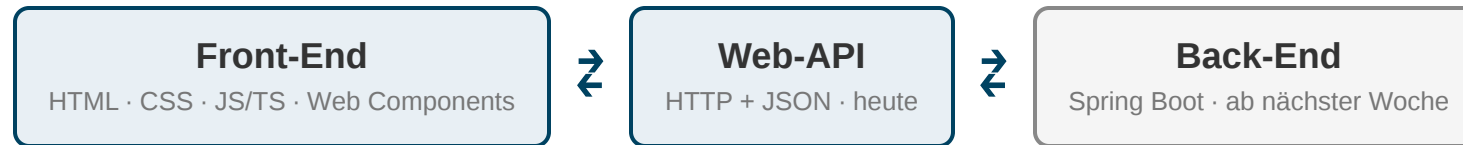


REST und Web-APIs

HTTP, Ressourcen, Statuscodes und JSON, ohne Framework

Informatik – 3. Semester



- Das Front-End kennt ihr: Seiten bauen, gestalten, per `fetch()` Daten nachladen (web-05)
- Das Back-End kommt in den nächsten zwei Wochen
- Dazwischen liegt eine Schnittstelle, über die beide Seiten reden: die Web-API

Die API ist unabhängig davon, womit Front-End und Back-End gebaut sind. Deshalb heute ganz ohne Framework.

API (Application Programming Interface)

Eine festgelegte Schnittstelle, über die ein Programm Daten oder Funktionen eines anderen Programms nutzt. Eine Web-API ist eine solche Schnittstelle, die über HTTP im Netz erreichbar ist.

Server liefert fertiges HTML

- Browser fordert eine Seite an
- Server baut das HTML zusammen
- Jeder Klick lädt eine neue Seite

Server liefert Daten (JSON)

- JavaScript fragt per `fetch()` Daten an
- Server antwortet mit JSON
- Das Front-End rendert selbst, auch Apps und andere Server können die API nutzen

1 HTTP-Grundlagen

3 HTTP-Methoden

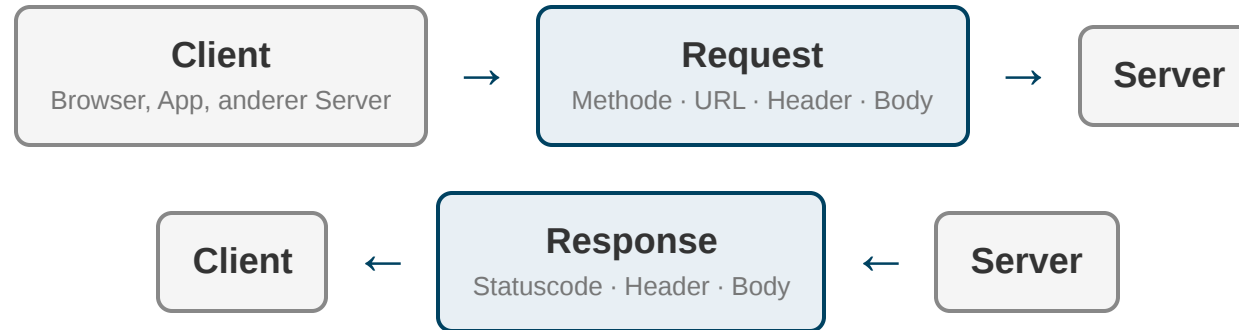
5 JSON & Fehlerbehandlung

2 Ressourcen & URLs

4 Statuscodes

6 Zusammenfassung

Was eigentlich über die Leitung geht.



- Der Client fragt, der Server antwortet: jede Anfrage bekommt genau eine Antwort
- **Zustandslos:** Der Server merkt sich zwischen zwei Requests nichts

Folge der Zustandslosigkeit: Jeder Request muss alles mitbringen, was der Server braucht, auch den Nachweis, wer fragt.

```
POST /api/teams/3/tasks HTTP/1.1
Host: taskboard.example.org
Content-Type: application/json
Accept: application/json

{
  "title": "Wireframes abstimmen",
  "dueDate": "2026-12-04"
}
```

- **Startzeile:** Methode, Pfad, HTTP-Version
- **Header:** Metadaten als `Name: Wert`
- **Leerzeile:** trennt Header und Body
- **Body:** die eigentlichen Daten, optional

HTTP/1.1 ist reiner Text. Was ihr hier seht, geht genau so über die Leitung.

