

Web Engineering II

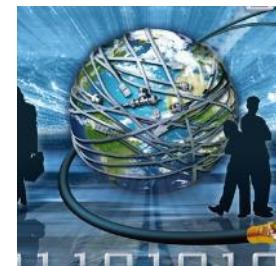
04 APIs und RESTful Design

Johannes Konert



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences





Agenda

- **Wiederholung**
- **APIs und WebAPIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**

- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**



Agenda

- **Wiederholung**

- **APIs und WebAPIs**

- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**

- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**



Fragen zur Wiederholung

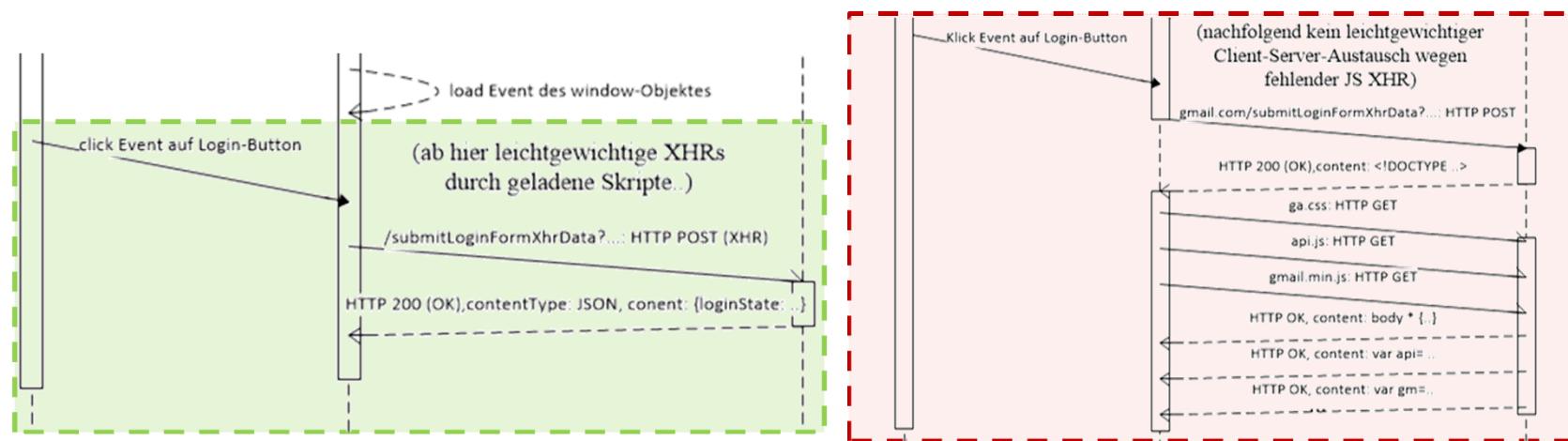
„Pick-and-Challenge“



1. Wieso ist nicht jede **Rich-Client Seite** auch eine **Single Page Application**?
2. Was sind die wesentlichen Unterschiede einer Webseite nach dem Thin-Client-Prinzip vs. Rich-Client-Prinzip?
3. Welchen Web Stack würden Sie für {die Beuth-Website, Gmail} nehmen? Warum?
4. Welche Kriterien wurden diskutiert für die Auswahl eines passenden Stacks? Wurden Kriterien Ihrer Meinung nach vergessen?
5. Warum ist es für eine SPA nachteilig einen Web Stack zu verwenden, bei dem Webserver und Skriptinterpreter getrennte Komponenten sind?
6. Wofür stehen die Kürzel **AJAX** und **XHR** und was haben diese miteinander zu tun?
7. Welche zwei Kenngrößen stehen für das **Nadelöhrproblem**?
8. Welche Vorteile bringt node.js mit sich? Nachteile?
9. Welchen Paketmanager können Sie **alternativ zu npm** verwenden? Welche Vorteile hat er?
10. Welche Angaben kommen in die **package.json** Datei? Wie wird die Datei aktualisiert?
11. Wie können Sie mit node.js/express **statische Dateien** aus einem Verzeichnis an den Client ausliefern?
12. Was ist isomorphes JavaScript?
13. **Bonusfrage:** Welches HTML-Gerüst steht bei einer radikal konzipierten Single Page Application in der index.html?

Rückblick / Wiederholung

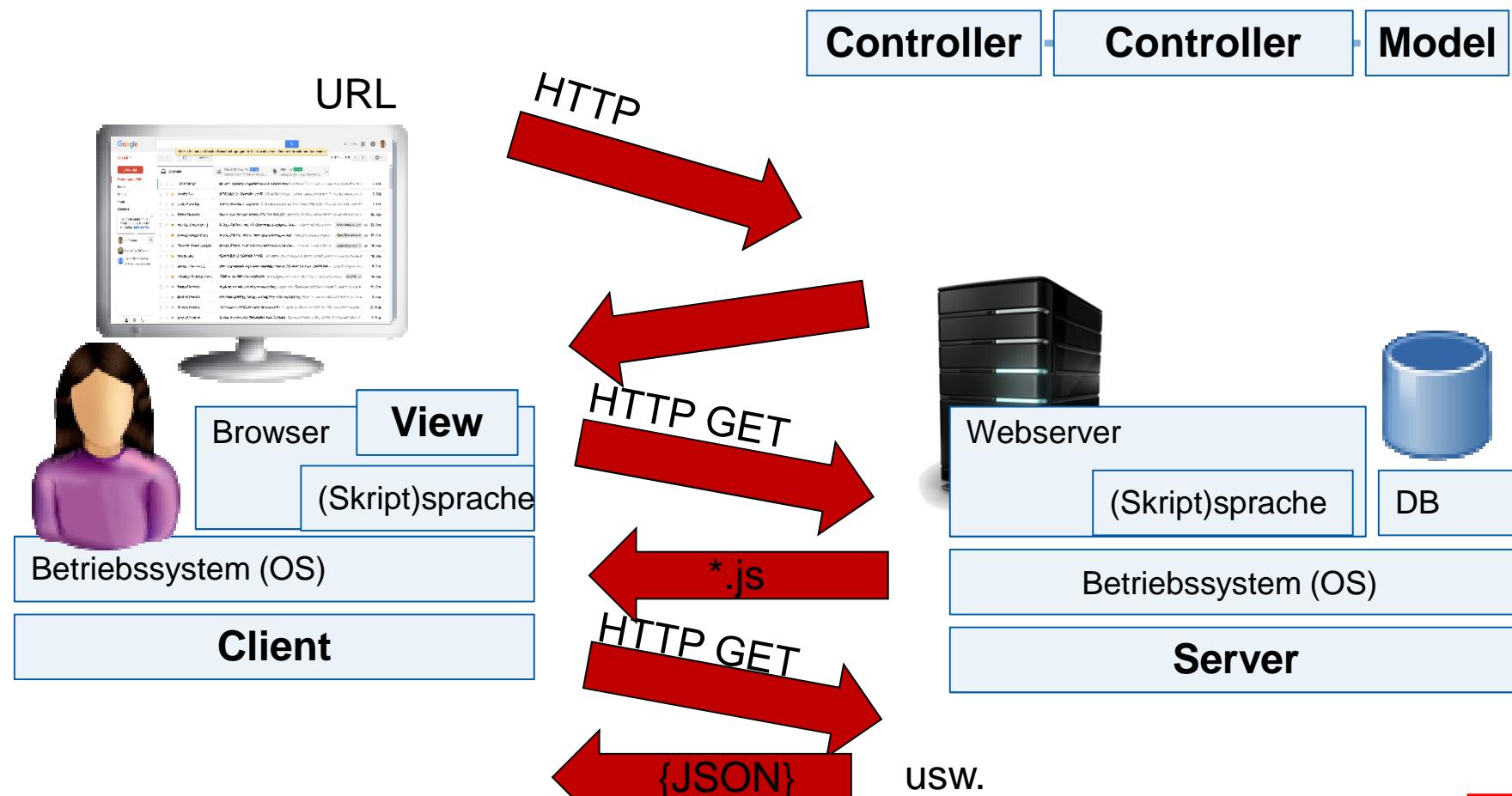
- Was sind die wesentlichen Unterschiede beim Laden der Webseite zwischen Thin-Client und Rich-Client Prinzip?
 - Klausur: ggf. „Beschriften Sie ein Sequenzdiagramm dazu“



