FH JOANNEUM Gesellschaft mbH

Python und RESTful Services

Bachelorarbeit 1
zur Erlangung des akademischen Grades
Bachelor of Science in Engineering (BSc)

eingereicht am 08.06.2015

Fachhochschul-Studiengang Internettechnik

Betreuer: DI (FH) Johannes Feiner

eingereicht von: Peter Justin
Personenkennzahl: 1210418013

Juni 2015

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Proleb, am 8. Junil 2015 Peter Justin

Abstract

Web services have existed since the beginning of the World Wide Web. In general, every website and web application is a service. Recently, one particular service the so called **RE**presentational **S**tate **T**ransfer (REST) Service is getting more and more important because REST aims at scalability and simplicity compared to various other services such as SOAP (Simple Object Access Protocol). It builds upon basic web technologies like HTTP, URI and XML.

This bachelor thesis aims to give the reader an insight into REST and its security aspects. Security is an important part of REST because sensitive information is transferred from the server to the users client. Another aspect is authorization and authentication. How do you identify a user? Which resource should be accessible for the user?

The empirical part discusses the implementation of a REST API on an existing open source project – FlaskBB. FlaskBB is a forum software written in Python using the micro framework Flask. Due to the fact that there are many possibilities to secure a REST API choosing the right one might be challenging. Therefore, the paper discusses the different ways of securing REST API's by taking their advantages and disadvantages into consideration.

Kurzfassung

Web Services gibt es seit dem Beginn des World Wide Web. Jede Webseite ist auch ein Service. Genau ein Service, der Representational State Transfer, kurz REST, Service der zugleich mit HTTP/1.1 entwickelt worden ist, wird immer wichtiger. REST ist im Vergleich zu anderen Web Services mit der Idee entworfen worden, gut skalierbar und einfach zu sein. Es baut auf den einfachsten Web Technologien wie HTTP, URI und XML auf.

Diese Arbeit soll dem Leser einen Einblick in REST und den Sicherheitsaspekten von REST geben. Besonders die Sicherheit ist ein wichtiger Teil von REST, da nicht jede Aktion von einem User ausgeführt werden darf. Dazu gehören auch Autorisierung und Authentifizierung. Was im Grunde heißt, wie ein User identifiziert wird und welche Ressourcen zugänglich für ihn sind.

Im Praktischen Teil, wird Anhand des Open Source Projektes – FlaskBB – gezeigt, wie dies implementiert werden kann. FlaskBB ist eine Forensoftware die in Python geschrieben ist und das Framework Flask benutzt.

Es gibt viele verschiedene Wege um eine REST API abzusichern und deshalb kann es schwer sein die richtige Methode zu finden. Deswegen wird im praktischen Teil erklärt welche Vorteile und Nachteile die verschiedenen Methoden haben.

INHALTSVERZEICHNIS

1 Einleitung 6
1.1 Motivation 6
1.2 Problemstellung 6
1.3 Zielsetzung 7
2 Stand der Technik 8
2.1 Was sind RESTful Web Services? 8
2.1.1 Die Resource-Oriented Architecture (ROA) 8
2.2 Darstellung 12
2.2.1 XML 12
2.2.2 JSON 13
2.3 Autorisierung und Authentifizierung 14
2.3.1 HTTP Basic Authentication 14
2.3.2 HTTP Digest Authentication 15
2.3.3 OAuth 2.0 15
2.4 Python 17
2.4.1 Flask 19
2.4.2 SQLAlchemy 21
3 Konzept 23
3.1 Vorgehensweise 23
4 Umsetzung am Beispiel FlaskBB 24
4.1 Verwendete Technologien 24
4.2 Implementierung 25
4.2.1 REST API 26
4.2.2 Validierung der Argumente 27
4.2.3 Serialisierung 28
4.2.4 HTTP Basic Authentication 29

4.2.5 OAuth Analyse 31
4.3 OAuth Provider Prototype 40
4.3.1 Ressourcen Bereiche 42
4.4 OAuth Client Prototype 42
5 Zusammenfassung 44
5.1 Erfahrungen und Ergebnisse 44
5.2 Mögliche Verbesserungen und Weiterentwicklungen 44
6 Literaturverzeichnis 46
7 Abbildungsverzeichnis 49
8 Listingverzeichnis 50
9 Abkürzungsverzeichnis 51

1 EINLEITUNG

1.1 MOTIVATION

In den letzten Jahren haben RESTful Services immer mehr an Bedeutung bekommen. Aufgrund dessen, dass der Begriff RESTful nur die Architektur beschreibt, werden RESTful Services oft durch Mangel an Wissen oft falsch umgesetzt. (vgl. (Davis, 2012, S. 3))

Besonders eine RESTful-Schnittstelle hat sich in letzter Zeit etabliert – die Resource-Oriented-Architecture, ROA. Ein weiterer wichtiger Punkt beim Bereitstellen einer API ist das einschränken von Daten auf bestimmte Benutzer oder Benutzerinnen. (vgl. (Yating & David, 2010, S. 164)) Diese zwei Punkte sind die Hauptthemen der vorliegenden Arbeit.

Hauptmotivation zur Entwicklung eines RESTful Services ist das Interesse an der Webentwicklung. Des Weiteren kommt hinzu, dass das Software Projekt „FlaskBB“, welches vom Autor der vorliegenden Arbeit entwickelt wird, bereits viele Entwickler/innen angezogen hat. Genau durch dieses Interesse wurden konkrete und zukunftsführende Ziele gesucht. Das ergab, dass eine REST API jenes ist, das FlaskBB von ähnlichen Software-Projekten abheben soll.

1.2 PROBLEMSTELLUNG

Für die vorliegende Arbeit soll ein System implementiert werden, das eine REST API zur Verfügung stellt. Vor allem der Aspekt „Sicherheit“ soll genauer untersucht werden. Außerdem soll dieses System in eine bereits bestehende Software integriert werden.

Konkret geht es um die Foren Software FlaskBB. Sie stellt den Benutzer oder den Benutzerinnen eine Plattform zur Verfügung, auf der themenorientierte Diskussionen geführt werden können. FlaskBB wird mit der Idee entwickelt, einen Baustein zu schaffen, mit der Communities gegründet werden können. So kann jeder die Software herunterladen und installieren. Damit für FlaskBB Mobile Apps oder dergleichen entwickelt werden können, wird eine für Entwickler/innen plattformunabhängige REST API benötigt. Genau diese Funktion soll im Zuge der Arbeit geschaffen werden.

1.3 ZIELSETZUNG

Die Schritte, die benötigt werden, um eine REST API zu entwerfen, sollen analysiert und umgesetzt werden. Außerdem soll auf die Sicherheit besonderen Wert gelegt werden. Um dies durchführen zu können, müssen zuerst die einzelnen Sicherheits-Methoden untersucht werden. Danach soll entschieden werden, welche dieser Methoden die Beste für den von oben genannten Zweck ist.

2 STAND DER TECHNIK

Dieses Kapitel beschäftigt sich mit den Technologien die in dieser Arbeit verwendet wurden. Unter anderem wird auch die genaue Bedeutung von REST und RESTful Services erläutert.

2.1 WAS SIND RESTFUL WEB SERVICES?

“The term “RESTful” is like the term “object-oriented.” A language, a framework, or an application may be designed in an object-oriented way, but that doesn’t make its architecture the object-oriented architecture.” (Richardson & Ruby, 2007)

Im Wesentlichen bedeutet dies, dass der Begriff RESTful nur angibt, wie eine REST API beschrieben werden kann. Es sagt nichts über die Architektur aus. REST ist nur eine Menge von Entwurfskriterien, an die sich eine Architektur halten soll. Dadurch können verschiedene Architekturen entworfen werden, wobei wiederum eine Architektur diese Kriterien besser implementiert als die andere. (vgl. (Richardson & Ruby, 2007, S. 16), (Davis, 2012, S. 3))

Eine Architektur, die angibt, wie so eine REST API entworfen werden kann ist die Resource-Oriented Architecture, kurz ROA, mehr dazu in Kapitel 2.1.1.

2.1.1 DIE RESOURCE-ORIENTED ARCHITECTURE (ROA)

Die Ressource (in Englisch – Resource) besteht aus mindestens einer URI (Uniform Resource Identifier). Die URI ist der Name und die Adresse der Ressource, denn ohne URI kann auf eine Ressource nicht zugegriffen werden – da sie dann nicht im Web wäre. (vgl. (Cesare, Olaf, & Frank, 2008, S. 807)) Eine jede Ressource ist einzigartig. Es ist aber möglich, dass zwei Ressourcen auf die gleichen Daten zeigen. Ein Beispiel hierfür ist das Downloaden einer App. Die neueste Version ist 1.0. Hier bietet sich an, eine eigene URI für die aktuellste Version und eine für zum Beispiel Version 1.0 anzubieten. Wenn die aktuellste Version auch Version 1.0 ist, dann zeigen beide auf die gleichen Daten. (vgl. (Richardson & Ruby, 2007, S. 83))

Außerdem sollten URIs eine Struktur haben und den Zweck der Ressource beschreiben. Da auf eine Ressource mehr als eine URI verweist, wird empfohlen, eine „Haupt-URI“ zu verwenden. Dies hat den Vorteil, dass alle URIs, die auf die gleiche Ressource verweisen, auf die „Haupt-URI“ weitergeleitet (HTTP Response Code 303) werden. (vgl. (Richardson & Ruby, 2007, S. 84))

Auch die Adressierbarkeit einer Ressource spielt eine wichtige Rolle. Da Ressourcen durch URIs angegeben werden, gibt es für jede Ressource eine eigene URI. Dadurch ist es zum Beispiel möglich eine Ressource als Lesezeichen zu speichern oder im Browser „zurück“ gehen. (vgl. (Davis, 2012, S. 4))

Eine weitere wichtige Eigenschaft von ROA ist die Zustandslosigkeit. Dies bedeutet, dass jede Anfrage für sich komplett unabhängig abläuft. Jede Anfrage enthält alle Informationen die benötigt werden, um die Anfrage abzuschließen. Aufgrund dessen, sind alle Anfragen auch unabhängig für den Server (eine Anfrage hängt nicht von vorherigen Anfragen ab) und der Client hat die komplette Kontrolle über den Zustand der Applikation. (vgl. (Richardson & Ruby, 2007, S. 87))

Zum Verarbeiten von Ressourcen setzt ROA bzw. REST auf HTTP Methoden wie HEAD, OPTIONS, GET, POST, PUT, DELETE und PATCH. (vgl. (Richardson & Ruby, 2007, S. 97), (Davis, 2012, S. 4))

- GET
  Diese Methode wird zum Holen der Information benötigt. Bei dieser Anfrage sendet der Server die Daten an den Client und übergibt auch einen Status Code. Falls alles geklappt hat, 200. Wenn der Server die gegebene Ressource nicht finden konnte wird der Status Code 404 zurückgegeben (dies gilt auch für die anderen HTTP Methoden).
- POST
  Die POST Anfrage wird benutzt um eine neue Ressource zu erstellen. Bei dieser Anfrage werden, auf eine bestehende URI, die Daten an den Server gesendet.
- PUT
  Um eine bestehende Ressource zu verändern, werden mit der PUT Anfrage die Daten an den Server gesendet. Mit der PUT Anfrage können auch Ressourcen erstellt werden, falls die endgültige URI der Ressource schon bekannt ist.
- DELETE
  Bei einer DELETE Anfrage wird eine bereits existierende Ressource gelöscht.
- HEAD
  Mit der HEAD Anfrage wird der HTTP Header einer Ressource abgefragt.

