REST in the Cloud – Einführung in Webservices ohne Seife und Sonnenschein

Lea Rain

Gulaschprogrammiernacht 22

Samstag, 01.06.2024

Einführung

WHOAMI

- lea@gpn22
- Binary Kitchen Regensburg
- ITlerin & your friendly neighborhood hacker
- Informatikstudium & IT (Security) Brötchen
- 🐃 🐧 ⌨️ ❤️

Motivation

- Webservices zur Bereitstellung von Funktionalität von Software im Internet $\Rightarrow$ Mehr als statische Seiten
- Aufruf von Funktionalität auf einem anderen Gerät als dem eigenen (potentiell über das Internet) $\Rightarrow$ Ausführung von Befehlen in einem verteilten System und nicht nur lokal auf dem eigenen Gerät
- Austausch von (verarbeiteten) Daten, klassische Probleme von gesprächsfreudigen, sehr weit ausreizbaren und potentiell gefährlichen Ausführungen auf externen Systemen

Disclaimer – Erfahrungen aus erster Hand?

- Thematisch: Punkte historischer Entwicklung des Internets als Kommunikationsmedium
- Darstellung einzelner Punkte mit Relevanz für Thematik REST und Webservices, nicht der Gesamtentwicklung
- Wenig Erfahrung aus erster Hand, Zurückgreifen auf historische Dokumentation
- Chronologisch nicht korrekt

Historische Internetnutzung – Clientseitig

Internet Früher (heute Vintage)

Abbildung: Quelle: `https://www.regensburger-busse.de/`

Statische Webseiten

- Zeigen Inhalt. Statisch.
- Etwas ausführlicher: Auslieferung von vorher definierten Dateien auf einem Webserver (meist HTML, CSS, clientseitiges Javascript (?))

```
resources
├── index html
├── foo html
├── css
│   └── bootstrap css
└── images
    └── cat jpg
```

Kommunikation statische Webseiten

Client: Webbrowser, läuft auf Endgerät von Nutzer:in

Server: Webserver, externes System, hostet Dateien

Client $\xrightarrow{\text{Anfrage zu Datei cat jpg}}$ Server

Client $\xleftarrow{\text{Antwort mit Datei cat jpg}}$ Server

Interpretation der Datei durch Client und Anzeige

CAT JPG

Statische Webseiten... bisschen Dynamik?

- Simpler Abruf von Dateien,
- Dynamisches HTML: bisschen Interaktion, Reaktion auf Ereignisse ausgelöst durch Client
- Beispiel: Wünschenswertes Ereignis? Dynamischer Wechsel zum Dark Mode

$\Rightarrow$HTML reicht nicht mehr

Javascript

```
<script>
    alert("Hello Gulasch!");
</script>
```

(kein XSS, promise!)

Sun?

Sun Microsystems

Historie von Javascript (ganz kurz)

- 1995 unter dem Namen LiveScript als Teil des Netscape Navigators
- „the most common use is to make pages a little smarter and more live for instance, make a click on a link load a different URL depending on the time of day“
- Kooperation mit Sun Microsystems (die mit Java)
- Gute Nacht Lektüre: JavaScript: the first 20 years – Allen Wirfs-Brock und Brendan Eich

Historische Client-Server-Architekturen

Client-Server-Architekturen

- Über statische Auslieferung von Dateien hinaus
- Bereitstellung von Diensten/Programmen/Funktionalität für Client durch Server
- Über simple Nutzung von Webbrowser und Webserver hinaus

Remote Procedure Call – Motivation

- Interprozesskommunikation: Kommunikation zwischen Prozessen eines Systems
- Herausforderung: Speicher teilen (und zugehörige Probleme) oder notwendige Informationen übermitteln
  $\Rightarrow$Remote Procedure Call: Funktionsaufruf wie gehabt – egal, ob lokal oder auf externem System (am Fnordpol)
- Zeitliche Einordnung?