Letzte Woche: Client-Server Architektur

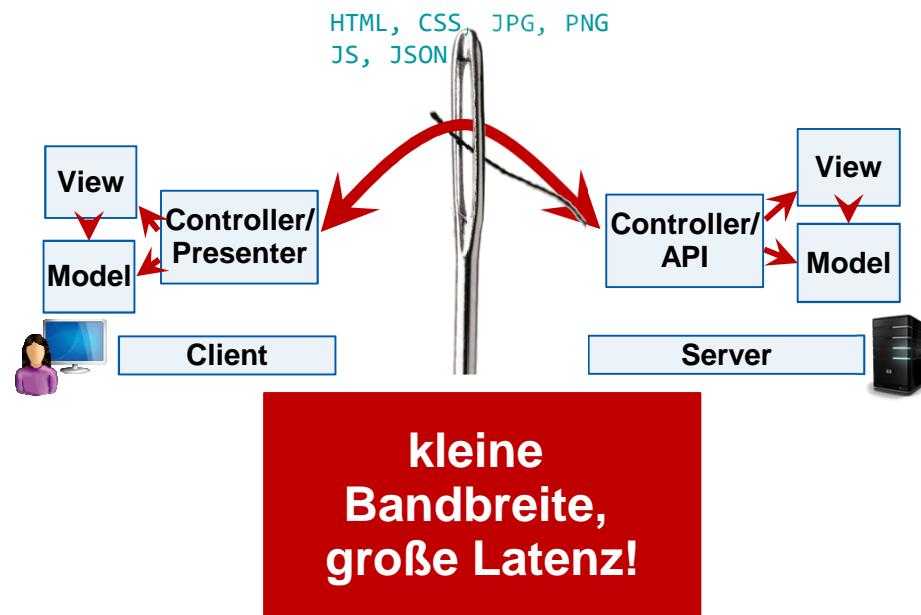
- **Rich-Client-Prinzip** (Beispiel: Szenario der SinglePageApp wie Gmail)

```
<!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
	<p>Performance</p> <ul style="list-style-type: none">▪ statische Dateien▪ Streaming▪ (Skript)- sprachen		<ul style="list-style-type: none">▪ Entwicklungs- geschwindigkeit▪ Modulari- sierung▪ Knowhow eigener Entwickler/in	<ul style="list-style-type: none">▪ Komplexität▪ Modulari- sierung▪ Verbindung mit Skript- sprache	<ul style="list-style-type: none">▪ Größe (kb)▪ Binding▪ Modulari- sierung▪ Kompatibili- tät

■ Generelle Kriterien für alle SW-Produkte

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

Entscheidung hängt vom Anwendungsszenario ab!



Rückblick / Wiederholung

(5.) Warum ist es für eine SPA nachteilig einen Web Stack zu verwenden, bei dem Webserver und Skriptinterpreter getrennte Komponenten sind?

- Laufzeitumgebung des Webservers != Laufzeitumgebung der Skriptsprache → mehr Speicherverbraucht und langsamere Reaktion
- Auch das initial ausgelieferte Dokument ggf. schon via Skriptsprache dynamisch erstellt wird

- Nach dem initialen Laden des Dokumentes (DOM)
 - überwiegend dynamische Inhalte via XHR geladen werden
 - JSON via API (idealerweise REST-API)
 - kleine Templatebausteine



Rückblick/Wiederholung

- Welches HTML-Gerüst steht bei einer radikal konzipierten Single Page Application in der index.html?

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1"
    <meta name="author" content="Johannes Konert">
    <meta name="description" content="Beispielinhalt einer SPA ...">
    <meta name="keywords" content="SPA">
    <title>Minimalistische SPA</title>

    <link rel="stylesheet" href="css/app.css">
    <script src="js/app.js" defer></script>
</head>
<body>
Lade...
</body>
</html>
```



Agenda

- **Wiederholung**

- **APIs und WebAPIs**

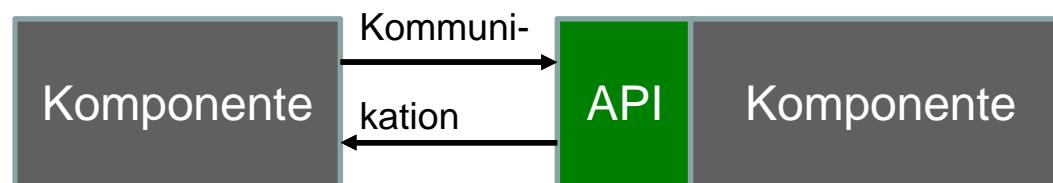
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**

- **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

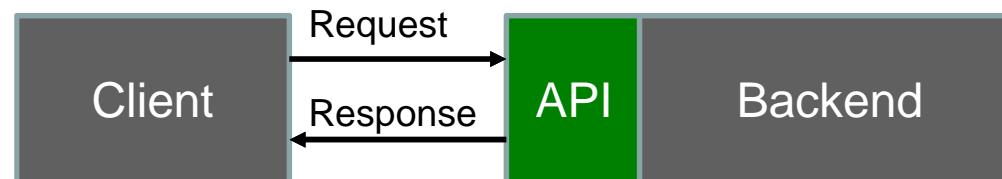
- **Welche der folgenden Elemente haben eine API?**

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

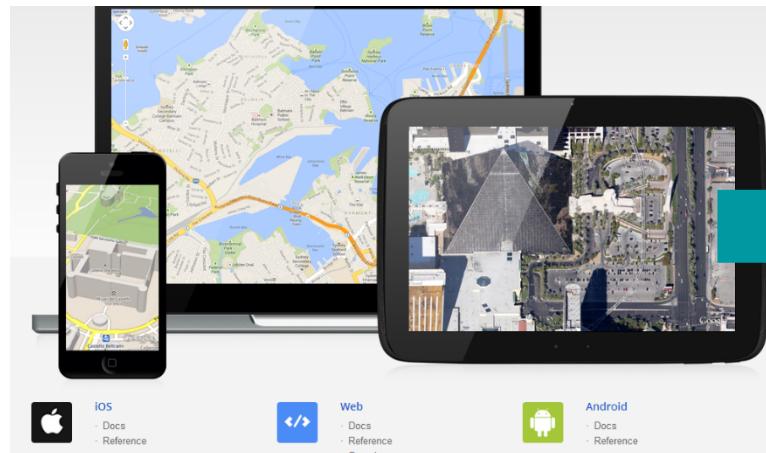
- Besondere Bedeutung beim Datenaustausch
 - Client-Seite (Frontend) und Server-Seite (Backend)
 - Abruf/Speichern von Daten und Zuständen
 - von Anwendungen untereinander
 - Nutzung und Verknüpfung von Datendiensten (Google Maps, Instagram, Facebook, DropBox, ..)



