

Erstellung eines Frameworks zur Realisierung von RESTful Webservices mit OAuth Authentifizierung

Bachelorarbeit im Studiengang Medieninformatik

Dominik Horb <dominik.horb@gmail.com>
Fachbereich Informatik und Medien, Matrikel: 741153
Beuth Hochschule für Technik Berlin
Betreuende Lehrkraft: Prof. Dr. Heike Ripphausen-Lipa

Inhaltsverzeichnis

1. Einleitung	1
1.1. Motivation	1
1.2. Aufgabenstellung	3
1.3. Vorausschau	3
1.4. Konventionen	4
2. Fachliches Umfeld	5
2.1. REST – Representational State Transfer und HTTP	6
2.1.1. HTTP-Requests und HTTP-Responses in REST Systemen	7
2.1.2. Weiterführende Literatur	12
2.2. Das OAuth Protokoll	12
2.2.1. Die OAuth Signatur	14
2.2.2. Redirection-Based Authorization – Der „OAuth Dance“	16
2.2.3. Weiterführende Literatur	19
2.3. REST und OAuth kombiniert – Ein Beispiel	19
3. Pflichtenheft	23
3.1. Zielbestimmung	23
3.1.1. Musskriterien	23
3.1.2. Wishkriterien	23
3.1.3. Abgrenzungskriterien	24
3.2. Produkteinsatz	24
3.2.1. Anwendungsbereiche	24
3.2.2. Zielgruppen	24
3.2.3. Betriebsbedingungen	25
3.3. Produktfunktionen	25
3.4. Produktdaten	26
3.5. Produktleistungen	27
3.6. Qualitätsanforderungen	28
3.7. Benutzungsoberfläche	28
3.8. Technische Produktumgebung	30
3.8.1. Software	30
3.8.2. Hardware	30
3.9. Entwicklungsumgebung	30
3.9.1. Software	30
3.9.2. Hardware	31

3.9.3. Orgware	31
4. Systementwurf und Implementierung	32
4.1. Die erste Planung	33
4.2. Bootstrapping mit mod_rewrite	35
4.3. Die entstandene Architektur	37
5. Systemtest	40
5.1. Testfälle	42
6. Schlussbetrachtung	47
A. User's Manual	49
A.1. Installation	49
A.2. Usage	50
A.2.1. Implementing Resources	50
A.2.2. Protecting Resources	52
A.2.3. Testing Resources	54
Listings	56
Abbildungsverzeichnis	57
Literaturverzeichnis	58

1. Einleitung

1.1. Motivation

„The largest known implementation of a system conforming to the REST architectural style is the World Wide Web.“ [Wikipedia, 2010]

Mein erster ernsthafter Kontakt mit dem Thema Webservices war in einer Vorlesung zu „Verteilten Systemen“ hauptsächlich in Form von RPC¹-Architekturen. Was mich allerdings sofort dazu gebracht hat, dem Ganzen, solange eine Auseinandersetzung damit nicht unbedingt nötig war, erstmal aus dem Weg zu gehen. All dieser ganze Aufwand nur, um einem entfernten Rechner mitzuteilen, dass er mir einen bestimmten Datensatz schicken soll oder eine bestimmte Methode ausführen soll?

Als ich mich dann beruflich in verschiedene APIs großer Internetportale eingearbeitet hatte, stieß ich plötzlich immer wieder auf sehr ähnlich geartete Systeme, die alle als RESTful Web Services gekennzeichnet waren. Das stimmt mal mehr und mal weniger, wenn man sich die Dissertation von Roy Fielding genauer anschaut, mit der er die Grundlagen dieser Architekturart gelegt hat [Fielding, 2000]. Gemeinsam bleibt ihnen jedenfalls, dass die Benutzung mit relativ einfachen Mitteln funktioniert.

Um z. B. bei Facebook, last.fm oder in den VZ Netzwerken mit einer API auf dem entfernten Rechner zu kommunizieren, reicht es oft aus, eine einfache URI² aufzurufen, um einen bestimmten Datensatz zu erhalten. Das Ganze funktioniert also im einfachsten Fall genauso intuitiv wie das World Wide Web, das wir tagtäglich nutzen und erfordert keine komplexen zusätzlichen Kommunikationsprotokolle.

Die Einsicht meiner öffentlich verfügbaren Daten bei Facebook erfordert nur den Aufruf der folgenden Adresse im Browser:

- <https://graph.facebook.com/100000791373270>

Das wird noch niemanden ernsthaft beeindrucken, stellt aber schon das Grundprinzip eines REST Systems vor, wie es in dieser Arbeit erstellt und beschrieben werden soll. Zur Kommunikation werden die im HTTP-Protokoll definierten Methoden benutzt ohne das zusätzliche Verpacken der Aufrufinformationen in ein XML Dokument, wie es z. B.

¹Remote Procedure Call

²Uniform Resource Identifier

bei der gängigen RPC-Variante SOAP passiert.

Im obigen Fall ist dies eine Anfrage an den entfernten Server mit der GET Methode. Die zurückgesendeten Daten sind zwar auch einfach über die normale Facebook Internetseite einsehbar, aber dort sind sie nicht so aufbereitet, dass sie auch maschinenlesbar wären. Das Spannende an REST Systemen ist also die Einfachheit und die Plattformunabhängigkeit, denn so gut wie jede heute verwendete Programmiersprache ist auch ohne Einbindung zusätzlicher Bibliotheken in der Lage, HTTP-Anfragen zu versenden.

Wird für die Anfrage eine andere HTTP-Methode, wie z. B. POST oder DELETE benutzt, ist nach dem REST Paradigma vorgesehen, dass sich damit die über die URI identifizierte Resource verändern bzw. löschen lässt. Zum Glück wird es jemand anderem dann aber doch nicht ganz so leicht gemacht, meine persönlichen Daten zu verändern oder gar zu löschen.

In den meisten Fällen werden diese Systeme durch die Verwendung eines Authentifizierungsmechanismus geschützt. Dabei werden meistens API-Keys vergeben, für die man sich beim Anbieter der API registrieren muss und die dann in irgendeiner Form mit den Anfragen mitgesendet werden, um Zugriff zu erhalten. Oft gibt es auch noch spezielle Schritte, die sicherstellen, dass der Besitzer der Daten auch spezifisch dem API-Key Besitzer eine Nutzung erlaubt hat.

Die Protokolle, die diese Mechanismen regeln, sind in vielen Fällen proprietär, allerdings kommt inzwischen auch in einigen Fällen, wie z. B. bei der relativ neuen Graph API von Facebook oder der API des StudiVZ, das offene OAuth Protokoll zum Einsatz, das einen Authentifizierungs- und Authorisierungsmechanismus beschreibt, mit dem diese Zugänge ebenfalls und in ähnlicher Form geregelt werden können.

Das Schöne an diesem offenen Ansatz ist, dass eine Community existiert, in der die Sicherheit des Protokolls durch sehr viele Augen überprüft wird und dass bereits Bibliotheken für die meisten Programmiersprachen zur Verfügung stehen, die die nötigen Signierungsprozesse übernehmen können.

Der Gedanke, dass APIs, wie sie oben angedeutet wurden (und damit letztendlich auch das von mir erstellte Framework), vor allem von sozialen Netzwerken genutzt werden, um persönliche Informationen von Nutzern für andere zur Verfügung zu stellen, ist sicherlich nicht nur für mich datenschutzrechtlich bedenklich.

Die Entwicklung hin zu offenen Standards wie OAuth oder auch dem [Opensocial-Protokoll](#) in diesem Bereich lässt mich aber darauf hoffen, dass sowohl Datensicherheit als auch die Kontrolle, die wir über unsere eigenen Daten haben, in Zukunft zunehmen werden. Zumindestens steigt die Wahrscheinlichkeit, dass Sicherheitslücken aufgedeckt werden können, wenn die Sicherheit nicht nur durch die Geheimhaltung der Authentifizierungsverfahren gewährleistet wird, sondern die ablaufenden Prozesse auch durch eine Community überprüft werden können.

1.2. Aufgabenstellung

Bei meiner Beschäftigung mit diesen Themen fiel mir auf, dass es bisher zwar viele Webservices gibt, die die oben beschriebenen Techniken bei der Erstellung ihrer APIs nutzen, dies aber meist nur bei großen Internetportalen, die sich problemlos eine Eigenentwicklung solcher Software leisten können, der Fall ist. Außerdem gibt es nach meinem Wissen bisher kein in PHP geschriebenes Framework, welches diese beiden Techniken in leicht nutzbarer Form verbindet.

Da mir die Kombination von REST und OAuth so gut gefiel, fand ich, dass es sich gut eignen würde, um daraus das Thema für meine Bachelorarbeit zu gestalten.

Die Aufgabe des Softwareanteils dieser Arbeit soll also die Erstellung eines Web Application Frameworks für RESTful Webservices mit OAuth Authentifizierung in PHP sein. Die erstellte Software soll von einem Benutzer auf einem eigenen Webserver installiert werden können und dabei nach dem Prinzip „Convention over Configuration“ so wenig Installations- und Konfigurationsaufwand wie möglich erfordern. Falls nötig sollten Anpassungen der festgelegten Konventionen aber dennoch einfach möglich sein.

Im Anschluss an die Installation der Software auf dem Server soll es dem Benutzer dann möglich sein, über geeignete Programmschnittstellen Resources erstellen zu können, die über URIs abgerufen werden können, die in ihrer Konstruktion dem REST Paradigma folgen.

Für den Benutzer soll das Framework vor allem die im Verlauf dieser Arbeit beschriebenen Prozesse bei der Authentifizierung von Benutzern mit Hilfe des OAuth Protokolls und das Wissen über die bei REST verwendeten HTTP Funktionalitäten abstrahieren oder wenn möglich verstecken, sodass für den Anwender eine oberflächliche Beschäftigung mit diesen Themen ausreicht.

Der schönste Abschluss dieses Teils der Arbeit wäre für mich, wenn am Ende ein qualitativ so hochwertiges Framework entstehen würde, dass ich es z. B. unter einer Open Source Lizenz veröffentlichen könnte. Wenn es also tatsächlich in der „freien Wildbahn“ eingesetzt würde und diese Arbeit damit nicht rein akademischer Natur wäre, sondern zusätzlich ein tatsächlicher Nutzen für andere daraus entstehen würde.

1.3. Vorausschau

In der schriftlichen Ausarbeitung dieser Arbeit soll im „Fachlichen Umfeld“ als erstes das nötige Grundverständnis für REST, OAuth und deren mögliche Verbindungspunkte hergestellt werden. Im Anschluss daran werden im „Pflichtenheft“ die Funktionalitäten des zu erstellenden Softwaresystems genau spezifiziert.

Da bei der Entwicklung des Frameworks ein testgetriebener Ansatz Verwendung finden

soll, bei dem es keinen großen anfänglichen Entwurf gibt, werden der Ablauf von Systementwurf und Implementierung in einem gemeinsamen Kapitel „Systementwurf und Implementierung“ behandelt, um die Gleichzeitigkeit dieser beiden Prozesse zu verdeutlichen.

Den Abschluss bilden die beiden Kapitel „Systemtest“ und „Schlussbetrachtung“, in denen der abschließende Test des Gesamtsystems und die gesammelten Erfahrungen erläutert werden.

Für die Benutzung des erstellten Frameworks findet sich in Anhang A eine englischsprachige Anleitung, in der die Installation und die Erstellung von Resources erklärt werden.

1.4. Konventionen

Zur Kenntlichmachung wichtiger Wörter innerhalb des Fließtextes werden folgende typographischen Konventionen verwendet:

Kursiv

erstmals verwendete Fachbegriffe

Konstante Breite

Programmteile innerhalb des Textes, Dateinamen, Pfade, URIs

Fremdsprachliche Fachbegriffe werden meist nicht übersetzt, sondern in ihrer Originalform verwendet.

2. Fachliches Umfeld

Die hauptsächlich verwendeten Verfahren bei dieser Arbeit lassen sich als Internettechnologien charakterisieren. Bei dieser Feststellung spielt natürlich einerseits das Haupteinsatzgebiet eine Rolle, aber auch die Prozesse, die bei der Entstehung der Protokolle und Spezifikationen ablaufen, sind oft sehr spezifisch.

Die Dokumente zu den meisten Protokollen, die im Internet bei der Kommunikation Verwendung finden, werden schon seit April 1969 [Hafner und Lyon, 1998], damals noch zu Zeiten des ARPANET, in so genannten *Request for Comments* oder kurz *RFCs* gesammelt. Dazu zählen inzwischen unter anderem *IPv6 (RFC 2460)*, *FTP¹ (RFC 959)*, *POP3 (RFC 1939)* und eben auch die hier im späteren Verlauf erläuterten und verwendeten *HTTP 1.1 (RFC 2616)* und *OAuth 1.0 (RFC 5849)* [RFC-Editor, kein Datum].

Alle RFCs sind öffentlich einsehbare, unveränderbare Dokumente, deren Sammlung durch den *RFC-Editor*, eine Untergruppe der *ISOC²*, geführt wird. Die ISOC ist eine Nichtregierungsorganisation ohne Gewinnabsicht, die sich vor allem mit der Organisation und Standardisierung von Internetverfahren beschäftigt [ISOC, kein Datum].

Bei Fehlern und Änderungswünschen können RFCs durch ihre Unveränderbarkeit nur durch Erstellung eines neueren Dokuments mit einer neuen Nummer abgelöst werden. Die öffentliche Einsehbarkeit dieser Dokumente dient dabei vor allem dem Prozess des Peer-Review durch die Internetgemeinde und einer etwaigen Standardisierung dieser Dokumente durch eine weitere Untergruppe der ISOC, die *IETF³*.

RFC Dokumente sind immer mit einem Status versehen, um z. B. anzuzeigen, mit welcher Intention sie geschrieben wurden, ihren derzeitigen Zustand im Standardisierungsprozess darzustellen oder ähnliches.

Durch die Schnelllebigkeit des Internets verharren viele Dokumente allerdings ab einem gewissen Zeitpunkt in ihrem derzeitigen Status. Dies geschieht vor allem, wenn nachfolgende RFCs schon in der Entwicklung sind, mit denen aufgetretene Fehler und Schwachstellen behoben, oder neue Ideen in die Dokumente integriert werden sollen.

So befindet sich HTTP 1.1 schon seit seiner Veröffentlichung im Status des „Draft Standard“ ohne bisher die Schwelle zum offiziellen Standard der IETF überschritten zu haben und OAuth wurde erst 2010 von der OAuth Community, die bis dahin mit der

¹File Transfer Protocol

²Internet Society

³Internet Engineering Task Force

Entwicklung beschäftigt war, an die IETF im Status „Informational“ übergeben.

Eine Arbeitsgruppe der IETF ist inzwischen schon mit der Weiterentwicklung von OAuth beschäftigt, um eine Standardisierung nach den Richtlinien der IETF zu ermöglichen. Da dazu schon ein erster Entwurf entstanden ist [IETF, 2010], wird sich der Status von OAuth 1.0 wohl auch in Zukunft nicht mehr ändern, sondern das Dokument einfach durch OAuth 2.0 abgelöst werden.

Dank der Unveränderbarkeit der RFCs lässt sich aber klar definieren, welche Protokollversion für eine Implementierung genutzt wird.

