

# SYTD

Kommunikation im verteilten System





# Kommunikation im verteilten System

- **Microservices laufen als einzelner Prozess**
  - auf einem bestimmten Port
- **Microservices kommunizieren über Netzwerkinterfaces**
  - egal ob lokal („localhost“) oder entfernt → Zugriffstransparenz
  - erreichbar unter URL
    - <http://myservice.htlkrems.at:9000/students/?class=4AHIT>



# OSI-Schichtenmodell

7	Application Layer	Human-computer interaction layer, where applications can access the network services
6	Presentation Layer	Ensures that data is in a usable format and is where data encryption occurs
5	Session Layer	Maintains connections and is responsible for controlling ports and sessions
4	Transport Layer	Transmits data using transmission protocols including TCP and UDP
3	Network Layer	Decides which physical path the data will take
2	Data Link Layer	Defines the format of data on the network
1	Physical Layer	Transmits raw bit stream over the physical medium

<https://www.imperva.com/learn/application-security/osi-model/>

= „Open Systems Interconnection Model“

- Standard für Netzwerkkommunikation
- ABER: nur konzeptionelles Modell
  - stattdessen: TCP/IP-Modell

# TCP/IP-Schichtenmodell

OSI Model	TCP/IP Model
Application Layer	Application layer
Presentation Layer	
Session Layer	
Transport Layer	Transport Layer
Network Layer	Internet Layer
Data link layer	Link Layer
Physical layer	

= OSI-Schichtenmodell in der Praxis

- statt TCP auch UDP möglich

• keine separate Schicht für

- Presentation-Layer und
- Session-Layer

→ übernimmt Application-Layer

z.B. HTTPS

# „HTL-Schichtenmodell“

OSI Model	TCP/IP Model	
Application Layer	Application layer	SYTD
Presentation Layer		
Session Layer		
Transport Layer	Transport Layer	NWTK
Network Layer	Internet Layer	
Data link layer	Link Layer	
Physical layer		

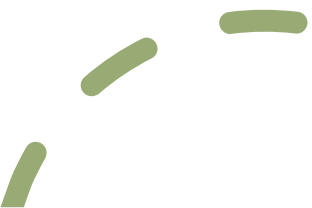
→ Annahme für SYTD:

- Funktionierendes Netzwerk
  - (W)LAN eingerichtet
  - IP-Adressen/Hostnamen
  - Ports frei (Firewall!)



# Transport Layer

- bietet Protokolle zur byteweisen Datenübertragung an
  - **UDP (User Datagram Protocol)**
    - wird von z.B. DNS, DHCP verwendet
  - **TCP (Transport Control Protocol)**
    - wird von z.B. HTTP, HTTPS, SSH verwendet
- außerdem: ICMP (Internet Control Message Protocol)
  - Übertragung von Status- und Fehlermeldungen
    - z.B. ping



# Transport Layer

- kennt nur Bytes, kann senden und empfangen
- **Protokoll = Vereinbarung, wie Daten gelesen werden**

+	Bits 0-15	Bits 16-31
0	Source port	Destination port
32	Length	Checksum
64	Data	

UDP Datagram

	Octet	0								1								2								3								
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
0	0	Source Port																Destination Port																
4	32	Sequence Number																																
8	64	Acknowledgment Number (If ACK Flag Set)																																
12	96	Data Offset				Reserved				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
...	...	...																																

TCP Segment Header



# UDP – User Datagram Protocol

- einfaches verbindungsloses Protokoll zur Byte-Übertragung
  - es muss keine Verbindung aufgebaut werden
- verschickt Datagramm (Data + Telegramm) = einzelne Nachricht mit fester Größe
  - Header: 8 byte
  - Maximale Datenlänge: 65 536 Byte
- **Vorteil:**
  - **Leichtgewichtige Nachrichten, weniger Overhead als TCP**
- **Nachteil: Keine Garantie, dass Nachrichten**
  - **beim Empfänger ankommen**
  - **in richtiger Reihenfolge ankommen**