API Beispiele

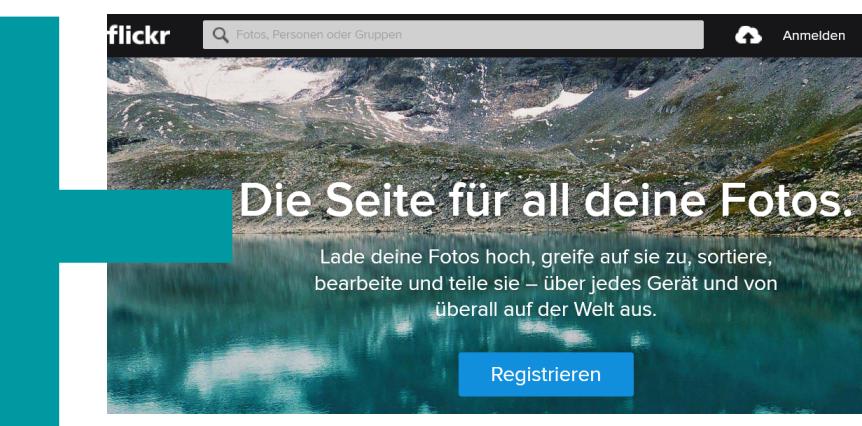
Welche Flickr Bilder wurden in der Nähe geschossen?

- **Google Maps**



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

- **flickr**



<https://www.flickr.com/services/developer/api/>

- **Developer-Seiten enthalten API-Doku und Beispielcode**



API Beispiele

Welche Instagram Bilder wurden in der Nähe geschossen?

- **Google Maps + flickr**

1. **Long/Lat Geokoordinaten abrufen**

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

2. **Flickr API abfragen**

<https://api.flickr.com/services/rest/? &lat=..&lon=..&method=flickr.photos.search> **

3. **Bilder anzeigen**

- **user-id und photo-id aus Ergebnis verwenden und in URL einbauen**

[https://www.flickr.com/photos/\[usr\]/\[photo\]](https://www.flickr.com/photos/[usr]/[photo])



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



REST

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



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

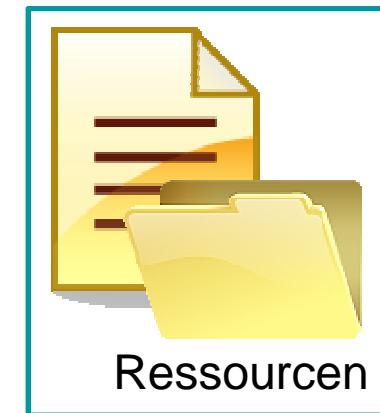
HTTP GET

/api/users/33245/retweets Gib mir alle Retweets des Nutzers 33245

HTTP DELETE

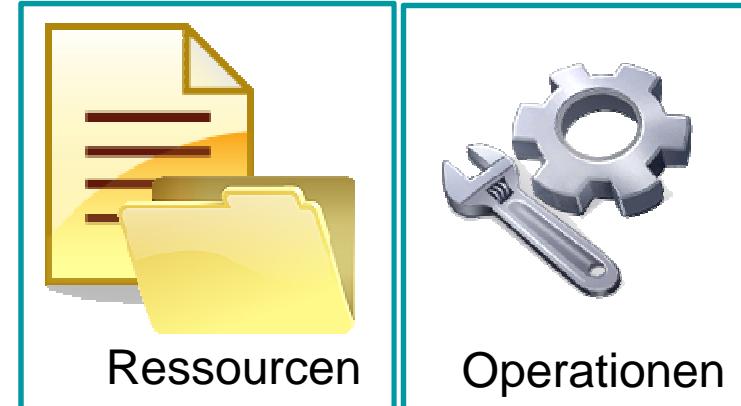
/api/tweets/332456 Lösche den Tweet 332456

- **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!



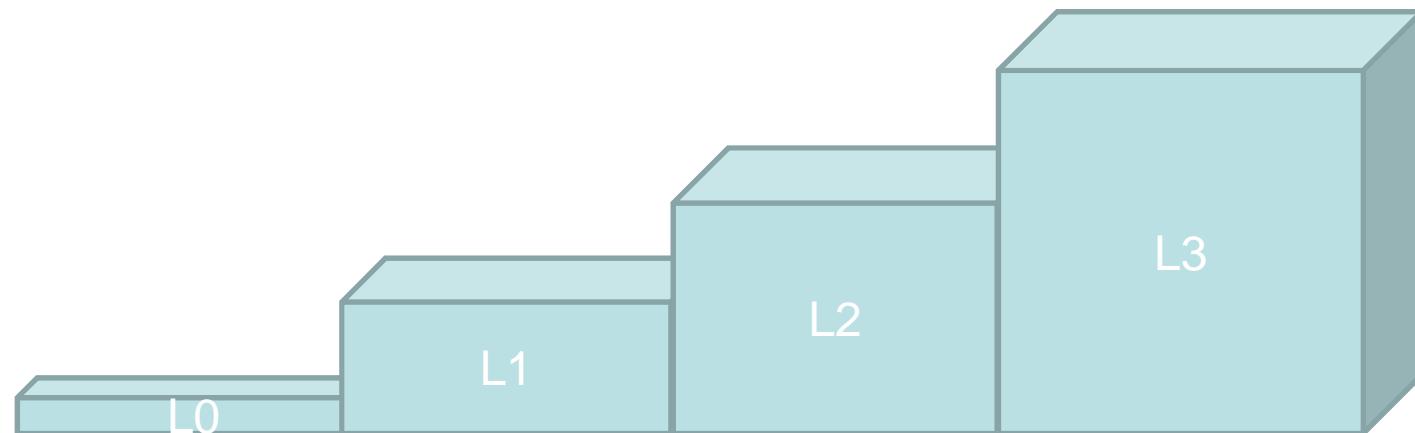
Agenda

- **Wiederholung**
- **APIs und WebAPIs**
- **Von „vor REST“ nach REST**
- **REST Level**
- **REST Richtlinien und Best Practice**
- **REST generieren, testen, dokumentieren**

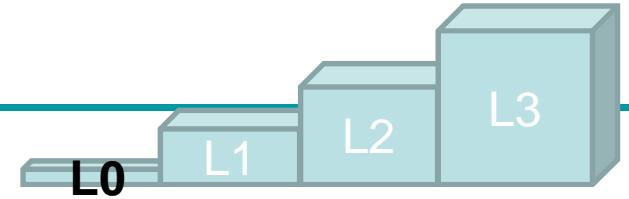
- **Zusammenfassende Fragen**
- **Ausblick: Nächstes Mal**

REST Level

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



REST Level 0



Konzept der „Remote Procedure Call“

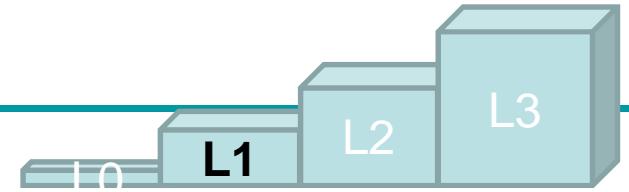
- **Einfacher Aufruf einer Serviceschnittstelle**
- **POST /api/warehouseService**

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

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



REST Level 1



Konzept der „Ressourcen-Zeiger“

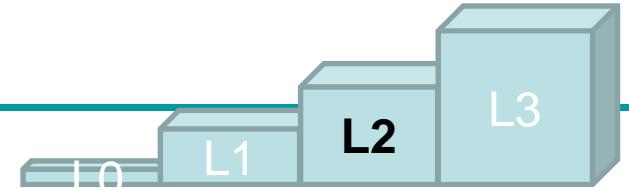
- Aufruf einer eindeutigen URL pro Ressource
- POST /api/customers/1234

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

- cascade=true
- action=delete



REST Level 2



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

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

Im Body keine Anweisungen!