2.1. REST – Representational State Transfer und HTTP

Bei REST handelt es sich um einen Architekturstil für eine Client-Server Kommunikation, der von Roy Fielding, einem der Autoren von RFC 2616, dem HTTP 1.1 Protokoll [Roy Fielding u. a., 1999], in seiner Doktorarbeit zum Thema „Architectural Styles and the Design of Network-based Software Architectures“ [Fielding, 2000] erstmals definiert wird.

Die Architektur wird von Fielding zwar auch in allgemeinerer Form beschrieben, durch seine Involvierung in die Spezifizierung von HTTP beruht sie aber sehr stark auf den Möglichkeiten des HTTP 1.1 Protokolls und soll hier auch hauptsächlich anhand dessen erläutert werden, da eine tiefere Analyse im Rahmen dieser Arbeit nicht sinnvoll erscheint.

Die Hauptelemente von REST sind *Resources*, bei denen es sich um irgendeine Art von „benennbarer Information“ [Fielding, 2000, Kapitel 5.2.1.1] handeln kann. Eine Resource wird dabei durch einen *Resource Identifier* eindeutig identifiziert. Bezogen auf das WWW oder „Human Web“ [Richardson und Ruby, 2007, Seite 18] sind diese beiden Elemente altbekannt und würden z. B. Webseiten und dazugehörige Internetadressen in Form von URIs sein.

Wichtig ist dabei, dass die Daten, die eine Resource auf eine Anfrage zurücksendet, zeit- und anfrageabhängig sind. Der Zustand der Daten wird dabei als *Representation* bezeichnet. Wenn man beispielhaft die Resource Identifier „`http://example.com/neusteVersion`“ und „`http://example.com/version/1.0`“ hätte, könnte es sein, dass diese zu einem bestimmten Zeitpunkt auf eine Anfrage mit der gleichen Representation antworten würden, es handelt sich dabei im Sinne von Fielding aber trotzdem um zwei verschiedene Resources, da sie semantisch unterschiedlich sind.

Außerdem kann es dem Client je nach Implementierung auch möglich sein in seiner Anfrage bestimmte Metainformationen mitzusenden, um z. B. ein bevorzugtes Datenformat anzugeben, in dem die angeforderten Daten zurückgegeben werden sollen.

Die Kommunikation von Client und Server findet bei REST immer zustandslos statt,

d. h. in jeder Anfrage des Clients müssen alle benötigten Informationen enthalten sein, die zur Generierung der Representation oder der Ausführung bestimmter Aktionen mit der Resource benötigt werden. Für den Versand dieser Metadaten und den Versand der Representations werden die Funktionen der im HTTP-Protokoll definierten *HTTP-Requests* und *HTTP-Responses* genutzt.

2.1.1. HTTP-Requests und HTTP-Responses in REST Systemen

HTTP-Requests und Responses sind sich im großen und ganzen sehr ähnlich. Sie bestehen hauptsächlich aus einer beliebigen Anzahl von *Headern* und einem optionalen *Message Body*. In der Spezifikation werden 46 verschiedene Header Felder definiert, die außerdem noch durch benutzerdefinierte erweitert werden können.

Der Unterschied der beiden Nachrichtentypen liegt vor allem in der ersten Zeile der gesendeten Nachricht. Bei einem Request besteht diese aus der gewünschten *HTTP-Methode*, die mit der angeforderten Resource ausgeführt werden soll, einem relativen Resource Identifier und der verwendeten Protokollversion[Roy Fielding u. a., 1999, Kapitel 4].

Listing 2.1: HTTP Request

```
1 POST ExampleResource HTTP/1.1
2 Host: localhost
3 Accept: text/html, application/xhtml+xml, application/xml;q=0.9,*
    /*;q=0.8
4 Accept-Language: de-de, de;q=0.8,en-us;q=0.5,en;q=0.3
5
6 Der Message Body.
```

Im obigen Beispiel soll so die Methode *POST* für die Resource mit dem Identifier „`http://localhost/ExampleResource`“ ausgeführt werden. Dieser absolute Identifier setzt sich dabei aus dem relativen Resource Identifier aus Zeile 1 und dem *Host* Header Feld aus Zeile 2 von Listing 2.1 zusammen. Außerdem wird durch die Header *Accept* und *Accept-Language* bekannt gegeben, dass die Antwort bevorzugt ein HTML Dokument in deutscher Sprache sein soll.

Bei einer Response beginnt dahingegen die erste Zeile mit der Protokollversion, gefolgt von einem *HTTP-Status*:

Listing 2.2: HTTP Response

```
1 HTTP/1.1 401 Unauthorized
2 Date: Tue, 11 Jan 2011 12:41:25 GMT
3
4 Der Message Body.
```

Diese beiden Arten von Nachrichten sind die Grundbausteine, die für das Zustandekommen einer RESTful Kommunikation benötigt werden. Im Message Body wird dabei die Representation und in den Header-Feldern die Metainformationen verstaut, während der HTTP-Status für eventuelle Fehler- oder Erfolgsmeldungen verwendet wird.

Der Unterschied und Vorteil im Nachrichtenversand von REST gegenüber dem der häufig verwendeten RPC Variante SOAP liegt darin, dass eine Ebene des Verpackens ausgelassen wird. SOAP verwendet HTTP einzig und allein als Transportmechanismus und sendet einen eigenen *Envelope* im Message Body des Requests, der in seiner Struktur einer HTTP Nachricht stark ähnelt.

Listing 2.3: SOAP HTTP Request

```
1 POST ExampleUri HTTP/1.1
2 Host: localhost
3
4 <?xml version="1.0"?>
5 <s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
6     <s:Header>
7     </s:Header>
8     <s:Body>
9     </s:Body>
10 </s:Envelope>
```

In Listing 2.3 ist beispielhaft eine solche einfache SOAP Nachricht im Message Body einer HTTP Anfrage zu sehen. Die Metainformationen für die SOAP Nachricht werden dabei in dem, in den Zeilen sechs und sieben zu sehenden Header des gesendeten XML Dokuments verstaut.

Die HTTP-Methode

HTTP hat acht verschiedene Standardmethoden [Roy Fielding u. a., 1999, Kapitel 9], mit denen dem Server im Request mitgeteilt werden kann, welche Aktion mit der angeforderten Resource ausgeführt werden soll und die genauso wie die Header Felder durch weitere ergänzt werden können. Die wichtigsten Standardmethoden und ihre Funktionalitäten für eine RESTful Kommunikation sind:

GET

Fordert eine Representation einer Resource an.

POST

Erstellt eine neue Resource als Unterpunkt der gesendeten URI aus den Daten des Message Body.

PUT

Verändert eine Resource mit den Daten des Message Body.

DELETE

Löscht eine Resource.

HEAD

Fordert nur den Header einer Response an, z. B. um Informationen über die zu erwartende Größe des Message Body zu erhalten.

OPTIONS

Gibt Aufschluss über die unterstützten HTTP Methoden der Resource, wird allerdings bisher nur selten eingesetzt.

POST, GET, PUT und DELETE bilden dabei in dieser Reihenfolge die Funktionsweisen der für die Datenmanipulation benötigten Grundfunktionen „CRUD – Create, Read, Update, Delete“ [Heller, 2007] ab und reichen damit für die meisten Anwendungsfälle üblicher REST-APIs aus.

Sie bilden außerdem das sogenannte *Uniform Interface* [Fielding, 2000, Kapitel 5.1.5]. Ein weiteres Merkmal von REST, welches hervorhebt, dass Resources ein Vokabular von wohldefinierten Methoden unterstützen können und ein Klient damit eine Erwartungshaltung gegenüber der Funktionsweise dieser Methoden einnehmen kann. Dies stellt einen großen Vorteil bei der Erlernbarkeit von REST Systemen dar, da bei einer intuitiv zugänglichen Benennung der Resources kaum Lernaufwand für eine neue API entsteht.

Die Erweiterbarkeit der HTTP-Methoden findet z. B. bei der Webversion von Subversion Anwendung, das *WebDAV*, eine Erweiterung von HTTP [IETF, 2007] verwendet, welches unter anderem die weiteren Methoden *LOCK* und *UNLOCK* definiert.

Header-Felder

Bei den vordefinierten Header-Feldern gibt es einige, die nur für einen der beiden Nachrichtentypen festgelegt und sinnvoll sind, im Folgenden werden die im Sinne von REST wichtigsten und interessantesten vorgestellt:

Accept

Request Header. Ermöglicht es einem Client präferierte MIME-Types⁴, also das Format für die Response anzugeben und über den „q“ Faktor deren Wichtigkeit zu definieren. Dies kann z. B. genutzt werden, um die Resource in einer XML oder JSON⁵ Representation anzufordern. Ein Beispiel ist in Zeile 3 von Listing 2.1 zu sehen.

⁴MIME steht für Multi Purpose Internet Mail Extension. Es handelt sich dabei um eine Angabe zum Datenformat einer gesendeten Nachricht, die zuerst beim Emailversand eingesetzt wurde. Sie bestehen aus einem Typ und einem Subtyp, z. B. `text/plain` für normalen Text, `text/html` für HTML oder `application/json` für JSON.

⁵JavaScript Object Notation – Ein im Internet häufig verwendetes Datenformat, welches aus der Sprache JavaScript entstanden ist. Es stehen in sehr vielen Programmiersprachen Parser zur Verfügung.

Accept-Language

Request Header. Kann genutzt werden, um die gewünschte Sprache der Representation der angeforderten Resource zu benennen und setzt sich aus einem jeweils von der ISO festgelegten Sprachcode und einem Ländercode zusammen, z. B. „en-gb“ für britisches Englisch, „en-us“ für amerikanisches Englisch oder „de-at“ für österreichisches Deutsch.

Content-Length

Response Header. Wird benutzt, um die Länge des Message Body anzugeben, vor allem interessant bei Verwendung der HEAD Methode, um vorab nur die zu erwartende Größe einer Representation anzufordern.

Content-Type

Response Header. Gibt den MIME-Type der Daten im Message Body an, funktioniert als Gegenstück zu Accept, um dem Client mitzuteilen wie die erhaltenen Daten zu interpretieren sind.

Location

Response Header. Weist den Klienten an, nach Erhalt der Antwort eine neue Anfrage an die im Location Header gesendete URI zu senden.

WWW-Authenticate

Response Header. Wird benutzt, um dem Anfragenden ein Authentifizierungsverfahren mitzuteilen, das er zu verwenden hat.

Mit diesen Headern können die üblichsten Metainformationen gesendet werden, die bei der Erstellung einer REST-API benötigt werden. Für Fälle, in denen die Standardheader nicht ausreichen, gibt es aber auch hier wieder die Möglichkeit eigene Felder zu definieren.

Listing 2.4: Header Feld

```
1 X-Custom-Header : Value
```

Diese werden üblicherweise mit einem vorangestellten „X-“ von den Standardheadern abgegrenzt. Die Header stehen dabei immer in einer eigenen Zeile, in der Name und Inhalt des Feldes durch einen Doppelpunkt getrennt werden.

Http-Status

Http-Status setzen sich aus einem *Status Code* in Form einer dreistelligen Zahl und einem *Reason Phrase*, der die Bedeutung des Codes meistens in drei bis vier Worten zusammenfasst, zusammen. Sie werden bei REST benutzt, um dem Client eine Rückmeldung über Erfolg oder Mißerfolg der Anfrage zu geben.

Es gibt dabei einige im HTTP Standard vordefinierte Status, es wird aber auch die Definition eigener Codes zugelassen. Die Status Codes sind in fünf Kategorien einge-

teilt, die durch die erste Ziffer des Codes bestimmt ist. In der folgenden Liste sind die Kategorien und die im Sinne von REST jeweils wichtigsten Status Codes aufgelistet:

1xx für Informational

nicht relevant für REST

2xx für Successful

- 200 OK – die Anfrage war erfolgreich
- 201 Created – z. B. nach erfolgreichem POST Request, um anzuzeigen, dass eine Resource erstellt wurde, oft in Verbindung mit einem Location Header um mitzuteilen, unter welcher URI die erstellte Resource zu finden ist.

3xx für Redirection

- 301 Moved Permanently – Die angeforderte Resource ist nun unter einem anderen Resource Identifier zu erreichen.
- 303 See Other – Bei zwei Resources, deren Representations zur Zeit gleich sind, kann eine von beiden damit darauf verweisen, die Representation an einer anderen Stelle abzurufen, z. B. in Verbindung mit einem Location Header.

4xx für Client Error

- 400 Bad Request – Das HTTP Request war fehlerhaft.
- 401 Unauthorized – Im Request waren nicht genug Informationen für eine Authorisierung zum Zugang zur angeforderten Resource enthalten.
- 403 Forbidden – z. B. bei Resources, die nur zu bestimmten Tageszeiten erreichbar sein sollen.
- 404 Not Found – Zu dem im Request genannten Resource Identifier kann keine Resource gefunden werden.
- 405 Method Not Allowed – Die angeforderte Methode wird nicht unterstützt.

5xx für Server Error

- 500 Internal Server Error – Es ist ein genereller Fehler beim Server aufgetreten, der nicht genauer spezifiziert werden kann.
- 501 Not Implemented – Die im Request geforderte HTTP Methode ist noch nicht vollständig implementiert.

2.1.2. Weiterführende Literatur

Für eine weitere Beschäftigung mit REST bieten sich die hervorragenden O'Reilly Bücher „RESTful Web Services“ von [Richardson und Ruby \[2007\]](#) und „RESTful Web Services Cookbook“ von [Allamaraju \[2010\]](#) an, die vor allem für den praktischen Einsatz von REST ein gutes Verständnis bieten.

Wer sich noch tiefgreifender mit diesen Themen auseinandersetzen möchte, der sollte wohl vor allem die Kapitel fünf und sechs aus [Fieldings Doktorarbeit](#) und die Spezifikation des [HTTP 1.1 Protokolls](#) lesen.

2.2. Das OAuth Protokoll

Die Arbeit am OAuth Protokoll begann Ende 2006 mit der Intention, einen offenen Standard für eine „bevollmächtigte Authentifizierung“ [[Hammer-Lahav, 2007](#)] zu schaffen.

Eine Notwendigkeit dafür wurde vor allem von Unternehmen aus dem Umfeld des Web 2.0 wie Twitter und Google gesehen, um den auf ihren Webseiten geschaffenen „User-generated content“ in einer für den Ersteller nachvollziehbaren und sicheren Weise Dritten zur Verfügung zu stellen.

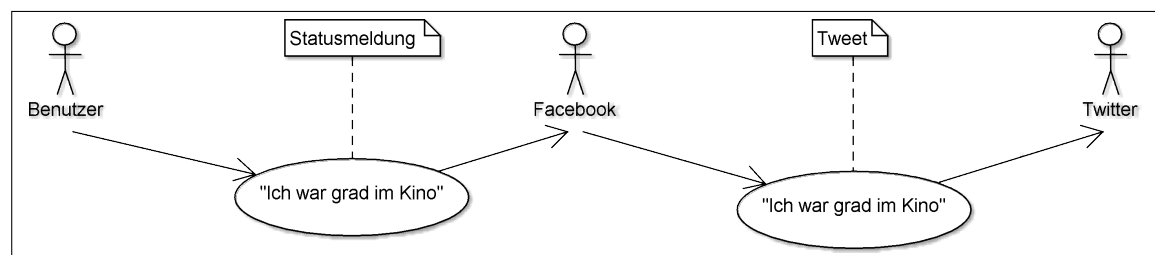


Abbildung 2.1.: Loginweitergabe

Ein Fall für solch einen Vorgang ist in Abbildung 2.1 zu sehen. Ein Benutzer, der eine Statusmeldung bei einem sozialen Netzwerk, wie z. B. Facebook postet und dann erreichen möchte, dass diese Meldung an seinen Twitter Account weitergereicht wird, damit auch seine dortigen „Follower“ diese Meldung lesen können.

