

RUHR-UNIVERSITÄT BOCHUM

WEB-ENGINEERING

Sommersemester 2023



Informatik
im Bauwesen

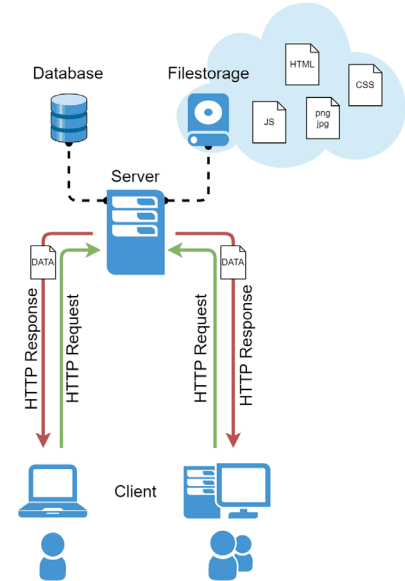
Webanwendung

Mit JavaScript alleine können schon umfangreiche Anwendungen geschrieben werden.

- Die Anwendung ist limitiert auf die Ressourcen des Benutzers
- Der gesamte Quellcode wird übermittelt und offengelegt

Durch die Zusammenarbeit von Client und Server wird Web-Development erst so richtig interessant!

- Der Server kann rechenintensive Aufgaben abnehmen und den Client entlasten
- Persistente Datenhaltung möglich (Datenbanken)
- Der Server kann in unterschiedlichen Sprachen geschrieben werden (Java, Python, C#, usw.) und so die jeweiligen Vorteile nutzen



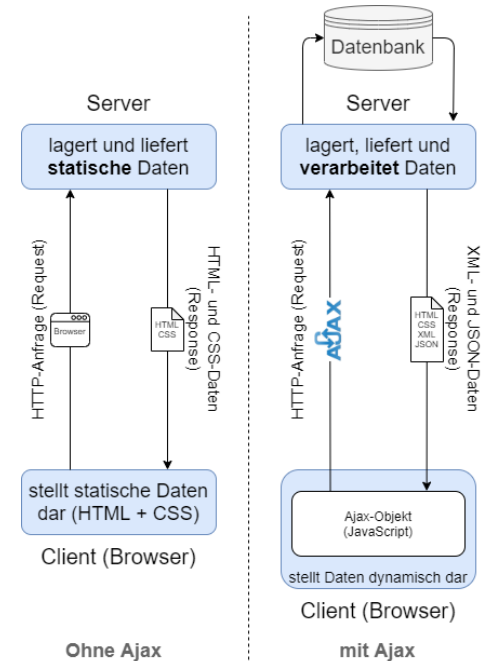
Web-Engineering

AJAX

AJAX

Ajax (**Asynchronous JavaScript and XML**) ist ein Konzept zur asynchronen Datenübertragung zwischen einem Browser und dem Server.

- Ermöglicht es HTTP-Anfragen (Request) durchzuführen, während eine HTML-Seite angezeigt wird
 - Liefer beliebige Datensätze **als XML und JSON** zur client-seitigen Verarbeitung (Response)
 - Realisiert dynamischen Datenaustausch
 - Seite verändern, ohne komplettes neuladen
- Ajax Kommunikation durchzuführen
 - Klassisch: XMLHttpRequest, jQuery.ajax
 - Modern: Fetch-API



AJAX

Eine klassische Ajax-Anfrage kann in vier Schritte unterteilt werden:

1. Ajax-Objekt erzeugen

- Per JavaScript muss ein Objekt generiert werden, das die HTTP-Anfrage vornimmt und die Daten entgegennimmt

```
var xhr = new XMLHttpRequest();
```

2. Verbindung zum Server definieren

- Hierfür wird die `open`-Methode verwendet, welche drei Parameter besitzt

```
xhr.open(HTTP-Methode, Zieladresse, Request-Methode);
```

↑
GET: Reine Daten-Anfrage an den Server.
POST: Daten-Anfrage an den Server, kann aber auch Daten senden. (langsamer)

↑
Adresse der Serverschnittstelle, welche die Anfrage verarbeiten soll.