- OPTIONS

  OPTIONS gibt eine Liste von Unterstützten HTTP Methoden für eine Ressource zurück.

- PATCH

  Anders als bei der PUT Methode, ist es mittels PATCH möglich nur einzelne Teile einer Ressource zu modifizieren.

Eine Antwort des Servers bei einer erfolgreichen GET-Anfrage könnte wie folgt aussehen:

```
HTTP/1.0 200 OK
Content-Length: 501
Content-Type: application/json
Date: Sat, 06 Jun 2015 10:23:00 GMT
Server: Werkzeug/0.10.4 Python/2.7.6

{
    "user": {
        "birthday": null,
        "date_joined": "Thu, 28 May 2015 07:21:13 -0000",
        "email": "test1@example.org",
        "gender": null,
        "id": 1,
        "language": "en",
        "lastseen": "Sat, 06 Jun 2015 09:44:30 -0000",
        "location": null,
        "notes": null,
        "post_count": 4,
        "primary_group": "Administrator",
        "signature": null,
        "theme": null,
        "username": "test1",
        "website": null
    }
}
```

Listing 1: GET-Anfrage

In der ersten Zeile steht welche HTTP Version verwendet wird (hier HTTP/1.0) und der Status-Code. Da die Anfrage erfolgreich ist, ist der Status Code *200 OK*. Gibt es die gewünschte Ressource nicht am Server geben würde der Status Code *404 Not Found* lauten. (vgl. (The Internet Engineering Task Force, Hypertext Transfer Protocol -- HTTP/1.1, 1999, *„HTTP Version", „Status Code and Reason Phrase"*))

Content-Length gibt an wie groß der Inhalt der Nachricht ist. Die Nachricht startet nach dem Zeilenumbruch. Im obigen Beispiel ist die Nachricht 501 Oktetts lang wobei hier ein Oktett ein Byte darstellt. (vgl. (The Internet Engineering Task Force, Hypertext Transfer Protocol -- HTTP/1.1, 1999, *„Content-Length"*)) Der Grund

warum Oktetts verwendet werden ist jener, weil nicht auf allen Systemen ein Oktett einem Byte entsprechen.[1]

Um den Typ der Nachricht zu bestimmen, wird das Feld *Content-Type* benutzt. In diesem Beispiel hat das Feld den Wert *application/json*. Bei einer HTML-Rückgabe wird das Feld *Content-Type* auf *text/html* gesetzt. Durch dieses Feld ist es dem Client möglich die Nachricht dementsprechend zu darstellen. (vgl. (The Internet Engineering Task Force, Hypertext Transfer Protocol -- HTTP/1.1, 1999, *„Content-Type“*))

Das Feld *Date* gibt lediglich an wann die Nachricht erstellt wurde.

Das Server-Feld beinhaltet informationen über welchen Server die Nachricht an den Client gesendet wurde. Jedoch sollte nicht die Server-Version inkludiert werden, da dies ein potentielles Sicherheitsrisiko darstellt. (vgl. (The Internet Engineering Task Force, Hypertext Transfer Protocol -- HTTP/1.1, 1999, *„General Header Fields“*))

Nach den Header-Feldern beginnt nach dem Zeilenumbruch die Nachricht. In diesem Beispiel ist die Nachricht ein JSON-Objekt das Informationen über einen *User* enthält. (vgl. (The Internet Engineering Task Force, Hypertext Transfer Protocol -- HTTP/1.1, 1999, *„Entity Body“*))

---

[1] http://www.tcpipguide.com/free/t_BinaryInformationandRepresentationBitsBytesNibbles-3.htm - Abgerufen am 5. Juni 2015

2.2 Darstellung

Für die Darstellung von Ressourcen gibt es verschiedene Möglichkeiten. Die wohl bekanntesten sind XML (e**X**tensible **M**arkup **L**anguage) und JSON (**J**ava**S**cript **O**bject **N**otation).

2.2.1 XML

“Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them.”

(W3C, Extensible Markup Language (XML) 1.0 (Fifth Edition), 2015, *„Introduction“*)

XML wird bei RESTful Services benutzt um eine Datenstruktur zu beschreiben. Weil mit XML eine Datenstruktur sehr ausführlich beschrieben werden kann, ist es eines der am weitesten verbreiteten Formate für RESTful Services. (vgl. (Richardson & Ruby, 2007, S. 294)) Durch XSD, die XML Schema Definition, kann definiert werden wie ein XML Dokument strukturiert werden muss – sie validiert das XML, und prüft, ob es Fehler enthält. (vgl. (W3C, XML Schema Definition Language (XSD), 2012, *„Purpose“*))

Eine solche Datenstruktur kann zum Beispiel so aussehen:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<category position="1">
    <forums>
        <forum unread="true">
            <name>News</name>
            <description>All FlaskBB related news.</description>
            <last_post>FlaskBB 1.0 delayed :(</last_post>
            <last_post_by>sh4nks</last_post_by>
        </forum>
        <forum unread="false">
            <name>Welcome</name>
            <last_post>Hello, I am Peter.</last_post>
            <last_post_by>sh4nks</last_post_by>
        </forum>
    </forums>
</category>
```

Listing 2: XML Beispiel

- jedes Dokument enthält mindestens ein Element
- es existiert genau ein Root Element
- ein Element kann mehrere Kind Elemente enthalten

Bei XML ist jeder Wert von Tags umgeben. Diese Tags können wiederum Eigenschaften besitzen, die den Wert besser beschreiben. Jeder Tag <*Tag*> muss auch geschlossen </*Tag*> werden. (vgl. (Richardson & Ruby, 2007, S. 40))

Ein Nachteil von XML ist, dass durch die ausführliche Beschreibung der Werte, bei großen Datenstrukturen, viel Aufwand beim Parsen und Validieren des Dokuments entsteht. Auch die Übertragung des Dokumentes vom Server zum Client ist dadurch etwas kostspieliger. (vgl. (Richardson & Ruby, 2007, S. 40))

2.2.2 JSON

„JSON is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript programming language, but is programming language independent. JSON defines a small set of structuring rules for the portable representation of structured data.“ (ECMA International, 2013, S. 1)

Ursprünglich ging JSON aus der Notation von JavaScript-Objekten hervor. Jedoch bedient sich JSON nur an einer kleinen Teilmenge von Datenformate an ECMAScript, für die Darstellung von Daten wie zum Beispiel *Text, Values, Objects, Arrays, Numbers* und *String.* (ECMA International, 2013, S. 1-6)

Meistens besteht ein JSON-Objekt aus Key-Value Paaren. Wobei die Values wiederum Arrays von Key-Value Paaren beinhalten können. (vgl. (json.org, 2015, *„Introducing JSON“*))

Am besten lässt sich dies anhand eines Beispiels darstellen:

```
{
    "users": [
        {
            "date_joined": "Thu, 28 May 2015 07:21:13 -0000",
            "email": "me@peterjustin.me",
            "gender": null,
            "id": 1,
            "language": "en",
            "username": "peter"
        },
        {
            "date_joined": "Thu, 28 May 2015 09:34:51 -0000",
            "email": "sh4nks@flaskbb.org",
            "gender": "male",
            "id": 2,
            "language": "de",
            "username": "sh4nks",
        }
    ]
}
```

Listing 3: JSON Beispiel

Wie am Beispiel zu erkennen ist, gibt es bei JSON im Gegensatz zu XML keine Tags. Dadurch ist JSON leichter zu lesen und durch das Wegfallen der Tags, entsteht ein viel geringerer Payload beim Übertragen der Daten. Auch das Parsen von JSON ist einfacher und schneller als bei XML. (vgl. (Richardson & Ruby, 2007, S. 47))

Ein Nachteil von JSON ist, das es keine integrierte Schema-Validation wie XML mit XSD besitzt.

2.3 AUTORISIERUNG UND AUTHENTIFIZIERUNG

> "[...] HTTP authentication and authorization are handled with HTTP headers— "stickers" on the HTTP 'envelope.'" (Richardson & Ruby, 2007, S. 238)

Ein integraler Bestandteil von modernen RESTful APIs ist die Autorisierung und Authentifizierung. Autorisierung besagt, welche Berechtigungen der Client (meistens ein Benutzer oder eine Applikation) hat und auf welche Ressourcen er zugreifen darf. Die Authentication identifiziert den Client mit seinen Anmeldedaten. HTTP bietet verschiedene Möglichkeiten um einen Client zu authentifizieren und autorisieren. Dabei gibt es zwei Methoden, die bereits in HTTP integriert sind: HTTP Basic Authentication und HTTP Digest Authentication. (vgl. (Boyd, 2012, S. 2))

Im Wesentlichen lassen sich Autorisierung und Authentifizierung mit zwei indirekten Fragen beantworten:

- Autorisierung
  Wozu ist der Client oder der Benutzer berechtigt?
- Authentifizierung
  Wer ist der Client oder Benutzer?

2.3.1 HTTP BASIC AUTHENTICATION

Die HTTP Basic Authentication Methode setzt voraus, dass der Client sich mit einer eindeutigen Benutzer ID und einem Passwort anmeldet. Sie basiert auf dem Prinzip der „Challenge-Response" – Methode, bei der ein Beteiligter die Frage stellt und ein anderer die richtige Antwort geben muss. Die Anmeldedaten werden unverschlüsselt, in base64 enkodiert und übertragen, sofern keine externe

Verschlüsselungsmethode wie SSL verwendet wird. Da HTTP zustandslos ist, müssen die Anmeldedaten bei jeder Anfrage mitgeschickt werden. (vgl. (The Internet Engineering Task Force, HTTP Authentication: Basic and Digest Access Authentication, 1999, *„HTTP Basic Authentication“*))

2.3.2 HTTP DIGEST AUTHENTICATION

Bei der Digest Authentication wird wie bei der HTTP Basic Authentication auf das Prinzip der „Challenge-Response“ – Methode gesetzt. Der Unterschied zu Basic Authentication ist, dass die Anmeldedaten vor dem Versenden verschlüsselt werden. Vorausgesetzt, das Programm, das die Antwort stellt, unterstützt auch diese Verschlüsselungsmethode. Unterstützt das Programm die Verschlüsselungsmethode nicht, können die Anmeldedaten nicht entschlüsselt werden und somit schlägt die Authentifizierung fehl. Auch bei dieser Methode müssen die Anmeldedaten jedes Mal mitgeschickt werden. Ursprünglich wurde HTTP Digest Authentication entwickelt um Anmeldedaten verschlüsselt zu transferieren, falls keine verschlüsselte Verbindung zwischen Server und Client besteht. Der Inhalt einer Anfrage wird jedoch nicht verschlüsselt. (vgl. (The Internet Engineering Task Force, HTTP Authentication: Basic and Digest Access Authentication, 1999, *„HTTP Digest Authentication“*))

Da aber nur die MD5 – Hashfunktion zur Verfügung steht, und diese heutzutage als nicht mehr sicher angesehen wird, wird auch von dieser Methode abgeraten, falls die Verbindung unverschlüsselt ist. (vgl. (The Internet Engineering Task Force, MD5 and HMAC-MD5 Security Considerations, 2011, *„Security Considerations“))*

2.3.3 OAUTH 2.0

„OAuth enables clients to access protected resources by obtaining an access token, which is defined in "The OAuth 2.0 Authorization Framework" as "a string representing an access authorization issued to the client", rather than using the resource owner's credentials directly.“ (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework: Bearer Token Usage, 2012, “*Introduction”*)

Im Vergleich zu HTTP Basic Authentication, geht die Version 2 von OAuth einen ganz anderen Weg, wie aus dem Zitat zu entnehmen ist. Dabei wird eine Kette von Zeichen, sogenannte Tokens, anstatt der Anmeldedaten verwendet. Mittels dieser

Tokens ist es einem Benutzer oder einer Benutzerin möglich, bestimmte Ressourcen durch Rollenverteilung einem Client zur Verfügung zu stellen. (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework, 2012, *„Introduction“*))

Im Autorisierungsprozess sind drei Parteien verwickelt:

- Benutzer
  Dem Benutzer gehört die Ressource. Die Applikation muss, um die Daten des Benutzers verwenden zu dürfen, die Berechtigung des Benutzers einholen.
- Client
  Der Client ist eine Applikation, die versucht auf die Ressourcen des Benutzers zuzugreifen. Der Benutzer muss dem Client zuerst die Berechtigung geben, damit der Client darauf zugreifen kann.
- Ressource Server
  Auf dem Ressource Server sind die Daten gespeichert. Sie werden mittels einer API dem Benutzer oder der Benutzerin bereitgestellt.

Um OAuth bei einem Client verwenden zu können, muss der Client zuerst am Autorisierungs-Server registriert werden. Es muss auch eine „Redirect URI“ am Server hinterlegt werden um die Autorisierung zu vervollständigen. Diese URI muss verschlüsselt sein, damit keine Tokens von Angreifern abgefangen werden können. (vgl. (Boyd, 2012, S. 9))

2.4 PYTHON

“Python is powerful… and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open.” (Python, 2015)

Python ist eine interpretierte, objektorientierte, höhere Programmiersprache. Die erstmalige Veröffentlichung von Python, nach knapp über einem Jahr Entwicklung, erfolgte im Februar 1991 über einen USENET Eintrag von Guido Van Rossum. (vgl. (Rossum, 1991, *„HISTORY“*); (Python Software Foundation, 2015, *„General Python FAQ“*))

Da Python eine interpretierte Sprache ist, muss der Quellcode davor nicht in Maschinencode kompiliert werden, so wie zum Beispiel bei C oder C++. Damit der Interpreter den Quelltext besser und schneller durchlaufen kann, wird der Quelltext zu Bytecode – welcher diverse Optimierungen enthält - kompiliert. Dadurch entfällt auch das wiederholte Parsen des Quelltextes. (vgl. (Weigand, 2010, S. 27); (Python Software Foundation, 2015, *„Design and History FAQ“*)). Außerdem ist Python plattformunabhängig, d.h., derselbe Quelltext und Bytecode läuft auf vielen verschiedenen Betriebssystemen wie z.B. Windows, OS X und Linux. Die einzige Voraussetzung ist, dass auf die gewählte Plattform ein Python Interpreter installiert wurde. (Theis, 2011, S. 14)

Wie bei den meisten höheren Programmiersprachen ist auch bei Python die „Syntax und Semantik [...] auf die Bedürfnisse von Menschen zugeschnitten und nicht auf die technischen Spezifika der Maschine, die das Programm ausführen soll.“ (vgl. (Weigand, 2010, S. 26)). Dies macht Python zu einer der beliebtesten Einsteiger Programmiersprachen überhaupt. Aber nicht nur im Einsteigerbereich ist Python beliebt – auch im professionellen Umfeld ist Python sehr weit verbreitet. (vgl. (Peter, Jeffrey, Allen B., & Chris, 2012, S. iii)) Bei Google ist Python eine der drei offiziellen Programmiersprachen. Guido Van Rossum, der Erfinder von Python, arbeitete von 2005 bis 2012 bei Google wo er die Hälfte seiner Zeit an Python arbeiten konnte. (vgl. (Rossum, Personal Home Page, *„Resume“*))

Die Referenzimplementierung von Python ist in C geschrieben und heißt CPython (nur um von anderen Implementierungen unterscheidbar zu sein). Es existieren auch andere Implementierungen von Python wie z.B. Jython – Jython kompiliert

Python Quelltext zu Java Bytecode welcher dann von der Java Virtual Machine, kurz JVM, ausgeführt werden kann. (vgl. (Jython, 2015, „*Why Jython*“))

Die aktuellste Version von Python ist 3.4 aber es wird weiterhin Python 2.7 bis 2020 unterstützt, da die Version 3 von Python Veränderungen mit sich brachte, die es verhindern, Python 2 Quelltext mit Python 3 auszuführen. (vgl. (Python Software Foundation, PEP 373 - Python 2.7 Release Schedule, 2008)) Zum Beispiel bPython 3 benutzt *unicode* wobei Python 2 *ASCII* verwendet.

Alles ist in Python ein Objekt und fast alle Objekte haben Attribute und Methoden. Anders als bei anderen objektorientierten Programmiersprachen, setzt Python auf einen lockeren Ansatz, so dass nicht alle Objekte Attribute oder Methoden besitzen müssen. Python unterscheidet sich darin, dass alle Objekte einer Variable zugeordnet oder auch einer Funktion übergeben werden können. Beispiele für Objekte sind: Zeichenketten, Listen, Funktionen und sogar Module, die importiert werden. (vgl. (Pilgrim, 2004, S. 12))

Es gibt in Python zahlreiche Web Frameworks wie zum Beispiel Flask (mehr dazu im Kapitel 2.4.1 Flask) oder Django. Generell hat Python ein sehr gutes Ökosystem mit über 58000[2] Paketen im Python Package Index, kurz PyPI, und dabei gibt es für fast jeden Anwendungsfall eine Bibliothek. Sehr gerne wird Python auch im wissenschaftlichen Bereich eingesetzt um verschiedenste Algorithmen und Berechnungen durchzuführen. (vgl. (Millman & Aivazis, 2011, S. 9)) SciPy ist zum Beispiel eine Bibliothek die speziell für den Wissenschaftlichen Bereich entwickelt wird. Für Performance kritische Algorithmen oder Methoden ist es möglich, sie in einer C Bibliothek auszulagern. Falls eine Bibliothek komplett in C geschrieben ist, kann mit einem speziellen Python-Wrapper die C Bibliothek in Python verfügbar gemacht werden. (vgl. (Python Software Foundation, ctypes - A foreign function library for python))

[2] https://pypi.python.org/pypi - Abgerufen am 20. April 2015

2.4.1 FLASK

“Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions” (Ronacher, Flask, 2015)

Auch in der Web Entwicklung braucht sich Python nicht zu verstecken. Wie bereits in Kapitel 2.4 angesprochen, gibt es zahlreiche Web Frameworks für Python. Eines davon ist Flask, welches einen ganz anderen Weg als die meisten Frameworks einschlägt. Es wird zwar als „microframework“ bezeichnet – das heißt aber nicht, dass es qualitativ minderwertig ist und nur für kleine Webseiten gedacht ist, sondern wurde es mit der Idee entwickelt, erweiterbar zu sein mittels Erweiterungen. (vgl. (Ronacher, Flask Documentation – What does „micro“ mean?, 2015))

Flask bietet nur die wichtigsten Funktionalitäten, die benötigt werden, um eine Webseite zu erstellen, sind dafür aber diese sehr gut getestet worden und entsprechend dokumentiert. Es baut selbst auf zwei weitere Bibliotheken auf. (vgl. (Grinberg, 2014, S. 3))

- Werkzeug

  Werkzeug ist im Prinzip nur eine Bibliothek, die das Arbeiten mit dem Web Server Gateway Interface (WSGI) erleichtert. Flask bedient sich unter anderem dem Routing, Debugging und den diversen anderen WSGI Werkzeugen. (vgl. (Ronacher, Werkzeug – Welcome, 2015), (Grinberg, 2014, S. 3))

- Jinja2

  Ein Vorteil von Flask ist, dass es Logik und HTML strikt voneinander trennt. Deshalb ist es nötig, eine sogenannte Templating Engine zu verwenden, um Variablen im HTML auszugeben. Für dies wird eine Vorlage – ein Template – verwendet mit welchem die Templating Engine ein HTML Dokument generieren kann. Flask verwendet hierbei Jinja2. (vgl. (Ronacher, Flask Documentation – Design Decisions in Flask, 2015))

Beide dieser Bibliotheken werden von den Flask-Entwicklern gewartet und weiterentwickelt.

Durch das Prinzip der Erweiterbarkeit von Flask, kann der Benutzer oder die Benutzerin, Erweiterungen vom Python Package Index installieren, oder selbst eine Erweiterung implementieren, um die gewünschte Funktionalität zu erhalten. Deshalb

gibt es auch keine native Unterstützung für Datenbanken, Formularvalidierung, Authentifizierungen und viele mehr in Flask. Aber genau hier kommt der große Vorteil von Flask ins Spiel – der Entwickler oder die Entwicklerin kann sich ganz einfach die Bibliotheken seiner oder ihrer Wahl aussuchen. So ist es zum Beispiel möglich, SQL-Datenbanken und NoSQL-Datenbanken mit unterschiedlichen Erweiterungen zu unterstützen. (vgl. (Ronacher, Flask Documentation – Flask Extensions, 2015))

Ein weiteres wichtiges Merkmal ist, dass es mit Flask sehr einfach ist, REST-Endpoints (siehe Kapitel 4.2.1) zu erstellen. Es gibt im Prinzip zwei Wege um eine REST API zu erstellen. Erstens, den Routen-Dekorator zu verwenden und die gewünschten HTTP-Methoden mit zu übergeben oder zweitens, die Klasse *`MethodView`* zu implementieren und je nachdem welche HTTP Methode verwendet wird, die dazugehörige Methode der Klasse überschreiben. (vgl. (Ronacher, Flask Documentation – Method Views for APIs, 2015)) Folgende Methoden sind verfügbar:

- `get` – HTTP „GET“
  Hiermit wird der Inhalt einer Ressource geholt.
- `post` – HTTP „POST“
  Mit dieser Methode wird eine noch nicht bestehende Ressource erstellt
- `delete` – HTTP „DELETE“
  Diese Methode wird verwendet um eine Ressource zu löschen.
- `put` – HTTP „PUT“
  Damit wird eine bestehende Ressource aktualisiert.
- `patch` – HTTP „PATCH“
  Mittels der PATCH Methode ist es möglich nur einen Teil einer Ressource zu aktualisieren.
- `options` – HTTP „OPTIONS“
  Liefert die erlaubten HTTP Methoden für eine Ressource zurück.
- `head` – HTTP „HEAD“
  Gibt nur den HTTP Header zurück.

2.4.2 SQLALCHEMY

„SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL." (Bayer, 2015)

Mittels SQLAlchemy ist es möglich mehrere verschiedene Datenbanken zu unterstützen, ohne dadurch den Quelltext des eigenen Programmes zu verändern.

Zu den bekanntesten zählen hier: PostgreSQL, MySQL und SQLite. SQLAlchemy besteht aus zwei Komponenten – dem Kern und einem optionalen Object-Relational-Mapper, kurz ORM. Der Kern besteht aus dem Abstraktionslayer, welcher eine Abstraktion über viele verschiedene Datenbanken ist, und der SQL Expression Language – eine Sprache, die es ermöglicht, SQL Abfragen mit Python zu erstellen. Ein weiterer Teil der Kernkomponente ist die Inspizierung von bereits existierenden Datenbanken. Dadurch ist möglich, die existierende Datenbank mit SQLAlchemy zu verwenden. Die SQL Expression Language ist Teil der Kernkomponente. Der Vorteil ist, dass die Abfragen an die Datenbank anders als bei einem ORM komplett kontrolliert werden können. (vgl. (Bayer, SQLAlchemy – Key Features of SQLAlchemy, 2015))

Das ORM, eine optionale Komponente von SQLAlchemy, basiert auf den Funktionen der Kernkomponente auf. Ein ORM bildet eine Tabelle auf Objektebene ab. Eine Tabelle hat beispielsweise mehrere Zeilen und jede dieser Zeilen ist für das ORM ein Objekt. Diese Reihen haben auch Spalten wobei die Spalten die Attribute eines Objekts sind. Am leichtesten lässt sich dies anhand eines Beispiels darstellen:

id	vorname	nachname
1	Peter	Justin
2	Max	Mustermann

Diese Tabelle würde sich so in SQLAlchemy abbilden lassen:

```
class Benutzer(db.Model):
    id = db.Column(db.Integer)
    vorname = db.Column(db.String)
    nachname = db.Column(db.String)
```

Wie bereits erwähnt, bildet „id“ die Spalte „id“, „vorname“ die Spalte „vorname“ und „nachname“ die Spalte „nachname“ ab. Mit dieser Klasse können dann die Datenbank-Einträge direkt mit Python-Funktionen, die durch die Klasse db.Model mitvererbt werden, abgefragt werden. (vgl. (Ronacher, Flask-SQLAlchemy – A Minimal Application, 2015)

Es ist auch möglich, datenbankspezifische Funktionen zu verwenden. Dadurch wird die Auswahl der verschiedenen Datenbanken meistens genau auf eine beschränkt. Eine Ausnahme davon ist der JSON Typ. PostgreSQL hat einen nativen JSON Typ, MySQL hingegen hat keinen. Beim JSON Typ von SQLAlchemy wird dies folgendermaßen gelöst. Bei PostgreSQL wird der native Typ verwendet, bei MySQL der String Typ. Außerdem werden bei MySQL die Werte jedes Mal neu verarbeitet, wodurch sich gegenüber PostgeSQL ein Performancenachteil ergibt. (vgl. (Bayer, SQLAlchemy – Custom Types, 2015))

3 KONZEPT

Bevor mit der Implementierung eines Prototyps begonnen werden kann, muss evaluiert werden an welchen Stellen in FlaskBB eingegriffen werden muss. Auch für das Testen des Prototyps soll eine Lösung gefunden werden.

3.1 VORGEHENSWEISE

Als erstes soll untersucht werden ob es sinnvoll ist, die API mit Hilfe einer Flask-Erweiterung umzusetzen.

Nachdem es einen ersten Prototyp für die REST API gibt soll dieser nur mit OAuth zugänglich gemacht werden d.h. die Endpunkte sollen geschützt werden, so dass nicht autorisierte Benutzer oder Benutzerinnen keinen Zugriff darauf haben.

Gleichzeitig mit dem implementieren von OAuth soll auch ein Test-Client implementiert werden. Mit Hilfe dieses Clients soll getestet werden ob das Autorisieren von Ressourcen für Benutzer oder Benutzerinnen funktioniert.

Bevor mit der eigentlichen Implementation von OAuth angefangen werden kann, müssen zuerst die einzelnen OAuth-Autorisierungsverfahren analysiert werden. Danach soll entschieden werden mit welchen Autorisierungsverfahren die REST API abgesichert werden soll.

4 UMSETZUNG AM BEISPIEL FLASKBB

Um ein besseres Verständnis zu bekommen, wie eine REST API funktioniert, wird diese mit Flask implementiert. Dies soll unter anderem auch zeigen, wie eine REST API in eine bestehende Software integriert wird. Ein weiterer Teil der Implementierung ist ein einfacher Client, der zeigt, wie auf die Ressourcen von einem Client aus, zugegriffen werden können. Der Client ist ein reiner Prototyp und ist nicht für einen Einsatz in produktiven Systemen gedacht.

FlaskBB ist eine Foren Software, die für den Zweck entwickelt wird, dass der Betreiber eine Community aufbauen kann. Sie ermöglicht eine einfache Kommunikation zwischen Benutzer und Benutzerinnen in dem ein Benutzer oder eine Benutzerin ein Thema erstellt, in dem andere Antworten erstellen können. Eine weitere Funktionalität ist, die Möglichkeit „Private Nachrichten" zu verwenden. FlaskBB wird unter einer von der Open Source Initiative anerkannten Open Source Lizenz entwickelt - der 3-Klausel BSD Lizenz. (vgl. (Justin, FlaskBB – Features, 2015)) Dadurch kann die Software frei von der Projektseite (http://flaskbb.org) heruntergeladen werden. Der Source Code von FlaskBB steht auf https://github.com/sh4nks/flaskbb bereit und kann von jedem eingesehen werden. Durch den Ansatz von Open Source ist es möglich, seine eigenen Ideen in das Projekt einzubringen – sofern sie vom Haupt-Entwickler akzeptiert werden.

4.1 VERWENDETE TECHNOLOGIEN

Da die Software in Python mittels Flask Frameworks geschrieben ist, werden auch bei der Implementierung diese Technologien verwendet. Hinzu kommt, dass anstatt das Rad neu zu erfinden, bestehende Bibliotheken, die es vereinfachen, eine REST API in Flask zu implementieren, verwendet werden.

Clientseitig wird zu Testzwecken auch Python mit Flask verwendet. Die REST API ist aber nicht limitiert auf Python mit Flask. Sie kann theoretisch von allen Programmiersprachen angesprochen werden, für die es möglich ist, JSON zu verarbeiten und die entsprechenden Anfragen für OAuth abzusetzen.

Als Übertragungsformat der Ressourcen wurde JSON verwendet. JSON ähnelt der Struktur eines Python „Dictionaries" sehr, wodurch die Verarbeitung vereinfacht wird. Ein weiterer Grund warum JSON verwendet wurde ist, dass der Overhead im Vergleich zu XML wegfällt. Die bessere Lesbarkeit von XML ist in diesem Falle auch

unnötig, da die Daten nicht zum Lesen für einen Benutzer oder eine Benutzerin gedacht sind.

Für die Authentifizierung und Autorisierung wurde zuerst HTTP Basic Authentication gewählt, das aber später durch OAuth 2.0 ersetzt wurde. Wie bereits im Kapitel 2.3 besprochen, muss bei der HTTP Basic Authentication bei jeder Anfrage der Benutzername und das Passwort, welche in base64 kodiert sind, mitgeschickt werden. Solange die Verbindung verschlüsselt ist, sprich, über SSL läuft, ist es sicherheitstechnisch kein Problem, die HTTP Basic Authentication zu verwenden. Sie wird auch gerne eingesetzt, da sie leicht zu implementieren ist. Nicht jeder Benutzer oder jede Benutzerin ist jedoch damit Einverstanden, die Anmeldedaten preiszugeben. Auch wenn die Verbindung über SSL läuft, besteht noch immer die Möglichkeit die Benutzer-Passwort-Kombination mittels einer „Man-in-the-middle“ Attacke zu sniffen. Ein weiterer Schwachpunkt von HTTP Basic Authentication ist, dass es keine einfache Möglichkeit gibt, den Zugriff auf nur bestimmte Ressourcen zu beschränken. OAuth verwendet hingegen ein komplett anderes Konzept um einem Benutzer oder einer Benutzerin Zugriff auf Ressourcen zu gewähren.

Bei OAuth werden anstatt von Benutzername und Passwort Tokens verwendet. Dies erhöht die Sicherheit enorm, denn durch die Tokenstrategie erhält man die Möglichkeit, die verbundenen Clients aufzulisten. Falls auf einem Smartphone ein solcher Client installiert ist, aber das Smartphone verloren geht, ist es möglich, dem Client den Zugriff zu verwehren, indem das Token invalidiert wird. Ein weiterer Vorteil ist, dass der Client somit nie das Passwort übermittelt bekommt.

HTTP Digest Authentication wurde sofort ausgeschlossen, da es nur den Hashalgorithmus MD5 unterstützt und da FlaskBB serverseitig PBKDF2 verwendet und es ansonsten zu Kompatibilitätsproblemen gekommen wäre. Außerdem ist es nicht zu empfehlen, den MD5-Hashalgorithmus für Passwörter zu verwenden, da er als nicht mehr sicher eingestuft wird.

4.2 Implementierung

Die Implementierung dient nur als Hilfe um zu verstehen und zu erforschen, wie eine REST API mit besonderem Augenmerk auf Sicherheit umgesetzt werden kann. Da FlaskBB in Python entwickelt wird, wird auch für die Implementierung der REST API mit dem OAuth Provider Python verwendet. In anderen Programmiersprachen ist die Implementierung von OAuth sehr ähnlich.

4.2.1 REST API

Die REST API wurde anfangs noch mit normalen Python-Funktionen entworfen, wobei jede Funktion einer Route zugeordnet wurde. Da dies bei großen APIs schnell unübersichtlich wird, wurde nach Alternativen gesucht. Zuerst wurde mit der Klasse *MethodView,* die Flask bereitstellt, versucht, die API in Klassen zu unterteilen – was sich im Endeffekt auch bewährt hat. Die Methoden in der Klasse sind nach den HTTP Methoden benannt. Falls eine HTTP Methode nicht benutzt wird, wird die Methode einfach nicht implementiert. Das Zuordnen der HTTP Methoden zu den Methoden in der Klasse wird intern von Flask mittels Meta-Klassen gehandhabt.

```
def get(self):
    users = {'users': [marshal(user, user_fields)
                       for user in User.query.all()]}
    return users
```

Listing 4: Flask - UserListAPI "GET"

Diese Methode ist Teil der Klasse *UserListAPI.* Sie Implementiert die HTTP Methode GET und dient dazu, alle Benutzer oder Benutzerinnen abzurufen.

Wie bereits schon erwähnt, ist JSON sehr ähnlich dem *Dictionary*-Typ von Python. Ein *Dictionary* kann entweder mit *{key: value}* erstellt werden, oder mit der *dict()* Funktion von Python. Im obrigen Beispiel wird ein *Dictionary* erstellt, mit dem Root-Element *users*, welches alle Benutzer enthält. Um die Felder der Benutzer/innen aus den SQLAlchemy Models zu bekommen, wird mittels der *marshal* Funktion und der Hilfe der *user_fields*, welche die relevanten Felder für die Benutzer/innen enthält, jedes Benutzer-Objekt mit seinen Feldern erstellt. Dies wird als Serialisierung von Objekten bezeichnet (siehe Kapitel 4.2.3).

Durch die Flask Erweiterung *Flask-RESTful* wird automatisch beim Zurückliefern des *users*-Objektes ein JSON Objekt erstellt und gleichzeitig der *Content-Type* im HTTP Header auf *application/json* umgestellt.

Das Aktualisieren eines Benutzer-Objektes wird mit der mit der PUT Methode realisiert. PUT wird deswegen verwendet, weil die volle Ressource URI bekannt ist.

