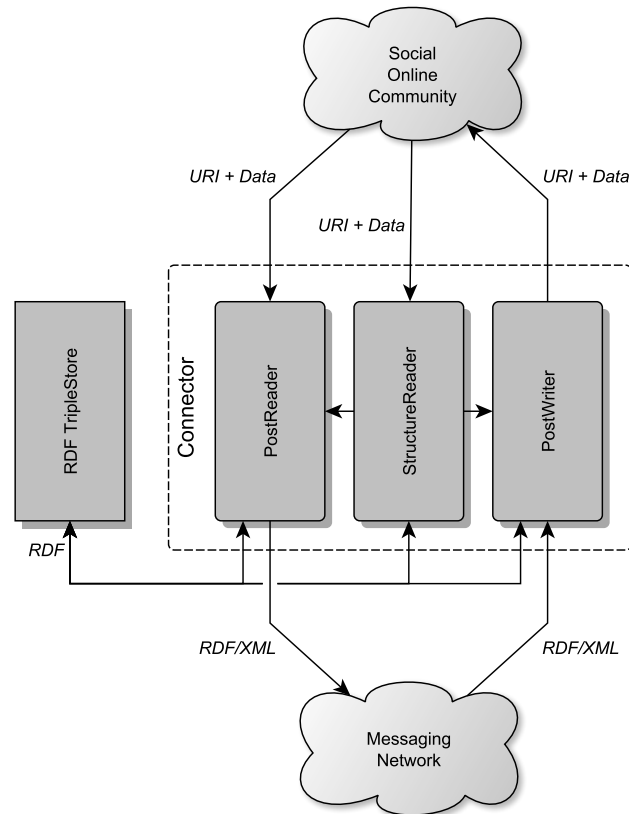


Abbildung 4.1.: Übersicht der Komponenten der SOCC

Plattform laden zu müssen. Zum Beispiel die Daten über die Struktur der verwendeten Plattform, die sich selten ändert.

Die Beiträge werden dann zwischen den einzelnen Connectoren über ein Nachrichtennetzwerk auf der Basis von Apache Camel ausgetauscht. Die Beschreibung dieser als *SOCC-Camel* bezeichneten Komponente erfolgt in Abschnitt 4.3.

4.1 Konfiguration

Damit ein Connector funktionieren kann, muss er von außen Informationen bekommen, welche er zum Betrieb braucht. Die sind zum Beispiel Informationen zu Benutzerkonten oder Parameter für die verwendete API. Da einige dieser Informationen nicht nur von einem Connector benutzt werden, ist es sinnvoll diese zusammen an einen Ort zu speichern und wiederverwenden zu können. Wofür der oben angesprochene Triplestore verwendet wird. Die wichtigsten Informationen für die Konfiguration der Connectoren stellen die Benutzerkonten dar. Sie enthalten unter anderem die Informationen, um Zugriff auf die Operationen der einzelnen APIs zu erhalten. Da die Benutzerkonten, wie später im Abschnitt 4.1.3 beschrieben, im FOAF-Format gespeichert werden, stellt es sich als Vorteil heraus die übrigen Informationen ebenfalls dort zu speichern und mit den schon vorhandenen zu verbinden.

Aus diesem Grund wurde für Konfiguration eines Connectors die *SOCC Connector Config Ontology* entwickelt. Diese Ontologie ist sehr einfach gehalten und baut auf schon vorhandenen

Ontologien auf. Zusätzlich musste die SIOC-Ontologie so erweitert werden, dass die Integration von Authentifizierungs- und Autorisierungsinformationen möglich war.

4.1.1 SOCC Connector Config Ontologie

Der Aufbau der SOCC Connector Config Ontologie (RDF-Präfix ccfg:) ist in der Abbildung 4.2 zu sehen. Die Konfigurationsdaten für einen Connector werden dabei durch die Klasse `ccfg:ConnectorConfig` modelliert. Diese Klasse enthält dann die folgenden Eigenschaften für den Connector.

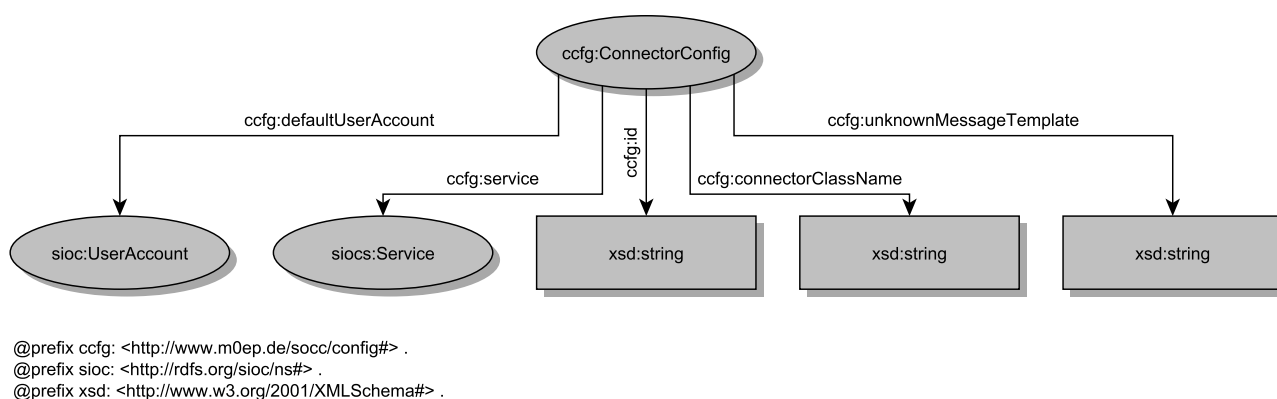


Abbildung 4.2.: Schema der SOCC Connector Config Ontology

Jeder Connector erhält einen eindeutigen `ccfg:id` zugewiesen, um jeden Connector später eindeutig identifizieren zu können. Die Eigenschaft `ccfg:connectorClassName` beschreibt den vollständigen Java-Klassennamen des beschriebenen Connectors. Diese wird für das Laden der richtigen Implementierung benötigt. Manchmal kann es passieren, dass keine passendes Benutzerkonto zum Schreiben eines Beitrags gefunden werden kann. Dadurch ist es wünschenswert, solche Beiträge dahingegen zu verändern, dass eine Verweis auf den original Autor und vielleicht wo der Beitrag gemacht wurde vorhanden ist. Durch die Eigenschaft `ccfg:unknownMessageTemplate` kann eine Vorlage für das Aussehen des Verweises definiert werden. Innerhalb dieser Vorlage stehen einige Variablen in der Form „{varName}“ zur Verfügung, wobei „varName“ durch den Namen der entsprechenden Variable zu ersetzen ist. Welche Variablen alle vorhanden sind und mit welchen Werten diese ersetzt werden, ist in Tabelle 4.1 zu sehen.

Für die Nutzung einiger APIs müssen zusätzlich bestimmte Parameter angegeben werden. Dies könnte zum Beispiel die genaue Adresse des Dienstes sein. Hierzu wird auf das *SIOC Services Modul* zurückgegriffen. Dieses stellt eine Klasse `sioc:Service` zur Verfügung und über die Eigenschaft `ccfg:service` kann eine solche Servicebeschreibung einem Connector zugewiesen werden. Der genaue Aufbau eines solchen Services wird im Abschnitt 4.1.2 dargestellt. Die letzte Information für die Konfiguration eines Connectors ist ein Standardbenutzer (Im Folgenden Defaultuser genannt) und wird mit der Eigenschaft `ccfg:defaultUserAccount` festgelegt. Dieser Defaultuser erfüllt im Großen und Ganzen zwei Aufgaben. Als Erstes wird er für lesende Zugriffe der API auf die verwendete Plattform genutzt. Hierzu ist ein einzelnes Benutzerkonto vollkommen ausreichend, da nur die gelesenen Daten wichtig sind und nicht von welchem Konto sie kommen. Es sollte aber ein Konto mit weitreichenden Befugnissen sein, um möglichst viel Daten lesen zu

Tabelle 4.1.: Variablennamen und Ersetzung innerhalb von `ccfg:unknownMessageTemplate`

varName	Ersetzt durch
message	Original Beitrag
sourceUri	URI des original Beitrags
connectorId	ID des aktuellen Connectors
serviceName	Name des vom Connector verwendeten Service
creationDate	Erstelldatum des Beitrags (falls bekannt)
authorName	Name des Autors (falls bekannt)

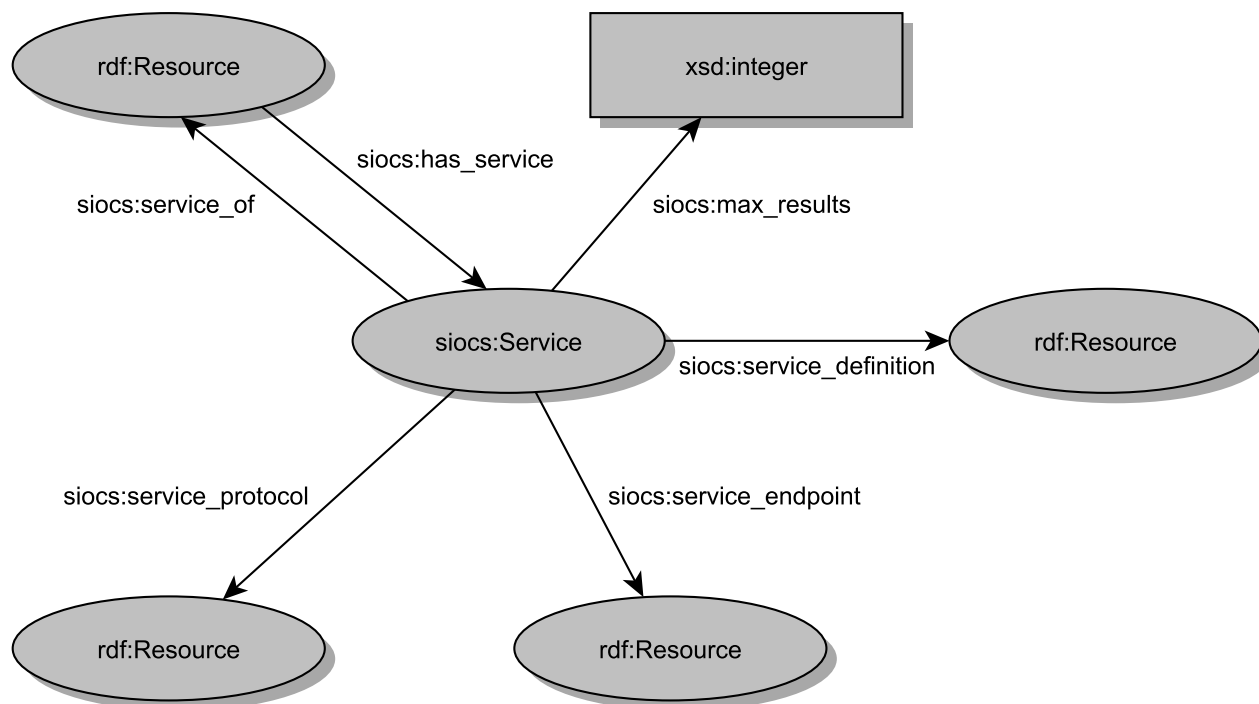
können. Die zweite Aufgabe bezieht sich auf das stellvertretende Schreiben einzelner Benutzer. Nicht immer werden die dazu notwendigen Daten von den Benutzer zur Verfügung gestellt oder sind unbekannt. In diesem Fall wird der Defaultuser genutzt und der Inhalt des Beitrags in die Form der mit der Eigenschaft `ccfg:unknownMessageTemplate` festgelegten Vorlage konvertiert.

4.1.2 Services

Wie eben schon beschrieben, existiert für SIOC ein Modul zur einfachen Modellierung von Diensten auf semantischer Ebene: Das SIOC Services Module (RDF-Präfix `siocs:`). Kernstück dieses Moduls ist die Klasse `siocs:Service`, wie auf Abbildung 4.3 zu sehen ist. Mit dieser Klasse kann durch eine Hand voll Eigenschaften ein Service beschrieben werden. Für diese Arbeit ist davon die wichtigste Eigenschaft `siocs:service_endpoint`. Durch diese kann die Adresse festgelegt werden, unter dem ein Service erreichbar ist. Gerade bei Plattformen die nicht an eine feste Adresse (Foren, Blogs, ...) gebunden sind, ist diese Angabe unerlässlich. Die Eigenschaften `siocs:has_service` und `siocs:service_of` sind ideal zur Verbindung von einzelnen `sioc:UserAccounts` mit einem Service. Diese Verbindung hilft dabei für das stellvertretende Schreiben von Beiträgen schnell die passenden Benutzerdaten zu finden. Ebenfalls nützlich ist `siocs:max_results`. Manche Dienste erlauben es nur eine maximale Anzahl an Ergebnissen pro Aufruf zurückgeben zu lassen. Da sich diese Anzahl über die Zeit ändern kann ist es nicht sinnvoll diese fest im Programm festzulegen. Für SOCC weniger interessant aber der Vollständigkeit halber seien noch `siocs:service_protocol` zum Angeben des verwendeten Übertragungsprotokolls (REST, SOAP, ...) und `siocs:service_definition` mit dem auf eine weiterführende Definition verwiesen werden kann erwähnt.

4.1.3 Benutzerdaten

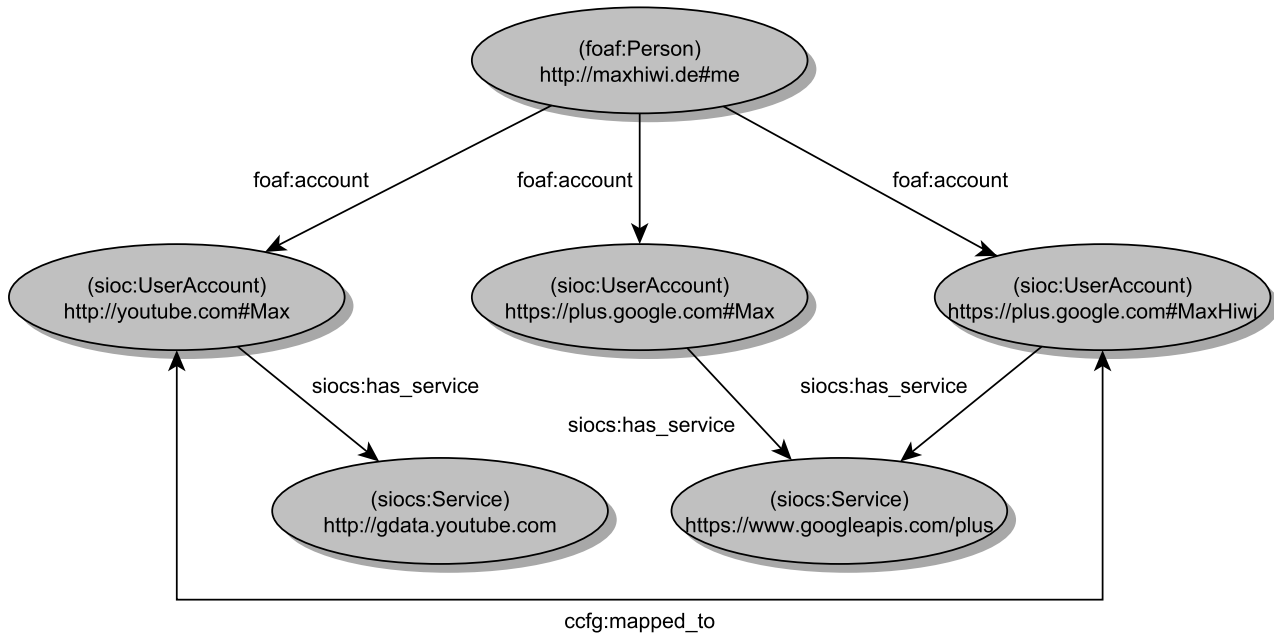
Soll ein Beitrag eines Benutzers von Google+ nach Facebook synchronisiert werden und es so aussehen, als hat er diesen Beitrag selbst auf Facebook geschrieben, sind gute Kenntnisse über alle Benutzerkonten dieser einen Person notwendig. Als erstes muss die Existenz dieser Person dem System bekannt sein. Hierzu kann diese durch die Klasse `foaf:Person` aus der FOAF Ontologie dargestellt werden. Für ein einzelnes Benutzerkonto wurde in SIOC die Klasse `sioc:UserAccount`



@prefix siocs: <http://rdfs.org/sioc/services#> .
 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

Abbildung 4.3.: SIOC Services Module

definiert. Da `sioc:UserAccount` eine Unterklasse von `OnlineAccount` aus FOAF ist, kann diese über die Eigenschaft `foaf:account` beziehungsweise `sioc:account_of` mit einer Person verbunden werden. Ebenso ist es wichtig zu wissen zu welchen Plattform ein Benutzerkonto gehört. Deshalb wird der `sioc:UserAccount` mit einem Objekt der Klasse `siocs:Service` über die Eigenschaft `siocs:has_service/siocs:service_of` zusammengebracht. Diese Verbindung ist für manche APIs besonders bedeutend, da in dem Serviceobjekt relevante Daten für den Zugriff darauf enthalten sind. Nun kann es vorkommen, dass eine Person mehrere Benutzerkonten für private und geschäftliche Dinge besitzt. Um nicht private Beiträge auf Webseite A mit dem geschäftlichen Benutzerkonto auf Webseite B zu schreiben, muss ein Mapping zwischen den verschiedenen Benutzerkonten festgelegt werden. Dieses Mapping kann über ein in der „SOCC Connector Config Ontologie“ definierte Eigenschaft `ccfg:mapped_to` realisiert werden. Diese Eigenschaft ist symmetrisch, also falls Benutzerkonto A mit Benutzerkonto B über `mapped_to` verbunden ist, dann gilt dies ebenfalls für B mit A. Abbildung 4.4 zeigt den Zusammenhang zwischen der Klasse `foaf:Person`, `sioc:UserAccount` und `siocs:Service` sowie der Eigenschaft `ccfg:mapped_to` noch einmal graphisch an einem Beispiel.



```

@prefix ccfg: <http://www.m0ep.de/socc/config#> .
@prefix sioc: <http://rdfs.org/sioc/ns#> .
@prefix siocs: <http://rdfs.org/sioc/services#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

```

Abbildung 4.4.: Zusammenhang von Person, UserAccount und Service. Die inversen Eigenschaften `sioc:account_of` und `siocs:service_of` wurden zu einer besseren Übersicht weggelassen

4.1.4 Authentifizierung

Die Ontologien FOAF und SIOC sind hervorragend für die Abbildung von sozialen Netzwerken und Diskussionen geeignet, jedoch ist es mit ihnen nicht möglich Daten zur Authentifizierung (Feststellen ob jemand der ist, den er vorgibt zu sein) zu speichern. Wegen dem Schutz der Privatsphäre ist dies verständlich, jedoch um stellvertretend für einen Benutzer Beiträge zu schreiben, ist es wichtig Zugriff auf diese Daten zu haben und sie einem Benutzerkonto zuordnen zu können. Zuerst muss dazu aber festgestellt werden, welche verschiedenen Mechanismen es zum Anmelden an ein solches Konto existieren.

Benutzername/Passwort ist wohl eine der ersten und häufigsten Mechanismen, um den Zugriff sensibler Daten vor Dritten zu schützen. Moodle und Youtube setzen zum Beispiel den Benutzername und Passwort eines angemeldeten Benutzers zu Authentifizierung ein.

OAuth ist technisch gesehen eher ein Mechanismus zur Autorisierung als zu Authentifizierung. Da es aber auch Informationen enthält die ein Programm zur Authentifizierung von sich gegenüber einer API enthält, wird diese in diesen Abschnitt behandelt. OAuth² wird heutzutage hauptsächlich für den zugriff auf webbasierte APIs verwendet. Benutzer können so temporär Programmen den Zugriff auf ihre Daten erlauben und später wieder verbieten. Die aktuelle

² OAuth Webseite: <http://oauth.net/>

Version stellt OAuth 2.0 dar und wird in dieser Version von den größten Seitenbetreibern wie Google, Facebook oder Microsoft eingesetzt³. Insgesamt sind für die Nutzung von OAuth vier Parameter wichtig. Für das Programm, das Zugriff erhalten möchte, sind es die Parameter *client_id* und *client_secret* (Siehe [22][S. 8]). Sie weisen das Programm als autorisiert für die Benutzung der API aus. Will man nun eine von OAuth geschützte Funktion nutzen, ist ein sogenannter Accesstoken nötig (Siehe [22][S. 9]). Da dieser Accesstoken in der Regel nur eine bestimmte Zeit gültig ist, wird je nach Implementierung des Standards noch ein RefreshToken mitgeliefert. Mit diesem RefreshToken ist das Programm in der Lage ohne Zutun des Benutzers einen abgelaufen Accesstoken wieder zu aktivieren. Dies kann beliebig oft wiederholt werden, bis der Benutzer beide Token für ungültig erklärt.

API Schlüssel sind eine dritte Möglichkeit Programmen Zugriff auf eine API zu gewähren. Der API Schlüssel entspricht ungefähr einer Kombination von *client_id* und *client_secret* von OAuth. Dieser Schlüssel schaltet in der Regel nicht den Zugriff auf persönliche Daten von Benutzer frei. Dafür ist noch ein weiterer Mechanismus wie die Verwendung von einem Benutzernamen und Passwort nötig. Die in Abschnitt 5.2.7 beschriebene Google Youtube API hierzu ein gutes Beispiel.

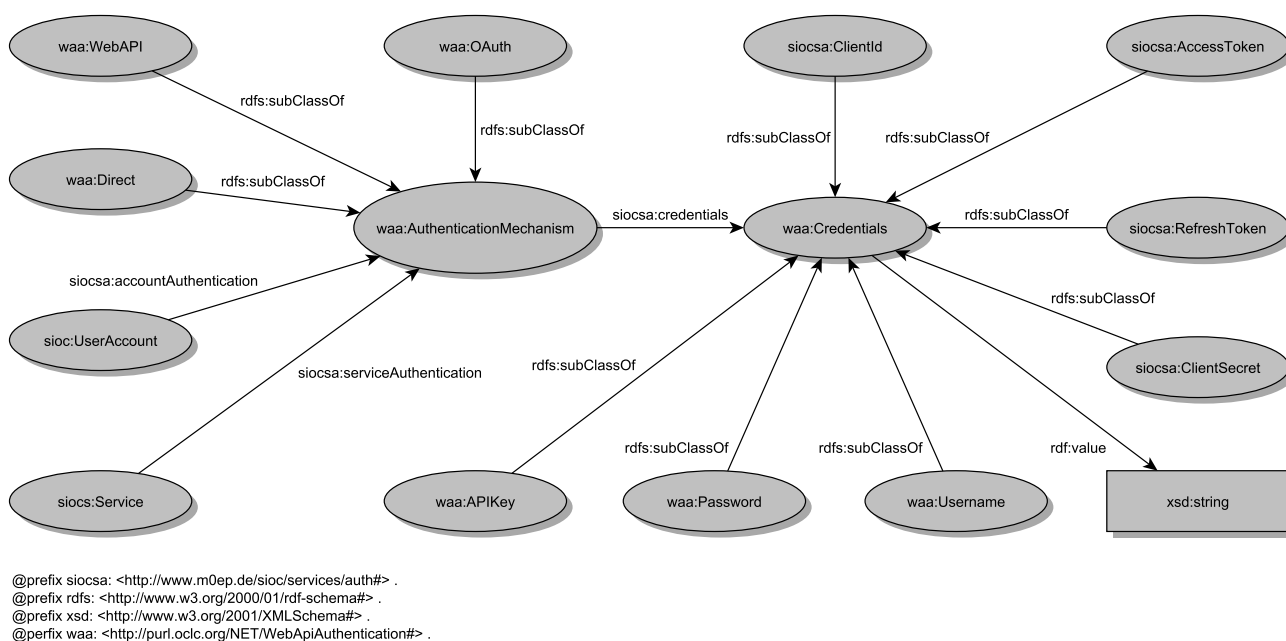


Abbildung 4.5.: SIOC Services Authentication Ontology

Neben diesen drei Mechanismen wäre noch der Vollständigkeit halber die HTTP-Authentifizierung zu nennen. Hierbei handelt es sich um eine Form des Benutzernamen-Passwort-Verfahrens, welches auf dem HTTP Protokoll aufsetzt. Für einfachen Webseiten ist dies eine unkomplizierte Art die Datei vor fremden Zugriffen zu schützen. Für aktuelle öffentliche APIs ist diese Form der Authentifizierung nicht mehr Stand der Technik.

Die Suche nach einer bestehenden Ontologie, welche zusammen mit SIOC verwendet werden könnte, gestaltete sich als sehr schwierig. Eine Ausnahme stellt die *Authentication Ontology*⁴

³ OAuth Versionen im Einsatz: http://en.wikipedia.org/wiki/OAuth#List_of_OAuth_service_providers

⁴ Authentication Ontology: <http://omnivoke.kmi.open.ac.uk/authentication/>

(RDF-Prefix `waa:`) des *OmniVoke*⁵ Frameworks dar. Die Art der Authentifizierung wird darin durch die Klasse `waa:AuthenticationMechanism` modelliert. Unterklassen davon für die wichtigsten Mechanismen wie `waa:OAuth`, `waa:WebAPI` und Benutzername/Passwort (dort `waa:Direct` genannt) sind vorhanden. Jedem `AuthenticationMechanism` Objekt können dann `waa:Credentials` (engl. für Anmeldedaten) angehängt werden.

Das einzige Manko an dieser Ontologie war das Fehlen von Credentials für OAuth in der Version 2.0. Im einzelnen waren dies Klassen für `client_id`, `client_secret` sowie für Access- und RefreshToken. Um auch diese OAuth Version unterstützen zu können, wurden hierfür die Klassen `siocsa:ClientId`, `siocsa:ClientSecret`, `siocsa:AccessToken` und `siocsa:RefreshToken` als Unterklassen von `waa:Credentials` abgeleitet. Als Letztes musste noch eine Verbindung zwischen Authentication Ontology und SIOC hergestellt werden. Zum Einen war eine Erweiterung der Klasse `sioc:UserAccount` notwendig, so dass die Anmeldedaten der Benutzer zur Verfügung standen. Zum Anderen werden Daten wie ein API-Schlüssel von einem Service benötigt, die von denen der Benutzer unabhängig sind. Für die Klasse `sioc:UserAccount` wurde die Eigenschaft `siocsa:accountAuthentication` geschaffen. Diese erwartet als Subjekt einen `sioc:UserAccount` und als Objekt ein `waa:AuthenticationMechanism`, welcher dann die Credentials enthält. Für die Klasse `sioc:Service` existiert das Äquivalent `siocsa:serviceAuthentication`.

