

Multimedia Engineering II

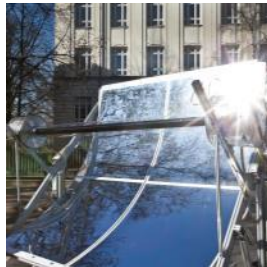
03 APIs und RESTful Design

Johannes Konert



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences



Agenda

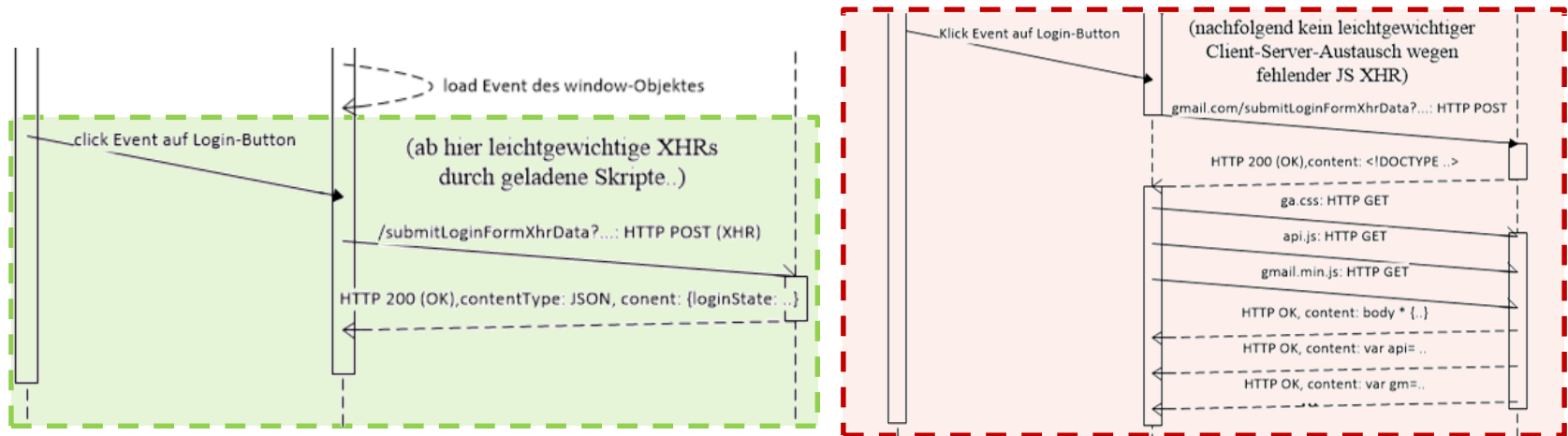
- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

Rückblick / Wiederholung

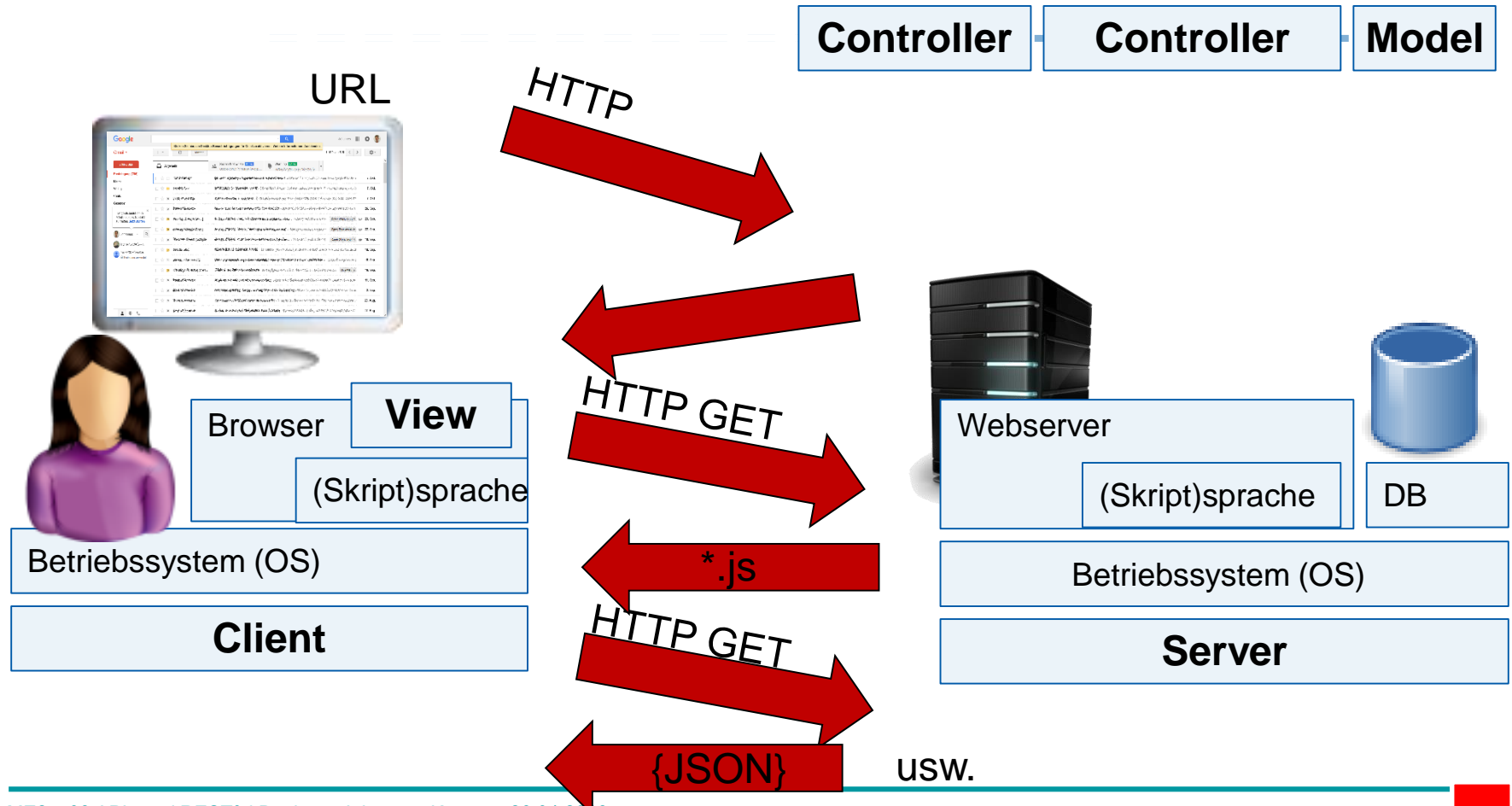
- Was sind die wesentlichen Unterschiede beim Laden der Webseite zwischen Thin-Client und Rich-Client Prinzip?
 - Klausur: ggf. „Beschriften Sie ein Sequenzdiagramm dazu“



Letzte Woche: Client-Server Architektur

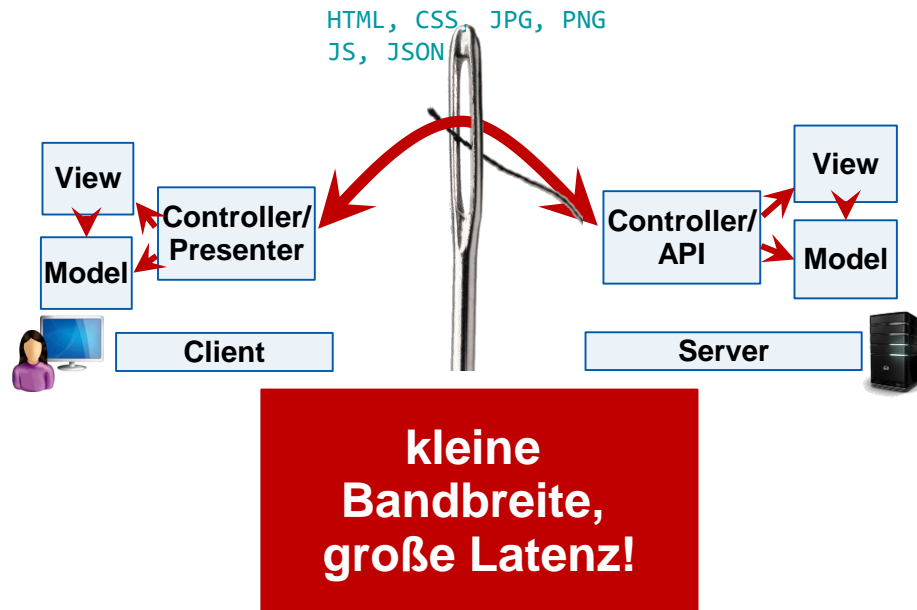
- **Rich-Client-Prinzip** (Beispiel: Szenario der SinglePageApp wie Gmail)

```
<!DOCTYPE html>
<html>..<script>..</script>..</html>
```



Rückblick / Wiederholung

- Was ist das Client-Server „Nadelöhr“ – Problem?