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/tasks/17

{
  "id": 17,
  "title": "Wireframes abstimmen",
  "dueDate": "2026-12-04",
  "status": "OPEN",
  "teamId": 3
}
```

- **Statuszeile:** Version, Statuscode, Kurztext
- **Header:** hier z.B. die URL der neuen Task
- **Body:** die angelegte Task, jetzt mit ID und Status

Der Statuscode ist die wichtigste Zeile: Er sagt dem Client, was passiert ist, bevor er den Body überhaupt liest.

Header	Richtung	Bedeutung	Beispiel
Content-Type	beide	Format des Bodys	application/json
Accept	Request	gewünschtes Format der Antwort	application/json
Authorization	Request	Nachweis, wer fragt	Bearer eyJhbGciOi...
Location	Response	URL einer neu angelegten Ressource	/api/tasks/17
Cache-Control	Response	ob und wie lange zwischengespeichert werden darf	max-age=60

Selbst nachsehen: DevTools öffnen, Tab „Network“, einen Request anklicken, Reiter „Headers“. Dort seht ihr jeden Request, den eine Seite abschickt.

1 HTTP-Grundlagen

3 HTTP-Methoden

5 JSON & Fehlerbehandlung

2 **Ressourcen & URLs**

4 Statuscodes

6 Zusammenfassung

Die API denkt in Dingen, nicht in Aktionen.

REST (Representational State Transfer)

Ein Architekturstil für verteilte Systeme, beschrieben von Roy Fielding in seiner Dissertation (2000). Kein Protokoll, kein Standard, keine Bibliothek, sondern eine Sammlung von Entwurfsregeln.

- **Client-Server:** klare Rollentrennung
- **Zustandslos:** jeder Request steht für sich
- **Cachebar:** Antworten sagen, ob man sie zwischenspeichern darf
- **Einheitliche Schnittstelle:** Ressourcen mit URLs, Standardmethoden, Repräsentationen
- **Schichten:** Proxies und Caches dürfen dazwischen liegen

Für eure Projekte zählt vor allem die einheitliche Schnittstelle. Um sie geht es im Rest der Vorlesung.

So nicht: Aktionen in der URL

```
GET /api/getTask?id=17
POST /api/createTask
POST /api/deleteTask?id=17
GET /api/teams/3/delete
```

So: Ressourcen und Methoden

```
GET /api/tasks/17
POST /api/teams/3/tasks
DELETE /api/tasks/17
DELETE /api/teams/3
```

Die URL benennt das Ding (Substantiv). Die HTTP-Methode sagt, was damit passieren soll (Verb).

Pfad: welche Ressource?

- Substantive im Plural: `tasks`, `teams`
- Sammlung `/api/tasks`, Element `/api/tasks/17`
- Beziehung eine Ebene tief: `/api/teams/3/tasks`
- Kleinbuchstaben, Bindestrich: `/api/team-members`
- Keine Verben, keine Dateiendungen

Query: welche Auswahl davon?

```
GET /api/tasks?status=OPEN
GET /api/teams/3/tasks?sort=dueDate
GET /api/tasks?dueBefore=2026-12-18
GET /api/tasks?page=2&size=20
```

Filtern, Sortieren und Seitenweise-Laden gehören in Query-Parameter, nicht in den Pfad.

Tiefer als eine Ebene wird es unübersichtlich: besser `/api/tasks/17` als `/api/teams/3/tasks/17`.

1 HTTP-Grundlagen

3 HTTP-Methoden

5 JSON & Fehlerbehandlung

2 Ressourcen & URLs

4 Statuscodes

6 Zusammenfassung

Fünf Verben reichen für fast alles.

Methode	Zweck	Beispiel	Request-Body
GET	lesen	GET /api/tasks/17	nein
POST	neu anlegen, Server vergibt die ID	POST /api/teams/3/tasks	ja
PUT	vollständig ersetzen	PUT /api/tasks/17	ja
PATCH	teilweise ändern	PATCH /api/tasks/17	ja
DELETE	löschen	DELETE /api/tasks/17	nein

CRUD auf HTTP: **C**reate = POST, **R**ead = GET, **U**ppdate = PUT oder PATCH, **D**eleete = DELETE.

- Daneben gibt es u.a. HEAD (nur Header) und OPTIONS (was ist erlaubt?), die euch bei CORS begegnen

Sicher

Der Request ändert auf dem Server nichts. Er darf beliebig oft gesendet, vorab geladen und zwischengespeichert werden.

Idempotent

Mehrfaches Senden hat dieselbe Wirkung wie einmaliges Senden. Geht die Antwort verloren, darf der Client es gefahrlos wiederholen.

Methode	sicher	idempotent
GET	ja	ja
PUT	nein	ja
DELETE	nein	ja
POST	nein	nein
PATCH	nein	nicht garantiert

Idempotent heißt gleiche Wirkung, nicht gleiche Antwort: Das zweite `DELETE /api/tasks/17` liefert 404, die Task ist trotzdem genau einmal weg.

PUT ersetzt die ganze Ressource