Diese Erweiterungen und die übernommenen Teile der Authentication Ontology wurden danach im *SIOC Services Authentication Module* (RDF-Präfix `siocsa:`) zusammengefasst. Graphisch ist sie in Abbildung 4.5 und im Anhang A.2 als OWL Schema zu sehen.

4.1.5 Autorisierung

Da für viele Menschen im Internet ihre Privatsphäre wichtig ist, sollte von den Benutzern für das Lesen und Schreiben ihrer Beiträge erst ihre Erlaubnis dazu eingeholt werden. Ein verbreitetes Mittel für eine solche Zugriffssteuerung sind Access Control Lists (ACL) (engl. für Zugriffssteuerungsliste). Mit ihnen wird geregelt wer bestimmte Operationen auf eine Ressource durchführen darf. Für den Einsatz in dieser Arbeit wurde die *Basic Access Control Ontologie* (siehe [25, 1]) (RDF-Präfix `acl:`) ausgewählt (Im Folgenden nur als ACL bezeichnet). Da sie Teilweise auf FOAF aufbaut, ließ sie sich sehr einfach in das bestehende System integrieren.

Zugriffsrechte für Ressourcen werden in dieser ACL mit der Klasse `acl:Access` modelliert. Von dieser Klasse werden einzelne Rechte wie `acl:Read` für das Lesen und `acl:Write` für das Schreiben abgeleitet. Nebenbei existieren noch die Ableitungen `acl:Control` für die Kontrolle über die ACL und `acl:Append` zum Anfügen von Daten an die Ressource. Die letzten beiden Rechte sind aber für SOCC unbedeutend.

Die Verbindung von einem Zugriffsrecht mit einer Ressource wird über die Klasse `acl:Authorization` erreicht. Der Besitzer dieser Autorisierung wird durch die Eigenschaft `alc:owner` festgelegt und gehört zur Klasse `foaf:Agent` beziehungsweise der davon abgeleiteten Klasse `foaf:Person`. Mittels der Eigenschaft `acl:agent` wird festgelegt für welche Person/Agenten diese Autorisierung gilt. Das selbe wird auch über die Eigenschaft `acl:agentClass` bestimmt, wobei hier eine abstrakte Klasse statt einer Instanz davon gemeint ist. Soll zum Beispiel der

⁵ OmniVoke: <http://omnivoke.kmi.open.ac.uk/framework/>



@prefix acl: <http://www.w3.org/ns/auth/acl#> .
 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
 @prefix gen: <http://www.w3.org/2006/gen/ont#> .
 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

Abbildung 4.6.: Basic Access Control Ontologie

öffentlicher Zugriff für eine Ressource definiert werden, wird bei `acl:agentClass` die Klasse `foaf:Agent` eingesetzt (vgl. [1, „Public Access“]). Dies besagt, dass alle Agenten auf diese Ressource zugreifen können. Innerhalb von SOCC werden nur Ressourcen verarbeiten, die so öffentlich zugänglich gemacht wurden. Welches Recht für eine Ressource eingeräumt wird, kann über die Eigenschaft `acl:mode` festgelegt werden. SOCC testet aber nur ob das Recht `acl:Read` zum Lesen oder `acl:Write` zum Schreiben von Beiträgen eingeräumt wurde. Auf welche Ressource sich letztendlich eine Autorisierung bezieht, wird über die Eigenschaft `acl:accessTo` geregelt. Die Angabe von „`http://www.facebook.com`“ würde sich für SOCC zum Beispiel auf alle Beiträge des Besitzers auf Facebook beziehen, „`https://canvas.instructure.com/courses/798152`“ dahingegen nur auf alle Beiträge innerhalb eines Canvas Kurses mit der ID „798152“. Für einen Zugriff auf alle Beiträge prüft SOCC, ob die Eigenschaft `acl:accessToClass` auf die Klasse `sioc:Post` verweist. So müsste nicht jede einzelne Webseite angegeben werden, wenn man ein generelles Zugriffsrecht auf seine Beiträge einräumen will.

Das Listing 4.1 zeigt ein Beispiel, wie eine Autorisierung mit der ACL in Turtle aussehen könnte. Der Besitzer dieser Autorisierung mit der URI `http://example.org#john` wird in Zeile 2 festgelegt. Er erlaubt damit SOCC einen öffentlichen (Zeile 3), lesenden (Zeile 4) Zugriff auf all seine geschriebenen Beiträge (Zeile 5).

Listing 4.1: Basic Access Control Beispiel in Turtle

```

1  [] a acl:Authorization ;
2    acl:owner <http://example.org#john> ;
3    acl:agentClass foaf:Agent ;
4    acl:mode acl:Read ;
5    acl:accessToClass sioc:Post .

```

4.2 Design eines Connectors

Die rechte Abbildung 4.7 zeigt die Schnittstelle `IConnector` gegen die ein Connector implementiert werden muss. Sie bietet Zugriff auf die Eigenschaften die mit der Klasse `ccfg:ConnectorConfig` für die Konfiguration eines Connectors im Triplestore bereitgestellt werden. Diese Schnittstelle definiert Methodenüberwelche die Komponenten zum Lesen der Seitenstruktur mit dem `StructureReader` sowie zum Lesen und Schreiben von Beiträgen mit `PostReader` und `PostWriter` zugänglich sind. Zuzüglich werden noch Methoden für den Zugriff auf Hilfskomponenten wie den *SOCC Context*, *AccessControl* und *ClientManger* bereitgestellt. Das UML-Klassendiagramm in Abbildung 4.8 zeigt die Beziehung zwischen den Connector und den anderen Komponenten aus dem er besteht, welche in den folgenden Unterabschnitten noch weiter erklärt werden.

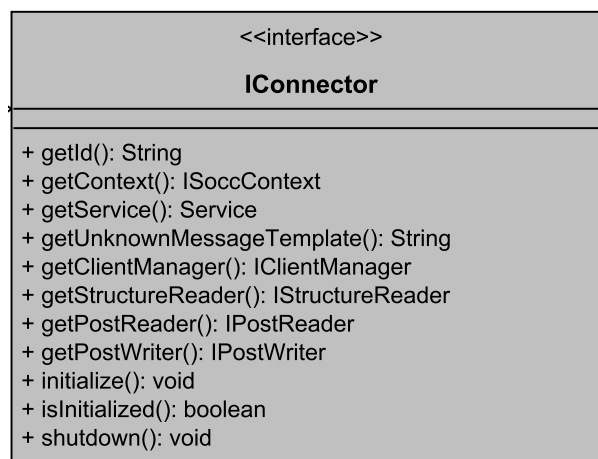


Abbildung 4.7.: PostReader

Für den Lebenszyklus eines Connectors sind noch die Methoden `initialize()` und `shutdown()` wichtig. Nach dem Erzeugen eines Connectors muss die Methode `initialize()` aufgerufen, um noch mögliche Vorarbeiten durchzuführen bevor der Connector genutzt werden kann. Ob ein Connector schon initialisiert wurde, kann mit der Methode `isInitialized()` überprüft werden. Bevor eine Connector gelöscht werden kann, sollte noch die Methode `shutdown()` aufgerufen werden. Sie dient dazu verwendete Ressourcen wieder freizugeben.

4.2.1 SOCC Context

Der `SOCCContext` eines Connectors, beschreibt die Umgebung die er zum Arbeiten braucht. Über ihn bekommt der Connector Zugriff auf den Triplestore, der durch die Klasse `Model` der *RDF2Go* Bibliothek abstrahiert wird (Siehe Abschnitt 5.1). In ihm befinden sich alle Daten die der Connector für seinen Betrieb benötigt und benutzt ihn gleichzeitig als Lagerplatz für Daten die während des Betriebs gespeichert werden müssen. Eine Referenz auf dieses Triplestore erhält der Connector über den Aufruf der Funktion `getModel()`. Durch die Methode `getAccessControl()` kann der Connector über die im folgenden Abschnitt beschriebene *AccessControl*-Schnittstelle auf die Information für die Autorisierung zugreifen.

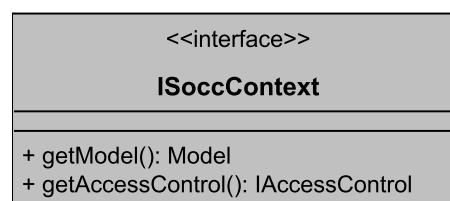


Abbildung 4.9.: SOCC Context

4.2.2 AccessControl

Die *AccessControl*-Schnittstelle ist sehr einfach gehalten und dient für den Zugriff auf die in Abschnitt 4.1.5 beschriebenen *ACL*-Information. Die Methode `checkAccessTo(...)` prüft, ob der Zugriff auf eine Ressource mit allen übergebenen Rechten erlaubt ist. Die andere Methode

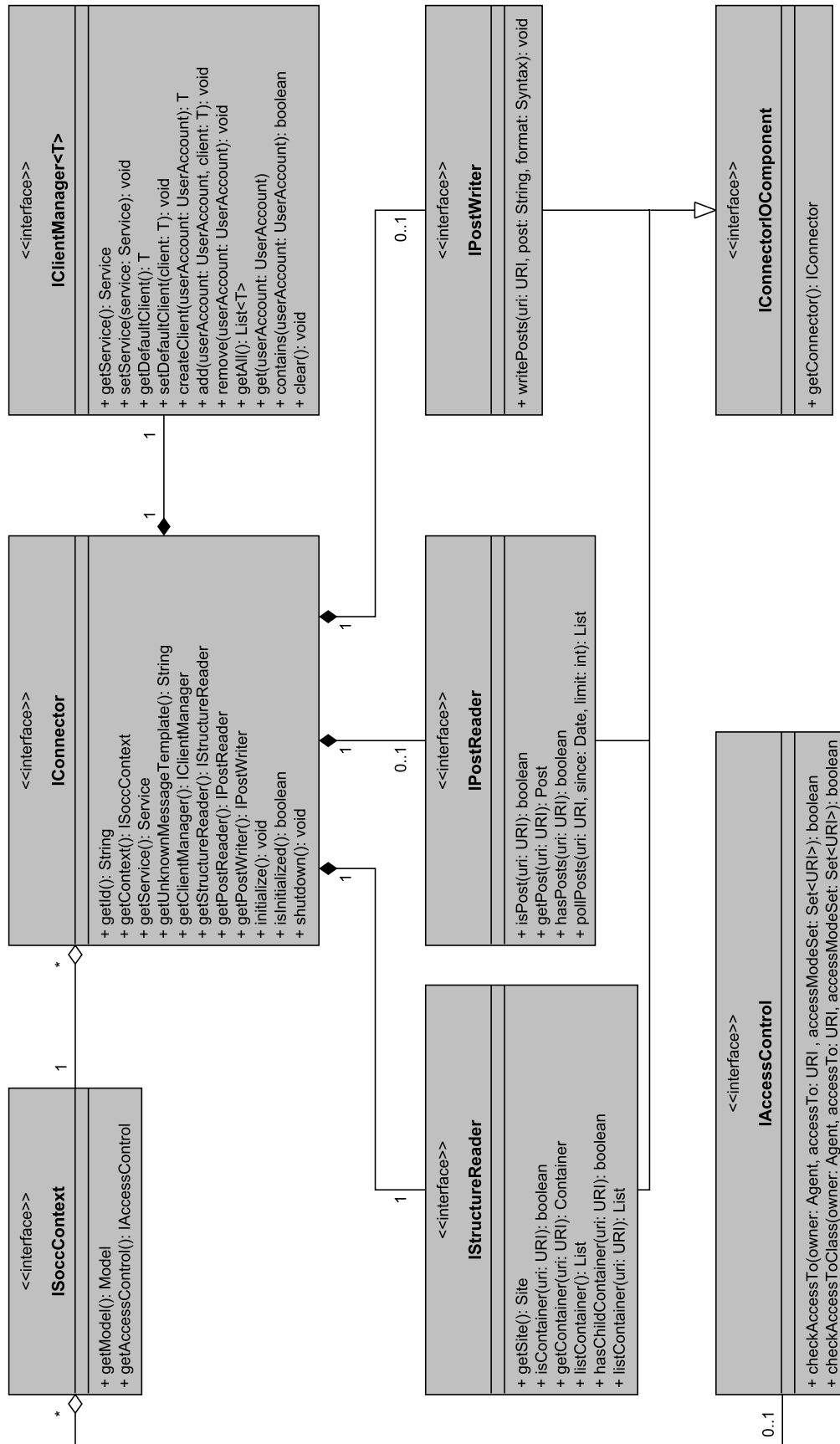


Abbildung 4.8.: UML-Klassendiagramm der Connectoren

`checkAccessToClass` ist zur Überprüfung, ob die Rechte für den Zugriff auf eine komplette Klasse von Ressourcen vorhanden sind.



Abbildung 4.10.: AccessControl

4.2.3 ClientManager

Der Zugriff innerhalb des Programms auf eine API erfolgt in der Regel über einen Client. Dieser Client erlaubt es mit den Anmeldedaten für ein Benutzerkonto auf die Funktionen der API über verschiedene Methoden zuzugreifen. Da ein Client immer nur mit einem Benutzerkonto verknüpft ist und von diesen eine große Anzahl verwaltet werden müssen, enthält jeder Connector einen eigenen ClientManager für diese Aufgaben. Für alle vom Benutzer unabhängigen Daten erhält der ClientManager ein wie Abschnitt 4.1.2 beschriebenes Objekt der Klasse `Service` das oftmals wichtige Daten wie `ClientID` und `ClientSecret` enthält. Das Erzeugen eines neuen Clients erfolgt dann durch den Aufruf der Methode `createClient(...)`. Als Parameter wird er ein Benutzerkonto (`sioc:UserAccount`) übergeben. Sind alle erforderlichen Authentifizierungsinformation aus Abschnitt 4.1.4 vorhanden, wird ein neuer Client erstellt und zurück an den Aufrufer gegeben. Dieser Client wird aber dadurch nicht automatisch vom ClientManager verwaltet. Hierzu muss der im vorherigen Schritt erzeugte Client durch die Übergabe an `add(userAccount: UserAccount, client: T)` dauerhaft mit den angegeben Benutzerkonto verknüpft und intern gespeichert werden. In der aktuellen Implementierung ist es wichtig, dass die Eigenschaften `foaf:accountName` und `foaf:accountServiceHomepage` des `sioc:UserAccount`-Objekts gesetzt sind. Aus diesen wird ein eindeutiger Schlüssel generiert, der zur Zuordnung von Benutzerkonto und Client innerhalb des ClientManagers dient. Des weiteren stehen noch Methoden `remove(userAccount: UserAccount)` zum Entfernen, `get(userAccount: UserAccount)` zum Suchen von Clients sowie `contains(userAccount: UserAccount)` zum Testen, ob ein Client zu einem Benutzerkonto existiert zur Verfügung. Sollen zum Beispiel am Ende der Laufzeit des Programms alle erzeugten Clients auf einmal abgemeldet und gelöscht werden, kann dies über die Methode `clear()` erfolgen. Der ClientManager verwaltet ebenfalls den Client für den in Abschnitt 4.1.1 angesprochenen Defaultuser. Dieser *Defaultclient* kann über die Methode `setDefaultClient(client: T)` gesetzt und durch `getDefaultClient()` jederzeit wieder abgerufen werden.

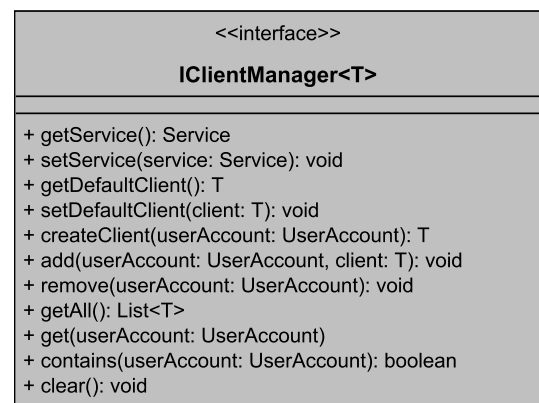


Abbildung 4.11.: ClientManager

4.2.4 StructureReader

Um auf Informationen über die Struktur von Foren, sozialen Online-Netzwerken und so weiter im SIOC-Format zugreifen zu können, implementiert jeder Connector dazu einen StructureReader. Die Struktur lässt sich, wie im Abschnitt 2.2.2 vorgestellt, durch die SIOC Klassen `sioc:Site` und `sioc:Container` (sowie Unterklassen davon) beschreiben. Für den Zugriff auf diese Struktur, enthält der StructureReader mehrere Methoden die in der UML-Klasse rechts in Abbildung 4.12) zu sehen sind.

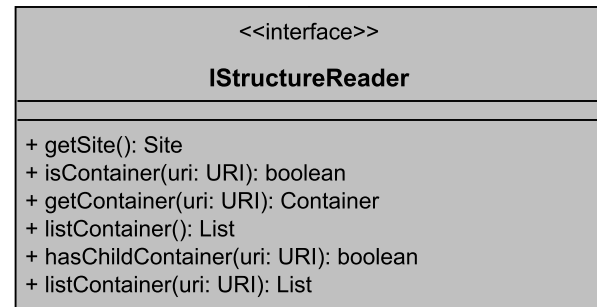


Abbildung 4.12.: StructureReader

getSite() ist eine Methode, welche die Beschreibung einer Seite (Forum, Blog, soziales Online-Netzwerk) als Objekt der SIOC-Klasse `sioc:Site` zurücklieft. Dieses wird relativ häufig gebraucht, um die Zugehörigkeit anderer Objekte durch einen Verweis auf diese Seite zu verdeutlichen. Dies kann bei einigen APIs nützlich sein, da dort manchmal keine Information zum Ursprungsort eines Beitrags mitgeliefert werden, über den man sonst eine Beziehung zwischen Seite und Beitrag herstellen könnte.

isContainer(uri: URI) wird zur Überprüfung verwendet, ob sich hinter einer URI ein potenzieller Container für Beiträge oder andere Container befindet.

getContainer(URI) liefert Informationen zu einem Container im SIOC-Format, der sich hinter der übergebenen URI befindet.

hasChildContainer(uri: URI) überprüft, ob der Container hinter der übergebenen URI weitere Container als Kinder besitzt. Diese Methode wird dazu eingesetzt, um vorab zu testen, ob der Aufruf von `listContainer(URI)` das gewünschte Ergebnis liefert oder ein Fehler auftreten würde.

listContainer(...) sind Methoden, welche für das Auflisten aller Container einer Seite (aus der Sicht des Defaultusers) zur Verfügung stehen. Die Methode ohne Parameter listet alle Container auf der ersten Ebene der Seitenstruktur auf. Dies könnten zum Beispiel alle Kurse innerhalb von Canvas oder alle beigetretenen Gruppen auf Facebook sein. Die zweite Methode mit einer URI als Parameter gibt eine Liste aller Container, welche den Container hinter der übergebenen URI als Elternteil haben zurück. Dies könnten im Falle von Canvas alle Diskussionsthemen innerhalb eines Kurses sein.

4.2.5 PostReader

Der `PostReader` dient als Schnittstelle für das Lesen geschriebener Beiträge innerhalb eines Containers oder der Kommentare auf einen anderen Beitrag. Er stellt nach außen hin Funktionen bereit mit denen entweder ein einzelner Beitrag oder alle Beiträge, die bestimmte Kriterien erfüllen, gelesen werden können. Bevor ein Beitrag zurück gegeben wird, müssen die Methoden prüfen, ob der Autor eines Beitrag das Lesen für diesen erlaubt hat. Falls nicht, wird der Beitrag aus der Ergebnisliste gelöscht. Die Funktionsweise der einzelnen Methoden ist wie folgt:

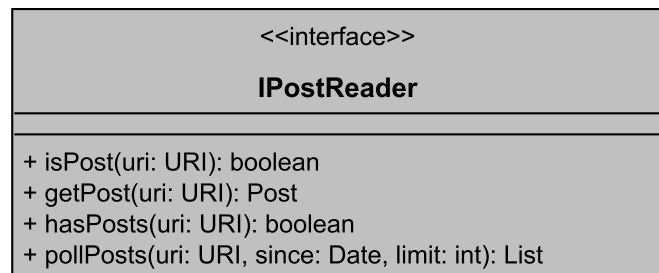


Abbildung 4.13.: PostReader

isPost(uri: URI) kann zur Überprüfung eingesetzt werden, ob sich hinter einer URI ein Beitrag befindet.

getPost(uri: URI) ist dazu gedacht einen einzelnen Beitrag anhand seiner URI zu lesen. Sie liefert dann den Beitrag als Objekt der SIOC-Klasse `sioc:Post` zurück.

hasPost(uri: URI) funktioniert ähnlich wie `isPost`, überprüft aber ob sich hinter der angegeben URI noch weitere Beiträge befinden können.

pollPosts(uri: URI, since: Date, limit: int) ist eine Methode die alle Beiträge hinter eine URI liest, welche die übergeben Kriterien erfüllen. Insgesamt erhält diese Methode drei Parameter. Der Erste ist eine URI die den Ort angibt, von der die Beiträge gelesen werden sollen. Mit dem zweiten Parameter kann ein Zeitpunkt angegeben werden, ab dem ein zu lesender Beitrag geschrieben sein muss. Zum Beispiel der Zeitpunkt als diese Methode das letzte mal aufgerufen wurde, um alle Beiträge die danach geschrieben wurden zu lesen. Der letzte Parameter gibt eine obere Schranke an, wie viele Beiträge maximal pro Aufruf dieser Methode gelesen werden dürfen. Ist dieses Limit erreicht, werden keine weiteren Beiträge in die Ergebnisliste aufgenommen.

4.2.6 PostWriter

In Abbildung 4.14 ist ein UML-Sequenzdiagramm der `PostWriter`-Komponente zu sehen. Dort ist visualisiert, welche Schritte für das stellvertretende Schreiben von Beiträgen eines Benutzers unternommen werden müssen. Soll nun ein Beitrag in Plattform des Connectors geschrieben werden, wird die Methode `writePosts(uri, String, Syntax)` mit dem Zielort als URI, den Beiträgen als serialisierte RDF-Objekte und dem verwendeten Serialisierungsformat aufgerufen. Die folgenden Schritte werden dann für jeden `sioc:Post` in den serialisierten Daten durchgeführt:

Es beginnt damit, dass zuerst nach einem Benutzerkonto des Beitragautors für die Plattform des aktuellen Connectors gesucht wird. Dieser sollte im Idealfall im Triplestore des Connectors nach dem im Abschnitt 4.1.3 Schema vorliegen. Mit diesem Benutzerkonto, in Form der Klasse `sioc:UserAccount` aus SIOC, kann dann vom `ClientManager` ein Client für die verwendete API

geholt werden. Sollte die Suche negativ verlaufen, steht noch der Defaultclient zur Verfügung. Bevor der Beitrag aber geschrieben werden kann, muss nachgeschaut werden, ob eine Erlaubnis für das Schreiben mit diesem Client vorliegt. Ist dies der Fall kann der Beitrag in das von der API verwendete Format konvertiert und an die richtige Stellen der Plattform geschrieben werden.

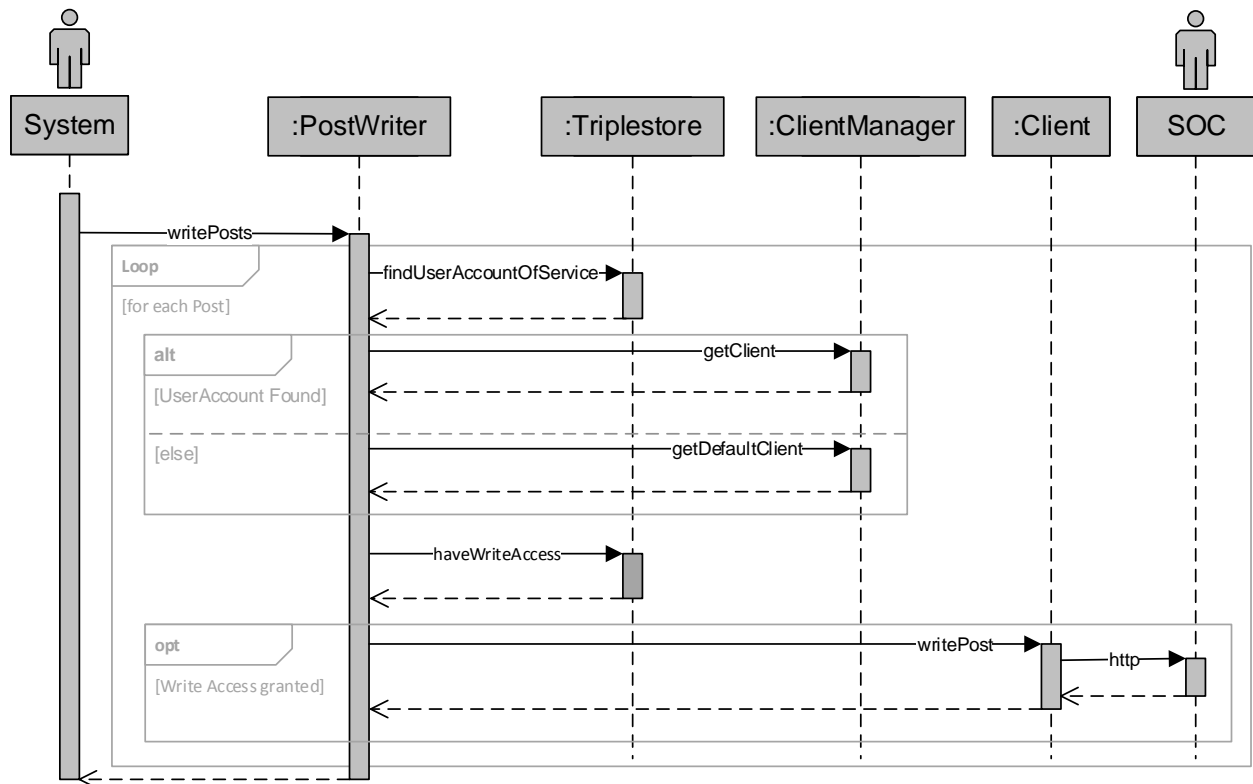


Abbildung 4.14.: UML-Sequenzdiagramm eines PostWriters

4.3 SOCC-Camel

SOCC-Camel ist eine Modul von SOCC für die Integration von Connectoren in Camel. Durch dieses ist auf flexible Weise möglich die gelesenen Beiträge von einer Plattform über die Connectoren in eine andere zu schreiben. Die Abbildung 4.15 zeigt SOCC-Camel als EIP-Diagramm.

