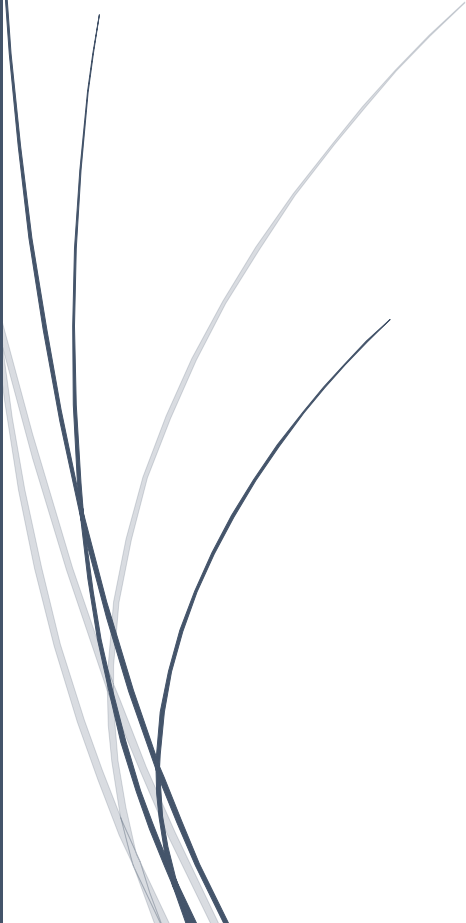


A dark blue vertical bar runs down the left side of the slide. A blue arrow points to the right from this bar, containing the date.

3.12.2015

RESTful Services

Geyer Stefan, Ritter Mathias
5BHIT

Several thin, dark blue curved lines originate from the bottom left corner and sweep upwards and to the right, creating a dynamic, abstract graphic element.

Inhaltsverzeichnis

1	Einleitung.....	4
2	Representational State Transfer	4
2.1.1	Definition und Zweck	4
2.2	Grundprinzipien.....	5
2.2.1	Eindeutige Identifikation	5
2.2.2	Hypermedia	5
2.2.3	Standardmethoden	6
2.2.4	Ressourcen und Repräsentationen	6
2.2.5	Statuslose Kommunikation	6
2.3	Verbreitete Repräsentationsformate.....	7
2.3.1	JSON	7
2.3.2	XML.....	7
2.3.3	Weitere Formate	7
2.4	Security.....	7
2.4.1	API-Keys versus Username und Passwort	8
2.5	Maturity Model	8
2.5.1	Level 0 – Plain Old XML	8
2.5.2	Level 1 – Ressourcen	9
2.5.3	Level 2 – HTTP Verben	10
2.5.4	Level 3 – Hypermedia	10
2.6	Schnittstellendesign.....	11
2.6.1	URI Design	11
2.6.2	Versionierung	11
2.6.3	Error-Handling	12
2.6.4	Repräsentationsformate	12
2.7	Implementierungen	13
2.7.1	Spring.....	13
2.7.2	Django	13
2.7.3	Lightweight-Umsetzungen	14
3	JavaScript Object Notation	15
3.1	Syntax	15
3.1.1	Objekte	15
3.1.2	Arrays	15
3.1.3	Datentypen.....	16
3.2	Anwendungen von JSON	17
3.2.1	JSON in Java	17
3.2.2	JSON in anderen Programmiersprachen	17
4	HTTP 2.0	18
4.1	Einleitung	18
4.1.1	Probleme von HTTP 1.1	18
4.1.2	Ziele	18
4.1.3	SPDY.....	18
4.2	Aufbau & Verbindung	19
4.2.1	Streams.....	19
4.2.2	Messages	19
4.2.3	Frames.....	19
4.3	Neue Funktionalität	20
4.3.1	Multiplexing	20
4.3.2	Header Compression	20

4.3.3	Server Push	21
4.3.4	Stream Priority	21
4.3.5	Flow Control	21
4.4	Methoden	22
4.4.1	POST	22
4.4.2	GET	22
4.4.3	PUT	22
4.4.4	DELETE	22
4.4.5	OPTIONS	22
4.5	Statuscodes	23
4.5.1	Informationen	23
4.5.2	Erfolgreiche Operation	23
4.5.3	Umleitung	23
4.5.4	Client-Fehler	23
4.5.5	Server-Fehler	23
4.6	Implementierungen	24
4.6.1	Server	24
4.6.2	Browser	24
5	OAuth 2.0	25
5.1	Einleitung	25
5.1.1	Definition & Zweck	25
5.1.2	Unterschiede zu OAuth 1	25
5.1.3	Client-Profile	25
5.2	Rollen	26
5.2.1	Resource Owner	26
5.2.2	Client Applikation	26
5.2.3	Resource Server	26
5.2.4	Authorization Server	26
5.3	Authorization Flows	27
5.3.1	Authorization Code Flow	27
5.3.2	Implicit Grant Flow	28
5.3.3	Ressource Owner Password Flow	29
5.3.4	Client Credentials Flow	30
5.4	Anwendungen bzw. Implementierungen	30
5.4.1	Apache Oltu	30
5.4.2	Spring Security OAuth	31
5.4.3	Client-Libraries für Google	31
5.4.4	Client-Libraries für Facebook	31
5.4.5	Beispiel einer Implementierung des Authorization Code Flows	31
6	Zusammenfassung & Schlussfolgerung	32
7	Literaturverzeichnis	33
8	Abbildungsverzeichnis	36

1 Einleitung

Die heute gängigen Übertragungsstandards erweitern deren Nachrichten um eigene Header oder verwenden eigene Protokolle. Aber warum alles selber implementieren, wenn bereits sehr gute Standards existieren? Der Representational State Transfer nutzt HTTP zur Datenübertragung und löst so die genannten Problemstellungen. Zur Repräsentation der zu sendenden Daten wird oft die JavaScript Object Notation verwendet.

Für Kommunikation im Web wird großteils HTTP genutzt. Doch auch dieses existiert in der Form, wie wir es kennen, seit vielen Jahren. Mit HTTP 2.0 werden einige neue Features eingeführt, welche einfachere und schnellere Kommunikation ermöglichen.

Authentifizierung ist ebenfalls ein großes Thema heutzutage. Es wäre wünschenswert, bereits existierende Plattformen für selbst entwickelte Systeme zu nutzen. Mit OAuth und speziell dem Release 2.0 stehen viele neue Funktionen zur Verfügung.

2 Representational State Transfer

Es gibt mittlerweile viele Architekturen, welche Kommunikation zwischen mehreren Systemen ermöglichen. Heutzutage sind Begriffe wie Remote Procedure Call, CORBA, RMI oder andere Webservices den meisten Entwicklern geläufig. In dieser Ausarbeitung wird allerdings nur auf einen Architekturstil weiter eingegangen. Dieser entstand im Jahr 2000 und hört auf den Namen Representational State Transfer.

2.1.1 Definition und Zweck

"Warum REST?" – Das ist vermutlich eine der am häufigsten gestellten Fragen zu diesem Thema. Im folgenden Abschnitt werden einige bekannte Problemstellungen aufgelistet und beschrieben, wie REST diese bearbeitet.

2.1.1.1 Lose Kopplung

Ein wichtiger Punkt ist, dass Systeme möglichst unabhängig voneinander sein sollten. Manchmal ist es zwar erwünscht, dass zwei Systeme so eng koppeln, dass sie quasi verschmelzen, doch in den meisten Fällen ist es wünschenswert, dass ein System so selbstständig wie möglich agiert. REST arbeitet mit dem HTTP-Protokoll, welches auf Anfrage und Antwort basiert. Über HTTP ist es einem Client möglich, entkoppelt Nachrichten zu senden und zu empfangen. [15]

2.1.1.2 Interoperabilität

Interoperabilität bezeichnet die Kommunikation verschiedener Systeme und die damit verbundenen Kommunikationsstandards. Gerade in dieser Hinsicht eignen sich HTTP und die damit verbundenen Webformate wie XML oder HTML besonders, da sie allgemein im Web gültig sind. Benutzt man selbstdefinierte Protokolle oder eine Architektur wie CORBA, RMI, usw. blockiert man eventuell die Kommunikation zu anderen Systemen, die diese nicht unterstützen. [15]

2.1.1.3 Wiederverwendbarkeit

Das Ziel vieler Software-Anwendungen ist es, wiederverwendbar zu sein. Das bedeutet, dass die Software auch außerhalb dessen, wofür diese eigentlich entwickelt wurde, verwendet werden kann. Ein typisches Problem ist, dass man meist die späteren Anforderungen nur mangelhaft vorhersehen kann. Viele der bereits genannten Technologien basieren darauf, eine klar definierte Schnittstelle bereitzustellen. Bei REST gibt es genau eine Schnittstelle welche generell über HTTP aufrufbar ist. Jeder Client, der also eine REST Schnittstelle verwenden kann, kann auch alle anderen verwenden. [15]

2.2 Grundprinzipien

REST-Architekturen sind zwar nach wie vor nicht standardisiert, jedoch gibt es einige Grundprinzipien, an welche beim Erstellen einer solchen Schnittstelle einhalten sollte.

2.2.1 Eindeutige Identifikation

Die Identifikation im Web erfolgt über ein einheitliches Konzept, welches jedem Entwickler bekannt sein dürfte – den Uniform Resource Identifier (URI). Durch sie können Elemente im Web weltweit eindeutig identifiziert werden. Das bringt den Vorteil mit sich, dass der Entwickler kein eigenes Schema für Namen erfinden muss, sondern das bereits bestehende verwenden kann. Es ist einfach, Datenstrukturen über einen gut strukturierten URI anzusprechen. Standardmäßig definiert man eine Ressource und fügt dieser noch einen Identifier hinzu, damit klar ist, auf welches Objekt Referenziert wird. Das könnte beispielsweise so aussehen:

```
http://api.domain.com/products/12345  
http://api.domain.com/products
```

Aber wie genau man einen solchen URI definieren sollte, wird später genauer erklärt. [2, 15]

2.2.2 Hypermedia

Das nächste Grundprinzip ist Hypermedia. Dabei handelt es sich um Verknüpfungen von Daten. Ohne Verknüpfungen wäre das Web nicht das, was es heute ist. Jede Website besitzt Links, welche auf eine andere Seite verweisen. Ebenso kann dieses Prinzip bei einer REST-Schnittstelle angewandt werden. Beispielsweise könnte ein Objekt über einen Link auf weitere Daten verweisen anstatt sie selbst mitzuliefern. Selbstverständlich ist das nicht die einzige Möglichkeit. Anstelle eines Links könnte auch einfach nur eine Integer-ID stehen. Dieses Konzept könnte umgesetzt beispielsweise so aussehen:

```
{  
  "product": {  
    "href": "http://api.domain.com/products/12345"  
  }  
}  
  
{  
  "product": {  
    "id": 12345  
  }  
}
```

Listing 1: Zwei Responses, die das Hypermedia Konzept umsetzen

Doch dieses Konzept bietet noch Raum für mehr. Einerseits können zusätzliche Daten übergeben werden, andererseits kann es als eine Art "nächster Schritt" verwendet werden. Der Server kann dem Client dadurch mitteilen, welchen Link er als nächstes aufrufen soll.