↑
false: Scriptausführung wird angehalten, bis der Server die Daten zurückliefert (synchron).
true: Scriptausführung läuft weiter; HTTP Anfrage wird im Hintergrund ausgeführt (asynchron).

AJAX

Eine klassische Ajax-Anfrage kann in vier Schritte unterteilt werden:

3. Schrittweise Verarbeitung der erwarteten Antwort definieren

接收

- Die Antwort (oder Fehlermeldung) des Servers entgegennehmen

```
xhr.onreadystatechange = function() {  
  if (this.readyState == 4 && this.status == 200) {  
    console.log(this.responseText);  
  }  
};
```

`readyState` gibt den aktuellen Zustand der Anfrage wieder.

- 0: Anfrage nicht initialisiert
- 1: Serververbindung hergestellt
- 2: Anfrage wurde erhalten
- 3: Anfrage wird bearbeitet
- 4: Anfrage bearbeitet und Antwort bereit

`status` gibt eine Meldung über die Anfrage als Zahlencode zurück.

- 200: OK
- 403: forbidden
- 404: page not found

`responseText` die zurückgegebene Antwort des Servers als Textnachricht.

AJAX

Eine klassische Ajax-Anfrage kann in vier Schritte unterteilt werden:

4. Daten Anfrage mit der send-Methode an den Server abgeschickt

```
xhr.send();
```

Neue und moderne Variante mit der Fetch-API.

- Die alte Schreibkonventionen werden zunehmend durch die Fetch-API abgelöst
- Synchrones/Asynchrones Verhalten der Fetch-API wird durch `async/await` gelöst
- Wichtige Unterschiede zu älteren Methoden (bsp. `jQuery.ajax`)
 - Zurückgegebene Fehlermeldungen (404, 500) werden nicht zurückgewiesen
 - Cookies werden nicht automatisch mitgesendet

AJAX

```
fetch('https://example.com/projects', {  
  method: 'post',  
  headers: {  
    'Accept': 'application/json',  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({  
    title: 'Beispielprojekt',  
    description: 'example',  
  })  
}).then(function(response) {  
  console.log(response);  
}).catch(function(error) {  
  console.error(error);  
});
```

← **Optional:** Upload von JSON-Daten
an den angefragten Server.

AJAX

Fetch Beispiel


- Im folgenden Beispiel wird mittels Fetch-API ein **asynchroner** HTTP-Request gesendet
- **Bei den Angefragten Daten** handelt es sich um **JSON-Objekte**
- **Die Entgegennahme der Abfrage der Zielseite erfolgt über die REST-API**
- Anschließend wird der zurückgegebene Inhalt dynamisch nachgeladen

Get users

- ID: 1
- Name: Leanne Graham
- Benutzername: Bret
- Stadt: Gwenborough
- ID: 2
- Name: Ervin Howell
- Benutzername: Antonette
- Stadt: Wisokyburgh
- ID: 3
- Name: Clementine Bauch
- Benutzername: Samantha
- Stadt: McKenziehaven
- ID: 4
- Name: Patricia Lebsack
- Benutzername: Karianne
- Stadt: South Elvis

AJAX

Asynchrone Datenabfrage durch
async & await in Kombination
mit der `Fetch-API`.



```
<body>
  <button id="button">Get users</button>
  <div id="content"></div>
  <script>
    const getUsers = async() => {
      let res = await fetch('https://jsonplaceholder.typicode.com/users');
      let data = await res.json();
      var output = '';
      for (var i in data) {
        output += '<ul>' +
          '<li>ID: ' + data[i].id + '</li>' +
          '<li>Name: ' + data[i].name + '</li>' +
          '<li>Benutzername: ' + data[i].username + '</li>' +
          '<li>Stadt: ' + data[i]['address']['city'] + '</li>' + '</ul>';
      }
      document.getElementById('content').innerHTML = output;
    };
    document.getElementById('button').addEventListener('click', getUsers);
  </script>
</body>
```