Hauptklasse dieses Moduls ist *SoccComponent*. Sie ist für die Verwaltung von Connectoren und Erstellen der Endpunkte für Camel zuständig. Von dieser Klasse muss zuerst ein Objekt erzeugt und ein *SoccContext* mit dem Triplestore und alle Informationen für die Connectoren wie für deren Konfiguration, Benutzerkonten oder Servicebeschreibungen übergeben werden. Also alle Daten die in Abschnitt 4.1 beschrieben wurden. Das *SoccComponent*-Objekt wird dann unter einen frei wählbaren Namen in Camel registriert. Als Namen ist „socc“ aber vorzuziehen. Nun kann wie in Abschnitt 2.3.2 mit dieser Komponente Routen zusammengesetzt werden.

Die URIs für die Konfiguration der Endpunkte mit SOCC-Camel habe dazu den folgenden Aufbau: „socc://connectorId?uri=targetUri[¶meter...]“. Der Anfang der URI mit „socc://“ sagt Camel, dass es für diesen Endpunkt die Komponente benutzen soll, die vorher unter dem Namen „socc“ registriert wurde - also *SoccComponent*. Für den Platzhalter „{connectorId}“ muss eine gültige ID eines Connectors sein, dessen Konfigurationsdaten im Triple-

Store hinterlegt wurden. Über den Parameter „uri“ wird dann die Quell- beziehungsweise Ziel-URI für den Connector angegeben, von der Beiträge gelesen beziehungsweise in die Beiträge geschrieben werden sollen. Je nach Endpunkt können dann noch weitere Parameter angegeben werden. Ein Endpunkt wird in SOCC-Camel durch die Klasse `SoccEndpoint` modelliert (nicht explizit in der Abbildung angegeben) und leitet sich von der Camel-Klasse `ScheduledPollEndpoint`, die das Verwenden der folgenden `SoccPostPollConsumer` Klasse vereinfacht.

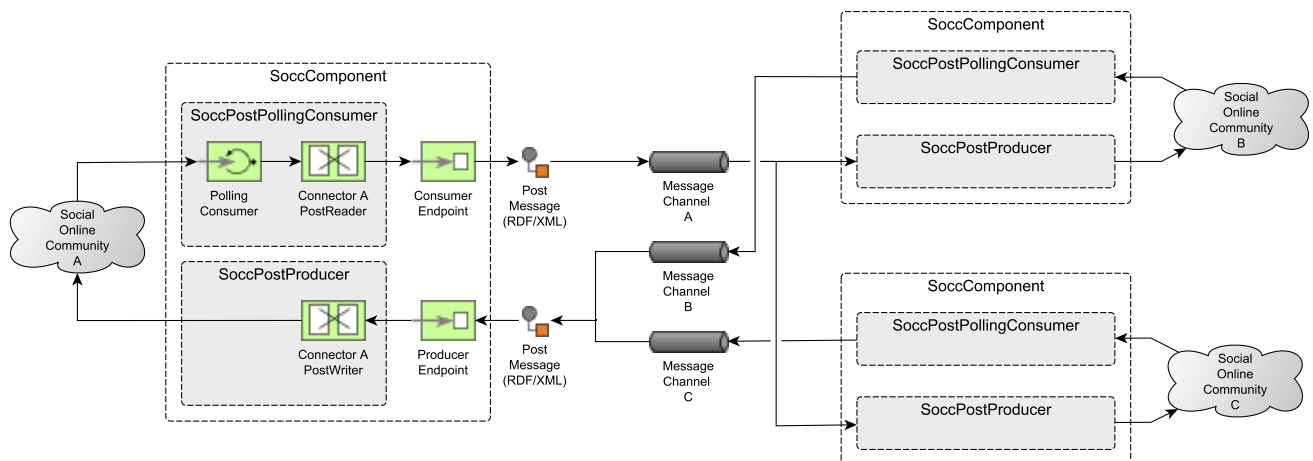


Abbildung 4.15.: Übersicht des SOCC-Camel Moduls als EIP-Diagramm.

4.3.1 SoccPostPollConsumer

Wird ein Endpunkt mit der Absicht zum Lesen von Beiträgen erstellt, erzeugt die Klasse `SoccEndpoint` ein Objekt der Klasse `SoccPostPollConsumer` und übergibt ihm die Parameter aus der Konfigurations-URI. Da `SoccPostPollingConsumer` sich von der Camel-Klasse `ScheduledPollConsumer` ableitet, ist es über den Parameter `delay` möglich in periodischen Abständen das Lesen von Beiträgen auszuführen. Die Angabe erfolgt dabei in Millisekunden. Der Parameter `limit` entspricht dabei dem gleichnamigen Argument der Methode `pollPosts(...)` des `PostReaders` aus Abschnitt 4.2.5. Das dort noch fehlende Argument `since`, für das Datum, ab wann ein Beitrag als neu gilt, wird mit dem Zeitpunkt belegt, wann die Methode `pollPosts(...)` das letzte Mal aufgerufen wurde. Alle neuen, gelesenen Beiträge werden am Ende in das RDF/XML-Format serialisiert und als Nachricht an Camel übergeben. Damit andere Komponenten diese Nachricht wieder korrekt de-serialisieren können, muss der Nachricht noch ein Header mit dem Schlüssel „Content-Type“ und dem MIME-Type⁶ vom RDF/XML „application/rdf+xml“ als Wert mitgegeben werden.

4.3.2 SoccPostProducer

Der `SoccPostProducer` ist das Gegenstück zum `SoccPostPollConsumer`. Er ist der Endpunkt, der zum Schreiben von Beiträgen in die Plattform des Connectors von der Klasse `SoccEndpoint` erzeugt wird. Intern verwendet er den `PostWriter` des betreffenden Connectors und leitet den Inhalt

⁶ Multipurpose Internet Mail Extension - Types: <http://tools.ietf.org/html/rfc2046>

der Nachricht an diesen weiter. Außer der Angabe über die Ziel-URI erhält der SoccPostProducer keine weiteren Parameter über die Konfigurations-URI. Der SoccPostProducer überprüft auch ob ein gültiger MIME-Type angegeben wurde. Akzeptiert wird auf jeden Fall „application/rdf+xml“ für RDF/XML. Je nach verwendeten Adapter für RDF2Go können auch „application/x-turtle“ für Turtle oder „application/rdf+json“ für RDFJson⁷ verwendet werden.

⁷ <https://dvcs.w3.org/hg/rdf/raw-file/default/rdf-json/index.html>



5 Implementierung und Proof of Concept

Dieses Kapitel beinhaltet Informationen über die Implementierung von Connectoren für die im Abschnitt 2.4 vorgestellten Plattformen. Diese wurden nicht nur ausgewählt, weil sie fast alle weit verbreitet sind, sondern weil sie auch verschiedenen Klassen von Plattformen angehören. Moodle und Canvas gehören zu der Klasse der Lernplattformen und beinhalten eine klassische Forenstruktur für Diskussionen. Die Plattformen Facebook und Google+ sind soziale Online-Netzwerke, die zurzeit einen großen Aufschwung erleben. Und zuletzt noch Youtube, das als Videoportal anderen das bereitstellen von Videomaterial und das Diskutieren über diese ermöglicht. Während alle fünf Plattformen auf REST (Moodle auch SOAP) als Architektur ihrer APIs setzten, sind die Formen für die Autorisierung teils sehr unterschiedlich, auch wenn sie den selben Mechanismus einsetzen.

In den folgenden Abschnitten wird erklärt wie man auf die Daten der einzelnen Plattformen über ihre APIs zugreifen kann und welche Voraussetzungen dafür erfüllt sein müssen. Gleichfalls wird beschreiben, wie der Zugriff auf diese API innerhalb der Programmiersprache Java funktionieren kann, da diese APIs in der Regel für Webanwendungen ausgelegt sind. Danach wird noch gezeigt welches Mapping zwischen den Ressourcen der Plattformen und dem SIOC-Format gewählt wurde. Traten bei der Implementierung Besonderheiten auf, die für andere Entwickler interessant sein könnten, wurden diese in einen eigenen Abschnitt des betreffenden Connectors aufgeführt.

Der Schluss des Kapitels bildet ein „Proof of Concept“ (Englisch für Machbarkeitsnachweis), der demonstrieren soll wie ein Programm zu Synchronisation von Beiträgen mit SOCC aussehen kann und wie es funktioniert.

5.1 Verwendete Programme und Bibliotheken

An dieser Stelle soll noch kurz auf zusätzliche Programme und Bibliotheken eingegangen werden, die für die Entwicklung der einzelnen Connectoren hilfreich waren.

Apache Maven: Für die Entwicklung der Programme und Bibliotheken dieser Arbeit wurde das Build-Management Programm Maven¹ von Apache eingesetzt. Mit diesem ist es auf einfache Art möglich Java-Programme zu Verwalten und zu Vrstellen. So ist nur mit einer einzigen XML-Datei (pom.xml) möglich die Abhängigkeiten festzulegen, das Programm zu kompilieren, zu testen und es auszuliefern. Maven schreibt eine feste Ordnerstruktur für ein Projekt vor wie der Produktivquellcode, dafür nötige Ressource (Bilder, Texte,...) und der Quellcode zum Testen geordnet werden sollen. Die Abhängigkeiten werden automatisch aus zentralen Repositorien heruntergeladen und in das Projekt eingebunden. So muss dies nicht umständlich von Hand geschehen und spart so Arbeit.

¹ <http://maven.apache.org/>

RDF2Go: Für die Programmiersprache Java gibt es einige Bibliotheken mit denen RDF-Daten verarbeitet werden können. Wie zum Beispiel *Sesame*², *Apache Jena*³ oder *JRDF*⁴. Jede dieser Bibliotheken bietet einen Triplestore und eine API für das Arbeiten mit RDF-Triples. Stellt sich nur die Frage welche man von ihnen auszuwählen soll?

Einen Ausweg aus dieser Misere bietet *RDF2Go*⁵. RDF2Go abstrahiert andere Triplestores über eine einheitliche API und erlaubt es den Triplestore später wechseln zu können. Dazu muss für den Triplestore nur ein Adapter für RDF2Go implementiert werden, welche aktuell für Sesame und Apache Jena vorhanden sind. Für RDF2Go existiert auch ein Programm *RDFReactor* mit dem aus Ontologien in RDFS und OWL Java-Klassen generieren werden können. So muss nicht jedes Objekt mühevoll Triple für Triple zusammengebaut werden.

Handy-URI-Templates: Bei der Arbeit mit RDF kommt man zwangsläufig um das Verarbeiten von URIs nicht herum. Um diese Arbeit zu erleichtern, wird bei der Implementierung die Java-Bibliothek *Handy-URI-Templates*⁶ eingesetzt. Diese implementiert den *RFC6570*⁷, welches eine Sprache für URI-Vorlagen definiert. Zum Erzeugen einer URI muss nur eine solche Vorlage definiert und mit der API von Handy-URI-Templates mit Werten gefüllt werden. Die Vorlage `http://www.example.org/{x,y}` würde so mit den Werten `x=42` und `y=23` die URI `http://www.example.org?x=42&y=23` entstehen lassen.

5.2 Implementierung der Connectoren

In diesen Abschnitt soll nun beschrieben werden, wie Connectoren für die in Abschnitt 2.4 vorgestellten Plattformen implementiert werden können. Dafür wird darauf eingegangen mit welcher Art von API auf die Ressourcen der einzelnen Plattformen zugegriffen werden und wie man deren Datenstruktur in SIOC abbilden kann. Ebenfalls werden auftretende Probleme und wenn möglich deren Lösung gezeigt.

5.2.1 Allgemeine Informationen zum Mapping nach SIOC

Innerhalb der Beschreibungen des Mappings nach SIOC werden, zur besseren Übersicht und einfacheren Lesbarkeit, Platzhalt für einige URIs benutzt. Welche Platzhalten dies sind und für welche URI sie stehen ist aus Tabelle 5.1 zu entnehmen.

Da einige Eigenschaften der Klassen von SIOC bei allen Plattformen mit ähnlichen Werten belegt werden und sich nur vom Aufruf der API unterscheiden, sollen zuvor für die SIOC-Klassen `sioc:UserAccount`, `sioc:Forum`, `sioc:Thread` und `sioc:Post` die Eigenschaften beschrieben werden, die überall verwendet werden.

sioc:UserAccount: Für die Klasse `sioc:UserAccount` ist die Angabe von vier Eigenschaften wichtig. Die erste ist `sioc:Id` mit der die ID des Benutzerkontos angegeben wird und

² <http://www.openrdf.org>

³ <http://jena.apache.org>

⁴ <http://jrdf.sourceforge.net/>

⁵ <http://semanticweb.org/wiki/RDF2Go>

⁶ <https://github.com/damnhandy/Handy-URI-Templates>

⁷ <http://tools.ietf.org/html/rfc6570>

`sioc:name` für den Benutzernamen. Aus Kompatibilitätsgründen mit FOAF und für den ClientManager wird die ID des Benutzers noch mit `foaf:accountName` angegeben und die URI zum verwendeten Service mit `foaf:accountServiceHomepage`. Die Angabe der Daten zu Authentifizierung erfolgt mit der Eigenschaft `siocsa:accountAuthentication`.

sioc:Forum: Jedes Forum hat seine eigene ID, die wie bei `sioc:UserAccount` unter der Eigenschaft `sioc:id` abgelegt wird. Ebenfalls bekommt jedes Forum einen Namen mit `sioc:name`. Jedes Forum hat eine `sioc:Site` von der es gehostet wird (`sioc:has_host`). Ein Forum kann als Container für Beiträge dienen, die mit der Eigenschaft `sioc:container_of` mit dem Forum verbunden werden. Zugleich kann es auch Threads mit `sioc:parent_of` enthalten.

sioc:Thread: Ein Thread enthält ebenfalls wieder eine ID und einen Namen und enthält nur Beiträge der Klasse `sioc:Post` mit `sioc:container_of` und ist immer ein Teil eines bestimmten Forums mit `sioc:has_parent`.

sioc:Post: Jeder Beitrag hat eine, für die Seite der Plattform eindeutige ID mit `sioc:id`. Mit der Eigenschaft `sioc:creator` wird der `sioc:UserAccount` des Autors festgelegt und mit `sioc:content` der Inhalt des Beitrags. Besitzt ein Beitrag einen Titel, wird dieser mit `sioc:title` gespeichert. Der Zeitpunkt der Erstellung kommt nach `dcterms:created` und der bei einer Modifikation nach `dcterms:modified`. Beiträge gehören immer zu einem Forum oder Thread, welches mit `sioc:has_container` angegeben wird. Auch können Beiträge Kommentare mit `sioc:has_reply` enthalten oder selber Kommentare auf andere Beiträge mit `sioc:reply_of` sein.

Tabelle 5.1.: Allgemeine Platzhalter und deren Beschreibung für das Mapping nach SIOC

Platzhalter	Bedeutung
{rootUri}	Die Basis-URI einer Webseite (Schema + Authority ⁸). Für Facebook wäre dies zum Beispiel <code>https://www.facebook.com</code>
{siteUri}	URI für eine <code>sioc:Site</code>
{serviceUri}	URI für einen <code>sioc:Service</code>
{userAccountUri}	URI für einen <code>sioc:UserAccount</code>
{forumUri}	URI für ein <code>sioc:Forum</code>
{threadUri}	URI für ein <code>sioc:Thread</code>
{postUri}	URI für ein <code>sioc:Post</code>

5.2.2 Allgemeine Probleme und Lösungen bei der Implementierung

Bei der Implementierung der Connectoren treten ab und zu Probleme auf, die unabhängig von der verwendeten API gelöst werden mussten. Hier sollen diese Probleme und deren Lösung vorgestellt werden.

⁸ <http://tools.ietf.org/html/rfc3986#section-3>

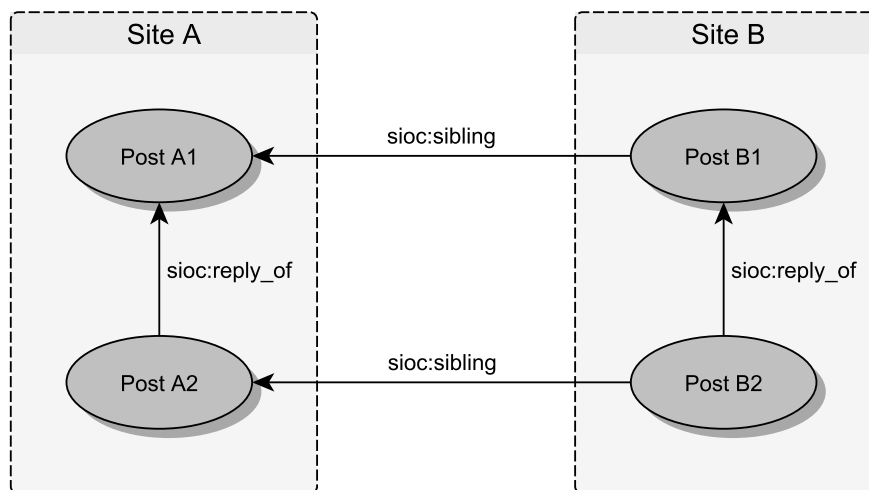
Doppelte Beiträge beim Synchronisieren

Sollen zwei Container auf unterschiedlichen Plattformen synchronisiert werden, tritt automatisch das Problem auf, dass Beiträge die von A nach B geschrieben wurden beim auslesen von B wieder bei A als „neuer“ Beitrag landen würde, obwohl er ursprünglich von dort kam. Dies würde ohne Eingreifen zu einer Dauerschleife führen, bei der die selben Beiträge immer hin und her geschrieben werden.

Aus diesem Grund wird vor dem Schreiben eines Beitrags eine Art Wasserzeichen zum Inhalt hinzugefügt. Dieses Wasserzeichen besteht aus der Zeichenkette - forwarded by SOCC siteUri -. Der Platzhalter „{siteUri}“ wird dabei durch die URI der Seite ersetzt von der der Beitrag gekommen ist. Hierzu muss jedem Beitrag mit der Eigenschaft `dc:terms:isPartOf` die URI des `sioc:Site`-Objekts zu dem der Beitrag gehört, hinzugefügt werden. Wird nun eine Beitrag gelesen, der ursprünglich von der Seite des Connectors kam, wird dieser ignoriert. Enthält dagegen der Beitrag schon eine Wasserzeichen mit der URI einer anderen Seite, wird kein neues Wasserzeichen hinzugefügt.

Rekonstruktion der Kommentarstruktur

Bei der Synchronisation ist es ebenso wichtig die Struktur von Kommentaren beizubehalten. Wenn Beitrag A2 ein Kommentar auf Beitrag A1 in Seite A muss diese Struktur beim Synchronisieren auf Seite B mit den Beiträgen B1 und B2 erhalten bleiben.



@prefix sioc: <http://rdfs.org/sioc/ns#> .

Abbildung 5.1.: Erhalten der Kommentarstruktur beim Schreiben von Beiträgen

Hierzu wird nach dem Schreiben eines Beitrags mit der Eigenschaft `sioc:sibling` (Englisch für Geschwister) die URI zum original Beitrag beim geschriebenen Beitrag festgehalten. Taucht nun beim Schreiben ein Beitrag auf, der ein Kommentar auf den mit `sioc:sibling` gespeicherten

Beitrags ist, kann dieser an die richtige Stelle als Kommentar geschrieben werden. Der RDF-Graph nach dem Synchronisieren ist in Abbildung 5.2.2 zu sehen.

Leider funktioniert diese Methode erst wenn die entsprechenden Beziehungen vorhanden sind. Werden Beiträge von Seite A zur Seite B geschrieben funktioniert dies tadellos. Wird aber nun auf Seite B ein Kommentar auf einen dieser Beiträge geschrieben und zu Seite A geschickt, kann diese den Beitrag nicht richtig einfügen, da `sioc:reply_of` natürlich auf einen Beitrag von Seite B zeigt und die `sioc:sibling` Informationen auf Seite A fehlen.

5.2.3 MoodleConnector

Der *MoodleConnector* die die Implementierung eines Connectors für Moodle. Da Diskussionen in Moodle in Foren geführt werden, war ein Mapping ohne Probleme möglich. Jedoch der Zugriff über die vorhandene Webservice API gestaltete sich mehr als schwierig.

Moodle Webservice API

Moodle bringt von Haus aus eine Webservice-Schnittstelle für den Zugriff auf seine Daten mit. Diese API unterstützt die Protokolle REST, SOAP und oder XML-RPC⁹. Mit ihr können Kurse, Foren und deren Threads (in Moodle Discussion Topics genannt), Benutzerprofile, Gruppen und viele andere Ressourcen ausgelesen oder geschrieben werden. Leider gilt dies nicht für Beiträge innerhalb von Threads. Die aktuelle Version (Beim verfassen dieser Arbeit war dies 2.5.1) erlaubt es nicht Beiträge weder zu lesen noch zu schreiben.

Doch es existiert ein Plugin für Moodle, dass auf dieser Webservice API aufbaut und sie erweitert - *MoodleWS*¹⁰. MoodleWS bietet den Zugriff über REST und SOAP an, wobei bei der Verwendung von REST JSON als Datenformat verwendet wird und nicht XML wie mit SOAP. Mit diesem Plugin ist ein Programm nun auch in der Lage Beiträge aus Threads zu lesen. Die REST-Schnittstelle hat aber noch eine paar Fehler und so können damit noch keine neuen Beiträge geschrieben werden. Mit SOAP ist dies aber problemlos möglich.

Für die Benutzung der API muss sich ein Client mit dem Benutzernamen und Passwort eines vorhandenen Benutzers anmelden. Ist dies erfolgreich, bekommt der Client einen zufällig ID und einen Sitzungsschlüssel die er bei jeder Abfrage angeben muss. Dieser Sitzungsschlüssel ist nur eine begrenzte Zeit gültig. So kann es passieren, dass eine Operation deswegen fehlschlägt und der Client sich neu anmelden muss.

Für die Verwendung der API in Java wird auch eine Bibliothek *moodle_ksoap2*¹¹ zur Verfügung gestellt. Sie abstrahiert komplett das SOAP-Protokoll im Hintergrund und enthält für alle Moodle-Ressource Java-Klassen und Methoden für den Zugriff. Bevor *moodlews_ksoap2* benutzt werden kann, ist darauf zu achten in Moodle unter „Site administration“, „Development“, „OK Tech Webservices (aka wspp)“ die Einstellung „Use an auto generated wsdl“ des MoodleWS Plugins auszuschalten, da es sonst zu Problemen mit *moodlews_ksoap2* kommen kann.

⁹ <http://xmlrpc.scripting.com/>

¹⁰ <https://github.com/patrickpollet/moodlews>

¹¹ https://github.com/patrickpollet/moodlews_ksoap2

Listing 5.1: MoodleWS mit moodlews_ksoap2: Lesebeispiel

```
1 Mdl_soapserverBindingStub client = new Mdl_soapserverBindingStub(  
2     MOODLE_URI + "/wspp/service_pp2.php",  
3     MOODLE_URI + "/wspp/wsdl2",  
4     false );  
5  
6 LoginReturn loginReturn = client.login(username, password);  
7  
8 ForumRecord[] forumRecords = client.get_all_forums(  
9     loginReturn.getClient(),  
10    loginReturn.getSessionKey(),  
11    "",  
12    "" );  
13  
14 for(ForumRecord forum : forumRecords){  
15     System.out.println(forum.getId() + " " + Forum.getName());  
16 }
```

Wie in der ersten Zeile von Listing 5.1 zu sehen, wird als Client ein Objekt der Klasse `Mdl_soapserverBindingStub` aus `moodlews_ksoap2` verwendet. Dem Konstruktor wird die URI auf die Datei „service_pp2.php“ (Zeile 3) und der XML-Namespace der WSDL-Datei des MoodleWS-Plugins übergeben. Das dritte Argument besagt, ob der Client im Debugmodus laufen soll. In Zeile 6 meldet man sich über den Client mit einem Benutzernamen und Passwort bei MoodleWS an und bekommt ein Objekt der Klasse `LoginResult` mit der oben angesprochenen ID für den Client und einen Sitzungsschlüssel zurück. Mit der Methode `get_all_forums(...)` in Zeile 8 kann nun ein Array mit allen Foren angefordert werden, auf die der angemeldete Benutzer Zugriff hat. Die ersten zwei Argumente dieser Methode sind die ID des Clients und der Sitzungsschlüssel. Mit den letzten zwei Argumenten kann das Ergebnis gefiltert werden. Dazu wird als erstes ein Spaltenname aus der Foren-Tabelle der Moodle-Datenbank angegeben und als zweites welcher Wert dieser haben soll. Am Ende werden dann alle Ergebnisse in den Zeilen 14 bis 16 mit der ID und Name des Forums ausgegeben.

Listing 5.2: MoodleWS mit moodlews_ksoap2: Beitrag schreiben

```
1 ForumPostDatum postDatum = new ForumPostDatum(client  
2     .getBindingStub()  
3     .getNAMESPACE());  
4  
5 postDatum.setMessage("Was kam bei der Aufgabe raus?");  
6 postDatum.setSubject("Aufgabe 2.1");  
7  
8 client.forum_add_reply(  
9     client.getAuthClient(),  
10    client.getSessionKey(),  
11    postId,  
12    postDatum);
```

Beiträge können nur als Kommentare mit der ID des kommentierten Beitrags geschrieben werden. Alle Daten für den Beitrag werden in einem Objekt der Klasse `ForumPostDatum` gespeichert,

wie in der ersten Zeile des Listings 5.2 zu sehen ist. Das Argument ist der beim Client angegebene XML-Namespace der WDSL-Datei. Als Daten können wie in Zeile 5 und 6 nur eine Nachricht (Message) und ein Thema (Subject) übergeben werden. Dieses Objekt wird dann an die Methode `forum_add_reply` mit der ID des Clients, dem Sitzungsschlüssel und der ID des Beitrags der kommentiert werden soll übergeben und dann zur Moodleinstanz übertragen.