Zusammengefasst kann man sagen, dass Hypermedia dazu verwendet werden kann, Beziehungen zwischen verschiedenen Ressourcen herzustellen. Das sollte beim Erstellen einer REST-API unbedingt beachtet und wo immer es möglich ist angewandt werden. [2, 15]

2.2.3 Standardmethoden

Wie kann man bestimmen, was mit den gesendeten Daten passiert? Wie kann man bestimmen, ob Daten erstellt, gelesen oder gelöscht werden sollen? REST bietet einen einfachen Ansatz. Das CREATE-READ-UPDATE-DELETE-Prinzip, kurz CRUD, relationaler Datenbanksysteme wird übernommen. Dieses kann nun über HTTP umgesetzt werden. Dafür werden verschiedene Methoden, manchen auch bekannt als Verben, verwendet. Für jede CRUD-Operation gibt es eine entsprechende Methode. Diese werden im Punkt 4.4 genauer beschrieben. [15]

2.2.4 Ressourcen und Repräsentationen

Damit der Client auch weiß, wie er seine Daten verarbeiten soll, benötigt dieser genauere Informationen. Um sicherzustellen, dass die Daten, die als HTTP-Response empfangen werden, auch einem Standard entsprechen, kann über den Header ein spezielles Format angefordert werden. Dieser könnte wie folgt aussehen:

```
GET products/12345 HTTP/1.1
Host: api.domain.com
Accept: application/json
```

Wenn der Server diese Formate unterstützt, ist es für den Client ein leichtes, je nach Bedarf das notwendige Format anzufordern. Kann man die Daten beispielsweise auch als HTML-Dokument abfragen, so sind diese in jedem Browser darstellbar. [15]

2.2.5 Statuslose Kommunikation

HTTP basiert auf einem statuslosen Prinzip. Das bedeutet, dass sich der Server nur für die Dauer eines Requests für den Client interessiert und verwirft danach jegliche Informationen, die er über diesen Client hat. Dadurch gibt es keine Bindung zwischen Server und Client. Deshalb müssen jegliche Informationen am Client aufbewahrt werden.

Außerdem ist dieses System viel skalierbarer. Requests können von verschiedenen Serverinstanzen bearbeitet werden. Das ist nur möglich, wenn zwischen einem Server und dem Client keine Bindung existiert. Des Weiteren können so auch bei einem Serverausfall keine für die Kommunikation notwendigen Daten verloren gehen. Wenn jemand einen URI nach einem Serverneustart aufruft, bekommt dieser trotzdem die gleiche Antwort wie zuvor. [15]

2.3 Verbreitete Repräsentationsformate

Bei einer REST-Schnittstelle können Daten auf viele Arten dargestellt werden. Welche und wie viele verwendet werden, ist unterschiedlich. Dennoch gibt es einige, weiter verbreitete Formate. Die wichtigsten werden nun angeführt.

2.3.1 JSON

Die JavaScript Object Notation ist wohl das am häufigsten verwendete Repräsentationsformat, das im Bereich REST eingesetzt wird. [15] JSON wird im Punkt 0 genauer beschrieben.

2.3.2 XML

Obwohl mittlerweile JSON das am meisten verwendete Repräsentationsformat ist, kann die Nutzung von XML dennoch einige Vorteile mit sich bringen. Einerseits kann es sein, dass ältere Systeme eher XML als JSON unterstützen. Andererseits bringt XML eine standardisierte Schemasprache und die Möglichkeit, ein Dokument gegen diese zu validieren, mit sich. Für XML wurde der Medientyp "application/xml" definiert. Dieser sagt allerdings nichts über das verwendete Vokabular aus, sondern nur, dass das XML Dokument syntaktisch korrekt ist. Für ein festgelegtes Vokabular können eigene Medientypen definiert werden. Diese können über ein "application/<name>+xml" erstellt werden. [15]

2.3.3 Weitere Formate

2.3.3.1 CSV

Die Comma Separated Values sind ein sehr schlicht gehaltenes Repräsentationsformat. Die Werte werden, wie der Name schon sagt, einfach durch ein Kommazeichen getrennt. [15]

2.3.3.2 HTML

Die Hyper Text Markup Language ist bis auf sehr spezifische Fälle eher nicht als REST Response Format geeignet, da sie für den menschlichen Konsum gedacht ist und aufgrund der grafischen Aufbereitung langsamer vom Client verarbeitet wird. Wenn die REST-Schnittstelle aber darauf ausgelegt, ist einen grafischen Output zurückzuliefern, kann selbstverständlich HTML benutzt werden. [15]

2.4 Security

Authentifizierung ist selbstverständlich ein wichtiger Teilbereich des Themas REST. Oft will man nicht, dass jeder auf die Schnittstelle zugreifen kann, oder nur unter gewissen Voraussetzungen. Um Authentifizierung zu ermöglichen, wurden verschiedene Methoden entwickelt. Diese setzen natürlich eine Verschlüsselung voraus und sollten entweder über TLS (HTTPS) oder OAuth abgehandelt werden. Auch der Einsatz anderer Protokolle ist möglich. Darauf wird hierbei allerdings nicht weiter eingegangen.

Es muss einem User möglich sein, sich an der Schnittstelle zu registrieren, damit die dabei aufgefassten Daten später verglichen werden können. [15]

2.4.1 API-Keys versus Username und Passwort

Ein API-Key wird vom System, das die Schnittstelle bereitstellt, für einen User generiert. Dieser besteht aus einer festgelegten Anzahl aus zufälligen Zeichen und ist in diesem System einmalig. Nachdem dieser generiert wurde, wird er in einer gewissen Art und Weise mit dem User verknüpft (zum Beispiel mit einem User Account am System). Den Key kann der User nun bei jedem Request mitschicken und sich darüber authentifizieren.

Die zweite Möglichkeit sich an einer Schnittstelle zu authentifizieren wäre, einfach den Usernamen und das Passwort bei jedem Request mitzuliefern.

Diese beiden Varianten werden im folgenden Abschnitt genauer verglichen. [33, 34]

2.4.1.1 Unabhängigkeit

Ein API-Key wird vom System generiert, unabhängig von den Account-Daten des Users. Das bringt einen großen Vorteil gegenüber der Passwort-Methode mit sich. Wenn der Nutzer beispielsweise sein Passwort zurücksetzt, würden alle Applikationen, die noch das alte Passwort zur Authentifizierung verwenden, nicht mehr ordnungsgemäß arbeiten. Das kann in einer Produktionsumgebung zu großen Problemen führen. [34]

2.4.1.2 Sicherheit

Passwörter bei Requests mitschicken ist äußerst riskant. Falls eine Nachricht abgefangen wird, fallen dem Angreifer wichtige Login-Daten in die Hände. Fällt einem Angreifer hingegen ein API-Key in die Hände, hat dieser somit nicht direkt Zugriff auf die Daten des Nutzers. In diesem Fall sollte der API-Key jedoch zurückgesetzt werden.

Ein weiterer Faktor ist die Länge des API-Keys. Während ein durchschnittliches Passwort vielleicht 8 bis 13 Zeichen lang ist, kann ein API-Key beispielsweise aus 40 Zeichen bestehen. Das macht einen Brute-Force Angriff wesentlich schwieriger. Es ist außerdem ratsam, keine fortlaufenden Nummern zu verwenden, sondern beispielsweise UUIDs. Dadurch wird die Wahrscheinlichkeit, dass ein Angreifer den Key errät, wesentlich verringert. [33]

2.5 Maturity Model

Um die Eigenschaften eines RESTful Webservices besser zu beschreiben, wurde von Leonard Richardson ein sogenanntes Maturity-Model definiert, welches den Aufbau einer solchen Schnittstelle schrittweise beschreibt. Es besteht aus 4 Stufen, welche aufeinander aufbauen.

2.5.1 Level 0 – Plain Old XML

Wenn man sich ein XML-RPC- oder SOAP-Service genauer ansieht, findet man etwas, das einer C-Bibliothek ähnelt. Es sind einige Funktionen verfügbar, welche alle einen POST-Request an exakt einen URI sendet. Hier wird nicht wirklich auf die Technologien im Web zugegriffen, sondern eher ein kleiner Teil für eine komplett andere Technologie genutzt.

Ein entsprechender Request-Header könnte demnach beispielsweise so aussehen:

```
POST /tgmTimeTable HTTP/1.1
```

```
<timeTableRequest date = "2015-10-11" student = "sgeyer"/>
```

Listing 2: XML Request

Auch die Response folgt demselben Schema:

```
HTTP/1.1 200 OK

<timeTable>
  <subject name="INSY" start = "0850" end = "1040">
    <teacher id = "MARM"/>
  </subject>
</timeTable>
```

Listing 3: XML Response

Im Gegensatz zu diesem Beispiel könnten Request und Response auch in JSON oder anderen Formaten gegeben sein. [31, 32]

2.5.2 Level 1 – Ressourcen

Dieses Prinzip wird seit einiger Zeit von vielen bekannten Firmen wie Amazon oder Flickr umgesetzt. Dennoch kann man nicht von einer vollwertigen REST-Umsetzung sprechen. Hier hat zwar jeder Befehl, also jede Methode, eine eigene URI, allerdings wird weiterhin nur HTTP POST zur Übertragung verwendet. Dadurch kann auf den ersten Blick nicht zwischen dem Erstellen und dem Löschen von Daten nach einem Request unterschieden werden.

Der Beispiel-Request ähnelt dem vorherigen zwar immer noch, aber dieses Mal wird nicht ein globaler URI verwendet. Hier hat jeder Slot seinen eigenen, welcher über einen eindeutigen Identifier ansprechbar ist. [31, 32]

```
POST /student/sgeyer/ HTTP/1.1

<timeTableRequest date = "2015-10-11">
```

Listing 4: Request mit XML Parametern

Die entsprechende Antwort sollte in etwa so aussehen:

```
HTTP/1.1 200 OK

<timeTable>
  <subject name="SEW" start = "0800" end = "0940">
    <teacher id = "MARM"/>
  </subject>
</timeTable>
```

Listing 5: XML Response

2.5.3 Level 2 – HTTP Verben

Dieses Prinzip erweitert Level 1 – die bereits vorhandenen URIs werden nun von mehreren HTTP Methoden unterstützt. Bei Requests kann je nach Abfrage eine andere Methode verwendet werden. So können im Header beispielsweise GET Variablen anstelle von XML verwendet werden. [31, 32]

Da es sich nur um eine Abfrage handelt, kann die GET Methode verwendet werden. Ein Request könnte beispielsweise so aussehen:

```
GET /students/sgeyer?date=20151013&start=0800 HTTP/1.1
```

Listing 6: GET Request

Die entsprechende Antwort:

```
HTTP/1.1 200 OK
```

```
<timeTable>
  <subject name="AM" start = "0800" end = "0940">
    <teacher id = "KRUC"/>
  </subject>
</timeTable>
```

Listing 7: GET Response

2.5.4 Level 3 – Hypermedia

Bei diesem Level wird das Response-Dokument um einen wichtigen Faktor erweitert. Wie bereits erwähnt, wird anstelle der tatsächlichen Daten eine Referenz auf das Dokument, welche diese enthält, zurückgeliefert. So kann ein Verweis auf Daten übertragen werden. Wenn benötigt, können die etwaigen Zusatzdaten via externer Requests übertragen werden. [31, 32]

```
GET /students/sgeyer?date=20151020&start=0800 HTTP/1.1
```

Listing 8: GET Request

Bei der Antwort findet man jetzt einen Verweis auf weitere Daten vor:

```
HTTP/1.1 200 OK
```

```
<timeTable>
  <subject name = "GGP" start = "0800" end = "0940">
    <link rel = "/linkrels/timetable/teacher" uri = "/teachers/KRAH0"/>
  </subject>
</timeTable>
```

Listing 9: GET Response

2.6 Schnittstellendesign

Damit eine Schnittstelle übersichtlich und einfach umzusetzen bleibt, sollten bei der Erstellung gewisse Kriterien beachtet werden. Mit der Zeit kamen immer mehr Probleme auf, welche von einigen klugen Köpfen bearbeitet wurden. Um die Konzepte zu veranschaulichen, wurden in den folgenden Absätzen Tiere als Beispiel verwendet.