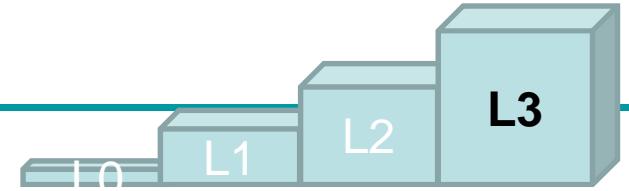
Der Body enthält ausschließlich (bei PUT/POST/PATCH) auszutauschende Objekt-Daten.

- **POST /api/customers/1234/orders**

```
{ "order-date" : "24.04.2017",
  "article-id": 18234,
  "amount" : 1
}
```



REST Level 3

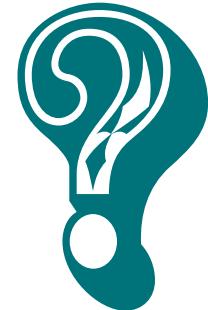


Konzept der „State machine“

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

- `GET /api/customers/1234`

Im Body keine Anweisungen



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

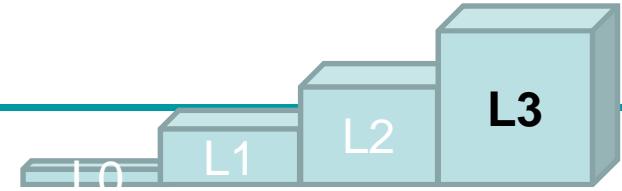
Die HTTP Antwort enthält neben angefragten Objekte(n)

- zusätzlich 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)

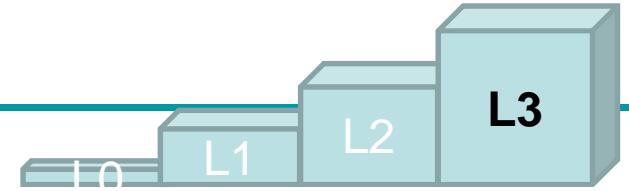
- GET /api/customers/1234

```
{ Antwort:
  "href" : "https://localhost/api/customers/1234",
  "name" : "Susan Sunny",
  ...
  "links" : [ {
    "rel" : "self",
    "href" : "/api/customers/1234"
  }, {
    "rel" : "customers.orders",
    "href" : "/api/customers/1234/orders"
  }
]
```

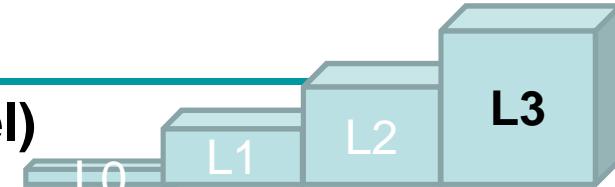
Ressource

Verweise (Ressourcen und Operationen)

State machine



- Ein Zustandsautomat bildet alle Zustände, die eine Maschine annehmen kann ab (siehe Touring Machine, Petrinetze, ..)
- Zusammenhang mit REST:
 - HTTP Antworten repräsentieren (den „Blick“ auf) einen (Client-seitigen) Zustand
 - HTTP Operationen (REST Operationen) die möglichen Zustandsübergänge
- Zusammenhang mit HATEOS/REST Level 3:
 - Die Meta-Daten in den JSON-Daten der HTTP Antworten listen alle möglichen Operationen vom aktuellen Zustand aus auf.



Zustandsübergänge in REST Level 3 (Beispiel)

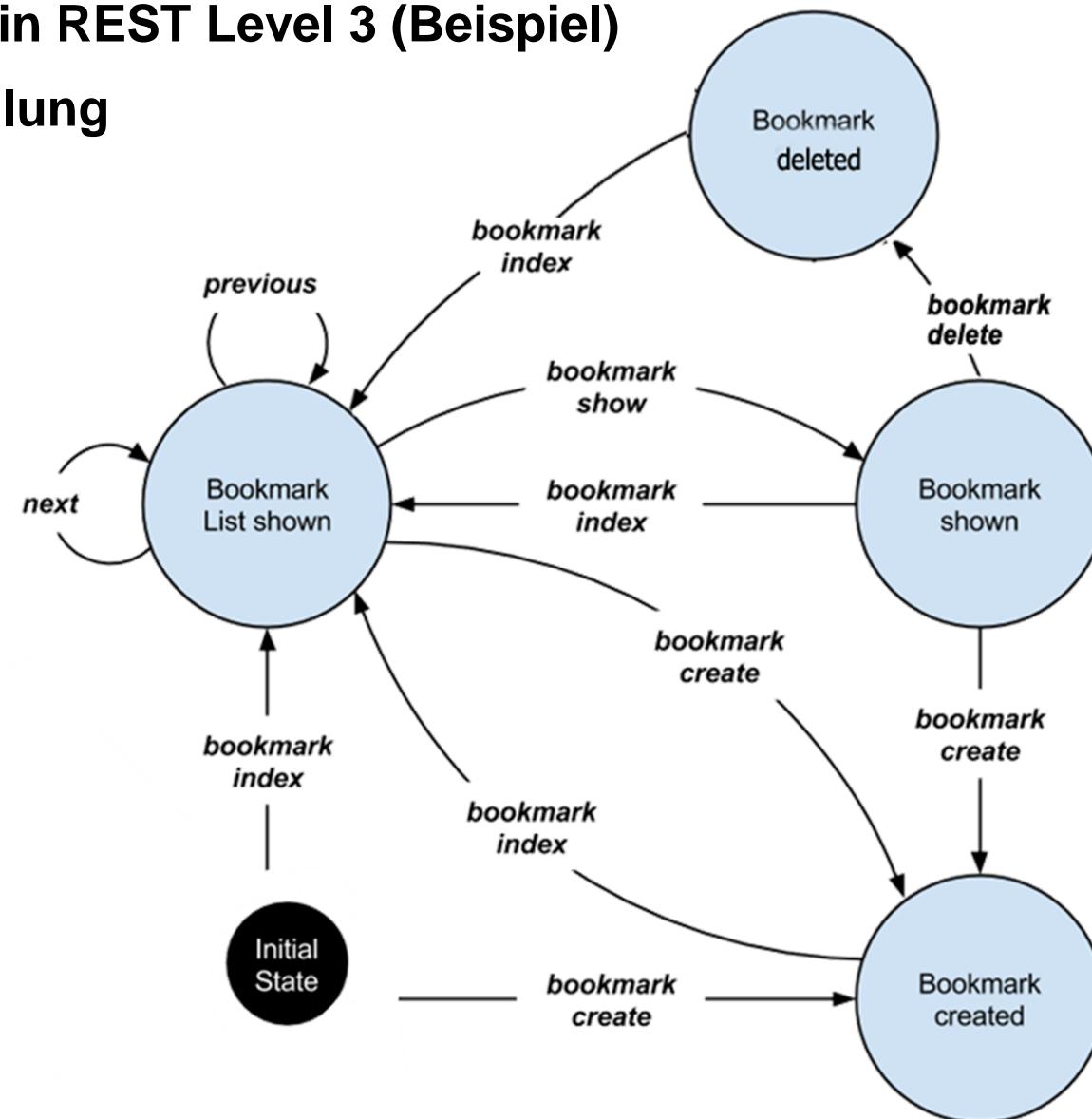
■ Operationen zum Übergang der Zustände

GET	/bookmarks	Liste aller Lesezeichen (der ersten 25)
GET	/bookmarks ?offset=x&limit=y	Liste der Lesezeichen von x bis x+y
GET	/bookmarks/:id	Zeige Bookmark Details eines Eintrags
POST	/bookmarks	Lege einen neuen Eintrag an
DELETE	/bookmarks/:id	Lösche einen Eintrag

- **Initialer Zustand unserer Maschine sei der leere Zustand vor der allerersten Anfrage an die REST-Schnittstelle. Dieser erlaubt per Definition nur zwei Operationen**
 - **Liste aller Lesezeichen**
 - **Erstellen eines Lesezeichens**