RFC 707

NWG/RFC# 707 JEW 14-JAN-76 19:51 34263
NCC 76 A High-Level Framework for Network-Based Resource Sharing

THE GOAL, RESOURCE SHARING 1

The principal goal of all resource-sharing computer networks, including the now international ARPA Network (the ARPANET), is to usefully interconnect geographically distributed hardware, software, and human resources [1]. Achieving this goal requires the design and implementation of various levels of support software within each constituent computer, and the specification of network-wide "protocols" (that is, conventions regarding the format and the relative timing of network messages) governing their interaction. This paper outlines an alternative to the approach that ARPANET system builders have been taking since work in this area began in 1970, and suggests a strategy for modeling distributed systems within any large computer network. 1a

The first section of this paper describes the prevailing ARPANET protocol strategy, which involves specifying a family of application-dependent protocols with a network-wide inter-process communication facility as their common foundation. In the second section, the application-independent command/response discipline that characterizes this protocol family is identified and its isolation as a separate protocol proposed. Such isolation would reduce the work of the applications programmer by allowing the software that implements the protocol to be factored out of each applications program and supplied as a single, installation-maintained module. The final section of this paper proposes an extensible model for this class of network interaction that in itself would even further encourage the use of network resources. 1b

Exkurs: RFC

- RFC: Request for Comments
- Organisation und technische Spezifikation zu allem, was das Internet-Herz begehrt (für Boomer: Arpanet)
- Beispiele:
  - RFC 42: Message Data Type (8 Bit für Datentyp)
  - RFC 854: Telnet
  - RFC 1855: Netiquette Guidelines
  - RFC 2324: Hyper Text Coffee Pot Control Protocol

Zurück zu RFC 707

- Teilen von Resourcen in verteilten Systemen und Netzwerken
- Anforderungen: Aufruf von benannten Remote-Funktionen, sinnvoller Output, Definition Parameteranzahl, verschiedene Parametertypen, Rückgabetypen, Aufruf weiterer Methoden, Nebenläufigkeit, leere Rückgabetypen (Reduktion Netzwerverkehr) $\rightarrow$ Schnittstellenbeschreibung (eigene Sprache dafür als Interface Definition Language)
- Procedure Call Model: Remote-Aufrufe von Funktionen teuer, mehrere Programmflüsse/Prozesse, Übertragung von Kontrolle an externen Dienst