```
def put(self, user_id):
        """Updates a user."""
        user = User.query.filter_by(id=user_id).first()

        if not user:
            abort(404)

        args = self.reqparse.parse_args()
        for k, v in args.items():
            if v is not None:
                setattr(user, k, v)
        user.save()
        return {'user': marshal(user, user_fields)}
```

Listing 5: Flask - REST "PUT"

Anhand des Beispiels ist es gut erkennbar, dass auch für die HTTP Methode *PUT* die Python Methode *put* verwendet wird. Nach der Überprüfung, ob das Benutzer-Objekt existiert, werden die Parameter, welche in der HTTP Anfrage mitgeschickt wurden, überprüft ob sie den richtigen Typ besitzen (siehe Kapitel 4.2.2), und es wird festgestellt, ob sie den richtigen Werte-Typ besitzen. Die Parameter werden dann am *user*-Objekt gesetzt und gespeichert. Anschließend wird mittels Serialisierung das aktualisierte *user*-Objekt zurückgegeben.

Ähnlich funktioniert dies auch für die anderen HTTP Methoden *POST* und *DELETE*.

Warum sollen Erweiterungen verwendet werden?

Anstatt das Rad neu zu erfinden, soll auf gut getestete Bibliotheken gesetzt werden. Überlicherweise hat eine jede Erweiterung von Flask genau einen Zweck – „Mach eines, dies aber gut“. Aufgrund dessen wird hier auch *Flask-RESTful* verwendet, denn bei dieser Bibliothek/Erweiterung wird es dem Entwickler oder der Entwicklerin erleichtert eine RESTful API zu implementieren. Sie bietet speziell für REST APIs, einige Möglichkeiten um sie so bequem wie möglich umzusetzen. Dazu zählen unter anderem die Validierung der Argumente (siehe Kapitel 4.2.2) und die Serialisierung von Objekten. Die oben beschriebenen Beispiele können ähnlicher Form in anderen Programmiersprachen oder Frameworks umgesetzt werden.

4.2.2 Validierung der Argumente

Die Argumente-Validierung wird verwendet um zu verhindern, dass die Datenbank mit falschen Werten gefüttert wird. Sie hat denselben Zweck, wie eine Formularvalidierung auf HTML Webseiten. Es ist möglich, die Validierung rein auf dem Client durchzuführen, von dem aber dringend abgeraten wird, da der Client Fehler enthalten kann und dadurch falsche Werte an den Server sendet. Es ist zu

empfehlen, beide Arten der Validierung zu benutzen – am Client, damit der Benutzer oder die Benutzerin Rückmeldung bekommt, ob die Eingabe in Ordnung ist, und noch ein letztes Mal am Server, um wirklich sicherzustellen, dass die Werte in Ordnung sind.

Zum Validieren der Parameter wird wieder auf die Flask-RESTful-Erweiterung zurückgegriffen. Dies wird durch das Definieren der Argumente erreicht. Welche Wert-Typen müssen die Argumente haben und ist ein Argument optional oder erforderlich? Wenn das Argument nicht den gewünschten Typ aufweist, wird eine Fehlermeldung ausgegeben. Dasselbe gilt auch bei Argumenten, die erforderlich sind.

Zur Veranschaulichung des Ganzen, ein kleines Beispiel:

```
addparser = reqparse.RequestParser()
addparser.add_argument('username', type=str, required=True, location="json")
addparser.add_argument('email', type=str, required=True, location='json')
addparser.add_argument('password', type=str, required=True, location='json')
addparser.add_argument(group_id, type=int, required=False, location='json')
```

Listing 6: Parametervalidierung

4.2.3 SERIALISIERUNG

Die Serialisierung von Objekten aus dem ORM, ist eine wesentliche Funktion. Durch das Anlegen sogenannter *Fields*, also Felder, wird festgelegt, aus welchen Feldern die Ressource bestehen soll. Damit die Felder in der Ressource auch die richtigen Werte-Typen besitzen, wird ein Dictionary erstellt, mit dem Namen des Feldes und dem Typ des Wertes. Der Name entspricht hier, dem Namen im Model.

```
user_fields = {
    'id': fields.Integer,
    'username': fields.String,
    'date_joined': fields.DateTime,
}
```

Listing 7: Flask-RESTful Felder

Wie im Code-Auszug zu sehen ist, werden hier drei verschiedene Typen verwendet – ein Zahlentyp (Integer), ein Texttyp (String) und einen Datumstyp (DateTime). Verknüpft wird das Ganze mit der von Flask-RESTful mitgelieferten Funktion *marshal,* die im Grunde nur die einzelnen Objekte auf die Felder abbildet. Bei

Ausführung der *marshal*-Funktion würde das Ergebnis ein JSON Objekt generieren. Für zwei User-Objekte sieht es so aus:

```
{
    "users": [
        {
            "id": 1,
            "username": "test1",
            "date_joined": "Thu, 23 Apr 2015 07:00:18 -0000",
        },
        {
            "id": 2,
            "username": "test2",
            "date_joined": "Thu, 23 Apr 2015 07:00:18 -0000",
        }
    ]
}
```

Listing 8: Generiertes JSON Objekt

Natürlich ist es auch möglich einen anderen Rückgabetyp als wie JSON zu verwenden. Um zum Beispiel XML zu verwendet muss hierfür ein eigener *to-XML* Serializer implementiert werden. Für komplexe Strukturen können Felder in einander verschachtelt werden. Außerdem sind sie nicht abhängig vom ORM – sie können genauso in einer eigenen Klasse oder ähnlichem verwendet werden.

4.2.4 HTTP Basic Authentication

Wie bereits erwähnt, ist HTTP Basic Authentication relativ einfach zu implementieren. Aufgrund dessen wurde für diese Implementierung die Flask Erweiterung Flask-HTTPAuth verwendet. Diese Erweiterung abstrahiert für uns das Modifizieren des Headers und setzt den gewünschten *Authentication Realm*. Bei Eingang einer unautorisierten Anfrage für eine geschützte Ressource, antwortet der Server mit diesem Eintrag im Header:

```
WWW-Authenticate: Basic realm="FlaskBBRealm"
```

Abbildung 1: Serverantwort für Authentifizierung

Der Realm wird vom Server zugewiesen, um die geschützte Ressource zu identifizieren.

Falls jedoch der/die Benutzer/in eine Anfrage auf eine geschützte Ressource mit den Anmeldedaten sendet, sieht die Anfrage in etwa so aus:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Abbildung 2: Clientanfrage für Authentifizierung

Wie aus dem Beispiel zu entnehmen ist, werden die Anmeldedaten mit base64 kodiert. In Python würde das so aussehen:

```
@auth.verify_password
def verify_password(username, password):
    user, authenticated = User.authenticate(username, password)
    if user and authenticated:
        return True
    return False
```

Listing 9: Passwortvalidierung

Hierbei wird mit dem Dekorator ein Callback *verify_password* registriert. Dieser Callback wird immer dann aufgerufen, wenn eine Resource geschützt ist. Bei einem Aufruf wird jedes Mal die *User.authenticate* ausgeführt, die den Benutzer oder die Benutzerin authentifizieren.

4.2.5 OAUTH ANALYSE

Um zu verstehen, wie ein OAuth Provider/Server funktioniert, werden in den nächsten Unterkapiteln die verschiedenen Verfahren und Typen analysiert.

Im Wesentlichen bietet OAuth vier verschiedene Autorisierungsverfahren, um einem Client die Ressourcen des *Ressource Owner* zu gewähren. Durch die sogenannten Grant-Typen ist OAuth sehr flexibel, denn mit ihnen ist es möglich, dem Client auf unterschiedlichen Arten Zugriff auf die Ressourcen zu gewähren.

In den Prozess der Autorisierung sind vier unterschiedliche Parteien verwickelt (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Roles, 2012)):

- Ressource Owner
  Der Ressource Owner ist der Endbenutzer, dem Zugriff auf eine geschützte Ressource gewährt wird.

- Ressource Server
  Am Ressource Server sind die Informationen, also Ressourcen, gespeichert. Der Server ist auch zuständig für das Freigeben der Ressourcen mittels Access Token.

- Client
  Der Client ist üblicherweise eine Applikation, welche auf die Ressourcen des Ressource Owners - mit dessen Berechtigung - zugreift. Ein Client kann eine Applikation auf einem PC, Smartphone, etc. sein.

- Authorization Server
  Nachdem ein Ressource Owner authentifiziert wurde, stellt der Authorization Server ein Access Token aus um den Ressource Owner zu autorisieren. Oft ist der Authorization Server ein Teil vom Ressource Server.

Weiters gib es zwei Typen von Clients – den öffentlichen Typ „public“ und den vertraulichen Typ „confidental“. (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Client Types, 2012))

- public – Typ
  Bei diesem Typ ist es dem Client nicht möglich, die Anmeldedaten des Benutzers oder der Benutzerin vertraulich zu behandeln. Meistens ist dies bei installierten Programmen oder Web-Apps der Fall.

- confidental – Typ
  Der confidental-Typ kann die Sicherheit der Anmeldedaten gewährleisten. Üblicherweise gehören Clients, die auf einem abgesicherten Server (mit SSL) laufen und nur beschränkten Zugriff auf die Anmeldedaten haben dem Typ „confidental“ an.

Zusätzlich zu den zwei unterschiedlichen Typen von Clients, gibt es noch das Access Token und das Refresh Token.

- Access Token
  Das Access Token ist eine Zeichenkette, die dem Client Zugriff auf geschützte Ressourcen gewährt. Diese Zeichenkette ist so kodiert, dass der kodierte Inhalt des Tokens vom Client aus nicht gelesen werden kann. Diese Zeichenketten enthalten Informationen darüber auf welche Ressourcen zugegriffen werden darf und wie lange die Zeichenkette an sich gültig ist. (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Access Token, 2012))

- Refresh Token
  Wie beim Access Token, ist auch das Refresh Token eine Kette von Zeichen. Sie werden verwendet um ein neues Access Token anzufordern, wenn es abläuft oder ungültig wird. Wenn ein Access Token beim Authorization Server angefordert wird, wird auch automatisch ein Refresh Token erstellt. Dies ist jedoch optional und muss daher nicht verwendet werden. (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Refresh Token, 2012))

4.2.5.1 Authorization Code Grant

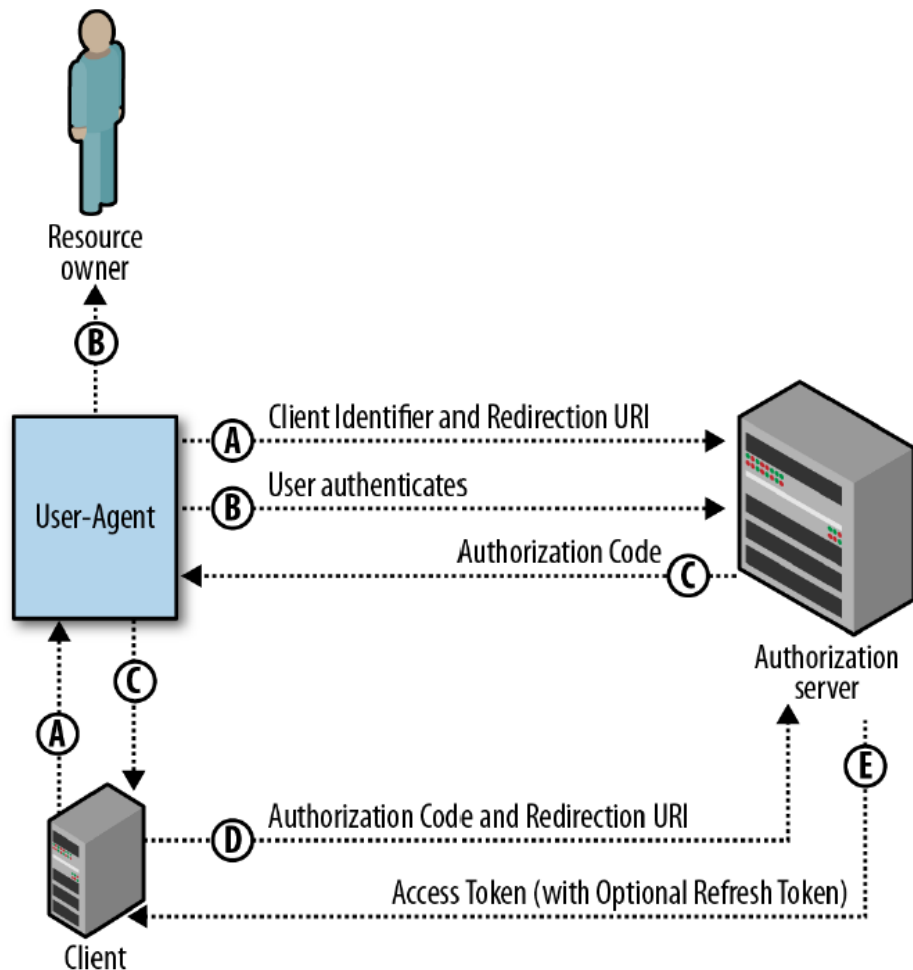

Abbildung 3: Authorization Code Grant (Boyd, 2012, S. 14)

Beim *Authorization Code Grant*-Verfahren wird ein Access Token und Refresh Token verwendet. Da Access Token nur eine bestimmte Lebenszeit haben, wird ein Refresh Token benötigt, um ein neues Access Token anzufordern. Es gehört zum oben beschriebenen Typ „confidental“. Wie aus der Abbildung zu entnehmen ist, ist bei diesem Typ ein Client erforderlich, der über einen User-Agent kommunizieren und eingehende Anfragen vom Authorization Server weiterleiten kann. Oft wird dieser Typ für Entwickler verwendet, die nichts mit dem API Betreiber zu tun haben – sogenannte „Third-Party-Developer“. (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Authorization Code Grant, 2012))

Der Ablauf in Abbildung 11 ist folgender:

A. Der Client startet den Autorisierungsprozess, in dem der Benutzer oder die Benutzerin zur Autorisierungs-URI weitergeleitet wird. Der Client muss hierbei seine eindeutige Identifikation, den Bereich der gewünschten Ressourcen und eine Weiterleitungs-URI mitsenden. Falls alles geklappt hat, wird der Client danach über die Weiterleitungs-URI zurückgeleitet.

B. Der Autorisierungs-Server authentifiziert dann den Benutzer oder die Benutzerin und liefert eine Meldung zurück, falls der Benutzer oder die Benutzerin auf die gewünschte Ressource Zugriff hat.

C. Wenn der/die Benutzer/in Zugriff auf die gewünschte Ressource hat, wird der Client über den User-Agent mit der Weiterleitungs-URI zurückgeleitet. Dieser URI wird auch noch ein Autorisierungscode angehängt.

D. Hier stellt der Client eine Anfrage am Autorisierungsserver mittels dem vom vorigen Schritt erhaltenen Autorisierungscode

E. Der Autorisierungsserver authentifiziert den Client, überprüft ob der Autorisierungscode in Ordnung ist und stellt sicher, ob die Weiterleitungs-URI dieselbe ist wie am Autorisierungsserver. Wenn alles in Ordnung ist, liefert der Autorisierungsserver einen Access Token und optional einen Refresh Token zurück.

4.2.5.2 Implicit Grant

Im Gegensatz zum obigen Verfahren, ist dieses Verfahren für den Typ „public“ optimiert. Auch wird hier, anstatt von unterschiedlichen Anfragen, um an das Access Token zu kommen, das Access Token direkt in der Autorisationsanfrage mitgeschickt. Außerdem bietet dieser Typ keine Unterstützung für Refresh Tokens. Diese Art wird vor allem von Clients verwendet, die mittels JavaScript im Browser laufen. Das Access Token wird direkt in die Weiterleitungs-URI angefügt, weswegen es auch für andere Programme am Gerät sichtbar sein kann. (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Implicit Grant, 2012))

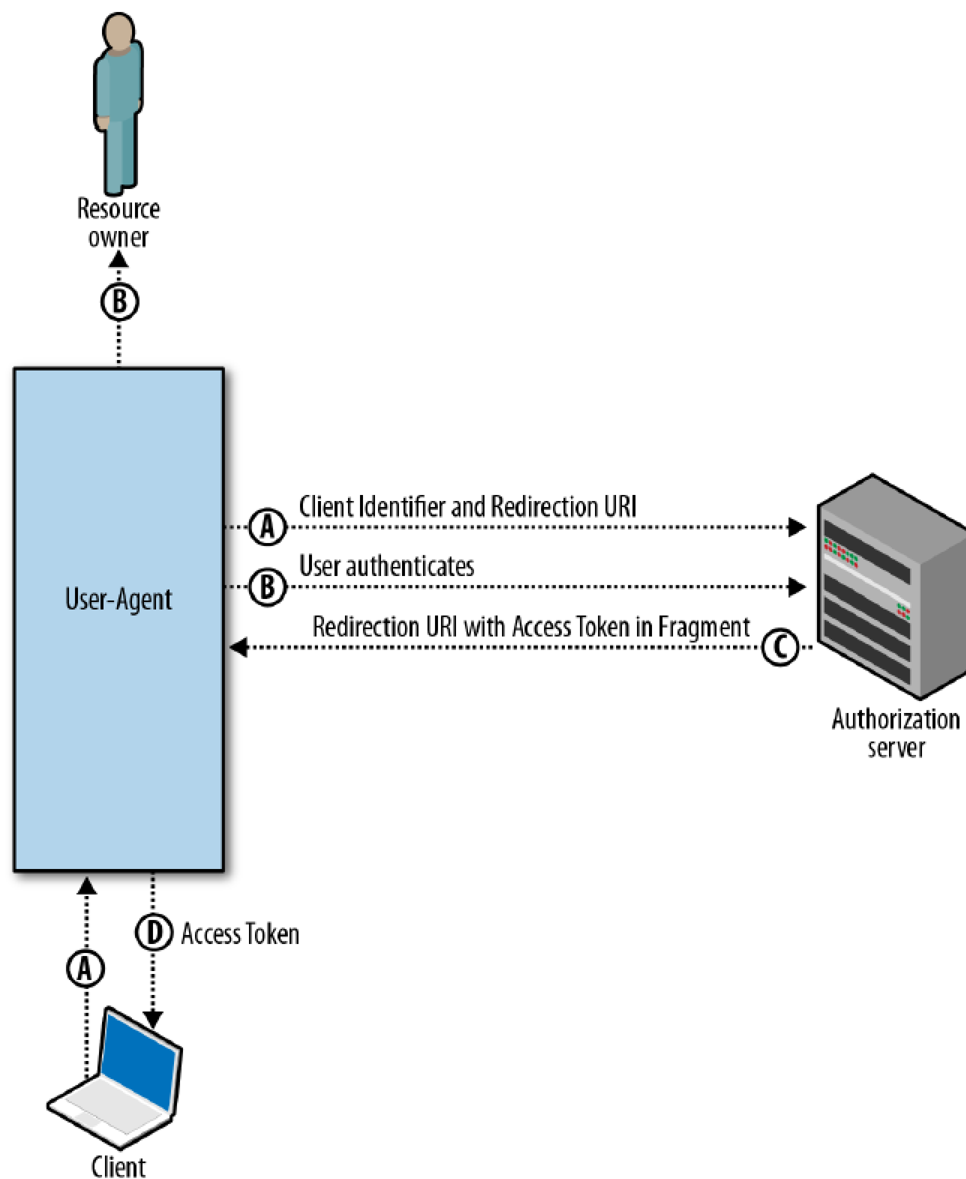

Abbildung 4: Implicit Grant (Boyd, 2012, S. 30)

A. und B. laufen gleich ab wie die Schritte von Authorization Code Grant in Abbildung 11.

C. Wenn der/die Benutzer/in Zugriff auf die gewünschte Ressource hat, wird der Client über den User-Agenten mit der Weiterleitungs-URI zurückgeleitet. Dieser URI wird dann noch das Access Token angehängt.

D. Der User-Agent (meistens ein Browser) wird durch eine Anfrage an die Web-Hosted-Client-Resource dorthin geleitet (ohne dem Fragment – dieses bleibt beim User-Agent).

E. Diese Ressource liefert dann eine Webseite mit eingebettetem Script zurück, dass die komplette Weiterleitungs-URI mit dem integrierten Access Token enthält.

F. Der User-Agent extrahiert aus dem Script das Access Token.

G. Das Access Token wird an den Client übergeben.

4.2.5.3 Resource Owner Password Credentials Grant

Dieser Typ der Autorisation wird dann empfohlen, wenn der Resource Owner dem Client voll und ganz vertrauen kann, wie zum Beispiel bei einem Betriebssystem. Jedoch sollte der OAuth Server diesen Typ nur dann zulassen, wenn dies die letzte Möglichkeit ist. Für den Client ist es möglich, bei diesem Typ die Anmeldedaten des Benutzers zu bekommen. Häufig wird dieser Typ auch zum Migrieren von HTTP Basic Authentication oder HTTP Digest Authentication verwendet, in dem sich der Resource Owner am Authorization Server mit seinen Anmeldedaten anmeldet, und der Authorization Server dann ein Access Token zurücksendet, welcher dann statt den Anmeldedaten verwendet wird (siehe Abb. 13). (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Resource Owner Password Credentials Grant, 2012))

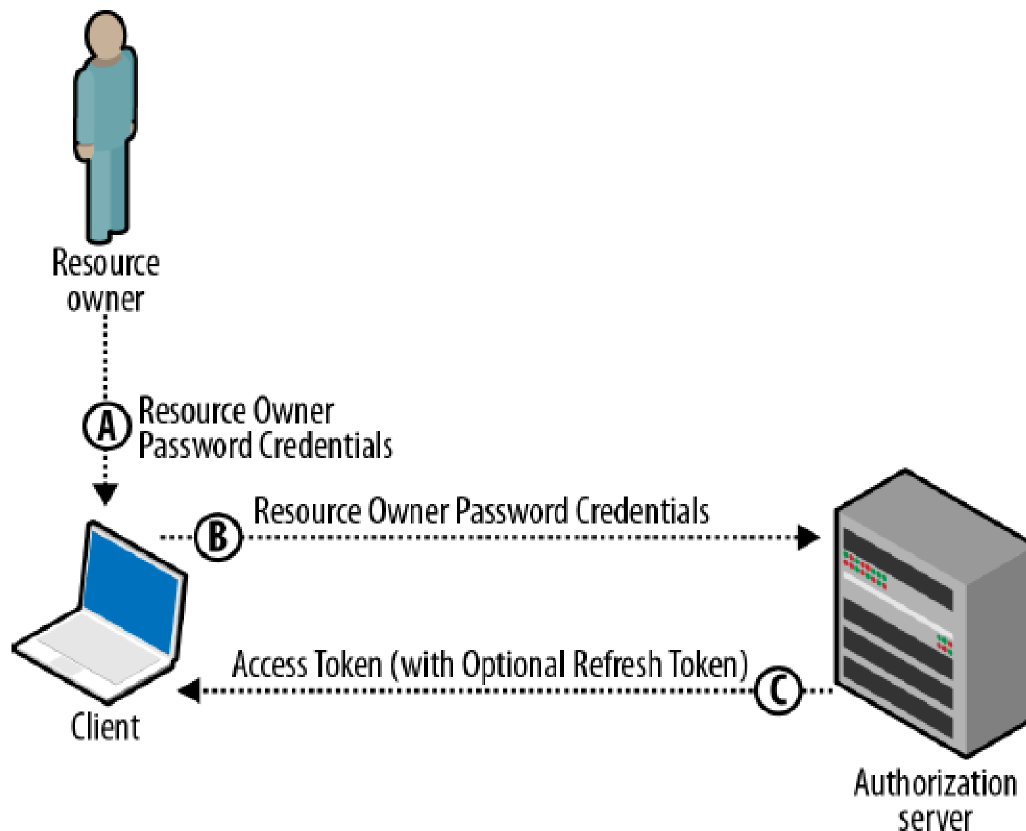

Abbildung 5: Resource Owner Password Credentials Grant (Boyd, 2012, S. 35)

A. Der Resource Owner übergibt dem Client seine Anmeldedaten
B. Der Client fordert einen Access Token vom Authorization Server an, in dem er die Anmeldedaten vom Resource Owner inkludiert.

C. Der Authorization Server authentifiziert den Client und überprüft die Anmeldedaten des Resource Owner – hat alles geklappt, wird ein Access Token zurückgesendet.

4.2.5.4 Client Credentials Grant

Beim Client Credentials Grant Typ kommuniziert nur der Client mit dem Authorization Server. Der Client erhält den Access Token, in dem er sich mit seinen Anmeldedaten am Authorization Server authentifiziert. Dieser Typ ist nur bei vertrauenswürdigen Clients zu verwenden. Ein gutes Beispiel für diesen Typ ist, wenn der Client auch der Resource Owner ist, zum Beispiel wenn ein REST-Backend programmiert wird und der Client auf das Backend zugreifen muss um zu funktionieren. (vgl. (The Internet Engineering Task Force, The OAuth 2.0 Authorization Framework – Client Credentials Grant, 2012))

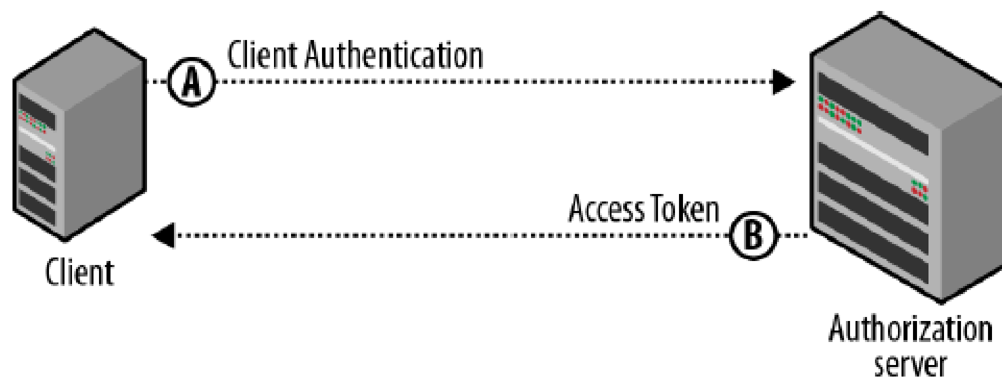

Abbildung 6: Client Credentials Grant (Boyd, 2012, S. 41)

Abbildung 14 zeigt folgendes Prozedere:

A. Der Client authentifiziert sich mit dem Authorization Server und fordert ein Access Token an.
B. Falls die Anmeldung erfolgreich war, sendet der Authorization Server ein Access Token zurück.

4.2.5.5 *Fazit*

Da es das ursprüngliche Ziel war, die REST API abzusichern und einen sicheren Weg zu finden, die API dennoch anderen Entwicklern und Entwicklerinnen bereitzustellen, ist der Authorization Code Grant Typ genau der, nachdem gesucht wurde. Die anderen Authentifizierungs- und Autorisierungsverfahren wurden komplett ausgeschlossen, da bei jenen immer die Anmeldedaten des Benutzers oder der Benutzerin benötigt werden. Ein weiterer Vorteil von OAuth ist, dass durch die Scopes nur bestimmte Bereiche freigegeben werden. Einen Scope kann man sich am besten wie eine Kategorie vorstellen. Es gibt zum Beispiel zwei Ressource URIs – eine um Benutzerinformationen zu bekommen und eine um Einträge zu verfassen. Hierbei wird die Ressource URI für die Benutzerinformation mit dem Scope „*user*“ definiert und für das Verfassen von Einträgen wird der Scope „*post*“ vergeben. Wenn jetzt jemand eine Applikation entwickelt und beide Scopes benötigt, wird (meistens) beim Autorisieren ein Hinweis angezeigt (siehe Abbildung 15), welche Informationen die Applikation lesen oder verändern darf.

Allow access?

The App **Test Client** would like to access some of your data.

Following scopes would be exposed to the client: **user**

Client ID: d1CZliGw50oyp1kOqUIHVK3dVa3IPFRuSnZ0X42V

User: test1

Abbildung 7: Autorisierungs Hinweis

4.3 OAUTH PROVIDER PROTOTYPE

Der OAuth Provider wurde mit Hilfe der Erweiterung Flask-OAuthlib realisiert. Diese Erweiterung übernimmt das Generieren der Tokens und unterstützt Entwickler/innen mit hilfreichen Funktionen zur Handhabung mit OAuth. Unter anderem gehören dazu Funktionen, die Bereiche (siehe Kapitel 4.3.1) kennzeichnen und diverse *getter* und *setter* für das Holen und Setzen der Tokens.

Zuerst werden zwei Datenbank-Relationen benötigt, die im Prototyp mit der Hilfe von SQLAlchemy angelegt werden. Im Prototyp wird auch noch eine dritte Relation *Grant* verwendet, die aber optional ist und auch mit diversen Caching-Optionen gelöst werden kann.

Das Client-Model, welches in der Datenbank als *clients* abgebildet ist, enthält die eindeutige Identifikation des Clients, das Client-Secret, die Redirect URI und den Standard Bereich, auch *Scope* genannt. Zudem gibt es auch einen Fremdschlüssel der auf den Entwickler oder die Entwicklerin verweist. Es sind noch weitere zwei Attribute inkludiert – ein Attribut für den Namen des Clients und ein weiteres für eine Beschreibung des Clients.

Die Datenbank-Relation *tokens* wird durch das Model *Token* dargestellt. Es enthält die Identifikation des Clients, ein Attribut für das Access Token und eines für das Refresh Token, die unbedingt eindeutig sein müssen, ein Ablaufdatum, und die Scopes die das Token gewährt. Auch ist dabei ein Fremdschlüssel für den Resource Owner zu berücksichtigen.

Das optionale Model Grant dient nur als Zwischenspeicher für das kurzlebige Grant-Token. Es wird während dem Autorisierungsverfahren angelegt und danach wieder gelöscht. Der Performance wegen wäre es besser, dieses in einem Cache zu speichern.

Damit die Bibliothek die Token setzen und abfragen kann, ist es erforderlich diese *setter* und *getter* zu implementieren. Es wird ein Client-*getter* benötigt um zu erfahren welcher Client die Anfrage stellt. Danach bedarf es noch des Access und Refresh Token *setter/getter*. Ersteres setzt einen Token und letzteres gibt einen Token zurück, abhängig davon, ob ein Access Token oder Refresh Token benötigt wird.

Ein Eintrag in der *token*-Tablle kann folgendermaßen aussehen:

```
{
    'access_token': '6JwgO77PApxsFCU8Quz0pnL9s23016',
    'refresh_token': '7cYSMmBg4T7F4kwoWfUQA99J8yqjp0',
    'token_type': 'Bearer',
    'expires_in': 3600,
    'scope': 'user admin'
}
```

Abbildung 8: Serverantwort - Tokens

Das Hauptaugenmerk liegt auf den zwei Endpunkten ohne denen nichts funktionieren würde. Als erstes wäre hier der *Authorization Handler* zu nennen – der *Resource Owner* wird auf die Webseite des Servers weitergeleitet und muss nun bestätigen, ob der Client seine Ressourcen benutzen darf. Falls der Benutzer oder die Benutzerin nicht authentifiziert sind, werden sie auf eine Login-Seite geleitet und nach erfolgreichem Einloggen wieder zurückgeleitet (siehe Beispiel darunter).

```
@oauth.route('/authorize', methods=['GET', 'POST'])
@login_required
@oauth_provider.authorize_handler
def authorize(*args, **kwargs):
    if request.method == 'GET':
        client_id = kwargs.get('client_id')
        client = Client.query.filter_by(client_id=client_id).first()
        kwargs['client'] = client
        kwargs['user'] = current_user
        return render_template('oauth/authorize.html', **kwargs)

    confirm = 'yes' == request.form.get('confirm', 'no')
    return confirm
```

Listing 10: Authorize Endpunkt

Bei bestätigen des HTML Formulars, wird die *client_id*, eine Liste der Scopes (getrennt mit Leerzeichen, wie es der RFC besagt), der Status, die Redirect URI und zu guter letzt, der Response Typ an den Autorisierungsserver gesendet.

Der nächste Endpunkt ist der Token Handler. Er wird verwendet, um Access Token auszutauschen oder eben um diese zu verlängern. Hierbei übernimmt alles der Dekorator – optional können zusätzliche Informationen zurückgegeben werden.

```
@oauth.route('/token', methods=['GET', 'POST'])
@oauth_provider.token_handler
def access_token():
    return None
```

Listing 11: Token Endpunkt

4.3.1 RESSOURCEN BEREICHE

Ressourcen-Bereich, oder auch Scopes genannt, sind dazu da, um genaue Bereiche zu definieren, auf die der Client Zugriff hat. Sie bieten keine extra Berechtigungen zusätzlich zu jenen, die der Benutzer oder die Benutzerin ohnehin schon besitzt. Normalerweise werden die Scopes auf der Autorisierungs-Seite aufgelistet um den Benutzer oder die Benutzerin zu benachrichtigen, welche Befugnisse der Client mit Zustimmung bekommt.

```
@oauth_provider.require_oauth("user")
def get(self, user_id):
    user = User.query.filter_by(id=user_id).first_or_404()
    return {'user': marshal(user, user_fields)}
```

Listing 12: "user"-Scope Beispiel

Mittels eines Dekorators wird die Ressource URI geschützt. Um nun an diese Ressource zu kommen, benötigt der Client den Scope „*user*“.

4.4 OAUTH CLIENT PROTOTYPE

Zum Testen der REST API wurde ein einfacher Prototyp erstellt. Der Prototype verwendet dabei einen OAuth Client, um sich mit dem OAuth Provider zu verbinden. Hierbei geht es lediglich darum, zu veranschaulichen, wie ein OAuth Client erstellt werden kann. Da OAuth ein anerkannter Standard ist, ist es möglich, durch Austausch einiger Optionen den Client für andere Services zu verwenden – vorausgesetzt das Service ist dem OAuth Standard treu. Es kann ein OAuth Client für eine breite Anzahl von Programmiersprachen entwickelt werden und ist nicht nur für Python verfügbar.

Als erstes muss beim OAuth Provider der Client registriert werden. Hierfür wurde in den Benutzereinstellungen bei FlaskBB eine weitere Option eingefügt in welcher dem Client ein Name gegeben und eine Beschreibung hinzugefügt werden kann. Bei Absenden dieses Formular wird automatisch eine eindeutige Identifikation für den Client generiert. Auch erhält der Client ein Client Secret. Diese beiden Werte müssen nun beim Client eingetragen werden. Auch benötigt der Client die Basis-URL der REST API, die URL wie er an das Access Token kommt und die URL für die Autorisierung. Außerdem sollte auch eine Liste mit den benötigten Scopes eingestellt werden.