Mapping von Moodle nach SIOC

Die interne Struktur von Diskussionen in Moodle entspricht der klassischen eines Forums. Das Forum enthält alle Beiträge zu einem übergeordneten Thema. Einzelne Unterthemen teilen sich dann vom Forum in Threads auf, die dann die einzelnen Beiträge enthalten. Die Abbildung 5.2 zeigt die Zuordnung von Moodle- zu SIOC-Klassen. Das Forum in Moodle entspricht dem `sioc:Forum`, dieses enthält mehrere Objekte der Klasse `ForumDiscussion`, welche dem `sioc:Thread` zugeordnet werden können. Die Beiträge werden von Moodle mit der Klasse `ForumPost` modelliert und stimmen mit der SIOC-Klasse `sioc:Post` überein. Besonderheit bei den Beiträgen in Moodle ist, dass alle Beiträge in einem Thread entweder ein Kommentar auf den ersten Beitrag im Thread oder auf andere Kommentare ist.

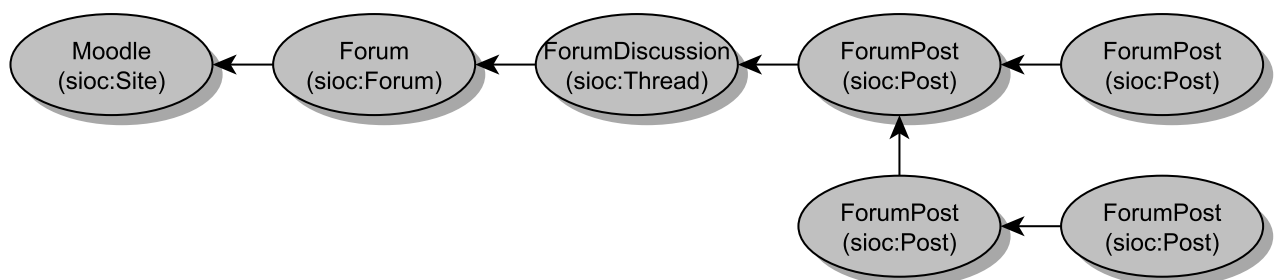


Abbildung 5.2.: Struktur von Moodle und Mapping nach SIOC

Für die Authentifizierung der Benutzerkonten werden für Moodle der entsprechende Benutzername und das Passwort benötigt. Diese werden als Credential für den `waa:Direct` Authentifizierungsmechanismus gespeichert, wie es in der Abbildung 5.3 zu sehen ist. Für die Servicebeschreibung mit der Klasse `sioc:Service` aus dem SIOC Services Modul ist die Angabe der Adresse der Moodleinstanz mit der Eigenschaft `siocs:serviceEndpoint` für die API äußerst wichtig. Sie bestimmt den Endpunkt an dem die API-Abfragen gesendet werden.

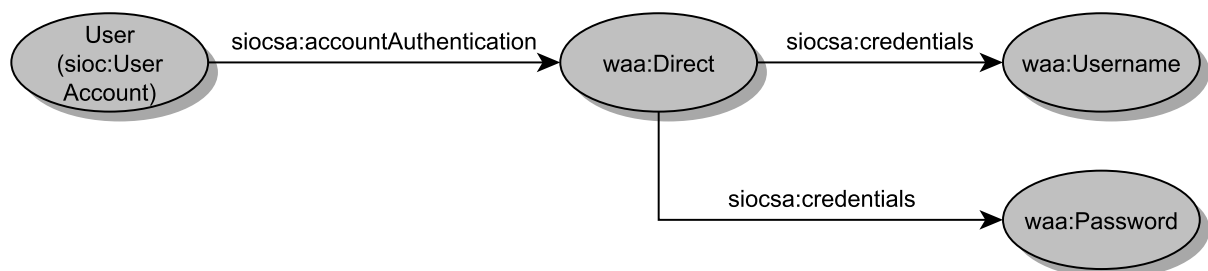


Abbildung 5.3.: Authentifizierungsdaten für Moodle in SIOC

Die URIs für die Objekte der SIOC-Klassen für Moodle entsprechen den URIs der graphischen Weboberfläche, da SOAP keine expliziten URIs für Ressourcen besitzt. Das genaue Format ist in der Tabelle 5.2 beschrieben. Der Platzhalter {rootId} entspricht den Wert der Eigenschaft `siocs:serviceEndpoint` aus der Servicebeschreibung.

Tabelle 5.2.: Format der URIs für Moodle in RDF

SIOC Klasse	URI-Format
<code>sio:Site</code>	{rootUri}
<code>sio:Service</code>	{rootUri}/wspp
<code>sio:UserAccount</code>	{rootUri}/user/profile.php?id={userId}
<code>sio:Forum</code>	{rootUri}/mod/forum/view.php?id={forumId}
<code>sio:Thread</code>	{rootUri}/mod/forum/discuss.php?d={discussionId}
<code>sio:Post</code>	{rootUri}/mod/forum/discuss.php?d={discussionId}#p{postId}

Besonderheiten bei der Implementierung des MoodleConnectors

Der Connector für Moodle war wohl einer der herausforderndsten bei der Implementieren von allen fünf Connectoren. Nicht nur weil erst nach mehreren Test sich herausstellte, dass die favorisierte REST-Schnittstelle von MoodleWS kein Schreiben von Beiträgen möglich macht, sondern dass auch die Java-Bibliothek `moodlews_ksoap2` so ihre kleinen Eigenheiten hat.

Zuerst mussten die Klassen für `moodlews_ksoap2` komplett neu generiert werden, das die auf GitHub bereitgestellten Klassen nicht kompatibel zur aktuellsten WSDL-Datei des MoodleWS Plugins waren. Der passende Generator wird aber vom Entwickler gleich mitgeliefert und ist mit dem folgenden Befehl schnell erledigt.

Listing 5.3: Neugenerierung von `moodlews_ksoap2`

```
1  java  org.ksoap2.wsdl.WSDL2ksoap -p de.m0ep.moodlews.soap -o /tmp http://localhost/moodle/wspp/wsd1_pp2.php
```

Mit dem Parameter „-p“ wir das Java-Package für die Java-Klassen festgelegt und mit „-o“ der Ordner in dem die Klassen geschrieben werden. Die URI am Ende gibt den Pfad zur Datei „wsdl_pp2.php“ vom MoodleWS-Plugin an. Diese URI ist dann an die jeweilige Moodleinstanz anzupassen.

Eine sehr nervende Eigenheit von `moodlews_ksoap2` ist die Tatsache, dass Java-Exceptions und Gründe für Fehler gerne verschluckt und nicht weitergegeben werden. So ist es zum Beispiel unmöglich zu wissen, ob eine Operation aufgrund eines Netzwerkfehlers, eines abgelaufenen Sitzungsschlüssels oder falschen Daten fehlschlug. Aus diesem Grund wurde der Client von `moodlews_ksoap2` für den ClientManager (siehe Abschnitt 4.2.3) in eine extra Klasse `Moodle2ClientWrapper` gepackt und um Mechanismen zur Fehlererkennung und Wiederherstellung des Clients erweitert. Alle Funktionsaufrufe des Clients wie `get_all_forums(...)`

werden dazu zuerst in die Methode `call` eines Objekts der Java-Schnittstelle `Callable<V>`¹² verpackt, so dass diese Funktion, ohne zu wissen um welche es sich genau handelt, mehrfach aufgerufen werden kann. Dieses Objekt wird dann an die Methode `callMethod(...)` der Klasse `Moodle2ClientWrapper` übergeben. Diese Methode ruft dann zuerst die `call()`-Methode des `Callable`-Objekts auf und prüft ob ein gültiges Ergebnis zurückgegeben wurde. Dies ist der Fall wenn es ungleich `null` ist. Ist dies nicht der Fall, wird geschaut ob die Moodleinstanz erreichbar ist. Sollte hier alles in Ordnung sein, muss geschaut werden ob der Sitzungsschlüssel abgelaufen ist. Als Test reicht es hier aus zu testen, ob die Methode `get_my_id(...)` des `moodlews_ksoap2` Clients eine Zahl ungleich `Null` zurück gibt. Diese liefert die ID des aktuell angemeldeten Benutzers zurück und im Fehlerfall ist dieser gleich `Null`. Ist dieser Test positiv, also der Sitzungsschlüssel abgelaufen, wird versucht sich neu anzumelden und die Anfrage erneut ausgeführt. Ansonsten wird ein Fehler gemeldet.

Eine Funktion mit der nur ein Beitrag mit einer bestimmten ID abgefragt werden könnte fehlt ebenfalls. Wenn nur ein Beitrag geladen werden soll, müssen zunächst alle innerhalb des selben Threads angefordert und diese nach dem passenden Beitrag durchgesehen werden.

Als ungewöhnlich kann auch der Inhalt der Klasse `ForumDiscussionRecord` bezeichnet werden, welche die Informationen zu einem Thread enthält. Normalerweise würde man erwarten, dass die Methode `getId()` die ID des Threads zurück geben wird. Doch stattdessen entspricht der Wert der ID des ersten Beitrags dieses Threads. An die richtige ID kommt man erst heran, wenn man sich mit der Methode `getPost()` ein Objekt des ersten Beitrags und darin den Rückgabewert von `getDiscussion()` benutzt.

5.2.4 CanvasConnector

Für den Zugriff auf die Lernplattform Canvas über SIOC wurde der *CanvasConnector* implementiert. Wie schon bei Moodle war das Mapping nach SIOC aufgrund der Forenstruktur unkompliziert. Das gleiche gilt auch für die bereitgestellte REST-API die einwandfrei funktioniert. Da aber noch keine API für Java existierte, musste erst eine eigene entwickelt werden. Diese unterstützt leider noch nicht alle Ressourcen von Canvas, ist aber für die Zwecke des CanvasConnectors erst einmal völlig ausreichend.

Canvas REST-API

Der öffentliche Zugriff auf die Daten der Lernplattform Canvas basiert auf einer REST-API und OAuth 2.0 zur Autorisierung. Die Adresse, über die auf die API zugegriffen werden kann, besteht aus der URI für die verwendete Canvasinstanz (`{rootUri}`) gefolgt von dem Pfad `„/api/v1“`. Für die Demoinstanz von Instructure würde dies der URI `https://canvas.instructure.com/api/v1` entsprechen. An diese können dann weitere Pfade für den Zugriff auf die einzelnen Ressourcen angehängt werden. Zur Autorisierung wird von OAuth nur der Accesstoken benötigt, der bei der REST-Abfrage als HTTP Authorization Header mitgeschickt wird. Listing 5.4 zeigt eine Beispielanfrage und Angabe des Accesstokens mit dem Programm *curl*¹³. Der Platzhalter `{accessToken}`

¹² <http://docs.oracle.com/javase/7/docs/api/index.html?java/util/concurrent/Callable.html>

¹³ <http://curl.haxx.se/>

muss dann natürlich erst durch einen validen Accesstoken ersetzt werden. Diesen kann jeder Benutzer in seinem Canvas-Profil unter „Settings“ und im Abschnitt „Approved Integrations“ selbst erstellen. Die Angabe von ClientId und ClientSecret sind nicht nötig.

Listing 5.4: Canvas Authorization Header

```
1 curl -H "Authorization: Bearer {accessToken}" https://canvas.instructure.  
com/api/v1/courses
```

Als Datenformat für die zurückgelieferten Daten wird von Canvas auf JSON gesetzt. Für die Verwendung von POST und PUT Operationen, zum Schreiben nach Canvas, können die Daten entweder im *HTML Form Encoding*¹⁴ Standard oder ebenfalls in JSON angegeben werden.

Da einige Abfragen eine Liste von Ergebnissen zurückliefern und diese möglicher Weise sehr lang werden können, teilt Canvas diese Listen auf mehrere Seiten auf, die jede einzeln abgefragt werden müssen. Für jede Seite schickt Canvas mehrere URIs als *HTTP Link Header*¹⁵ der Antwort mit. Diese URIs erhalten zusätzlich noch ein Attribut *rel* mit, das beschreibt in welcher Relation die URI zu dieser Seite steht. Als Wert für diese Relation können „current“ für eine URI auf die aktuelle, „next“ auf die nächste, „prev“ auf die vorherige, „first“ auf die erste und „last“ für eine URI auf die letzte Seite vorkommen. Um also die nächste Seite vom Ergebnis zu bekommen, muss eine neue REST-Abfrage mit der URI aus dem Link mit der Relation „next“ ausgeführt werden. Fehlt eine URI mit dieser Relation, ist die letzte Seite des Ergebnisses erreicht.

CanvasLMS4J

Ein Problem mit der API von Canvas war es, dass es zwar eine gute REST-basierte Anbindung gab, aber noch keine Bibliothek, um sie mit der Programmiersprache Java anzusprechen. Es musste also erst eine eigne Java API entwickelt werden, die den Namen *CanvasLMS4J* (Kurzform für „Canvas LMS API für Java“) bekam.

Anhand der für die REST-API verwendeten URIs ist auffällig, dass die einzelnen Bestandteile aufeinander aufbauen. Zum Beispiel ist der Ablauf für den REST-Zugriff auf DiscussionTopics in Gruppen und Kursen der gleiche, nur der Anfang der verwendeten URIs unterscheidet sich. Aus diesem Grund wurden die einzelnen Ressource (Course, Groupe, DiscussionTopic, Entries, ...) als einzelne Endpunkte implementiert, die sich von der Klasse *EndPoint* ableiten. Jeder Endpunkt kann einen Eltern-Endpunkt haben, wobei sich die endgültige URI für die REST-Abfrage aus dem Pfad des Eltern-Endpunktes und dem des aktuellen Endpunktes zusammensetzt. Zur Verdeutlichung sei hier die URI `https:{canvasUri}/api/v1/courses/1/discussion_topics` als Demonstration genannt. Sie besteht aus den statischen Teil `https:{canvasUri}/api/v1/` der den Ort für die verwendete Canvasinstanz angibt. Darauf folgt ein Kurs als erster Endpunkt mit der Kurs-ID „1“. Für diesen Kurs sollen nun alle Diskussionen abgefragt werden. Dies geschieht durch die Angabe des zweiten Endpunktes `/discussion_topics`. der Pfad `„/courses/1“` bildet hier also den Eltern-Endpunkt von `/discussion_topics`. Sollen aber nun alle Diskussionen in einer Gruppe abgefragt werden, reicht es aus den Kurs-Endpunkt durch einen Gruppen-Endpunkt auszutauschen: `https:{canvasUri}/api/v1/groups/42/discussion_topics`.

¹⁴ <http://www.w3.org/TR/html4/interact/forms.html#h-17.13.4>

¹⁵ <http://www.w3.org/Protocols/9707-link-header.html>

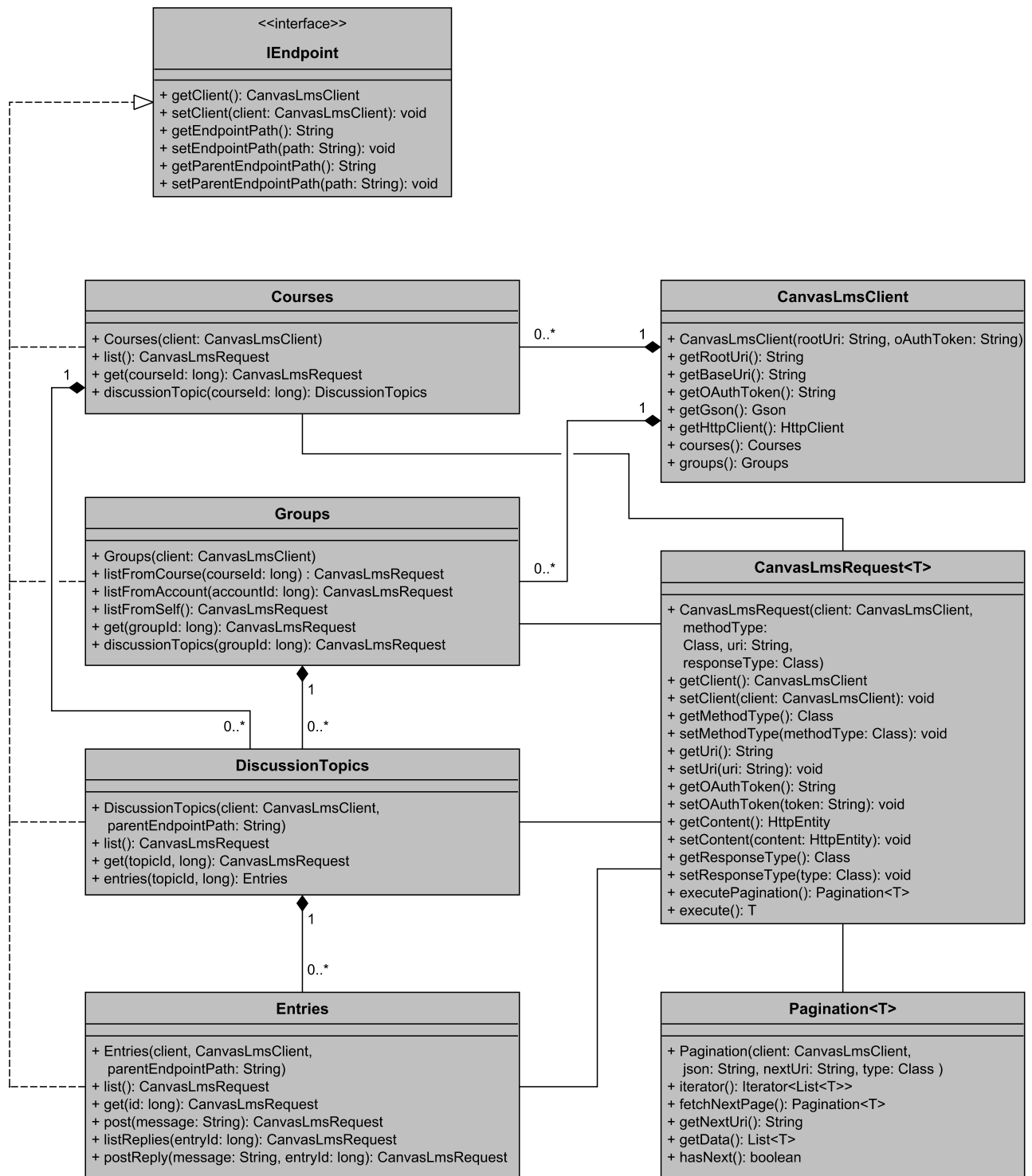


Abbildung 5.4.: UML Klassendiagramm von CanvasLMS4J

Ausgangspunkt für CanvasLMS4J ist der Client mit der Klasse `CanvasLmsClient` (siehe Abbildung 5.4). Über sie werden alle Endpunkte verwaltet, die keinen Eltern-Endpunkt besitzen. Bei der Erzeugung eines Objektes dieser Klasse, werden ihr die URI zur verwendenden Canvasinstanz (ohne „/api/v1“) und der Accesstoken des Benutzerkontos übergeben. Im aktuellen Stadium können von einem Client aus nur auf Endpunkte für Kurse oder Gruppe zugegriffen werden.

Endpunkte können dann mit der Klasse `CanvasLmsRequest` REST-Anfragen an die Canvas API stellen. Hierzu verwendet CanvasLMS4J die `HttpClient`¹⁶ Bibliothek von Apache mit der einzelne HTTP-Operationen ausgeführt werden können. Um die im JSON-Format zurück gelieferten Antworten der von Canvas in Java verwenden zu können, wird auf die Funktionen der von Google entwickelten `Gson`¹⁷ Bibliothek zurück gegriffen. Sie erlaubt es Java Objekte in das JSON-Format zu konvertieren und genauso aus Daten in JSON ein Java-Objekt zu erzeugen (eine passende Klasse muss aber vorhanden sein). Dadurch verringert sich der Aufwand für das Verarbeiten der Daten von Canvas auf das Erstellen der entsprechenden Java-Klassen. `CanvasLmsRequests` können mit zwei verschiedenen Methoden ausgeführt werden. Die erste `execute()` wird dazu verwendet, wenn die REST-Abfrage nur die Rückgabe eines Objektes zur Folge hat. Also zum Beispiel, wenn nur die Daten eines bestimmten Kurses abgefragt werden sollen. Die zweite Methode ist `executePagination`. Diese Methode dient für Abfragen die eine in Seiten aufgeteilte Liste von Ergebnissen zurückliefern und gibt für eine einfache Handhabbarkeit ein Objekt der Klasse `Pagination` zurück.

Diese Klasse `Pagination` ist nach dem Iterator-Muster aufgebaut und liefert vom kompletten Ergebnis bei jedem Iterationsschritt eine einzelne Seite zurück. Diese Seite enthält dann je eine Teilliste der angefragten Objekte. Ist eine Seite fertig ausgelesen, holt `Pagination` über eine vorgegeben REST-Abfrage die nächste Seite. Dies geschieht solange bis keine Seiten mehr geladen werden können oder das Programm das Lesen abbricht.

Listing 5.5: CanvasLMS4J Beispielprogramm

```
1 CanvasLmsClient client = new CanvasLmsClient(  
2     "https://canvas.instructure.com",  
3     "7~LUpV7B3lJY...");  
4  
5 Pagination<DiscussionTopic> discussionPages = client.courses()  
6     .discussionTopics( 798152 )  
7     .list()  
8     .executePagination();  
9  
10 for ( List<DiscussionTopic> discussions : discussionPages ) {  
11     for ( DiscussionTopic discussion : discussions ) {  
12         System.out.println( discussion );  
13     }  
14 }
```

In Listing 5.5 ist ein Beispiel für die Anwendung der CanvasLMS4J API zu sehen. In den ersten drei Zeilen wird eine neuer `CanvasLmsClient` für die Canvasinstanz auf `https://canvas.instructure.com` und einem `AccessToken` erstellt. Als nächstes soll für einen

¹⁶ <http://hc.apache.org/httpclient-3.x/>

¹⁷ <https://code.google.com/p/google-gson/>

Kurs mit der ID „798152“ alle Diskussionen aufgelistet werden. Mit dem Aufruf der Methode `courses()` des Clients wird der Endpunkt für die Kurse und von diesem aus der Endpunkt für die Diskussionen im Kurs „798152“ abgefragt. In der siebten Zeile wird die Art der Abfrage genauer festgelegt. Da eine Liste aller Diskussionen gesucht ist, wird die Methode `list()` aufgerufen die ein `CanvasLmsRequest`-Objekt für die gewünschte Abfrage erstellt. Da die Antwort aus mehrere Objekten mit der Beschreibung der einzelnen Diskussionen besteht, wird die Anfrage in Zeile 8 mit der Methode `executePagination()` ausgeführt. Die einzelnen Seiten des Ergebnisses werden dann in der zehnten Zeile mit einer For-Schleife durchlaufen. Das nachladen der Seiten erfolgt dabei automatisch durch die `Pagination`-Klasse. Jede Seite besteht nun aus einer Liste mit den Beschreibung der Diskussionen in einem `DiskussionTopic`-Objekt, welche dann wiederum in einer weiteren Schleife ausgegeben werden.

Mapping von Canvas nach SIOC

Die Struktur von Canvas ist der von Moodle sehr ähnlich (siehe Abbildung 5.5), nur dass es explizit keine Foren gibt sondern die Diskussionen innerhalb der Kurse stattfindet. Demzufolge kann ein Kurs (`Course`) der SIOC-Klasse `sioc:Forum` zugeordnet werden. Dies gilt ebenfalls für Gruppen, in denen separate Diskussionen geführt werden können. Die Diskussionsthemen in Canvas werden durch die Klasse `DiscussionTopic` modelliert und entspricht `sioc:Thread`. Die Klasse für Beiträge in Canvas ist `Entry` und ist entweder ein Kommentar auf den ersten Beitrag einer Diskussion oder eins auf dessen Kommentare. Die SIOC-Klasse für einen `Entry` entspricht demzufolge `sioc:Post`.

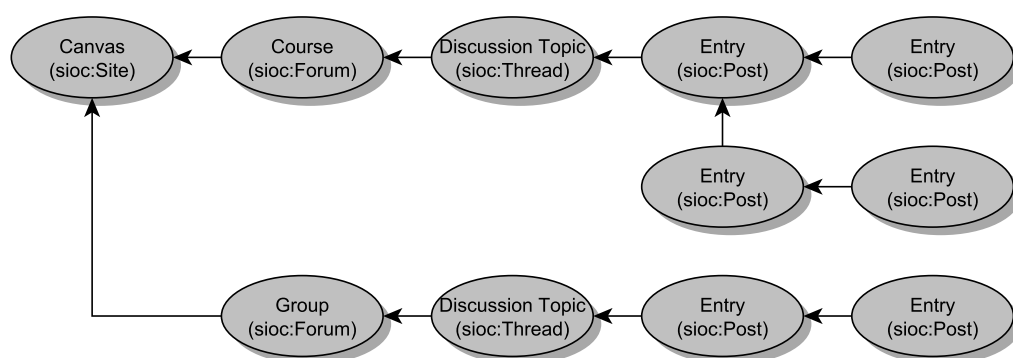


Abbildung 5.5.: Struktur von Canvas und Mapping nach SIOC

Zum Benutzen der REST-API von Canvas muss eine Accesstoken für den betreffenden `sioc:UserAccount` des Benutzerkontos vorliegen. Wie in Abbildung 5.6 wird der Accesstoken dem `sioc:UserAccount` mit `siocsa:accountAuthentication` als Credential der Klasse `waa:OAuth` mitgegeben. In der Servicebeschreibung ist wieder die Angabe von `siocsa:serviceEndpoint` mit der Adresse der Canvasinstanz für die API Voraussetzung für den Betrieb des `CanvasConnectors`.