Implementierung – Modell

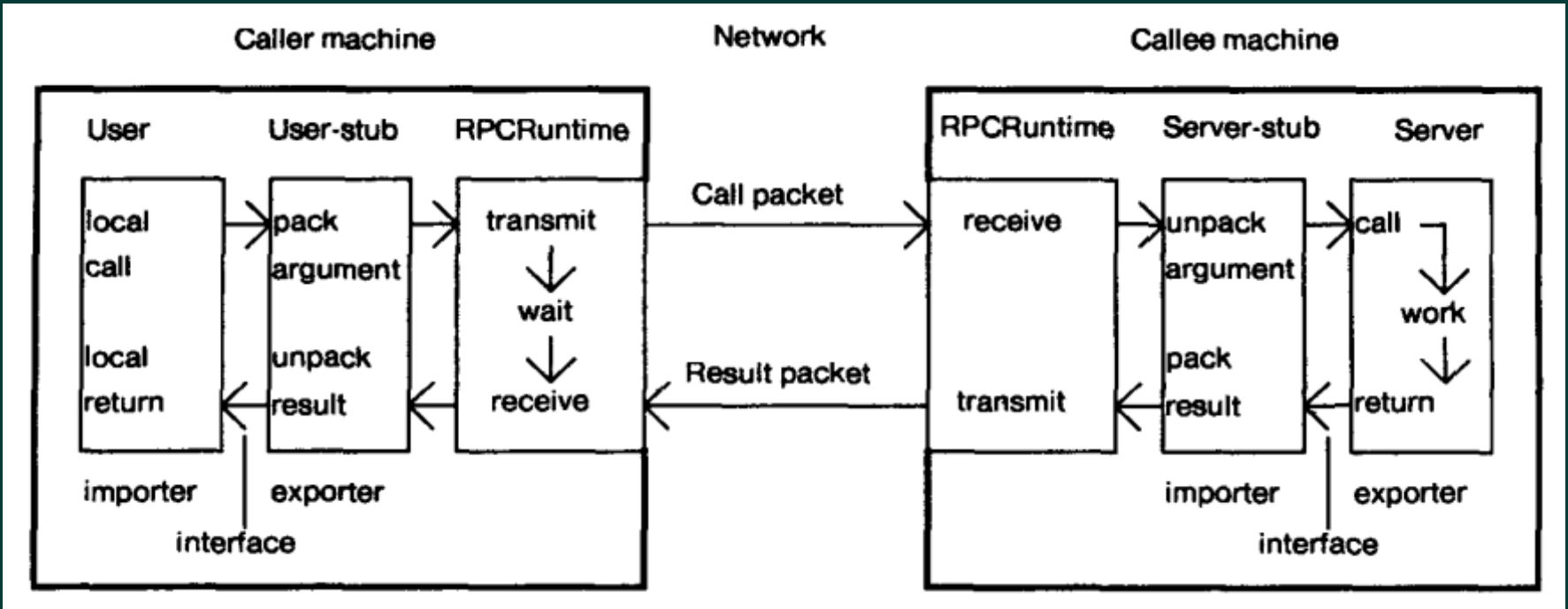