```
PUT /api/tasks/17 HTTP/1.1
Content-Type: application/json

{
  "title": "Wireframes abstimmen",
  "dueDate": "2026-12-04",
  "status": "IN_PROGRESS"
}
```

Was fehlt, ist danach weg.

PATCH ändert nur, was mitkommt

```
PATCH /api/tasks/17 HTTP/1.1
Content-Type: application/json

{
  "status": "IN_PROGRESS"
}
```

Alle anderen Felder bleiben, wie sie sind.

Methode	URL	Bedeutung	Erfolg
GET	/api/teams	alle Teams	200
GET	/api/teams/3	ein Team lesen	200
POST	/api/teams	Team anlegen	201 + Location
GET	/api/teams/3/tasks?status=OPEN	offene Tasks eines Teams	200
POST	/api/teams/3/tasks	Task anlegen	201 + Location
GET	/api/tasks/17	eine Task lesen	200
PUT	/api/tasks/17	Task vollständig ersetzen	200
PATCH	/api/tasks/17	nur den Status ändern	200
DELETE	/api/tasks/17	Task löschen	204

Neun Endpunkte, zwei Ressourcen, keine einzige erfundene Aktion.

Eine App für Hochschul-Veranstaltungen. Welche Methode und welche URL braucht ihr für:

1. alle Veranstaltungen im Dezember anzeigen
2. eine neue Veranstaltung anlegen
3. die Details einer Veranstaltung abrufen
4. sich zu einer Veranstaltung anmelden
5. nur den Raum einer Veranstaltung ändern
6. die eigene Anmeldung stornieren

5 Min

zu zweit

1 HTTP-Grundlagen

3 HTTP-Methoden

5 JSON & Fehlerbehandlung

2 Ressourcen & URLs

4 **Statuscodes**

6 Zusammenfassung

Drei Ziffern, die sagen, was passiert ist.

Erfolg

2xx

Hat geklappt.

200, 201, 204

Umleitung

3xx

Schau woanders nach.

301, 304

Client-Fehler

4xx

Deine Anfrage ist falsch.

400, 401, 403, 404, 409, 422

Server-Fehler

5xx

Ich habe ein Problem.

500, 503

Die erste Ziffer sagt, wer handeln muss. Bei 4xx muss der Client die Anfrage ändern. Bei 5xx hilft höchstens, es später noch einmal zu versuchen.

- 1xx (Zwischenmeldungen) sieht man in APIs praktisch nie

Code	Name	Wann?
200	OK	Erfolg mit Antwort-Body, z.B. GET
201	Created	Ressource angelegt, URL im Location -Header
204	No Content	Erfolg ohne Body, z.B. DELETE
400	Bad Request	Anfrage kaputt: kein gültiges JSON, falscher Datentyp
401	Unauthorized	nicht angemeldet: Wer bist du?
403	Forbidden	angemeldet, aber nicht berechtigt
404	Not Found	Ressource gibt es nicht
409	Conflict	widerspricht dem aktuellen Zustand: Titel im Team schon vergeben
422	Unprocessable Content	formal korrekt, fachlich ungültig: Fälligkeit liegt in der Vergangenheit
500	Internal Server Error	unerwarteter Fehler auf dem Server

1. GET /api/tasks/999 , diese Task gibt es nicht
2. POST /api/teams/3/tasks , das Team hat schon eine Task mit diesem Titel
3. POST /api/teams/3/tasks , der Body ist kein gültiges JSON
4. DELETE /api/tasks/17 , erfolgreich gelöscht
5. Eine Studentin will ein fremdes Team löschen, darf das aber nicht
6. Beim Speichern tritt auf dem Server eine unerwartete Exception auf
7. POST /api/teams , das Team wurde angelegt

3 Min

Handzeichen

- 1 HTTP-Grundlagen
- 2 Ressourcen & URLs
- 3 HTTP-Methoden
- 4 Statuscodes
- 5 JSON & Fehlerbehandlung**
- 6 Zusammenfassung

Daten im Body, Fehler mit Ansage.