2.6.1 URI Design

Eine Ressource benötigt zwei Basis URLs. Die erste repräsentiert die Collection, also die Gruppe, in der sich die Elemente befinden. Hierbei sollten Nomen im Plural als Gruppenname verwendet werden. Außerdem ist es wichtig sich auf eine konkrete Gruppe zu beziehen, (z.B. nicht "animals" sondern "dogs") da sich die Gruppennamen sonst eventuell mit zukünftigen Namen überschneiden. (z.B. eine Gruppe mit dem Namen "cats")

Die zweite identifiziert das anzusprechende Element. Die Identifizierung kann über einen Zahlenwert, ein Datum, einen Namen oder vieles anderes erfolgen. Wichtig ist nur, dass der Identifier eindeutig und verständlich ist. Das könnte beispielsweise wie folgt aussehen.

```
http://api.funnyanimalpics.com/dogs/47
```

```
http://api.funnyanimalpics.com/dogs/68/toys/5
```

Listing 10: Ansteuerung einer Ressource am Server

Hierbei repräsentiert "dogs" die Gruppe und "47" bzw. "68" das gewünschte Element. Innerhalb eines Elements kann weiter auf Unterelemente, wie z.B. "toys", nach demselben Prinzip zugegriffen werden. [8]

2.6.2 Versionierung

Manchmal kann es vorkommen, dass man eine REST-Schnittstelle aktualisieren muss. Doch wie ist das möglich, ohne Applikationen, die noch die alte Schnittstelle unterstützen, zu zerstören? Ein Lösungsansatz wäre, den URI mit einem Versionsverweis zu erweitern. Dadurch kann bei einem Update die Versionsnummer erhöht werden, und alte Programme können die erste API weiterverwenden. Erweitert man nun die vorherige URL um eine Versionsnummer, könnte das beispielsweise wie folgt aussehen. [8]

```
http://api.funnyanimalpics.com/v1/dogs/12345
```

```
http://api.funnyanimalpics.com/dogs/12345?v=1.0
```

Listing 11: Arten der Versionierung

2.6.3 Error-Handling

Selbstverständlich können bei Anfragen an der Schnittstelle Fehlermeldungen oder ein unerwünschtes Verhalten auftreten. Wenn beispielsweise auf einen unerlaubten Bereich zugegriffen wird, sollte eine entsprechende Rückmeldung vordefiniert sein.

Im Fehlerfall sollte ein entsprechender HTTP-Statuscode und eine kurze Problembeschreibung in einfachen Worten für den Benutzer zurückgegeben werden. Zusätzlich kann ein Link zu einer detaillierteren Problembeschreibung eingebunden werden. Oft wird der Statuscode auch in den Response-Body eingebunden, um das Parsen zu vereinfachen. [8]

Bei diesem Beispiel versucht ein User auf eine für Administratoren vorgesehene Ressource zuzugreifen:

```
GET /admins/1234 HTTP/1.1

HTTP/1.1 403 Forbidden

{
  "code": 403,
  "message": "The accessed site can only be viewed with admin privileges."
  "more_info": "http://api.funnyanimalpics.com/errors/12345"
}
```

Listing 12: Fehlerbehandlung

2.6.4 Repräsentationsformate

Um die Schnittstelle so benutzerfreundlich wie möglich zu gestalten, können mehrere Datenformate bereitgestellt werden. Die beiden am häufigsten benutzten Formate sind JSON und XML. Wird bei der Anfrage kein Datentyp angegeben, sollte die Ausgabe in JSON erfolgen. Auch hierbei wird über die URI der Datentyp definiert: [8]

```
http://api.funnyanimalpics.com/v1/dogs/12345.xml

http://api.funnyanimalpics.com/v1/dogs.json

http://api.funnyanimalpics.com/v1/dogs?type=json
```

Listing 13: Auswahl verschiedener Repräsentationsformate

2.7 Implementierungen

Selbstverständlich existieren einige Frameworks, um eine REST-Schnittstelle so einfach wie möglich umzusetzen. Während es unzählige Umsetzungen für die verschiedensten Programmiersprachen gibt, wurden hier zwei wichtige ausgewählt. Zum einen Spring, ein weit verbreitetes Framework für Java und zum anderen Django, ein Webframework für Python. Eine weitere sehr bekannte Umsetzung ist die für Java EE entwickelte Library Jersey. Diese wird hier aber nicht weiter analysiert.

2.7.1 Spring

Spring bietet viele gut dokumentierte Funktionen. Tutorials zu diesen können unter der Domain <http://spring.io/guides> eingesehen werden. Die Funktion, eine REST-Schnittstelle umzusetzen, wird anhand eines Beispiels dargestellt.

Zu Beginn muss selbstverständlich eine Model Klasse erstellt werden. In dem Beispiel wird sie "Greeting" genannt. Diese enthält eine ID, einen Content-String, einen Konstruktor und entsprechende Getter-Methoden. Die Serialisierung wird automatisch von Spring übernommen, wobei dazu die Jackson JSON-Library verwendet wird.

Um Objekte der Klasse via REST zu repräsentieren, muss eine entsprechende Controller-Klasse angelegt werden. Dazu wird eine eigene Klasse erstellt und mit der `@RestController` Annotation versehen. Für jeden zu konfigurierenden URI muss nun eine eigene Methode definiert werden, welche über `@RequestMapping` entsprechend verknüpft wird. Dieser Annotation können nun einige wichtige Parameter übergeben werden. Dazu zählen der Parameter "value", welcher die Ressource angibt und "method", welcher die HTTP-Methode beschreibt.

Auch die Parameter der Methode selbst, welche die per HTTP mitgelieferten Parameter simulieren, müssen mit Annotationen versehen werden. Über die `@RequestParam` Annotation kann der Name des Parameters und ein Default-Wert festgelegt werden.

Die "Application" Klasse startet die eben definierte REST-Schnittstelle. Über die Adresse `127.0.0.1:8080/greeting` kann, nach dem Ausführen der Applikation, nun das Ergebnis eingesehen werden. [14, 35]

Das ganze Tutorial kann auf der Spring Website eingesehen werden. [35]

2.7.2 Django

Django ist das wohl größte Python Webframework. Neben vielen anderen wichtigen Funktionen kann auch eine REST-Schnittstelle erstellt und konfiguriert werden. Dokumentation und Tutorials dafür können auf der Django Website eingesehen werden.

Um eine REST-Schnittstelle mit Django erstellen zu können, müssen die Module "django" und "djangorestframework" über den Paketmanager `pip` installiert werden. Nach dem Aufsetzen eines Projekts und einer untergeordneten Applikation muss die Datenbankstruktur initialisiert und ein Superuser erstellt werden.

Im Gegensatz zu Spring werden die Modelklassen nicht automatisch, sondern über selbst erstellte Klassen serialisiert. Dadurch wird dem User ermöglicht, selbst zu entscheiden, welche Daten wie serialisiert werden.

In eigenen Views, welche zuvor definiert wurden, werden nun die eingehenden Requests beantwortet. Django bietet eine Vielfalt an vordefinierten Views. Dazu gehören Plain Old JSON und XML Ansichten, aber auch eine Art Editoransicht über welche auch Request gesendet werden können.

Anschließend werden den gewünschten URLs die entsprechenden Views zugewiesen. [36]

2.7.3 Lightweight-Umsetzungen

2.7.3.1 *RESTX*

RESTX ist ein schnelles, modulares, funktionsreiches Lightweight-REST-Framework für Java. Über eine einfache Annotation werden Anfragen mit Methoden verknüpft. Das Framework ist noch relativ neu und bietet deshalb leider keine allzu große Community. [37]

2.7.3.2 *Slim Framework*

Slim ist ein PHP Micro Framework und ermöglicht dem Benutzer schnell einfache Anwendungen und APIs zu schreiben. Slim hat es sich zur Aufgabe gesetzt, bei eingehenden HTTP-Requests einen Callback aufzurufen und eine entsprechende Antwort zu senden. [38]

2.7.3.3 *CherryPy*

CherryPy, ein Framework für Python, ermöglicht dem Anwender in nur wenigen Codezeilen eine einfache REST-Schnittstelle zu erstellen. Das Framework hat in den letzten zehn Jahren von seiner Geschwindigkeit und Verlässlichkeit überzeugt und wird heute in vielen Produktionssystemen eingesetzt. [39]

3 JavaScript Object Notation

Die JavaScript Object Notation ist ein Format zum Datenaustausch und ein Teilgebiet von JavaScript. Demnach besitzt JSON keine Features die in JavaScript nicht ohnehin schon vorhanden sind. Wie bereits erwähnt wird JSON oft zum Datenaustausch zwischen mehreren Systemen genutzt und kommt auch oft in REST-Schnittstellen zum Einsatz.

Beim Senden eines JSON Strings gibt es keine Grenzen in Bezug auf die Größe und Kapazität. Jedoch kann es vorkommen, dass ein Server, welcher den JSON Code empfängt, Limits verwendet, um eine zu hohe Rechenlast zu verhindern. [5]

3.1 Syntax

3.1.1 Objekte

Ein Objekt ist eine ungeordnete Liste mit Key/Value Paaren. Objekte werden mit "{" gestartet und enden mit "}". Key und Value werden durch einen Doppelpunkt getrennt. Jedes Paar wird durch ein Komma von einem anderen getrennt. [21]

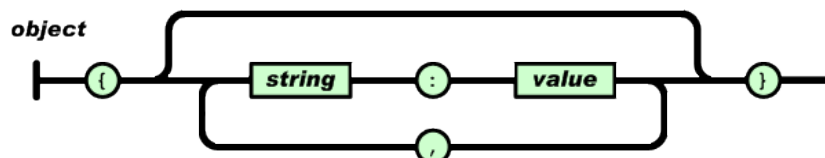


Abbildung 1: Struktur eines JSON Objekts [21]

3.1.2 Arrays

Ein Array ist eine geordnete Liste an Werten. Das können normale Werte, Objekte oder weitere ineinander verschachtelte Arrays sein. Arrays werden mit "[" gestartet und enden mit "]". [21]

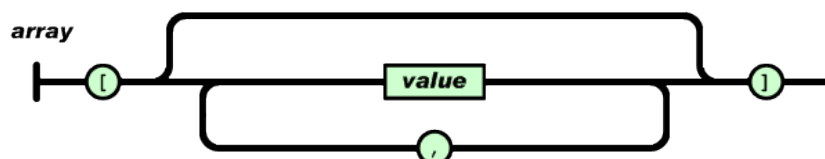


Abbildung 2: Struktur eines JSON Arrays [21]

3.1.3 Datentypen

JSON verfügt über Strings (in doppelten Anführungszeichen), Zahlenwerte, Objekte, Arrays, Wahrheitswerte und Null als Datentypen. [21]

value

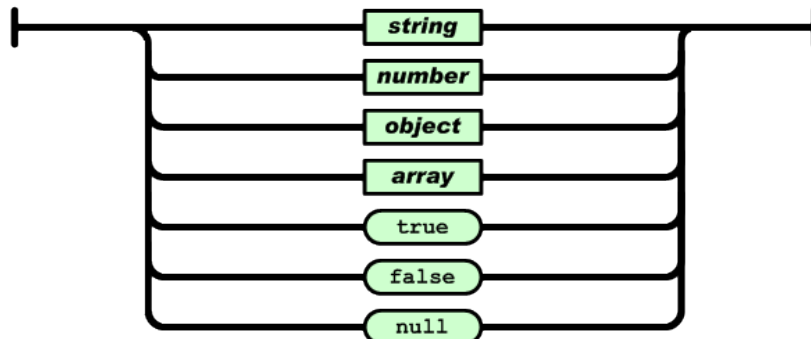


Abbildung 3: Verfügbare Datentypen in JSON [21]