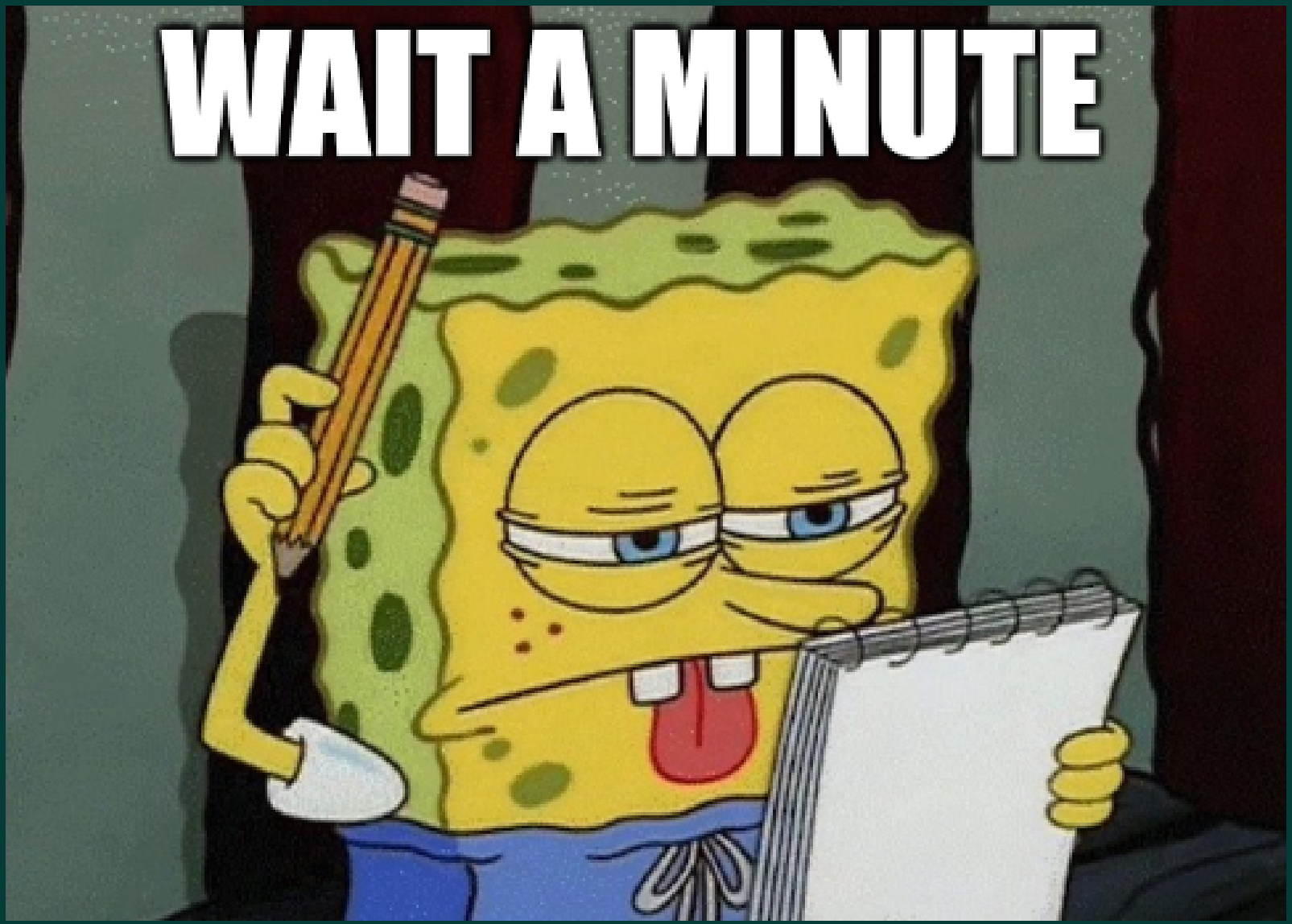