+	Bits 0-15	Bits 16-31
0	Source port	Destination port
32	Length	Checksum
64	Data	







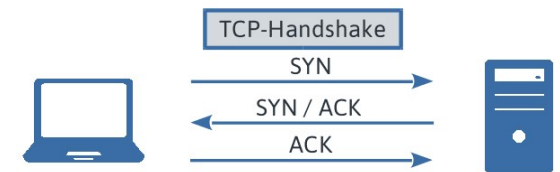
# **UDP – Live demo**

# TCP – Transport Control Protocol

= Verbindungsorientiertes Protokoll zur wechselseitigen Byte-Übertragung zwischen zwei Endpunkten

- **Verbindungsorientiert:**

- ausgehend vom Client wird ein Kommunikationskanal aufgebaut (3-Way-Handshake: SYN, SYN ACK, ACK RECEIVED)



- **Wechselseitig:**

- beide Teilnehmer können gleichzeitig senden und empfangen  
→ **vollduplex**



# TCP – Transport Control Protocol

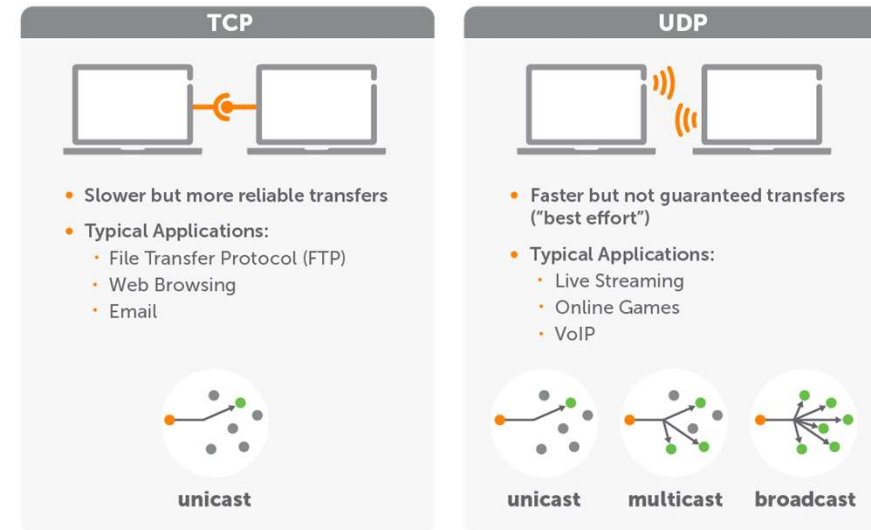
- **Verlässliche Übertragung:**
  - Pakete garantiert in richtiger Reihenfolge (Sequenznummer)
  - verlorene Pakete können erneut angefordert werden
- **Flusskontrolle**
  - Übertragung kann verlangsamt werden, falls Empfänger die Pakete nicht schnell genug verarbeiten kann
- ABER:
  - **Mehr Overhead** (= Zusatzinformation) als UDP



# Warum UDP verwenden?

- **schneller**
  - TCP muss zuerst Verbindung aufbauen
- unterstützt **Multicast/Broadcast**
  - TCP kann nur Punkt-zu-Punkt-Verbindung

→ trotzdem oft TCP ausreichend



TCP



UDP






# TCP in C#

- Namespace: `System.Net.Sockets`
- Client: verwendet **TcpClient**
- Server: verwendet **TcpListener**
  - Wartet auf eingehende Verbindungsanfragen
  - Wenn verbunden: liefert `TcpClient`
- Nach Verbindungsaufbau:
  - `TcpClients` kommunizieren mittels **NetworkStream**  
→ in beide Richtungen möglich