3.1.3.1 Strings

Ein String kann aus keinem oder mehr Unicode Zeichen bestehen. Dieser muss an beiden Enden mit einem doppelten Anführungszeichen versehen sein. Wird das Zeichen "\" verwendet, so wird das darauffolgende Zeichen maskiert. Abbildung 4 enthält eine Liste der Zeichen, die maskiert werden können. [21]

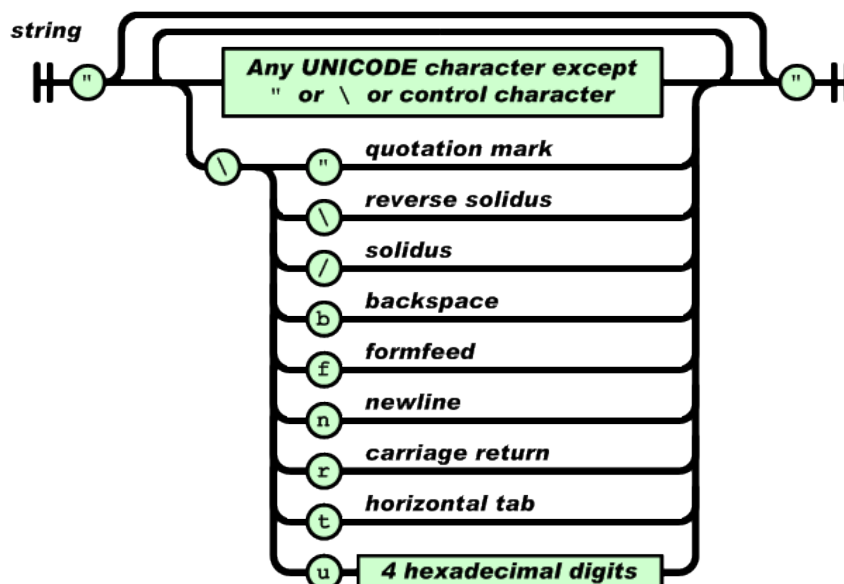


Abbildung 4: Struktur eines Strings in JSON [21]

3.1.3.2 Binärdaten

Da JSON ein textuelles Übertragungsformat ist, können Binärdaten oder BLOBs nicht ohne weiteres übertragen werden. Jedoch ist es mithilfe von Base64 möglich, eben solche Binärdaten in Text umzuwandeln, diesen zu senden und auf der Seite des Empfängers wieder in das Binärformat zu konvertieren. [5]

3.2 Anwendungen von JSON

Damit JSON leicht in den unterschiedlichen Programmiersprachen verwendet werden kann, existieren viele Libraries. Der Hauptfokus liegt hier aber auf Java. Die beiden relevanten Bibliotheken werden anschließend genauer beschrieben.

3.2.1 JSON in Java

3.2.1.1 GSON

Das Ziel von GSON ist es, Java Objekte in die entsprechenden JSON Repräsentation umzuwandeln. Über die Klasse "Gson" können Objekte ganz einfach serialisiert und deserialisiert werden. Die Klasse "Gson" kann ohne Konstruktor Parameter initialisiert werden. Die Klasse besitzt die Methoden "fromJson" und "toJson" mit denen die Konvertierungen durchgeführt werden können. Wenn man nur einfach strukturierte Objekte serialisieren möchte, ist GSON das perfekte Tool. [22]

Dennoch hat die Bibliothek einige negative Aspekte. Aufgrund des Konzepts, existierende Objekte zu konvertieren und umgekehrt, kann über GSON kein unabhängiger JSON String erstellt werden, bei dem die Daten selbst eingetragen werden.

Der viel größere Nachteil ist allerdings, dass bei GSON keine Referenzen serialisiert werden. Das bedeutet, dass Objekte redundant abgespeichert werden und nach einem erneuten laden unabhängig voneinander sind. Die Bibliothek XStream löst dieses Problem z.B. mit Integer Referenzen auf das gewünschte Objekt. XStream ist jedoch eine XML Bibliothek und wird deshalb hier nicht genauer beschrieben. [22]

3.2.1.2 json-simple

Die json-simple Bibliothek löst eines der Probleme von GSON. Json-simple ist darauf ausgelegt JSON Objekte und Arrays zu erstellen ohne sich an eine Java Klasse binden zu müssen. Die Library verfügt über die Klassen "JSONObject" und "JSONArray" über welche Daten über einfache Methoden hinzugefügt werden können. Es ist auch ein Parser vorhanden, welcher JSON aus Dateien ausliest und diese auf Gültigkeit überprüft.

Diese Bibliothek vereinfacht das Erstellen von JSON Strings immens, hat aber sonst eigentlich wenige weitere Funktionen. [23]

3.2.2 JSON in anderen Programmiersprachen

3.2.2.1 JavaScript

JavaScript bietet standardmäßig zwei Methoden um JSON Strings oder JSON Dokumente zu bearbeiten. Einerseits die "JSON.stringify()" Methode, welche existierende JavaScript Objekte in einen JSON String konvertiert und andererseits die "JSON.parse()" Methode welche einen JSON String in ein entsprechendes Objekt umwandelt. [24]

3.2.2.2 Python

Python enthält standardmäßig ein JSON Modul. Dieses enthält die Methoden "json.dumps()" und "json.loads()". Die Python Syntax von Listen und Verzeichnisse ähnelt JSON sehr. Daher repräsentieren Verzeichnisse die JSON Objekte und Listen die JSON Arrays. Null Werte in JSON werden von None in Python repräsentiert. Mithilfe der "dumps" Methode werden Objekte zu JSON Strings konvertiert. Mit der "loads" Methode werden JSON Strings zu Objekten umgewandelt. [25]

4 HTTP 2.0

4.1 Einleitung

"HTTP/2 enables a more efficient use of network resources and a reduced perception of latency by introducing header field compression and allowing multiple concurrent exchanges on the same connection. It also introduces unsolicited push of epresentations from servers to clients." ([13], S. 1)

4.1.1 Probleme von HTTP 1.1

Der derzeit weit verbreitete Standard HTTP 1.1 (Hypertext Transfer Protocol) stammt aus dem Jahr 1999. Somit erschien dieser nur 3 Jahre nach HTTP 1.0 (1996). Durch HTTP 1.1 wurden einige Performance-Optimierungen durchgeführt, dazu gehört z.B. HTTP-Pipelining. [13]

HTTP-Pipelining ermöglicht die parallele Abarbeitung von HTTP-Anfragen. Das Prinzip hinter HTTP-Pipelining kann man mit dem Anstellen bei einer Kassa im Supermarkt vergleichen: Es gibt mehrere Warteschlangen, deren Kunden nacheinander abgearbeitet werden. Ein Nachteil dieser Warteschlangen ist, dass man nicht genau weiß, wie lange der Kunde bzw. die Kunden vor einem noch benötigen. Eventuell benötigen diese sehr viel Zeit und blockieren somit alle anderen Kunden. Genau von dieser Problematik ist auch HTTP-Pipelining betroffen. [10, 7]

Eine weitere Schwachstelle sind HTTP-Header. Diese sind oftmals überladen, d.h. in den meisten Fällen werden die Angaben nicht benötigt oder sind redundant. Dies führt zu steigender Netzwerkauslastung und erhöht die Latenzzeiten unnötigerweise. [13]

Letztendlich ist das Hauptproblem von HTTP 1.1, dass dieser Standard den heutigen Anforderungen von immer mehr Übertragungsvolumen nicht mehr gerecht werden kann und die Verfügbarkeit von höheren Bandbreiten auch nicht optimal ausnutzt. [13, 10]

4.1.2 Ziele

Der Schwerpunkt dieser Version von HTTP liegt auf der Verbesserung der Übertragung zwischen Client und Webserver. Die verfügbare Bandbreite soll optimal ausgenutzt werden, um Ladezeiten erheblich zu verringern. [7, 13]

Allerdings sind diesen Intentionen auch Grenzen bzw. Einschränkungen entgegengesetzt:

- Beibehaltung des Grundprinzips: Der Client sendet via TCP Anfragen an den Server
- Gleichbleibende URLs
- Mapping der HTTP 2 Features auf ältere HTTP 1.1 Clients
- Keine weiteren Minor-Releases (entweder HTTP 2 wird unterstützt oder nicht)

[7, 10]

4.1.3 SPDY

SPDY (gesprochen "speedy") ist ein experimentales Protokoll, welches von Google entwickelt und im Jahr 2009 angekündigt wurde. Hauptziel war, die Performance-Probleme von HTTP 1.1 zu beheben, dazu gehörte unter anderem Folgendes:

- Reduktion der Ladezeiten um 50%
- Änderungen an vorhandenen Webseiten nicht erforderlich
- Minimaler Aufwand, SPDY zu implementieren
- Entwicklung erfolgt zusammen mit der Open Source Community

Wie sich im Jahr 2012 herausstellte, konnten SPDY die Versprechen halten. Somit begann die Entwicklung von HTTP 2. SPDY hatte starken Einfluss auf den neuen Standard und die neuen Erfahrungen durch SPDY wurden berücksichtigt. [7, 10]

4.2 Aufbau & Verbindung

HTTP 2 führt eine neue Schicht auf der Application-Ebene, den Binary Framing Layer, ein (siehe Abbildung 6). Die Kommunikation zwischen Server und Client erfolgt daher binär und ist unterteilt in Streams, Messages und Frames. Diese können parallel innerhalb einer aktiven TCP-Verbindung abgearbeitet werden. Die Unterteilung der Nachrichten in Frames bzw. das Framing übernehmen der Client bzw. der Server. Somit sind beispielsweise die HTTP-Methoden (GET, POST etc.) davon nicht betroffen. Bestehende Software und Web-Anwendungen können somit wie bisher weiterverwendet werden. [7, 10]

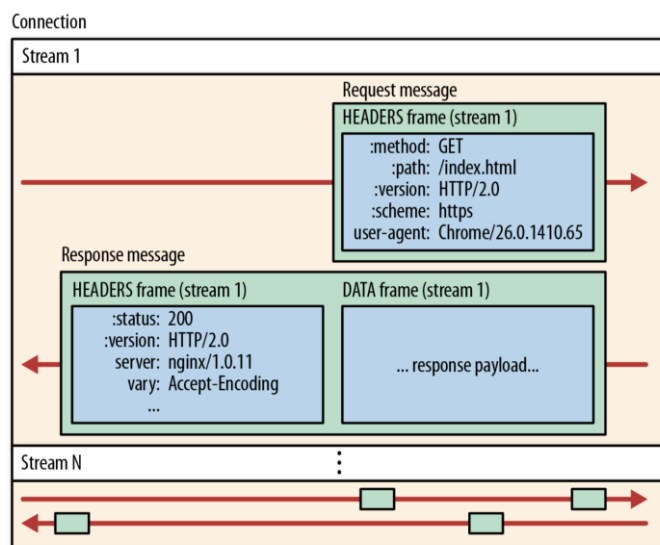


Abbildung 5: HTTP 2 Verbindung

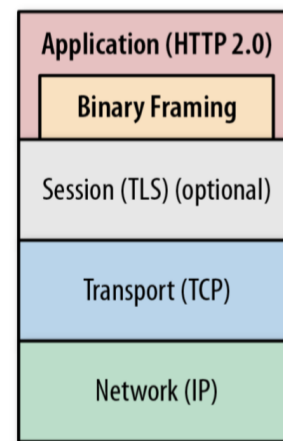


Abbildung 6: HTTP 2 Binary Framing

4.2.1 Streams

Innerhalb einer Verbindung können 1 bis n Streams vorhanden sein, welche Messages bidirektional zwischen Client und Server übertragen. Jeder Stream ist eindeutig identifizierbar. Ein Stream kann sowohl vom Server als auch vom Client aufgebaut bzw. beendet werden. [7, 10, 13]