```
remote = oauth.remote_app(
  'remote',
  consumer_key='u06X2qzc5KW5vgKwGP29avEH5OtpJynLOGqxuIaU',
  consumer_secret='zBArOaBonuHt6aCycsFmSX7RAq79VYYwZiO0eSg7eRN6ht',
  request_token_params={'scope': 'user'},
  base_url='http://localhost:8080/api/v1/',
  request_token_url=None,
  access_token_url='http://localhost:8080/oauth/token',
  authorize_url='http://localhost:8080/oauth/authorize'
)
```

Listing 13: Client Konfiguration

Ist der Client noch nicht berechtigt, wird auf der Hauptseite folgende Funktion aufgerufen:

```
remote.authorize(
    callback=url_for('authorized', next=next_url, _external=True)
)
```

Listing 14: Client Callback

Hiermit wird der Client auf die Seite, die beim obigen Parameter *authorize_url* angegeben ist, weitergeleitet und wenn die Berechtigung gewährt wurde, wird er auf den Client-Endpunkt *authorized* weitergeleitet. Der *authorized* Endpunkt überprüft lediglich ob die Autorisierung erfolgreich war.

Wenn die Autorisierung erfolgreich war kann ganz bequem auf die Ressourcen des Resource Owners zugegriffen werden.

```
resp = remote.get('users')
```

Listing 15: Zugriff auf Benutzerdaten

5 ZUSAMMENFASSUNG

Dieses Kapitel gibt einen kleinen Überblick über die in Erfahrung gebrachten Ergebnisse und über mögliche Verbesserungen. Außerdem soll die ursprüngliche Frage – ob es sinnvoll ist, eine REST API in eine Foren Software zu integrieren – beantwortet werden.

5.1 ERFAHRUNGEN UND ERGEBNISSE

Es ist auf jeden Fall sinnvoll, eine REST API in eine Foren Software zu integrieren – dies gilt nicht nur für FlaskBB, sondern generell für Software, die vor allem auf die Interaktion von Benutzern oder Benutzerinnen angewiesen ist. Dadurch erhalten Drittentwickler oder Drittentwicklerinnen die Möglichkeit, die API zu nutzen, um eigene Applikationen, seien es Apps fürs Smartphones oder Web Apps, zu erstellen.

Für das Autorisieren und Authentifizieren ist OAuth einer der sichersten Methoden weil OAuth eine verschlüsselte Verbindung zwischen Client und Server voraussetzt. Ein weiterer Vorteil von OAuth ist, dass die Anmeldedaten der Benutzer oder Benutzerinnen nicht benötigt werden. Sie müssen sich sich nur beim OAuth-Provider anmelden und können über diesen, andere Dienste oder Applikationen verwenden.

Das Verknüpfen von OAuth und einer REST API ist eine gute Kombination, um Benutzer oder Benutzerinnen die Möglichkeit zu geben, die Verwendung ihrer Daten zu kontrollieren. Durch die einzelnen Autorisierungsverfahren, die OAuth bietet, ist für so gut wie jeden Anwendungsfall etwas dabei.

5.2 MÖGLICHE VERBESSERUNGEN UND WEITERENTWICKLUNGEN

Im weiteren Verlauf der Entwicklung soll die REST API finalisiert und verfeinert werden. Hinzu kommt, dass mehr Ressourcen zugänglich gemacht werden sollen. Im weiteren Verlauf sollen die Scopes mit dem Berechtigungssystem von FlaskBB angepasst werden.

Mit der Serialisierungs-Methode von *Flask-RESTful* stößt man bei FlaskBB bald an die Grenzen, da es zu komplexeren Strukturen kommt. Deshalb wäre es von Vorteil diese mit einer besseren zu ersetzen. Eine mögliche Bibliothek die sowohl die

Serialisierung und Validierung von REST-Argumenten beherrscht, wäre *marshmallow*. Dazu muss aber zuvor untersucht werden, in wie fern diese Bibliothek sich von der Funktionalität von *Flask-RESTful* unterscheidet.

Weiters soll gegen Ende des Jahres eine FlaskBB Version (1.0) veröffentlicht werden, die eine RESTful API mit OAuth Unterstützung, besitzt. Außerdem soll nach dieser Veröffentlichung die Arbeit an einer neuen Version mit Verbesserungen und begonnen werden.

6 LITERATURVERZEICHNIS

Bayer, M. (2015). *SQLAlchemy*. Abgerufen am 27. April 2015 von sqlalchemy.org: http://sqlalchemy.org

Boyd, R. (2012). *Getting Started with OAuth 2.0* (1. Ausg.). O'Reilly Media Inc.

Cesare, P., Olaf, Z., & Frank, L. (2008). RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision. *WWW '08 Proceedings of the 17th international conference on World Wide Web* , 805-814.

Davis, C. (2012). What if the Web Were not RESTful? *WS-REST '12 Proceedings of the Third International Workshop on RESTful Design* , 3-10.

Deutschl, M. (2012). *Empirische Analyse von Javascript-Frameworks.*

ECMA International. (Oktober 2013). ECMA-404: The JSON Data Interchange Format. (1).

Grinberg, M. (2014). *Flask Web Development* (1. Ausg.). O'Reilly Media Inc.

JSON. (2015). *json.org*. Abgerufen am 20. April 2015 von json.org: json.org

Justin, P. (2015). *FlaskBB - Features*. Abgerufen am 23. April 2015 von FlaskBB: flaskbb.org

Jython. (2015). Abgerufen am 12. April 2015 von Jython Documentation: https://wiki.python.org/jython

Millman, K. J., & Aivazis, M. (2011). Python for Scientists and Engineers. *Computing in Science & Engineering* , 9-12.

Peter, W., Jeffrey, E., Allen B., D., & Chris, M. (2012). *How to Think Like a Computer Scientist* (3 Ausg.).

Picard, R. (2014). *Explore Flask.*

Pilgrim, M. (2004). *Dive Into Python.* Apres.

Python Software Foundation, P. (2015). *ctypes - A foreign function library for python*. Abgerufen am 20. April 2015 von docs.python.org: https://docs.python.org/2/library/ctypes.html

Python Software Foundation, P. (2015). *FAQ*. Abgerufen am 12. April 2015 von Python Documentation: https://docs.python.org/2

Python Software Foundation, P. (2008). *PEP 373 - Python 2.7 Release Schedule*. Abgerufen am 25. April 2015 von python.org: https://www.python.org/dev/peps/pep-0373/#id2

Python Software Foundation, P. (2015). *Python*. Abgerufen am 20. April 2015 von Python: https://www.python.org/

Richardson, L., & Ruby, S. (2007). *RESTful Web Services* (1. Ausg.). O'Reilly Media Inc.

Rieber, P. (2009). *Dynamische Webseiten in der Praxis* (2. Ausg.). mitp.

Ronacher, A. (2015). *Flask*. Abgerufen am 10. 04 2015 von flask.pocoo.org: http://flask.pocoo.org

Ronacher, A. (2015). *Flask Documentation*. Abgerufen am 21. April 2015 von flask.pocoo.org: http://flask.pocoo.org/docs/0.10/

Ronacher, A. (2015). *Flask-SQLAlchemy*. Abgerufen am 24. April 2015 von Flask-SQLAlchemy: https://pythonhosted.org/Flask-SQLAlchemy/quickstart.html

Ronacher, A. (2015). *Werkzeug*. Abgerufen am 19. April 2015 von werkzeug.pocoo.org: http://werkzeug.pocoo.org

Rossum, G. V. (February 1991). *HISTROY*. Abgerufen am 12. April 2015 von Python Mercurial Repository: https://hg.python.org/cpython/file/08adaaf08697/Misc/HISTORY

Rossum, G. V. (kein Datum). *Personal Home Page*. Abgerufen am 20. April 2015 von Guido Van Rossum: https://www.python.org/~guido/

The Internet Engineering Task Force, I. (Juni 1999). *HTTP Authentication: Basic and Digest Access Authentication*. Abgerufen am 18. April 2015 von ietf.org: https://tools.ietf.org/html/rfc2617

The Internet Engineering Task Force, I. (Juni 1999). *Hypertext Transfer Protocol -- HTTP/1.1*. Abgerufen am 6. Juni 2015 von ietf.org: http://tools.ietf.org/html/rfc2616

The Internet Engineering Task Force, I. (March 2011). *MD5 and HMAC-MD5 Security Considerations*. Abgerufen am 24. April 2015 von ietf.org: https://tools.ietf.org/html/rfc6151

The Internet Engineering Task Force, I. (März 2010). *PATCH Method for HTTP*. Abgerufen am 4. Juni 2015 von ietf.org: http://tools.ietf.org/html/rfc5789

The Internet Engineering Task Force, I. (Oktober 2010). *The OAuth 2.0 Authorization Framework*. Abgerufen am 20. April 2015 von ietf.org: http://tools.ietf.org/html/rfc6749

The Internet Engineering Task Force, I. (Oktober 2012). *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Abgerufen am 19. April 2015 von ietf.org: https://tools.ietf.org/html/rfc6750

Theis, T. (2011). *Einstieg in Python* (3. Ausg.). Galileo Computing.

W3C. (20. April 2015). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Von w3.org: http://www.w3.org/TR/2008/REC-xml-20081126 abgerufen

W3C. (5. April 2012). *XML Schema Definition Language (XSD)*. Von www.w3.org: http://www.w3.org/TR/xmlschema11-1/ abgerufen

Weigand, M. (2010). *Objektorientierte Programmierung mit Python 3* (4. Ausg.). mitp.

Yating, H., & David, L. (2010). Authentication and Authorization Protocol Security Property Analysis with Trace Inclusion Transformation and Online Minimization. *Network Protocols (ICNP), 2010 18th IEEE International Conference* , 164-173.

7 ABBILDUNGSVERZEICHNIS

ABBILDUNG 1: SERVERANTWORT FÜR AUTHENTIFIZIERUNG 29
ABBILDUNG 2: CLIENTANFRAGE FÜR AUTHENTIFIZIERUNG 30
ABBILDUNG 3: AUTHORIZATION CODE GRANT (BOYD, 2012, S. 14) 33
ABBILDUNG 4: IMPLICIT GRANT (BOYD, 2012, S. 30) 35
ABBILDUNG 5: RESOURCE OWNER PASSWORD CREDENTIALS GRANT (BOYD, 2012, S. 35) 37
ABBILDUNG 6: CLIENT CREDENTIALS GRANT (BOYD, 2012, S. 41) 38
ABBILDUNG 7: AUTORISIERUNGS HINWEIS 39
ABBILDUNG 8: SERVERANTWORT - TOKENS 41

8 LISTINGVERZEICHNIS

LISTING 1: GET-ANFRAGE 10
LISTING 2: XML BEISPIEL..... 12
LISTING 3: JSON BEISPIEL 13
LISTING 4: FLASK - USERLISTAPI "GET" 26
LISTING 5: FLASK - REST "PUT" 27
LISTING 6: PARAMETERVALIDIERUNG..... 28
LISTING 7: FLASK-RESTFUL FELDER..... 28
LISTING 8: GENERIERTES JSON OBJEKT 29
LISTING 9: PASSWORTVALIDIERUNG..... 30
LISTING 10: AUTHORIZE ENDPUNKT 41
LISTING 11: TOKEN ENDPUNKT 41
LISTING 12: "USER"-SCOPE BEISPIEL 42
LISTING 13: CLIENT KONFIGURATION 43
LISTING 14: CLIENT CALLBACK..... 43
LISTING 15: ZUGRIFF AUF BENUTZERDATEN..... 43

9 ABKÜRZUNGSVERZEICHNIS

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

ORM – Object Relational Mapper

REST – Representational State Transfer

API – Application Program Interface

JSON – JavaScript Object Notation

XML – Extensible Markup Language