# Exkursion: C#-Streams (System.IO)


- Abstrakte Klasse zum Schreiben und Lesen von Bytes
    - CanRead, CanWrite
    - Fungiert als Puffer
      - z.B. Lesen von Files > RAM
  - Wichtigste Implementationen:
    - FileStream: aktuelle Position und Länge bekannt (= Dateigröße in Bytes)
    - **NetworkStream**: Länge unbekannt
  - Aufbauend auf Stream:
    - BinaryReader/BinaryWriter:
      - Primitive Datentypen lesen/schreiben (bool, int, double, ...)
    - **StreamReader/StreamWriter**:
      - Text lesen/schreiben (Kodierung beachten)
- 



# **TCP – Live demo**




# HTTP – Hypertext Transfer Protocol

- Application Layer Protokoll zur Übertragung von Hypertext und/oder sonstigen Dokumenten/Inhalten (z.B. XML-Dateien)
    - Hypertext = Dokumente, die Hyperlinks enthalten
    - Inhalt wird mittels **Media type** identifiziert
  - benutzt TCP
    - **ACHTUNG:** HTTP ist **verbindungslos**  
→ Applikation selbst für Sitzungsverwaltung zuständig (z.B. Cookies)
- 






# Media/MIME type

- MIME = Multipurpose Internet Mail Extensions
    - ursprünglich nur für Beschreibung von Mail-Inhalten verwendet
  - **Identifizierer für Dateiformate/-inhalte im Web**
    - Zweiteilig, getrennt durch Schrägstrich: ***type/subtype***
    - Type: größere Verwendungskategorie wie z.B. Text, Bild, Video, etc.
    - Subtype: genauere Beschreibung (abhängig von Type, oft wie Dateierweiterung)
  - Häufigste Vorkommen:
    - `text/html`, `text/xml`, `text/plain`
    - `image/jpg`, `image/png`
    - `application/json`, `application/octet`
- 




# HTTP Ablauf

1. Client baut TCP-Verbindung zum Server auf
    - Port muss bekannt sein (80, 443, 8080, etc.)
  2. Client stellt Anfrage → **HTTP Request**
  3. Server antwortet → **HTTP Response**
  4. optional: weitere Anfragen
    - z.B. CSS- und JS-Files, Bilder, Videos, usw.
  5. Verbindung wird getrennt
- 




# HTTP Format

- Sowohl Request als auch Response sind:
    - **textbasiert** (ASCII-codiert)
    - einzelne Segmente durch **Zeilenumbruch** getrennt (`\n`)
  - Gleiches Schema für Anfrage und Antwort:
    1. **Request-/Response-Zeile**
    2. **Header**
    3. **Body**
- 



# HTTP Request

besteht aus mehreren Zeilen:

- 1. Request-/Anfrage-Zeile** (genau 1x)
    - [Methode] [Pfad] [HTTP-Version], z.B. GET / HTTP/1.1
  - 2. Header** (0 oder mehrere Zeilen)
    - [Schlüssel]: [Wert], z.B. Content-Length: 0
    - **endet mit Leerzeile**
  - 3. Body** (0 oder mehrere Zeilen)
    - Format im Header definiert
    - **endet mit Leerzeile**
- 

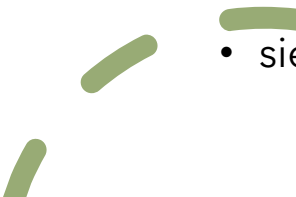


# Übung: Textbrowser

- C# Console application
- URL from Console.ReadLine
- Print server response




# HTTP Request Pfad

- Oberster Pfad: /
    - z.B. GET / HTTP/1.1
    - kann im Browser weggelassen werden
      - Z.B. [www.orf.at](http://www.orf.at) statt [www.orf.at/](http://www.orf.at/)
    - wird von den Webservern in eine „default page“ (meistens */index.html*) übersetzt
  - Kann entweder **Dateistruktur** des Servers reflektieren (Apache, nginx)
    - <http://www.example.org/myfolder/mypage.html>  
→ Odner *myfolder* mit Datei *mypage.html*
  - Oder **dynamisch verarbeitet** werden
    - **URI = virtueller Name → Ortstransparenz**  
*https://api.spotify.com/v1/playlists/37i9dQZF1DZ06ev005tE88/tracks*
    - siehe Web-Frameworks (ASP.NET Core, flask, Node.js, etc.)
- 