- Lösung: Asynchrone, leichtgewichtige Kommunikation via AJAX (als JSON Daten)

Letzte Woche: Client-Server Architektur



| Datenbank | Webserver | Full Solution Framework | Server: Web App Framework (+Sprache) | Template Engine | Client: Web App Framework |
|-----------|---|-------------------------|---|--|--|
| | Performance <ul style="list-style-type: none"> statische Dateien Streaming (Skript)-sprachen | | <ul style="list-style-type: none"> Entwicklungsgeschwindigkeit Modularisierung Knowhow eigener Entwickler/in | <ul style="list-style-type: none"> Komplexität Modularisierung Verbindung mit Skriptsprache | <ul style="list-style-type: none"> Größe (kb) Binding Modularisierung Kompatibilität |

■ Generelle Kriterien für alle SW-Produkte

- Speicherverbrauch
- Stabilität
- Sicherheit
- Geschwindigkeit
- Dokumentation
- Weiterentwicklung/Community

Entscheidung hängt vom Anwendungsszenario ab!

Rückblick / Wiederholung

- **Warum ist für eine SPA auf Server-Seite eine Verwendung verschiedener Komponenten für Webserver und (Skript)-Sprache nachteilig?**
- Auch das initial ausgelieferte Dokument ggf. schon via Skriptsprache dynamisch erstellt wird
- Laufzeitumgebung des Webserver != Laufzeitumgebung der Skriptsprache → mehr Speicherverbraucht und langsamere Reaktion
- Nach dem initialen Laden des Dokumentes (DOM)
 - überwiegend dynamische Inhalte via XHR geladen werden
 - JSON via API (idealerweise REST-API)
 - kleine Templatebausteine

Agenda

- **Wiederholung**

- **APIs**

- **Web APIs**

- **Von „vor REST“ nach REST**

- **REST Level**

- **REST Richtlinien und Best Practice**

- **REST generieren, testen, dokumentieren**

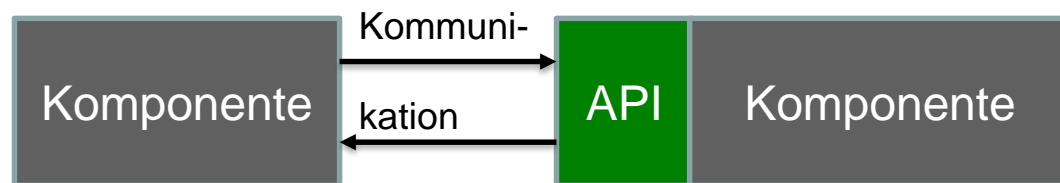
- **Zusammenfassende Fragen**

- **Ausblick: Nächstes Mal**

API

Application Programming Interface

- Schnittstelle zur Anbindung verschiedener Hard- und Software-Komponenten untereinander
- Ermöglicht die Kommunikation und dient somit als Bindeglied



API

■ Welche der folgenden Elemente haben eine API?

Bibliotheken (Libs)

Betriebssysteme

Datenbanken

Webserver

Klassen (Java)

Webseiten

Webanwendungen (SPA)



API

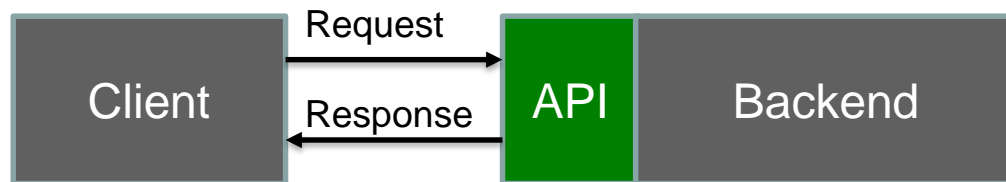
■ Welche der folgenden Elemente haben eine API?

| | |
|----------------------|--|
| Bibliotheken (Libs) | Interface-Implementierung |
| Betriebssysteme | Syscalls |
| Datenbanken | Abfragesprache (wie SQL) |
| Webserver | Pluginschnittstelle und Protokollschnittstelle (wie HTTP) |
| Klassen (Java) | Interfaces/Signaturen |
| Webseiten | Clientseitig: Via DOM im Browser (schwach strukturiert) |
| Webanwendungen (SPA) | Serverseitig: Datenschnittstelle |

- **Alle.**
APIs sind also „überall“ in unterschiedlichster Form definiert.

APIs im Web

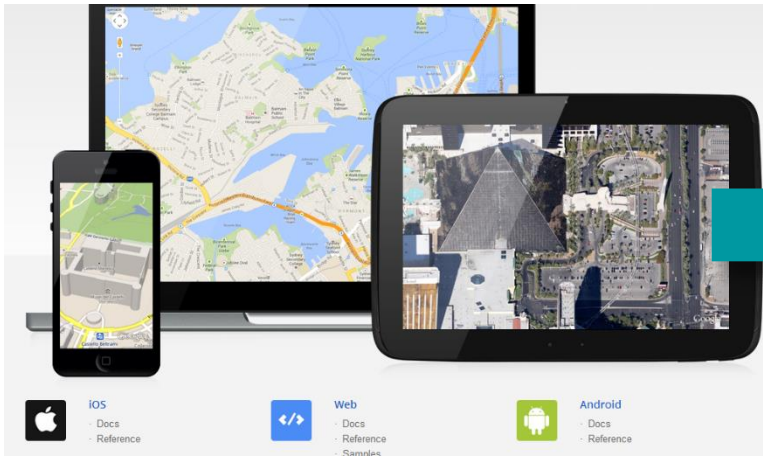
- **Besondere Bedeutung beim Datenaustausch**
 - **Client-Seite (Frontend) und Server-Seite (Backend)**
 - Abruf/Speichern von Daten und Zuständen
 - **von Anwendungen untereinander**
 - Nutzung und Verknüpfung von Datendiensten (Google Maps, Instagram, Facebook, DropBox, ..)



API Beispiele

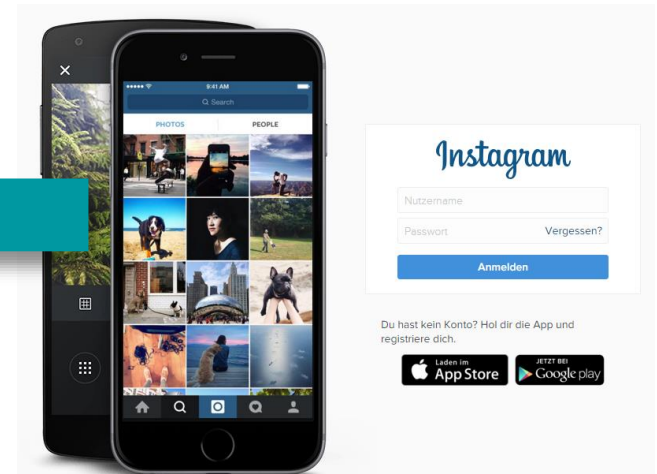
Welche Instagram Bilder wurden in der Nähe geschossen?

■ Google Maps



<https://developers.google.com/maps/>

■ Instagram



<https://instagram.com/developer/>

■ Developer-Seiten enthalten API-Doku und Beispielcode

API Beispiele

Welche Instagram Bilder wurden in der Nähe geschossen?

■ Google Maps + Instagram

1. Long/Lat Geokoordinaten abrufen

<https://maps.googleapis.com/maps/api/geocode/json?address=Berlin>

2. Instagram abfragen

<https://api.instagram.com/v1/media/search?lat=...&lng=...> **

3. Bilder anzeigen

API – Was sind die Merkmale guter APIs?

- APIs sind wie Kleber, der die losen Systeme zusammenfügt.



API – Was sind die Merkmale guter APIs?