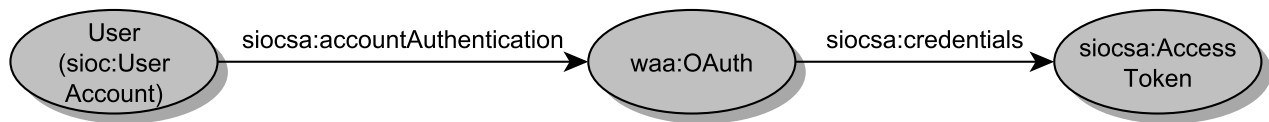


Abbildung 5.6.: Authentifizierungsdaten für Canvas in SIOC

Die URIs für RDF bei Canvas wurden so gewählt, dass sie den URIs für die REST-Abfragen entsprechen. An ihnen ist sehr leicht zu erkennen, wo die Ressource sich innerhalb der Canvass-Struktur befindet, da sie aufeinander aufbauen. Das einzige Problem liegt nur in der URI für den Startbeitrag einer Diskussion, da dieser nur innerhalb des DiskussionTopics existiert und keine eigene ID besitzt. Aus diesem Grund wurde für die URI des Startbeitrags die Zusammensetzung aus der URI eines DiscussionTopics mit angehängten Fragment „#discussion_topic“ gewählt. Dieser Zusatz wurde nicht zufällig ausgewählt. Die ID „discussion_topic“ wird von Canvas innerhalb der HTML-Darstellung für genau diesen Beitrag verwendet.

Tabelle 5.3.: Format der URIs für Canvas in RDF

SIOC Klasse	URI-Format
sioc:Site	{rootUri}
sioc:Service	{rootUri}/api/v1
sioc:UserAccount	{rootUri}/api/v1/users/{userId}/profile
sioc:Forum	{rootUri}/api/v1/courses/{courseId} <i>oder</i> {rootUri}/api/v1/groups/{groupId}
sioc:Thread	{forumUri}/discussion_topics/{topicId}
sioc:Post	{threadUri}/entries/{entryId} <i>oder</i> {threadUri}#discussion_topic

5.2.5 FacebookConnector

Dieser Abschnitt beschreibt, wie man auf die Daten von Facebook zugreifen kann, um einen FacebookConnector zu entwickeln. Zuerst wird dazu die *Graph API* von Facebook vorgestellt. Danach erfolgt eine Beschreibung der Java-Bibliothek *RestFB*, welche das Verwenden der Graph API aus Java heraus ermöglicht. Zum Abschluss folgt noch das Mapping der Facebook Ressourcen in die entsprechenden SIOC-Klassen.

Facebook Graph API

Der primäre Weg an die von Facebook gespeicherten Daten zu gelangen, geht über die *Graph API* (siehe. [16]). Sie erlaubt einen REST-basierten Zugriff auf den sogenannten *Social Graph* von Facebook. Dieser Graph enthält alle Beziehungen die ein Person auf Facebook zwischen

anderen Personen, Gruppen, Seiten, Beiträge, Fotos und so weiter besitzt. Der einfachste Weg zum Kennenlernen der Graph API ist der *Graph API Explorer*¹⁸. Er ist eine Onlineanwendung für REST-Abfragen an die Graph API, um schnell Sachen auszuprobieren.

Jede Ressource im Social Graph von Facebook hat seine eigene, eindeutige ID. Die URI für den REST-Zugriff auf eine Ressource besteht aus `https://graph.facebook.com/` gefolgt von der ID und optionalen Parametern. Dies macht die Form der URIs zwar sehr einheitlich, aber es ist so unmöglich anhand der URI herauszufinden welcher Typ von Ressource sich dahinter befindet. Dies widerspricht zwar nicht die am Anfang von Kapitel 4 beschreiben Prinzipien von Berners-Lee macht aber die spätere Verarbeitung etwas komplizierter.

Graph API Datenformat

Als Datenformat setzt Facebook mit der Graph API auf das weit verbreitete JSON-Format. Neben den eigentlichen Daten enthalten Ressourcen noch Verbindungen zu anderen Ressourcen. Diese Verbindungen werden „Connections“ genannt. Um zu lesen welche Connections eine Ressource besitzt, muss bei der Abfrage der Parameter `metadata=1` angehängt werden. Ist dies geschehen, enthalten die JSON-Daten einen neuen Eintrag `metadata` und darin unter dem Eintrag `connections` eine Liste der vorhandenen Connections. Die wichtigsten Connections für den FacebookConnector sind `feed` und `comments`. Die Connection `feed` sagt über die Ressource aus, dass sie eine Liste von Beiträgen enthält und in sie Beiträge geschrieben werden können. Das könnte zum Beispiel die Wall einer Person oder die einer Gruppe sein. Dahingegen zeigt die Connection `comments` an, dass die betreffende Ressource Kommentare enthält und auch kommentiert werden kann. Um die Beiträge eines Feeds lesen zu können, muss nur die URI der Ressource mit dem Pfad `/feed` erweitert werden → „`http://graph.facebook.com/me/feed`“. Die ID „`me`“ ist dabei eine spezielle ID, die sich auf den aktuell angemeldete Benutzer bezieht. Möchte man nun nicht alle Daten einer Ressource abfragen, sondern zum Beispiel nur den Name und das Datum der letzten Änderung, kann der Parameter „`fields`“ der URI hinzugefügt werden. Als Wert wird dann eine Liste mit den Namen der gesuchten Datenfelder angegeben, wobei jeder Name durch ein Komma getrennt wird.

Facebook Login

Die Graph API verwendet zur Autorisierung für die Abfragen OAuth 2.0. Auch *Facebook Login*¹⁹ genannt. Accesstoken bei Facebook nur für zwei Stunden gültig. Es ist aber möglich einen *Long-Lived Accesstoken*²⁰ zu erstellen, der zwei Monate gültig ist. Ist ein Accesstoken irgendwann abgelaufen, muss vom Benutzer ein neuer erstellt werden. Da aber die Graph API hauptsächlich nur für Webanwendungen gedacht ist, kann dies ein Desktopprogramm nur mit dem Umweg über einen Internetbrowser machen. Für das Anlegen eines neuen Accesstokens werden vom Programm eine OAuth 2.0 ClientID und das ClientSecret benötigt. Diese bekommt man von Facebook, wenn man für sein Programm auf der Webseite `https://developers.facebook.com/apps` eine „App“ anlegt. Die Daten befinden sich dann in den „Einstellungen“ unter „Grundlegendes“ und werden dort „App ID“ und „App Secret“ genannt.

¹⁸ <https://developers.facebook.com/tools/explorer>

¹⁹ <https://developers.facebook.com/docs/facebook-login/>

²⁰ <https://developers.facebook.com/docs/facebook-login/access-tokens/>

Jeder Accesstoken hat nur Zugriff auf bestimmte Bereiche der Graph API, die beim Erstellen des selbigen als „Permission“²¹ (engl. für Berechtigungen), beziehungsweise von OAuth 2.0 „Scope“ (engl. für Geltungsbereich) genannt, angegeben werden muss. Es ist absolut wichtig diese Berechtigungen auch in den App-Einstellungen unter „Berechtigungen“ anzugeben. Ohne diese Angabe verweigert Facebook das Verwenden der dazugehörigen Ressourcen. Die für SOCC wichtigen Berechtigungen sind in der Tabelle 5.4 beschrieben.

Tabelle 5.4.: Für SOCC wichtige Facebook Berechtigungen

Berechtigung	Beschreibung
<i>read_stream</i>	Lesen von Beiträgen und Kommentaren eines Benutzers.
<i>publish_actions</i>	Schreiben von Beiträgen und Kommentaren für einen Benutzer.
<i>user_groups</i>	Zugriff auf die Liste der beigetretenen Gruppen eines Benutzers.

Um nun einen Accesstoken für einen Benutzer zu Erstellen, muss in einen Internetbrowser folgende URI eingegeben werden:

```
https://www.facebook.com/dialog/oauth?client_id={appId}
&redirect_uri={redirectUri}&responseType=code&scope={scopeList}.
```

Anstatt des Platzhalters {appId}, muss dann die oben angesprochene ClientID beziehungsweise App ID eingetragen werden. Mit dem Parameter redirect_uri wird bei dieser URI eine Adresse angegeben, auf die der Internetbrowser nach der Anfrage mit der Antwort umgeleitet wird. Dies kann die Adresse eines externen Webservers oder die eines Webservers den die Anwendung nur für diesen Zweck laufen lässt sein. Einen solchen Webserver kann man mit der Jetty²² Bibliothek für Java erstellen und würde dann als redirect_uri die URI http://localhost:port/ angeben. Der Parameter responseType=code besagt, dass diese Anfrage einen Code zurück liefert mit dem dann der eigentliche AccessToken abgeholt werden kann. Dies ist der Standardweg. Mit dem letzten Parameter scope können dann noch die oben erwähnten Berechtigungen übergeben werden, wobei alle durch eine Komma getrennt hintereinander zu schreiben sind. Wird dann diese URI im Browser aufgerufen, wird er danach bei Erfolg auf die Adresse redirectUri?code=code umgeleitet. Der Platzhalter {code} würde dann einen zufälligen Code darstellen, mit dem über eine weitere Anfrage mit der folgenden URI der Accesstoken geholt werden kann:

```
https://www..facebook.com/oauth/access_token?client_id={appId}
&redirect_uri={redirectUri}&client_secret={appSecret}&code={code}
```

Der Platzhalter {appId} und {appSecret} entsprechen wieder der ClientID und ClientSecret. Bei dieser URI ist es sehr wichtig, dass {redirectUri} den selben Wert hat, wie er weiter oben schon einmal angegeben wurde. Für {code} muss dann noch der oben zurückgeliefert Code angegeben werden. Für diese Abfrage ist aber kein Internetbrowser mehr nötig. Als Antwort wird

²¹ <https://developers.facebook.com/docs/reference/login/#permissions>

²² <http://www.eclipse.org/jetty>

dem Programm dann die Daten `access_token=accessToken&expires=secondsTilExpiration` zurück geliefert. Der Wert des Parameters `access_token` ist der gewollte Accesstoken und der Wert von `expires` ist die Haltbarkeit des Accesstokens in Sekunden.

Soll nun die Haltbarkeitsdauer des Tokens auf zwei Monate verlängert werden muss eine neue Anfrage mit der folgende URI gemacht werden:

```
https://www.facebook.com/oauth/access_token?grant_type=fb_exchange_token
    &client_id={appId}&client_secret={app-secret}
    &fb_exchange_token={accessToken}
```

Die Werte für die einzelnen Parameter dürften nun offensichtlich sein. Als Antwort kommt dann ein Accesstoken mit verlängerter Haltbarkeit zurück. Dabei kann es passieren, dass er genau so aussieht wie der alte, was aber kein Problem darstellt.

RestFB

Facebook selber bietet direkt keine API für die Verwendung der Graph API in Desktopprogrammen an. Speziell für die Programmiersprache Java existiert von Facebook nur eine API für das mobile Betriebssystem Android²³. Als Alternative dazu kann die *RestFB*²⁴ Java-Bibliothek von Mark Allen verwendet werden. RestFB bietet dabei die Möglichkeit von Java aus die Graph API von Facebook zu benutzen. Eine Abfrage könnte wie in Listing 5.6 aussehen.

Listing 5.6: RestFB Beispielprogramm

```
1  FacebookClient client = new DefaultFacebookClient(USER_ACCESS_TOKEN);
2
3  User user = client.fetchObject("me", User.class);
4  System.out.println("User name: " + user.getName());
5
6  Connection<Post> myFeed = client.fetchConnection("me/feed", Post.class);
7  for (List<Post> feedPage : myFeed){
8      for (Post post : feedpage){
9          System.out.println("Post: " + post);
10     }
11 }
```

In Zeile Eins wird eine Objekt der Klasse `FacebookClient` mit einem Accesstoken als Parameter erstellt, über dem die Methoden für die Abfragen aufgerufen werden können. Die Zeile 3 zeigt eine solche Abfrage nach den Benutzerdaten für den aktuell angemeldeten Benutzer mit der speziellen ID „me“. Der Anfangsteil der URI für die Graph API „`https://graph.facebook.com`“ muss dabei weggelassen werden. Zurück gegeben wird ein Objekt der Klasse `User` und über dies wird dann der Benutzername ausgegeben. Die Angabe der Klasse mit `User.class` ist ebenfalls essenziell, so weiß RestFB in welche Klasse es die von der Graph API zurückgelieferten JSON-Daten konvertieren soll. Zeile 6 zeigt noch wie man an die Daten einer Connection, hier der Beiträge auf der Wall des aktuellen Benutzers, gelangen kann. Enthält eine Antwort eine Liste

²³ <https://developers.facebook.com/docs/android/>

²⁴ <http://restfb.com/>

mit zu vielen Einträgen, wird die Antwort auf mehrere Seiten aufgeteilt und ein Verweis auf die nächste Seite der aktuellen Seite mitgegeben. Die Klasse `Connection` sorgt dann alleine dafür, dass die neue Seite gelesen wird, wenn die alte abgearbeitet wurde. Die erste `for`-Schleife in Zeile 7 iteriert dann über alle vorhandenen Seiten und die zweite in Zeile 8 über alle Einträge auf der aktuellen Seite.

Listing 5.7: RestFB: Schreiben von Beiträgen auf eine Wall

```
1 FacebookType result = client.publish(  
2     userId + "/feed",  
3     FacebookType.class,  
4     Parameter.with("message", "Tolles Wetter heute \o/"));
```

Beiträge können bei Facebook nur an Ressourcen geschrieben werden, die eine „feed“ oder „comment“ Connection haben. In Listing 5.7 ist zu sehen wie ein neuer Beitrag auf die Wall eines Benutzers geschrieben werden kann. Zu diesem Zweck enthält der Client aus RestFB die Methode `publish(...)` zum Schreiben von Daten in Ressourcen. Das erste Argument ist die Zielressource, die in Zeile 2 aus der ID des Benutzers gefolgt von einem „/“ und dem Namen der Connection, hier „feed“, besteht. Da `publish(...)` bei Erfolg die ID des erstellten Beitrags zurückliefert, wird als Klasse für das erwartete Ergebnis die Basisklasse `FacebookType` angegeben. Als letztes muss noch der Inhalt des Beitrags festgelegt werden. Dies geschieht nicht im JSON-Format, sondern wird als Parameter in der URI für die HTTP-POST-Operation angegeben. RestFB bietet dazu die Klasse `Parameter` an, mit der ein Parameter „message“ und dem Inhalt des Beitrags als Wert erzeugt wird.

Facebook erlaubt mit der Graph API das Schreiben von Beiträgen auf die Wall eines Benutzers nur, wenn das beim Schreiben das verwendete Benutzerkonto mit dem der Wall übereinstimmt. Also das stellvertretende Schreiben auf eine fremde Wall ist nicht möglich. Diese Einschränkung gilt aber nicht für Walls von Facebook-Gruppen.

Mapping von Facebook nach SIOC

Die Struktur von Facebook baut auf Walls von Benutzern, Gruppen und Facebook-Seiten auf. Darin können Beiträge geschrieben und diese kommentiert werden. Solch eine Wall entspricht der SIOC-Klasse `sioc:Forum`. Diese enthält aber keine weiteren Threads, sondern die Beiträge und Kommentare als `sioc:Post` Objekte. Abgesehen von Facebook-Seiten können Kommentare in Facebook zum aktuellen Zeitpunkt nicht kommentiert werden. Abbildung 5.7 zeigt dies noch einmal graphisch mit einer Gruppen-Wall als Beispiel.

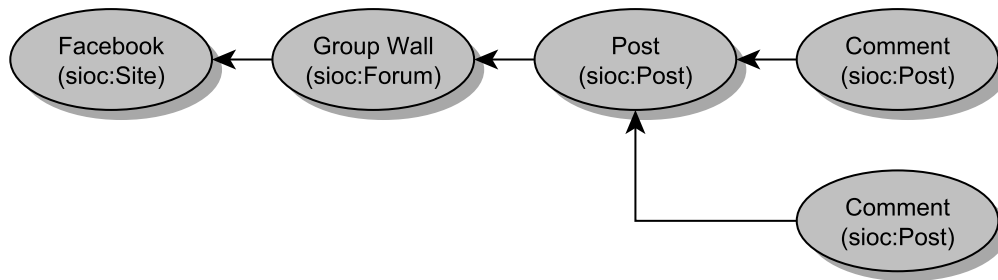


Abbildung 5.7.: Struktur von Facebook und Mapping nach SIOC

Wie weiter oben erklärt, verwendet Facebook für die Autorisierung der Abfragen in der Graph API OAuth 2.0. Die Benutzerkonten, die für das stellvertretende Schreiben verwendet werden sollen, müssen also einen Accesstoken für das Konto enthalten. Das gleiche gilt für die Servicebeschreibung für die Graph API mit der `siocs:Service` Klasse. Die Eigenschaft `siocsa:serviceAuthentication` zeigen auf ein Objekt der Klasse `waa:OAuth` das die ClientID und das ClientSecret enthält, wie es in der Abbildung 5.8 zu sehen ist.

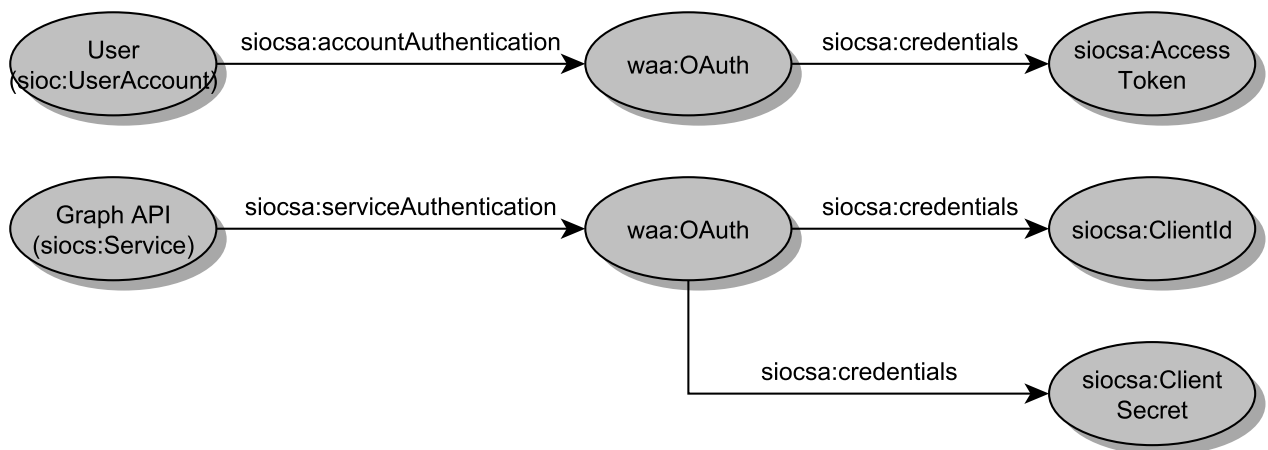


Abbildung 5.8.: Authentifizierungsdaten für Facebook in SIOC

Die in RDF verwendeten URIs für die Klassen `sioc:Post` und `sioc:UserAccount` von Facebook bekommen das Format „`https://graph.facebook.com/{facebookId}`“, wobei `{facebookId}` der ID der Ressource entspricht. Ausnahme bildete die URIs für die Klasse `sioc:Forum`. Da die ID eines Benutzers sowohl für sein Konto als auch für seine Wall stehen würden, wird der URI für die Wall noch der Pfad „`/feed`“ angehängt. Die URI „`https://graph.facebook.com/{facebookId}/feed`“ ist dann immer noch gültig und liefert von der Graph API die enthaltenen Beiträge zurück. Als URI für die Klasse `sioc:Site` wurde „`https://www.facebook.com`“ und für `siocs:Service` wurde „`https://graph.facebook.com`“ gewählt.

Besonderheiten bei der Entwicklung des FacebookConnectors

Wie im Abschnitt über die Graph API schon beschrieben, ist es nahezu unmöglich an einer URI von Facebook zu erkennen, welcher Typ von Ressource sich dahinter befindet. Deshalb muss bei jeder unbekannten URI erst einmal alle Informationen mit dem Parameter `metadata=1` heruntergeladen werden und anhand der vorhandenen Connections ermittelt werden, ob es sich um einen Beitrag, eine Wall oder was komplett anderes handelt. Deshalb ist es wichtig solche Erkenntnisse sofort in den Triplestore zwischenspeichern, um erneute Abfragen zu vermeiden.

5.2.6 GooglePlusConnector

Der GooglePlusConnector ist die Implementierung eines Connectors für das soziale Online-Netzwerk Google+. Die Struktur ist der von Facebook in etwa gleich, wo bei Google sie anders umgesetzt hat. Google+ verwendet, wie viele andere Plattformen auch, eine REST-basierte API mit JSON als Datenformat und OAuth zur Autorisierung der Abfragen. Leider ist zurzeit nur das Lesen auf öffentliche Beiträge, aber kein Schreiben möglich. Trotzdem wurde für diese Plattform ein Connector implementiert, da gehofft wurde, dass bis zum Ende dieser Arbeit die noch zum Schreiben fehlenden Operationen fertig sein würden, was sich aber nicht bewahrheitet hatte.

Google+ API

Die API von Google für den Zugriff auf sein soziales Online-Netzwerk (siehe [21]) baut wie viele andere auch auf den REST-Prinzipien auf. Die URIs für die Anfragen beginnen mit `https://www.googleapis.com/plus/v1/` und danach folgt der Pfad zu den Ressourcen mit etwaigen Parametern. Die wichtigsten Parameter sind `access_token`, zum Angeben eines OAuth Access-Tokens und `parameterFields`, mit dem nur die dort angegebenen Datenfelder im Ergebnis zurückgeliefert werden. Als Datenformat wird JSON eingesetzt, aber mit einem von Facebook vollkommen unterschiedlichen Aufbau. Beiträge, die als `Activity` und `Comments` bezeichnet werden, bestehen im Grunde aus drei Teilen:

Actor: Der Actor-Eintrag sagt über die Ressource aus, von wem sie erstellt wurde.

Verb: Das Verb beschreibt, auf welche Weise diese Ressource erstellt wurde. Für `Activity` ist die Angabe von „post“, wenn ein Beitrag vom Actor selber geschrieben oder „share“, wenn dieser nur geteilt wurde und von jemand anderem stammt.

Object: Der Object-Eintrag enthält den eigentlichen Inhalt.

Neben diesen drei enthält eine Ressource noch Metadaten wie ID, Zeitpunkt der Veröffentlichung oder Änderung, Ortsdaten und andere mehr.

Die Google+ API teilt zu lange Ergebnislisten, ebenfalls wie Facebook, in mehrere Teillisten, die auf einzelne Seiten aufgeteilt werden. Ob eine Seite der aktuellen folgt, kann über die Existenz des `nextPageToken` Eintrags erkannt werden. Dieser `PageToken` muss für die Abfrage der nächsten Seite an die gleiche URI als Parameter `pageToken` angehängt werden.

Wie schon erwähnt setzt die Google+ API für die Autorisierung auf OAuth 2.0. Ein Accesstoken ist aber nur nötig, wenn Operationen stellvertretend für einen Benutzer ausgeführt werden müssen. Für alle anderen Fälle ist ein API-Schlüssel vollkommen ausreichend. Zusätzlich zum Accesstoken liefert Google+ auch einen Refreshtoken mit. So können abgelaufene Accesstoken vom Programm automatisch wieder reaktiviert werden, ohne dass der Benutzer dies von Hand machen muss. Die für die Erstellung von Access- und Refreshtoken gebrauchten ClientID und ClientSecret werden über die *Google API Console*²⁵ erstellt. Wichtig dabei ist eine „Client ID for installed applications“ auszuwählen. Sowohl für den Zugriff auf die REST-API, als auch für die Erstellung der Token, stellt Google eine Java-Bibliothek zur Verfügung.

Listing 5.8: Google+ API Java: Access- und Refreshtoken

```
1 GoogleAuthorizationCodeFlow flow = new GoogleAuthorizationCodeFlow.Builder(  
2     new NetHttpTransport(),  
3     new JacksonFactory(),  
4     CLIENT_ID,  
5     CLIENT_SECRET,  
6     Arrays.asList("https://www.googleapis.com/auth/plus.login"))  
7     .build();  
8  
9 Credential credentials = new AuthorizationCodeInstalledApp(  
10     flow,  
11     new LocalServerReceiver())  
12     .authorize("user");
```

Listing 5.8 zeigt wie man sich Access- und Refreshtoken für Google+ in Java holen kann. Von Zeile Eins bis 9 wird ein sogenannter „AuthorizationCodeFlow“ erzeugt. Dieser abstrahiert für das Programm den OAuth-Mechanismus, über den man die Token von Google bezieht. In der Zeile 2 und 3 bekommt dieser CodeFlow externe Objekte übergeben mit dem er HTTP-Operationen ausführen und die zurückgelieferten Daten verarbeiten kann. In den zwei Folgezeilen werden die ClientID und das ClientSecret übergeben. In der sechsten Zeile muss, wie schon bei Facebook, ein Geltungsbereich (Scope) für den Accesstoken festgelegt werden. In diesem Falle wollen wir mit „https://www.googleapis.com/auth/plus.login“ Operationen stellvertretend für einen Benutzer ausführen. In Zeile 8 wird das Code-Flow-Objekt über die build()-Methode erzeugt. Für Desktopprogramme wird wieder ein externer Internetbrowser zum Holen der Token benötigt. Doch auch dafür bietet Google eine fertige Klasse an, die (fast) alle Schritte automatisiert. Diese Klasse heißt AuthorizationCodeInstalledApp. Ihr werden der obige CodeFlow und ein lokaler Webserver (ebenfalls schon fertig vorhanden) übergeben und über die Methode authorize(...) die Token erzeugt. Die Zeichenkette „user“ ist nur von Bedeutung, wenn man dem CodeFlow noch einen „DataStore“ für die Accesstoken mitgibt, was hier weggelassen wurde. .