# HTTP Methods

= String zu Definition, **wie** mit den übertragenen Daten umgegangen werden soll

- sind Richtlinien (Eigenkreationen „theoretisch“ möglich)
  - Alle gängigen HTTP-Server-Frameworks unterstützen diese vier Methoden:
    - **GET**
    - **POST**
    - **PUT**
    - **DELETE**
- 



# HTTP GET

- Zum **Abfragen von Daten** → Webseiten, JSON-Files, etc.
    - Eingabe in die Browser-Adressleiste
    - Klick auf Hyperlink
    - Web-API-Zugriff
  - Daten werden **via URL** übertragen
    - Pfad identifiziert Resource (in Web-Frameworks „Route“ genannt)
    - Query String listet Parameter auf nach Fragezeichen
      - Format: `?name1=value1&name2=value2`
      - Illegale Zeichen müssen umgewandelt werden (z.B. Leerzeichen zu „%20“)
- 



# HTTP POST

- Hinzufügen von Daten
  - im Browser via HTML-Formular  
`<form method="POST" action="...">`
- Daten werden **im Body** übertragen
  - Format wird durch Content-Type bestimmt (application/json)

```
POST /foo HTTP/1.1
Content-Length: 68137
Content-Type: multipart/form-data; boundary=-----974767299852498929531610575

-----974767299852498929531610575
Content-Disposition: form-data; name="description"

some text
-----974767299852498929531610575
Content-Disposition: form-data; name="myFile"; filename="foo.txt"
Content-Type: text/plain

(content of the uploaded file foo.txt)
-----974767299852498929531610575--
```



# Sonstige HTTP Methoden

- **PUT**

- Daten aktualisieren
- gleicher Request mehrmals hintereinander liefert selbes Ergebnis

- **DELETE**

- Daten löschen (bzw. ausblenden)

- **OPTIONS**


- Abfrage, welche HTTP Methoden am Server erlaubt sind





# HTTP Request Header

- Wichtige Zeilen:
  - Host: Zielhost, zB für Unterscheidung von virtual hosts
  - **Accept:** Gewünschte MIME-types, üblicherweise
    - bei Browser-Anfragen → text/html
    - bei API-Anfragen → application/json
  - User-Agent: Informationen über Client-Browser und -Betriebssystem
    - zB wenn Android oder iOS → Umleitung auf mobile-optimierte Website



```
GET / HTTP/3
Host: www.google.at
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:107.0) Gecko/20100101 Firefox/107.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
...
```



# HTTP Response

Besteht aus:

## 1. Statuszeile

- HTTP/[HTTP-Version] [Status-Code] [Nachricht]

## 2. Header

- selbes Format wie HTTP Request
- endet mit Leerzeile

## 3. Body:

- beinhaltet die eigentlichen Daten
- endet mit Leerzeile



# HTTP Response Beispiel

**HTTP/1.1 200 OK**

Date: Mon, 09 Jan 2023 20:38:49 GMT

Server: Apache

...

Content-Type: text/html; charset=utf-8

Content-Length: 26492

...

Connection: close

<!DOCTYPE html>

<html lang="de" dir="ltr">

<head>

  <meta charset="utf-8" />

  <meta name="viewport" content="width=device-width, initial-scale=1">

...