- **Gute Dokumentation (idealerweise selbsterklärend)**
 - **Sonst praktisch unbrauchbar**
- **Langfristig**
 - **Stabile Schnittstelle auf die „gebaut“ werden kann**
- **Unabhängig**
 - **Kann von verschiedenen Endsystemen genutzt werden**
- **Erweiterbar**
 - **Versionierung für zukünftige Veränderungen**

= **GLUE** (engl. für Kleber, ..der die Sachen zusammenhält)

Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

Web APIs

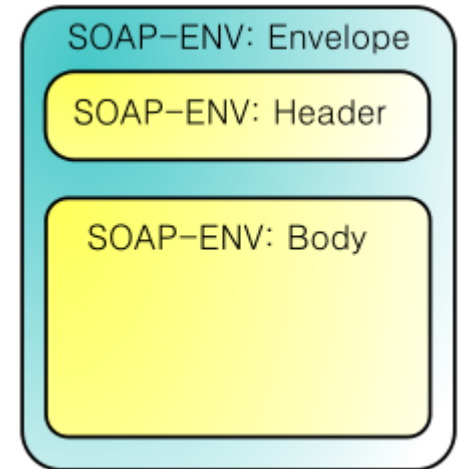
- **SOAP**
- **WSDL**
- **REST**

Web APIs

■ SOAP: Simple Object Access Protocol

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"
```

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock/Surya">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```



■ Fazit: Totaler Overhead

Web APIs

■ SOAP



■ WSDL: Web Service Description Language

- Wenn SOAP das Format für den Datenaustausch ist, dann ist WSDL die Sprache (on top) zur Beschreibung der Service-Schnittstellen und wie SOAP auszusehen hat

```
<!-- Concrete Binding with SOAP-->
```

```
<binding name="SoapBinding" interface="tns:Interface1"
  type="http://www.w3.org/ns/wsdl/soap"
  wssoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
  wssoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
  <operation ref="tns:Get" />
</binding>
```





```
<!-- Web Service offering endpoints for both bindings-->
```

```
<service name="Service1" interface="tns:Interface1">
  <endpoint name="HttpEndpoint"
    binding="tns:HttpBinding"
    address="http://www.example.com/rest/" />
  <endpoint name="SoapEndpoint"
    binding="tns:SoapBinding"
    address="http://www.example.com/soap/" />
</service>
```

```
</definitions>
```

■ Fazit: Gedacht für das Semantic Web, leider sehr komplex

Web APIs

- SOAP 
- WSDL 
- REST: Representational State Transfer
 - Definiert Regeln zur Gestaltung des Zugriffs auf Ressourcen
- Systeme, die REST-Regeln einhalten werden RESTful genannt

Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

REST

■ „typische“ WEB APIs vor REST

HTTP GET /api/tweetService.php?
action=list&entities=retweets&uid=33245&items=all

Gib mir alle Retweets des Nutzers 33245

HTTP GET /api/tweetService.php?action=delete&tid=332456

Lösche den Tweet 332456

Problematisch

- Verständlichkeit der API ist niedrig
- Parameter nicht standardisiert
- Bis auf HTTP GET und HTTP POST wird keine Methode verwendet

REST

■ „typische“ WEB APIs vor REST

HTTP GET /api/tweetService.php?
action=list&entities=retweets&uid=33245&items=all

Gib mir alle Retweets des Nutzers 33245

HTTP GET /api/tweetService.php?action=delete&tid=332456

Lösche den Tweet 332456

■ mit REST (zum Beispiel)

HTTP GET
/api/users/33245/retweets

Gib mir alle Retweets des Nutzers 33245

HTTP DELETE
/api/tweets/332456

Lösche den Tweet 332456

REST

HTTP GET

/api/users/33245/retweets

Gib mir alle Retweets des Nutzers 33245

HTTP DELETE

/api/tweets/332456

Lösche den Tweet 332456

- **Nutzung von HTTP Methoden für Operationen**
- **Nutzung von URLs für Ressourcenidentifikation**
- **Zustandslos**
 - Keine Sessions
 - Alle Anfragen enthalten die notwendigen Informationen komplett



REST konkret

- Operationen: HTTP Methoden
- Ressourcen-ID: URLs
- Zustandslos



- Beispiel-Ressourcen

- Tweets
- Users
- Comments



= Einzelressourcen,
Collections,
1:n Beziehungen,
n:n Beziehungen

- Operationen

- 1,2,3,4

**Aufgabe: Welche vier
Datenoperationen gibt es**



- (1min) Sammeln Sie diese
im kleinen Team
- Bis zu vier Teams kommen dann dran

REST konkret

- Operationen: HTTP Methoden
- Ressourcen-ID: URLs
- Zustandslos



- Beispiel-Ressourcen

- Tweets
- Users
- Comments



- Operationen

- Erstellen (Create)
- Lesen (Read)
- Aktualisieren (Update)
- Löschen (Delete)

= auch bekannt als CRUD



REST Operationen: Beziehung von CRUD und HTTP

| CRUD | HTTP Methode |
|--------|--------------|
| Create | POST |
| Read | GET |
| Update | PUT ** |
| Delete | DELETE |