Ohne eine Möglichkeit der bevollmächtigten Authentifizierung ist der Weg dies zu erreichen die Herausgabe der Twitter Logindaten an Facebook, um Facebook ein Einloggen im Namen des Benutzers und damit das „twittern“ zu ermöglichen.

Dass eine Verteilung von Logindaten kein besonders sicheres Verfahren für den Benutzer darstellt, dürfte einleuchten und wird vor allem deutlich anhand einer Microsoft Studie aus dem Jahr 2007, bei der festgestellt wurde, dass ein Durchschnittsbenutzer 6,5 verschiedene Passwörter verwendet und jedes Passwort von ihm auf rund vier verschie-

denen Seiten genutzt wird [Florencio und Herley, 2007].

Sollte also eine Sicherheitslücke auf einer dieser vier Seiten auftreten und die Logindaten eines Benutzers gestohlen werden, so wären auch die Accounts auf den anderen drei Seiten, auf denen dieses Passwort benutzt wurde, gefährdet oder sogar kompromittiert.

Um diese Problematik zu vermeiden definiert das OAuth Protokoll ein Verfahren, um im Auftrag eines *Resource Owners*, also dem Benutzer aus der obigen Abbildung, einen *Token* zu generieren, der es einem *OAuth Client*, im Beispiel Facebook, ermöglicht eine geschützte Resource aufzurufen [Hammer-Lahav, 2010].

Die geschützte Resource wäre in diesem Fall also ein Teil der Twitter-API, der es bei Empfang des richtigen Token von einem authentifizierten Client ermöglicht, im Namen des Resource Owners zu twittern.

Ohne die Generierung solch eines Tokens kann das Verfahren, wie wir später sehen werden, auch genutzt werden, um per HTTP einen Login eines Client zu ermöglichen. Diese Variante wird häufig als *2-legged OAuth* bezeichnet, wohingegen der oben beschriebene Prozess *3-legged OAuth* genannt wird [Google, kein Datum].

Dies ist jeweils ein Hinweis auf die Anzahl der am Prozess beteiligten Teilnehmer. Beim 2-legged Verfahren findet die Kommunikation nur zwischen zwei Teilnehmern, dem Client und dem Server statt, wohingegen beim 3-legged Prozess noch der Resource Owner hinzu kommt.

Die eigentliche Basis des Protokolls ist die Möglichkeit authentifizierte Anfragen an einen Server zu senden. Dies geschieht durch das Anhängen einiger genau definierter erforderlicher Parameter an das HTTP-Request und dem anschließenden Erstellen einer digitalen Signatur über die Daten dieses Requests, die dann wie die Parameter ebenfalls an das Request angehängt wird.

Für das Versenden der Parameter gibt es drei verschiedene Wege:

Authorization Header

Die Parameter werden im „Authorization“ Header Feld verstaut:

Listing 2.5: OAuth Credentials im Authorization Header

```
1 GET /Example HTTP/1.1
2 Host: example.com
3 Authorization: OAuth realm="http://example.com/Example",
  oauth_consumer_key="
  fd9bb31bf9c95d0905e8eb86499ac871bb5ef3fa2", oauth_token="
  ", oauth_nonce="gfshdsfhdfhtz", oauth_timestamp="
  1296492784", oauth_signature_method="HMAC-SHA1",
  oauth_version="1.0", oauth_signature="yEG8Fo%2
  FQXSItYxEDre3hIsDx6U%3D"
```


Query String

Die Parameter werden in Form eines Query String an die URI angehängt:

Listing 2.6: OAuth Credentials im Query String einer URI

```
1 http://example.com/path?oauth_consumer_key=
  fd331bf9c95d0905e8eb86499ac871bb5ef3fa2&oauth_nonce=
  gfsfhsfhdftz&oauth_signature_method=HMAC-SHA1&
  oauth_timestamp=1296492784&oauth_token=&oauth_version
  =1.0&oauth_signature=yEG8Fo/QSXSIYxEDre3hIsDx6U=
```

Message Body

Die Parameter werden im Message Body gesendet, diese Technik ist nur in sehr wenigen Einzelfällen erlaubt.

Zu den Parametern zählen u. a. ein aktueller Zeitstempel, eine einzigartige Zeichenkette, die vom Klienten nur für diese Anfrage verwendet wird und ein Identifier des Klienten.

2.2.1. Die OAuth Signatur

Um eine HTTP Anfrage mit OAuth signieren zu können, bedarf es zu allererst *Client Credentials*, die aus einem *Client Key* und einem *Client Shared Secret* bestehen [Hammer-Lahav, 2010, Seite 13 ff.]. Wie genau diese auszusehen haben wird nicht definiert. Im Prinzip handelt es sich um irgendeine Form von Zeichenketten, die als Logindaten verwendet werden.

Das Shared Secret ist dabei also nicht öffentlich zugänglich, sondern wird nur zwischen dem Server und dem Client geteilt und zur Erstellung bzw. der Kontrolle der Signatur genutzt.

Da die Berechnung der Signatur programmatisch erfolgt und die Daten somit nicht von einem Benutzer auswendig gelernt werden müssen, werden bei der Registrierung eines Clients in den meisten Fällen 32-40 Zeichen lange Hashwerte für diesen Zweck automatisch generiert:

Listing 2.7: Beispiel Flickr Api Credentials

```
1 oauth_consumer_key=9a0554259914a86fb9e7eb014e4e5d52
2 oauth_consumer_secret=a02506b31c1cd46c2e0b6380fb94eb3d
```

Der erste Schritt im eigentlich Signierungsprozess ist die Generierung des sogenannten *Signature Base String*, der eine, in der Spezifikation beschriebene, normalisierte Form der im HTTP-Request enthaltenen Daten ist. Dies ist notwendig, um eine Vergleichbarkeit von erstellten Signaturen zu erreichen, da auf der Empfängerseite eine erneute Signierung vorgenommen wird, die dann mit der vom Client gesendeten verglichen wird.

Listing 2.8: Beispiel eines Signature Base String aus der OAuth Spezifikation[[Hammer-Lahav, 2010](#), Kapitel 3.4.1.1]

```
1 POST&http%3A%2F%2Fexample.com%2Frequest&a2%3Dr%2520b%26a3%3D2
   %2520q%26a3%3Da%26b5%3D%25253D%25253D%26c%2540%3D%26c2%3D%26
   oauth_consumer_key%3D9djdj82h48djs9d2%26oauth_nonce%3
   D7d8f3e4a%26oauth_signature_method%3DHMAC-SHA1%26
   oauth_timestamp%3D137131201%26oauth_token%3Dkkk9d7dh3k39sjv7
```

Wie in Listing 2.8 zu erkennen, besteht diese Zeichenkette aus einer Aneinanderreihung von HTTP-Methode, dem Host, dem Resource Identifier, aller mitgesendeten Parameter des Protokolls und unter speziellen Voraussetzungen auch weiterer Parameter. Diese werden mit „&“ verbunden und anschließend mit einer ebenfalls spezifizierten *Percent Encoding* umgewandelt [[Hammer-Lahav, 2010](#), Seite 17].

Für die Erstellung der Signatur werden, wie schon beim Datenversand, auch wieder drei verschiedene Verfahren angeboten: *Plaintext*, *RSA-SHA1* und *HMAC-SHA1* von denen nur HMAC-SHA1 ohne eine weitere Sicherung des Transportmechanismus, wie z. B. SSL/TLS⁶, auskommt und deswegen wohl das am häufigsten verwendete Verfahren darstellt.

Wie diese Prozesse im Detail ablaufen kann in der Spezifikation des OAuth Protokolls nachgelesen werden. Wichtig für die spätere Überprüfung der Signatur ist lediglich, dass Signaturmethode, Signatur und Identifier des Client in den oben genannten Parametern enthalten sein müssen.

Dadurch kann auf der Serverseite das Client Secret aus den gespeicherten OAuth Client Daten ausgelesen werden, ebenfalls der Signature Base String gebildet werden und dann die gleiche Methode zur Erstellung der Signatur angewandt werden.

Der Signierungsprozess funktioniert dabei im Prinzip unabhängig von der im folgenden erläuterten *Redirection-Based Authorization*, durch die die oben schon erwähnten Token generiert werden. Der Server muss also gesondert prüfen, ob ein Token für die angefragte Resource benötigt wird und ob dieser valide ist.

Dadurch dass der `oauth_token` Parameter auch eine leere Zeichenkette enthalten kann, wird die Möglichkeit des 2-legged OAuth gegeben, da bei einer validen Signatur davon ausgegangen werden kann, dass die Anfrage von jemandem gesendet wurde, der sowohl Client Key als auch Client Secret kennt. Damit wird die eindeutige Identifizierung eines Clients ermöglicht.

⁶Secure Socket Layer/Transport Layer Security – Ein Verschlüsselungsmechanismus für Internetkommunikation, wie z. B. bei HTTPS.

2.2.2. Redirection-Based Authorization – Der „OAuth Dance“

Das Verfahren für die bevollmächtigte Authentifizierung, oben schon als 3-legged OAuth bezeichnet, wird wahrscheinlich ebenso häufig auch „OAuth Dance“ genannt. Dies ist darauf zurückzuführen, dass es sich um einen Prozess handelt, der aus mehreren Schritten besteht, bei denen der Resource Owner zwischen Consumer und dem Server hin und her geleitet wird, um einen Token zu generieren. Der offizielle Name innerhalb des Protokolls lautet allerdings Redirection-Based Authorization.

Um die Generierung des Token zu ermöglichen schreibt das Protokoll vor, dass der implementierende Server drei URIs, die auch als *OAuth Endpoints* bezeichnet werden, zur Verfügung stellt, die jeweils genau definierte Schritte ausführen müssen [Hammer-Lahav, 2010, Seite 7 ff.]:

Temporary Credential Request URI

z. B. unter der URI `https://api.example.com/initiate`

Resource Owner Authorization URI

z. B. unter der URI `http://api.example.com/authorize`

Token Request URI

z. B. unter der URI `https://api.example.com/token`

Die Benennung der URIs ist dabei dem Ersteller überlassen, muss aber in irgendeiner Form den Nutzern des Dienstes mitgeteilt werden. Meist geschieht dies entweder über eine online verfügbare Dokumentation des bereitgestellten Service oder während des Registrierungsprozesses für Client Credentials.

Sowohl der Temporary Credential als auch der Token Request Endpunkt müssen laut Protokoll über eine HTTPS URI angefragt werden, da bei diesen beiden Anfragen sonst sicherheitskritische Parameter im Klartext übertragen würden.

Ablauf

Im Folgenden werden nun anhand der Nummerierungen innerhalb des Sequenzdiagramms aus Abbildung 2.2 die einzelnen Schritte der Redirection-Based Authorization erläutert. Vorausgesetzt werden dafür ein beim Server registrierter OAuth Client und ein dem Server bekannter Resource Owner.

1. Eine Aktion des Resource Owners sorgt dafür, dass der Client Zugriff auf Daten des Servers benötigt. Meist handelt es sich beim Resource Owner um einen normalen Internetbenutzer, der z. B. wie im Beispiel aus Abbildung 2.1 über die Facebook Internetseite eine „App“ installiert, die seine Statusmeldungen an Twitter weiterreichen soll.

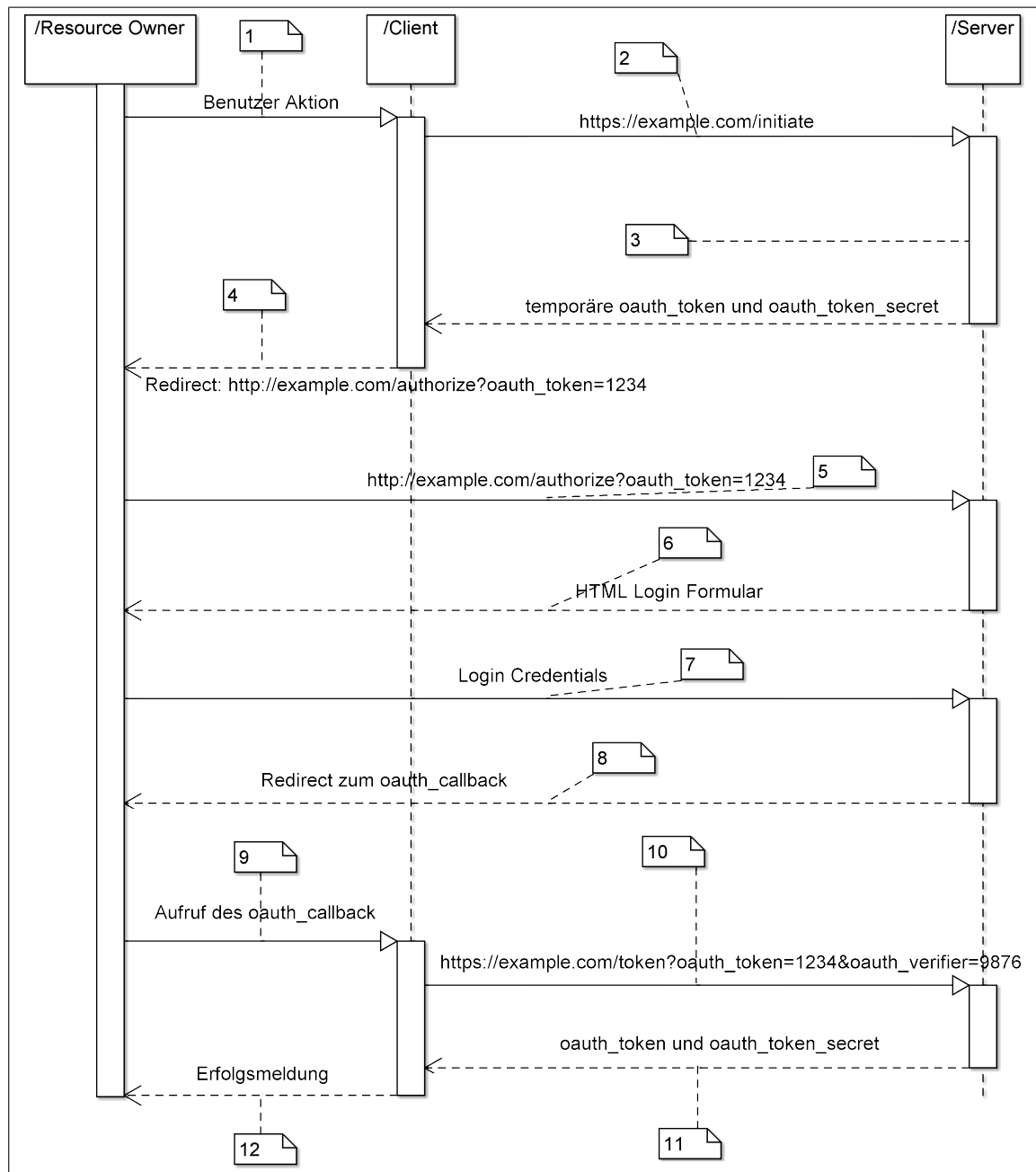


Abbildung 2.2.: OAuth Sequenzdiagramm

2. Der Client beginnt nun den Versuch, einen Token zum Zugriff auf die Daten des Resource Owners zu erhalten. Dazu sendet er eine signierte Anfrage an den **Temporary Credentials Request** Endpunkt, bei der er den später benötigten `oauth_callback` Parameter mitsendet, der einfach eine URI auf der Seite des

Client ist, die wissen muss, was mit den in Schritt 9 zurückgesendeten Daten anzufangen ist.