Web-Engineering

REST

REST

Representational state transfer API

- HTTP ist die Sprache des World-Wide-Web und REST ein Architekturansatz, der diese übersetzt und verfügbar macht
- Über diese API können die HTTP-Methoden genutzt werden (GET, PUT, POST, DELETE, HEAD, OPTIONS, CONNECT und TRACE)

GET

- Retrieves a resource
- Guaranteed not to cause side-effect (SAFE)
- Cacheable

POST

- Creates a new resource
- Unsafe, effect of this verb isn't defined by HTTP

PUT

- Updates an existing resource
- Used for resource creation when client knows URI
- Can call N times, same thing will always happen (idempotent)

DELETE

- Removes a resource
- Can call N times, same thing will always happen (idempotent)

Quelle:

<https://www.oreilly.com/library/view/restful-net/9780596155025/ch04.html>

REST

Representational state transfer API

- Wurde erstmalig von Roy Fielding im Jahre 2000 vorgestellt
- Eine Webanwendung, welche REST-konform implementiert wurde, wird auch als RESTful Web-Service bezeichnet
- REST und SOAP werden häufig als Alternativen zueinander betrachtet

GET

- Retrieves a resource
- Guaranteed not to cause side-effect (SAFE)
- Cacheable

POST

- Creates a new resource
- Unsafe, effect of this verb isn't defined by HTTP

PUT

- Updates an existing resource
- Used for resource creation when client knows URI
- Can call N times, same thing will always happen (idempotent)

DELETE

- Removes a resource
- Can call N times, same thing will always happen (idempotent)

Quelle:

<https://www.oreilly.com/library/view/restful-net/9780596155025/ch04.html>

REST

Eine REST- Architektur wird durch sechs Prinzipien (Constraints) definiert:

1. Client-Server-Modell

- Sieht die Trennung von Nutzerinterface und Datenhaltung vor

2. Stateless

- Jede Anfrage an den Server **soll vollständig sein**, und benötigt keine Zusatzinformationen (keine extra Zustandserfassung)

3. Cacheable

- Vom Server bereitgestellte Antworten können vom Client zwischengehalten werden, um bei gleicher Anfrage die Informationen wiederzuverwenden
- Erfordert Angabe von `cacheable` oder `non-cacheable`

REST

Eine REST- Architektur wird durch sechs Prinzipien (Constraints) definiert:

4. Uniform interface

- REST-konforme Komponenten bilden einheitliche und vom implementierten Dienst entkoppelte Schnittstellen

5. Layered system

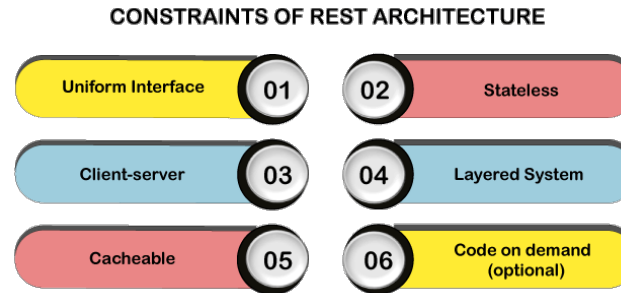
- REST nutzt mehrschichtige, hierarchische Systeme
- Dem Nutzer werden dabei nur bestimmte Schnittstellen angeboten
- Dahinterliegende Ebenen können verborgen bleiben

REST

Eine REST- Architektur wird durch sechs Prinzipien (Constraints) definiert:

6. Code-On-Demand

- Eine optionale Forderung von Fielding
- Ausführbare Programmteile werden vom Client nachgeladen und zur Entwicklung bereitgestellt (Applets oder Skripte)



REST

Einen RESTful-Service Anfragen

- Eine einfache Anfrage setzt sich zusammen aus der HTTP-Methode, den Ressource Locator mit Parametern und optional einen übergebenen Datensatz
- Effektiv kann ein RESTful-Service (server-seitig) über AJAX (client-seitig) Angefragt werden
- Die Anfrage wird vom Server entgegengenommen, verarbeitet und eine Antwort zurückgegeben
- Der RESTful-Service wird also server-seitig implementiert und zur Verfügung gestellt

REST

Entscheidend bei dem Entwurf REST konformer Web Services ist die Identifikation und Benennung von Ressourcen.