Abbildung: RPC nach Andrew D. Birrell, Bruce Jay Nelson – Implementing remote procedure calls (1984)

Sun RPC

- Sun Microsystems – mal wieder
- ONCRPC
- Verwendung für NFS (Network File System) zum Dateiaustausch (remote fühlt sich wie lokal an), für unix(oide) Systeme und in C
- RFC-Bingo: RFC 1050 (1988), RFC 1057 (1988), RFC 1831 (1995), RFC 5531 (2009)
- Noch mehr RFC-Bingo mit Authentifizierung: RFC 2203 (1997), RFC 2623 (1999) RFC 2695 (1999)

Authentifizierung?



(Sun) RPC und Authentifizierung

- Authentifizierung: Verifizierung einer Entität (Gegenüber wirklich gegenüber?)
- 10 Jahre ohne Authentifizierungsspezifikation (über ursprüngliches RFC-Konzept hinaus)
- Einfache Lösung: Null Authentication
- Anhand RFC 1050: Simple Übergabe von (u.a.) `machinename`, `uid` und `gid`
- Anhand RFC 1050: Alternative mit DES (Data Encryption Standard, Standard 1977)
- Anhand RFC 1831: Null Authentication oder System absichern, viel Spaß!

Weitere Kommunikation in verteilten Systemen

- ☕
- 🐍
- RFC
- XML-RPC (bald Händewaschen – hier kommt Seife) und JSON-RPC
- gRPC

gRPC – Wolke statt Sonnenschein

- Moderne Art von RPC, entwickelt von Google (2016)
- Großer Augenmerk auf Cloud-Nutzbarkeit (nutzbare Ressourcen anderer Entitäten) und zwischen Services
- C, Java, Python, Schnittstellenbeschreibung (Protocol Buffers) macht's möglich
- Unterstützung aktueller Authentifizierungsmethoden

Aktuelle Architekturen und (Web)Services

Serviceorientierte Architektur

- Motivation: Alternative zu Anwendungen und Systemen, die ALLES können (müssen)
- Service: Bereitstellung einer einzelnen Funktionalität oder eines Geschäftsprozesses (nicht notwendigerweise technisch)
- Idee für Unix-Fans: Make each program do one thing well.
- Interoperabilität zwischen Services, durch einheitliche Kommunikationsdefinition unabhängig von Implementierung
- Ausprägung Webservices: Schnittstelle für Kommunikation zwischen Maschinen oder für Anwendungen als standartisierte Implementierung

SOAP und WSDL

- 
- Austausch von Daten zwischen Webservices mittels XML über HTTP (SMTP auch möglich)
- XML-RPC als historischer Vorgänger (1998), Kommunikation mit RPC möglich
- WSDL – Web Services Description Language (2001): Schnittstellendefinition und Beschreibungssprache (besonderes XML)

WTFoo

XML – Extensible Markup Language

Sowohl maschinen- als auch menschenlesbares Format für Strukturierung von Daten

```
<?xml version="1.0" encoding="UTF-8"?>
<postcard>
    <send-from-event>GPN22</send-from-event>
    <receivable-at-event>GPN22</receivable-at-event>
    <sender>
        <name>Lea</name>
        <hint>caffeine addicted</hint>
        <hint>Talk@Medientheater</hint>
    </sender>
    <recipient>
        <name>GPN-Orga</name>
        <location>probably ZKM?</location>
        <hint>check the phonebook</hint>
    </recipient>
    <message>Thanks for organizing this event!</message>
</postcard>
```

HTTP – Hypertext Transfer Protocol

- Protokoll zur Übertragung von Daten in Rechnernetzen (populär: Übertragung von Daten in Hypertext-Format (Webseiten) an Browser)
- HTTP/1.0 – RFC 1945 (1996), HTTP/1.1 – RFC 2616 (1999), Spezifizierung für bestimmte Funktionalitäten – RFC 7230-7235 (2014), HTTP/2 – RFC 7540, 7541 (2015), HTTP/3 – RFC 9114 (2022)
- Zustandslos: Jede Anfrage/Kommunikation voneinander unabhängig