Zustandsübergänge in REST Level 3 (Beispiel)

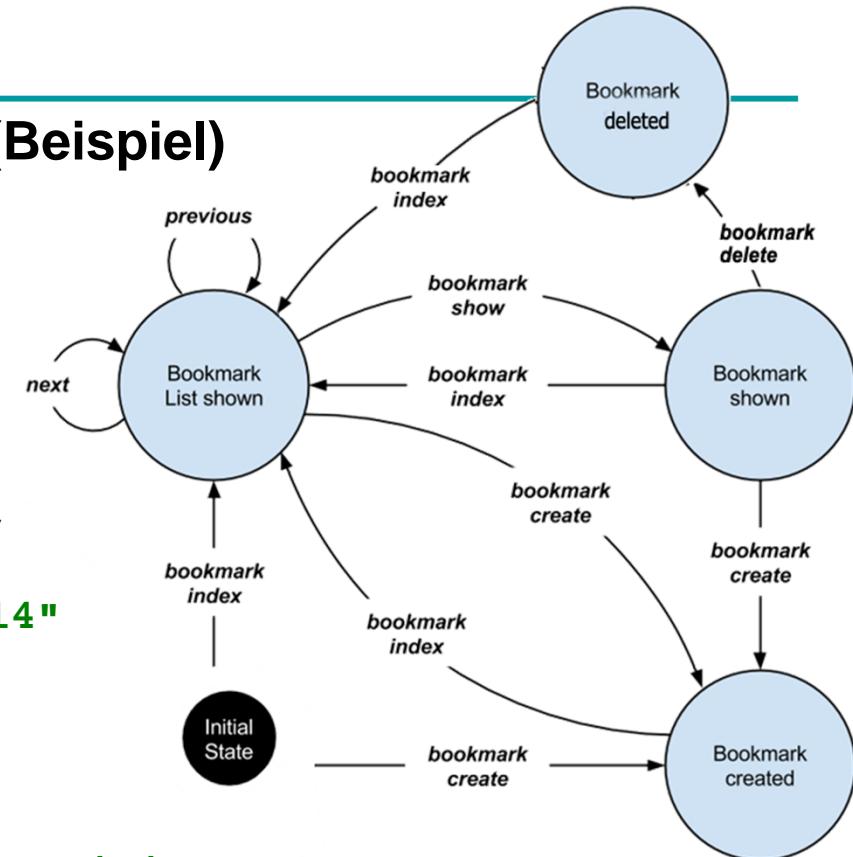
■ Grafische Darstellung



Zustandsübergänge in REST Level 3 (Beispiel)

- GET /bookmarks
- Antwort (z.B.):

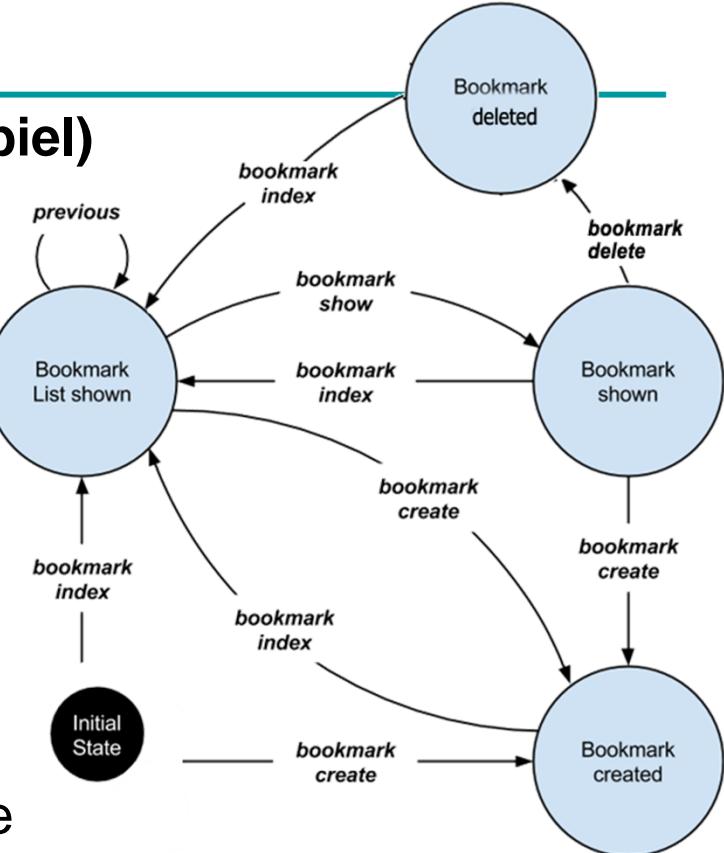
```
{  
    "items": [ {  
        "title": "Beuth HS",  
        "url": "http://www.beuth...",  
        "link": {  
            "show": "/bookmarks/121314"  
        }  
    } //... 25 items  
],  
    "links": {  
        "next": "/bookmarks/?offset=25&limit=25",  
        "create": "/bookmarks"  
    }  
};
```



- Konvention/Regeln dieser HATEOAS-Daten:
 - Unter „links“ finden sich mögliche nächste Operationen.
 - Operationen sind immer als HTTP GET auszuführen, außer bei den Schlüsselwörtern create (POST) und remove (DELETE)

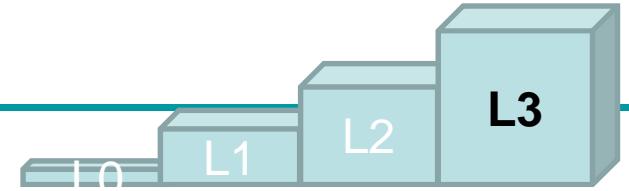
Zustandsübergänge in REST Level 3 (Beispiel)

- **HATEOAS**
- ..liefert also alle möglichen Operationen (Zustandsübergänge) als Meta-Daten mit
- Die HTTP Antworten sind die Zustände (für den Client)
- Die HTTP Operationen auf bestimmten Ressourcen-URLs die Zustandsübergänge





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 derzeit! (+ Dokumentation)
- REST Level 3 kommt aber, siehe auch
<https://www.slideshare.net/lanthaler/building-next-generation-web-apps-with-jsonld-and-hydra>
 - Verweise auf weitere Operationen und Ressourcen dann mittels JSON-LD (semantisch)



Agenda

- **Wiederholung**
- **APIs und WebAPIs**
- **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. Rückgabetyp**
- 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