# HTTP Status Codes

- **1xx**
  - Information
- **2xx**
  - Erfolg
- **3xx**
  - Weiterleitung
- **4xx**
  - Fehler in der Anfrage („*Client ist schuld*“)
- **5xx**
  - Server-Fehler („*Server ist schuld*“)



[https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)





# HTTP Response Header

- Wichtige Zeilen:
  - **Content-Type**
    - beschreibt Body-Format
  - **Content-Length**
    - Body-Länge in Bytes
  - **Set-Cookie**
    - Anweisung an Client, ein Cookie anzulegen





# Übung: Webserver „nginy“

- C# Console application
- Wait for HTTP GET requests
- Respond with HTML-Content



# JSON

= JavaScript Object Notation

- Ursprung in JavaScript, aber von allen gängigen Sprachen unterstützt
- offener Standard (<https://www.rfc-editor.org/info/std90>)

- **textbasiertes Format zum Austausch von Daten**

- leichtgewichtig
- von Menschen lesbar
- MIME-Typ: *application/json*

- Verwendung:

- Konfigurationsdateien
- Kommunikation zwischen Webservices
- Datenbank: MongoDB (via BSON = Binary JSON) → „NoSQL-DB“



# Vorteile JSON vs. XML

- beides zur Serialisierung von Objekten verwendet
- **weniger Zeichen benötigt**
  - XML: redundante Tags für Anfang und Ende jedes Elements  
→ weniger Overhead
- **einfachere Struktur**
  - keine Attribute
  - keine Namespaces  
→ einfacher zu lesen



# JSON Struktur

- JSON ist als **Baum** strukturiert
  - Wurzel/Root: **Object** oder **Array**
- Einrückung ist optional
- **Elemente** (durch Beistrich getrennt)
  - String
  - Number
  - Object { ... }
    - enthält 0...\* benannte Elemente
  - Collection
    - Enthält 0...\* Elemente
- Beispiele: <https://www.json.org/example.html>

```
1  {
2    "country": "string",
3    "display_name": "string",
4    "email": "string",
5    "explicit_content": {
6      "filter_enabled": true,
7      "filter_locked": true
8    },
9    "external_urls": {
10     "spotify": "string"
11   },
12   "followers": {
13     "href": "string",
14     "total": 0
15   },
16   "href": "string",
17   "id": "string",
18   "images": [
19     {
20       "url": "https://i.scdn.co/image/ab67616d00001e02ff9ca10b",
21       "height": 300,
22       "width": 300
23     }
24   ],
25   "product": "string",
26   "type": "string",
27   "uri": "string"
28 }
```



# JSON Format

- **String**
  - mit doppelten Anführungszeichen oder *null*
- **Number**
  - Ganz- oder Kommazahl, wissenschaftliche Notation
  - **nicht** erlaubt: *null*, *NaN*
- **Object**
  - *null* erlaubt
- **Collection**
  - *null* erlaubt



# JSON Object

- bezeichnet durch geschwungene Klammern { }
  - enthält **name-value-pairs**
    - Name in doppelten Anführungszeichen
      - muss einzigartig sein
- entspricht Properties in C#

```
1 {  
2   "url": "https://i.scdn.co/image/ab67616d00001e02ff9ca10b55ce82ae553c8228",  
3   "height": 300,  
4   "width": 300  
5 }
```



# JSON Array

- bezeichnet durch eckige Klammern [ ]
- Inhalt:
  - 0 oder mehrere Elemente
  - gemischte Typen möglich
- z.B.:

```
[ 1, 2, „text“, [ 3, 0 ], { name: „test“ } ]
```



# JSON Schema

- zur Validierung von JSON Daten
  - ähnlich XSD

```
{
  "productId": 1,
  "productName": "A green door",
  "price": 12.50,
  "tags": [ "home", "green" ]
}
```