HTTP

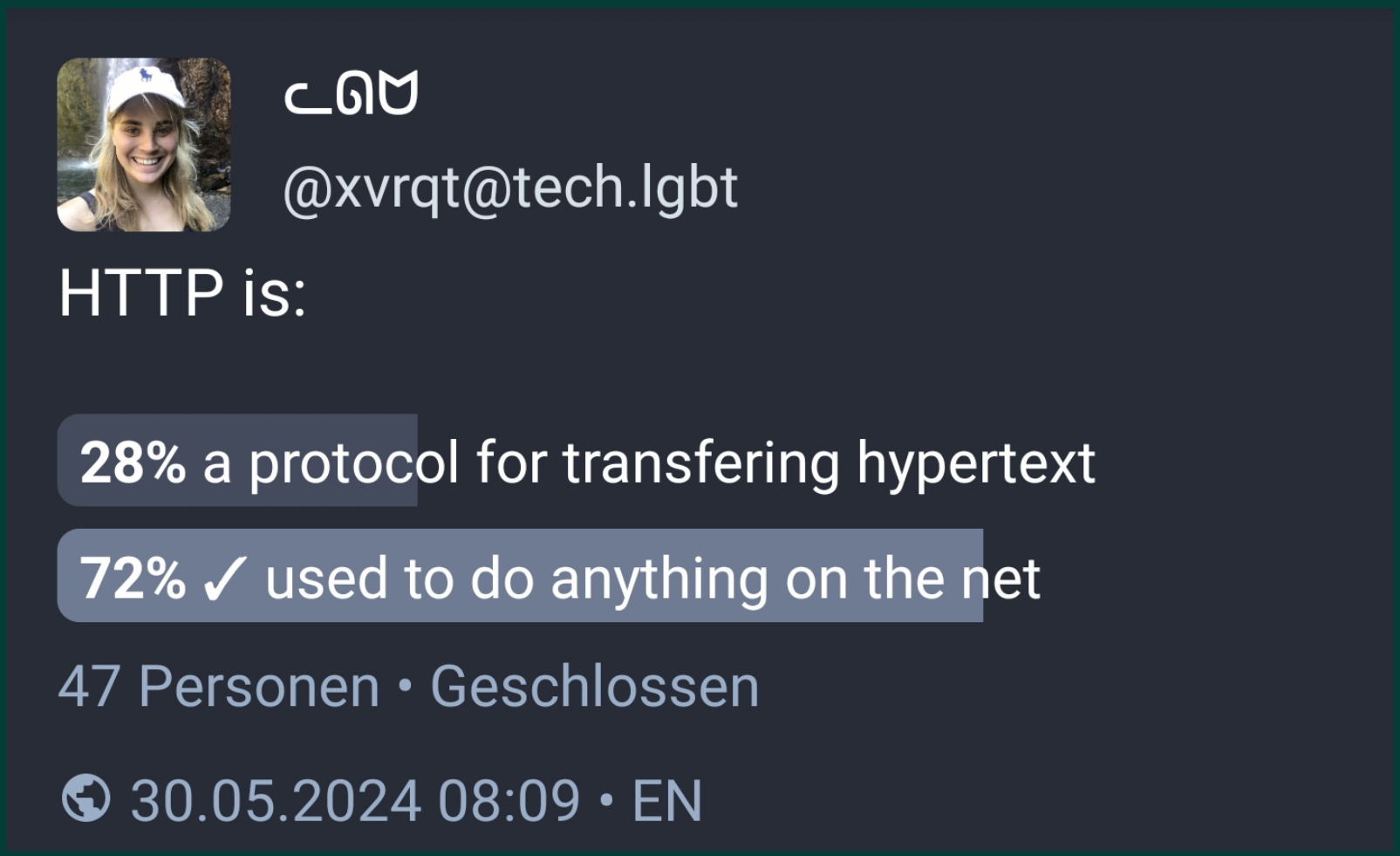


Abbildung: Quelle: `https://tech.lgbt/@xvrqt/112528548006801769`

HTTP – Beispielkommunikation

```
nc trollen.jetzt 80
GET / HTTP/1.0

HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Tue, 28 May 2024 11:45:57 GMT
Content-Type: text/html
Content-Length: 178
Connection: close
Location: https://archive.gulas.ch/

<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

HTTP-Methoden und Datenübertragung

- Daten erhalten: GET-Request
- Daten senden: POST-Request
- Daten aktualisieren: UPDATE-Request
- Daten löschen: DELETE-Request

(halbe Wahrheit, mehr Methoden verfügbar)

HTTP-Statuscodes

Statuscodes: Klassifizierung der jeweiligen Antwort des Servers an Client

- 1xx – Anfrage in Bearbeitung/Informationen
- 2xx – Erfolg
- 3xx – Redirect
- 4xx – Client-Fehler
- 5xx – Server-Fehler

Beispiele Statuscodes

- 200
- 204
- 301
- 302
- 400
- 404
- 418
- 429
- 451
- 500
- 501
- 507

WSDL-Elemente

WSDL: Beschreibungssprache für SOAP-Kommunikation
Definition eines Webservices inklusive...

- `<types>`: Datentypen-Definition innerhalb der Messages
- `<message>`: Nachrichten für Request/Response
- `<interface>`: Operationen mit Input/Output
- `<binding>`: Kommunikationsprotokoll
- `<service>`: Verbindung zugehöriger Interfaces