- Für unterschiedliche Typen von Ressourcen haben sich Namenskonventionen herausgebildet, die beim Entwurf der URIs berücksichtigt werden sollten
 - Stehen Ressourcen in einer hierarchischen Beziehung zueinander, dann wird dies über Pfadvariablen zum Ausdruck gebracht:
`http://127.0.0.1:8080/Erde/Europa/Deutschland/Bochum`
 - Besteht ein semantischer Zusammenhang zwischen Ressourcen, die aber nicht hierarchisch miteinander verbunden sind, werden sie als Tupel dargestellt:
`http://127.0.0.1:8080/Erde/51.4439,7.261572`
`http://127.0.0.1:8080/Mischfarben/rot;blau`

REST

- Ein Komma wird verwendet, wenn die Reihenfolge relevant ist:

Erde/51.4439,7.261572 \neq Erde/7.261572,51.4439



REST

- Ein **Semikolon** wird verwendet, falls die Reihenfolge nicht wichtig ist.

Mischfarben/rot;blau = Mischfarben/blau;rot

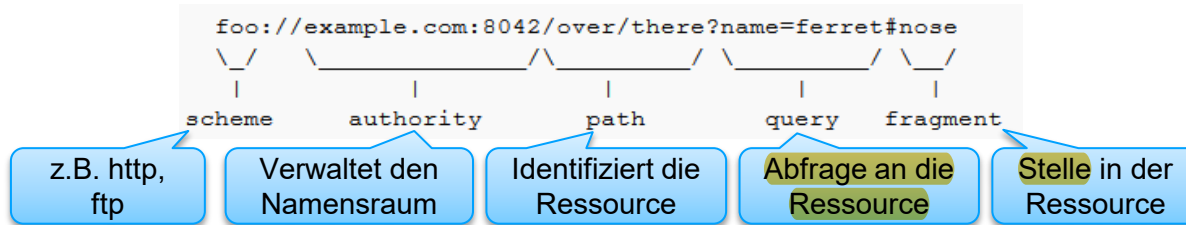


- Stellt eine Ressource das Ergebnis eines Algorithmus dar, der auf Daten angewendet wird, kommen **Query-Variablen** zum Einsatz:

<https://www.google.de/search?q=Ruhr-Uni>

REST

- Zur Beschreibung der Methoden sind die Eigenschaften der Sicherheit und Idempotenz hilfreich
 - Ein Methodenaufruf ist `sicher`, wenn er **keine Veränderungen** an Ressourcen bewirkt
 - Ein Methodenaufruf ist `idempotent`, wenn ein einmaliger Aufruf dieselben Effekte hat wie mehrfache identische Aufrufe
- Aufbau eines Uniform Resource Identifier (URI):