Listing 5.9: Google+ API: Zugriff auf Activity-Feed mit Java

```
1 Plus plus = new Plus.Builder(  
2     new NetHttpTransport(),  
3     new JacksonFactory(),  
4     credential)  
5     .setApplicationName(APPLICATION_NAME)
```

²⁵ <https://code.google.com/apis/console>

```

6      .build();
7
8      ActivityFeed activityFeed = plus.activities()
9          .list( "me","public" )
10         .execute();
11
12     for(Activity activity : activityFeed.getItems()){
13         System.out.println(
14             activity.getActor().getDisplayName()
15             + ": "
16             + activity.getObject().getContent());
17     }

```

Das kleine Beispiel in Listing 5.9 zeigt wie auf einen Activity-Feed, also die Liste der geschriebenen Beiträge eines Benutzers, in Google+ mit Java zugegriffen werden kann. Von Zeile Eins bis 6 wird der Client für den Zugriff auf Google+ mit einem Builder²⁶ gebaut. Als Argument wird wieder das Objekt einer Klasse für die Verwendung von HTTP-Operationen und eines für das Arbeiten mit JSON übergeben. Als dritter Parameter in Zeile 4 kommen noch die Credentials aus Listing 5.8 mit dazu. Mit der Methode `setApplicationName(...)` kann noch ein Name für die Anwendung angegeben werden bevor der Client mit der Methode `build()` in Zeile 6 erstellt wird. Der Client enthält mehrere Methoden mit denen man auf die verschiedenen Ressource wie Activitys, Kommentare und Personen zugreifen kann. In Zeile 8 wird die Methode `activities()` eingesetzt, um an eine Hilfsklasse für alle Activitys zu gelangen, welche die Operationen zum Lesen eines Activity-Feeds, nur eines einzelnen Activitys oder der Suche nach anderen Activitys enthält. Mit `list("me", "public")` wird eine Abfrage für den „public“ Activity-Feed (nur dieser ist aktuell verfügbar) des Benutzers mit der ID „me“ (Steht immer für den aktuell angemeldeten Benutzer) erstellt, die dann mit dem Aufruf der Methode `execute()` des zurückgelieferten Objekts ausgeführt wird. Das Ergebnis dieser Abfrage enthält ein Array von Activitys, das von der Methode `getItems()` zurückgegeben wird. Zum Schluss werden von allen Activitys der Name des Autors und der Inhalt ausgegeben. Ist ein oben erwähnter Pagetoken für eine weitere Ergebnisseite vorhanden, kann dieser an das Objekt, was von `list("me", "public")` zurückgegeben wird, mit der Methode `setPageToken(...)` übergeben werden bevor `execute()` ausgeführt wird.

Mapping von Google+ nach SIOC

Die Struktur in Google+ ist annähernd mit der von Facebook gleich. Jeder Benutzer hat seinen eigenen ActivityFeed auf dem er Beiträge in der Form von Activitys veröffentlichen kann. Demzufolge entspricht ein ActivityFeed der SIOC-Klasse `sioc:Forum`, wobei dieser die Activitys als `sioc:Post` enthält. Kommentare auf Activitys werden ebenfalls `sioc:Post` zugeordnet, wobei sie sich immer genau auf ein Activity beziehen und niemals auf ein anderen Kommentar (Siehe Abbildung 5.9).

²⁶ „Builder“-Entwurfsmuster: <http://www.oodeesign.com/builder-pattern.html>

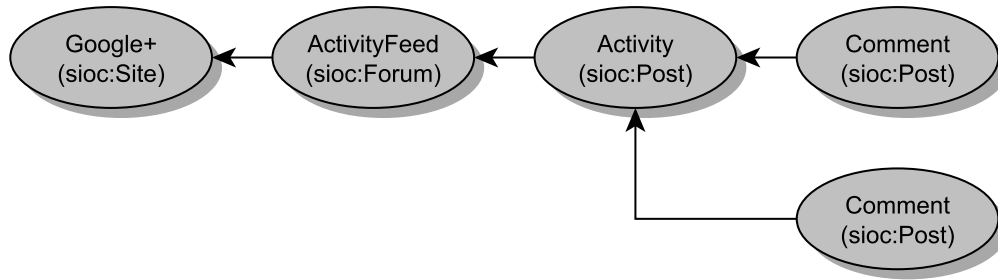


Abbildung 5.9.: Struktur von Google+ und Mapping nach SIOC

Wie schon Facebook setzt Google+ auf OAuth 2.0 für die Google+ API. Jedoch verwendet Google+ für den Client noch einen zusätzlichen RefreshToken, um die Accesstoken automatisch aktualisieren zu können, falls diese abgelaufen sind (Siehe Abbildung 5.10). Bei der Servicebeschreibung muss wieder die ClientID und das ClientSecret als Credentials einem Objekt der Klasse `waa:OAuth` mit der Eigenschaft `siocsa:serviceAuthentication` mitgegeben werden.

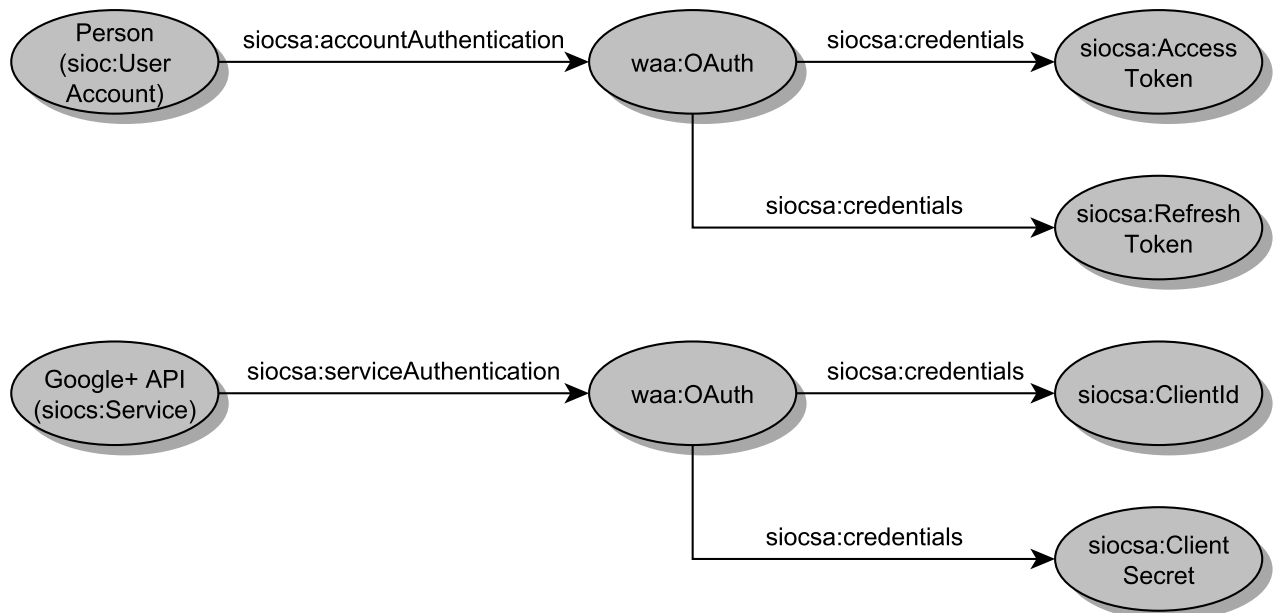


Abbildung 5.10.: Authentifizierungsdaten für Google+ in SIOC

In Tabelle 5.5 stehenden URIs für die Google+ Ressourcen in RDF entsprechen selben wie für die REST-Abfragen. Die einzige Einschränkung ist, dass in der aktuellen Version der Google+ API bei der URI für den ActivityFeed (`sioc:Forum`) für den Platzhalter „{collection}“ nur der Wert „public“ zulässig ist.

5.2.7 YoutubeConnector

Der YoutubeConnector ist der letzte Connector, das als Demonstration für SOCC implementiert wurde. Google ist zum Zeitpunkt an dem diese Arbeit geschrieben wurde dabei seine API und Youtube allgemein umzubauen. Da die neue API noch nicht 100% alle Funktionen unterstützt

Tabelle 5.5.: Format der URIs für Youtube in RDF

SIOC Klasse	URI-Format
sioc:Site	http://plus.google.com/
sioc:Service	https://www.googleapis.com/plus/v1
sioc:UserAccount	https://www.googleapis.com/plus/v1/people/{userId}
sioc:Forum	{userAccountUri}/activities/{collection}
sioc:Thread	https://www.googleapis.com/plus/v1/activities/{activityId}
sioc:Post	https://www.googleapis.com/plus/v1/comments/{commentId}

musste eine älter Version eingesetzt werden. Die Verwendung der API und das Mapping nach SIOC gingen aber ohne große Probleme.

Youtube Data API

Für Youtube existieren zurzeit zwei verschiedene APIs, eine Version 3 und die ältere Version 2. Die *Youtube Data API v3* befindet sich noch im Entwicklungsstadium. Dort ist es zum Beispiel noch nicht möglich Kommentare von Video zu lesen, geschweige denn zu schreiben. Aus diesem Grund muss für den YoutubeConnector die älter *Youtube Data API v2* benutzt werden, welche diese Funktionen noch bietet.

Die Youtube Data API v2 baut auf dem *Google Data Protokoll* auf. Eine API nach dem REST-Prinzip mit Atom und JSON als Datenformat. Für Youtube wird aber primär Atom eingesetzt (umschalten auf JSON ist möglich). Die einzelnen Ressourcen können als Atom-Feed, also eine Liste einzelnen Einträgen wie die einer Playliste, oder als ein einzelnes Entry-Element, zum Beispiel Informationen zu einen Benutzerkonto, zurückgegeben werden. Youtube verwendet aber nicht nur die vom Atom-Standard definierten Elemente, sondern ergänzt das Format durch eine große Zahl eigener Erweiterungen wie Spieldauer eines Videos, Statistiken, Bewertungen oder Informationen zu Benutzerkonten. Jeder Feed/Eintrag enthält mehrere Link-Element die auf weiterführenden Ressourcen verweisen. Ist die Menge der Einträge eines Feedes zu groß, wird das Ergebnis auf mehrere Feeds aufgeteilt. Um zwischen diesen Teilergebnissen zu navigieren, enthalten die Teilfeeds Links auf ihren Vorgänger- und/oder Nachfolge-Feed (siehe Listing 5.10).

Listing 5.10: Youtube Data API v2: Navigation durch die Ergebnisse

```

1 <link rel='previous' type='application/atom+xml'
2   href='http://gdata.youtube.com/feeds/api/videos?start-index=1&max-results
   =25...' />
3 <link rel='next' type='application/atom+xml'
4   href='http://gdata.youtube.com/feeds/api/videos?start-index=51&max-
   results=25...' />

```

Alle Ressourcen auf die öffentlich zugegriffen werden kann, brauchen keine vorherige Anmeldung. Sollen aber zum Beispiel Kommentare geschrieben werden, ist eine Anmeldung notwendig.

Für der Programm muss ein ein API-Schlüssel (von Google „Developer Key“ genannt) im *Product Dashboard*²⁷ erstellt werden. Dieser API-Schlüssel wird dann jeder Anfrage als URI Parameter `key=apiKey` angehängt. Soll auch stellvertretend für einen Benutzer Kommentare geschrieben werden, ist noch zusätzlich der Benutzername und das Password von ihm Voraussetzung.

Google bietet selber eine Java API für die Youtube Data API v2 an, mit der das Auslesen, Verarbeiten und Senden der Atom Feeds und Einträge sehr einfach ist. Das Programmbeispiel in Listing 5.11 zeigt eine Beispiel für die Verwendung der Youtube Data API in Java. Ausgangspunkt in Zeile Eins ist die Klasse `YouTubeService` die als Client eingesetzt wird. Bei der Erzeugung eines Objektes dieser Klasse wird eine beliebige Zeichenkette als `ClientId` (nicht zu verwechseln mit der `ClientId` von OAuth 2.0) und der oben beschriebene API-Schlüssel benötigt. In der zweiten Zeile werden der Benutzername und das Password an den Client übergeben. Für das Auslesen eines Atom-Feeds wird die Methode `getFeed(...)` eingesetzt. Ihr werden in Zeile 5 die URI zum gesuchten Feed und in Zeile 6 die Klasse in die der Feed aus dem XML-Format konvertiert werden soll übergeben. In diesen Fall steckt hinter der URI eine Playliste mit mehreren Videoeinträgen. In Zeile 8 bis 11 werden dann die Titel und der Link auf das Video ausgegeben.

Listing 5.11: Youtube Data API v2: Java `YouTubeService`

```
1 YouTubeService service = new YouTubeService(clientId, apiKey);
2 service.setUserCredentials(username, password);
3
4 VideoFeed videoFeed = service.getFeed(
5     new URL("http://gdata.youtube.com/feeds/api/playlists/
6         PL59AFCB0F92BB89A9"),
7     VideoFeed.class);
8 for(VideoEntry videoEntry : videoFeed.getEntries() ) {
9     System.out.println(videoEntry.getTitle()
10         + ": "
11         + videoEntry.getSelfLink());
12 }
```

Wurde das Ergebnis in mehrere Teilfeeds aufgeteilt, dann ist der Rückgabewert der Funktion `videoFeed.getNextLink()` ungleich `null`. Ist dies der Fall kann der nächste Teilfeed wieder mit der Methode `service.getFeed(...)` und der URI des von `getNextLink()` zurückgegebenen Links geholt werden.

Listing 5.12: Youtube Data API v2: Kommentar schreiben

```
1 commentUrl = "http://gdata.youtube.com/feeds/api/videos/oHg5SJYRHA0/
2   comments";
3 CommentEntry newComment = new CommentEntry();
4 newComment.setContent(new PlainTextConstruct("Tolles Video!! :));
5 service.insert(new URL(commentUrl), newComment);
```

Das Schreiben eines Eintrags ist in Listing 5.12 dargestellt. Die URI zum Schreiben in Zeile Eins zeigt auf einen Kommentar-Feed für das Video mit der ID „oHg5SJYRHA0“. In der zweiten Zeile wird dann zunächst eine Objekt der Klasse `CommentEntry` erstellt und in Zeile 3 mit dem Inhalt

²⁷ <http://code.google.com/apis/youtube/dashboard/>

der Nachricht befüllt. Dieses Objekt wird dann zusammen mit der URI aus Zeile Eins der Methode `insert(...)` übergeben. Für diese Methode ist das vorherige übergeben von Benutzername und Passwort Pflicht, da sonst Youtube mit einen Fehler antwortet.

Mapping von Youtube nach SIOC

Der Aufbau von Youtube unterscheidet sich etwas von den anderen vier Plattformen, da es vorrangig zum Austausch von Videos gedacht ist. Nichtsdestotrotz ist es möglich in den Kommentaren über diese Videos zu diskutieren. Videos können dabei an zwei unterschiedlichen Orten liegen. Entweder sind sie unter den hochgeladenen Videos eines Benutzers zu finden oder in einer von ihm angelegten Playlist. Sowohl die Liste der hochgeladenen Videos (Uploads) als auch die Menge angelegter Playlisten (Playlists) können der SIOC-Klasse `sioc:Forum` zugeordnet werden. Eine Playlist enthält in der Regel Videos mit einem thematischen Schwerpunkt, wie zum Beispiel Musik Genre, Rezepte, Katzenvideos und so weiter. Dies ist den Threads in einen Forum sehr ähnlich, wodurch `sioc:Thread` für eine Playlist die beste Wahl für das Mapping nach SIOC ist. Die in den Uploads und Playlisten enthaltenen Videos entsprechen den Startbeitrag einer Diskussion, die den Inhalt des Videos als Thema hat. Eine Zuordnung von Videos zu `sioc:Post` ist daher naheliegend. Das Video wird dann als `sioc:attachment` an diesen `sioc:Post` angehängt. Videos können seit September 2013 nicht mehr eine Antwort auf ein anderes Video sein (siehe [40]), somit sind Kommentare die einzigen Beiträge, die mit einem Video verbunden sind. Die genauen Beziehungen sind noch einmal in Abbildung 5.11 zu sehen.

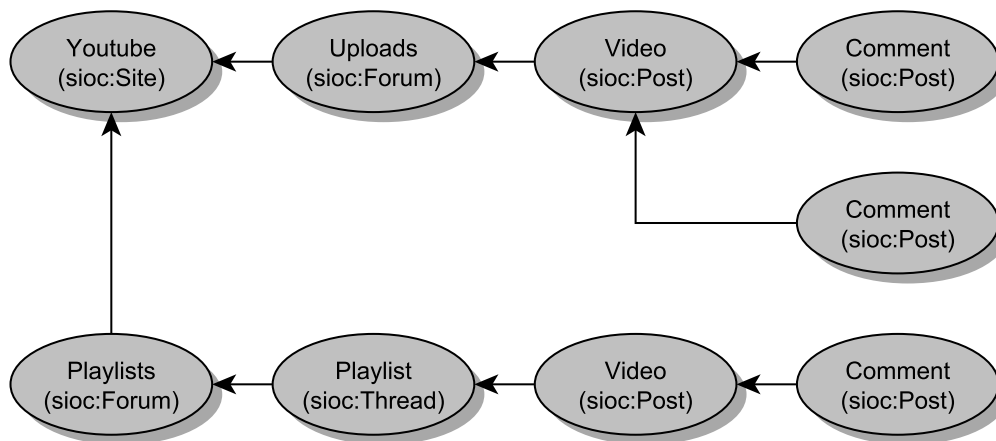


Abbildung 5.11.: Struktur von Youtube und Mapping nach SIOC

Der Zugriff auf die Funktionen der Youtube Data API, die Daten verändern können, wird nur gewährt, wenn der Benutzername und das Passwort eines Benutzers angegeben werden. Die Angabe erfolgt schon wie bei Moodle Über die Klasse `waa:Direct` mit der Eigenschaft `siocsa:accountAuthentication` für den betreffenden `sioc:UserAccount`. Zusätzlich muss der API-Schlüssel, der für das Programm erstellt wurde, der Servicebeschreibung `sioc:Service` hinzugefügt werden. Dies gelingt über die Eigenschaft `siocsa:serviceAuthentication` mit der Klasse `waa:WebAPI` und dem API-Schlüssel als Credential mit der Klasse `waa:APIKey`.

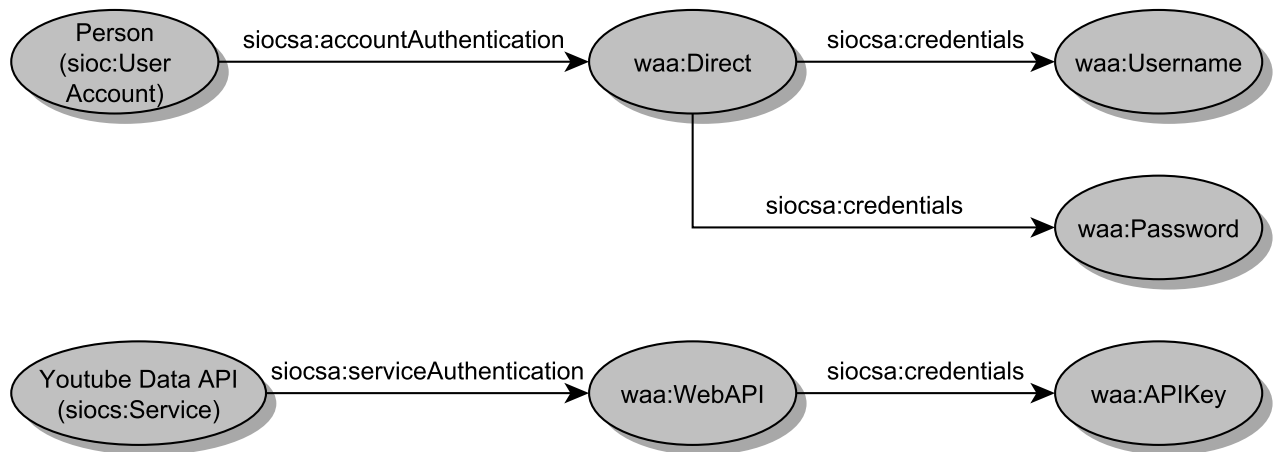


Abbildung 5.12.: Authentifizierungsdaten für Youtube in SIOC

Für die RDF-URIs der Youtube Ressource wurden die gleiche wie für den Zugriff auf die Youtube Data API verwendet (Siehe Tabelle 5.6. Die URIs bauen teilweise wieder aufeinander auf, so dass ein Teil der Struktur in der sie sich befinden rekonstruiert werden kann. Zum Beispiel baut die URI für ein Kommentar auf der für ein Video auf. So erkennt man sofort zu welchen Video der Kommentar gehört, ohne die Daten des Kommentars herunterladen zu müssen.

Tabelle 5.6.: Format der URIs für Youtube in RDF

SIOC Klasse	URI-Format
sioc:Site	http://www.youtube.com/
sioc:Service	http://gdata.youtube.com
sioc:UserAccount	http://gdata.youtube.com/feeds/api/users/{userId}
sioc:Forum	{userAccountUri}/uploads oder {userAccountUri}/playlists
sioc:Thread	http://gdata.youtube.com/feeds/api/playlists/{playlistId}
sioc:Post	http://gdata.youtube.com/feeds/api/videos/{videoId} oder http://gdata.youtube.com/feeds/api/videos/{videoId} /comments/{commentId}

Besonderheiten bei der Implementierungs des YoutubeConnectors

Vorteil der Youtube Data API v2 ist die Verwendung von Atom als Datenformat, so sind schon viele URIs auf weiterführende Ressourcen vorhanden. Leider wird eine Anwendung sehr schnell von Youtube blockiert, wenn diese zu oft hintereinander Abfragen sendet. Aus diesem Grund muss zwischen zwei Abfragen immer eine bestimmte Zeit vergangen sein, bevor die nächste abgeschickt werden kann. Am Besten hat sich eine Zeitspanne von 500 ms gezeigt, welche aber

ab und zu immer noch Probleme macht. Diesen Wert aber höher zu stellen verlangsamt das komplette System, da manchmal mehrere Abfragen gemacht werden müssen, bis alle gesuchten Informationen vorhanden sind. Zusätzlich sind nicht alle Informationen explizit über die Java-API zugänglich. Zum Beispiel erhält die Klasse für Kommentare keine Methode für den Zugriff auf die Video und Kommentar-ID. Diese muss aus der URI des Kommentars erst extrahiert werden.

5.3 Implementierung von SOCC-Camel

Die Implementierung des SOCC-Camel Moduls als Komponente für Camel ist relativ unkompliziert. Die Java-API von Camel enthält schon viele vorbereitete Klassen, von denen die Klassen für SOCC-Camel abgeleitet werden können. Danach ist nur noch wenig Arbeit zusätzlich zu machen.

SoccComponent: Die Klasse SoccComponent muss die in der Konfigurations-URI angegebenen Connectoren mit den Daten aus den Triplestore erzeugen und verwalten und für jeden Endpunkt ein Objekt der Klasse SoccEndpoint mit den Parametern und dem Connector erzeugen.

SoccEndpoint: Die Klasse SoccEndpoint ist wiederum nur eine Fabrik für den SoccPostPollConsumer und den SoccPostProducer.

SoccPostPollConsumer: Die komplette Arbeit des SoccPostPollConsumer findet in der Methode `poll()` statt, die aus der abgeleiteten Klasse `ScheduledPollConsumer` von Camel überschrieben wird. Das Listing 5.13 zeigt den Quellcode dieser Methode. In der zweiten Zeile wird der `PostReader` des Connectors aufgerufen, wobei die Variable `lastPollTime` den Zeitpunkt enthält, an dem `poll()` das letzte mal aufgerufen wurde. Ist mindestens ein Beitrag in der zurückgelieferten Liste, werden diese ab Zeile 4 serialisiert, in eine Nachricht verpackt und an Camel zum Versenden übergeben. Mit `RDF2Go` können leider nur komplette Triplestores serialisiert werden, deswegen müssen die Daten der Beiträge erst in einen neuen, leeren Triplestore kopiert werden, so dass nicht auch unerwünschte Daten mitgeschickt werden die noch im Triplestore des Connectors liegen. Dieser temporäre Triplestore wird in Zeile 5 und 6 angelegt. Der Quellcode in den Zeilen 9 bis 13 kopiert erst alle Prefixe aus dem originalen Triplestore in den neuen. Dieser Schritt ist zwar nicht zwingend notwendig, erleichtert aber das Lesen der serialisierten RDF-Daten für das Debuggen. In den Zeilen 15 bis 20 werden dann alle RDF-Daten der Beiträge in das temporäre Modell kopiert. Danach kann ab Zeile 22 angefangen werden die zu sendende Nachricht zusammen zu bauen. Die serialisierten Daten kommen dann in Zeile 23 als „Body“ in die Nachricht. Zusätzlich wird in Zeile 27 das verwendete Serialisierungsformat als MIME-Type angegeben. Nachrichten werden in Camel zusätzlich in eine Art Umschlag der Klasse `Exchange` verpackt. Das verwenden der Methode `setIn(...)` sagt dabei Camel, dass diese Nachricht als eingehende Nachricht für einen anderen Endpunkt gedacht ist. Möchte man auf eine Nachricht eine Antwort schicken, macht man dies mit der Methode `setOut(...)` tun. In Zeile 33 wird die Nachricht dann abgeschickt. Am Ende muss dann nur noch die Variable `lastPollTime` auf die aktuell Zeit aktualisiert werden.

Listing 5.13: Quellcode zum Polling im SoccPostPollConsumer

```
1  protected int poll() throws Exception {  
2      List<Post> posts = postReader.pollPosts(uri, lastPollTime, limit);
```

```

3      if ( !posts.isEmpty() ) {
4          Model tmpModel = RDF2Go.getModelFactory().createModel();
5          tmpModel.open();
6
7          try {
8              Model model = postReader.getConnector().getContext()
9                  .getModel();
10             for (Entry<String, String> entry : model.getNamespaces().
entrySet()) {
11                 tmpModel.setNamespace(entry.getKey(), entry.getValue());
12             }
13
14             for (Post post : posts) {
15                 tmpModel.addAll(
16                     RdfUtils.getAllStatements(
17                         model,
18                         post ).iterator() );
19             }
20
21             Message msg = new DefaultMessage();
22             msg.setBody(
23                 RdfUtils.modelToString(
24                     tmpModel,
25                     Syntax.RdfXml));
26             msg.setHeader(
27                 Exchange.CONTENT_TYPE,
28                 Syntax.RdfXml.getMimeType());
29
30             Exchange ex = getEndpoint().createExchange();
31             ex.setIn(msg);
32             getProcessor().process(ex);
33         } finally {
34             tmpModel.close();
35         }
36     }
37
38     lastPollTime.setTime( System.currentTimeMillis() );
39 }

```

SoccPostProducer: Die Klasse SoccPostProducer ist wieder nur sehr unspektakulär. Sie nimmt alle ankommenden Nachrichten, holt die serialisierten Daten aus den Body, überprüft ob ein gültiger MIME-Type vorliegt und gibt alles an den PostWriter des Connectors weiter.

