



Web-Technologien

Hypertext Transfer Protocol (HTTP)

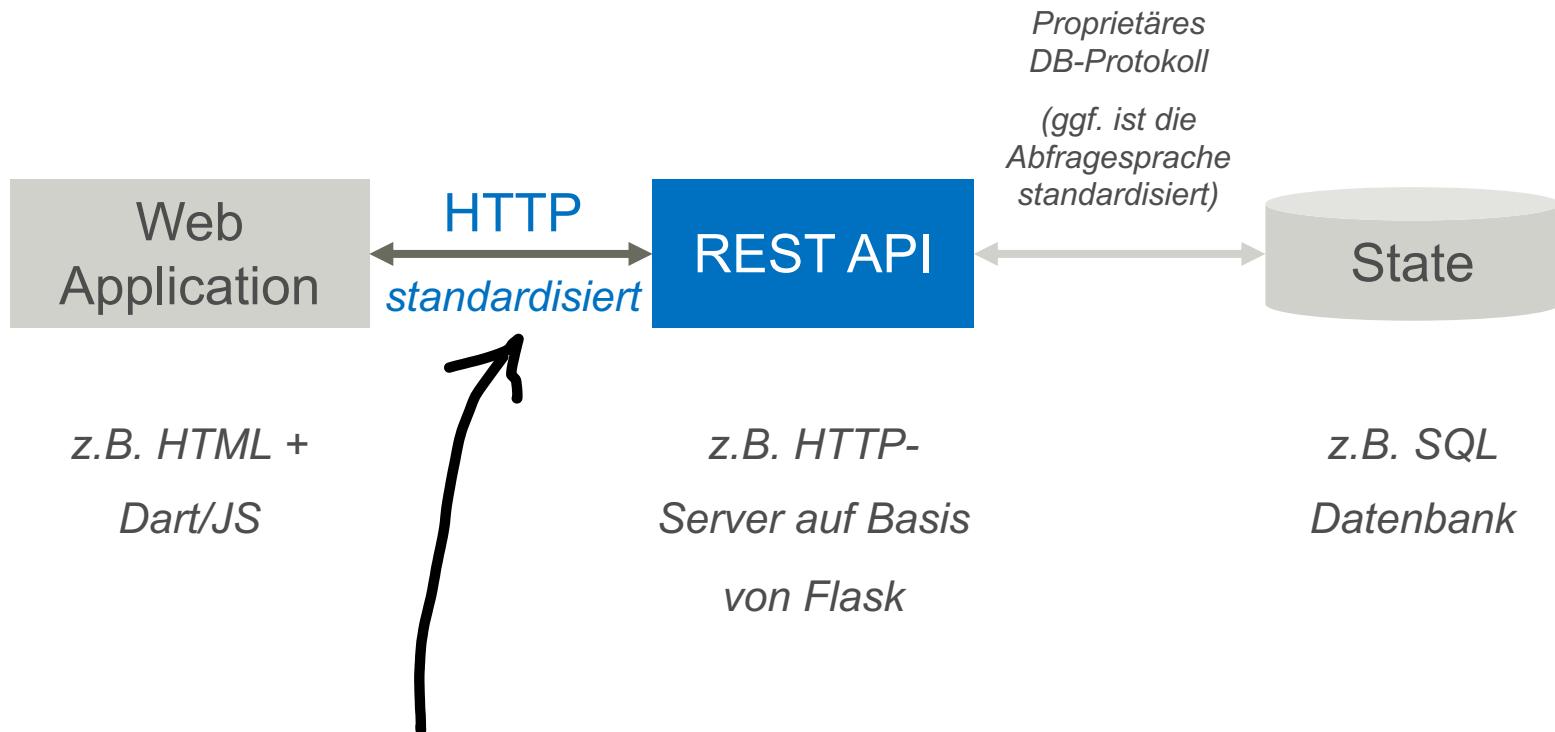
Representational State Transfer (REST)



Prof. Dr. rer. nat. Nane Kratzke

*Praktische Informatik und
betriebliche Informationssysteme*

- **Raum:** 17-0.10
- **Tel.:** 0451 300 5549
- **Email:** nane.kratzke@th-luebeck.de
- **Sprechzeiten:** Mi. 14:00 bis 16:00
(nach Mail-Voranmeldung, oder jederzeit mit Termin)



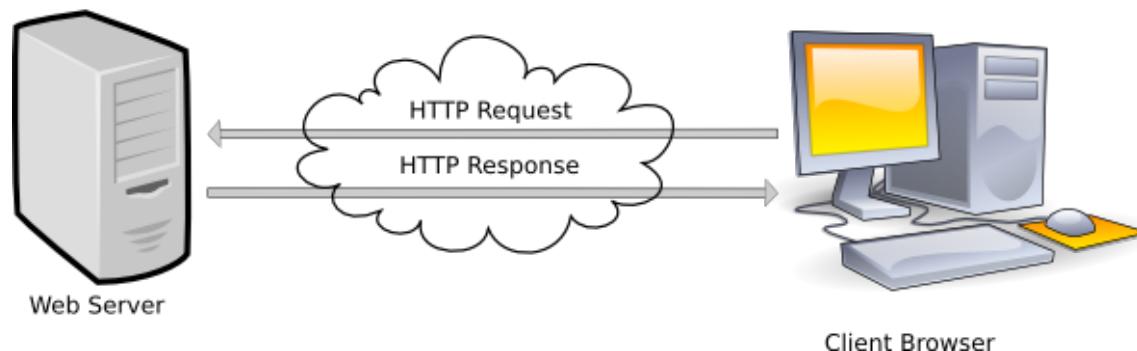
```
import 'dart:html';

const path = 'https://service.dev/resources';
final jsonString = await HttpRequest.getString(path);
```

Inhalte

- **HTTP Protokoll**
 - Hypertext Transfer Protocol (HTTP)
 - URI und URL
 - HTTP Request + Response
 - Content-Types (MIME)
 - HTTP Status Codes
- **REST**
 - Request/Response
 - Representational State Transfer
 - 5 REST Prinzipien
 - HATEOAS (Hypermedia as the Engine of Application State)
- **Ein REST-basierter Greeter Service**
 - Python + Flask
 - HTTP-Routes
 - CORS (Cross-Origin Resource Sharing)