4.2.2 Messages

Innerhalb eines Streams können 1 bis n Messages vorhanden sein, welche einen oder mehrere Frames beinhalten. Eine Message ist z.B. "Request" oder "Response". [7]

4.2.3 Frames

Ein Frame ist die kleinste Einheit der Kommunikation. Innerhalb einer Message können 1 bis n Frames vorhanden sein, wie z.B. ein HEADERS und ein DATA Frame. Jedes Frame benötigt einen Frame Header, welcher zumindest den zugeordneten Stream angeben muss. Die Reihenfolge der Frames innerhalb einer Message ist wichtig, da diese in genau dieser Reihenfolge abgearbeitet werden. [7, 13]

4.3 Neue Funktionalität

4.3.1 Multiplexing

Durch Multiplexing können Client und Server mit Hilfe mehrerer paralleler Streams über eine einzige TCP-Verbindung miteinander kommunizieren. [13]

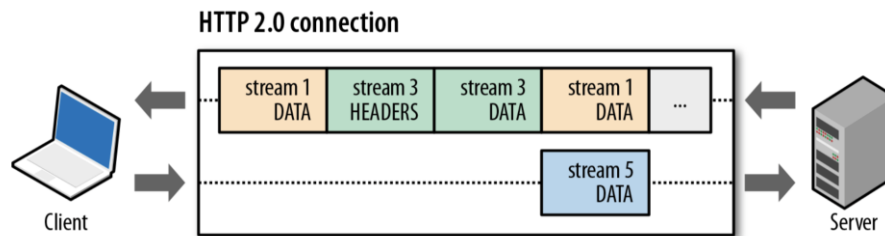


Abbildung 7: HTTP 2 Multiplexing [7]

In diesem Beispiel sind 3 Streams parallel aktiv. Durch Multiplexing können mehrere Frames aus verschiedenen Streams abwechselnd gesendet werden. Lediglich die Reihenfolge der Frames eines Streams ist von Bedeutung, da die Frames nach dem Empfangen genau in dieser Reihenfolge wieder zusammengesetzt werden. [7, 10]

4.3.2 Header Compression

HTTP ist ein zustandsloses Protokoll: Bei jeder Anfrage müssen alle Infos, die der Server zum Bearbeiten der Anfrage benötigt, vom Client im Header mitgeschickt werden. Durch viele Client-Anfragen ist der Inhalt des Headers oft redundant. Diese Informationen lassen sich daher auch sinnvoll komprimieren. [7, 10]

HTTP 2 verwendet sogenannte "Header Tables", die sowohl am Client als auch am Server verwendet werden. Darin werden gesendete Informationen als Key-Value-Pairs für die gesamte Verbindungsdauer gespeichert. Werden nun Header-Metadaten gesendet, werden diese entweder neu in die Tabelle eingefügt oder aktualisiert. In einem nachfolgenden Request müssen somit nur die neuen Metadaten übertragen werden, da die bestehenden Daten aus der Tabelle ausgelesen werden. [7, 13]

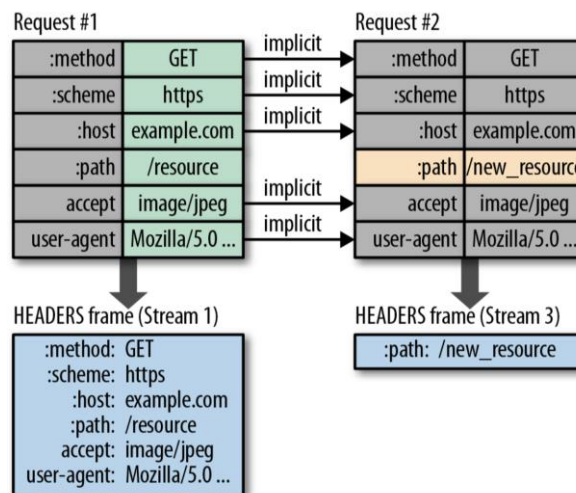


Abbildung 8: Header Compression [7]

4.3.3 Server Push

Mit Hilfe von Server Push kann der Server auf eine Anfrage des Clients mit mehr Ressourcen antworten, als ursprünglich vom Client angefordert wurden. Zum Beispiel könnte der Server nach Anfrage einer .html-Datei zusätzlich auch CSS und Javascript mitliefern. Der Client kann kontrollieren, ob er dies zulässt bzw. inwiefern er dies zulässt, da die Zusatzressourcen vom Client zwischengespeichert werden. Der Server kann die Ressourcen priorisieren, um bei einer Einschränkung durch den Client wichtige Ressourcen zuerst zu übertragen. [7, 10, 13]

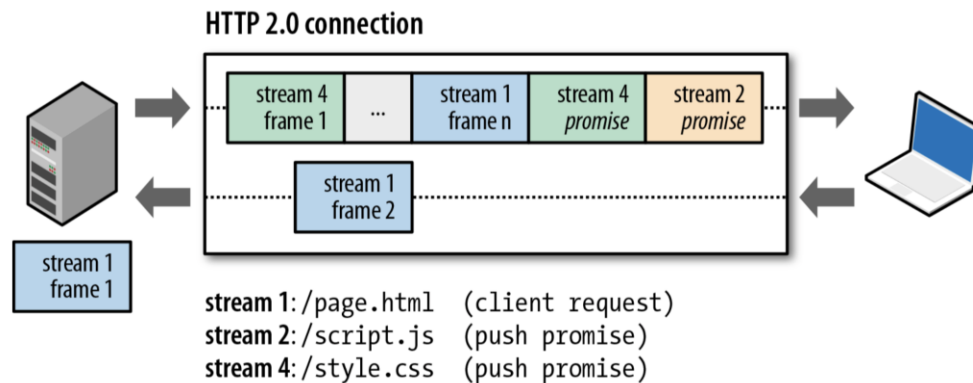


Abbildung 9: Server Push [7]

4.3.4 Stream Priority

Um die Performance weiter zu steigern und wichtige Infos zuerst zu übertragen, können Streams priorisiert werden. Die Priorität wird als 31-bit Integer ausgedrückt, wobei ein höherer Wert eine niedrigere Priorität bedeutet: 0 bedeutet höchste Priorität, 2^{31} die niedrigste. Am wichtigsten ist Priorisierung, wenn die Bandbreite zum Senden nur gering ist. Zum Beispiel sollte bei einer Nachrichtenseite der Text vor den Bildern laden. Außerdem kann der Client dem Server somit mitteilen, welche Abhängigkeiten zwischen den einzelnen Streams vorhanden sind. [7, 10]

Die Priorität wird beim Aufbau eines neuen Streams im HEADER Frame festgelegt und kann per PRIORITY Frame jederzeit geändert werden. Zum Beispiel können beim Hinunterscrollen einer Webseite mit vielen Bildern die Bilder, die näher in den Fokus rücken, einer immer höheren Priorität zugewiesen werden. [10, 13]

4.3.5 Flow Control

Durch Flow Control wird gesteuert, ob bzw. inwiefern der Empfänger eines Streams in der Lage ist, weitere DATA Frames zu empfangen. Sowohl Server als auch Client teilen sich per WINDOW_UPDATE Frame für jeden Stream oder für die gesamte Verbindung mit, wie viel Information diese jeweils noch empfangen können. Ist das Kontingent aufgebraucht, muss der Server/Client per WINDOW_UPDATE Frame dieses wieder anheben. [7, 13]

4.4 Methoden

Um auf eine Ressource per HTTP zuzugreifen, stehen verschiedene Methoden zur Verfügung. In diesem Kapitel werden Methoden beschrieben, die äquivalent zum CRUD-Prinzip (Create, Read, Update, Delete) sind, und daher oftmals benötigt werden. Zusätzlich wird auf die Methode OPTIONS näher eingegangen. Weitere Methoden findet man im RFC 2616.

4.4.1 POST

POST ist das Äquivalent zu CREATE im CRUD Prinzip. Mit POST sollen somit neue Ressourcen angelegt werden. Als passenden Response-Code sollte man "201 Created" zurückgeben, falls die Aktion erfolgreich war. Die POST-Methode wird allerdings auch oft verwendet, um Aktionen durchzuführen, für die es keine eigenen Methoden gibt. [9, 15]

4.4.2 GET

GET ist das Äquivalent zu READ im CRUD Prinzip. Mit GET sollen bestehende Informationen ausgelesen werden und es soll keine Änderung der Informationen erfolgen. Mittels GET-Parameter kann die Suche nach angegebenen Kriterien eingeschränkt werden. GET liefert bei mehrmaligen Aufrufen immer dasselbe Ergebnis (vorausgesetzt, die Daten haben sich nicht geändert). [9, 15]

Mittels Conditional GET kann zusätzlich im Header angegeben werden, ob überhaupt ein Auslesen der Resource erfolgen soll. Beispielsweise kann zuerst überprüft werden, ob sich Daten überhaupt geändert haben. [9, 15]

4.4.3 PUT

PUT ist das Äquivalent zu UPDATE im CRUD Prinzip. Mit PUT sollen bestehende Ressourcen aktualisiert werden, falls diese existieren. Falls diese nicht existieren, wird in manchen Fällen auch eine neue Ressource angelegt. Meist wird allerdings ein entsprechender Fehlercode zurückgegeben. [9, 15]

4.4.4 DELETE

DELETE ist das Äquivalent zu DELETE im CRUD Prinzip. Mit DELETE sollen bestehende Ressourcen gelöscht werden. Falls die Ressource nicht existiert, wird meist ein entsprechender Fehlercode zurückgegeben. [9, 15]

4.4.5 OPTIONS

OPTIONS liefert Metadaten zurück, welche sich auf die für diese Ressource erlaubten Methoden beziehen. Ein OPTIONS Request könnte demnach zurückliefern, dass für eine spezielle Ressource nur POST und GET Requests möglich sind. [9, 15]

4.5 Statuscodes

Die folgenden Tabellen bieten einen Überblick über häufig eingesetzte HTTP-Statuscodes. [9]
Weitere Statuscodes sind im RFC 2616 definiert.

4.5.1 Informationen

Code	Bedeutung	Erklärung
100	Continue	Teil des Requests übermittelt, Client soll mit weiteren Teilen fortfahren
101	Switching Protocols	Der Server wechselt wie vom Client im Upgrade-Header angegeben das Protokoll (z.B. HTTP 1.1 → HTTP 2.0)

4.5.2 Erfolgreiche Operation

Code	Bedeutung	Erklärung
200	OK	Die Anfrage ist OK.
201	Created	Die vom Client angefragte Ressource wurde angelegt (POST-Request).
202	Accepted	Die Anfrage wurde vom Server angenommen, allerdings noch nicht weiterverarbeitet.
204	No Content	Die Antwort des Servers enthält keinen Response-Body.
205	Reset Content	Browser soll HTML-Form-Elemente zurücksetzen.

4.5.3 Umleitung

Code	Bedeutung	Erklärung
300	Multiple Choices	Der Client hat eine URL angefragt, die auf mehrere Ressourcen verweist. Er erhält eine Liste an Optionen.
301	Moved Permanently	Die Ressource ist unter einer neuen URL aufrufbar.
302	Found	Wie 301, nur zeitlich begrenzt.
304	Not Modified	Ressource ist weiterhin normal unter vorhandener URL aufrufbar.
307	Temporary Redirect	Wie 302. Zusätzlich soll der Client die Methode nicht ändern.