5.4 Proof of Concept

Um die Funktionsweise von SOCC und SOCC-Camel zu demonstrieren, soll nun ein „Proof of Concept“ erstellt werden. Hierzu wird ein Programm vorgestellt, das die Beiträge einer Diskussion aus Canvas mit den Kommentaren eines Beitrags aus einer Facebook-Gruppe synchronisiert.

Danach die Ausführung dieses Programm Schritt für Schritt erklärt und gezeigt welche Daten ausgetauscht werden. In der Realität sind die gezeigten SIOC-Daten im RDF/XML-Format, aber aufgrund der besseren Lesbarkeit sind sie hier in Turtle dargestellt.

5.4.1 Vorbereitungen

Die Konfigurationsdaten für dieses Programm sind im Anhang A.3 im Turtle-Format zu sehen. Dort befindet sich die Definitionen von den zwei Personen „Max Hiwi“ und „Florian“ mit jeweils ihren Benutzerkonten für Canvas und Facebook. Für beide Plattformen wurde für die Benutzerkonten Accesstoken erstellt und dort hinterlegt. Beide Personen erlauben einen öffentlichen Zugriff auf all ihre Beiträge durch eine ACL-Autorisierung. Ebenfalls wurden die Servicebeschreibungen und die Konfigurationen für die Canvas und Facebook Connectoren angegeben. Der Service für Facebook enthält zusätzlich die ClientID und den ClientSecret, um aus kurzlebigen Accesstoken welche mit einer Haltbarkeit von zwei Monaten generieren zu können, falls sie das nicht schon sind. Sowohl Accesstoken als auch ClientID und ClientSecret wurden aus Datenschutzgründen nicht vollständig angeben.

Die Synchronisation zwischen Canvas und Facebook mit Camel soll dabei auf zwei Arten demonstriert werden. Der Nachrichtenweg von Canvas aus wird erst an ein JMS-Topic geschickt, bevor die Beiträge an den FacebookConnector weitergeben werden. Der umgekehrte Weg von Facebook nach Canvas erfolgt direkt durch Camel, ohne eine Verwendung von JMS. Als JMS-Provider wird das freie ActiveMQ von Apache eingesetzt. Dieser kann sowohl als eigener Server als auch eingebettet in einem Programm ausgeführt werden.

Zuerst muss für die Verwendung von ActiveMQ und SOCC der CamelContext mit den entsprechenden Komponenten vorbereitet werden. In Zeile Eins des Listings 5.14 wird ein Objekt dieses CamelContextes aus der Klasse DefaultCamelContext erzeugt. Um auf ActiveMQ von Camel aus nutzen zu können, muss ein Objekt der Klasse ActiveMQComponent²⁸ in Zeile 2 vorzugsweise unter dem Namen „activemq“ registriert und die Adresse zum ActiveMQ Server angegeben werden. Danach folgt in Zeile 5 die Registrierung von SOCC-Camel mit dem Namen „socc“. Übergeben wird das Objekt des CamelContextes und ein SoccContext mit einem RDF2Go Model, in das zuvor die Konfigurationsdaten geladen wurden.

Listing 5.14: Proof of Concept: CamelContext vorbereiten

```
1 DefaultCamelContext camelContext = new DefaultCamelContext();
2 camelContext.addComponent(
3     "activemq",
4     ActiveMQComponent.activeMQComponent( "tcp://localhost:61616" ) );
5 camelContext.addComponent(
6     "socc",
7     new SoccComponent( camelContext, new SoccContext( model ) ) );
```

Neben den in Abschnitt 2.3.2 beschriebenen Methode mit der Klasse RouteDefinition, können Routen auch mit der Klasse RouteBuilder erstellt werden. Wie in Listing5.15 zu sehen. Die erste Route in Zeile 4 ist die von der Diskussion aus Canvas in das

²⁸ <http://camel.apache.org/activemq.html>

JMS-Topic mit der Bezeichnung „canvas-topic“. Die URI für den Start-Endpunkt enthält die ID des konfigurierten CanvasConnectors „poc-canvas“, der URI zur Diskussion [https://\[...\]/api/v1/courses/798152/discussion_topics/1540697](https://[...]/api/v1/courses/798152/discussion_topics/1540697) und soll dieses alle fünf Minuten (300000 Millisekunden) nach neuen Beiträgen abfragen. Die zweite Route in Zeile 9 beschreibt den Weg aus dem JMS-Topic „canvas-topic“ in den Connector mit der ID „poc-facebook“, der dann die Beiträge als Kommentare in den Beitrag hinter der URI https://graph.facebook.com/520312298060793_520417398050283 schreibt. Die letzte Route in Zeile 13 definiert den umgekehrten Weg, wobei der FacebookConnector ebenfalls alle fünf Minuten nach neuen Beiträgen schauen soll. Der eben erstellte RouteBuilder muss jetzt noch in Zeile 21 an den CamelContext übergeben und in Zeile 22 Camel gestartet werden.

Listing 5.15: Proof of Concept: Routen vorbereiten

```
1 RouteBuilder routeBuilder = new RouteBuilder() {
2     @Override
3     public void configure() throws Exception {
4         from("socc://poc-canvas?uri=https://canvas.instructure.com/api/v1/"
5             + "courses/798152/discussion_topics/1540697"
6             + "&delay=300000" )
7             .to("activemq:topic:canvas-topic");
8
9         from( "activemq:topic:canvas-topic" )
10            .to("socc://poc-facebook?uri=https://graph.facebook.com/"
11                + "520312298060793_520417398050283");
12
13        from("socc://poc-facebook?uri=https://graph.facebook.com/"
14            + "520312298060793_520417398050283"
15            + "&delay=300000" )
16            .to( "socc://poc-canvas?uri=https://canvas.instructure.com/api/"
17                + "v1/"
18                + "courses/798152/discussion_topics/1540697");
19    };
20
21 camelContext.addRoutes( routeBuilder );
22 camelContext.start();
```

5.4.2 Ausgangssituation

In einen fiktiven Szenario für einen Kurs werden mehrere Gruppenübungen über das Semester verteilt durchgeführt. Für jede Gruppenübung existiert in Canvas eine Diskussion in der über allgemeine Dinge zu jeweiligen Übungen diskutiert werden kann. In Facebook wird dafür die Kommentarfunktion von Beiträgen zur jeweiligen Übung genutzt.

Vor dem Start des Programms befinden sich in der Canvas-Diskussion für die erste Gruppenübung schon zwei Beiträge, wie in der rechten Abbildung 5.13 oben zu sehen ist. Der Benutzer Florian stellte eine Frage und diese wurde von Max Hiwi mit einer Gegenfrage kommentiert. Der Beitrag auf Facebook ist dagegen noch leer (Rechte Abbildung unten).

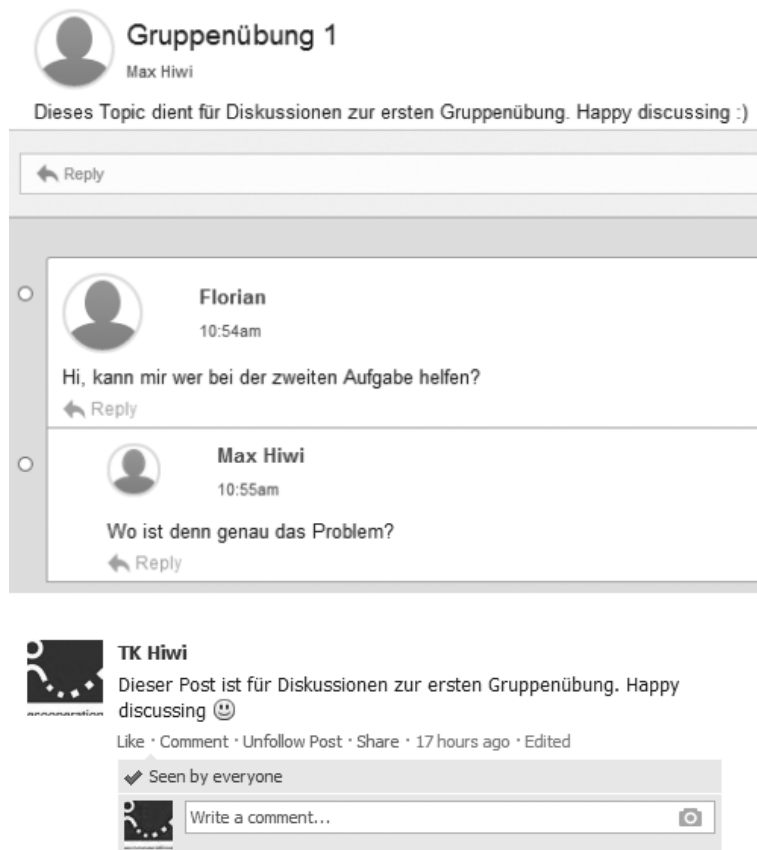


Abbildung 5.13.: Proof of Concept: Ausgangssituation

5.4.3 Ablauf der Synchronisation

Nach dem Camel gestartet wurde, beginnt es die in den Routen definierten Komponenten anzulegen und zu starten. Mit den angegebenen URIs werden die einzelnen Endpunkte und von denen die passenden SoccPostPollingConsumer und SoccPostProducer erzeugt. Dabei werden auch die in den URIs angegebenen Connectoren mit den Daten aus dem Triplestore angelegt.

Zuerst beginnt im SoccPostPollingConsumer der ersten Route der PostReader des Canvas-Connectors die Beiträge mit den festgelegten Defaultuser aus der Diskussion zu lesen. Da dies der erste Aufruf ist, werden die Beiträge noch nicht nach den Erstellungszeitpunkt gefiltert. Beide Beiträge von Florian und Max Hiwi kommen also in die Ergebnisliste, werden in das SIOC Format konvertiert und im Triplestore gespeichert. Das Resultat der Konvertierung wird im Listing 5.16 gezeigt. In Zeile 11 ist zu sehen, dass der erste Beitrag einen Kommentar und zwar den zweiten Beitrag besitzt. Genau so ist im Zeile 22 zu sehen, dass der zweite Beitrag ein Kommentar auf den ersten ist. Diese stimmt mit dem Aussehen der Ausgangssituation überein. Diese beiden Ergebnisse werden dann an den aufrufenden SoccPostPollingConsumer zurückgegeben. Dieser serialisiert das Ergebnis dann in das RDF/XML-Format, verpackt diese in eine Nachricht und gibt diese an Camel weiter.

Listing 5.16: Die gelsenen Beiträge aus Canvas im SIOC-Format


```

1 <https://canvas.instructure.com/api/v1/courses/798152/discussion_topics
  /1540697/entries/3284842> a sioc:Post ;
2   dcterms:created "2013-09-27T10:54:43+02:00" ;
3   dcterms:isPartOf <https://canvas.instructure.com> ;
4   dcterms:modified "2013-09-27T10:54:43+02:00" ;
5   sioc:content "Hi, kann mir wer bei der zweiten Aufgabe helfen?" ;
6   sioc:has_container <https://canvas.instructure.com/api/v1/courses
  /798152/discussion_topics/1540697> ;
7   sioc:has_creator <https://canvas.instructure.com/api/v1/users/3457836/
  profile> ;
8   sioc:has_reply <https://canvas.instructure.com/api/v1/courses/798152/
  discussion_topics/1540697/entries/3284844> ;
9   sioc:id "3284842" ;
10  sioc:last_reply_date "2013-09-27T10:55:48+02:00" ;
11  sioc:num_replies "1"^^xsd:int ;
12  sioc:reply_of <https://canvas.instructure.com/api/v1/courses/798152/
  discussion_topics/1540697#discussion_topic> .
13
14 <https://canvas.instructure.com/api/v1/courses/798152/discussion_topics
  /1540697/entries/3284844> a sioc:Post ;
15   dcterms:created "2013-09-27T10:55:48+02:00" ;
16   dcterms:isPartOf <https://canvas.instructure.com> ;
17   dcterms:modified "2013-09-27T10:55:48+02:00" ;
18   sioc:content "Wo ist denn genau das Problem?" ;
19   sioc:has_container <https://canvas.instructure.com/api/v1/courses
  /798152/discussion_topics/1540697> ;
20   sioc:has_creator <https://canvas.instructure.com/api/v1/users/3478501/
  profile> ;
21   sioc:id "3284844" ;
22   sioc:reply_of <https://canvas.instructure.com/api/v1/courses/798152/
  discussion_topics/1540697/entries/3284842> .

```

Camel schaut nun anhand der definierten Routen, wohin diese Nachricht weitergeleitet werden soll. In diesem Fall wird sie an die ActiveMQ-Komponente übergeben, die sie in das JMS-Topic „canvas-topic“ schickt. Von dort kann sie von jedem JMS-Client gelesen werden, der dieses Topic abonniert hat, wie zum Beispiel in der zweiten Route von vorhin. Dies nimmt die über das Topic gesendete Nachricht entgegen und gibt sie an den SockPostProducer des Endpunktes weiter. Dieser holt die serialisierten RDF-Daten und die Syntaxbeschreibung aus der Nachricht und ruft damit dann die Methode `writePosts(...)` vom `PostWriter` des `FacebookConnectors` auf. Dieser de-serialisiert die beiden Beiträge wieder in RDF und sucht dann im Triplestore nach den passenden Benutzerkonten für Facebook von den beiden Autoren. Dann schreibt er die Beiträge als Kommentare in den festgelegten Facebook-Beitrag.

In der Rechten Abbildung 5.14 sind die eben in Facebook geschriebenen Beiträge aus Canvas zu sehen. Da Facebook kommentieren von anderen Kommentaren in Gruppen nicht ermöglicht, sind diese nun untereinander angelegt. Zugleich ist zusehen, dass Florian gleich danach die Gegenfrage von Max Hiwi, der auf Facebook das Benutzerkonto „TK Hiwi“ benutzt, in Facebook beantwortet hat.



Abbildung 5.14.: Proof of Concept: Beiträge aus Canvas in Facebook

Im Hintergrund arbeitet derweil Camel weiter und startet alle fünf Minuten den PostReader des FacebookConnectors aus der dritten Route. Dieser liest nun diese drei, für ihn neuen Beiträge, und konvertiert diese dann aus dem Facebook- in das SIOC-Format (Siehe Listing 5.17). Die Beiträge werden dann wieder an den SoccPostPollingConsumer zurück gegeben, serialisiert und als Nachricht an Camel geschickt. Diesmal geht diese Nachricht auf direkten Wege über den SoccPostProducer und an den PostWriter des CanvasConnectors. Dieser verarbeitet nun die gesendeten Beiträge und merkt, dass zwei von ihnen ursprüngliche aus Canvas kommen, worauf er diese verwirft. Der dritte Beitrag wird aber mit den Benutzerkonto von Florian in das Diskussionsthema eingefügt.

Wie in Abbildung 5.15 zu sehen ist, enthalten die Diskussion in Canvas und in Facebook die gleichen Beiträge. Dieser Ablauf wiederholt sich nun so lange, wie Camel die Routen und Connectoren ausführt. Es wurde also gezeigt, dass mittels SOCC, SOCC-Camel zwei verteilte Diskussionen synchronisiert werden können.



Abbildung 5.15.: Proof of Concept: Finales Ergebnis in Canvas

Listing 5.17: Die gelesenen Beiträge aus Facebook im SIOC-Format

```
1 <https://graph.facebook.com/520890111336345> a sioc:Post ;
2   dcterms:created "2013-09-27T10:56:09+02:00" ;
3   dcterms:isPartOf <https://www.facebook.com> ;
4   sioc:attachment <https://canvas.instructure.com/> ;
5   sioc:content ""Hi, kann mir wer bei der zweiten Aufgabe helfen?
6 --- forwarded by SOCC from https://canvas.instructure.com ---"";
7   sioc:has_creator <https://graph.facebook.com/100000490230885> ;
8   sioc:id "520890111336345" ;
9   sioc:reply_of <https://graph.facebook.com/520312298060793
10 _520417398050283> ;
11   sioc:sibling <https://canvas.instructure.com/api/v1/courses/798152/
12 discussion_topics/1540697/entries/3284842> .
13
14 <https://graph.facebook.com/520890124669677> a sioc:Post ;
15   dcterms:created "2013-09-27T10:56:13+02:00" ;
16   dcterms:isPartOf <https://www.facebook.com> ;
17   sioc:attachment <https://canvas.instructure.com/> ;
18   sioc:content ""Wo ist denn genau das Problem?
19 --- forwarded by SOCC from https://canvas.instructure.com ---"";
20   sioc:has_creator <https://graph.facebook.com/100003876610187> ;
21   sioc:id "520890124669677" ;
22   sioc:reply_of <https://graph.facebook.com/520312298060793
23 _520417398050283> ;
24   sioc:sibling <https://canvas.instructure.com/api/v1/courses/798152/
25 discussion_topics/1540697/entries/3284844> .
26
27 <https://graph.facebook.com/520890171336339> a sioc:Post ;
28   dcterms:created "2013-09-27T11:00:23+02:00" ;
29   dcterms:isPartOf <https://www.facebook.com> ;
30   sioc:content "hallo welt" ;
31   sioc:has_creator <https://graph.facebook.com/100003876610187> ;
32   sioc:id "520890171336339" ;
33   sioc:reply_of <https://graph.facebook.com/520312298060793
34 _520417398050283> .
```



6 Abschlussbetrachtung und Ausblick

Einleitung schreiben

6.1 Zusammenfassung

Diskussionen ein wichtiger Bestandteil des E-Learnings ist, aber diese oftmals auf verschiedene Plattformen verteilt stattfinden. Dadurch verpassen Personen, die nur auf einer dieser Plattformen präsent sind, vielleicht für sie wichtige Informationen auf einer anderen. Außerdem werden die selben Diskussionsthemen immer wieder an unterschiedlichen Orten an den unterschiedlichsten Orten doppelt und dreifach behandelt. Aus diesem Grund sollte ein System entwickelt werden das den Austausch von Diskussionen zwischen den unterschiedlichen Plattformen ermöglicht.

In Kapitel 3 wurde analysiert, welche Schritte für eine solche Synchronisation nötig sind. Zuerst musste eine Zwischenformat gefunden werden in das die Daten der unterschiedlichen Plattformen konvertiert werden konnten, da dieser Weg effizienter ist, als die einzelnen Formate untereinander zu konvertieren. Also ein solches Zwischenformat bat sich die SIOC Ontologie in Verbindung mit FOAF wunderbar an. Ontologien bietet nicht nur einen guten Ansatz für die Integration von unterschiedlichen Datenformaten, mit RDF ist es sogar möglich dass Programme diese Daten verstehen und aus ihnen neues Wissen ableiten können. Um die Daten letztendlich in das Zwischenformat konvertieren und verarbeiten zu können, braucht es eine einheitliche Schnittstelle mit der dies umgesetzt werden kann. Zusätzlich wurde noch ein System gebraucht, dass die konvertierten Daten zwischen den Schnittstellen für die einzelnen Plattformen austauscht. Zu eignete sich der Austausch über Nachrichten am besten, da dadurch die einzelnen Schnittstellen zeitlich und räumlich entkoppelt werden konnten und nicht voneinander abhängig sind. Als Basis für dieses Nachrichtensystem wurde Camel ausgewählt. Mit dieser Java-Bibliothek können die Routen, welche die Nachrichten nehmen, flexibel konfiguriert werden und sind so leicht erweiterbar. Privatsphäre spielt in der heutigen Zeit ebenfalls eine wichtige Rolle. Deswegen muss es Benutzern es ermöglicht werden das automatischen Lesen und/oder Schreiben für seine Beiträge zuzustimmen oder abzulehnen.

Das Kapitel 4 widmet sich der Beschreibung des Systems, welches die Anforderungen aus Kapitel 3 erfüllen soll. Dieses SOCC genannte System definiert dazu Connectoren, welche die Schnittstellen zu den einzelnen Plattformen abstrahiert und die unterschiedlichen Formate in das SIOC-Format konvertiert. Ein Connector bestehen aus drei Komponenten die je für eine bestimmte Teilaufgabe des Systems verantwortlich sind. Der StructureReader hilft dabei die Struktur der jeweiligen Plattform zu lesen und in SIOC abzubilden, so dass die Beiträge von den richtigen Stellen gelesen und an die richtigen Stellen geschrieben werden können. Der PostReader ist dabei für das Lesen von Beiträgen und deren Konvertierung vom Format der gelesenen Plattform in das SIOC-Format. Für den umgekehrten Weg wird dann der PostWriter eingesetzt. Für den Betrieb der Connectoren brauchen diese einige Informationen von den Benutzern, für den Zugriff auf die

einzelnen APIs. Für diese Informationen konnte auf der vorhandene Service Modul von SIOC für API Beschreibungen, die Basic Access Control Ontologie für die Autorisierung, sowie auf die aus vorhandenen Ontologien zusammengesetzte und erweiterte Services Authentication Ontologie zur Authentifizierung aufgebaut werden. Gespeichert wird alles in einen RDF-Triplestore und erlaubt so einen einfachen Zugriff auf die untereinander verbunden Datensätze. Für die Integration der Connectoren in Camel wurde die Komponente SOCC-Camel entwickelt. Mit den EIP aus Camel können so die Connectoren auf verschiedenste Weisen verbunden werden.

Im letzten Kapitel wurde die Implementierung von Connectoren beschrieben. Dabei wurde drauf eingegangen, auf welche Art und Weise man über die APIs der Plattformen Moodle, Canvas, Facebook, Google+ und Youtube auf deren Daten zugreifen kann und welche Voraussetzungen dafür getroffen werden müssen. Außerdem wurde für jede Plattform gezeigt, wie ihre Struktur den Klassen von SIOC zugeordnet werden kann. Nicht bei allen Plattformen funktionierte alle Einwandfrei. Es wurde beschrieben welche Probleme auftraten und falls möglich eine Lösung präsentiert. Gleiches galt für die Implementierung von SOCC-Camel. Abgeschlossen wurde das Kapitel mit einem Proof of Concept mit einen Schritt-für-Schritt-Beispiel für den Einsatz von SOCC und SOCC-Camel in einem Programm.

6.2 Reflektion

Abschließend betrachtet bin ich sehr zufrieden mit der Implementierung von meinen SOCC-Ansatz. Die Entscheidung für RDF und SIOC als Datenformat für die Integration von verteilten Diskussionen hat sich von Anfang an als eine gute erwiesen. Es stand zwar immer die Frage im Raum, ob sich in SIOC wirklich alle Inhalte der einzelnen Plattformen zusammenführen lassen. Diese war aber vollkommen unbegründet. Bis das endgültige Konzept für SOCC feststand, waren aber trotzdem einige Versuche notwendig. Dies lag nicht nur daran, dass das Thema Datenintegration mit RDF und Ontologien für mich relativ neu war, aber auch dass mit fortschreitender Zeit man immer mehr Erfahrung sammelt und merkt was man am Anfang hätte besser machen können. Gerade solche Erfahrungen immer wieder etwas neues zu entdecken was man vorher noch nicht kannte macht den ganzen Reiz aus. Eine ebenfalls gute Wahl war auch die Verwendung von Camel für den Datenaustausch. Der Implementierungsaufwand war sehr gering und durch die Nutzung der beliebig konfigurierbaren Routen ist ein sehr flexibles System entstanden.

Aber dass alles perfekt ist, kann man auch nicht sagen. Die in Abschnitt 5.2.2 beschrieben Lösungen zu doppelten Beiträgen und Rekonstruktion der Kommentarstruktur funktionieren zwar, sind aber alles andere als perfekt. Statt dem Einfügen eines Wasserzeichens könnte man nach einen Beitrag im Triplestore suchen der dem vermeintlichen Duplikat gleicht. Die Rekonstruktion der Kommentarstruktur könnte auch besser funktionieren. Es ist schade, dass dies nur in eine Richtung reibungslos vonstatten geht. Um die Connectoren weiterhin unabhängig zu halten, müsste es in einer Verbesserung möglich sein, auch Daten über die Struktur der Plattformen auszutauschen.

Zum Schluss noch ein paar Worte zu RDF2Go. Um es gleich zu sagen, RDF2Go ist ein guter Weg innerhalb von Java mit RDF-Daten zu arbeiten. Dadurch dass man nicht an einen bestimmten Triplestore und dessen API gebunden ist (dafür aber an die von RDF2Go) kann man für seine Zwecke den richtigen wählen. Auch der Generator zum Generieren von Java-Klassen aus

Ontologien arbeitet besser als der von anderen Frameworks und das Arbeiten mit Klassen ist wesentlich einfacher als alle Triple einzeln zu manipulieren. Doch hat auch der Generator und die generierten Klassen so ihre Macken. Zum Beispiel werden für die einzelnen Eigenschaften keine Methoden generiert, um sich nur einen Wert zurückgeben zu lassen. Man bekommt immer einen Iterator den man durchlaufen musste. Diese Methoden wurden aber dann von Hand hinzugefügt. Ein Problem mit der Generierung von Java-Klassen war das fehlen der Mehrfachvererbung. Java kennt so etwas nicht, aber RDFSchemata und OWL sehr wohl. So mussten einige Methoden aus der Java-Klasse `OnlineAccount` von FOAF in `UserAccount` von SIOC kopiert werden, um die Eigenschaften aus FOAF auf dort nutzen zu können.