- **Achtung: Alle HTTP-Methoden sind als **idempotent** definiert (außer HTTP POST), also GET, PUT, DELETE**
 - Mehrfacher Aufruf ändert nicht den Systemzustand
- **Achtung2: REST erfordert **Zustandslosigkeit**, also wie einen Datensatz nur teilweise aktualisieren?**
 - **HTTP PATCH statt PUT** (PATCH ist nicht zwingend idempotent)

REST Zusammenhänge: Beispiele

| Text | Altes API Konzept | REST API |
|---|---|--|
| Alle Nutzer | GET /api/users.php?action=list&items=all | GET /api/users |
| Alle Daten des Nutzers a1b2c3 | GET /api/users.php?action=profile&uid=a1b2c3 | GET /api/users/a1b2c3 |
| Aktualisiere Nutzerdaten von a1b2c3 mit Hobby=Malen | POST /api/users?action=update BODY FORM-data: uid=a1b2c3, hobby=Malen, submit=true | PUT /api/users/a1b2c3 BODY: { name: Max Muster, gebdat: 11.12.1986, hobby: Malen, pw: '*****' } |
| “ ” | “ ” | PATCH /api/users/a1b2c3 BODY: { hobby: Malen} |
| Lösche Nutzer a1b2c3 | POST /api/users?action=delete BODY FORM-data: uid=a1b2c3, submit=true | DELETE /api/users/a1b2c3 |

REST Zusammenhänge: Beispiele

| Text | REST API |
|-------------------------|---|
| | GET /api/users/a1b2c3/tweets |
| Lösche den Tweet d4e5f6 | |
| | POST /api/followRelations/ BODY { userFrom: a1b2c3, userTo: c3b2a1 } |



REST Zusammenhänge: Beispiele

| Text | REST API |
|---|---|
| Gib mir alle Tweets des Nutzers a1b2c3 | GET /api/users/a1b2c3/tweets |
| Lösche den Tweet d4e5f6 | |
| | POST /api/followRelations/ BODY { userFrom: a1b2c3, userTo: c3b2a1 } |



REST Zusammenhänge: Beispiele

| Text | REST API |
|--|---|
| Gib mir alle Tweets des Nutzers a1b2c3 | GET /api/users/a1b2c3/tweets |
| Lösche den Tweet d4e5f6 | DELETE /api/tweets/d4e5f6 |
| | POST /api/followRelations/ BODY { userFrom: a1b2c3, userTo: c3b2a1 } |



REST Zusammenhänge: Beispiele

| Text | REST API |
|---|---|
| Gib mir alle Tweets des Nutzers a1b2c3 | GET /api/users/a1b2c3/tweets |
| Lösche den Tweet d4e5f6 | DELETE /api/tweets/d4e5f6 |
| Erstelle eine neue Follow-Beziehung von Nutzer a1b2c3 zu Nutzer c3b2a1 | POST /api/followRelations/ BODY { userFrom: a1b2c3, userTo: c3b2a1 } |

Antwort bei POST:

HTTP Status: 201 (Created)

Location: /api/followRelations/**k6l7m8**

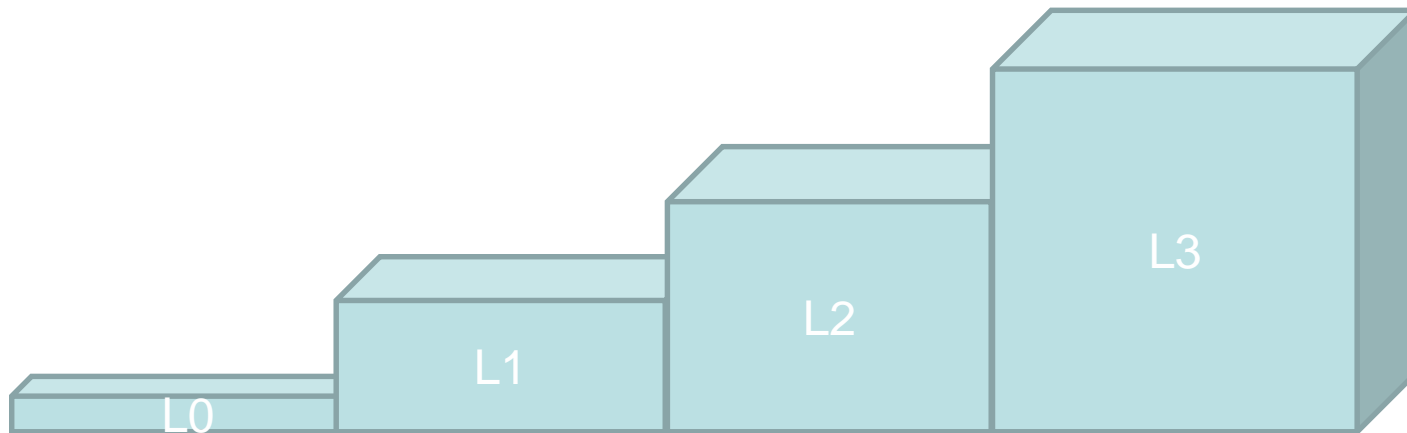
Body: leer!

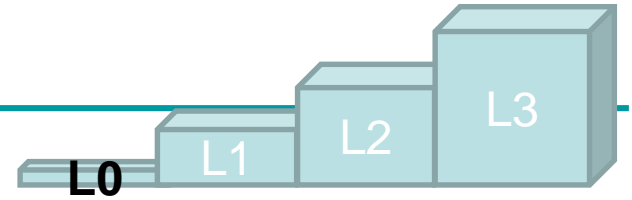
Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

REST Level

Unterteilung in REST-Level nach dem REST Maturity Model (RMM) von Leonard Richardson





REST Level 0

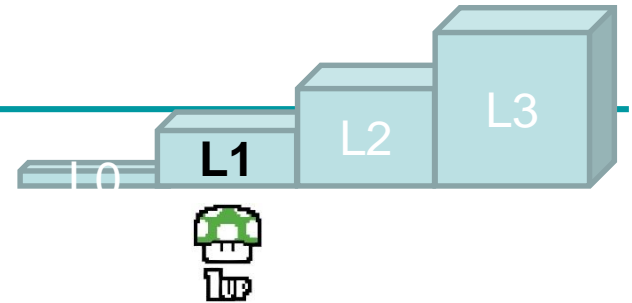
Konzept der „Remote Procedure Call“

- Einfacher Aufruf einer Serviceschnittstelle
- `POST /api/warehouseService`

Im Body werden die Befehle
(Operationen, Ressourcen-Parameter, Daten)
übergeben z.B.:

- `customer=1234`
- `orders=all`
- `action=update`
- `paid = true`
- `..`

REST Level 1

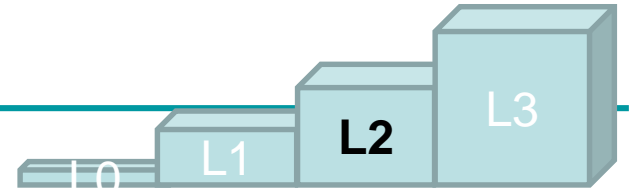


Konzept der „Ressourcen-Zeiger“

- Einfacher Aufruf einer Serviceschnittstelle
- `GET /api/customers/1234`

Im Body werden immer noch die Anweisungen (Operationen, Daten) übergeben, aber Ressource via URL identifiziert:

- `orders=all`
- `action=list`



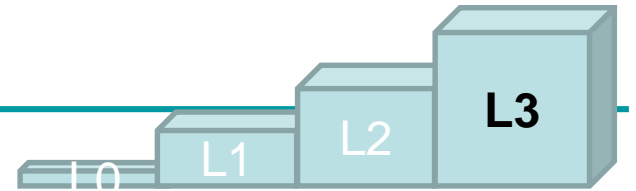
REST Level 2

Konzept der Navigation über Ressourcen-URLs und Manipulation über HTTP-Methoden

- **GET /api/customers/1234/orders**

Im Body keine Anweisungen!

Der Body enthält (bei PUT/POST/PATCH) auszutauschende Objekt-Daten.



REST Level 3

Konzept der „State machine“

**Hypermedia as the engine of application state
(HATEOAS)**

- **GET** `/api/customers/1234/orders`

Im Body keine Anweisungen

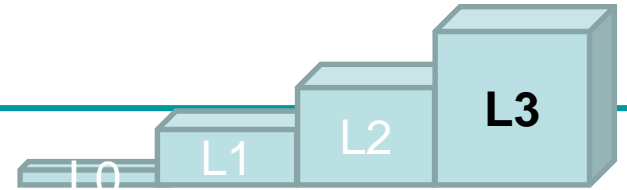
Der Body enthält (bei PUT/POST/PATCH) auszutauschende Objekt-Daten.

Die HTTP Antwort enthält

- **angefragten Objekte,**
- **Verweise auf weitere Operationen und Ressourcen**

(Client kann komplett dynamisch anhand der Daten operieren/navigieren)





REST Level 3

Konzept der „State machine“

Hypermedia as the engine of application state
(HATEOAS)

■ GET /api/customers/1234

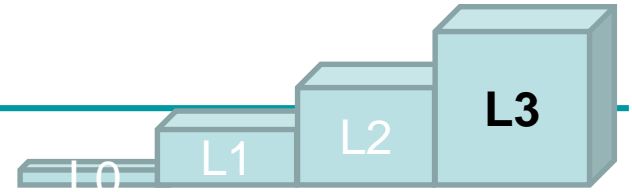
{ Antwort:

```
{  "href": "https://localhost/api/customers/1234",  
  "name": "Susan Sunny",  
  ...
```

Ressource

```
  "links": [ {  
    "rel": "self",  
    "href": "/api/customers/1234"  
  }, {  
    "rel": "customers.orders",  
    "href": "/api/customers/1234/orders"  
  }  
  ...
```

Verweise (Ressourcen und Operationen)



REST Level 3

HATEOAS

- Laut REST Spezifikation Voraussetzung für RESTful
- Benutzt jedoch kaum einer vollständig
 - Guter Stil ist die Nutzung von self-Relations als { „href“: .. } Attribute in der Antwort
- REST Level 2 reicht! (+ Dokumentation)

Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

REST Design Richtlinien

- 1. Rückgabetypp**
- 2. API Versionierung**
- 3. HTTP Status und Fehler**
- 4. Sicherheit**
- 5. Filtern und Blättern**
- 6. Verweise und Expansion**

Ein möglicher Aufruf

```
GET https://localhost/api/v1/tweets.json/b3f4d5/retweets?
3 4 2 1 6
contains=BeuthHS&offset=50&limit=25
5
```

1. REST Rückgabetypen

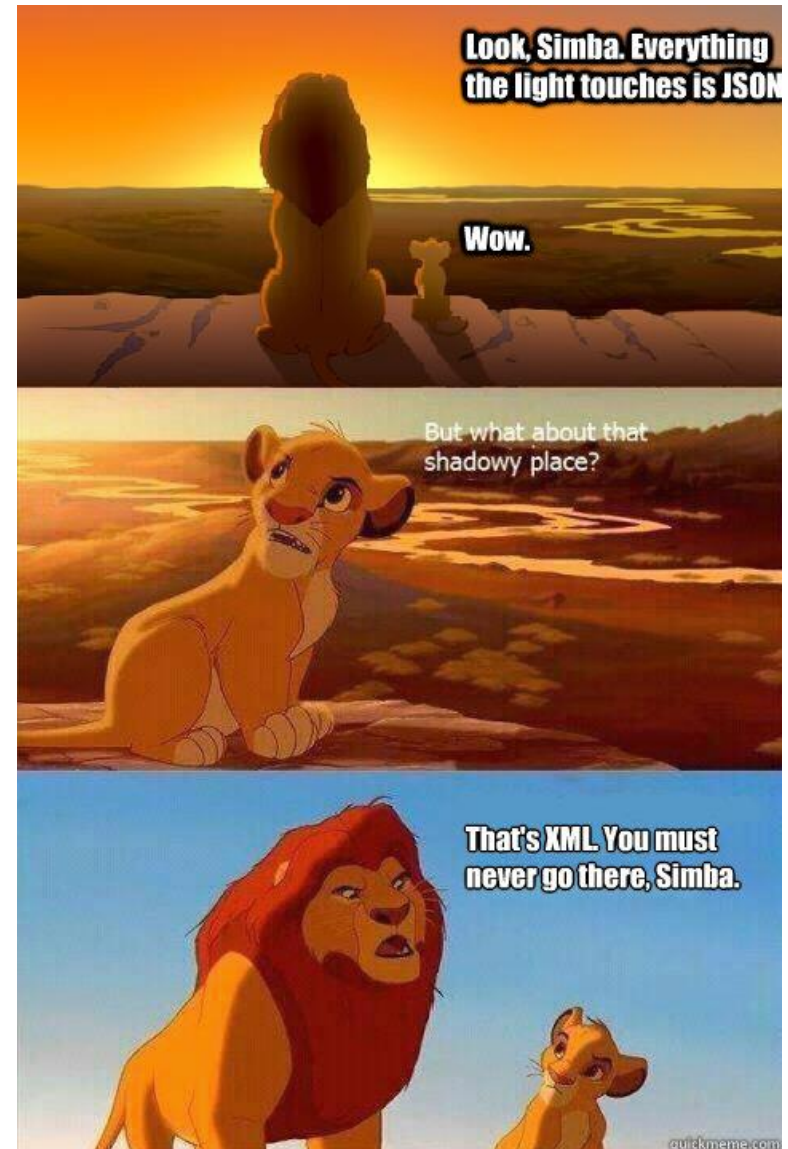
- REST trennt Ressourcen, Operationen und Repräsentationen



- Möglichkeiten
 - a. In der URL: `/api/tweets.json`
 - b. HTTP Header Feld: „Accept: application/json, text/csv, text/plain, text/html“ (sortiert nach Präferenz)
- Antwort vom Server, z.B.
 - HTTP Header Feld: „Content-Type: application/json“
 - oder Fehler: **HTTP Status 406 Not Acceptable** (Ressource in der Media Form nicht verfügbar)

1. REST Rückgabetypen

- JSON ist Standard-MIME-Type
- Einfach als HTTP Header
 - **Accept** und
 - **Content-Type**setzen und dann ist gut
(weitere Formate nicht nötig)



2. REST API Versionierung

- **Problem: Ihre API wird geändert und liefert ggf. andere Inhalte zurück.**
- **API einfach ändern? Alle Nutzer/Kunden werden sauer**
- **Möglichkeiten**
 - a. In der URL: **/api/v1/tweets**
 - b. Über HTTP Header Feld „**Accept-Version: 1.0**“
 - c. Über HTTP Header Feld Accept mit eigenem Mimetype: „**Accept: application/v1+json, text/v1+plain**“
- **Derzeit wechseln einige (wie Github.com) von a. zu c.**
<https://developer.github.com/v3/>

2. REST: HTTP Status Codes und API-Fehler

- 2xx alles ok
- 3xx keine Veränderungen
- 4xx dein Fehler
- 5xx mein Fehler :)

3. REST: HTTP Status Codes und API-Fehler

| Code | Titel | Beschreibung | Anwendung | Body |
|------|------------------------|--|--|--|
| 200 | OK | Anfrage gültig | GET, PUT, DELETE | Angefragte, bzw. veränderte Ressource, leer bei DELETE |
| 201 | Created | Ressource erstellt | POST | Angelegte Ressource |
| 204 | No Content | Anfrage gültig | DELETE | Leer! |
| 400 | Bad Request | Unzureichende Anfrage | PUT, POST – fehlende Parameter | Fehlermeldung |
| 401 | Unauthorized | Fehlende Authentifizierung | Alle Routen, die nicht öffentlich sind | Fehlermeldung |
| 403 | Forbidden | Unzureichende Rechte | Alle Routen, die höhere Freigabe erfordern | Fehlermeldung |
| 404 | Not Found | Ressource nicht gefunden | Alle Routen, wenn ID ungültig ist | Fehlermeldung |
| 406 | Not Acceptable | Accept: mime types können nicht bedient werden | GET | Leer! im Header Content-Type: listet Möglichkeiten |
| 415 | Unsupported Media Type | Falscher Content-Type | POST, PUT | Leer! |

4. REST Sicherheit

- REST ist zustandslos, wie also „erst einloggen“ ?
- Cookies? Keine gute Idee, können kopiert werden
- Sessions: Zustand auf dem Server! Also auch nicht
- **Lösung:** API-Accesskeys (od. Tokens) mitgeben
 - a. In der URL: `/api/v1/tweets?access_token=b2k3h5u65b3o5b3tg`
 - b. Im HTTP Header (am Beispiel OAuth 1.0a):
`Authorization: OAuth realm="http://localhost",
oauth_consumer_key="0685bd9184jfhq22",
oauth_token="ad180jjd733klru7",
oauth_timestamp="137131200"`
- Lösung b. über Header ist aktuell de facto Standard + HTTPS natürlich



5. REST: Filtern und Blättern

- **Problem: GET /api/tweets liefert 2 Milliarden Einträge?**

- **Lösung: Wie bei SQL Blättern mit Offset und Limit**
GET /api/tweets?offset=50&limit=25

- Liefert nur die 25 Einträge ab Eintrag 50 (also Seite 3)

- { ■ **Elegante JSON Antwort** (mit etwas **HATEOAS**)

```
„href“: „http://localhost/api/tweets?offset=50&limit=25“,  
„items“: [{  
    „href“: „http://localhost/api/tweets/b4n3d5“,  
    „text“: „This is my first tweet“,  
    „owner“: „http://localhost/api/users/a1b2c3“,  
    }, ...  
],  
„offset“: 50,  
„limit“: 25,  
„first“: „http://localhost/api/tweets?limit=25“,  
„next“ : „http://localhost/api/tweets?offset=75&limit=25“,  
„previous“: ...
```

5. REST: Filtern und Blättern

2 Arten von Filtern (muss man in die API Doku schauen)

I. Rückgabe auf **bestimmte Felder** beschränken

GET /api/tweets?fields=text,date

II. Such-Parameter

GET /api/tweets?contains=BeuthHS

Beispiel-Antwort:

```
{
  „href“: „http://localhost/api/tweets?fields=text,date&contains=BeuthHS“,
  „items“: [{
    „text“: „This is my first tweet for BeuthHS“,
    „date“: „Sat, 24 Oct 2015 19:43:38 GMT“,
  },
  „text“: „Today at BeuthHS we had ME2 exercise“,
  „date“: „Thu, 22 Oct 2015 22:43:38 GMT“,
  }, ...
]
```

6. REST: Verweise und Expansion

■ Verweise aus 1:n und n:n **Relationen mit Auflisten**

```
{  
  „href“: „http://localhost/api/tweets/h2j8k4s“,  
  „text“: „This is my first tweet for BeuthHS“,  
  „date“: „Sat, 24 Oct 2015 19:43:38 GMT“,  
  „owner“: „http://localhost/api/users/a1b2c3“,  
  „retweets“: {  
    „href“: „http://localhost/api/tweets/h2j8k4s/retweets“  
  }  
  ...  
}
```

6. REST: Verweise und Expansion

- Verweise aus 1:n und n:n **Relationen mit Auflisten**
..und mit **?expand=retweets** direkt mit Befüllen