```
{  
  "id": 17,  
  "title": "Wireframes abstimmen",  
  "dueDate": "2026-12-04",  
  "status": "OPEN",  
  "teamId": 3,  
  "archived": false,  
  "labels": ["Entwurf", "UI"],  
  "description": null  
}
```

- **Ressource:** die Task selbst, egal wie gespeichert
- **Repräsentation:** eine Darstellung davon, hier JSON
- **Content-Type:** `application/json` sagt, was im Body steht
- **Konventionen:** Feldnamen einheitlich in camelCase, Datum als ISO-8601-String, Listen als Array

JSON kennt nur String, Zahl, Boolean, null, Array und Objekt. Ein Datum gibt es nicht, deshalb schickt man es als String im ISO-Format.

So nicht: `200 OK` mit `{ "success": false, "error": "Titel vergeben" }`. Jeder Client, jeder Cache und jedes Monitoring hält das für einen Erfolg.

```
HTTP/1.1 409 Conflict
Content-Type: application/problem+json

{
  "type": "/problems/duplicate-title",
  "title": "Titel bereits vergeben",
  "status": 409,
  "detail": "Team 3 hat diesen Titel schon.",
  "instance": "/api/teams/3/tasks"
}
```

Problem Details (RFC 9457): ein Standardformat für Fehler-Bodys

- **type:** Fehlerart als URI
- **title:** kurz, für Menschen
- **status:** wiederholt den Code
- **detail:** dieser konkrete Fall

```
const response = await fetch("/api/teams/3/tasks", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
  body: JSON.stringify({
    title: "Wireframes abstimmen",
    dueDate: "2026-12-04",
  }),
});
const task: Task = await response.json();
```

- **method:** Standard wäre GET
- **headers:** Body ist JSON
- **body:** ein String, also JSON.stringify
- **json():** liest den Body, auch asynchron

```
try {
  const res = await fetch("/api/teams/3/tasks", options);
  if (res.ok) {
    renderTask(await res.json());
  } else if (res.status === 409) {
    const problem = await res.json();
    showError(problem.detail);
  } else {
    showError(`Unerwarteter Fehler (${res.status})`);
  }
} catch {
  showError("Server nicht erreichbar");
}
```

fetch wirft nur bei Netzwerkfehlern. Ein 404 oder 500 ist für fetch eine erfolgreich empfangene Antwort.

- `res.ok` ist `true` bei 200 bis 299
- Erwartete Fehler gezielt behandeln
- `catch` fängt nur: Server weg, kein Netz

Was festgelegt sein muss

- Methode und URL jedes Endpunkts
- Aufbau von Request- und Response-Body
- Statuscodes für Erfolg und für Fehler
- Fehlerformat, z.B. Problem Details

Wie man ihn festhält

- Eine Tabelle wie die Task-API reicht für den Anfang
- Beispiel-JSON zu jedem Endpunkt dazuschreiben
- **OpenAPI:** verbreiteter Standard für maschinenlesbare API-Beschreibungen

Tipp für die Umsetzung: Vereinbart die Endpunkte zuerst. Dann kann das Front-End mit Beispiel-JSON arbeiten, bevor das Back-End steht, und später nur die URL tauschen.

- 1 HTTP-Grundlagen
- 3 HTTP-Methoden
- 5 JSON & Fehlerbehandlung
- 2 Ressourcen & URLs
- 4 Statuscodes
- 6 Zusammenfassung**

Was ihr aus heute mitnehmt.

- **HTTP:** Request aus Methode, URL, Header, Body; Response aus Statuscode, Header, Body; zustandslos
- **Ressourcen:** Substantive im Plural, Sammlung `/api/tasks` und Element `/api/tasks/17`, Auswahl per Query
- **Methoden:** GET lesen, POST anlegen, PUT ersetzen, PATCH ändern, DELETE löschen
- **Eigenschaften:** GET ist sicher, GET, PUT und DELETE sind idempotent, POST nicht
- **Statuscodes:** 2xx Erfolg, 4xx Fehler des Clients, 5xx Fehler des Servers
- **JSON:** `Content-Type` setzen, Konventionen einheitlich halten
- **Fehler:** Statuscode plus Problem Details, im Client immer `response.ok` prüfen

REST ist kein Framework. Alles von heute gilt unverändert, egal ob das Back-End in Java, TypeScript oder etwas anderem geschrieben ist.

Die nächsten zwei Wochen

- **Spring Boot, Teil 1:** das Framework für die Serverseite
- **Teil 2:** Jede Zeile einer API-Tabelle wird zu einer Methode im Controller

Praktikum und Klausur

- Nach dem Projektwechsel: API des übernommenen Projekts als Tabelle festhalten
- REST kommt in der Klausur vor: Endpunkte entwerfen, Statuscodes wählen, Methoden einordnen

Zum Üben: Nehmt drei Anforderungen aus dem Katalog eures neuen Projekts und schreibt die passenden Endpunkte auf.

Fragen?