6.3 Ausblick

Text schreiben



A Anhang

A.1 SOCC Connector Config Ontologie

```
1  <?xml version="1.0"?>
2  <!--
3      Author: Florian Mueller
4      Date: 2013-09-28
5      Version: 1.3
6  -->
7
8  <!DOCTYPE rdf:RDF [
9      <!ENTITY sioc "http://rdfs.org/sioc/ns#" >
10     <!ENTITY foaf "http://xmlns.com/foaf/0.1/" >
11     <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
12     <!ENTITY siocs "http://rdfs.org/sioc/services#" >
13     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
14     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
15     <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
16 ]>
17
18 <rdf:RDF xmlns="http://www.m0ep.de/socc/config#"
19     xml:base="http://www.m0ep.de/socc/config#"
20     xmlns:sioc="http://rdfs.org/sioc/ns#"
21     xmlns:dcterms="http://purl.org/dc/terms/"
22     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
23     xmlns:siocs="http://rdfs.org/sioc/services#"
24     xmlns:foaf="http://xmlns.com/foaf/0.1/"
25     xmlns:owl="http://www.w3.org/2002/07/owl#"
26     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
27     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
28
29     <owl:Ontology
30         rdf:about="http://www.m0ep.de/socc/config#"
31         rdf:type="http://www.w3.org/2002/07/owl#Thing">
32         <dcterms:title>SOCC Connector Configuration Ontology</
dcterms:title>
33         <dcterms:description>
34             Ontology for connector configurations of the SOCC (Social
Online Community Connectors) Framework.
35         </dcterms:description>
36
37         <owl:imports rdf:resource="http://rdfs.org/sioc/services#"/>
38         <owl:imports rdf:resource="http://rdfs.org/sioc/ns#"/>
```

```

39     </owl:Ontology>
40
41
42     <!--
43     //////////////////////////////////////
44     // Object Properties
45     //////////////////////////////////////
46     -->
47
48     <!-- http://www.m0ep.de/socc/config#defaultUser -->
49     <owl:ObjectProperty rdf:about="http://www.m0ep.de/socc/config#
50 defaultUserAccount">
51         <rdfs:range rdf:resource="&sioc;UserAccount"/>
52         <rdfs:domain rdf:resource="http://www.m0ep.de/socc/config#
53 ConnectorConfig"/>
54     </owl:ObjectProperty>
55
56     <!-- http://www.m0ep.de/socc/config#service -->
57     <owl:ObjectProperty rdf:about="http://www.m0ep.de/socc/config#service"
58 >
59         <rdfs:domain rdf:resource="http://www.m0ep.de/socc/config#
60 ConnectorConfig"/>
61         <rdfs:range rdf:resource="&siocs;Service"/>
62     </owl:ObjectProperty>
63
64     <!-- http://www.m0ep.de/socc/config#mappedTo -->
65     <owl:SymmetricProperty rdf:about="http://www.m0ep.de/socc/config#
66 mappedTo">
67         <rdfs:domain rdf:resource="&sioc;UserAccount"/>
68         <rdfs:range rdf:resource="&sioc;UserAccount"/>
69     </owl:SymmetricProperty>
70
71     <!--
72     //////////////////////////////////////
73     // Data properties
74     //////////////////////////////////////
75     -->
76
77     <!-- http://www.m0ep.de/socc/config#connectorClass Name
78         Java class of the connector
79     -->
80     <owl:DatatypeProperty rdf:about="http://www.m0ep.de/socc/config#
81 connectorClassName">
82         <rdfs:domain rdf:resource="http://www.m0ep.de/socc/config#
83 ConnectorConfig"/>
84         <rdfs:range rdf:resource="&xsd:string"/>
85     </owl:DatatypeProperty>
86
87     <!-- http://www.m0ep.de/socc/config#id -->

```

```

82     <owl:DatatypeProperty rdf:about="http://www.m0ep.de/socc/config#id">
83         <rdfs:domain rdf:resource="http://www.m0ep.de/socc/config#
ConnectorConfig"/>
84         <rdfs:range rdf:resource="&xsd:string"/>
85     </owl:DatatypeProperty>
86
87     <!-- http://www.m0ep.de/socc/config#unknownMessageTemplate -->
88     <owl:DatatypeProperty rdf:about="http://www.m0ep.de/socc/config#
unknownMessageTemplate">
89         <rdfs:domain rdf:resource="http://www.m0ep.de/socc/config#
ConnectorConfig"/>
90         <rdfs:range rdf:resource="&xsd:string"/>
91     </owl:DatatypeProperty>
92
93     <!--
94     //////////////////////////////////////
95     // Classes
96     //////////////////////////////////////
97     -->
98
99     <!-- http://www.m0ep.de/socc/config#ConnectorConfig -->
100    <owl:Class rdf:about="http://www.m0ep.de/socc/config#ConnectorConfig"/
>
101 </rdf:RDF>
102
103 <!-- Generated by the OWL API (version 3.4.2) http://owlapi.sourceforge.
net -->

```

A.2 SIOC Services Authentication Module

```

1  <?xml version="1.0"?>
2  <!--
3      Author:      Florian Mueller
4      Date:        2013-08-07
5      Version:     2.0
6  -->
7
8  <!DOCTYPE RDF [
9      <!ENTITY sioc "http://rdfs.org/sioc/ns#" >
10     <!ENTITY dcterms "http://purl.org/dc/terms/" >
11     <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
12     <!ENTITY sioc_services "http://rdfs.org/sioc/services#" >
13     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
14     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
15     <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
16     <!ENTITY waa "http://purl.oclc.org/NET/WebApiAuthentication#" >
17 ]>
18 <rdf:RDF

```

```

19   xml:base="http://www.m0ep.de/sioc/services/auth#"
20   xmlns="http://www.m0ep.de/sioc/services/auth#"
21   xmlns:dcterms="http://purl.org/dc/terms/"
22   xmlns:owl="http://www.w3.org/2002/07/owl#"
23   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
24   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
25   xmlns:sioc="http://rdfs.org/sioc/ns#"
26   xmlns:siocs="http://rdfs.org/sioc/services#"
27   xmlns:waa="http://purl.oclc.org/NET/WebApiAuthentication#"
28   xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
29
30   <owl:Ontology
31     rdf:about="http://www.m0ep.de/sioc/services/auth#"
32     rdf:type="http://www.w3.org/2002/07/owl#Thing">
33     <dcterms:title>SIOC Service Authentication Module</dcterms:title>
34     <dcterms:description>
35       Extends the SIOC Core and Service Module to add information
36       about authentication mechanisms and their required credentials. It
37       reuses some parts from the WebApiAuthentication Ontology.
38     </dcterms:description>
39     <rdfs:seeAlso rdf:resource="http://purl.oclc.org/NET/
40       WebApiAuthentication#" />
41     <rdfs:seeAlso rdf:resource="http://rdfs.org/sioc/services#" />
42     <owl:imports rdf:resource="http://rdfs.org/sioc/services#" />
43   </owl:Ontology>
44
45   <!--
46   //////////////////////////////////////
47   // Object Properties
48   //////////////////////////////////////
49   -->
50
51   <!-- http://purl.oclc.org/NET/WebApiAuthentication#hasInputCredentials
52   -->
53   <owl:ObjectProperty
54     rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
55       hasInputCredentials">
56     <owl:equivalentProperty
57       rdf:resource="http://www.m0ep.de/sioc/services/auth#
58       credentials" />
59     </owl:ObjectProperty>
60
61     <!-- http://www.m0ep.de/sioc/services/auth#serviceAuthentication -->
62     <owl:ObjectProperty rdf:about="http://www.m0ep.de/sioc/services/auth#
63       serviceAuthentication">
64       <rdfs:range
65         rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
66         AuthenticationMechanism" />

```

```

60     <rdfs:domain rdf:resource="http://rdfs.org/sioc/services#Service"/
61 >
62     </owl:ObjectProperty>
63     <!-- http://www.m0ep.de/sioc/services/auth#accountAuthentication -->
64     <owl:ObjectProperty rdf:about="http://www.m0ep.de/sioc/services/auth#
65 accountAuthentication">
66         <rdfs:range
67             rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
68 AuthenticationMechanism"/>
69         <rdfs:domain rdf:resource="http://rdfs.org/sioc/ns#UserAccount"/>
70     </owl:ObjectProperty>
71     <!-- http://www.m0ep.de/sioc/services/auth#credentials -->
72     <owl:ObjectProperty rdf:about="http://www.m0ep.de/sioc/services/auth#
73 credentials">
74         <rdfs:domain
75             rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
76 AuthenticationMechanism"/>
77         <rdfs:range rdf:resource="http://purl.oclc.org/NET/
78 WebApiAuthentication#Credentials"/>
79     </owl:ObjectProperty>
80
81     <!--
82     //////////////////////////////////////
83     // Classes
84     //////////////////////////////////////
85     -->
86
87     <!-- http://purl.oclc.org/NET/WebApiAuthentication#APIKey -->
88     <owl:Class rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
89 APIKey">
90         <rdfs:subClassOf rdf:resource="http://purl.oclc.org/NET/
91 WebApiAuthentication#Credentials"/>
92     </owl:Class>
93
94     <!-- http://purl.oclc.org/NET/WebApiAuthentication#
95 AuthenticationMechanism -->
96     <owl:Class rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
97 AuthenticationMechanism">
98
99     <!-- http://purl.oclc.org/NET/WebApiAuthentication#Credentials -->
100    <owl:Class rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
101 Credentials">
102
103    <!-- http://purl.oclc.org/NET/WebApiAuthentication#Direct -->
104    <owl:Class rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
105 Direct">
106        <rdfs:subClassOf

```

```

97         rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
AuthenticationMechanism"/>
98     </owl:Class>
99
100     <!-- http://purl.oclc.org/NET/WebApiAuthentication#OAuth -->
101     <owl:Class rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
OAuth">
102         <rdfs:subClassOf
103             rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
AuthenticationMechanism"/>
104     </owl:Class>
105
106     <!-- http://purl.oclc.org/NET/WebApiAuthentication#Password -->
107     <owl:Class rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
Password">
108         <rdfs:subClassOf rdf:resource="http://purl.oclc.org/NET/
WebApiAuthentication#Credentials"/>
109     </owl:Class>
110
111     <!-- http://purl.oclc.org/NET/WebApiAuthentication#Username -->
112     <owl:Class rdf:about="http://purl.oclc.org/NET/WebApiAuthentication#
Username">
113         <rdfs:subClassOf rdf:resource="http://purl.oclc.org/NET/
WebApiAuthentication#Credentials"/>
114     </owl:Class>
115
116     <!-- http://www.m0ep.de/sioc/services/auth#AccessToken -->
117     <owl:Class rdf:about="http://www.m0ep.de/sioc/services/auth#
AccessToken">
118         <rdfs:subClassOf rdf:resource="http://purl.oclc.org/NET/
WebApiAuthentication#Credentials"/>
119         <owl:equivalentClass
120             rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
OAuthToken"/>
121     </owl:Class>
122
123     <!-- http://www.m0ep.de/sioc/services/auth#ClientId -->
124     <owl:Class rdf:about="http://www.m0ep.de/sioc/services/auth#ClientId">
125         <rdfs:subClassOf rdf:resource="http://purl.oclc.org/NET/
WebApiAuthentication#Credentials"/>
126         <owl:equivalentClass
127             rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
OAuthConsumerKey"/>
128     </owl:Class>
129
130     <!-- http://www.m0ep.de/sioc/services/auth#ClientSecret -->
131     <owl:Class rdf:about="http://www.m0ep.de/sioc/services/auth#
ClientSecret">
132         <rdfs:subClassOf rdf:resource="http://purl.oclc.org/NET/
WebApiAuthentication#Credentials"/>

```

```

133         <owl:equivalentClass
134             rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
OauthConsumerSecret"/>
135     </owl:Class>
136
137     <!-- http://www.m0ep.de/sioc/services/auth#RefreshToken -->
138     <owl:Class rdf:about="http://www.m0ep.de/sioc/services/auth#
RefreshToken">
139         <rdfs:subClassOf rdf:resource="http://purl.oclc.org/NET/
WebApiAuthentication#Credentials"/>
140     </owl:Class>
141
142     <!-- http://www.m0ep.de/sioc/services/auth#WebAPI -->
143     <owl:Class rdf:about="http://www.m0ep.de/sioc/services/auth#WebAPI">
144         <rdfs:subClassOf
145             rdf:resource="http://purl.oclc.org/NET/WebApiAuthentication#
AuthenticationMechanism"/>
146     </owl:Class>
147 </rdf:RDF>
148 <!-- Generated by the OWL API (version 3.4.2) http://owlapi.sourceforge.
net -->

```

A.3 Proof of Concept: Konfigurationsdaten

```

1  @prefix ccfg: <http://www.m0ep.de/socc/config#> .
2  @prefix sioc: <http://rdfs.org/sioc/ns#> .
3  @prefix waa: <http://purl.oclc.org/NET/WebApiAuthentication#> .
4  @prefix siocs: <http://rdfs.org/sioc/services#> .
5  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
6  @prefix acl: <http://www.w3.org/ns/auth/acl#> .
7  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
8  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
9  @prefix siocsa: <http://www.m0ep.de/sioc/services/auth#> .
10 @prefix dcterms: <http://purl.org/dc/terms/> .
11
12 #####
13 # Definitions of the default User
14 #####
15
16 <http://www.example.org#max> a foaf:Person ;
17     foaf:account <https://canvas.instructure.com/api/v1/users/3478501/
profile> ,
18     <https://graph.facebook.com/100003876610187> ;
19     foaf:name "Max Hiwi" .
20
21 [] a acl:Authorization ;
22     acl:accessToClass sioc:Post ;
23     acl:agentClass foaf:Agent ;

```

```

24     acl:mode acl:Read , acl:Write ;
25     acl:owner <http://www.example.org#max> .
26
27 <https://canvas.instructure.com/api/v1/users/3478501/profile> a sioc:
    UserAccount ;
28     ccfg:mappedTo <https://graph.facebook.com/100003876610187> ;
29     foaf:accountName "3478501" ;
30     foaf:accountServiceHomepage <https://canvas.instructure.com/api/v1> ;
31     sioc:account_of <http://www.example.org#max> ;
32     siocs:has_service <https://canvas.instructure.com/api/v1> ;
33     siocsa:accountAuthentication [
34         a waa:OAuth ;
35         siocsa:credentials [
36             a siocsa:AccessToken ;
37             rdf:value "7~wCpRKiFl91vr..."
38         ]
39     ] .
40
41 <https://graph.facebook.com/100003876610187> a sioc:UserAccount ;
42     ccfg:mappedTo <https://canvas.instructure.com/api/v1/users/3478501/
    profile> ;
43     foaf:accountName "100003876610187" ;
44     foaf:accountServiceHomepage <https://graph.facebook.com> ;
45     sioc:account_of <http://www.example.org#max> ;
46     siocs:has_service <https://graph.facebook.com> ;
47     siocsa:accountAuthentication [
48         a waa:OAuth ;
49         siocsa:credentials [
50             a siocsa:AccessToken ;
51             rdf:value "CAAF1MmpN3J4BA..."
52         ]
53     ] .
54
55 #####
56 # Definition of another User
57 #####
58
59 <http://www.example.org#florian> a foaf:Person ;
60     foaf:account <https://canvas.instructure.com/api/v1/users/3457836/
    profile> ,
61     <https://graph.facebook.com/100000490230885> ;
62     foaf:name "Florian" .
63
64 [] a acl:Authorization ;
65     acl:accessToClass sioc:Post ;
66     acl:agentClass foaf:Agent ;
67     acl:mode acl:Read , acl:Write ;
68     acl:owner <http://www.example.org#florian> .
69

```



```

70 <https://canvas.instructure.com/api/v1/users/3457836/profile> a sioc:
    UserAccount ;
71     ccfg:mappedTo <https://graph.facebook.com/100000490230885> ;
72     foaf:accountName "3457836" ;
73     foaf:accountServiceHomepage <https://canvas.instructure.com/api/v1> ;
74     sioc:account_of <http://www.example.org#florian> ;
75     siocs:has_service <https://canvas.instructure.com/api/v1> ;
76     siocsa:accountAuthentication [
77         a waa:OAuth ;
78         siocsa:credentials [
79             a siocsa:AccessToken ;
80             rdf:value "7~45nTvodrOtuaDNb..."
81         ]
82     ] .
83
84 <https://graph.facebook.com/100000490230885> a sioc:UserAccount ;
85     ccfg:mappedTo <https://canvas.instructure.com/api/v1/users/3457836/
    profile> ;
86     foaf:accountName "100000490230885" ;
87     foaf:accountServiceHomepage <https://graph.facebook.com> ;
88     sioc:account_of <http://www.example.org#florian> ;
89     siocs:has_service <https://graph.facebook.com> ;
90     siocsa:accountAuthentication [
91         a waa:OAuth ;
92         siocsa:credentials [
93             a siocsa:AccessToken ;
94             rdf:value "CAAF1MmpN3J4BAI0N..."
95         ]
96     ] .
97
98 #####
99 # Definition of the Canvas Service and Connector
100 #####
101
102 <https://canvas.instructure.com/api/v1> a siocs:Service ;
103     siocs:service_endpoint <https://canvas.instructure.com> ;
104     siocs:service_of <https://canvas.instructure.com/api/v1/users/3478501/
    profile> ,
105     <https://canvas.instructure.com/api/v1/users/3478501/profile> .
106
107 [] a ccfg:ConnectorConfig
108     ccfg:connectorClassName "de.m0ep.socc.core.connector.canvaslms.
    CanvasLmsConnector" ;
109     ccfg:defaultUserAccount <https://canvas.instructure.com/api/v1/users
    /3478501/profile> ;
110     ccfg:id "poc-canvas" ;
111     ccfg:service <https://canvas.instructure.com/api/v1> ;
112     ccfg:unknownMessageTemplate "{authorName} wrote:<br>{message}" .
113
114 #####

```

```

115 # Definition of the Facebook Service and Connector
116 #####
117
118 <https://graph.facebook.com> a siocs:Service ;
119     siocs:service_endpoint <https://graph.facebook.com> ;
120     siocs:service_of <https://graph.facebook.com/100003876610187> ,
121         <https://graph.facebook.com/100000490230885> ;
122     siocsa:serviceAuthentication [
123         a waa:OAuth ;
124         siocsa:credentials [
125             a siocsa:ClientId ;
126             rdf:value "4103343..."
127         ] , [
128             a siocsa:ClientSecret ;
129             rdf:value "5988954e39fc9ca4d..."
130         ]
131     ] .
132
133 [] a ccfg:ConnectorConfig ;
134     ccfg:connectorClassName "de.m0ep.socc.core.connector.facebook.
FacebookConnector" ;
135     ccfg:defaultUserAccount <https://graph.facebook.com/100003876610187> ;
136     ccfg:id "poc-facebook" ;
137     ccfg:service <https://graph.facebook.com> ;
138     ccfg:unknownMessageTemplate "{authorName} wrote: {message}" .

```

A.4 OAuthTool

Da während der Entwicklungsphase zum Testen immer mal wieder neue OAuth Accesstoken gebraucht werden, war es nach einer weile zu Umständlich diese sich immer per Hand zu holen. Deswegen wurde ein kleines Hilfsprogramm implementiert, dass ohne großen Aufwand Accesstoken für Facebook und Google+ holen kann. Abbildung A.1 zeigt die Oberfläche des OAuthTools mit den Dialog für Google+.

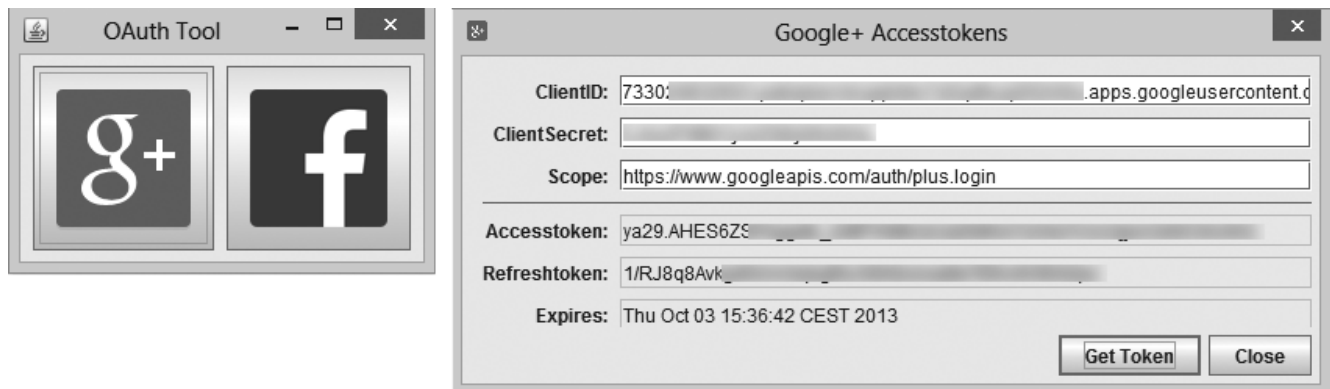


Abbildung A.1.: OAuthTool Fenster

Literaturverzeichnis

- [1] WebAccessControl. <http://www.w3.org/wiki/WebAccessControl>, Zugriff: 2013-09-12.
- [2] Apache Software Foundation. Apache Camel. <http://camel.apache.org>, Zugriff: 2013-09-15.
- [3] Tim Berners-Lee. Linked Data. <http://www.w3.org/DesignIssues/LinkedData.html>, Zugriff: 2013-09-17, 2009.
- [4] Tim Berners-Lee. Socially Aware Cloud Storage. <http://www.w3.org/DesignIssues/CloudStorage.html>, Zugriff: 2013-08-26, 2011.
- [5] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.
- [6] Uldis Bojars, John G. Breslin, and Stefan Decker. Porting social media contributions with SIOC. *Recent Trends and Developments in Social Software*, 6045:116–122, 2011.
- [7] Uldis Bojars, Alexandre Passant, John G. Breslin, and Stefan Decker. Social Network and Data Portability using Semantic Web Technologies. (1):5–19, 2008.
- [8] John G. Breslin, Andreas Harth, Uldis Bojars, and Stefan Decker. Towards semantically-interlinked online communities. *The Semantic Web: Research and Applications*, pages 71–83, 2005.
- [9] John G. Breslin, Alexandre Passant, and Stefan Decker. *The Social Semantic Web*. Springer, Berlin, Heidelberg, 2009.
- [10] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. <http://www.w3.org/TR/rdf-schema/>, Zugriff: 2013-09-14, 2004.
- [11] Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.98. <http://xmlns.com/foaf/spec/>, Zugriff: 2013-09-13, 2010.
- [12] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL). <http://www.w3.org/TR/wsd1>, Zugriff: 2013-09-09, 2001.
- [13] Jo Davies and Martin Graff. Performance in e-learning: online participation and student grades. *British Journal of Educational Technology*, 36(4):657–663, 2005.
- [14] Digital Enterprise Research Institute. SIOC - Semantically-Interlinked Online Communities. <http://sioc-project.org/>, Zugriff: 2013-08-15.
- [15] Stephen Downes. E-learning 2.0. *eLearn Magazine*, 2005.
- [16] Facebook. Graph API. <https://developers.facebook.com/docs/reference/api>: Zugriff: 2013-09-20.

-
- [17] Facebook. Key Facts. <http://newsroom.fb.com/Key-Facts>, Zugriff: 2013-09-11, 2013.
- [18] Facebook in Education and Anthony Fontana. Using a Facebook Group As a Learning Management System. <https://www.facebook.com/notes/facebook-in-education/using-a-facebook-group-as-a-learning-management-system/10150244221815570>, Zugriff: 2013-09-14, 2010.
- [19] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.
- [20] Anthony Fontana. The Multichronic Classroom : Creating an Engaging Environment for All Students. *FOUNDATIONS IN ART: THEORY AND EDUCATION, FATE IN REVIEW*, 30:13—18, 2009.
- [21] Google. Google+ API. <https://developers.google.com/+/api/>, Zugriff: 2013-09-22.
- [22] D. Hardt. The OAuth 2.0 Authorization Framework. <http://tools.ietf.org/html/rfc6749>, Zugriff: 2013-08-30, 2012.
- [23] Pascal Hitzler, Arkus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web: Grundlagen*. Springer, 2008 edition, 2008.
- [24] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [25] James Hollenbach, Joe Presbrey, and Tim Berners-Lee. Using RDF Metadata To Enable Access Control on the Social Semantic Web. *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, 514, 2009.
- [26] Iwen Huang. The effects of personality factors on participation in online learning. In *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication, ICUIMC '09*, pages 150–156, New York, NY, USA, 2009. ACM.
- [27] Graham Klyne and Jerme J. Carrol. Resource Description Framework (RDF): Concepts and Abstract Syntax. <http://www.w3.org/TR/rdf-concepts/>, Zugriff: 2013-08-19, 2004.
- [28] Frank Manola and Eric Miller. RDF Primer. <http://www.w3.org/TR/rdf-primer/>, Zugriff: 2013-08-19, 2004.
- [29] Frank McCown and Michael L. Nelson. What happens when facebook is gone. *Proceedings of the 2009 joint international conference on Digital libraries JCDL 09 (2009)*, pages 251–254, 2009.
- [30] Nilo Mitra and Yves Lafon. SOAP Version 1.2. <http://www.w3.org/TR/2007/REC-soap12-part0-20070427/>, Zugriff: 2013-09-09, 2007.
- [31] Stuart Palmer, Dale Holt, and Sharyn Bray. Does the discussion help? The impact of a formally assessed online discussion on final student results. *British Journal of Educational Technology*, 39(5):847–858, 2008.
- [32] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. <http://www.w3.org/TR/owl-semantics/>, Zugriff: 2013-09-14, 2004.

-
- [33] Andrea Schröder. Web der Zukunft. *XML & Web Services Magazin*, pages 40—43, 2003.
- [34] Felix Schwenzel and Sascha Lobo. Reclaim Social. <http://reclaim.fm/>, Zugriff: 2013-08-14.
- [35] Sun/Oracle. Java Message Service. <http://www.oracle.com/technetwork/java/jms/index.html>, Zugriff: 2013-09-15.
- [36] Watkins Thomas. Suddenly, Google Plus Is Outpacing Twitter To Become The World's Second Largest Social Network. <http://www.businessinsider.com/google-plus-is-outpacing-twitter-2013-5>, Zugriff: 2013-09-11, 2013.
- [37] Mike Uschold and Michael Gruninger. Ontologies: Principles, methods and applications. *Knowledge engineering review*, (February), 1996.
- [38] Qiyun Wang, Huay Lit Woo, Choon Lang Quek, Yuqin Yang, and Mei Liu. Using the Facebook group as a learning management system: An exploratory study. *British Journal of Educational Technology*, 43(3):428–438, May 2012.
- [39] Youtube. Statistik. <http://www.youtube.com/yt/press/de/statistics.html>, Zugriff: 2013-09-10.
- [40] YT Creators. So long, video responses... Next up: better ways to connect. <http://youtubecreator.blogspot.de/2013/08/so-long-video-responsesnext-up-better.html>: Zugriff: 2013-09-29, 2013.