```
{
  „href“: „http://localhost/api/tweets/h2j8k4s“,
  „text“: „This is my first tweet for BeuthHS“,
  „date“: „Sat, 24 Oct 2015 19:43:38 GMT“,
  „owner“: „http://localhost/api/users/a1b2c3“,
  „retweets“: {
    „href“: „http://localhost/api/tweets/h2j8k4s/retweets“,
    „items“: [{
      „href“: „http://localhost/api/tweets/k3s4a3“,
      „date“: „Sat, 24 Oct 2015 20:11:08 GMT“,
      „owner“: „http://localhost/api/users/b2c3d4“,
    }, ...
  ]
}
...
}
```

Kleiner Test für REST



■ Übungsaufgabe

■ Einzel (2min)

Ordnen Sie die HTTP Methoden sinnvoll den REST URLs zu
(Mehrfach möglich)

- | | |
|-----------|--|
| 1) GET | a) http://api.youtube.com/users |
| 2) POST | b) http://api.youtube.com/users/i4h5k9 |
| 3) PUT | c) http://api.youtube.com/channels/b3k4o9/videos?sort=latest&limit=10 |
| 4) PATCH | |
| 5) DELETE | |

Gültige Lösungen:

1-a, 1-b, 1-c

2-a (POST erstellt eine ID, daher b/c Unsinn)

3-b~ (a falsch ohne ID, c mit GET-Parametern unsinnig)

4-b,a~ (wie 3, nur statt aller PUT-Daten, nur Änderungen als PATCH senden.
a geht auch, wenn bspw. alle user um bestimmte Daten ergänzt werden
sollen (Admin-Aufruf))

5-a**, 5-b

** wird in der Praxis keine Berechtigung haben, formal ist das aber gültig.
~ wird nur der Nutzende i4h5k9 selbst dürfen)

Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

REST Schnittstellen Generieren, Testen, Dokumentieren

■ Generieren

- REST Beschreibung
→ Code Generator
- Frameworks für Annotationen
- Frameworks für REST-Layer

■ Testen

- Client-seitig
- Entwickler-Tools
(Browser)
- Teilautomatische Tests

■ Dokumentieren

- REST Beschreibung
→ Code Generator
- Frameworks für
Annotationen
- Frameworks für
REST-Layer

REST Testen

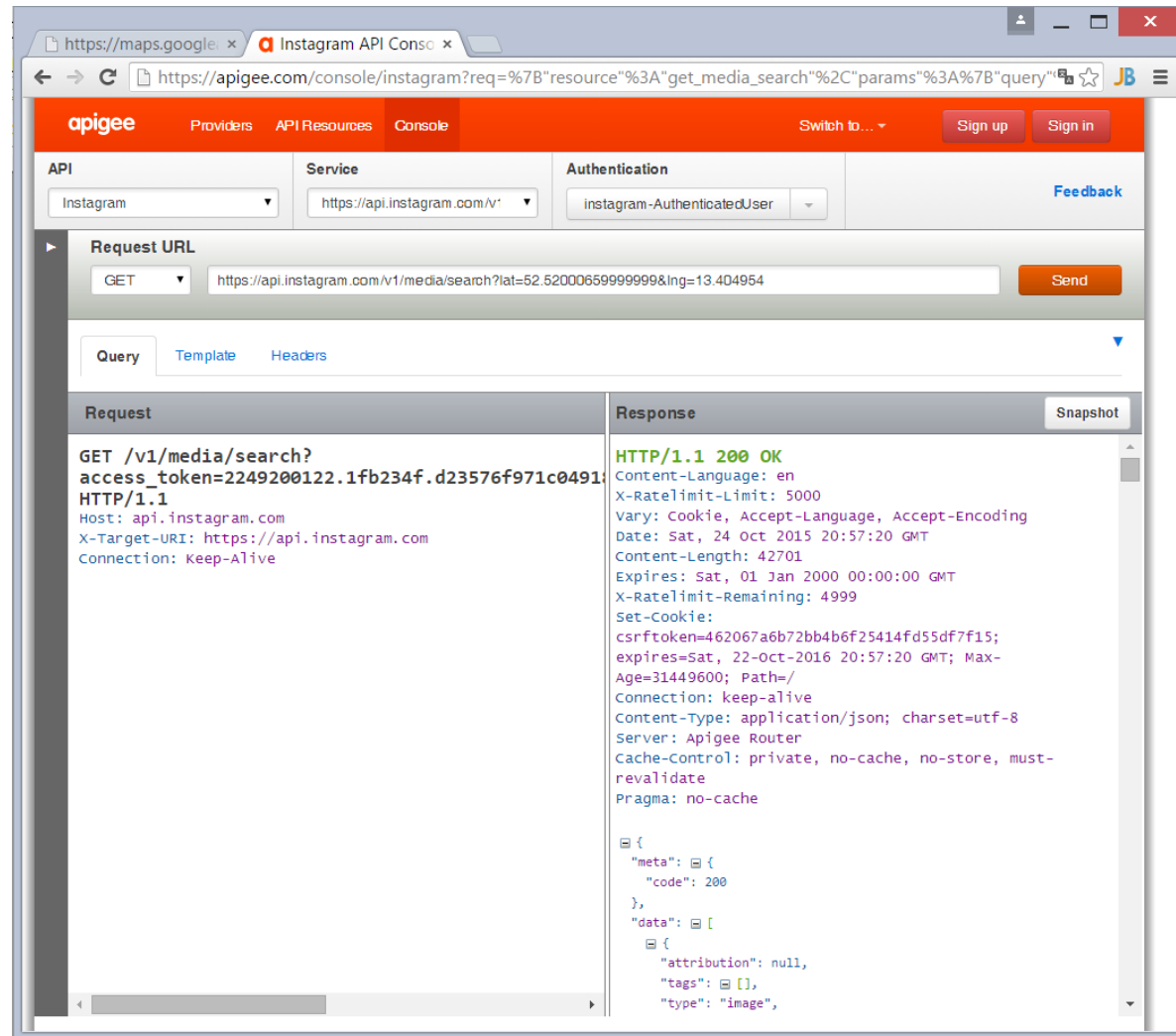
- **Restify.io: Leichtgewichtiges Online-Tool**



restify.io

REST Testen

- **Apigee.com/console**
 - Für viele große US-Anbieter auch Parameter bereits mit dokumentiert



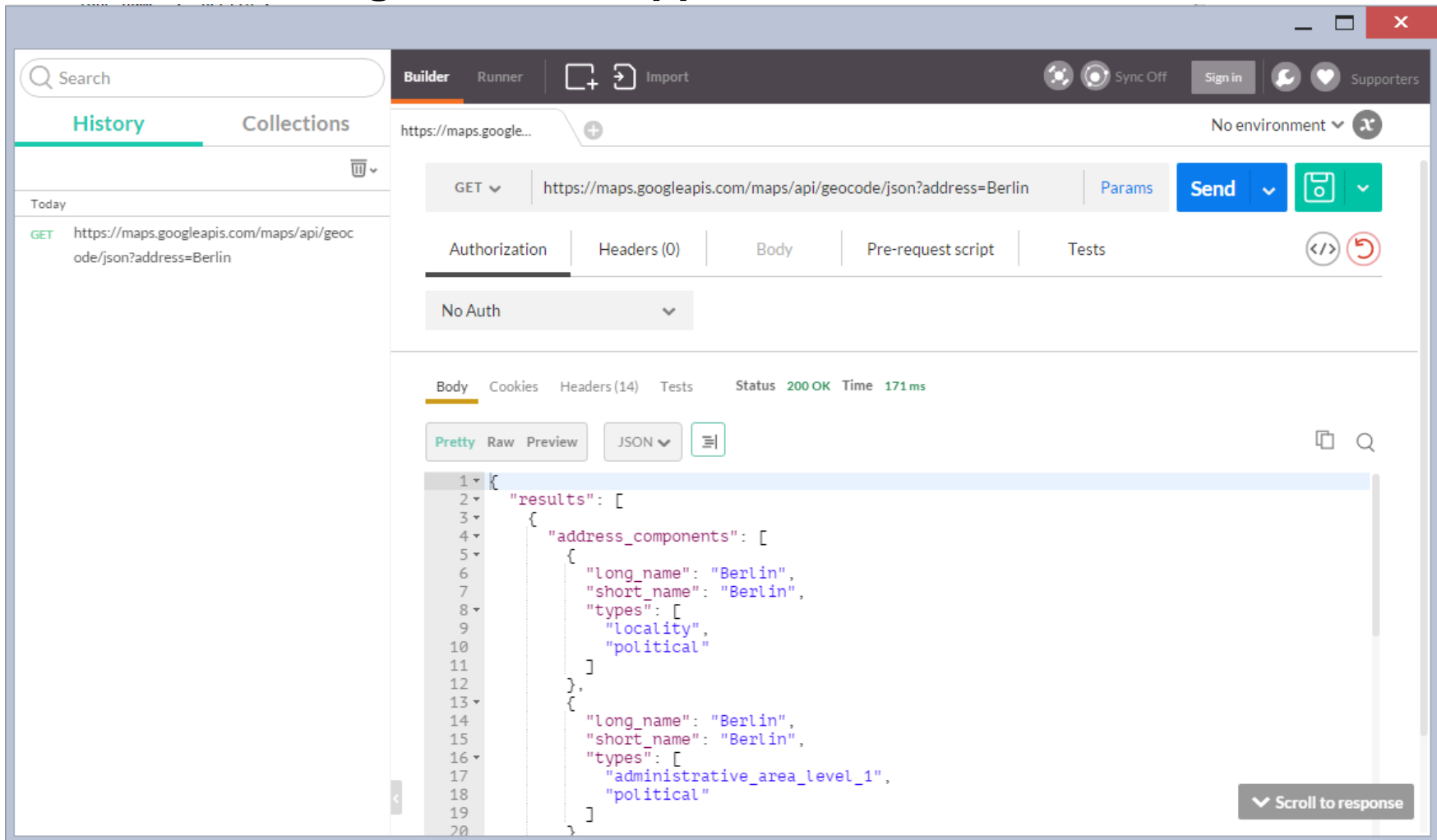
REST Testen

■ Postman Google Chrome App



Welcome to Postman 3.0

A great new experience, jam-packed with features



The screenshot displays the Postman Google Chrome App interface. On the left, the 'History' tab shows a recent GET request to `https://maps.googleapis.com/maps/api/geocode/json?address=Berlin`. The main workspace shows the 'Builder' tab with the same URL. The 'Send' button is highlighted. Below the URL bar, the 'Authorization' tab is selected, showing 'No Auth'. The 'Body' tab is also visible. The response is displayed in the 'Body' tab, showing a JSON object with 'results' array containing address components for Berlin. The status is '200 OK' and the time is '171 ms'.