3. Der Server generiert einen temporären `oauth_token` und ein zugehöriges `oauth_secret` und speichert diese zusammen mit dem `oauth_callback` und den Informationen über den anfragenden Client.

An den Client werden `oauth_token` und `oauth_secret` zurückgesendet. Dies ist der Grund für die Erfordernis einer HTTPS-Kommunikation, da diese Daten ansonsten einfach abgefangen werden könnten, u. U. für einen Angriff verwendet werden könnten.

4. Dem Resource Owner wird vom Client die **Resource Owner Authorization** URI des Servers mit dem im vorherigen Schritt generiertem temporären Token im Query String übergeben.
5. Der Resource Owner ruft die ihm übergebene URI auf. Meist geschieht dies durch eine Weiterleitung mittels eines Location Headers, der in der HTTP Response zu dem in Punkt 1 gesendeten Request enthalten ist, die in Punkt 4 gesendet wird. Daraufhin ruft der Browser des Resource Owners automatisch die dort gesendete URI auf.
6. Der Resource Owner erhält vom Server ein Login Formular, um ihn als Resource Owner zu authentifizieren. Im Formular wird dem Resource Owner mitgeteilt, welcher Client Zugriff auf seine Daten erhalten möchte.
7. Der Resource Owner sendet seine Login-Daten an den Server.
8. Bei erfolgreichem Login wird der Resource Owner an eine URI weitergeleitet, die aus `oauth_callback` und einem zum temporären Token gehörenden `oauth_verifier` im Query String besteht, der vom Server zum temporären Token gespeichert wird.
9. Der Resource Owner ruft den `oauth_callback` auf und teilt dem Client damit den zu seinem temporären Token gehörenden `oauth_verifier` mit.
10. Indem der Client nun den temporären `oauth_token` und den `oauth_verifier` an den **Token Request** Endpunkt sendet, teilt er dem Server mit, dass er vom Resource Owner autorisiert wurde einen „echten“ Token zu erhalten, der Zugriff auf die Resources des Resource Owners gewährt. Da der Server den Verifier und den Token zusammengehörend gespeichert hat, kann dies überprüft werden.
11. Sollten der Token und der Verifier zusammenpassen und die Gültigkeit des temporären Tokens noch nicht abgelaufen sein, sendet der Server einen `oauth_token` und das dazugehörige `oauth_token_secret` zurück.
12. Der Resource Owner erhält mit der HTTP Response eine Erfolgsmeldung vom Client.

Der erhaltene Token wird auf Seiten des Clients gespeichert und kann damit jedesmal, wenn eine Resource des Resource Owners angefragt wird, mitgesendet werden. Um die Sicherheit für und die Kontrollierbarkeit durch den Resource Owner zu erhöhen, sollte ein Token über ein Verfallsdatum verfügen und außerdem ein Mechanismus bereitgestellt werden, mit dem die Gültigkeit des Token widerrufen werden kann. Dies liegt allerdings in der Verantwortung des Servers.

2.2.3. Weiterführende Literatur

Um ein genaues Bild der Verfahren von OAuth zu erlangen ist es praktisch unerlässlich, die Spezifikation aus RFC 5849 zu studieren:

- <http://tools.ietf.org/html/rfc5849>

Für eine Verbindung von OAuth und REST findet sich ein kleiner Abschnitt in Kapitel 12 von „RESTful Web Services Cookbook“ von Allamaraju [2010] und einige gut geschriebene kurze Einführungen lassen sich unter folgenden Internetadressen finden:

- <http://www.heise.de/developer/artikel/Autorisierungsdienste-mit-OAuth-845382.html>
- <http://hueniverse.com/oauth/>
- <http://developers.facebook.com/docs/authentication/>
- <http://developer.studivz.net/wiki/index.php/OAuth>

2.3. REST und OAuth kombiniert – Ein Beispiel

Um die Verbindung aus REST und OAuth noch einmal zu verdeutlichen, sollen hier nun eine imaginäre Foto-Community, ähnlich „flickr“ und eine Anwendung eines Beispielclients beschrieben werden, die diese beiden Techniken nutzen.

Der imaginäre Fotodienst ist sehr einfach gestaltet, er stellt unter `http://photos.example.com` eine Webseite bereit, an der sich beliebige Internetnutzer registrieren können und denen nach einem Login folgende Funktionalitäten zur Verfügung stehen:

- eigene Bilder hochladen
- eigene Bilder löschen

Hochgeladene Bilder sind dabei über die Webseite für jeden angemeldeten Nutzer zugänglich.

Außerdem betreibt diese Community unter der URI `http://api.photos.example.`

com einen eigenen RESTful Web Service, für den sich Clients unter `https://api.photos.example.com/register` registrieren können und dort OAuth Credentials erhalten.

Die Idee einer Softwarefirma ist es nun zum Beispiel, eine Handyanwendung für diesen Dienst bereitzustellen, mit dem die gleichen Funktionalitäten wie auf der Webseite ausgeführt werden können, mit dem Vorteil einer an den kleinen Bildschirm angepassten Benutzungsoberfläche. Diese Anwendung soll dann über einen „Market“ oder „Appstore“ verkauft werden.

Aus Sicht von OAuth wäre diese Anwendung der Client. Der Accountbesitzer der Community, der sich diese „App“ installiert, wäre der Resource Owner und die Community der Server.

Um die Funktionalitäten nachzubilden, erstellt sich die Softwarefirma also als erstes einen Account für die API der Fotocommunity, um dessen Credentials in der Anwendung zu verwenden. Nach der Installation und vor der ersten Verwendung der Software auf dem Handy eines Resource Owners wird dann die Benutzer-ID abgefragt und über das unter 2.2.2 beschriebene Verfahren der Resource Owner Authorization ein Token generiert.

Mit diesem Token und der Benutzer-ID kann die Anwendung dann die von der API bereitgestellten Endpunkte aufrufen, deren Verwendung unter der URI `http://developers.photos.example.com` erläutert wird:

- `http://api.photos.example.com/photo`
- `http://api.photos.example.com/photos`

Nach dem Studium der Dokumentation ist dem Entwickler der Anwendung klar, dass er zum Hochladen eines Bildes ein mit OAuth signiertes HTTP-Request an den `photo` Endpunkt erstellen muss, bei dem ein Token mitgesendet wird. Das Request muss die Methode `POST` verwenden, der HTTP-Header `Content-Type` gibt an, welches Format das gesendete Bild hat und das eigentliche Bild wird im Message-Body verstaut:

Listing 2.9: Upload eines Fotos

```
1 POST /photo HTTP/1.1
2 Host: api.photos.example.com
3 Content-Type: image/gif
4 Authorization: OAuth realm="http://api.photos.example.com",
    oauth_consumer_key="fdbb31bf9c95d0905e8eb86499ac871bb5ef3fa2",
    oauth_token="8be97cc2e765421ae63271f0e657c778c24050df",
    oauth_nonce="gfshdsfhdfhtz", oauth_timestamp="1296492784",
    oauth_signature_method="HMAC-SHA1", oauth_version="1.0",
    oauth_signature="yEG8Fo%2FQXSXSIYxEDre3hIsDx6U%3D"
5
6 Bilddaten
```

Die Antwort des Servers besteht aus einer HTTP-Response, in der, falls der Status 201 **Created** lautet, innerhalb des **Location** Header die URI der neu erstellten Resource zurückgesendet wird. In diesem Fall der Endpunkt mit einer angehängten ID des neuen Bildes:

Listing 2.10: RESTful upload eines Fotos

```
1 HTTP/1.1 201 Created
2 Location: http://api.photos.example.com/photo/123456789
```

Sollte der Status anders lauten ist an irgendeiner Stelle ein Fehler aufgetreten und die Anwendung kann eine Fehlermeldung anzeigen.

Für das direkte Löschen dieses Bildes müsste als Methode des nächsten HTTP-Requests nur **DELETE** verwendet werden, als URI die aus dem oben erhaltenen **Location** Header angegeben werden und natürlich eine Signatur erstellt werden:

Listing 2.11: Löschen eines Fotos

```
1 DELETE /photo/123456789 HTTP/1.1
2 Host: api.photos.example.com
3 Authorization: OAuth realm="http://api.photos.example.com",
  oauth_consumer_key="fdbb31bf9c95d0905e8eb86499ac871bb5ef3fa2",
  oauth_token="8be97cc2e765421ae63271f0e657c778c24050df",
  oauth_nonce="gfshdsfhdfhtz", oauth_timestamp="1296492784",
  oauth_signature_method="HMAC-SHA1", oauth_version="1.0",
  oauth_signature="yEG8Fo%2FQXSXItYxEDre3hIsDx6U%3D"
```

Da im Normalfall Bilder nicht sofort gelöscht werden sollen, sondern diese in irgendeiner Form ausgewählt werden müssten, kann die Anwendung über den **photos** Endpunkt der API eine Liste aller Bilder eines Resource Owners anfordern.

Da Bilder jedem angemeldeten Nutzer zur Verfügung stehen, reicht der API in diesem Fall eine Authentifizierung über einen 2-legged Prozess, es muss also kein Token mitgesendet werden:

Listing 2.12: Anforderung einer Liste von Fotos eines Benutzers

```
1 GET /photos/Benutzer-ID HTTP/1.1
2 Host: api.photos.example.com
3 Authorization: OAuth realm="http://api.photos.example.com",
  oauth_consumer_key="fdbb31bf9c95d0905e8eb86499ac871bb5ef3fa2",
  oauth_token="", oauth_nonce="gfshdsfhdfhtz",
  oauth_timestamp="1296492784", oauth_signature_method="HMAC-SHA1",
  oauth_version="1.0", oauth_signature="yEG8Fo%2FQXSXItYxEDre3hIsDx6U%3D"
```

Als Antwort würde die Anwendung daraufhin eine XML-Datei erhalten, in der die URIs aller für diesen Nutzer gespeicherten Fotos enthalten wären:

Listing 2.13: Liste von Fotos eines Benutzers

```
1 HTTP/1.1 200 OK
2
3 <?xml version="1.0" encoding="UTF-8" ?>
4 <photos user="Benutzer-ID">
5   <photo>http://api.photos.example.com/photo/123456789</photo>
6   <photo>http://api.photos.example.com/photo/987654321</photo>
7 </photos>
```

Mit diesen URIs können dann wiederum neue Anfragen, z. B. mit der `GET` Methode, zum Abruf einer Representation dieser Fotos, gestellt werden.

Dieses Beispiel ist natürlich nur eine stark vereinfachte Version der Funktionsweise und der Nutzung eines solchen Dienstes, sollte aber einen groben Überblick bieten, wie ein solches System letztendlich funktionieren würde.

3. Pflichtenheft

3.1. Zielbestimmung

Das im Folgenden spezifizierte Softwaresystem „Restful Legs“ ist ein Web Application Framework, das es dem Benutzer ermöglicht, eine REST API zu erstellen, bei der der Zugang zu den vom Benutzer erstellbaren Resources mit Hilfe von OAuth 1.0 kontrolliert werden kann.

3.1.1. Musskriterien

- REST – Das System unterstützt eine RESTful Architektur, wie beschrieben in Kapitel 6 von „Architectural Styles and the Design of Network-based Software Architectures“ [[Fielding, 2000](#)]
 - Es wird dem Benutzer ermöglicht, eigene Resources zu programmieren, die über RESTful HTTP-Requests abgerufen werden können.
- OAuth – Das System unterstützt OAuth in Version 1.0 nach [RFC 5849](#).
 - Anfragen an geschützte Resources werden auf eine gültige OAuth Signatur überprüft.
 - Beim Aufruf eines der speziellen OAuth Endpunkte werden die in der Protokollspezifikation erläuterten Schritte ausgeführt.
 - Es ist neuen OAuth Clients möglich, sich über ein Formular zu registrieren und dadurch Zugangsdaten zur API zu erhalten.

3.1.2. Wunschkriterien

- Installer
 - Nach dem Kopieren des Frameworks auf den Server erscheint beim erstem Aufruf im Browser eine Webseite, die durch den Installationsprozess leitet.
 - Es werden Verbindungsdaten der Datenbank abgefragt und getestet, ob eine Verbindung zustande kommt.

- Es gibt einen Übersetzungsmechanismus, mit dem an den Client des vom Nutzer erstellten Frameworks gerichtete Ausgaben in verschiedene Sprachen übersetzt werden können.
- Administrationsbereich
 - Es gibt einen Administrationsbereich, der über ein Webinterface erreichbar ist.
 - Es wird die Sperrung, Entsperrung, Löschung und Erstellung von OAuth Clients ermöglicht.
 - Alle erstellten Resources und ihre implementierten Methoden werden aufgelistet und es lassen sich die Restriktionslevel einstellen.

3.1.3. Abgrenzungskriterien

- Es wird keine API zum Speichern oder Nachschlagen von Daten bereitgestellt. Die Datenverwaltung der Resources muss vom Benutzer selbst geregelt werden.
- Die Erstellung von OAuth Resource Ownern muss vom Benutzer geregelt werden.
- Die OAuth Signierung wird mit Hilfe einer geeigneten Bibliothek vorgenommen und nicht eigenständig implementiert.

3.2. Produkteinsatz

3.2.1. Anwendungsbereiche

Das Framework wird im Bereich der Webentwicklung und API Erstellung zum Einsatz kommen.

3.2.2. Zielgruppen

Das Softwaresystem richtet sich an zwei verschiedene Typen von Webentwicklern. Als erstes an den Ersteller der eigentlichen REST API, im Folgenden als „Entwickler“ bezeichnet, und als zweites an die Benutzer der erstellten API, im Folgenden als „Klienten“ bezeichnet.

Entwickler

Für Entwickler wird die Beherrschung von objektorientiertem PHP auf einem fortgeschrittenem Niveau und ein Verständnis der Grundlagen von REST Systemen vorausgesetzt. Für eine Anpassung der an die Klienten gerichteten, vom System verwendeten Formulare wären auch Grundkenntnisse von HTML und CSS empfehlenswert. Diese sind jedoch nicht zwingend erforderlich.

Klienten

Für Klienten wird die Kenntnis einer beliebigen Programmiersprache, die Bibliotheken zum Versenden von HTTP Anfragen und OAuth Signierung bereitstellt oder damit erweiterbar ist, vorausgesetzt. Außerdem sollten Grundkenntnisse über die Verwendung eines RESTful Webservice vorhanden sein.

3.2.3. Betriebsbedingungen

Das Softwaresystem wird im Kontext eines Webserverns laufen und daher die üblichen Betriebsbedingungen von Webanwendungen haben:

- unbeaufsichtigt
- Betriebsdauer: 24/7 (ständige Verfügbarkeit)
- wartungsfrei bis auf etwaige Datenbanksicherungen

3.3. Produktfunktionen

/F10/ Eine neue REST Resource lässt sich erstellen, indem in einem festgelegtem Ressourcen Verzeichnis eine neue PHP Datei erstellt wird, die eine PHP Klasse des gleichen Namens enthält.

/F20/ Die Resources werden automatisch aus dem Resources Ordner ausgelesen und anhand der Verzeichnisstruktur auf erreichbare URIs abgebildet.