SOAP-Nachrichten – Beispiel

```
<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
envelope">
 <env:Header>
 </env:Header>
 <env:Body>
  <m:warning xmlns:m="http://gulas.ch/warning">
   <m:msg>Mate ist leer, Alarm!</m:msg>
  </m:warning>
 </env:Body>
</env:Envelope>
```

WTFoo

REST in Peace, SOAP!

Representational State Transfer

- Motivation: Software-Architektur nahe an den Kommunikationsstrukturen des Internets
- Client-Server-Architektur
- Zustandslos
- Caching
- Verwendung einheitlicher Schnittstellen
- Mehrschichtige Systeme mit einer Kommunikation nach außen
- potentiell auch alt – 2000

HTTP-Methoden reloaded

- Datenhaltung in Systemen mit CREATE, READ, UPDATE, DELETE (CRUD)
- Zugriff über URL/URI in Kombination mit
  - POST – CREATE
  - GET – READ
  - PUT – UPDATE
  - DELETE – DELETE

REST-APIs und OpenAPI

- REST für Maschine-zu-Maschine-Kommunikation in beliebigen Datenformaten (XML, JSON, whatever floats/soaps your boat)
- API – Application Programming Interface
- Schnittstellenbeschreibung: OpenAPI (Specification), eigenständiges Projekt seit 2016
- intuitiver als vorherige Beschreibungsdefinitionen
- REST: unabhängige Wahl des übertragenen Datenformates (JSON, XML, YAML, <weirdes Datenformat>...)

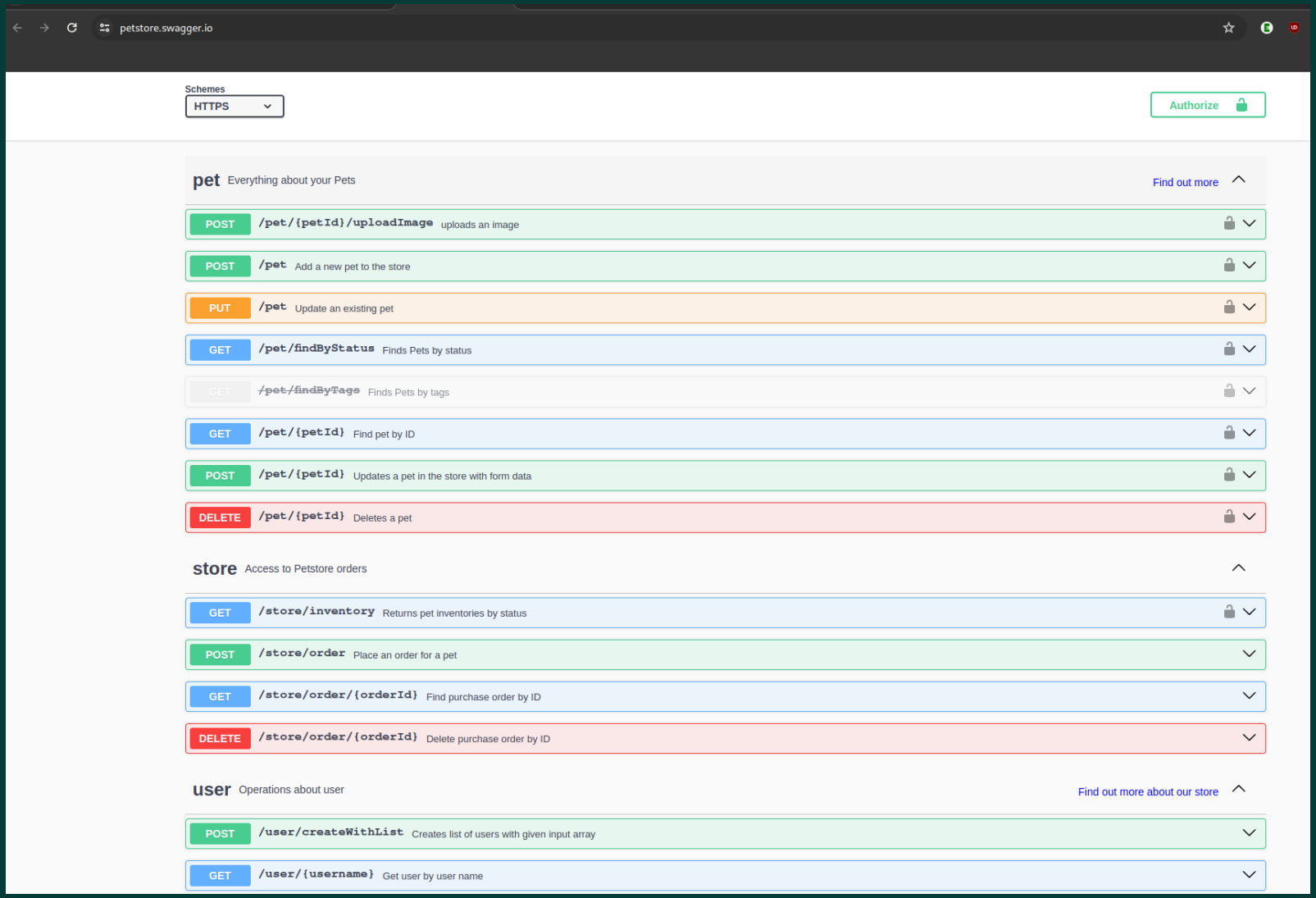
JSON-Beispiel: Request