4.5.4 Client-Fehler

Code	Bedeutung	Erklärung
400	Bad Request	Client sendete eine ungültige (falsche Syntax) Anfrage.
401	Unauthorized	Autorisierung des Clients erforderlich.
403	Forbidden	Client darf nicht auf Ressource zugreifen.
404	Not Found	Server kann die angefragte Ressource nicht finden.
410	Gone	Wie 404, nur dass der Server die Ressource vorher angeboten hat.

4.5.5 Server-Fehler

Code	Bedeutung	Erklärung
500	Internal Server Error	Während der Bearbeitung der Anfrage ist ein Fehler aufgetreten.
502	Bad Gateway	Proxy/Gateway erhält keine Antwort von Backend-Servern.
501	Not Implemented	Der Server hat die geforderte Funktionalität nicht implementiert.
503	Service Unavailable	Der Server kann derzeit diese Anfrage nicht beantworten, später aber schon

4.6 Implementierungen

Damit HTTP 2 verwendet werden kann, müssen sowohl der Server als auch der Client kompatibel sein. HTTP 2 ist allerdings abwärtskompatibel, wie bereits im Kapitel 4.1.2 erwähnt. Beispielweise kann ein HTTP2-Server auch per HTTP 1.1 mit einem HTTP 1.1 Client kommunizieren.

4.6.1 Server

Es wurden die Kompatibilität mit HTTP 2 von den Webservern Apache, NGINX und IIS überprüft, da diese am weitesten verbreitet sind.

4.6.1.1 Apache

Ab Version 2.4.17 des Apache Webserver wird durch das Modul mod_h2 HTTP 2 unterstützt. [16]

4.6.1.2 NGINX

Ab Version 1.9.5 von NGINX wird HTTP 2 unterstützt. Dabei wurde allerdings nur die verschlüsselte Variante, HTTP 2 über TLS, implementiert. [17]

4.6.1.3 IIS

Ab Version 10 unterstützt Microsoft IIS HTTP 2. [40]

4.6.2 Browser

Folgende Graphik zeigt die Kompatibilität der gängigsten Webbrowser mit HTTP 2:

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			31					4.1	
8		² 38	² 43					4.3	
9		² 39	² 44					4.4	
10		² 40	² 45	8		8.4		4.4.4	
¹ 11	12	² 41	² 46	9	32	9	8	44	² 46
	13	² 42	² 47		33				
		² 43	² 48		34				
		² 44	² 49						

Abbildung 10: Browser-Kompatibilität mit HTTP 2 [18]

¹ Nur IE 11 unter Windows 10 unterstützt HTTP 2

² Nur HTTP 2 über TLS wird unterstützt (HTTPS)

5 OAuth 2.0

5.1 Einleitung

5.1.1 Definition & Zweck

OAuth ist ein Standard, welcher Webservices eingeschränkten Zugriff auf Daten bzw. die API anderer Webservices (z.B. Google, Facebook etc.) ermöglicht. Die User-Authentifizierung wird dabei auf dem Service, der die Daten beinhaltet, durchgeführt. Dieser ermöglicht anschließend anderen Web-Services eingeschränkten Zugriff auf die Daten. [4, 12]

Die Benutzerdaten, die der User zur Autorisierung eingeben muss, gelangen somit nicht zu den Services Dritter. Beispielsweise benötigt ein Webservice Zugriff auf die Daten der Dropbox eines Users. Dieser möchte bei dem Webservice allerdings nicht seine Dropbox-Daten eingeben. Einerseits kann der User sich nicht sicher sein, wie dieser Webservice mit den Daten umgehen würde, andererseits würde der Webservice Vollzugriff auf das Dropbox-Konto erhalten. Stattdessen erhält der Webservice eine eingeschränkte Zugriffsberechtigung auf das Dropbox-Konto durch OAuth. [6, 12]

5.1.2 Unterschiede zu OAuth 1

OAuth 2 unterscheidet sich von OAuth 1 unter anderem in folgenden Punkten:

- Neue Workflows, um zwischen verschiedenen Anwendungsfällen von OAuth zu unterscheiden
- Tokens müssen nicht signiert werden, stattdessen kann man sich auch nur auf eine verschlüsselte Verbindung per SSL verlassen.
- Statt einem Token mit langer Gültigkeit können Tokens mit kürzerer Gültigkeit und ein Refresh Token mit langer Gültigkeit verwendet werden.
- Klare Trennung der Rollen: Trennung von Authorization Server und Resource Server (siehe "Rollen" unten)

[6, 20]

5.1.3 Client-Profile

In OAuth 2 bedeutet die Bezeichnung "Client" nicht, dass immer eine Desktop-Anwendung beim End-Anwender gemeint ist. Daher wurden folgende Client-Profile definiert:

- Serverseitige Webanwendung: Ein OAuth Client in Form einer serverseitigen Webanwendung. Diese hat Zugriff auf eine externe Provider-API, nicht aber z.B. ein Script im Browser des Clients.
- Clientseitige Anwendung in einem Webbrowser: Ein OAuth Client, welcher im Webbrowser des User ausgeführt wird (z.B. als JavaScript-Anwendung). Diese hat direkten Zugriff auf eine externe Provider-API.
- Native Anwendung: Ähnlich zur clientseitigen Anwendung im Webbrowser, allerdings handelt es sich um eine nativ beim Client installierte Applikation.

[12, 19]

5.2 Rollen

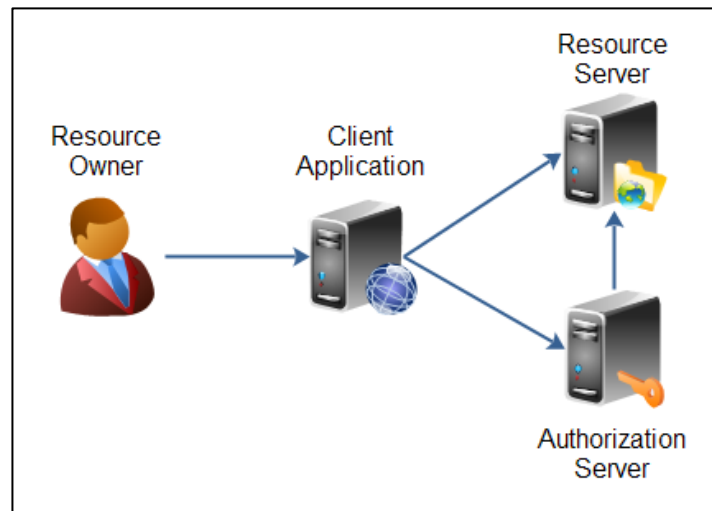


Abbildung 11: Rollen in OAuth 2 [19]

5.2.1 Resource Owner

Der Resource Owner ist meist der Benutzer einer Applikation, welche Daten enthält, auf die die Client-Applikation Zugriff erhalten möchte. Die Daten dieser Applikation sind auf dem Resource Server gespeichert. Bei dem Resource Owner kann es sich auch um eine weitere Applikation handeln. [12, 19]

Beispiel: Benutzer von Facebook, Google etc.

5.2.2 Client Applikation

Eine Applikation, die unter der Identität des Resource Owners auf dessen Daten, welche sich am Resource Server befinden, zugreifen möchte. Der Begriff "Client" gibt nicht Aufschluss darüber, ob die Anwendung auf einem Server, einem Desktop-Computer oder auf anderen Geräten ausgeführt wird. [12, 19]

Beispiel: Ein Spiel, welches Zugriff auf den Facebook-Account des Benutzers benötigt

5.2.3 Resource Server

Der Resource Server beinhaltet die Daten, auf die der Benutzer der Client-Applikation Zugriff gewährleisten kann. Dazu erhält dieser Zugriffs-Anfragen auf die Daten des Benutzers und entscheidet, ob Zugriff gewährleistet wird oder nicht. [12, 19]

Beispiel: Server von Facebook, Google etc. mit Daten der Benutzer

5.2.4 Authorization Server

Der Authorization Server autorisiert die Client-App, auf die Ressourcen des Resource Owners zuzugreifen. Der Authorization und der Resource Server können zusammengefasst werden. Der OAuth-Standard legt nicht fest, wie die beiden Server kommunizieren sollten, wenn sie getrennt betrieben werden. [12, 19]

5.3 Authorization Flows

Damit man OAuth 2 in einer Webanwendung verwenden kann, diese also somit zum OAuth Client wird, muss zuerst beim jeweiligen API-Provider (Google, Facebook etc. oder ein selbst implementierter API-Provider) eine Client ID sowie ein Client Secret angefordert werden. Danach wählt man je nach Anwendungsfall einen der beschriebenen Authorization Flows. Diese beschreiben, wie der OAuth Client die Zugriffsberechtigung auf eine gewünschte Ressource erhält.

5.3.1 Authorization Code Flow

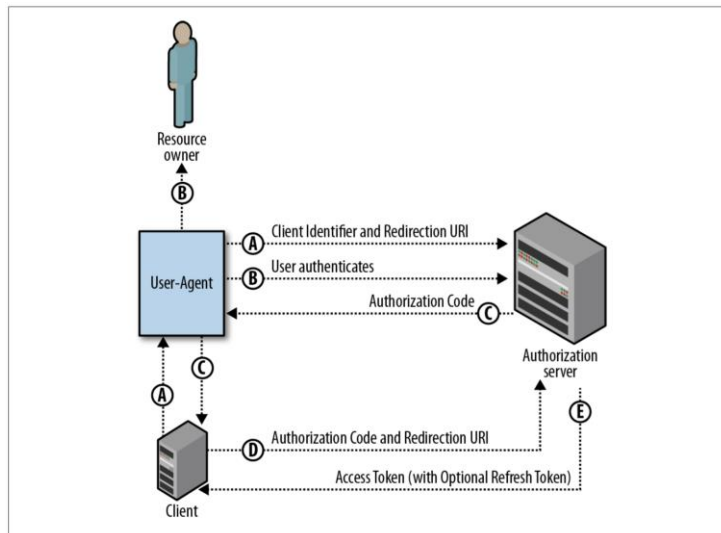


Abbildung 12: Authorization Code Flow [6]

Beim Authorization Code Flow erhält der Client nach Authentifizierung des Users einen Authorization Code vom Authorization Server. Diesen tauscht der Client beim Authorization Server gegen einen Access Token. Mit Hilfe des Access Tokens kann auf die Provider-API zugegriffen werden.

Der Authorization Code Flow soll angewandt werden, wenn ...