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "productName": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    }
  },
  "required": [ "productId", "productName", "price" ]
}
```



# Definition API

## = **Application Programming Interface**

- Satz von Befehlen, Funktionen, Protokollen und Objekten
- APIs ermöglichen Kommunikation zwischen Anwendungen
- Achtung:
  - API Definition != API Implementierung
- Web-API = API, die über Internet erreichbar ist







# REST – Representational State Transfer

= eine **Sammlung von Regeln** für Kommunikation zwischen Client und Server (**kein** Protokoll/Standard/Spezifikation)

- häufigste Art von Web-APIs (derzeit)
    - andere Möglichkeiten: SOAP, gRPC, GraphQL
  - Systeme, die sich an REST-Prinzipien halten:  
**„RESTful“ systems**
- 



# Definition REST

- **REpresentational**

- Verwendung von offenen Dateiformaten, zB HTML, XML, JSON
- tatsächliche Daten werden nur repräsentiert
  - zB als JSON, aus Server-Datenbank gelesen (Spalten/Relationen ausgeblendet)

- **State**


- der aktuelle Zustand eines Systems (= Webapplikation/-service)

- **Transfer**

- Übertragung und Kommunikation mittels HTTP
  - bidirektional → vom Client (initiiert) zum Server
- 

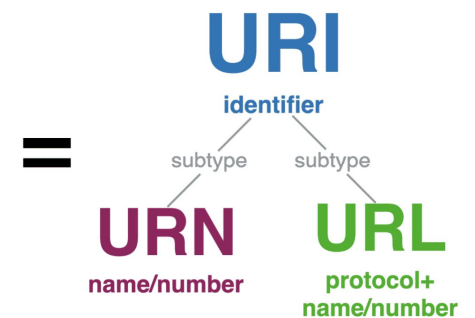
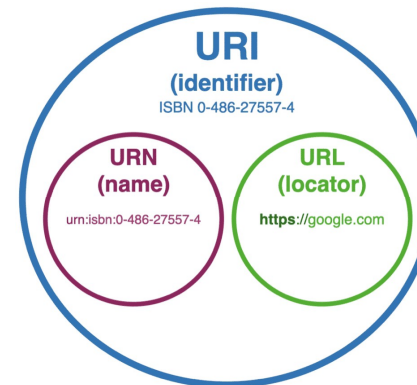


# REST Regeln

- **zustandslose Client-Server-Kommunikation über HTTP**
    - Verbindung wird nicht aufrecht erhalten (nur Request & Response)
    - einzelne Anfragen werden separat behandelt
    - (Authentifizierung mittels Cookies oder Access Tokens)
  - **Verwendung einer einheitlichen Schnittstelle (endpoint)**
    - z.B. `example.com/api/v3/`
  - **(Server-)Ressourcen werden mittels URI eindeutig identifiziert**
    - z.B. `example.com/api/v3/product/13`
- 

# Exkursion: URL vs URI (vs URN)

- **URI** = Uniform Resource Identifier
  - String zur eindeutigen Identifikation von Ressourcen
  - z.B. `mylibrary.com/api/book/147`
- **URL** = Uniform Resource Locator
  - Subtyp von URI
  - beschreibt zusätzlich den Zugriff auf Ressource → enthält Protokoll
  - z.B. `https://mylibrary.com/api/book/147`
- **URN** = Uniform Resource Name
  - Subtyp von URI
  - bestimmtes Schema
  - z.B. `urn:isbn:0451450523`






# REST Verbs

- HTTP Methoden werden im Request wie (sprachliche) Verben verwendet:
    - **POST („lege an“)** → **Create**
      - Hinzufügen von Ressourcen
    - **GET („gib mir“)** → **Read**
      - Abfrage von Ressourcen (auch Webseiten)
    - **PUT („ändere“)** → **Update**
      - aktualisieren von Datensätzen
    - **DELETE („lösche“)** → **Delete**
- 



# REST APIs

- viele bekannte Dienst bieten REST APIs an
    - frei zugreifbar oder mit Authentifizierung
  - Beispiele:
    - <https://developers.google.com/calendar/api/guides/overview>
    - <https://developer.spotify.com/documentation/web-api/>
    - <https://steamcommunity.com/dev>
    - <http://data.wien.gv.at/pdf/wienerlinien-echtzeitdaten-dokumentation.pdf>
- 




# REST Clients

- werden zum Testen von REST APIs verwendet
  - **jeder HTTP Client kann als REST Client verwendet werden**
  - z.B.
    - Browser
      - relevante Details in Web Developer Tools
    - Swagger UI
      - wird von REST API selbst erstellt
    - Postman