/F30/ Eine Resource kann beliebige HTTP-Methoden unterstützen.

/F40/ Ein Benutzer kann aus den Methoden seiner selbsterstellten Resources HTTP-Responses erstellen.

/F50/ Es gibt eine Möglichkeit für jede Resource und ihre unterstützten HTTP Methoden jeweils ein Restriktionslevel festzulegen. Es werden die Restriktionslevel „Public“, „Protected“, und „Private“ unterstützt.

- /F51/** Public – Die Resource ist öffentlich verfügbar, d. h. jede korrekte REST Anfrage erhält eine Antwort der Resource.
- /F52/** Protected – Die Resource ist zugänglich für jeden mit einer gültigen OAuth Signatur, d. h. jeder vorher registrierte Client bekommt eine Antwort auf eine korrekt gestellte REST Anfrage, die mit OAuth signiert wurde.
- /F53/** Private – Die Resource ist nur zugänglich, wenn eine signierte Anfrage inklusive eines gültigen OAuth Tokens, der vorher von einem Resource Owner genehmigt wurde, gesendet wird.
- /F60/** Das System kontrolliert eingehende Anfragen an Methoden der Restriktionslevel Protected und Private auf eine gültige OAuth Signatur
- /F70/** Tritt ein Fehler innerhalb des Frameworks auf wird ein laut HTTP-Spezifikation passender HTTP-Status als Antwort zurückgesendet, z. B. 404, für eine nicht gefundene Ressource, oder 405 für eine nicht unterstützte HTTP-Methode.
- /F80/** Es ist neuen OAuth Clients möglich sich über ein Formular mit Email, Name und Vorname zu registrieren, um OAuth Credentials zu erhalten.
- /F90/** Das System stellt die drei im OAuth Protokoll definierten Endpunkte zur Redirection-Based Authorization bereit.
- /F91/** Die URIs der OAuth Endpunkte sind anpassbar.
- /F110/** Für die OAuth Registrierung und Authorisierung werden optisch anpassbare Formulare bereitgestellt.

3.4. Produktdaten

Folgende Daten werden persistent gespeichert:

/D10/ OAuth Client Daten, ca. 2kb pro Datensatz:

- Client Key
- Client Secret
- Email
- Vorname
- Nachname

/D20/ OAuth Resource Owner Daten, ca. 1kb pro Datensatz:

- Benutzername

- Passwort (gehasht)
- Email

/D30/ OAuth Temporary Credentials, ca. 2kb pro Datensatz:

- Temporary Token
- Temporary Secret
- Callback
- Erstellungsdatum
- Verifier

/D40/ OAuth Token Credentials, ca. 1kb pro Datensatz:

- Token
- Secret
- Resource Owner
- OAuth Client
- Erstellungsdatum

Die Speicherung der Daten erfolgt in der unter 3.8.1 spezifizierten Datenbank. Eine genaue Voraussage des zu erwartenden Speicherbedarfs lässt sich nicht treffen, da dies von der Auslastung der vom Nutzer generierten API abhängt.

3.5. Produktleistungen

/L10/ Das Bearbeiten einer Anfrage nach einer privaten Resource dauert auf dem Server max. eine halbe Sekunde. Dies gilt allein für die vom Framework benötigte Zeit. Ausgenommen davon ist die vom Nutzer des Frameworks benötigte Zeit für die Erstellung der Representation der angeforderten Resource.

/L20/ Das Framework lässt sich auf einem gängigen XAMP Stack¹ innerhalb von zehn Minuten installieren.

¹Eine sehr weit verbreitete Server Konfiguration, die auf jeden Fall aus einem Apache Webserver, MySQL und PHP besteht. Oft in Form von LAMP, mit Linux oder WAMP mit Windows als Betriebssystem.

3.6. Qualitätsanforderungen

Der Code des Frameworks wird in folgendem Maße durch PHPUnit Tests abgedeckt:

- min. 85% der Codezeilen
- min. 75% aller Methoden
- min. 60% aller Klassen

Als Kriterien für die Messung werden die Werte aus dem, im von PHPUnit bereitgestellten, CodeCoverage Report herangezogen:

- <https://github.com/sebastianbergmann/php-code-coverage>

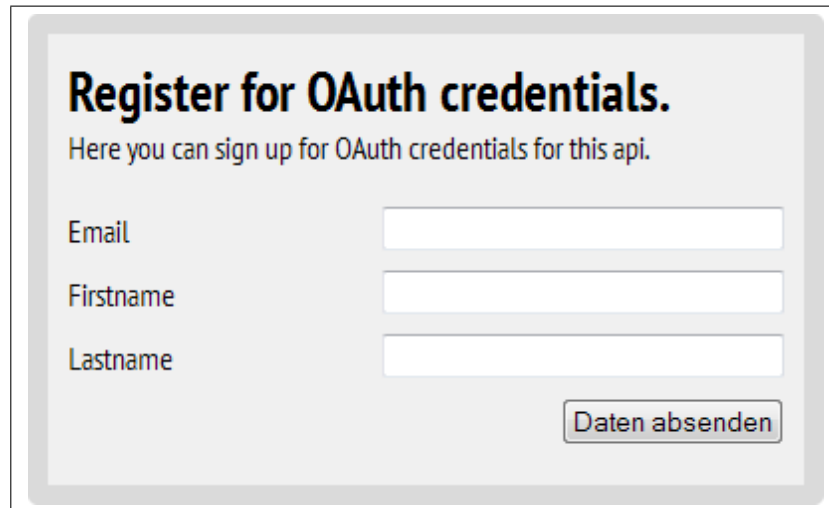
Außerdem werden die in der Tabelle dargestellten Kriterien bei der Entwicklung wie folgt gewichtet:

	sehr wichtig	wichtig	weniger wichtig	unwichtig
Sicherheit	X			
Effizienz		X		
Kompatibilität		X		
Erweiterbarkeit			X	
Benutzerfreundlichkeit	X			
Zuverlässigkeit		X		

3.7. Benutzungsoberfläche

Alle Benutzungsoberflächen des Frameworks sind zur Ansicht in einem Webbrowser vorgesehen.

Für die Registrierung und die Authorisierung von OAuth Clients werden die in den Abbildungen 3.1 und 3.2 zu sehenden Formulare standardmäßig bereitgestellt. Die endgültige Ausgestaltung ist dabei allerdings durch den Nutzer anpassbar.



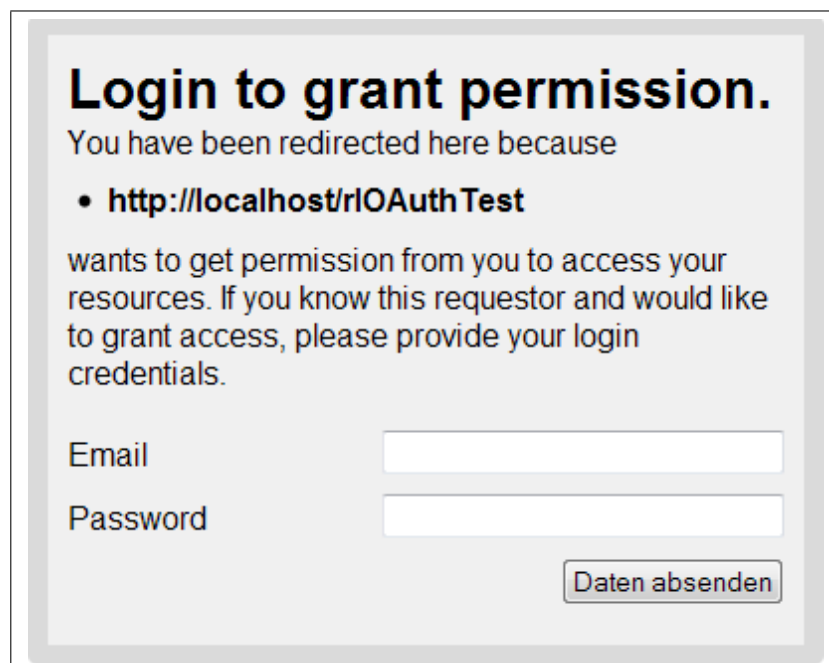
Register for OAuth credentials.
Here you can sign up for OAuth credentials for this api.

Email

Firstname

Lastname

Abbildung 3.1.: OAuth Registrierungsformular



Login to grant permission.
You have been redirected here because

- **<http://localhost/r/OAuthTest>**

wants to get permission from you to access your resources. If you know this requestor and would like to grant access, please provide your login credentials.

Email

Password

Abbildung 3.2.: OAuth Authorize Formular

3.8. Technische Produktumgebung

3.8.1. Software

Das Framework ist auf gängigen XAMP Servern lauffähig. Als Referenz werden folgende Versionen verwendet:

- Apache 2
- MySQL 5.1.41
- PHP 5.3.1

3.8.2. Hardware

Im Live-Betrieb wird ein Server mit folgenden Eigenschaften benötigt:

- Speicherplatz – Für die Installation des Frameworks werden min. 10MB Festplattenspeicher benötigt. Zusätzlich wird noch weiterer Speicher für die verwendete Datenbank und die erstellten Resources benötigt, der von der Klientenzahl der vom Nutzer erstellten API abhängt und sich nicht genau spezifizieren lässt.
- Rechenleistung – Die benötigte Rechenleistung ist abhängig von der Klientenanzahl.

Für die vorbereitende Entwicklung einer API wird ein üblicher Rechner benötigt, für den eine XAMP Umgebung zur Verfügung steht und lauffähig ist.

3.9. Entwicklungsumgebung

3.9.1. Software

Bei der Entwicklung wird folgende Software verwendet:

- Betriebssysteme: Windows 7, Ubuntu 8.04 LTS
- Netbeans 6.9.1
- ArgoUML 0.30.2
- XAMPP for Windows 1.7.3
- Git 1.7.3.1

3.9.2. Hardware

An die Hardware werden im wesentlichen keine stärkeren Anforderungen gestellt, als für die Nutzung des entwickelten Frameworks erforderlich sind. Im Speziellen kommen ein:

- Windows 7 Desktop Rechner für die Entwicklung
- und ein Linux Virtual Server zum Testen

zum Einsatz.

3.9.3. Orgware

Es wird ein englischsprachiges Benutzerhandbuch bereitgestellt, welches die Installation und Nutzung des Frameworks beschreibt.

4. Systementwurf und Implementierung

Wie bereits erwähnt erfolgt die Implementierung des zu erstellenden Frameworks nach einem testgetriebenen Verfahren, wie es zum Beispiel von Kent Beck in seinem Buch „Test-Driven Development By Example“ [Beck, 2002] vorgestellt wird.

Kurz zusammengefasst geht es darum, für eine zu programmierende Programmeinheit einen Unit Test vor der eigentlichen Implementierung zu schreiben und das durch den Test vorgegebene Problem zu lösen. Im Anschluss daran können dann, durch weitere Tests, das Programm erweitert, die Architektur verbessert und Codewiederholungen eliminiert werden.

Außerdem beinhaltet dieser Ansatz, dass zu Beginn kein vollständiges Klassendiagramm erstellt wird, wie dies in klassischen Entwurfsprozessen üblich ist, sondern, dass Entwurfsentscheidungen dann getroffen werden, wenn sie für die Implementierung benötigt werden. Aus diesem Grund wird der Ablauf von Systementwurf und Implementierung hier gemeinsam erläutert.

Der Ansatz einer testgetriebenen Entwicklung wird aus verschiedenen Gründen verfolgt:

1. Das Verfahren soll zu besonders robuster und fehlersicherer Software führen, mit nur einer geringen Erhöhung der Entwicklungszeit [Nagappan u. a., 2008, Kapitel 8].
2. Automatisierte Tests sind generell wünschenswert, um die „Flexibilität, Wartbarkeit und Wiederverwendbarkeit“ des Codes zu stärken, da keine „Angst vor Änderungen des Codes“ besteht [Martin, 2009, Seite 124].
3. Die Erstellung eines Webframeworks führt u. U. zu schlecht testbaren Funktionalitäten, da einige Schritte ansonsten nur auf einem Webserver getestet werden könnten. Dies soll möglichst auf ein Mindestmaß beim Integrationstest reduziert werden.
4. Die automatisierten Unit Tests können leicht durch Nutzer des Frameworks verwendet werden. Dies könnte vor allem bei Anpassungen an spezielle Anforderungen durch den Nutzer hilfreich sein, um zu garantieren, dass alles andere noch funktioniert wie gewünscht.
5. Für den Autor stellt diese Arbeit eine hervorragende Möglichkeit dar, diese En-

wicklungsweise einmal in einem größeren Rahmen zu testen.

6. Da die Arbeit am Framework nur durch eine Person erfolgt, sind weitere agile Prozesse oder ein Klassendiagramm zur Kommunikation der Entwurfsentscheidungen zwischen Teammitgliedern im Voraus nicht zwingend erforderlich.
7. Bei der Erstellung von Folgeversionen sollte eine stabile API¹ gewährleistet werden können, um nach Möglichkeit Updates des Frameworks ohne Anpassungen im vom Nutzer erstellten Code zu ermöglichen. Wenn bisherige Tests nach Änderungen noch funktionieren, stellt dies immerhin einen Indikator dar, dass dies der Fall ist.

Im Folgenden werden nun ausgewählte Abschnitte und Entscheidungen während des Entwicklungsprozesses erläutert und ein Überblick über die entstandene Architektur gegeben.

4.1. Die erste Planung

Als grundlegende Ideen standen am Anfang der Entwicklung die folgenden Punkte:

- Es soll einen zentralen Ordner innerhalb des Frameworks geben, in dem der Nutzer erstellte Resources in Form von PHP Klassen speichert, die dann von dort automatisch ausgelesen werden und anhand des Dateipfads, also der Unterordner des Verzeichnisses, auf URIs abgebildet werden.
- Es soll eine zentrale Konfigurationsdatei geben, in der alle vom Nutzer benötigten Informationen über die Datenbankverbindung oder ähnliches hinterlegt werden und Anpassungen der Konfiguration vorgenommen werden können. Dies auch in Hinblick auf einen späteren Installer, der diese Datei dann erstellen würde.

Außerdem konnte durch die Beschäftigung mit REST und OAuth der voraussichtliche Ablauf beim Eingang von Anfragen beim Server identifiziert werden und damit das in Abbildung 4.1 zu sehende Aktivitätsdiagramm der Antwortmöglichkeiten erstellt werden.

Der Startzustand stellt dabei immer das eingehende HTTP-Request und die Endzustände die HTTP-Responses dar. Der Kommentar gibt Aufschluss über den mit der Response zu sendenden HTTP-Status. Die Aktivitätszustände sind als angefragte Endpunkte/URIs zu interpretieren.

Innerhalb des Diagramms sind vier Spezialfälle zu erkennen, bei denen die Anfrage entweder an einen der drei OAuth Endpunkte oder an den nicht direkt im OAuth Proto-

¹Eine stabile API wird hier im Sinne von Jaroslav Tulachs Buch „Practical API Design“ als eine API gesehen, bei der sich Sichtbarkeiten, Benennung, Parameter, Rückgabewerte und Funktionsweise der vom Nutzer des Frameworks verwendeten Methoden nach Möglichkeit nicht verändern [Tulach, 2008, Seite 3 ff.].