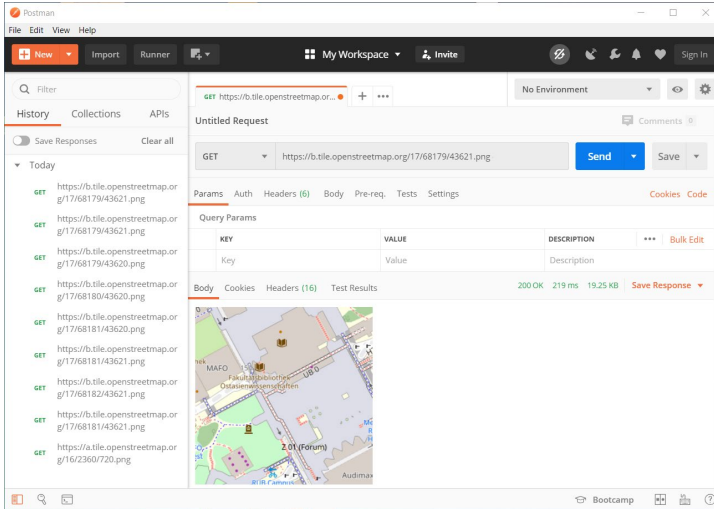
REST

Anfragen generieren und Debuggen

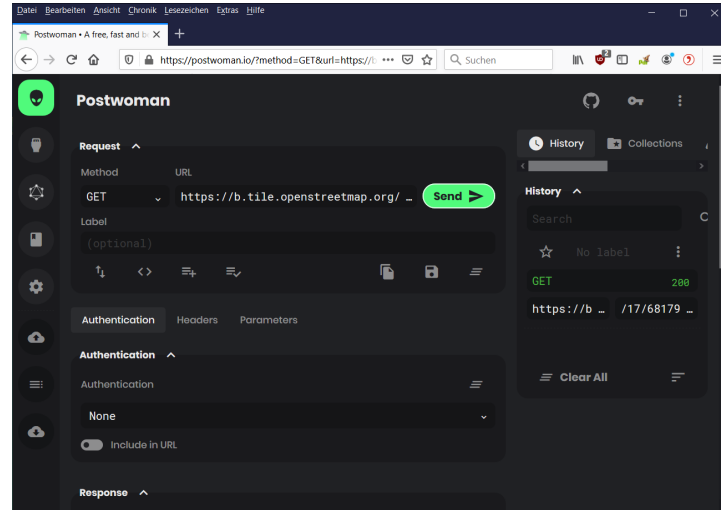
- Mit Werkzeugen wie Postman und Postwoman wird die Entwicklung mit AJAX und REST vereinfacht.
 - Ermöglicht es HTTP-Anfragen zu erzeugen und die Antwort zu analysieren
 - Postman ist kommerziell vertrieben, verfügt aber über eine kostenfreie Version
 - Postwoman ist Open-Source und wird über den Browser bedient

REST

Anfragen generieren und Debuggen



Quelle: <https://www.postman.com/>



Quelle: <https://postwoman.io/>

Einführung in Node.js

Serverside JavaScript

Einführung in Node.js

- Node.js ist eine Open-Source-Plattform, die zur Ausführung von JavaScript außerhalb von Browsern genutzt wird
- Node.js basiert auf der JavaScript-Laufzeitumgebung V8, die ursprünglich für Google Chrome entwickelt wurde
- Die **asynchrone Architektur** von Node.js ermöglicht eine **parallele Verarbeitung von Client-Verbindungen oder Datenbank-Zugriffen**
- Code kann zwischen beiden Seiten (Server, Browser) geteilt und gemeinsam verwendet werden, da beide Seiten JavaScript verwenden