      - standalone API Tool, enthält auch HTTP Client
    - uvm.
- 




# REST-Frameworks

- oft Teil von Web-Frameworks
    - unterstützen weitere Technologien, wie zB SignalR, gRPC, etc.
  - REST-Frameworks existieren in allen gängigen höheren Programmiersprachen, zB
    - Spring Boot (Java)
    - ASP.NET Core (C#)
    - flask (python)
    - Node.js (JavaScript)
- 





# REST-Architektur

- REST-Frameworks funktionieren oft nach dem selben Prinzip:
  - **Webservice** (läuft als **Prozess**)
    - hat mehrere **Controller** (= Endpoints)
      - haben mehrere **Actions**
    - bei eingehendem HTTP-Request:
      - **Routing** entscheidet, welcher Controller und welche Action
      - Gewählte Action **entscheidet selbst über HTTP-Response**
- 

# Controller

- **Logische Einheit zur Verwaltung einer Ressource**
- erreichbar mittels bestimmter **Route** (= relative URL)
- enthält (miteinander verwandte) **Actions**
  - betreffen (meistens) die selbe Ressource
  - unterscheiden sich durch URL und/oder HTTP-Methode

→ **im Code als Klasse implementiert**

## ENDPOINTS

Albums	>
Artists	>
Shows	>
Episodes	>
Audiobooks	>
Chapters	>
Tracks	>
Search	>
Users	>
Playlists	>
Categories	>
Genres	>
Player	>
Markets	>

# Actions

- stellen **Anweisungen** an das Service dar
- **mehrere Actions** können **unter gleicher URL erreichbar** sein
  - Unterscheidung durch HTTP-Methode und Parameter (ähnlich Überladung in C#)
- z.B. bei Web-Services:
  - meistens CRUD-Operationen
- z.B. bei Web-Applikationen:
  - meistens Generierung von HTML-Seiten

→ **im Code als Methoden implementiert**

## ENDPOINTS

### Albums

Get Album	GET
Get Several Albums	GET
Get Album Tracks	GET
Get User's Saved Albums	GET
Save Albums for Current User	PUT
Remove Users' Saved Albums	DELETE
Check User's Saved Albums	GET
Get New Releases	GET

### Artists

Get Artist	GET
Get Several Artists	GET
Get Artist's Albums	GET
Get Artist's Top Tracks	GET
Get Artist's Related Artists	GET



# Actions – URL Parameter

- auch „*Query Parameter*“ genannt
- **Format:**
  - Trennung vom vorhergehenden Pfad mittels **?**
  - Trennung untereinander mittels **&**
  - **{Name}={Wert}**
    - Name kann mehrfach vorkommen, z.B. bei Array
- Beispiel:
  - <https://example.com/api/item?name1=value1&name2=value2&...>
- **Sonderzeichen in URL-Parametern müssen kodiert werden**  
z.B. Leerzeichen zu +, / zu %2F






# Actions – Header Parameter

- siehe Folie „HTTP Request Header“
- **Format:** {name}: {value}
  - z.B.  
`Cookie: mycookie1=123; mycookie2=456;`
- außer zur Authentifizierung eher weniger verwendet
  - enthält meistens nur Metadaten
  - „Kollisionsgefahr“ mit bestehenden Headern (z.B. Content-Type)






# Actions – Body Parameter

- Format durch Content-Type im Header bestimmt, üblicherweise
    - bei Webservices `application/json`
    - bei Webapplikationen: `multipart/form-data`
- 



# Beispiel für Controller & Actions

- Spotify-API-URL
    - <https://api.spotify.com/v1>
  - Playlist-Controller mit Route `/playlists`:
    - GET <https://api.spotify.com/v1/playlists/7/tracks>  
→ listet Songs der Playlist mit ID 7 auf
    - POST <https://api.spotify.com/v1/playlists/7/tracks>  
→ fügt Songs in die Playlist mit ID 7 ein (erwartet Daten im Body)
- 



# **Live demo: Nachbau der Spotify-API**