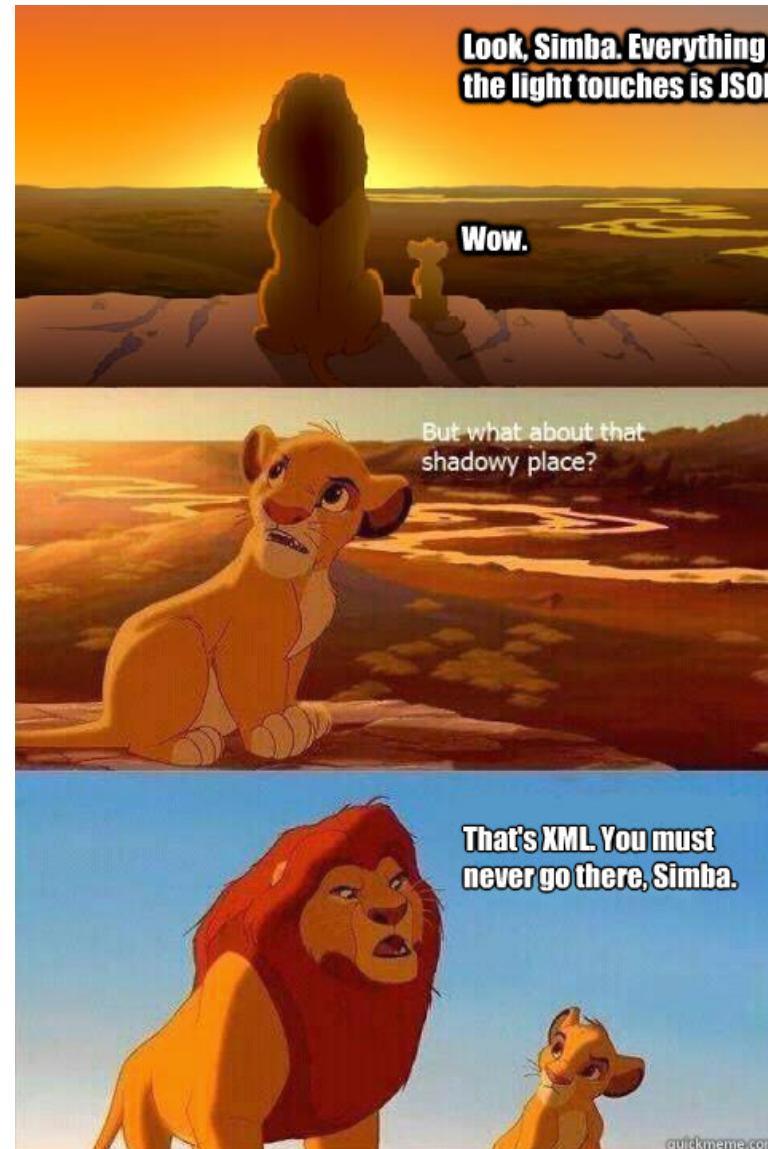


- Möglichkeiten
 - a. In der URL: `/api/tweets.json`
 - b. HTTP Header Feld: „Accept: application/json, text/csv, text/plain, text/html“ (sortiert nach Präferenz)
- Antwort vom Server, z.B.
 - HTTP Header Feld: „Content-Type: application/json“
 - oder Fehler: **HTTP Status 406 Not Acceptable** (Ressource 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 wechselten 2016 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 „Authentifizieren ohne Logins“ ?
- Sessions: Zustand auf dem Server! Keine gute Idee
- **Lösung: Signed self contained tokens.** Diese müss(t)en:
 - alle Zugriffsrechte und Angaben des Nutzers enthalten
 - signiert sein vom Server gegen Manipulation (und zur Prüfung)
- Übermittlung
 - a. In der URL: /api/v1/tweets?accesstoken=b2k3h5u65b3o5b3tg
 - b. Im HTTP Header (am Beispiel OAuth 1.0a):
`Authorization: OAuth realm="http://localhost",
oauth_consumer_key="0685bd9184jfhq22",
oauth_token="ad180jjd733klru7"`
- 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“  
        }], ...  
    }  
...  
}
```

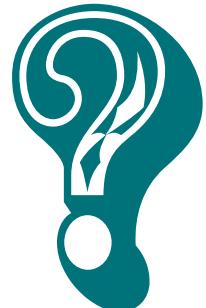


Kleiner Test für REST

■ 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

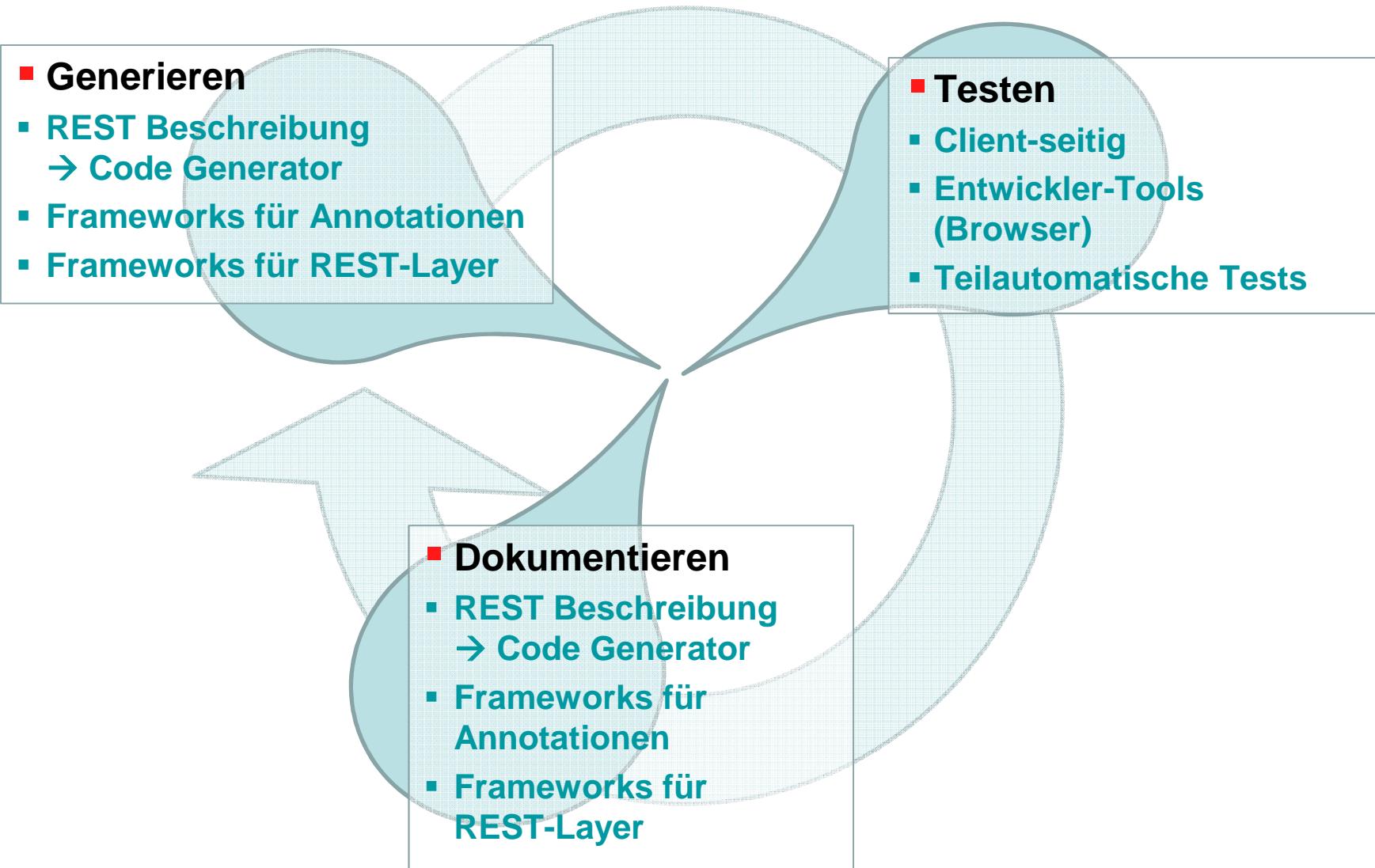
** wird in der Praxis keine Berechtigung haben, formal ist das aber gültig.
~ wird nur der Nutzende i4h5k9 selbst dürfen)



Agenda

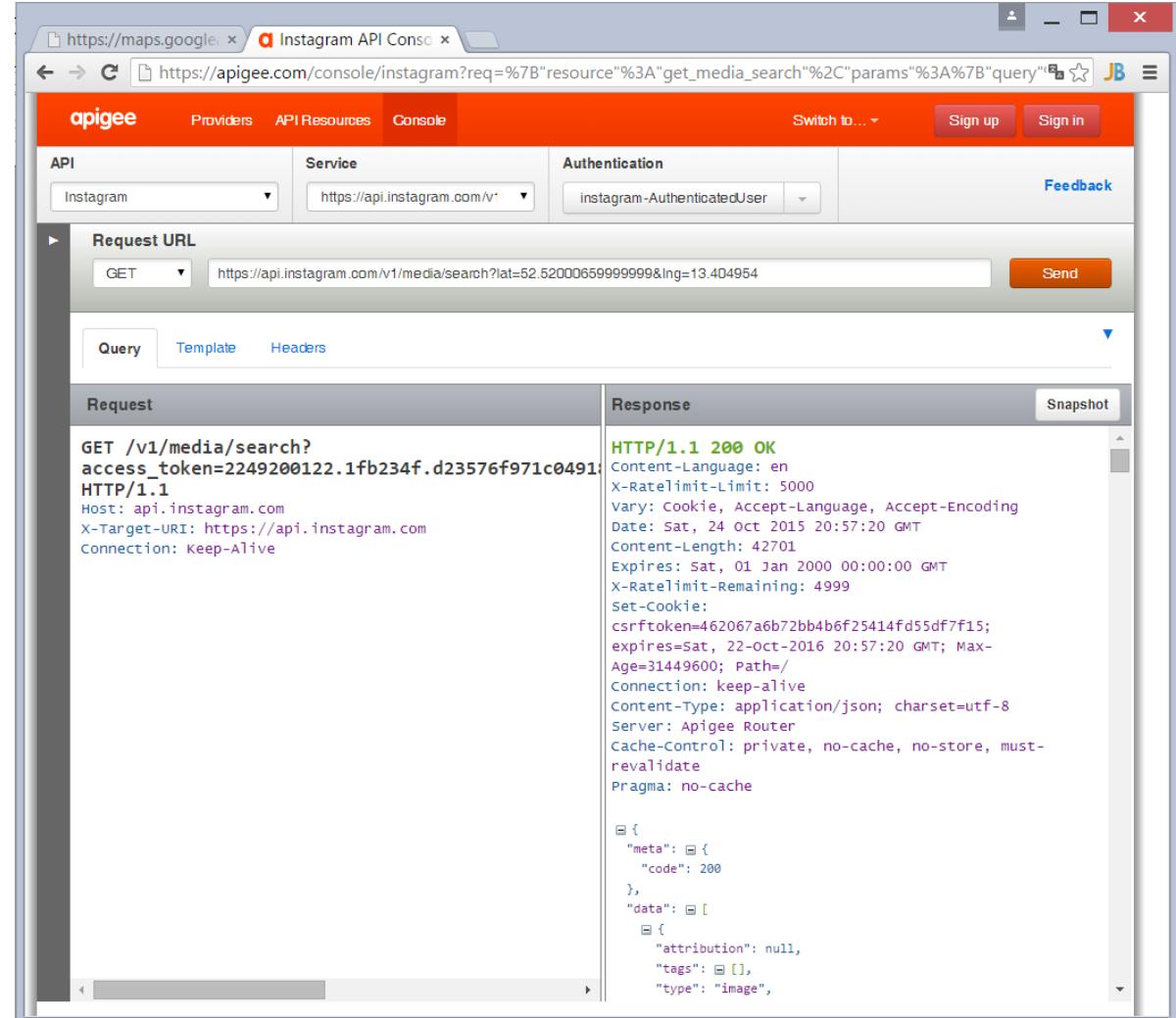
- **Wiederholung**
 - **APIs und WebAPIs**
 - **Von „vor REST“ nach REST**
 - **REST Level**
 - **REST Richtlinien und Best Practice**
 - **REST generieren, testen, dokumentieren**
-
- **Zusammenfassende Fragen**
 - **Ausblick: Nächstes Mal**

REST Schnittstellen Generieren, Testen, Dokumentieren



REST Testen

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



The screenshot shows the Apigee API Console interface. The top navigation bar includes tabs for 'apigee', 'Providers', 'API Resources', and 'Console'. The 'Console' tab is active. The 'API' dropdown is set to 'Instagram', and the 'Service' dropdown is set to 'https://api.instagram.com/v1'. The 'Authentication' dropdown is set to 'instagram-AuthenticatedUser'. The 'Request URL' section shows a GET request to 'https://api.instagram.com/v1/media/search?lat=52.52000659999999&lng=13.404954'. Below this, there are tabs for 'Query', 'Template', and 'Headers'. The 'Request' pane displays the raw HTTP request:

```
GET /v1/media/search?
access_token=2249200122.1fb234f.d23576f971c0491
HTTP/1.1
Host: api.instagram.com
X-Target-URI: https://api.instagram.com
Connection: Keep-Alive
```

The 'Response' pane shows the raw HTTP response and its JSON payload:

```
HTTP/1.1 200 OK
Content-Language: en
X-RateLimit-Limit: 5000
Vary: Cookie, Accept-Language, Accept-Encoding
Date: Sat, 24 Oct 2015 20:57:20 GMT
Content-Length: 42701
Expires: Sat, 01 Jan 2000 00:00:00 GMT
X-RateLimit-Remaining: 4999
Set-Cookie:
csrfToken=462067a6b72bb4b6f25414fd55df7f15;
expires=Sat, 22-Oct-2016 20:57:20 GMT; Max-
Age=31449600; Path=/;
Connection: keep-alive
Content-Type: application/json; charset=utf-8
Server: Apigee Router
Cache-Control: private, no-cache, no-store, must-
revalidate
Pragma: no-cache

{
  "meta": {
    "code": 200
  },
  "data": [
    {
      "attribution": null,
      "tags": [],
      "type": "image"
    }
  ]
}
```



REST Testen

■ Postman Google Chrome App



Welcome to Postman 3.0

A great new experience, jam-packed with features

The screenshot shows the Postman Chrome extension window. At the top, there's a navigation bar with tabs for 'Builder' (which is active), 'Runner', 'Import', and 'Sign in'. Below the tabs, the URL 'https://maps.googleapis.com/maps/api/geocode/json?address=Berlin' is entered. To the right of the URL are buttons for 'Send', 'Params', and 'Sync Off'. A message 'No environment' is displayed with a delete icon. On the left side, there's a sidebar with 'History' (selected) and 'Collections'. Under 'History', there's a single entry: 'GET https://maps.googleapis.com/maps/api/geocode/json?address=Berlin'. The main panel shows the response body in JSON format. The JSON structure is as follows:

```
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 ]
```

Below the JSON, the status is shown as '200 OK' with a time of '171 ms'. At the bottom right of the main panel, there's a button labeled 'Scroll to response'.

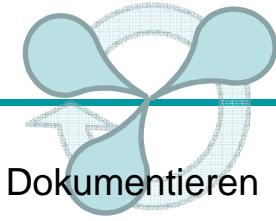


REST Testen

- **Frisby on Jasemine 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();  
      })  
  } );  
});
```

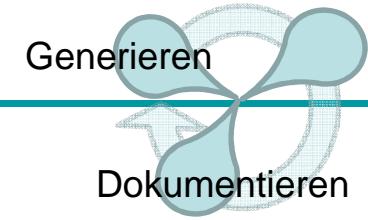


REST Dokumentieren

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

The screenshot shows the official jQuery API documentation page. The top navigation bar includes links for Download, API Documentation (which is currently selected), Blog, Plugins, and Browser Support, along with a search bar. The main content area is titled "jQuery API". On the left, there's a sidebar with a tree view of the API structure, including categories like Ajax, Attributes, Core, CSS, Data, Deferred Object, Deprecated, Dimensions, Effects, and Basics. To the right, several API method entries are listed in boxes:

- .add()** (Traversing > Miscellaneous Traversing): Adds elements to the set of matched elements.
- .addBack()** (Traversing > Miscellaneous Traversing): Add the previous set of elements on the stack to the current set, optionally filtered by a selector.
- .addClass()** (Attributes | Manipulation > Class Attribute | CSS): Adds the specified class(es) to each of the set of matched elements.
- .after()** (Manipulation > DOM Insertion, Outside): Insert content, specified by the parameter, after each element in the set of matched elements.
- .ajaxComplete()** (Ajax > Global Ajax Event Handlers): Register a handler to be called when Ajax requests complete. This is an AjaxEvent.
- .ajaxError()** (Ajax > Global Ajax Event Handlers): Register a handler to be called when Ajax requests complete with an error. This is an AjaxEvent.