- **Hypertext Transfer Protocol (HTTP)** ist ein zustandsloses Protokoll zur Übertragung von Daten auf der Anwendungsschicht.
- Es wird hauptsächlich eingesetzt, um Webseiten (HTML-Dokumente) mit einem Webbrowser zu laden.
- Es ist jedoch nicht darauf beschränkt und kann auch als allgemeines Datenprotokoll eingesetzt werden (siehe bspw. REST).



https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol

URI – Uniform Resource Identifier

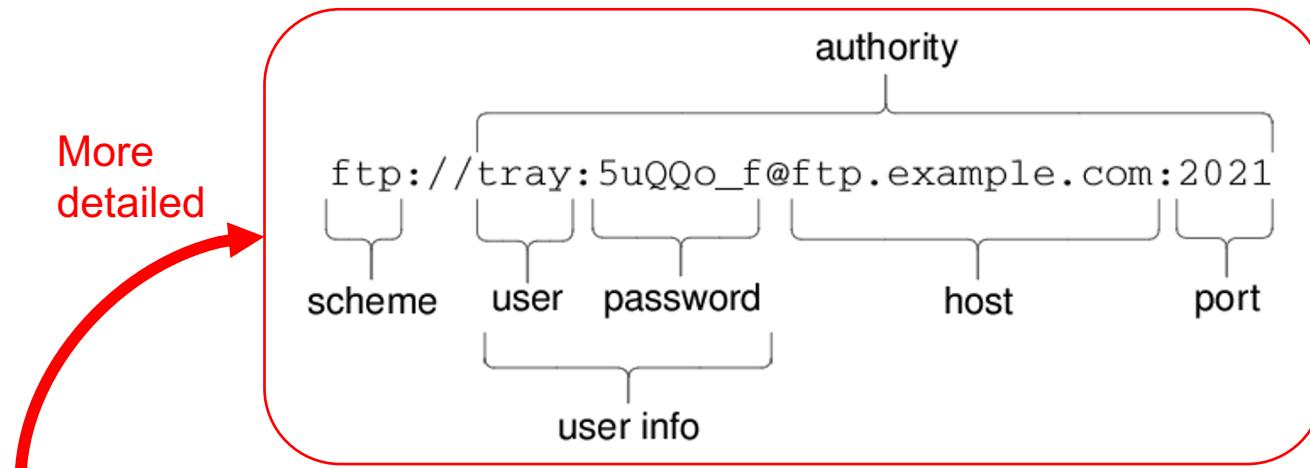
Ein Uniform Resource Identifier (Abk. URI) ist ein Identifikator und besteht aus einer Zeichenfolge, die zur Identifizierung einer Ressource (wie z.B. Webseiten, Dateien, Webservices, E-Mail-Empfängern, etc.) dient.

Name	Verwendung	Beispiel
http	HTTP	http://www.cs.vu.nl:80/globe
mailto	E-Mail	mailto:steen@cs.vu.nl
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Lokale Datei	file:/edu/book/work/chp/11/11
data	Eingefügte Daten	data:text/plain;charset=iso-8859-7,%e1%e2%e3
telnet	Anmeldung übers Netzwerk	telnet://flits.cs.vu.nl
tel	Telefon	tel:+31201234567
modem	Modem	modem:+31201234567;type=v32

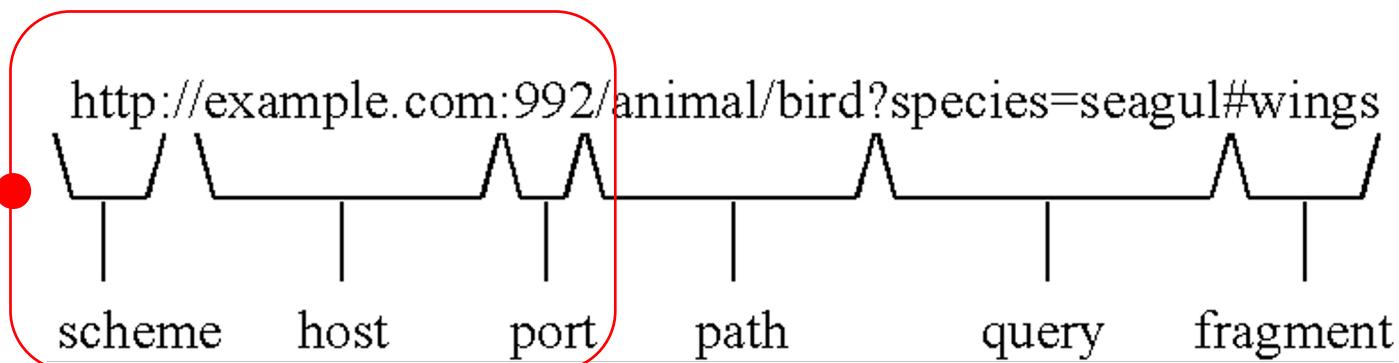
Aus: Tanenbaum, van Steen; Verteilte Systeme; Pearson; Abb. 12.16 (Bsp. für URIs)

URL – Uniform Resource Locator

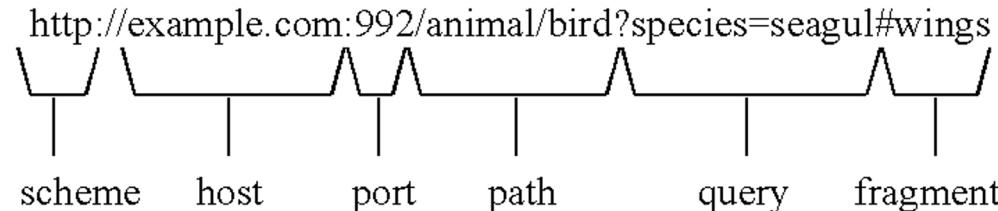
More detailed



Ein Uniform Resource Locator (Abk. URL) ist eine spezielle Form einer URI, die vor allem im Webkontext gebräuchlich ist.