```
1 {
2   "results": [
3     {
4       "address_components": [
5         {
6           "long_name": "Berlin",
7           "short_name": "Berlin",
8           "types": [
9             "locality",
10            "political"
11          ]
12        },
13        {
14          "long_name": "Berlin",
15          "short_name": "Berlin",
16          "types": [
17            "administrative_area_level_1",
18            "political"
19          ]
20        }
21      ]
22    }
23  ]
24 }
```

REST Testen

- **Frisby on Jasmine oder Mocha für REST**

- **Test-Case Definitionen
(so wie bei JUnit-Tests)**

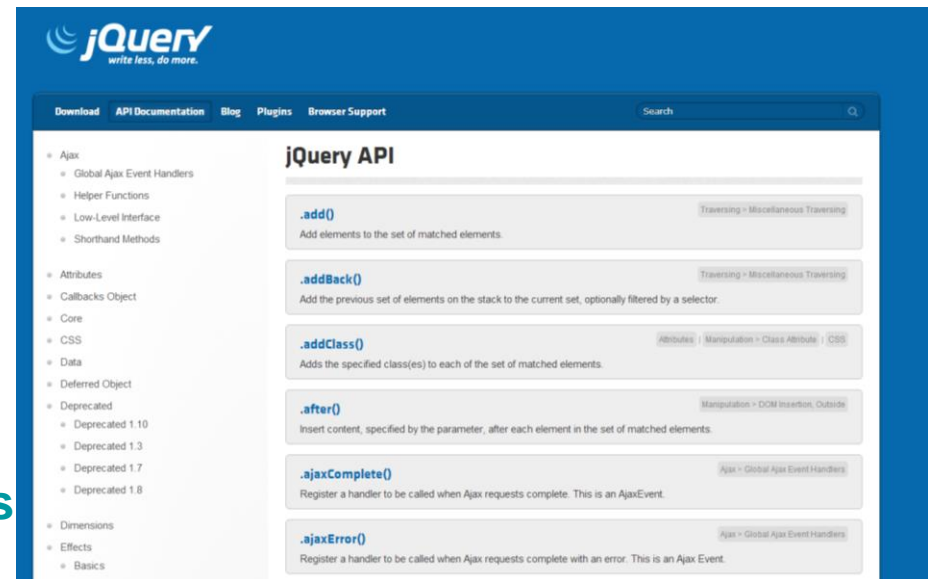


- **Werden wir im 4. Übungsblatt nutzen**

```
describe('Clean /video REST API ', function() {  
  it('should send status 204 and empty body ', function (done) {  
    request(videoURL)  
      .get('/')  
      .set('Accept-Version', '1.0')  
      .set('Accept', 'application/json')  
      .expect(codes.nocontent)  
      .end(function (err, res) {  
        should.not.exist(err);  
        res.body.should.be.empty();  
        done();  
      })  
  });  
});
```

REST Dokumentieren

- **Manuell (Word, LaTeX, ...)**
 - Für kleine APIs passend
- **Per Framework integriert generieren**
 - Bspw. Restler3 für PHP (s.u.)
- **Swagger.io (s.u.)**
 - Für mittlere APIs und
 - Unterstützte Code-Frameworks (node.js, PHP, .. JAX-RS)



REST Generieren (und Dokumentieren)

- Mit RESTLER 3



- PHP Rest Framework

- Einfache OAuth2 Integration
- Annotationen in PHPDoc
 - Automatische API Dokumentation

REST Generieren (und Dokumentieren)

- **RESTLER nutzt die Annotationen**
(Hier am Beispiel der Klasse Authors, Methode GET)






```
/**
 * @param int $id
 *
 * @return array
 */
function get($id)
{
    $r = $this->dp->get($id);
    if ($r === false)
        throw new RestException(404);
    return $r;
}
```


REST Generieren (und Dokumentieren)

- **RESTLER** nutzt die Annotationen..und generiert eine Doku

/authors

Show/Hide | List Operations | Expand Operations | Raw

| | | |
|--------|--------------------|--|
| GET | /authors.json | routes to <code>improved\Authors::index();</code>  |
| GET | /authors.json/{id} | routes to <code>improved\Authors::get();</code>  |
| POST | /authors.json | routes to <code>improved\Authors::post();</code>  |
| PUT | /authors.json/{id} | routes to <code>improved\Authors::put();</code>  |
| PATCH | /authors.json/{id} | routes to <code>improved\Authors::patch();</code>  |
| DELETE | /authors.json/{id} | routes to <code>improved\Authors::delete();</code>  |

REST Generieren (und Dokumentieren)

- **Swagger.io Editor**
- **Design->CodeGen**
- **API-Code->Doku**

The screenshot shows the Swagger.io Editor interface. On the left, the Swagger JSON definition is displayed in a code editor. On the right, the rendered API documentation is shown, including the title 'Uber API', version '1.0.0', and a list of paths. The 'Products' tag is selected, and the '/products' endpoint is highlighted. The documentation for this endpoint includes a summary, description, and a table of parameters.