```
GET /gulaschküche/food/01-06-2024 HTTP/1.1
Host: gulas.ch
User-Agent: Mozilla/5.0
Accept: application/json
Connection: close
```

JSON-Beispiel: JSON-Objekt

```
{
    "id": 2342,
    "valid-date": "2024-06-01",
    "event": {
        "name": "Gulaschprogrammiernacht",
        "year": 2024,
        "motto": "Common Code <> Different Roots"
     },
    "evening-food": "Gulasch",
    "persons-in-queue": [23, 37, 53, 119]
}
```

OpenAPI – Petstore



Einfaches Testen

- Zugriff auf REST-APIs zum Testen einfach realisierbar
- Simple HTTP-Requests
- Tools:
  - Browser-Konsole (quick and dirty)
  - cURL (für Boomer
  - Postman & Insomnia (angenehmeres Interface)
  - IDE-Integration (wenn IDE eh da)

Nächster Schritt: Eigene Anwendungen

- Motivation: Cool, Projekt X oder System Y bietet eine offene API an!
- Schwierige Webinterfaces, verwirrende Frontends, ... goodbye
- Build your own application – von kleinen Bash-Skripten zu großen eigenen Webanwendungen, ausprobieren erlaubt und erwünscht

Ende?

Ente gut, alles gut?

How not to design REST APIs

Not intended

Zustandslosigkeit – Authentifizierung?

- Zustandslosigkeit: Jede Anfrage unabhängig und steht für sich
- Problem: Nutzung bestimmter Funktionen nur nach Authentifizierung
- Möglicher Umgang: `POST`-Request mit Credentials, Response mit Token (für künftige Anfragen), API-Keys
- Nähe zu HTTP: Verwendung von für HTTP definierte Authentifizierung

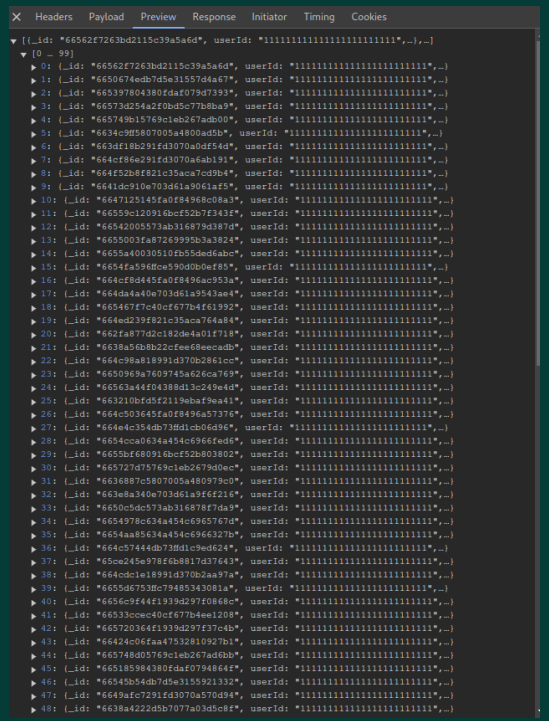
Mögliche Sicherheitsprobleme

- gar keine Authentifizierung
- Unsichere Übertragung von Credentials (HTTP)

  `gulas.ch/gulaschküche/food?user=entropia`
  `&password=gulasch`
  `gulas.ch/gulaschküche/food?api-key=famousapikey`
- One Token to Rule Them All – Authentifizierung ohne Autorisierung/Rechtemanagement

Mangelndes Rechtemanagement

- Beispiel: API mit Buchungsdaten auf Endpunkt `/bookings`, Einschränkungen über Query-Parameter mit Datum, Ort, ...
- API: alle Buchungen aller Personen/IDs in diesem Zeitraum



Mangelndes Rechtemanagement – weiter getrieben

- Query-Parameter oder Frontend-Validierung nicht ausreichend als Authentifizierung (auch nicht in Kombination mit Login)
- cURL auf Endpunkt ohne Parameter?
- Alles in Fakten und Zahlen:
  - Buchungsdaten mit Zeiten, Kosten, Notizen (ohne direkt personenbezogene Informationen)
  - `cat data.json | jq . | wc -l` 9.190.718
  - knapp 155 MB an Buchungsdaten

Nicht genutzte Endpunkte

- Zu implementierende/halb-fertige oder legacy Endpunkte
- Prinzipiell erreichbar, aber nicht genutzt
- ... mehr oder weniger freundliche Finder:innen werden sie nutzen

Auskunftsfreudige Endpunkte

- Gefahr der Information Disclosure und Einschränkung der Vertraulichkeit
- Hier: Funktion zur Berechnung von Kosten einer Buchung... auf Basis von Nutzer-IDs
- Rückgabe nicht nur von Kosten, sondern unter anderem auch von
  - Mail
  - Name
  - Adresse
  - SEPA-Lastschriftinformationen
  - Letzter Login

Mangelnde Kapselung von dahinter liegenden Systemen

- Potentiell Zugriff auf Daten anderer Systeme dahinter
- Beispiel: Andere ähnliche aufgebaute Anwendungen derselben Software
- Keine ausreichende Kapselung: Richtiger Query-Parameter mit Name des Kunden $\rightarrow$ Standorte mit Adresse beliebiger Kunden

Input bereinigen

Generell: Input Validation

- Trust no one
- Anfragen im Zweifelsfall von allen (externen) Entitäten möglich – sowohl nett als auch nicht nett
- Keine Datenbank direkt an die REST-API hängen, bitte
- Bibliotheken zur Validierung verwenden und regelmäßig updaten
- Trust no one ist ernst gemeint – insbesondere auch dem eigenen Code von vor drei Monaten nicht

Datenhungrige Anfragen

- Potentiell: Häufiges Abfragen Daten aus einer API
- Viele häufig Anfragen $\rightarrow$ Too Many Requests (Vorschlag der Implementierung, ungewolltes Too Many Requests bei fehlendem Rate Limit/Throttling)
- Ressourcen von (externen) Anfragen beschränken (hallo Cloud)

Happy Hacking

EOF

- Generell: REST coole Sache, intuitiv
- Rest zu REST: nicht schlimm, unintuitiver, eine Frage des Geschmacks
- bitte interessiert euch für Security (sowohl beim Entwickeln als auch beim Testen – und seid nicht überrascht, wenn ihr plötzlich näher an anderer Infrastruktur seid als geplant)
- Enjoy 🥘 🥣
- Fragen?