https://de.wikipedia.org/wiki/Uniform_Resource_Identifier

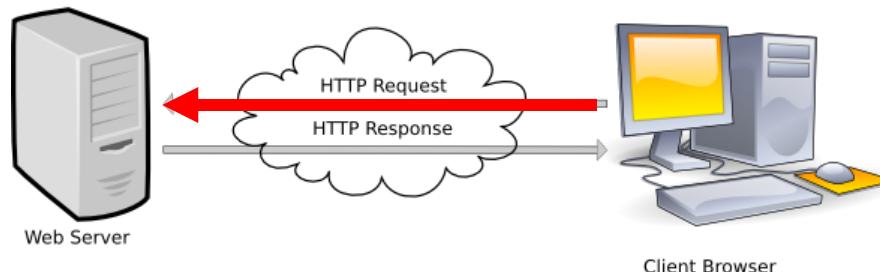


Uri

<https://api.dart.dev/stable/2.16.2/dart-core/Uri-class.html>

```
void main() {  
  Uri url = Uri.parse(  
    "http://example.com:992/animal/bird?species=seagul#wings"  
  );  
  print("Scheme:      ${url.scheme}");  
  print("Authority:  ${url.authority}");  
  print("Host:        ${url.host}");  
  print("Port:        ${url.port}");  
  print("Path:        ${url.path}");  
  print("Query:       ${url.query}");  
  print("Params:      ${url.queryParameters}");  
  print("Fragment:    ${url.fragment}");  
}
```

HTTP Request



GET

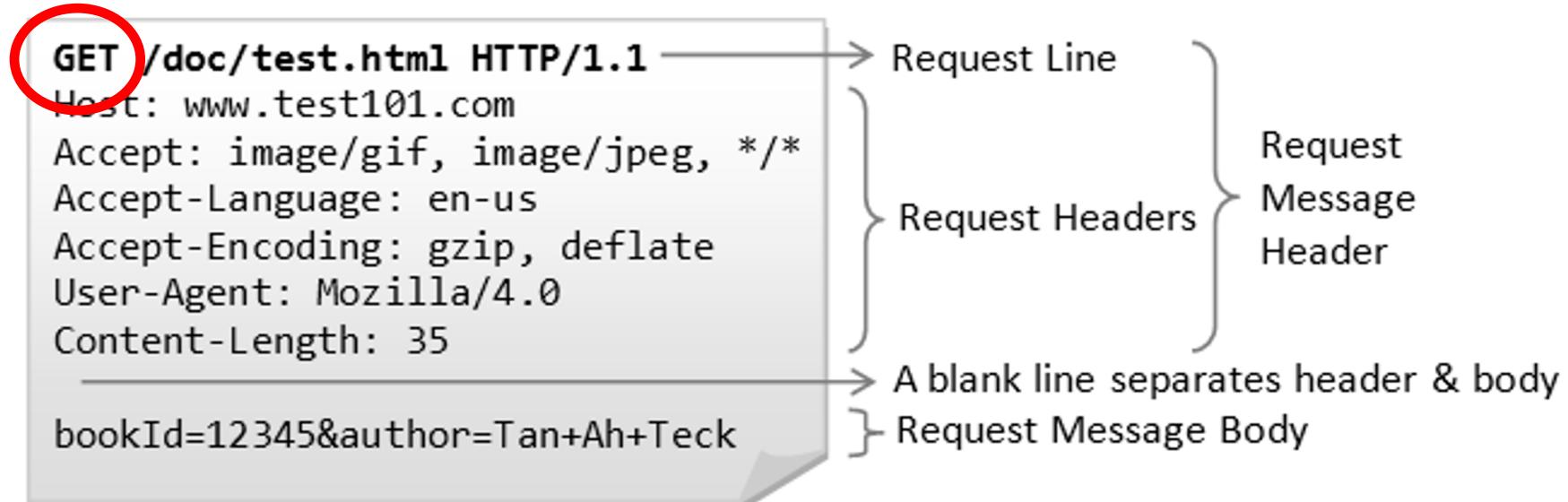
POST

HEAD

PUT

DELETE

OPTIONS



HTTP Methoden

HTTP wurde als allgemein verwendbares Client-Server Protokoll entworfen. D.h. Ressourcen lassen sich nicht nur lesend (GET) von einem Server anfordern (was im Web sicherlich die am meisten verwendete Form ist, wenn man mit Webbrowsern „surft“), sondern auch erzeugen (POST), verändern (PUT), löschen (DELETE) bzw. nur Meta Informationen abrufen (HEAD).

Welche Methoden auf einer Ressource anwendbar sind, lässt sich mittels OPTIONS herausfinden.

Operation	Beschreibung
Head	Anforderung der Rückgabe eines Dokumentenkopfes
Get	Anforderung der Rückgabe eines Dokumentes zum Client
Put	Anforderung der Speicherung eines Dokumentes
Post	Verfügbar machen von Daten, die einem Dokument (bzw. einer Sammlung) hinzugefügt werden sollen
Delete	Anforderung der Löschung eines Dokumentes

Aus: Tanenbaum, van Steen; Verteilte Systeme; Pearson; Abb. 12.11 (Von HTTP unterstützte Operationen/Methoden; nicht vollständig, umfasst nur die häufigsten Operationen)

HTTP Request (Header Fields)

Aus: Tanenbaum, van Steen; Verteilte Systeme; Pearson; Abb. 12.13
(Einige HTTP-Nachrichtenköpfe)