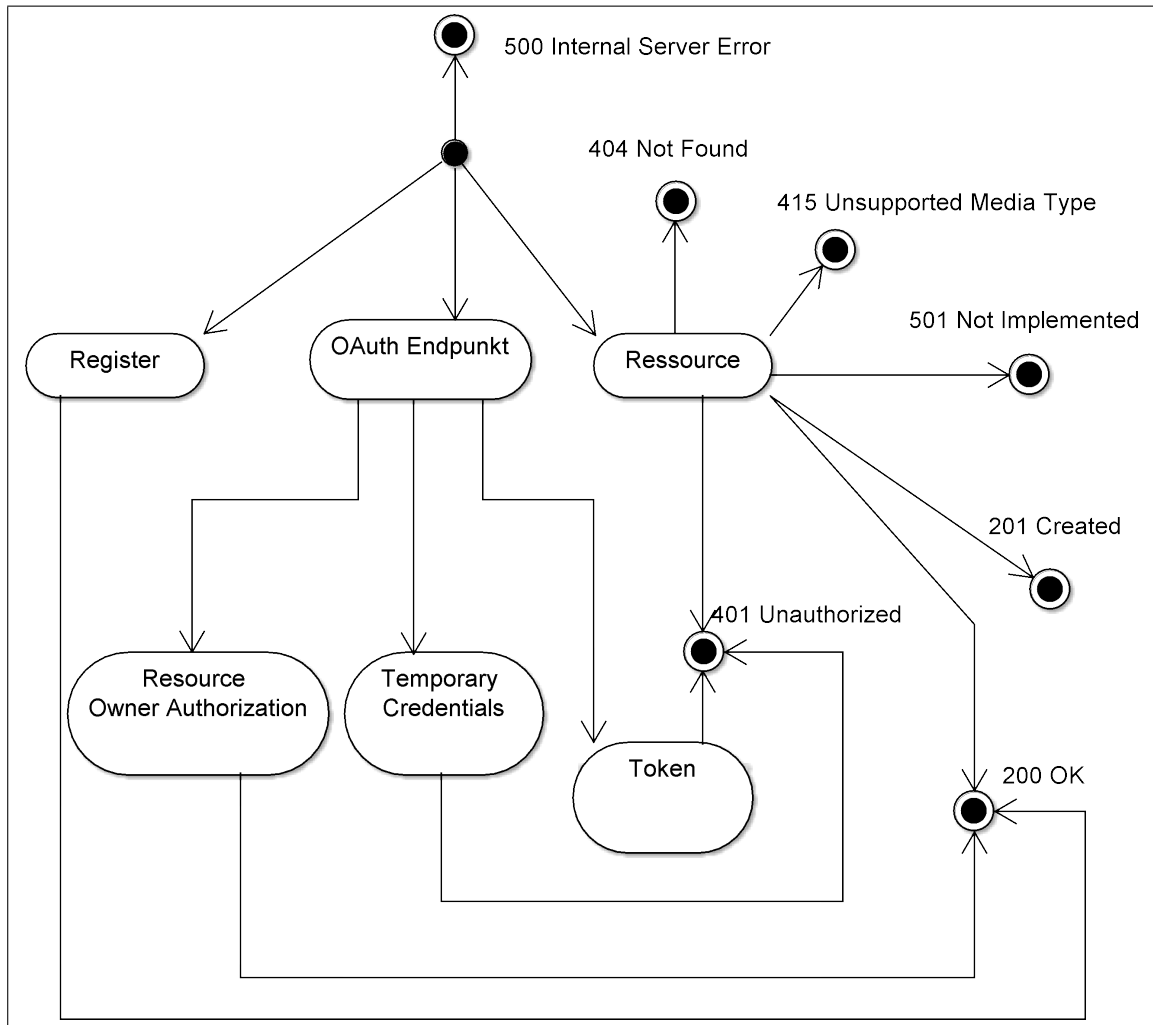


Abbildung 4.1.: Anfragebearbeitung

koll definierten Prozess zur Registrierung von OAuth Clients gerichtet ist. Diese müssen jeweils gesondert behandelt werden und unterschiedliche Antworten senden.

Der vermutlich häufigste und deswegen als Normalzustand anzusehende Fall ist die Anfrage an eine beliebige, erstellte Resource.

Dadurch wurde vor allem deutlich, dass alle Aktionen im Framework immer mit einem HTTP-Request beginnen und mit einer HTTP-Response enden. Repräsentationen dieser Konzepte eignen sich also auch innerhalb des Programmcodes gut als Kommunikationsmittel.

Auf Grund dieser Ideen konnten dann folgende voraussichtlich benötigten Elemente identifiziert werden:

- **Request** – Repräsentiert die Anfrage an den Server
- **Response** – Repräsentiert die Antwort des Server
- **HttpError** – Repräsentiert einen Fehlercode im Sinne des HTTP Protokolls
- **Dispatcher** – Hauptklasse, die alle anderen Aktionen auslöst
- **RequestParser** – Erstellt aus dem beim Server ankommenden HTTP Request ein Objekt der Klasse Request zur weiteren Verwendung innerhalb des Frameworks
- **OAuthFacade** – u. U. wird eine Klasse nach dem Facade Pattern benötigt, die Funktionen der verwendeten OAuth Bibliothek abstrahiert, damit eine spätere Änderbarkeit möglich wäre
- **IResource** – Interface für die vom Nutzer des Frameworks zu implementierenden Resources
- **AbstractResource** – Abstrakte Implementierung von **IResource**, die schon einige vermutlich benötigte Funktionen einer Resource zur Verfügung stellt
- **ResourceReader** – liest alle Resources aus dem Ordner aus, in dem der Nutzer seine Dateien erstellt
- **DBConnector** – Datenbankfunktionen
- **Installer** – unabhängig vom restlichen Framework, leitet durch den Installationsprozess
- **Representation** – Die Repräsentation einer Resource
- **RepresentationGenerator** – Vielleicht eine Factory, die Repräsentationen aus Resources erstellt
- **OAuthController** – Authentifizierungsprozesse

Durch diese Vorgehensweise ließen sich dann genug Ansatzpunkte für zu erstellende Module finden und mit der Implementierung beginnen.

Bevor dies erfolgen konnte, war allerdings noch der im nächsten Kapitel erläuterte Schritt einer üblichen PHP Technik nötig, um in einem generell objektorientierten Kontext zu arbeiten.

4.2. Bootstrapping mit mod_rewrite

Da das verfolgte Ziel ist, mit dem Framework auf beliebige, eingehende HTTP-Requests antworten zu können, ist es nötig diese an einer Stelle zu bündeln. Dort muss dann die

angeforderte URI analysiert werden, um eine geeignete Antwort zu senden. Um dies zu erreichen und auf Anfragen auch immer aus einem objektorientierten Kontext antworten zu können bedient man sich bei PHP einem Verfahren, welches häufig als „Bootstrapping“ [Minard, 2006] bezeichnet wird.

Der einfachste Fall einer Webanwendung ist der, dass HTTP-Anfragen an URIs wie:

- <http://www.example.org/index.html> oder
- <http://www.example.org/register.php>

direkt Dateien im Stammverzeichnis der Domain auf dem Server aufrufen.

Im Beispielfall also *index.html* und *register.php*. Da der gewünschte Effekt aber ist, dass auch auf Requests an unbekannte URIs, oder URIs mit variablen integrierten Parametern geantwortet werden kann, ist diese Vorgehensweise nicht ausreichend, da sonst unendlich viele PHP-Dateien erstellt werden müssten.

Um dies zu umgehen, verwendet man eine Rewrite-Engine, die es ermöglicht eingehende Requests umzuleiten und dadurch dann Zieldateien auf dem Server aufzurufen. Für den bei dieser Implementierung verwendeten Apache Server ist dafür das Modul `mod_rewrite` zuständig, welches Regeln in Form regulärer Ausdrücke ausliest, die in speziellen Apache Konfigurationsdateien namens *.htaccess* hinterlegt werden können.

Die Regeln, die in diesem Fall benötigt werden, sind dabei sehr einfach gehalten:

Listing 4.1: Restful Legs *.htaccess*

```
1 RewriteEngine on
2 RewriteRule !\.(js|ico|gif|jpg|png|css)$ index.php
```

Jegliche Anfrage, die nicht nach einer Javascript, CSS oder Bilddatei fragt, wird dadurch an *index.php* weitergeleitet. Die Aussparung einiger Dateitypen ist dabei vorhanden, um vor allem für die OAuth Formulare, bei denen es sich um einfachen HTML Code handelt, optische Anpassungen wie das Einbetten von Bildern zu ermöglichen.

Durch dieses Verfahren erreicht man, dass die Datei *index.php* analog zu einer *main* Methode in Java funktioniert. Sie wird also zum statischen Kontext, der die eigentliche Objektorientierung initialisiert:

Listing 4.2: Auszug *index.php*

```
1 if( isInstalled() )
2 {
3     require BASEPATH . 'RestfulLegs.php';
4
5     $restfulLegs = new RestfulLegs( 'rl-config.php' );
6     $restfulLegs->render();
7 }
```

4.3. Die entstandene Architektur

Die schließlich entstandene Architektur ähnelt zwar in vielen Teilen den unter 4.1 erläuterten aufgestellten Vermutungen. Durch ein ständiges Refactoring, welches durch die testgetriebene Entwicklung begünstigt wurde, sind aber auch an einigen Stellen Unterschiede entstanden oder neue Konzepte hinzugekommen.

Die wichtigste Gruppe von Klassen und Schnittstellen ist in Abbildung 4.2 zu erkennen. Darin sind alle Elemente abgebildet, mit denen der Benutzer des Frameworks bei der Nutzung in Kontakt kommen soll.

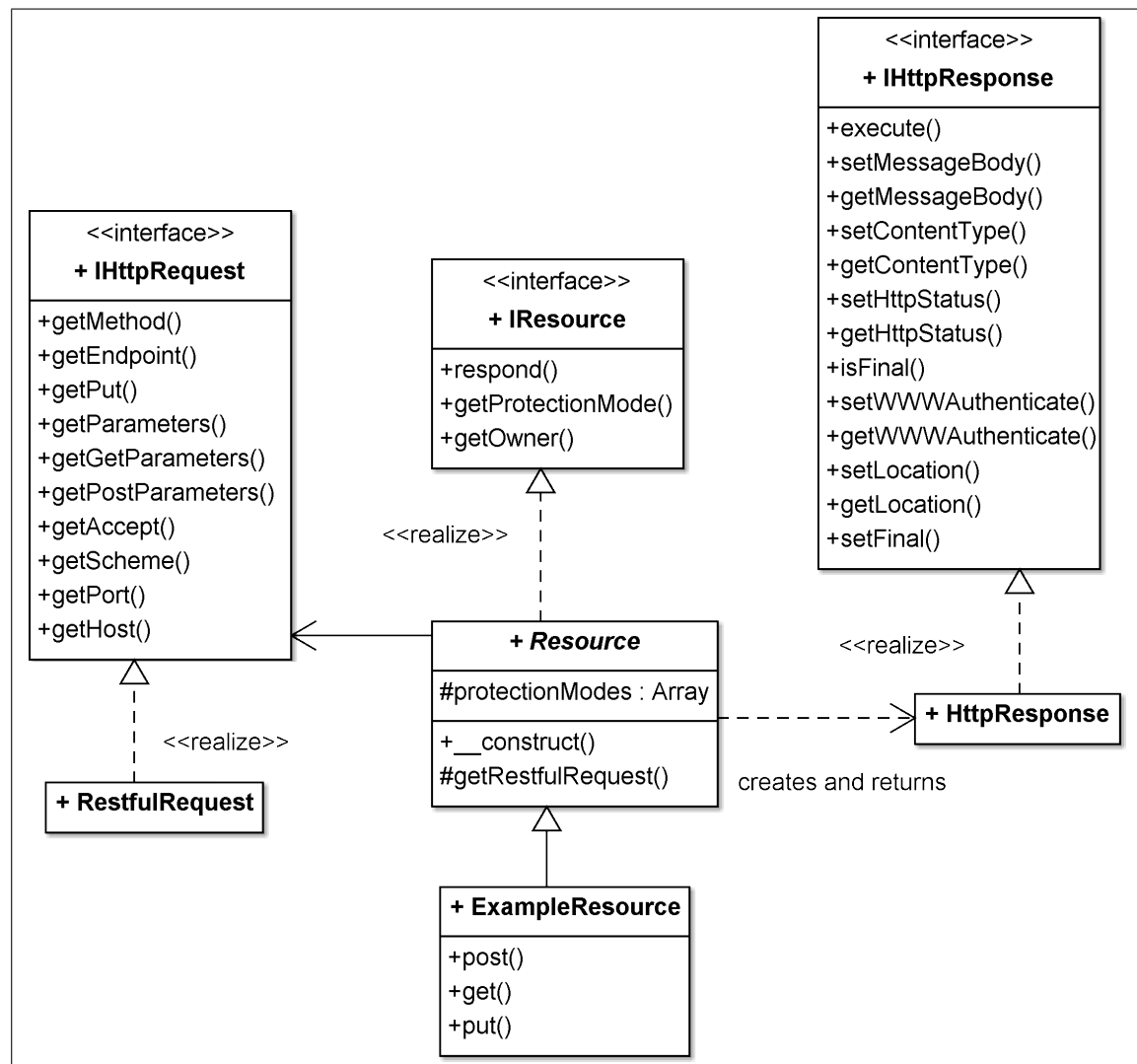


Abbildung 4.2.: Klassendiagramm Benutzerschnittstellen

auf. Aus ihnen wird ein Objekt des Typs `IHttpResponse` erwartet, welches dann vom Framework ausgeführt wird.

Beim restlichen System, wie es in leicht vereinfachter Form in 4.3 abgebildet ist, beginnt die Ausführung immer in der Klasse `RestfulLegs`, dem vormalig als „Dispatcher“ bezeichneten Konzept. Daraus werden alle anderen wichtigen Bausteine initialisiert und gegebenenfalls an die Controller weitergereicht. Dazu zählen z. B. die Herstellung einer Datenbankverbindung über `RestfulLegsMySQL` und das Erstellen des `RestfulRequest` mit Hilfe des `RequestParser`.

Der `ResponseController` ist im System die Verbindungsstelle zur oben erläuterten Benutzerschnittstelle, die - bevor die Anfrage an vom Benutzer erstellte Resources weitergeleitet wird - beim `EndpointController` klärt, ob überhaupt eine zu dieser Anfrage passende Resource existiert. Außerdem wird beim `AuthenticationController` die OAuth Signatur überprüft.

Ausgelassen wurden bei der hier genutzten Darstellung des Systems lediglich die Klassen, die zur Repräsentation der für OAuth benötigten Datenbanktabellen erforderlich sind.

5. Systemtest

Trotz des testgetriebenen Verfahrens steht am Ende der Implementierung natürlich noch ein Systemtest, bei dem die erstellten Module im zusammengesetzten Zustand überprüft werden.

Einige der im weiteren Verlauf spezifizierten Testfälle sind dabei auch schon in sehr ähnlicher Form durch die im Vorfeld erzeugten UnitTests abgedeckt, wodurch, soweit dies durch Tests möglich ist, eine gewisse Sicherheit in Bezug auf die ordnungsgemäße Funktionalität besteht. Der Vollständigkeit halber sollten sie aber auch hier noch einmal überprüft werden.