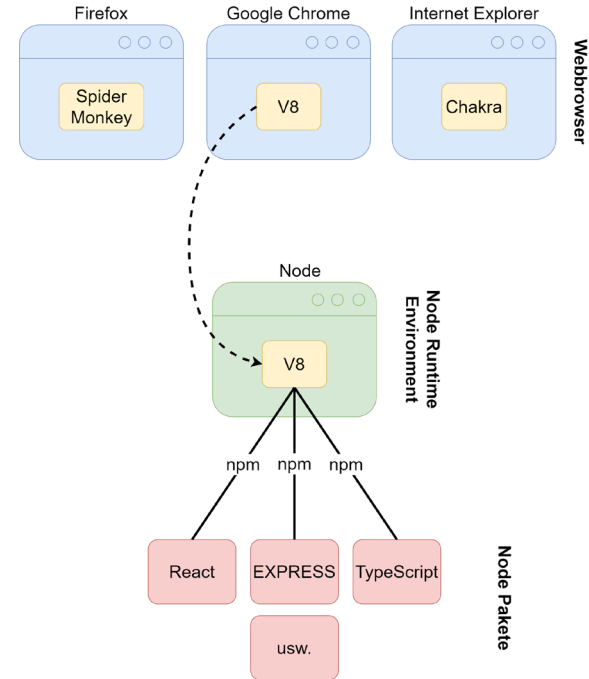


Quelle: https://de.wikipedia.org/wiki/Datei:Node.js_logo.svg

Einführung in Node.js

Node.js ist ein Runtime Environment, welche die JS-Laufzeitumgebung v8 integriert.

- JavaScript-Laufzeitumgebungen unterscheiden sich von Browser zu Browser
 - Kommt oft zu Problemen der Kompatibilität
 - V8 zählt zu den schnellsten und vollständigsten Laufzeitumgebungen

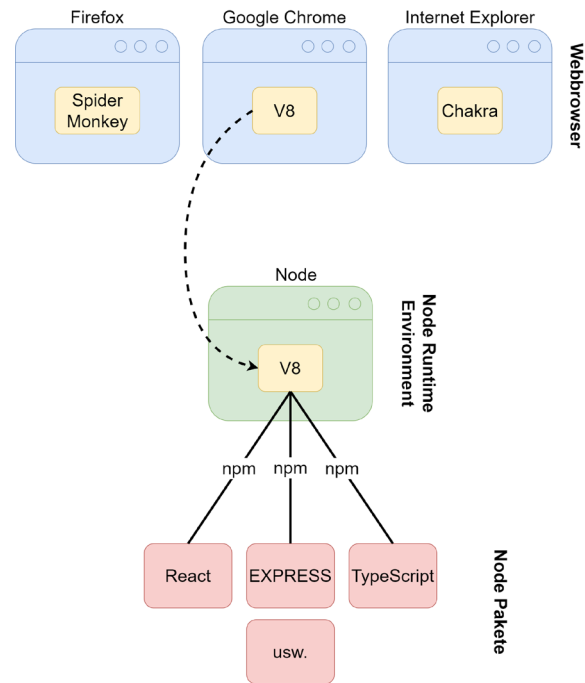


Einführung in Node.js

操作时的软硬件
组合和配置的集
合

Node.js ist ein **Runtime Environment**,
welche die JS-Laufzeitumgebung v8
integriert.

- Node.js ist kein Browser, verfügt daher über eigene vordefinierte Funktionen
 - Hat kein `document`-Objekt oder eine `alert`-Funktion
 - Besitzt stattdessen Zugriff auf das File-System (`fs.readFile`) oder kann Server erstellen (`http.createServer`)



Einführung in Node.js

- Die JavaScript Engine V8 von Google als Node.js Basis
 - JavaScript-Code wird bei der Ausführung durch so genannte Just-in-time-Kompilierung zunächst in nativen Maschinencode übersetzt
 - Besitzt eine exakte automatische Speicherbereinigung, die Speicher schnell in kleinen Portionen zuweist und wieder freigibt



Quelle: <https://de.wikipedia.org/wiki/V8>

Einführung in Node.js

- Node.js ist modular aufgebaut.
 - Ein Modul ist eine abgeschlossene funktionale Einheit (bestehend aus Objekten, Funktionen und Datenstrukturen)
 - Weitere Module können über den mitgelieferten Paketmanager npm nachinstalliert werden



Quelle: <https://de.wikipedia.org/wiki/V8>

Node Package Manager (npm)

Über die Kommandozeile können mit `npm` bekannte Pakete/Module oder Anwendungen in Node.js integriert werden.