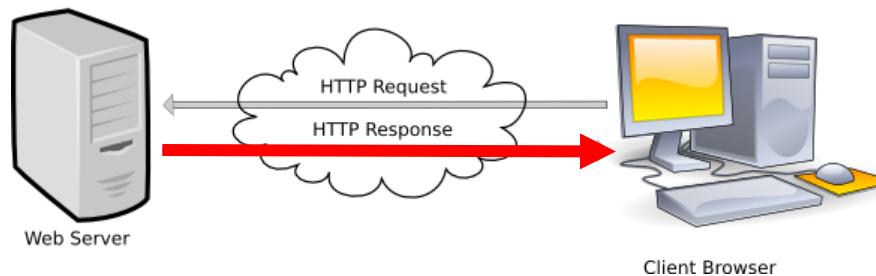
Kopf	Quelle	Inhalt
Accept	Client	Die Dokumententypen, mit denen der Client umgehen kann
Accept-Charset	Client	Die für den Client akzeptablen Zeichensätze
Accept-Encoding	Client	Die Dokumentkodierungen, mit denen der Client umgehen kann
Accept-Language	Client	Die natürliche Sprache, mit denen der Client umgehen kann
Authorization	Client	Eine Liste der Anmeldeinformationen des Clients
WWW-Authenticate	Server	Sicherheitsaufforderung, die der Client beantworten muss
Date	Beide	Datum und Uhrzeit, zu der die Nachricht versandt wurde
ETag	Server	Die dem zurückgegebenen Dokument zugeordneten Tags
Expires	Server	Die Zeitdauer, für die die Antwort Gültigkeit behält
From	Client	Die E-Mail-Adresse des Clients
Host	Client	Der DNS-Name des Servers, auf dem sich das Dokument befindet
If-Match	Client	Die Tags, die das Dokument aufweisen sollte
If-None-Match	Client	Die Tags, die das Dokument nicht aufweisen sollte
If-Modified-Since	Client	Fordert den Server auf, ein Dokument nur dann zurückzugeben, wenn es nach dem angegebenen Zeitpunkt verändert wurde
If-Unmodified-Since	Client	Fordert den Server auf, ein Dokument nur dann zurückzugeben, wenn es nach dem angegebenen Zeitpunkt nicht mehr verändert wurde
Last-Modified	Server	Der Zeitpunkt der letzten Änderung des Dokumentes
Location	Server	Ein Dokumentenverweis, auf den der Client seine Anforderung umleiten sollte
Referer	Client	Bezieht sich auf das vom Client zuletzt angeforderte Dokument
Upgrade	Beide	Das Anwendungsprotokoll, zu dem der Absender wechseln will
Warning	Beide	Informationen zum Status der Daten in der Nachricht

Über eine Reihe von Headern kann der Client dem Server mitteilen, welche Arten von Antworten er verarbeiten kann.

Umgekehrt, kann der Server dem Client über Header mitteilen, welche Art von Dokument ausgeliefert wurde.

<https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>

HTTP Response



HTTP/1.1 200 OK

Date: Sun, 08 Feb xxxx 01:11:12 GMT
 Server: Apache/1.3.29 (Win32)
 Last-Modified: Sat, 07 Feb xxxx
 ETag: "0-23-4024c3a5"
 Accept-Ranges: bytes
 Content-Length: 35
 Connection: close
 Content-Type: text/html

<h1>My Home page</h1>

Status Line

Response Headers

Response Message Header

A blank line separates header & body

Response Message Body

Content Types

Typ	Untertyp	Beschreibung
Text	Plain	Unformatierter Text
	HTML	Text mit HTML-Auszeichnungsbefehlen
	XML	Text mit XML-Auszeichnungsbefehlen
Image	GIF	Standbild im GIF-Format
	JPEG	Standbild im JPEG-Format
Audio	Basic	Audio, 8-Bit PCM, Samplingrate 8000 Hz
	Tone	Ein bestimmter hörbarer Ton
Video	MPEG	Film im MPEG-Format
	Pointer	Darstellung eines Zeigegerätes für Präsentationen
Application	Octet-Stream	Eine uninterpretierte Bytefolge
	Postscript	Ein druckbares Dokument im PostScript-Format
	PDF	Ein druckbares Dokument im PDF-Format
Multipart	Mixed	Unabhängige Teile in der angegebenen Reihenfolge
	Parallel	Einzelbestandteile müssen gleichzeitig betrachtet werden

MIME =
 Multipurpose Internet Mail Exchange
 Ursprünglich für Mail Protokolle gedacht, wird aber auch für Content-Types bei HTTP genutzt, um die Art des Dokumentes in der HTTP Response anzugeben.

Aus: Tanenbaum, van Steen; Verteilte Systeme; Pearson; Abb. 12.2 (Sechs MIME-Typen der allgemeinen Ebene und einige verbreitete Untertypen, es fehlt allerdings bspw. application/json)

HTTP Statuscodes

- **1xx** Informationen
- **2xx** Erfolgreiche Operation
- **3xx** Umleitung (Redirect)
- **4xx** Client Fehler (z.B. Bad Request, Not Found)
- **5xx** Server Fehler



Gute Übersicht gebräuchlicher HTTP Statuscodes:

<http://www.restapitutorial.com/httpstatuscodes.html>

Inhalte

- **HTTP Protokoll**
 - Hypertext Transfer Protocol (HTTP)
 - URI und URL
 - HTTP Request + Response
 - Content-Types (MIME)
 - HTTP Status Codes
- **REST**
 - Request/Response
 - Representational State Transfer
 - 5 REST Prinzipien
 - HATEOAS (Hypermedia as the Engine of Application State)
- **Ein REST-basierter Greeter Service**
 - Python + Flask
 - HTTP-Routes
 - CORS (Cross-Origin Resource Sharing)

REST in a Nutshell ...

- **Representational State Transfer (REST)** ist ein Architekturstil für verteilte Systeme auf Basis von **HTTP**
- REST APIs dienen primär der **Maschine-Maschine-Kommunikation**
- REST ist eine Alternative zu ähnlichen Ansätzen wie SOAP oder RPC
- REST Requests arbeiten auf **Ressourcen** (anders als bspw. RPC)
- Client und Server können in anderen Sprachen realisiert sein (**language agnostic**)
- **REST Responses** nutzen daher gerne sprachunabhängige Serialisierungsformate wie bspw. JSON oder XML
- Die für REST nötige **Infrastruktur und Protokolle** sind im WWW bereits vorhanden
- Da das WWW (horizontal) skalierbar ist, sind es REST-basierte Systeme zumeist auch (wenn man ein paar Dinge beachtet)



https://de.wikipedia.org/wiki/Representational_State_Transfer

Zum Nachlesen ...

UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

2000

Kapitel 5: Representational State Transfer (REST)

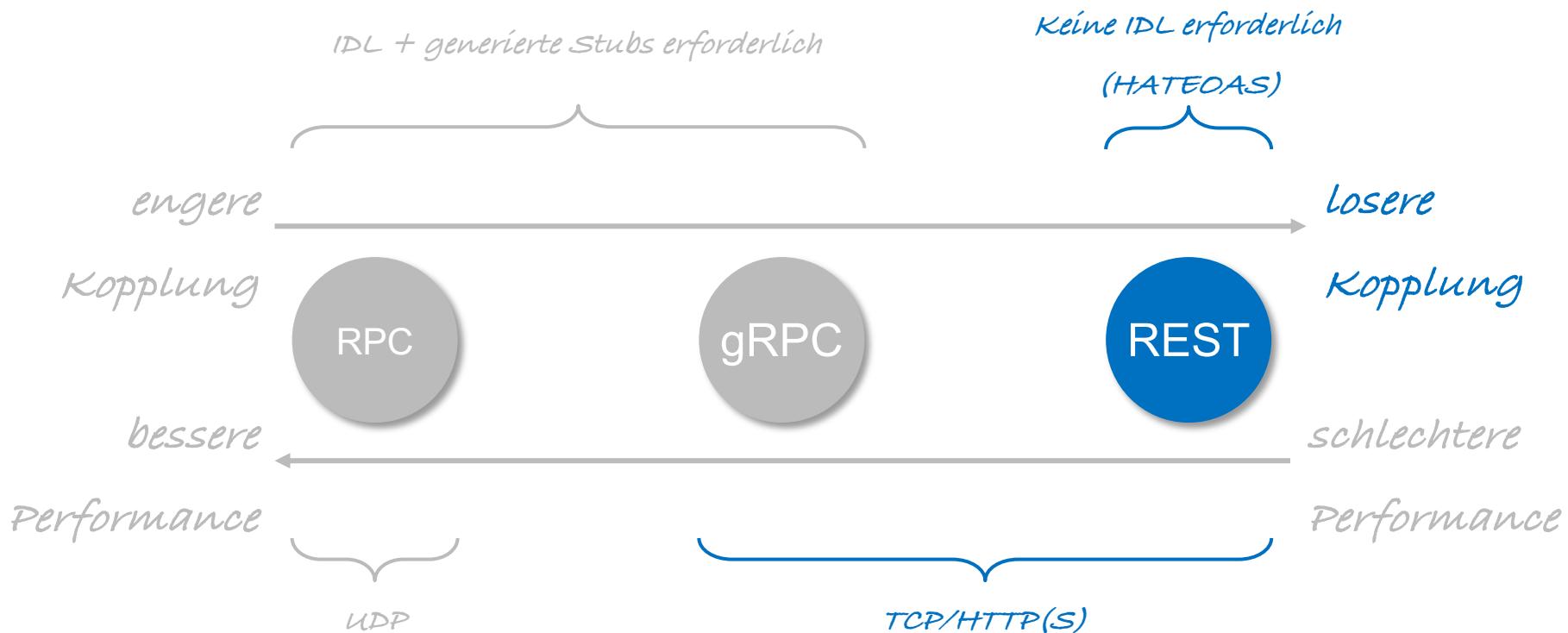
5.1 Deriving REST

5.2 REST Architectural Styles

5.3 REST Architectural Views

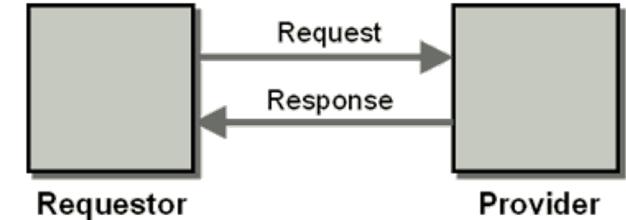
https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Request-Response basierte Kopplung



Request/Response: Representational State Transfer (REST)

REST ist ein Paradigma für die Softwarearchitektur von verteilten Systemen, insbesondere für Webservices. REST hat das Ziel, einen Architekturstil zu schaffen, der die Anforderungen des modernen Web besser darstellt. REST fordert eine einheitliche Schnittstelle zu Ressourcen im Netz.



Der Zweck von REST liegt schwerpunktmäßig auf der Maschine-zu-Maschine-Kommunikation. REST stellt eine einfache Alternative zu ähnlichen Verfahren wie SOAP und WSDL und dem verwandten Verfahren RPC dar. Anders als bei vielen verwandten Architekturen kodiert REST keine Methodeninformation in Uniform Resource Identifiern (URI), da der URI Ort und Namen der Ressource angibt, nicht aber die Funktionalität, die der Web-Dienst zu der Ressource anbietet.

Der Vorteil von REST liegt darin, dass im WWW bereits ein Großteil der für REST nötigen Infrastruktur (z. B. Web- und Application-Server, HTTP-fähige Clients, usw.) vorhanden ist, und viele Web-Dienste per se REST-konform sind. Eine Ressource kann dabei über verschiedene Medientypen repräsentiert werden.

REST: 5 Prinzipien

Client-Server

1

Dabei stellt der Server (Provider) einen Dienst bereit, der bei Bedarf vom Client (Requestor) angefragt werden kann. Der Hauptvorteil den diese Anforderung bringt ist die einfache Skalierbarkeit der Server, da diese unabhängig vom Client agieren. Dies ermöglicht u.a. eine Unterschiedlich schnelle Entwicklung der beiden Komponenten.

Caching

2

HTTP Caching kann zur Performance-Optimierung genutzt werden. Mittels Caching kann es allerdings passieren, dass Clients auf veraltete Cache-Daten zurückgreifen.

Zustandslosigkeit (Stateless)

3

Jede REST-Nachricht enthält alle Informationen, die für den Server bzw. Client notwendig sind, um die Nachricht zu verstehen. Weder der Server noch die Anwendung soll Zustandsinformationen zwischen zwei Nachrichten speichern. Jeder Request eines Clients an den Server ist insofern in sich geschlossen, als dass sie sämtliche Informationen über den Anwendungszustand beinhaltet, die vom Server für die Verarbeitung der Anfrage benötigt werden.

Mehrschichtige Systeme

4

Die Systeme sollen mehrschichtig aufgebaut sein. Dadurch reicht es, dem Anwender lediglich eine Schnittstelle anzubieten.

Dahinterliegende Ebenen können verborgen bleiben und somit die Architektur insgesamt vereinfacht werden. Dies ermöglicht eine bessere horizontale Skalierbarkeit der Server, sowie eine mögliche Abkapselung durch Firewalls.