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)  
{  
    ...  
}
```



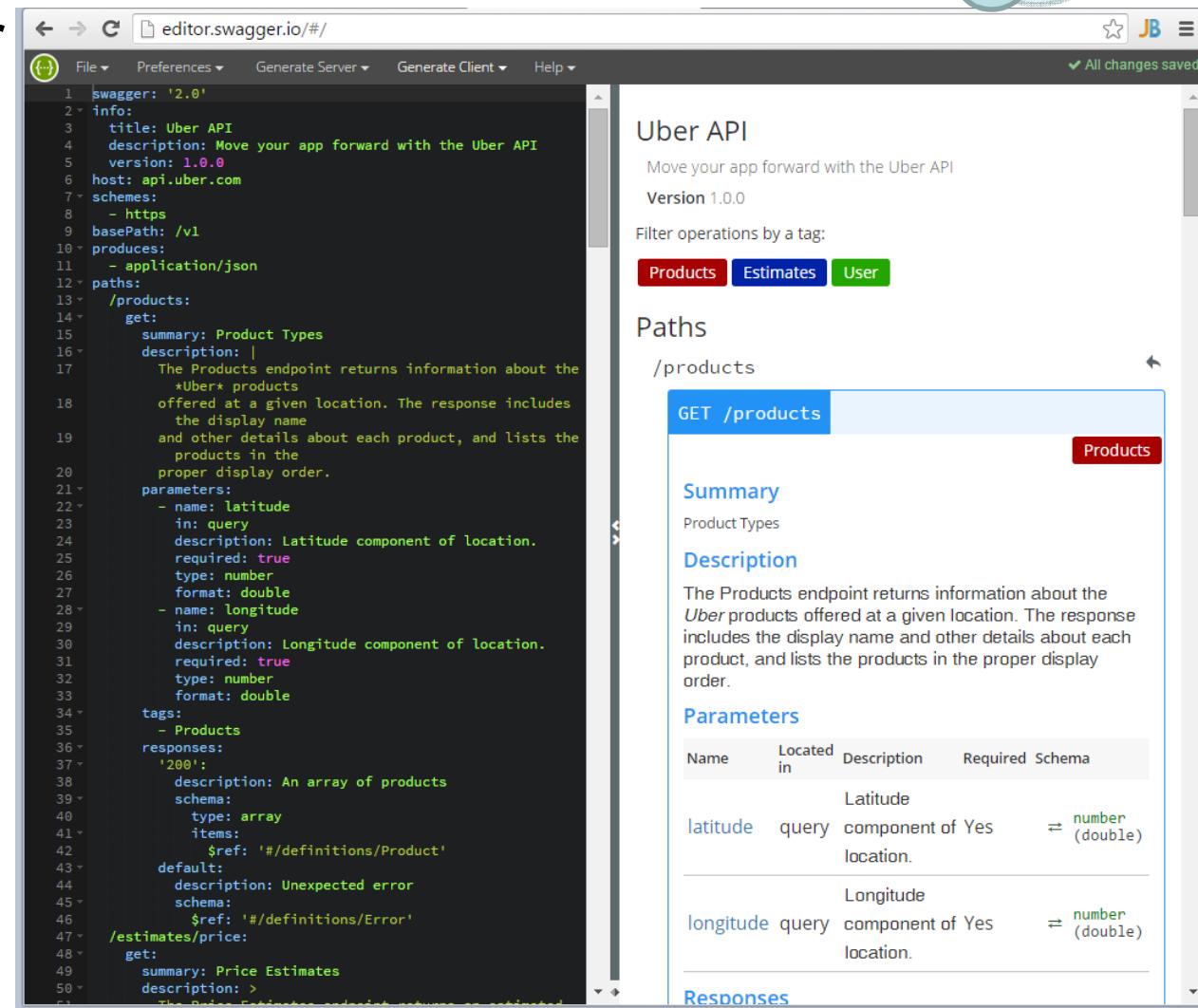
REST Generieren (und Dokumentieren)

- RESTLER nutzt die Annotationen..und generiert eine Doku

/authors		Show/Hide	List Operations	Expand Operations	Raw
GET	/authors.json			routes to 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();	

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 for the Uber API is displayed:

```
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: >
```

The right side shows the generated API documentation for the Uber API. It includes the title "Uber API", description "Move your app forward with the Uber API", and version "1.0.0". It also shows the "Paths" section for the "/products" endpoint, which includes a "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.), and "Parameters" (latitude and longitude). The "Responses" section is also visible.



Agenda

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



Zusammenfassende Fragen

1. Was sind die **vier Merkmale** einer guten API?
2. Welche HTTP Methoden werden für welche **CRUD Operationen** verwendet?
3. Welche HTTP Methode ist niemals **idempotent**, welche kann idempotent sein?
4. Ab welchem **REST-Level** müssen die Ressourcen eine eindeutige URL haben und die Operationen über die HTTP-Methoden durchgeführt werden?
5. Welche **REST-Level** gibt es?
6. Was ist **HATEOS** und was hat es mit REST zu tun?
7. Welches REST-Level erfüllt die flickr-API, bei der wir mittels HTTP GET an `https://api.flickr.com/services/rest/?lat=..&lon=..&method=flickr.photos.search` angefragt haben?
8. Welche vier **HTTP Status-Code Gruppen** gibt es?
9. Wie unterstützen Sie REST-konforme (**zustandslose**) **Authentifizierung**?
10. Wie versionieren Sie eine REST-API?
11. Wozu ist die Unterstützung eines `?expand=` GET-Parameters gut?
12. Wie beschränken Sie die Menge an Einträgen beim Abruf von Ressourcen-Listen?
Welche Teile von HATEOS wären hier ebenfalls gut/sinnvoll?
13. Wie erfährt ein Client nach dem **HTTP POST** die gerade neu erstellte ID des Eintrages?
14. **Bonus-Frage:** Wie sehen die Ressourcen-URLs einer REST-Level 2 API aus für n:m Beziehungen bspw. von Person:Ort ("Abgesehen")?



Agenda

- **Wiederholung**
- **APIs und WebAPIs**
- **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 (via yarn)

Vielen Dank
und bis
zum nächsten Mal

```
// 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
  }
});
```

Lust auf REST APIs ?

www.programmableweb.com/apis/directory

Follow ProgrammableWeb to get API news and alerts as they break +Follow Share Sign In/Sign Up

ProgrammableWeb API News API Directory For API Providers For Developers Listings Forum

MuleSoft Rock the Mullet API: RESTful in the front, WSDL in the back Learn more

ProgrammableWeb: the world's largest API repository, GROWING DAILY

Search Over 14,183 APIs Search APIs

Filter APIs By Category By Protocols/Formats 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

API Directory Search Search over 14,183 APIs updated daily

Search APIs, mashups, developers

Browse by Category Newest APIs Latest Mashups

Add an API +

PW Research Center Our data. Your PowerPoints. Use our API research for your next presentation. See all →

Fastest Growing Web API Categories (6 Months Ended Dec 2013)



Category	Value
SOCIAL	70
FINANCIAL	66
ENTERPRISE	52
BACKEND	43
PAYMENTS	43
MESSAGING	38
ADVERTISING	35
GOVERNMENT	31
MAPPING	28
SCIENCE	25