- `npm` = Standardpaketmanager für die JavaScript-Laufzeitumgebung `Node.js`
- Es gibt aber auch für die JavaScript front-end Entwicklung Pakete
<https://docs.npmjs.com/about-npm/>
- Es empfiehlt sich Node.js über IDEs mit integrierter Kommandozeile zu verwenden (bsp. Visual Studio Code)
- Um zu überprüfen ob die Programme Installiert sind können Sie diese Kommandos zur Versionsabfrage in ihrem Terminal ausführen:

```
C:\Users\...> node -v  
C:\Users\...> npm -v
```



Quelle:
[https://de.wikipedia.org/wiki/Npm_\(Software\)](https://de.wikipedia.org/wiki/Npm_(Software))

Node Package Manager (npm)

- Pakete werden über die Kommandozeile mit `npm install` installiert

```
C:\Users\...> npm install express  
C:\Users\...> npm install mysql --save  
C:\Users\...> npm install nodemon --global
```

Kommandozeile

Mit `--global` werden
Pakete global installiert
MySQL gibt express als
Voraussetzung an

- Eigenes Node.js Projekt initialisieren mit `npm init`
 - Generiert eine `package.json` Datei
 - Innerhalb dieser JSON-Datei werden Metadaten angegeben (z.B. Name, Version, Beschreibung und Schlüsselwörter)

```
C:\Users\...> npm init --yes
```

Kommandozeile

Durch `--yes` wird die JSON-Datei mit Standardinformationen generiert. Ansonsten muss ein Dialog mit allen Eingaben einzeln durchlaufen werden.

Node Package Manager (npm)

Beispiel einer `package.json` Datei:

```
{
  "name": "dbConnection", // Name des Projekts
  "version": "0.92.12", // Version des Projekts
  "description": "Datenbankverbindung.", // Beschreibung des Projekts
  "main": "index.js"
  "license": "MIT", // Lizenz des Projekts
  "dependencies": { // Ausführungs-Abhängigkeiten
    "fill-keys": "^1.0.2",
    "module-not-found-error": "^1.0.0",
    "resolve": "~1.1.7"
  },
  "devDependencies": { // Entwicklungs-Abhängigkeiten
    "mocha": "~3.1",
    "native-hello-world": "^1.0.0",
    "should": "~3.3",
    "sinon": "~1.9"
  }
}
```

Die `package.json` enthält ebenfalls alle Angaben über die Abhängigkeiten, welche für das Ausführen des Projektes notwendig sind. Diese werden nicht mit in das Projekt, sondern separate über den npm nachgeladen

Es werden zwischen Paketen für die Entwicklung und für die Ausführung unterschieden.

Node.js Server

Es gibt eine Vielzahl in Node.js integrierte Module, auch für **das erstellen eines Servers**.

- Installierte Module werden über die Funktion `require()` geladen. Es wird auch automatisch ein entsprechendes Objekt erstellt
- Es ist zu empfehlen Module **als `const`** zu definieren um eine Überschreibung zu vermeiden (`best practice`)
- Das `http` Modul stellt ein Objekt mit Funktionalitäten zur Erstellung eines Webserver und Webclients zur Verfügung
- Das **`url` Modul** stellt ein Objekt mit Funktionalitäten zur Verarbeitung von URL-Adressen zur Verfügung

```
const http = require('http');  
const url = require('url');
```

Node.js Server

Node.js Server reagiert auf Anfragen eines Clients (z.B. Webbrowsers) und überträgt gewisse Daten.

- Die `createServer`-Methode des HTTP-Objektes erstellt einen Webserver
- Der Funktion `createServer` wird eine Callback-Funktion übergeben
- Die Callback-Funktion des Webserver besitzt zwei Parameter
 - `Request`-Objekt, welches die Anfrage und sämtliche übermittelte Daten des Clients an den Server zwischenhält
 - Das `Response`-Objekt dient der Beantwortung einer Anfrage

Node.js Server


```
const server = http.createServer((request, response) => {  
  response.statusCode = 200; ← HTTP-Statuscode 200 = OK  
  response.setHeader('content-type',  
    'text/plain; charset=utf-8'); ← setHeader setzt den  
                                     Content-Type auf einfachen  
                                     Text, in einem bestimmten  
                                     Zeichensatz  
  //DO SOMETHING HERE  
});
```