REST: 5 Prinzipien

Einheitliche Schnittstelle

5.1 Selbstbeschreibende Nachrichten

REST-Nachrichten sollen selbstbeschreibend sein. Dazu zählt u. a. die Verwendung von Standardmethoden (bspw. HTTP-Verben). Über diese Standardmethoden lassen sich Ressourcen manipulieren.

5.2 Adressierbarkeit von Ressourcen

Jede Ressource eines Service wird über einen Uniform Resource Identifier (URI) identifiziert. Jeder REST-konforme Service ist über einen Uniform Resource Locator (URL) adressierbar. Dieser Locator standardisiert den Zugriffsweg zum Angebot eines Webservices für eine Vielzahl von Anwendungen (Clients).

5.3 Repräsentationen von Ressourcen

Ressourcen können unterschiedliche Darstellungsformen (Repräsentationen) haben. Ein REST-konformer Service kann verschiedene Repräsentationen einer Ressource ausliefern (häufig HTML, JSON oder XML) oder auch die Beschreibung oder Dokumentation des Dienstes. Die Veränderung einer Ressource (also deren Status) soll nur über eine Repräsentation erfolgen.

5.1

Hypermedia as the Engine of Application

State (HATEOAS)

Bei HATEOAS navigiert der Client einer REST-Schnittstelle ausschließlich über URLs, die vom Service bereitgestellt werden. Abhängig von der gewählten Repräsentation geschieht die Bereitstellung der URLs über Hypermedia,

- also z. B. in Form von „href“- und „src“-Attributen bei HTML-Dokumenten, oder
- in für die jeweilige Schnittstelle definierten und dokumentierten JSON- bzw. XML-Attributen-/Elementen.

HATEOAS ermöglicht dadurch lose Kopplung von Services und gewährleistet, dass die Schnittstelle verändert werden kann.

5.2

5.4

5.3

HATEOS - Beispiel

Beispiel für einen GET-Request, der Konto-Informationen im JSON-Format abruft.

```
GET /accounts/123abc
HTTP/1.1 Host: bank.example.com
Accept: application/json
...
...
```

HATEOAS

Examplarische Antwort inkl. HATEOAS-konformer Folgeoperationen (Links)

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: ...

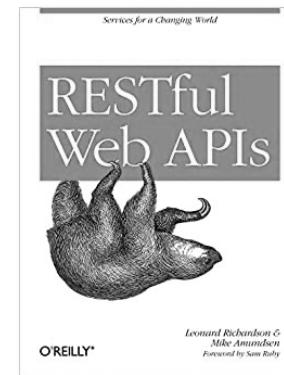
{
  "account": {
    "account_id": "123abc",
    "balance": {
      "currency": "EUR",
      "value": 100.0
    },
    "links": {
      "deposit": "/accounts/123abc/deposit",
      "withdraw": "/accounts/123abc/withdraw",
      "transfer": "/accounts/123abc/transfer",
      "close": "/accounts/123abc/close"
    }
  }
}
```

REST Maturity Model

Level Eigenschaften

- | | |
|---|---|
| 0 | <ul style="list-style-type: none"> • verwendet XML-RPC oder SOAP • der Service wird über einen einzelnen URI adressiert • verwendet eine einzelne HTTP-Methode (oft POST) |
| 1 | <ul style="list-style-type: none"> • verwendet verschiedene URIs und Ressourcen • verwendet eine einzelne HTTP-Methode (oft POST) |
| 2 | <ul style="list-style-type: none"> • verwendet verschiedene URIs und Ressourcen • verwendet mehrere HTTP-Methoden |
| 3 | <ul style="list-style-type: none"> • basiert auf HATEOAS und verwendet daher Hypermedia für Navigation • verwendet verschiedene URIs und Ressourcen • verwendet mehrere HTTP-Methoden |

Das Richardson Maturity Model (RMM) ist ein von Leonard Richardson entwickelter Maßstab, der angibt, wie REST-konform ein Service implementiert wurde.



Für REST benötigte HTTP Methoden

Methode	Bedeutung	Idempotent	Sicher	CRUD
POST	Fügt eine neue Ressource hinzu			create
GET	Fordert eine Ressource an	x	x	read
PUT	Ändert eine Ressource	x		update
DELETE	Löscht eine Ressource	x		delete

Sicher (safe): Eine Operation ändert nicht den Zustand (nebeneffektfreie Operation)

Idempotent: Wird eine Operation zweimal hintereinander ausgeführt, ist die zweite Operation sicher (d.h. ändert nichts mehr am Zustand)

Inhalte

- **HTTP Protokoll**
 - Hypertext Transfer Protocol (HTTP)
 - URI und URL
 - HTTP Request + Response
 - Content-Types (MIME)
 - HTTP Status Codes
- **REST**
 - Request/Response
 - Representational State Transfer
 - 5 REST Prinzipien
 - HATEOAS (Hypermedia as the Engine of Application State)
- **Ein REST-basierter Greeter Service**
 - Python + Flask
 - HTTP-Routes
 - CORS (Cross-Origin Resource Sharing)

Ein einfacher REST-basierter Hello World Service

Methode	Resource	CRUD	Beschreibung	Beispiel
POST	/greet/:lang	Create	Erzeugt einen neuen Gruß	POST /greet/deutsch greet=hallo%20welt
GET	/greet/:lang	Read	Liest einen Gruß für Sprache :lang	GET /greet/deutsch
PUT	/greet/:lang	Update	Ändert einen Gruß für Sprache :lang	PUT /greet/deutsch Greet=Hallo%20Welt
DELETE	/greet/:lang	Delete	Löscht den Gruß der Sprache :lang	DELETE /greet/deutsch
GET	/greets	List	Listet alle im Service gespeicherten Grüße (Antwort als JSON Map)	GET /greets

```
>HELLO WORLD  
>_
```

Das REST-Prinzip soll exemplarisch an einem kleinen Hello World Service demonstriert werden, der mehrsprachige Gruß Ressourcen („Hello World“ Floskeln) verwaltet. Der Service soll um Grüße in fremden Sprachen ergänzt werden, einen Gruß in einer spezifischen Sprache ausgeben, Grüße bestehender Sprachen ändern und löschen können. Ferner soll er einen Überblick über alle hinterlegten Sprachen und zugeordneten Grüße liefern können.

Ein einfacher REST Greeting-Server in Flask