```

1  swagger: '2.0'
2  info:
3    title: Uber API
4    description: Move your app forward with the Uber API
5    version: 1.0.0
6    host: api.uber.com
7  schemes:
8    - https
9  basePath: /v1
10 produces:
11   - application/json
12 paths:
13   /products:
14   get:
15     summary: Product Types
16     description: |
17       The Products endpoint returns information about the
18       *Uber* products
19       offered at a given location. The response includes
20       the display name
21       and other details about each product, and lists the
22       products in the
23       proper display order.
24   parameters:
25     - name: latitude
26       in: query
27       description: Latitude component of location.
28       required: true
29       type: number
30       format: double
31     - name: longitude
32       in: query
33       description: Longitude component of location.
34       required: true
35       type: number
36       format: double
37   tags:
38     - Products
39   responses:
40     '200':
41       description: An array of products
42       schema:
43         type: array
44         items:
45           $ref: '#/definitions/Product'
46     default:
47       description: Unexpected error
48       schema:
49         $ref: '#/definitions/Error'
50   /estimates/price:
51   get:
52     summary: Price Estimates
53     description: >
54       The Price Estimates endpoint returns information about the
55       *Uber* products offered at a given location. The response
56       includes the display name and other details about each
57       product, and lists the products in the proper display
58       order.
59   
```

Uber API
Move your app forward with the Uber API
Version 1.0.0
Filter operations by a tag:
Products Estimates User

Paths
/products

GET /products **Products**

Summary
Product Types

Description
The Products endpoint returns information about the *Uber* products offered at a given location. The response includes the display name and other details about each product, and lists the products in the proper display order.

Parameters

| Name | Located in | Description | Required | Schema |
|-----------|------------|----------------------------------|----------|-------------------|
| latitude | query | Latitude component of location. | Yes | ⇒ number (double) |
| longitude | query | Longitude component of location. | Yes | ⇒ number (double) |

Responses

Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

Zusammenfassende Fragen

- **Aus welchen drei wesentlichen Bausteinen besteht eine REST-Schnittstelle?**



Zusammenfassende Fragen

- **Welcher Zusammenhang besteht zwischen HTTP Methoden und RESTful APIs?**

| CRUD Operation | HTTP Methode |
|----------------|--------------|
| Create | POST |
| Read | GET |
| Update | PUT |
| Delete | DELETE |

- **Beispiel:**

/authors

Show/Hide | List Operations | Expand Operations | Raw

| | | |
|--------|--------------------|---|
| GET | /authors.json | routes to improved\Authors::index(); 🔒 |
| GET | /authors.json/{id} | routes to improved\Authors::get(); 🔒 |
| POST | /authors.json | routes to improved\Authors::post(); 🔒 |
| PUT | /authors.json/{id} | routes to improved\Authors::put(); 🔒 |
| PATCH | /authors.json/{id} | routes to improved\Authors::patch(); 🔒 |
| DELETE | /authors.json/{id} | routes to improved\Authors::delete(); 🔒 |

Zusammenfassende Fragen

- **Was ist HATEOAS und was hat es zu tun mit REST?**
- **Hypermedia as the engine of application state**
- **Sendet in Antwort**
 - neben den Ressourcen
 - **AUCH Verweise** (auf weitere Ressourcen und Operationen)
- Komplettes HATEOAS ist Overkill (das wäre Level 3)
- aber in Teilen nützlich (siehe REST Design Richtlinien 5 u. 6)
(das ist dann ggf. Level 2)

Zusammenfassende Fragen

■ Welche Design Guidelines für RESTful Design kennen Sie?

1. Rückgabetyt
2. API Versionierung
3. HTTP Status und Fehler
4. Sicherheit
5. Filtern und Blättern
6. Verweise und Expansion

Ein möglicher Aufruf

GET `https://localhost/api/v1/tweets.json/b3f4d5/retweets?`

3

4

2

1

6

`contains=BeuthHS&offset=50&limit=25`

5

Agenda

- **Wiederholung**
- **APIs**
- **Web APIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

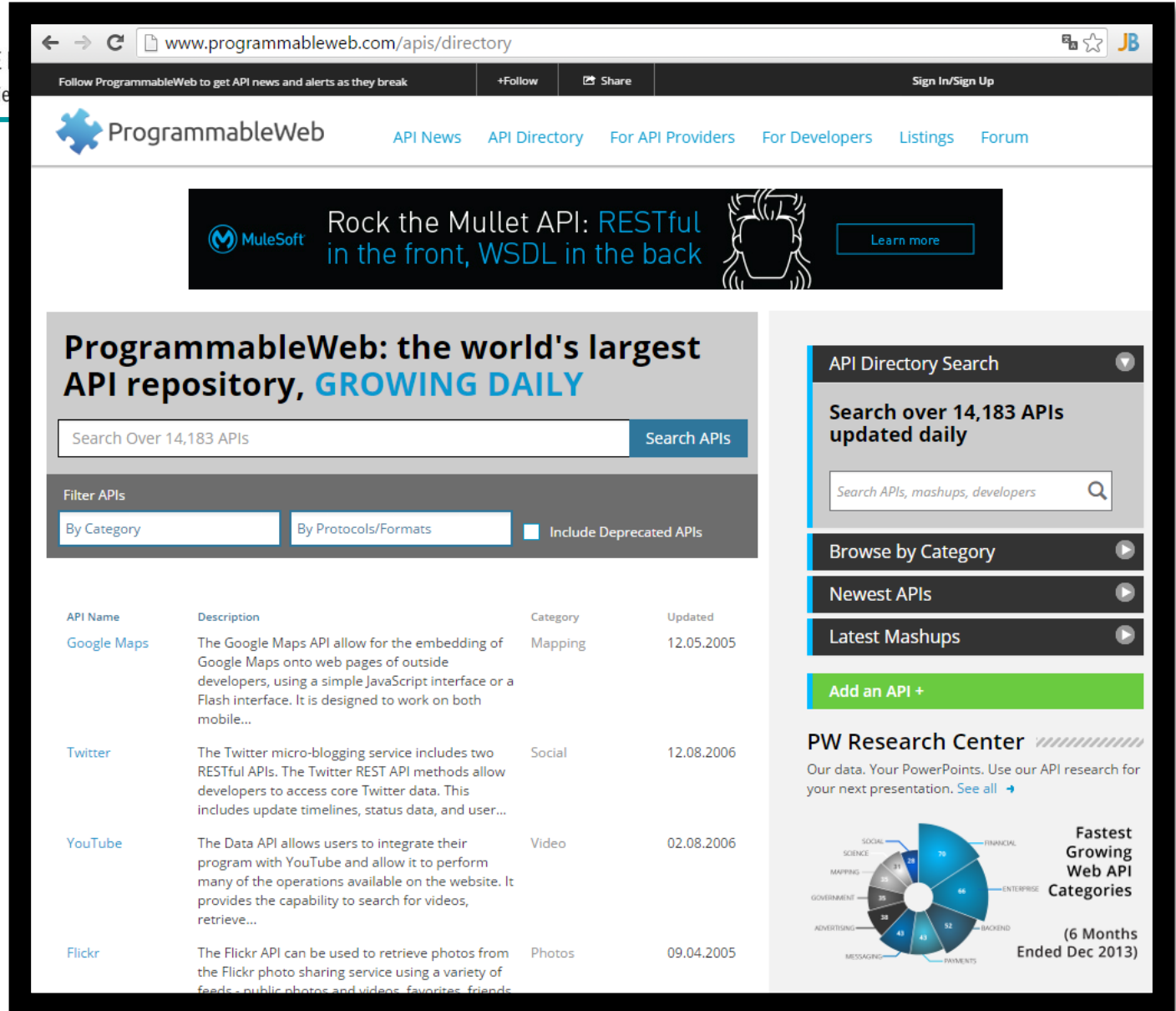
Ausblick auf nächstes Mal

- REST mit Node
 - Content-Types lesen/setzen
 - JSON auslesen/liefern
 - GET, POST, DELETE bedienen
- Einige npm Pakete dafür

```
// API-Version control. We use HTTP Header field Accept-Version i
app.use(function(req, res, next){
  // expect the Accept-Version header to be NOT set or being 1
  var versionWanted = req.get('Accept-Version');
  if (versionWanted !== undefined && versionWanted !== '1.0')
    // 406 Accept-* header cannot be fulfilled
    res.status(406).send('Accept-Version cannot be fulfill
  } else {
    next(); // all OK, call next handler
  }
});
```

Vielen Dank
und bis
zum nächsten Mal

Lust auf REST APIs ?



The screenshot shows the ProgrammableWeb website, which is described as "the world's largest API repository, GROWING DAILY". The page features a search bar with the text "Search Over 14,183 APIs" and a "Search APIs" button. Below the search bar, there are filters for "Filter APIs" with options "By Category" and "By Protocols/Formats", and a checkbox for "Include Deprecated APIs".

| API Name | Description | Category | Updated |
|-------------|---|----------|------------|
| Google Maps | The Google Maps API allow for the embedding of Google Maps onto web pages of outside developers, using a simple JavaScript interface or a Flash interface. It is designed to work on both mobile... | Mapping | 12.05.2005 |
| Twitter | The Twitter micro-blogging service includes two RESTful APIs. The Twitter REST API methods allow developers to access core Twitter data. This includes update timelines, status data, and user... | Social | 12.08.2006 |
| YouTube | The Data API allows users to integrate their program with YouTube and allow it to perform many of the operations available on the website. It provides the capability to search for videos, retrieve... | Video | 02.08.2006 |
| Flickr | The Flickr API can be used to retrieve photos from the Flickr photo sharing service using a variety of feeds: public photos and videos, favorites, friends... | Photos | 09.04.2005 |

On the right side of the page, there is a section titled "API Directory Search" with a search bar and a "Search APIs, mashups, developers" button. Below this, there are links for "Browse by Category", "Newest APIs", and "Latest Mashups". A green button labeled "Add an API +" is also present.

At the bottom right, there is a section titled "PW Research Center" with the text "Our data. Your PowerPoints. Use our API research for your next presentation. See all →". Below this, there is a pie chart showing the distribution of APIs by category, with the text "Fastest Growing Web API Categories (6 Months Ended Dec 2013)".

The pie chart data is as follows:

| Category | Count |
|-------------|-------|
| FINANCIAL | 70 |
| ENTERPRISE | 66 |
| BACKEND | 52 |
| PAYMENTS | 43 |
| MESSAGING | 43 |
| ADVERTISING | 38 |
| GOVERNMENT | 35 |
| MAPPING | 25 |
| SCIENCE | 21 |
| SOCIAL | 18 |