- ... der Zugriff über längere Zeit benötigt wird
- ... der OAuth Client ein Web-Applikations-Server ist.
- ... der OAuth Token aus Sicherheitsgründen nicht zum Browser des Benutzers gelangen soll

5.3.1.1 Schritt 1: Autorisierung des Clients & Erhalten des Authorization Codes

Im ersten Schritt wird der User von der Webanwendung an den Authorization Server weitergeleitet, um sich anzumelden und Zugriff zu erfragen. Bei dieser Weiterleitung wird unter anderem als Parameter der Scope, also die Dateien, die die Webanwendung erhalten möchte, angegeben. Falls der User Zugriff gestattet, antwortet der Authorization Server mit einem Authorization Code. Falls nicht, antwortet dieser mit einem Error, welchen die Webanwendung korrekt verarbeiten sollte. [4, 6, 12]

5.3.1.2 Schritt 2: Tausch Authorization Code gegen Access Token

Die Webanwendung hat einen Authorization Code erhalten, der nun gegen einen Access Token getauscht werden muss, um Zugriff auf die Provider-API zu erhalten. Dazu sendet die Webanwendung ein POST-Request mit dem Authorization Code an den Authorization Server, welcher bei korrektem Code mit einem Access Token antwortet. Der Access Token kann mit einer

zeitlichen Beschränkung versehen werden. In diesem Fall wird zum Erneuern des Tokens zusätzlich ein Refresh Token übertragen. [4, 6, 12]

5.3.1.3 Schritt 3: Aufruf der Provider-API

Als letzter Schritt kann nun mittels Access Tokens die Provider-API aufrufen. Dabei wird dieser meist im HTTP Authorization Header angegeben. Falls der Token nicht mehr gültig ist, erhält man einen HTTP 4xx Error, den die Webanwendung korrekt verarbeiten sollte. Falls man einen Refresh Token erhalten hat, kann die Webanwendung mit diesem einen neuen Token vom Authorization Server anfordern. [4, 6, 12]

5.3.2 Implicit Grant Flow

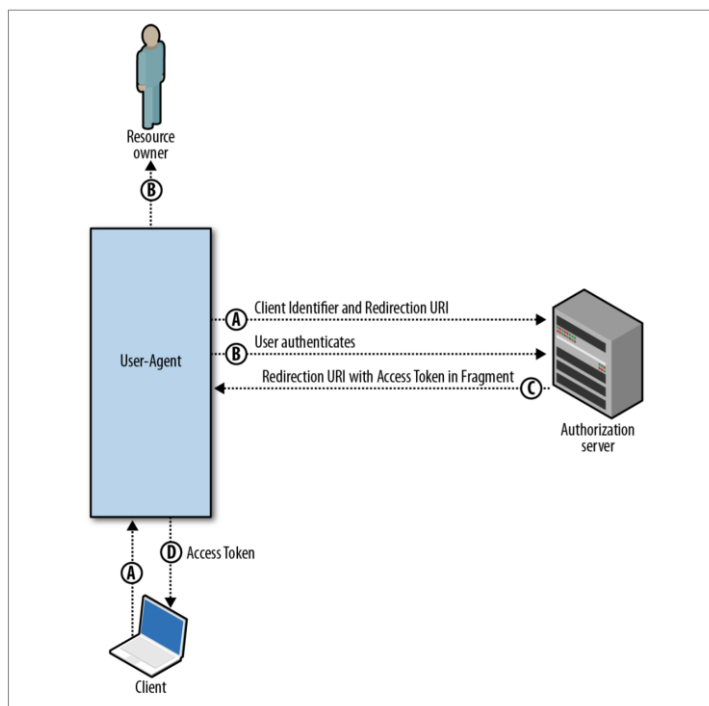


Abbildung 13: Implicit Grant Flow [6]

Beim Implicit Grant Flow erhält der Client nach Authentifizierung des Users einen Access Token vom Authorization Server. Mit Hilfe des Access Tokens kann auf die Provider-API zugegriffen werden.

Der Implicit Grant Flow soll angewandt werden, wenn ...

- ... der Zugriff nur temporär benötigt wird.
- ... der OAuth Client im Webbrowser ausgeführt wird (JavaScript, Flash etc.).
- ... dem Browser insofern vertraut wird, dass der Access Token nicht nach außen dringt.

5.3.2.1 Schritt 1: Autorisierung des Clients & Erhalten des Access Tokens

Der erste Schritt funktioniert ähnlich wie beim Authorization Code Flow: Der User wird von der Webanwendung zum Authorization Server weitergeleitet, um sich dort zu authentifizieren. Nachdem dieser Zugriff gestattet hat, erhält die Webanwendung im Unterschied zum Authorization Code Flow direkt den Access Token. [4, 6, 12]

5.3.2.2 Schritt 2: Aufruf der Provider-API

Ähnlich wie beim Authorization Code Flow kann nun die Provider-API aufgerufen werden. Falls der Token abgelaufen ist, gibt es im Unterschied zum Authorization Code Flow keinen Refresh Token zum Erneuern des Tokens. Stattdessen müssen wie beim initialen Anfordern des Tokens Schritt 1 wieder durchgeführt werden. [4, 6, 12]

5.3.3 Ressource Owner Password Flow

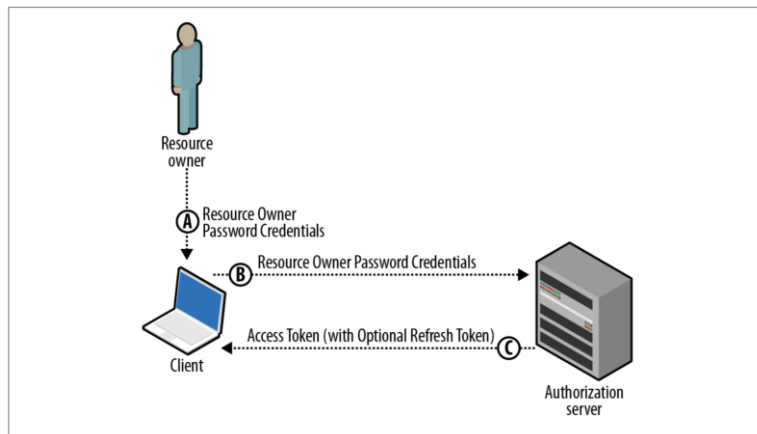


Abbildung 14: Ressource Owner Password Flow [6]

Beim Ressource Owner Password Flow gibt der Benutzer die Zugangsdaten direkt am OAuth Client ein, welcher diese an den Authorization Server überträgt und somit einen Access Token erhält. Mit Hilfe des Access Tokens kann auf die Provider-API zugegriffen werden.

Der Ressource Owner Password Flow sollte möglichst selten angewandt werden, da die Anwendung die Zugangsdaten des Users erhält. Deshalb sollte dieser nur verwendet werden, wenn es sich bei der Anwendung um eine "offizielle" Anwendung vom API-Provider handelt.

5.3.3.1 Schritt 1: Anfordern der Zugangsdaten vom User

Im ersten Schritt wird der User aufgefordert, die Zugangsdaten bei der Anwendung direkt einzugeben. Daher erfolgt im Unterschied zu den bisher beschriebenen Workflows keine Weiterleitung zu einem Authorization Server. [4, 6, 12]

5.3.3.2 Schritt 2: Tauschen der Zugangsdaten gegen einen Access Token

Ähnlich wie beim Tausch des Authorization Codes gegen den Access Token im Authorization Code Flow werden dieses Mal direkt die Zugangsdaten des Users gegen einen Access Token getauscht. Dabei wird ebenfalls als Parameter angegeben, auf welche Daten der Zugriff benötigt wird. Der Authorization Server antwortet bei korrekten Zugangsdaten und erlaubtem Zugriff mit dem Access Token und optional auch mit einem Refresh Token. [4, 6, 12]

5.3.3.3 Schritt 3: Aufruf der Provider-API

Als letzter Schritt kann nun mittels Access Tokens die Provider-API aufrufen. Ähnlich wie beim Authorization Code Flow kann mittels Refresh Token ein neuer Access Token angefordert werden. [4, 6, 12]

5.3.4 Client Credentials Flow

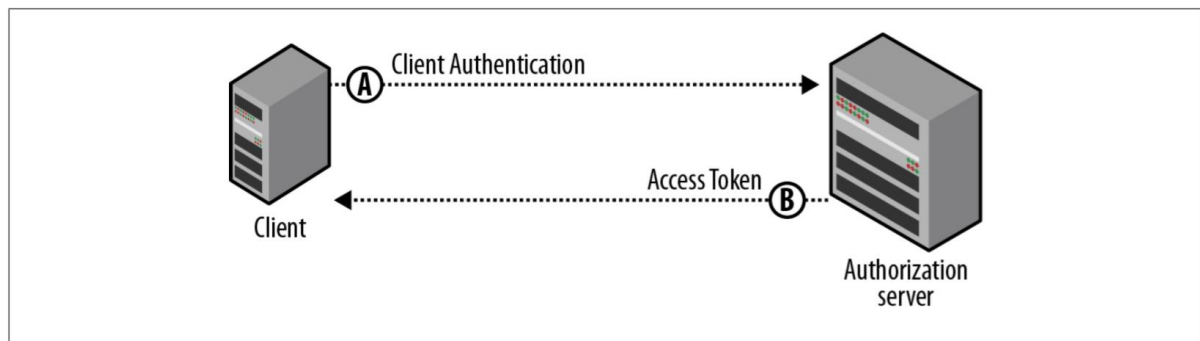


Abbildung 15: Client Credentials Flow [6]

Beim Client Credentials Flow werden die Client-Zugangsdaten gegen den Access Token getauscht, daher wird dabei der User nicht miteinbezogen. Mit Hilfe des Access Tokens kann auf die Provider-API zugegriffen werden.

Der Resource Owner Password Flow sollte dann verwendet werden, wenn beispielsweise eine Webanwendung auf externe Ressourcen zugreift, die nicht für den individuellen Benutzer gedacht sind.

5.3.4.1 Schritt 1: Tausch der Client-Zugangsdaten gegen einen Access Token

Der Client tauscht seine Client-Zugangsdaten (Client ID & Secret) beim Authorization Server gegen einen Access Token. [4, 6, 12]

5.3.4.2 Schritt 2: Aufruf der Provider-API

Mittels des erhaltenen Access Tokens kann nun die API des Providers aufgerufen werden. Eine Erneuerung des Access Tokens, sollte dieser nicht mehr gültig sein, erfolgt wie in Schritt 1 angegeben. [4, 6, 12]

5.4 Anwendungen bzw. Implementierungen

5.4.1 Apache Oltu

Apache Oltu ist eine Library für Java-Anwendungen. Mit dieser lassen sich folgende Rollen aus OAuth 2 implementieren:

- OAuth Authorization Server
 - Authorization Code Endpoint
 - Token Endpoint
- OAuth Resource Server
- OAuth Client

Die Implementierung kann beispielsweise in einem Servlet erfolgen. Auf der Website von Apache Oltu ist eine gute Dokumentation inklusive Beispielen vorhanden. [26]

5.4.2 Spring Security OAuth

Spring ist ein Web-Framework für Java-Anwendungen und bietet ebenfalls die Möglichkeit, folgende Rollen aus OAuth 2 zu implementieren:

- OAuth Authorization Server
 - Authorization Code Endpoint
 - Token Endpoint
- OAuth Resource Server
- OAuth Client

Die Konfiguration des Authorization Servers kann entweder in einer Klasse mit der Annotation `@EnableAuthorizationServer` oder in einer XML-Datei mit dem Element `<authorization-server/>` erfolgen. Token des Authorization Servers können entweder im Speicher oder in einer Datenbank per JDBC gespeichert werden. [27]

Die Konfiguration des Resource Servers und des Clients erfolgt ähnlich wie die des Authorization Servers entweder per Annotation oder in einer XML-Datei. [27]

5.4.3 Client-Libraries für Google

Google bietet zahlreiche Client-Libraries zum Zugriff auf die Google APIs wie z.B. Google Maps oder YouTube. Dabei sind Implementierungen für zahlreiche Programmiersprachen, wie z.B. Java, Python, JavaScript oder C++ vorhanden. Für jede Programmiersprache ist ein "Getting Started"-Tutorial und Beispielcode vorhanden. [28]

5.4.4 Client-Libraries für Facebook

Facebook bietet zwar nicht direkt Client-Libraries an, es sind allerdings zahlreiche Libraries vorhanden. Dazu zählt beispielsweise "SocialAuth" für Python oder "Scribe" für Java, welche auch andere APIs (Twitter, Yahoo, Vimeo usw.) unterstützt. [29, 30]

5.4.5 Beispiel einer Implementierung des Authorization Code Flows

Beispielsweise soll ein OAuth-Login mittels Google-Account realisiert werden. Laut Authorization Code Flow muss man das Folgende implementieren:

- Weiterleitung des Users zur Authentifizierung beim API-Provider (Google). Dabei darf man nicht auf den Scope, also eine Definition der Daten, auf die man Zugriff erhalten möchte, vergessen.
- Callback, welches nach der User-Authentifizierung (bei Google) aufgerufen wird. Dabei bekommt man den Authorization Code und muss diesen gegen einen Access Token (bei Google) tauschen, um letztendlich Zugriff auf die Provider-API (Google) zu erhalten.