```
from flask import Flask, request, jsonify

app = Flask(__name__)

greets = {
    "nl" : "Hello wereld",
    "en" : "Hello world",
    "fr" : "Bonjour monde",
    "de" : "Hallo Welt",
    "it" : "Ciao mondo",
    "pt" : "Olá mundo",
    "es" : "Hola mundo"
}
# [...]
app.run(host="0.0.0.0")
```

<https://git.mylab.th-luebeck.de/webtech/rest-api>

OPTIONS /greet/

Es gibt diverse HTTP Client Libraries, die bei POST, DELETE und PUT vorher per OPTIONS anfragen, was für Methoden auf einer Ressource überhaupt zulässig sind.

Es macht für diese Fälle Sinn für jede Ressource einen entsprechenden OPTIONS Handler vorzusehen, ansonsten scheitert ein Request bereits beim BITTE BITTE sagen.



```
# Test with:  
  
# curl -X OPTIONS http://localhost:5000/greet  
  
@app.route("/greet", methods=["OPTIONS"])  
def options_greet():  
    return "POST, GET, DELETE, PUT, OPTIONS"
```

POST /greet/<lang>

```
# Test with:
# curl http://localhost:5000/greet/python
# curl -X POST -d "text=print('Hello Python')" \
http://localhost:5000/greet/py
# curl http://localhost:5000/greet/py

@app.route("/greet/<lang>", methods=["POST", "PUT"])
def post_greet(lang):
    orig = greets.get(lang, None)
    text = request.values.get("text", None)
    if not text:
        return f"Bad request: Invalid text parameter", 400
    greets[lang] = text
    return str(orig)
```



text=Hallo%20Welt

GET /greet/<lang>



```
# Test with:  
  
# curl http://localhost:5000/greet/de  
# curl http://localhost:5000/greet/en  
# curl http://localhost:5000/greet/missing  
  
@app.route("/greet/<lang>", methods=["GET"] )  
def get_greet(lang):  
    if lang not in greets:  
        return f"{ lang } not found", 404  
    return greets[lang]
```

DELETE /greet/<lang>

```
# Test with:  
  
# curl -X DELETE http://localhost:5000/greet/de  
# curl -X DELETE http://localhost:5000/greet/de  
# curl -X DELETE http://localhost:5000/greet/missing  
  
@app.route("/greet/<lang>", methods=["DELETE"] )  
def delete_greet(lang):  
    orig = greets.get(lang, None)  
    if orig:  
        del greets[lang]  
    return str(orig)
```

Eine DELETE Operation ist idempotent.

D.h. wiederholte Löschung (bzw. das Löschen nicht existenter Ressourcen) sollten nicht in einem Fehler enden.



GET /greets

```
# Test with:  
  