Um die Funktionalitäten des Gesamtsystems zu testen, ist die sinnvollste Vorgehensweise HTTP-Requests zu generieren und dann die Daten der gesendeten HTTP-Response auf erwartete Werte zu überprüfen. Dadurch lassen sich die Eingabe- und erwarteten Ausgabedaten relativ genau definieren, mit nur einigen Unwägbarkeiten in Bezug auf die verwendete Server- und Betriebssystemversion und damit verbundene spezielle Werte für einige HTTP-Header. Dies sollte allerdings keine Auswirkungen auf die eigentliche Funktionalität haben.

Für die Durchführung des Tests wird eine neue Installation des Frameworks vorausgesetzt, wie sie im Kapitel A.1 des Benutzerhandbuchs im Anhang erläutert wird.

Als Testumgebung wird „rlTest“, das in Abbildung 5.1 dargestellte Webformular, verwendet. Dabei handelt es sich um ein vom Framework unabhängiges PHP Skript, welches es ermöglicht, aus den entgegengenommenen Daten HTTP-Requests zu erstellen und diese an eine beliebige URI zu senden. Gegebenenfalls kann auch eine benötigte OAuth-Signatur erzeugt und hinzugefügt werden. Im Anschluss daran werden dann die zurückgesendeten Werte angezeigt und können überprüft werden.

Dieses Skript muss auf einem Webserver installiert werden, der die gleichen Systemvoraussetzungen wie der für das Framework verwendete, erfüllen muss. Es wird mit den Dateien des Frameworks mitgeliefert und kann deswegen auch sehr gut vom Nutzer des Frameworks zum Testen seiner selbsterstellten Resources genutzt werden, da die Vorgehensweise dazu identisch zu einigen der hier beschriebenen Testfälle ist.

Im Folgenden werden nun in Tabellenform Testfälle spezifiziert, mit denen die im Pflichtenheft definierten Funktionalitäten des Systems getestet werden können. Sollten keine genaueren Angaben unter „Testdaten/Testablauf“ gemacht werden, so besteht der Testablauf aus den folgenden Schritten:

1. Aufruf von rlTest in einem Browser.
2. Eingabe der unter „Testdaten“ des Testfalls bereitgestellten Eingabewerte an der durch die Nummerierung in der Abbildung 5.1 gekennzeichneten Stelle.

RESTful OAuth Test

By inputting the following data you can make a call to a RESTful, OAuth secured, resource.

REST Resource Data

URI:

1

Method:

2

GET

Accept:

3

Message Body:

4

OAuth Data

OAuth Client Key:

5

OAuth Client Secret:

6

OAuth Callback:

7

OAuth Token:

8

OAuth Token Secret:

9

OAuth Verifier:

10

Daten absenden

Request Result:

Request URI:

11

Response Header:

12

Message Body:

13

Abbildung 5.1.: Testumgebung rlTest

3. Absenden des Formulars
4. Vergleich der Daten, die an den mit 12 und 13 markierten Stellen erscheinen, mit den in „erwartetes Resultat“ dargestellten Rückgabewerten.

Die Rückgabewerte sind dabei nicht vollständig dargestellt, sondern müssen häufig nur in den erhaltenen Daten enthalten sein, da es z. B. oft ausreicht, die erste Zeile des Response Header auf einen erwarteten HTTP-Status zu überprüfen.

Auch wenn eine vollständige Unabhängigkeit von Tests generell wünschenswert ist, so gibt es doch Testfälle, die aufeinander aufbauen. Dies liegt einerseits daran, dass dadurch die Abläufe zum Aufruf einer Resource so nachgebildet werden können, wie sie auch durch einen Endnutzer erfolgen würden und andererseits daran, dass eine Bereitstellung einer Testdatenbank unnötig wird, die unter Umständen ein Sicherheitsrisiko bedeuten könnte. Vor allem im angestrebten Open Source Kontext müsste diese öffentlich bereitgestellt werden, wodurch diese Daten doch einmal versehentlich in einer „Live-Umgebung“ verwendet oder vergessen werden könnten.

5.1. Testfälle

Für variable Elemente werden folgende Platzhalter innerhalb der Testfälle verwendet:

%BASEURI% – Die Pfadbestandteile, über die das Framework zu erreichen ist. Bei einem lokalen Test z. B. *localhost/RestfulLegs*, wenn die gesamte URI <https://localhost/RestfulLegs/initiate> lauten würde.

%TESTURI% – Die URI des Testskripts.

Einige Inhalte der Platzhalter werden erst in den unter „Abhängigkeiten“ jeweils benannten Testfällen generiert, zu diesen zählen:

%CLIENT_KEY% und %CLIENT_SECRET% Ein zusammengehörender Satz Client Credentials.

%TEMPORARY_TOKEN_KEY% und %TEMPORARY_TOKEN_SECRET% Ein zusammengehörender Satz Temporary Credentials.

%VERIFIER% Ein OAuthVerifier.

%TOKEN_KEY% und %TOKEN_SECRET% Ein zusammengehörender Satz Token Credentials.

1.	Testfall	Aufruf einer nicht existierenden Resource
	Testdaten/Testablauf	1: <code>http://%BASEURI%/NonExistentResource</code> 2: GET
	erwartetes Resultat	12: 404 Not Found
	Abhängigkeiten	keine
2.	Testfall	Aufruf einer nicht implementierten Methode einer Resource
	Testdaten/Testablauf	1: <code>http://%BASEURI%/ExampleResource</code> 2: PUT
	erwartetes Resultat	12: 501 Not Implemented
	Abhängigkeiten	keine
3.	Testfall	Aufruf einer <i>PROTECTION_MODE_PUBLIC</i> Methode einer Resource
	Testdaten/Testablauf	1: <code>http://%BASEURI%/ExampleResource</code> 2: GET
	erwartetes Resultat	200 OK
	Abhängigkeiten	keine
4.	Testfall	nicht signierte Anfrage an eine <i>PROTECTION_MODE_PROTECTED</i> Methode einer Resource
	Testdaten/Testablauf	1: <code>http://%BASEURI%/ExampleResource</code> 2: POST
	erwartetes Resultat	401 Unauthorized
	Abhängigkeiten	keine
5.	Testfall	nicht signierte Anfrage an eine <i>PROTECTION_MODE_PRIVATE</i> Methode einer Resource
	Testdaten/Testablauf	1: <code>http://%BASEURI%/ExampleResource</code> 2: DELETE
	erwartetes Resultat	401 Unauthorized
	Abhängigkeiten	keine
6.	Testfall	Aufruf der Registrierung für OAuth Clients ohne HTTPS
	Testdaten/Testablauf	1: <code>http://%BASEURI%/register</code>
	erwartetes Resultat	12: 403 Forbidden
	Abhängigkeiten	keine
7.	Testfall	Aufruf des „token“ Endpunkts ohne HTTPS
	Testdaten/Testablauf	1: <code>http://%BASEURI%/token</code>
	erwartetes Resultat	12: 403 Forbidden
	Abhängigkeiten	keine

8.	Testfall	Aufruf des „initiate“ Endpunkts ohne HTTPS
	Testdaten/Testablauf	1: <code>http://%BASEURI%/initiate</code>
	erwartetes Resultat	12: 403 Forbidden
	Abhängigkeiten	keine
9.	Testfall	Erstellung eines OAuth Clients
	Testdaten/Testablauf	Direkter Aufruf von: <code>https://%BASEURI%/register</code> im Browser Eingabe von Emailadresse, Vor- und Nachname
	erwartetes Resultat	Webseite mit %CLIENT_KEY% und %CLIENT_SECRET%
	Abhängigkeiten	keine
10.	Testfall	signierte Anfrage an eine <i>PROTECTION_MODE_PROTECTED</i> Methode einer Resource
	Testdaten/Testablauf	1: <code>http://%BASEURI%/ExampleResource</code> 2: POST 5: %CLIENT_KEY% 6: %CLIENT_SECRET%
	erwartetes Resultat	12: 200 OK
	Abhängigkeiten	9
11.	Testfall	signierte Anfrage an eine <i>PROTECTION_MODE_PRIVATE</i> Methode einer Resource ohne Token
	Testdaten/Testablauf	1: <code>http://%BASEURI%/ExampleResource</code> 2: DELETE 5: %CLIENT_KEY% 6: %CLIENT_SECRET%
	erwartetes Resultat	12: 401 Unauthorized
	Abhängigkeiten	9
12.	Testfall	Initiate Endpunkt signiert ohne Callback
	Testdaten/Testablauf	1: <code>https://%BASEURI%/initiate</code> 2: GET 5: %CLIENT_KEY% 6: %CLIENT_SECRET%
	erwartetes Resultat	12: 400 Bad Request
	Abhängigkeiten	9

13.	Testfall	Initiate Endpunkt signiert
	Testdaten/Testablauf	1: https://%BASEURI%/initiate 2: GET 5: %CLIENT_KEY% 6: %CLIENT_SECRET% 7: %TESTURI%
	erwartetes Resultat	12: 200 OK 13: %TEMPORARY_TOKEN_KEY% und %TEMPORARY_TOKEN_SECRET%
	Abhängigkeiten	9

14.	Testfall	Authorize Endpunkt
	Testdaten/Testablauf	1: http://%BASEURI%/authorize 2: GET 8: %TEMPORARY_TOKEN_KEY% 9: %TEMPORARY_TOKEN_SECRET% Zusätzlich Login in das angezeigte Formular. Username: <i>testowner</i> Password: <i>password</i>
	erwartetes Resultat	12: 200 OK 13: %VERIFIER%
	Abhängigkeiten	9, 13

15.	Testfall	Token Endpunkt signiert mit Verifier
	Testdaten/Testablauf	1: https://%BASEURI%/token 2: GET 5: %CLIENT_KEY% 6: %CLIENT_SECRET% 8: %TEMPORARY_TOKEN_KEY% 9: %TEMPORARY_TOKEN_SECRET% 10: %VERIFIER%
	erwartetes Resultat	12: 200 OK 13: %TOKEN_KEY% und %TOKEN_SECRET%
	Abhängigkeiten	9, 14

16.	Testfall	signierter Aufruf des Token Endpunkt ohne Temporary Credentials
	Testdaten/Testablauf	1: https://%BASEURI%/token 2: GET 5: %CLIENT_KEY% 6: %CLIENT_SECRET%
	erwartetes Resultat	12: 401 Unauthorized
	Abhängigkeiten	9

17.

Testfall	signierte Anfrage an eine <i>PROTECTION_MODE_PRIVATE</i> Methode einer Resource mit Token
Testdaten/Testablauf	1: http://%BASEURI%/ExampleResource 2: DELETE 5: %CLIENT_KEY% 6: %CLIENT_SECRET% 8: %TOKEN_KEY% 9: %TOKEN_SECRET%
erwartetes Resultat	12: 200 OK
Abhängigkeiten	9, 15

6. Schlussbetrachtung

Ziel der vorliegenden Arbeit war es, ein Softwaresystem zu erstellen und zu beschreiben, welches die Technologien OAuth und REST verbindet. Neben diesen Hauptpunkten sollte außerdem noch ein testgetriebenes Verfahren bei der Entwicklung zur Anwendung kommen, um diese Entwicklungsweise einmal in einem größeren Rahmen zu testen.

Dass eine Bachelorarbeit bei diesen vielen neuen Konzepten und Verfahren dabei immer auch ein großer Lernprozess ist, vor allem, da sie im Studiengang Medieninformatik die erste große, alleine zu bewältigende schriftliche Arbeit darstellt, war daher im Voraus abzusehen. Trotz dieser Faktoren funktionierten die Zeitplanung und Bearbeitung der gestellten Aufgaben relativ genau und gut.

Die größten Schwierigkeiten innerhalb des Softwareanteils dieser Arbeit stellten sich durch OAuth und die ungewohnte testgetriebene Entwicklung.

Innerhalb von OAuth waren vor allem die zur Verfügung stehenden Bibliotheken zur Kontrolle der Signaturen und die hohe Komplexität der Redirection-Based Authorization problematisch.

Um hier nur ein kurzes Beispiel zu geben: So wirft die schließlich verwendete OAuth-Bibliothek in vielen Situationen Exceptions und fängt diese nicht selbst, sondern überlässt diese Aufgabe ihrem Benutzer. Auch wenn Exceptions in vielen Fällen ein sinnvolles Konzept sind, so sollte man doch meinen, dass bei einer Funktion mit dem Namen `check_signature` ein boolescher Wert zurückgegeben werden könnte und nicht eine Exception im Fall einer fehlerhaften und gar nichts bei einer richtigen Signatur.

Bei der testgetriebenen Entwicklung wurden die Probleme meistens durch Abhängigkeiten der Module und Faktoren wie die in einigen Fällen benötigte Datenbankverbindung hervorgerufen. Glücklicherweise können solche Problematiken in den meisten Fällen durch die Verwendung von Pattern, wie z. B. *Dependency Injection*, vermieden werden.

Der schriftliche Teil stellte eine Herausforderung dar, weil es dabei das erste Mal nötig war, in einer wirklich wissenschaftlichen Arbeitsweise, mit Zitaten und Literaturverzeichnis vorzugehen, die in dieser Form in den bisherigen Aufgaben während des Studiums eher selten vorkam. Außerdem dauern natürlich Formulierungen und Formatierungen für jemanden, der nicht häufig Texte in dieser Größenordnung verfasst, auch eher länger.

Das mit dieser Arbeit erstellte Framework ist ein voll verwendbares Softwaresystem, trotzdem gibt es noch genügend Punkte, die sich für eine Erweiterung der Features und

auch für eine Verbesserung der Architektur anbieten würden. Da als Ziel des Softwareanteils auch eine Veröffentlichung unter einer Open Source Lizenz vorgegeben wurde, wäre es schön noch weitere Funktionalitäten hinzuzufügen, um die Verwendbarkeit und Akzeptanz zu verbessern und dann vielleicht tatsächlich diesen Schritt zu gehen.

Bei den bereits im Pflichtenheft aufgezählten Wunschkriterien wie Administrationsbereich und Installationsskript war im Prinzip schon vor der Implementierung klar, dass diese nicht während, sondern eher nach dieser Bachelorarbeit erfüllt werden würden. Da diese Ideen aber schon im Vorfeld bestanden, erschien es sinnvoll, sie trotzdem mit aufzunehmen, um einen Ausblick auf weitere Planungen zu geben. Teilweise sind dadurch auch schon Vorarbeiten geleistet oder Ansatzpunkte während der Implementierung bedacht worden.

Abschließend kann festgestellt werden, dass der Bearbeitungszeitraum eine sehr lehrreiche Zeit war, in der viele Erfahrungen gesammelt werden konnten, die sich hoffentlich auch in der Bearbeitung zukünftiger Projekte und Aufgaben widerspiegeln werden.

A. User's Manual

A.1. Installation

For installing the „RESTful Legs“ Framework, all you need is a webserver running MySQL and PHP 5. If you want to use the redirection-based authorization process from OAuth it should also be able to answer to HTTPS requests.

If those requirements are met, you can go ahead with the following steps involved in the installation process:

1. Copy the framework files found in the `/restfulLegs` folder into the webroot or a subfolder of the webroot on your server.
2. Set up a MySQL database and run the SQL script found in `/installer/install.sql`, to create the necessary tables.
3. Open the `rl_config_example.php` file from the base folder of the framework and fill in the database connection details for your newly created database:
 - `$dbName`
 - `$dbUser`
 - `$dbPassword`
 - `$dbHost`
4. Save the config file as `rl_config.php` into the base folder of the framework.