Oftmals treten Probleme auf, weil sich die URL der API geändert hat oder man den Scope nicht richtig definiert. Außerdem muss man wissen, dass man beim Authorization Code Flow nicht direkt mit dem Authorization Code auf die API zugreifen kann, sondern diesen vorher gegen einen Token tauschen muss.

6 Zusammenfassung & Schlussfolgerung

REST-Schnittstellen bieten die Möglichkeit einer einfachen Kommunikation per HTTP. Eine Anwendung kann durch unterschiedliche Requests (GET, POST etc.) an einen bestimmten URI auf Ressourcen einer anderen Anwendung zugreifen. Eine Response enthält einen HTTP-Statuscode und im Body die angeforderte Ressource im XML- oder JSON-Format. URIs sollten sinnvoll festgelegt werden: Grundsätzlich verwendet man als Ausgangspunkt eine Collection von Elementen. Im Maturity-Model ist definiert, wie umfangreich eine API die REST-Konzepte umsetzt. Bei Level 0, nutzt eine Anwendung immer nur eine URI und POST-Requests. Bei Level 3 nutzt sie hingegen mehrere URIs, unterschiedliche HTTP-Methoden und Hypermedia. Hypermedia ermöglicht es, Verlinkungen in der Response auf andere Ressourcen anzugeben. Ein großer Vorteil von REST ist die lose Kopplung zwischen den Systemen.

Als Response auf einen Request wird meistens JSON, ein textuelles Datenformat, verwendet. JSON nutzt die Syntax von JavaScript: Objekte werden als Key-Value-Paare geschrieben. Binäre Daten wie z.B. Bilder können base64-kodiert ebenfalls übertragen werden. Außerdem wird oftmals XML verwendet.

Durch HTTP 2 soll die Bandbreite, die heutzutage verfügbar ist, besser genutzt werden. Somit sollen vor allem Ladezeiten beim Webbrowsing reduziert werden. Grundlage hierfür ist die neue binäre Übertragung der Daten, welche eine Unterteilung in einzelne Frames erleichtert. Durch neue Features wie z.B. Multiplexing kann die Kommunikation zwischen Server und Client weiter optimiert werden. Bestehende Webanwendungen können nach einer Aktualisierung des Webserver auf HTTP 2 wie bisher weiterverwendet werden, da die Grundsätze wie z.B. HTTP-Verben beibehalten werden. Auf bestehende REST-Schnittstellen hat diese Aktualisierung daher nur Auswirkungen im Performancebereich. Nachteil des neuen binären Protokolls ist, dass das Debugging erschwert wird. Allerdings gibt es beispielsweise Plugins für Wireshark, welche das Debugging von HTTP 2 ermöglichen.

Mittels OAuth ist es möglich, auf Ressourcen eines anderen Services nach Authentifizierung des Benutzers zuzugreifen. Somit kann beispielsweise ein Spiel den Facebook-Benutzeraccount verwenden, um den Highscore eines Benutzers zu speichern. Als Provider einer REST-API (in diesem Beispiel aus der Sicht von Facebook) sollte man seine Ressourcen gegen unautorisierte Zugriffe schützen, was ebenfalls durch OAuth garantiert wird. Letztendlich erhält der OAuth-Client immer einen Token, wodurch dieser Zugriff auf die API erlangt. Die Neuerungen in OAuth 2 beziehen sich vor allem auf eine strikere Trennung der Rollen und die Einführung von neuen Authorization Flows.

7 Literaturverzeichnis

- [1] O'Reilly – RESTful Web Services (2007); Leonard Richardson & Sam Ruby; ISBN: 978-0-596-52926-0
- [2] O'Reilly – REST API (2012); Mark Massè; ISBN: 978-1-449-31050-9
- [3] O'Reilly – RESTful Web APIs (2013); Leonard Richardson, Mike Amundsen & Sam Ruby; ISBN: 978-1-449-35806
- [4] An Introduction to OAuth2 (2014) [online]
verfügbar unter: <https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>
[zuletzt abgerufen am 20.10.2015]
- [5] Beginning JSON (2015); Ben Smith; ISBN: 978-1-4842-0203-6
- [6] O'Reilly – Getting Started with OAuth 2.0 (2012); Ryan Boyd; ISBN: 978-1-449-31160-5
- [7] O'Reilly – Browser Networking (2013); Ilya Grigorik; ISBN: 978-1-449-34476-4
- [8] Web API Design (2011) [online]; Brian Mulloy;
verfügbar unter: <https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf>
[zuletzt abgerufen am 20.10.2015]
- [9] O'Reilly – HTTP: The Definite Guide (2002); David Gourley & Brian Totty; ISBN: 978-1-56592-509-0
- [10] http2 explained (2015) [online]; Daniel Stenberg;
verfügbar unter: <http://http2-explained.haxx.se/>
[zuletzt abgerufen am 20.10.2015]
- [11] RFC 5849 (2010) [online]; Internet Engineering Task Force (IETF);
verfügbar unter: <https://tools.ietf.org/html/rfc5849>
[zuletzt abgerufen am 20.10.2015]
- [12] RFC 6749 (2012) [online]; Internet Engineering Task Force (IETF);
verfügbar unter: <https://tools.ietf.org/html/rfc6749>
[zuletzt abgerufen am 20.10.2015]
- [13] RFC 7540 (2015) [online]; Internet Engineering Task Force (IETF);
verfügbar unter: <https://tools.ietf.org/html/rfc7540>
[zuletzt abgerufen am 20.10.2015]
- [14] Spring REST (2015); Balaji Varanasi & Sudha Belida; ISBN: 978-1-4842-0824-3
- [15] dpunkt.verlag – REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web (2015); Stefan Tilkov, Martin Eigenbrodt, Silvia Schreier, Oliver Wolf; ISBN 978-3-86490-120-1

- [16] Apache mod_h2 Modul (2015), Github-Repository [Online];
verfügbar unter: https://github.com/icing/mod_h2
[zuletzt abgerufen am 27.10.2015]
- [17] NGINX Open Source 1.9.5 Released with HTTP/2 Support (2015) [Online]; Faisal Memon;
verfügbar unter: <https://www.nginx.com/blog/nginx-1-9-5/>
[zuletzt abgerufen am 27.10.2015]
- [18] HTTP/2 Protocol; CanIUse.com (2015);
verfügbar unter: <http://caniuse.com/#feat=http2>
[zuletzt abgerufen am 28.10.2015]
- [19] OAuth 2.0 Tutorial, Jakob Jenkov (2014)
verfügbar unter: <http://tutorials.jenkov.com/oauth2/index.html>
[zuletzt abgerufen am 14.11.2015]
- [20] Introducing OAuth 2, hueniverse (2010)
verfügbar unter: <http://hueniverse.com/2010/05/15/introducing-oauth-2-0/>
[zuletzt abgerufen am 14.11.2015]
- [21] JavaScript Object Notation; json.org (2015) [online]
verfügbar unter <http://json.org/>
[zuletzt abgerufen am 15.11.2015]
- [22] Google GSON Java Bibliothek (2015) [online]
verfügbar unter <https://github.com/google/gson>
[zuletzt abgerufen am 15.11.2015]
- [23] json-simple Java Bibliothek (2015) [online]
verfügbar unter <https://code.google.com/p/json-simple/>
[zuletzt abgerufen am 15.11.2015]
- [24] MDN JavaScript Reference (2015) [online]
verfügbar unter
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/JSON
[zuletzt abgerufen am 15.11.2015]
- [25] Python JSON Module (2015) [online]
verfügbar unter <https://docs.python.org/3/library/json.html>
[zuletzt abgerufen am 15.11.2015]
- [26] Apache Oltu, OAuth 2.0 Library; Apache (2013) [online]
verfügbar unter: <https://cwiki.apache.org/confluence/display/OLTU/Index>
[zuletzt abgerufen am 18.11.2015]
- [27] Oauth 2 Developers Guide, Spring; Pivotal Software (2015) [online]
verfügbar unter: <http://projects.spring.io/spring-security-oauth/docs/oauth2.html>
[zuletzt abgerufen am 18.11.2015]
- [28] API Client Libraries; Google (2015) [online]
verfügbar unter: <https://developers.google.com/api-client-library/>
[zuletzt abgerufen am 18.11.2015]

- [29] Python Social OAuth; Matias Aguirre (2015) [online]
verfügbar unter: <https://github.com/omab/python-social-auth>
[zuletzt abgerufen am 18.11.2015]
- [30] ScribeJava, Simple OAuth Library for Java; Garanina, Fromov (2015) [online]
verfügbar unter: <https://github.com/scribejava/scribejava>
[zuletzt abgerufen am 18.11.2015]
- [31] Richardson's Maturity Model; Martin Fowler (2015) [online]
verfügbar unter: <http://martinfowler.com/articles/richardsonMaturityModel.html>
[zuletzt abgerufen am 18.11.2015]
- [32] Richardson's Maturity Model; Leonard Richardson (2015) [online]
verfügbar unter: <http://www.crummy.com/writing/speaking/2008-QCon/act3.html>
[zuletzt abgerufen am 18.11.2015]
- [33] Secure your REST API; Sormpath (2015) [online]
verfügbar unter: <https://stormpath.com/blog/secure-your-rest-api-right-way/>
[zuletzt abgerufen am 18.11.2015]
- [34] Top Six Reasons to Use API Keys; Sormpath (2015) [online]
verfügbar unter: <https://stormpath.com/blog/top-six-reasons-use-api-keys-and-how/>
[zuletzt abgerufen am 18.11.2015]
- [35] Spring REST Example; Spring Framework (2015) [online]
verfügbar unter: <https://spring.io/guides/gs/rest-service/>
[zuletzt abgerufen am 18.11.2015]
- [36] Django REST Framework; Django (2015) [online]
verfügbar unter: <http://www.django-rest-framework.org/>
[zuletzt abgerufen am 18.11.2015]
- [37] Lightweight REST Framework for Java; RESTX (2015) [online]
verfügbar unter: <http://restx.io/>
[zuletzt abgerufen am 18.11.2015]
- [38] Lightweight REST Framework for PHP; Slim Framework (2015) [online]
verfügbar unter: <http://www.slimframework.com/>
[zuletzt abgerufen am 18.11.2015]
- [39] Lightweight REST Framework for Python; CherryPy (2015) [online]
verfügbar unter: <http://cherrypy.org/>
[zuletzt abgerufen am 18.11.2015]
- [40] HTTP/2 on IIS; David So's Blog; (2015) [online]
verfügbar unter: <http://blogs.iis.net/davidso/http2>
[zuletzt abgerufen am 29.11.2015]

8 Abbildungsverzeichnis

Abbildung 1: Struktur eines JSON Objekts [21].....	15
Abbildung 2: Struktur eines JSON Arrays [21].....	15
Abbildung 3: Verfügbare Datentypen in JSON [21]	16
Abbildung 4: Struktur eines Strings in JSON [21]	16
Abbildung 5: HTTP 2 Verbindung.....	
Abbildung 6: HTTP 2 Binary Framing.....	19
Abbildung 7: HTTP 2 Multiplexing [7]	20
Abbildung 8: Header Compression [7]	20
Abbildung 9: Server Push [7].....	21
Abbildung 10: Browser-Kompatibilität mit HTTP 2 [18]	24
Abbildung 11: Rollen in OAuth 2 [19]	26
Abbildung 12: Authorization Code Flow [6]	27
Abbildung 13: Implicit Grant Flow [6]	28
Abbildung 14: Ressource Owner Password Flow [6]	29
Abbildung 15: Client Credentials Flow [6].....	30