# curl http://localhost:5000/greets  
  
@app.route("/greets", methods=["GET"] )  
def get_greets():  
    return jsonify(greets)
```



Liefert alle greet Ressourcen
in einem JSON Format.

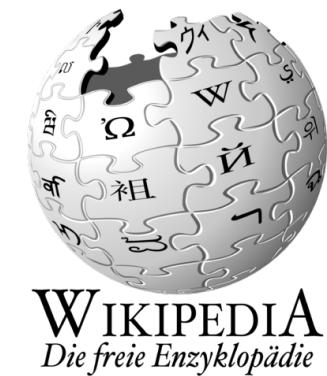
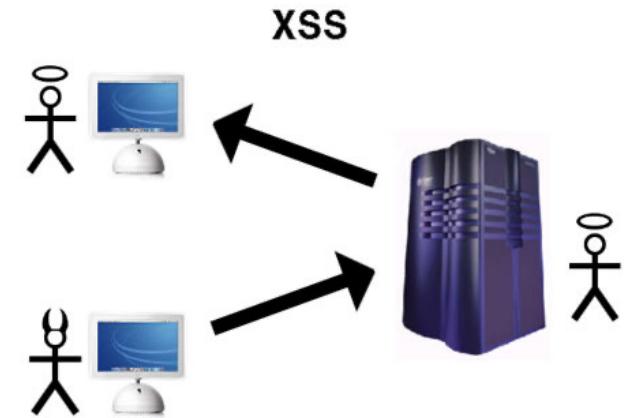
Same-Origin-Policy

Die Same-Origin-Policy (SOP) ist ein Sicherheitskonzept, das clientseitigen Skriptsprachen untersagt, auf Ressourcen zuzugreifen, deren Speicherort nicht dem Origin entspricht. Sie stellt ein Sicherheitselement in modernen Browsern zum Schutz vor Angriffen dar.

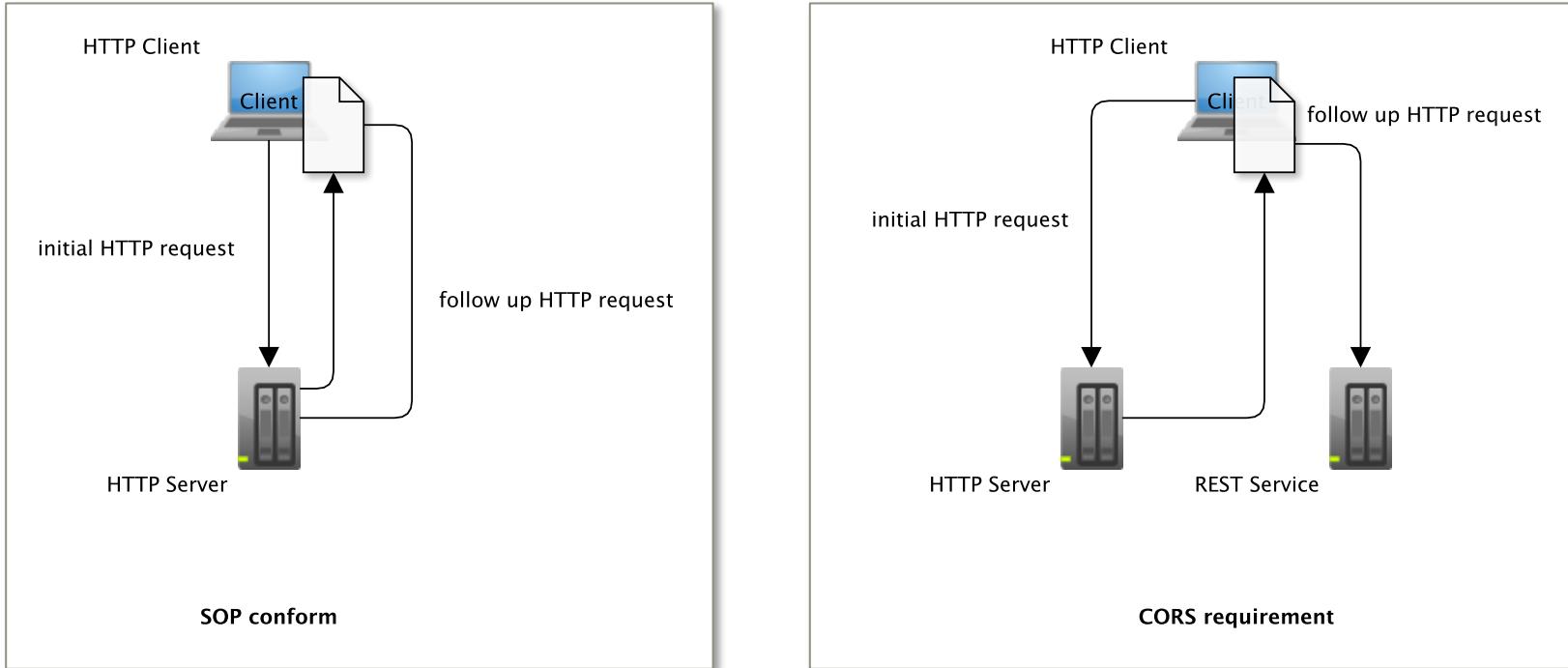
Hintergrund: Skriptsprachen im Browser haben Zugriff auf Kommunikation zwischen Browser und Web-Server. Dies beinhaltet sowohl das Auslesen, die Manipulation, das Empfangen und Senden von Daten.

Daraus ergibt sich die Sicherheitsanforderung, dass keine Daten aus einem Kontext (zum Beispiel der Verbindung des Browsers zu der Seite einer Bank) von einem Skript aus einem anderen Kontext zugreifbar oder manipulierbar sein darf. Um dies zu erreichen, wird beim Zugriff eines Skriptes auf eine Ressource die Herkunft (origin) von beiden verglichen. Der Zugriff wird nur erlaubt, wenn Skript und angeforderte Ressource dieselbe Herkunft (origin) haben (vom selben Server stammen).

Quelle: <https://de.wikipedia.org/wiki/Same-Origin-Policy>



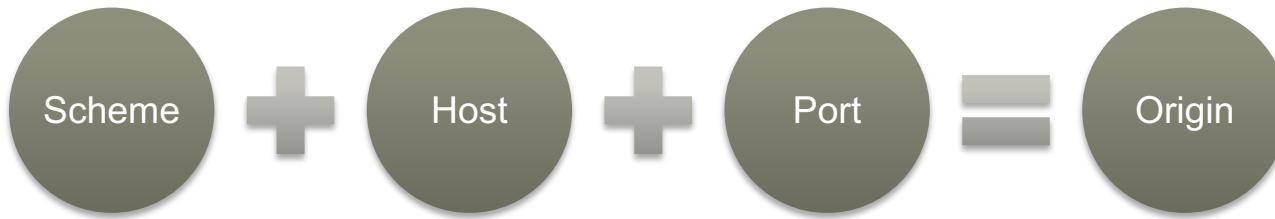
SOP und REST vertragen sich nicht immer



Die Same-Origin-Policy (SOP) ist dann ggf. hinderlich, wenn von einem Server eine Webapplikation geladen wird, die mit einem zweiten (nicht notwendig identischem) Server eine z.B. REST-basierte Kommunikation aufbauen muss.

In solchen Fällen kann der HTTP Server, der follow up requests bearbeitet, seine Responses mittels CORS Headers (Cross Origin Resource Sharing) kennzeichnen. Der HTTP Client verzichtet dann auf eine Same Origin Prüfung.

Same Origin Definition



`http://www.example.com/dir/page.html`

Compared URL	Outcome	Reason
<code>http://www.example.com/dir/page.html</code>	Success	Same protocol and host
<code>http://www.example.com/dir2/other.html</code>	Success	Same protocol and host
<code>http://www.example.com:81/dir/other.html</code>	Failure	Same protocol and host but different port
<code>https://www.example.com/dir/other.html</code>	Failure	Different protocol
<code>http://en.example.com/dir/other.html</code>	Failure	Different host
<code>http://example.com/dir/other.html</code>	Failure	Different host (exact match required)
<code>http://v2.www.example.com/dir/other.html</code>	Failure	Different host (exact match required)

Die Same-Origin-Policy (SOP) ist erfüllt, wenn bei zwei Ressourcen deren URL-Anteile Scheme, Host und Port übereinstimmen.

```
from flask import Flask, request, jsonify
from flask_cors import CORS

app = Flask(__name__)
CORS(app) # This allows CORS for all domains on all routes

# [...]
app.run(host="0.0.0.0")
```

Cross-Origin Resource Sharing (CORS) ist ein Mechanismus, der Webclients Cross-Origin-Requests ermöglicht. Die Einschränkungen, die durch die SOP auferlegt sind, können vom Server aufgehoben werden. Der Server kann den Zugriff durch entsprechender Access-Control-Allow-* Response Header erlauben/einschränken.

- **HTTP**

- Uniform Resource Identifiers (URI) und Locators (URL)
- HTTP Request (Client) und Response (Server)
- HTTP Methoden
- HTTP Header
- HTTP Content Types (MIME)
- HTTP Status Codes
- Dart Libraries für HTTP



- **Same Origin Policy (SOP) und Cross Origin Resource Sharing (CORS)**

- **REST**

- CRUD und HTTP Methoden
- Sichere und idempotente Operationen
- REST am Bsp. eines Hello World Service
- Flask (Python) Library