After this you should be able to see the OAuth register form, the owner authorization form and an example resource, by opening your browser and pointing it to the base URI of the framework and appending `register`, `authorize` or `ExampleResource` to it, i. e. :

- `http://example.com/your/sub/folders/register`
- `http://example.com/your/sub/folders/authorize`
- `http://example.com/your/sub/folders/ExampleResource`

A.2. Usage

The basis for the usage of this framework, as it is for REST generally, is the notion of a *resource*. For creating such resources, you will need to create `.php` files in the `\resources` folder of the framework, containing a class of the exact same name as the file and extending the class `Resource`.

Listing A.1: File `MyResource.php`

```
1 class MyResource extends Resource
2 {
3
4 }
```

After creating your own resource class, you should be able to type in a URI like this:

- `http://example.com/your/sub/folders/MyResource`

into your browser and see a notification, that you are not authorized to access the requested resource. This is because the default protection of the resources is very restrictive. As you will learn, however, it is not that difficult to change.

You can also create subfolders inside the `\resources` folder, that will also get integrated into the URI of your resource.

A.2.1. Implementing Resources

Your created class needs to provide a function for every HTTP method it wants to support and a `$protectionMode` array containing a protection mode for each of those methods. The functions also need to return an object of type `IHttpResponse`.

To make things a bit more understandable, here is an example of a resource allowing public access via its GET method:

Listing A.2: Implementing GET

```
1 class MyResource extends Resource
2 {
3     protected $protectionModes = array( 'get' => self::
4         PROTECTION_MODE_PUBLIC );
5
6     protected function get()
7     {
8         $response = new HttpResponse();
9
10        //200 means everything OK
```

```

10     $response->setHttpStatus(new HttpStatus(200));
11     $response->setMessageBody( 'The content' );
12
13     return $response;
14 }
15 }

```

If you update your own class with this code and reload the page, you should be able to see the string „The content“ in your browser.

The response should certainly contain some more meaningful data than „The content“, to make your own API valuable for your users, but this should give you an idea, of how to get some content to your requestor.

There are four HTTP methods, which you are most likely to use, those are GET, POST, PUT and DELETE. The matching functions for those would be `get()`, `post()`, `put()` and `delete()`, but it's also possible to support custom HTTP methods, by changing adding those to the `$allowedHttpMethods` array in your `rl_config.php`.

To gain more information about the called data, it's possible to look into the data that was sent with the HTTP request. For example, this could be useful to check, which content type the requestor would like to get in return to his request:

Listing A.3: Content type negotiation

```

1 protected function get()
2 {
3     $contentType = $this->getRestfulRequest()->getAccept();
4
5     $response = new HttpResponse();
6
7     if($contentType=='text/html')
8     {
9         $response->setMessageBody( '<p>Some html content</p>' );
10        $response->setContentType($contentType);
11        $response->setHttpStatus(new HttpStatus(200));
12    }
13    else
14    {
15        $response->setHttpStatus(new HttpStatus(406));
16        $response->setMessageBody( 'The content type you requested
17                                   is not supported.' );
18    }
19    return $response;
20 }

```

Another possibility is to look at the parameters that are sent with the request. If you created a resource named `MyResource` the request also contains all additional parts of the called URI, which couldn't be matched to your resource. For example, a URI like:

- `http://example.com/your/sub/folders/MyResource/123456789/`

would contain the parameter `123456789`. Here's a little example of how to use those parameters:

Listing A.4: Request parameters label

```
1 protected function post()  
2 {  
3     $uriParameters = $this->getRestfulRequest()->getParameters();  
4  
5     $response = new HttpResponse();  
6     $response->setHttpStatus(new HttpStatus(200);  
7     //the getContentForId() function is made up, it's up to you,  
        to know what to do with the sent parameters  
8     $response->setMessageBody($this->getContentForId(  
        $uriParameters[0]));  
9  
10    return $response;  
11 }
```

You should look into the phpdoc of `RestfulRequest` found under `\doc\index.html` to see what else is possible.

A.2.2. Protecting Resources

The framework uses OAuth 1.0 for authentication and authorization purposes. It provides three different protection modes, which can be specified separately for every method of a resource by specifying the field `$protectionModes` as seen in A.2.1. You have the choice between the PHP constants:

PROTECTION_MODE_PUBLIC

Everybody can use this method.

PROTECTION_MODE_PROTECTED

Every valid OAuth consumer can use this method

PROTECTION_MODE_PRIVATE

Resource owner authorization is needed to use this method.

The public mode should be clear, so we will go on to check out how protected resources are accessed. The first step would be for your client (or yourself, for testing purposes) to

sign up for some OAuth credentials. By default this is done by calling the frameworks base URI appended with `/register`. After receiving those credentials the HTTP request to your resource just has to be signed, like described in the OAuth protocol with the parameters given to you.

The signing process is a bit beyond the scope of this guide, because it's more of a problem on the client side. You can find some descriptions and libraries for the programming language of your choice on the page of the OAuth project: <http://oauth.net/> and will also learn how to test your created resources without knowledge about the signature process in A.2.3.

Private Mode

If you want to create methods with `PROTECTION_MODE_PRIVATE`, things get slightly more complicated, because your class also needs to provide the knowledge about the owner of the resource. This is necessary because OAuth has a process called *Resource Owner Authorization*, in which an owner of a resource authorizes someone who has obtained OAuth credentials for your API to access the resource he owns.

To provide the information about the owner, you need to overwrite the `getOwner()` function in your resource and provide a string from it which identifies the owner of the currently requested resource.

The string must be a username which was entered before into the `rl_oauth_owner` table in your database. Creating those owners is a process that is up to you. The most important thing is that you need to hash your password with the `sha1()` function of php.

The most likely case to obtain knowledge about the owner of the requested resource is by looking at the parameters that were provided with the request. With most APIs, there will be some kind of ID sent with the requests, if the resource belongs to someone.

Here's an example of a class and the PUT method for this protection mode:

Listing A.5: `PROTECTION_MODE_PRIVATE` class

```
1 class MyPrivateResource extends Resource
2 {
3     protected $protectionModes = array( 'put' => self::
        PROTECTION_MODE_PRIVATE );
4
5     protected function put()
6     {
7         $response = new HttpResponse();
8         $uriParameters = $this->getRestfulRequest()->getParameters
            ();
```



```

9
10     //200 means everything OK
11     $response->setHttpStatus(new HttpStatus(200));
12     $response->setMessageBody('You requested data for ' .
        $uriParameters[0]);
13
14     return $response;
15 }
16
17 protected function getOwner()
18 {
19     $uriParameters = $this->getRestfulRequest()->getParameters
        ();
20     $owner = '';
21
22     //isValidId() is made up, you need to make the criterias
        about a valid user id for this resource yourself
23     if($this->isValidId($uriParameters[0]))
24     {
25         $owner = $uriParameters[0];
26     }
27
28     return $owner;
29 }
30 }

```

The framework does the job of checking whether the owner you specified, allowed the current requestor to access the resource via the resource owner authorization steps, described in the OAuth protocol.

A.2.3. Testing Resources

To test your resources you need a way to make HTTP requests to your server at least with the standard methods and most probably also signed with OAuth credentials.

As you may have noticed by now, the easiest HTTP method to test is GET, because it is the default way a browser retrieves content for a URI you enter into the adress field. After implementing the GET method for a resource, the only thing you have to do is enter the appropriate URI into your browser and make the method publicly available.

This is not always possible, mainly in a live context, but to your rescue there is a script provide with the framework which you can use to make authenticated request to an URI of your choice within the `/r1Test` folder.

By putting this folder somewhere onto your webserver and calling it in your browser, you are enabled to make HTTP requests with methods defined in the HTTP specification. Additionally you also can provide your OAuth credentials, which the script will use to sign your requests.

Listings

2.1. HTTP Request	7
2.2. HTTP Response	7
2.3. SOAP HTTP Request	8
2.4. Header Feld	10
2.5. OAuth Credentials im Authorization Header	13
2.6. OAuth Credentials im Query String einer URI	14
2.7. Beispiel Flickr Api Credentials	14
2.8. Beispiel eines Signature Base String aus der OAuth Spezifikation[Hammer-Lahav, 2010 , Kapitel 3.4.1.1]	15
2.9. Upload eines Fotos	20
2.10. RESTful upload eines Fotos	21
2.11. Löschen eines Fotos	21
2.12. Anforderung einer Liste von Fotos eines Benutzers	21
2.13. Liste von Fotos eines Benutzers	22
4.1. Restful Legs .htaccess	36
4.2. Auszug index.php	36
A.1. File MyResource.php	50
A.2. Implementing GET	50
A.3. Content type negotiation	51
A.4. Request parameters label	52
A.5. PROTECTION_MODE_PRIVATE class	53

Abbildungsverzeichnis

2.1. Loginweitergabe	12
2.2. OAuth Sequenzdiagramm	17
3.1. OAuth Registrierungsformular	29
3.2. OAuth Authorize Formular	29
4.1. Anfragebearbeitung	34
4.2. Klassendiagramm Benutzerschnittstellen	37
4.3. Klassendiagramm Restful Legs	38
5.1. Testumgebung rlTest	41

Literaturverzeichnis

- [Allamaraju 2010] ALLAMARAJU, Subbu: *RESTful Web Services Cookbook*. O'Reilly, 2010
- [Beck 2002] BECK, Kent: *Test Driven Development By Example*. Addison Wesley, 2002
- [Fielding 2000] FIELDING, Roy: *Architectural Styles and the Design of Network-based Software Architectures*. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. 2000. – [Online; accessed 23-November-2010]
- [Florencio und Herley 2007] FLORENCIO, Dinei ; HERLEY, Cormac: *A Large-Scale Study of Web Password Habits*. <http://research.microsoft.com/en-us/um/people/cormac/Papers/www2007.pdf>. 2007. – [Online; accessed 23-Januar-2011]
- [Google kein Datum] GOOGLE: *2-legged OAuth for the OpenSocial REST API*. <http://sites.google.com/site/oauthgoog/2leggedoauth/2opensocialrestapi>. kein Datum. – [Online; accessed 26-Januar-2011]
- [Hafner und Lyon 1998] HAFNER, Katie ; LYON, Matthew: *Where wizards stay up late - The origins of the internet*. Touchstone, 1998
- [Hammer-Lahav 2010] HAMMER-LAHAV, E.: *The OAuth 1.0 Protocol*. <http://tools.ietf.org/html/rfc5849>. 2010. – [Online; accessed 30-November-2010]
- [Hammer-Lahav 2007] HAMMER-LAHAV, Eran: *Explaining OAuth*. <http://oauth.net/about/>. 2007. – [Online; accessed 22-Januar-2011]
- [Heller 2007] HELLER, Martin: *REST and CRUD: the Impedance Mismatch*. <http://www.infoworld.com/d/developer-world/rest-and-crud-impedance-mismatch-927>. 2007. – [Online; accessed 20-Januar-2011]
- [IETF 2007] IETF: *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. <http://tools.ietf.org/html/rfc4918>. 2007. – [Online; accessed 21-Januar-2011]
- [IETF 2010] IETF: *The OAuth 2.0 Protocol Framework draft-ietf-oauth-v2-11*. <http://tools.ietf.org/html/draft-ietf-oauth-v2-11>. 2010. – [Online; accessed 20-Januar-2011]

- [ISOC kein Datum] ISOC: *About The Internet Society (ISOC)*. <http://www.isoc.org/isoc/>. kein Datum. – [Online; accessed 23-Januar-2011]
- [Martin 2009] MARTIN, Robert C.: *Clean Code A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009
- [Minard 2006] MINARD, Jayson: *Blueprint for PHP Applications: Bootstrapping (part 1)*. <http://devzone.zend.com/article/70>. 2006. – [Online; accessed 04-Januar-2011]
- [Nagappan u. a. 2008] NAGAPPAN, Nachiappan ; MAXIMILIEN, E. M. ; BHAT, Thirumalesh ; WILLIAMS, Laurie: *Realizing quality improvement through test driven development: results and experiences of four industrial teams*. http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf. 2008. – [Online; accessed 04-Februar-2011]
- [RFC-Editor kein Datum] RFC-EDITOR: *RFC Index*. <http://www.rfc-editor.org/rfc-index2.html>. kein Datum. – [Online; accessed 20-Januar-2011]
- [Richardson und Ruby 2007] RICHARDSON, Leonard ; RUBY, Sam: *RESTful Web Services*. O'Reilly, 2007
- [Roy Fielding u. a. 1999] ROY FIELDING, Tim Berners-Lee u. a.: *Hypertext Transfer Protocol – HTTP/1.1*. <http://tools.ietf.org/html/rfc2616>. 1999. – [Online; accessed 10-Januar-2011]
- [Tulach 2008] TULACH, Jaroslav: *Practical API Design: Confessions of a Java Framework Architect*. Apress, 2008
- [Wikipedia 2010] WIKIPEDIA: *Representational State Transfer* — *Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Representational_State_Transfer&oldid=397043041. 2010. – [Online; accessed 22-November-2010]

Index

API, 1, 2, 30
API-Key, 2
ArgoUML, 30
ARPANET, 5
Authorization Header, 13

Benutzerhandbuch, 31

Convention over Configuration, 3

DELETE, 2, 9, 21, 51
Dependency Injection, 47

Facebook, 1
FTP, 5

GET, 2, 8, 50, 51
Graph API, 2

HEAD, 9, 10
HMAC-SHA1, 15
HTTP, 1–3, 5
HTTP 1.1, 5
HTTP-Methoden, 9
HTTP-Request, 7, 8, 21, 40
HTTP-Response, 7, 40
HTTP-Status, 7, 10, 42
HTTPS, 16, 49

Internet Society, 5
IPv6, 5

last.fm, 1
Location, 21

Message Body, 14
MySQL, 49
Netbeans, 30

OAuth, 2, 3, 5, 6, 49
OAuth 1.0, 5
OAuth Dance, 16
OAuth Endpoints, 16
oauth_token, 18
Opensocial, 2
OPTIONS, 9

Percent Encoding, 15
PHP, 3
Plaintext, 15
POP3, 5
POST, 2, 7, 8, 11, 20, 51
PROTECTION_MODE_PRIVATE, 43, 52
PROTECTION_MODE_PROTECTED,
43, 52
PROTECTION_MODE_PUBLIC, 43, 52
PUT, 8, 51

Query String, 14, 18

Reason Phrase, 10
Redirection Based Authorization, 15
Redirection-Based Authorization, 16, 49
Representation, 6
Request for Comments, 5
Resource, 2, 3, 6, 7, 11, 21, 40, 43, 50, 52
Resource Identifier, 6
Resource Owner, 18
Resource Owner Authorization, 16, 18
REST, 1–3, 6, 8, 9, 50
RESTful, 1
RESTful Legs, 49
RFC, 5, 6
RFC 1939, 5
RFC 2460, 5
RFC 2616, 5

- RFC 5849, 5
- RFC 959, 5
- RFC-Editor, 5
- RPC, 1, 8
- rpc, 2
- RSA-SHA1, 15

- Signature Base String, 14
- SOAP, 2, 8
- SSL/TLS, 15
- Status Code, 10
- StudiVZ, 2
- Systemtest, 40

- Temporary Credential Request, 16
- Temporary Credentials Request, 17
- Token, 16–18
- Token Request, 16

- Ubuntu, 30
- Uniform Interface, 9
- URI, 1, 2, 16, 21, 52

- WebDAV, 9
- Windows, 30

- XAMP, 27, 30
- XAMPP, 30
- XML, 1