- Mit dem url-Objekt werden die einzelnen Elemente der URL ausgewertet
- Die Attribute werden in einem eigenen Query-Objekt abgelegt und können anschließend abgefragt werden

Node.js Server

```
var urlString = url.parse(request.url, true);
```

```
response.write('Hello');  
response.end(urlString.query.name);
```



Durch die Angabe von true werden auch die übergebenen Query-Attribute geparkt (beispielsweise name=Markus).

- Mit Hilfe der Methoden `write` und `end` des Response-Objektes können Antworten an den Client geschickt werden.
- Die Methode `end` beendet die Antwort und der Client wird informiert, dass die übertragende Information vollständig ist

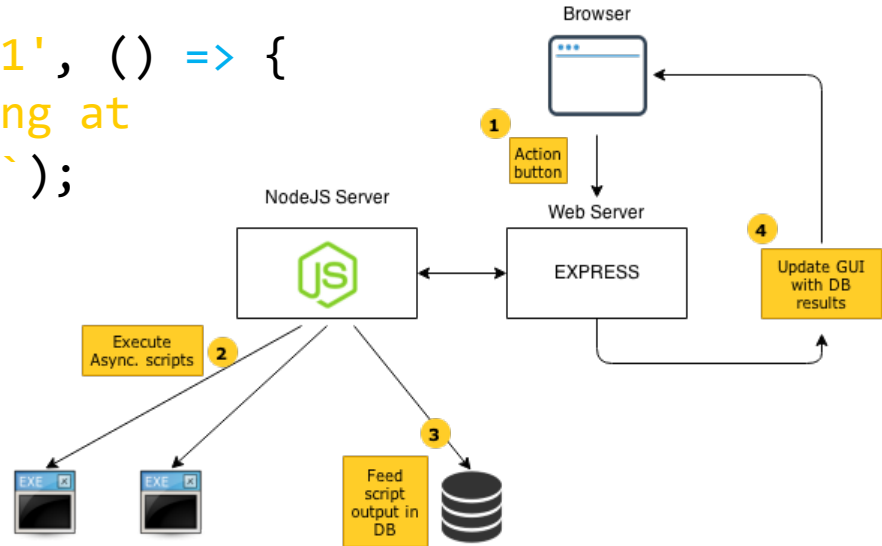
Node.js Server

- Ein Server wird über die Freigegebene Adresse und Port des ausführenden Gerätes angefragt
- Die Entwicklung eines Servers findet in der Regel Lokal statt, dafür kann die lokale IP des Gerätes (127.0.0.1 oder localhost) als Adresse verwendet werden
- Viele Ports sind offiziell reserviert für spezielle Dienste
 - Port: 20 → File Transfer Protocol (FTP)
 - Port: 80 → Hypertext Transfer Protocol (HTTP)
 - Port: 443 → Hypertext Transfer Protocol over SSL/TSL (HTTPS)

Node.js Server

- Über den Aufruf der Methode `listen()` wird der Webserver unter der Angabe eines Ports und einer Adresse gestartet

```
server.listen(80, '127.0.0.1', () => {  
  console.log(`Server running at  
http://${hostname}:${port}/`);  
});
```



Node.js Server

Mit Node.js können JavaScript Server aufgesetzt werden.

```
const http = require('http');
const url = require('url');
const hostname = '127.0.0.1';
const port = 8080;
const server = http.createServer((request, response) => {
  response.statusCode = 200;
  response.setHeader(
    'content-type', 'text/plain; charset=utf-8');

  let urlString = url.parse(request.url, true);
  let name = urlString.query.name;
  response.end('Hello ' + name);
});
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Erforderliche Node.js Standard-Module werden geladen. Module als `const` für `best practice` (verhindert Überschattung).

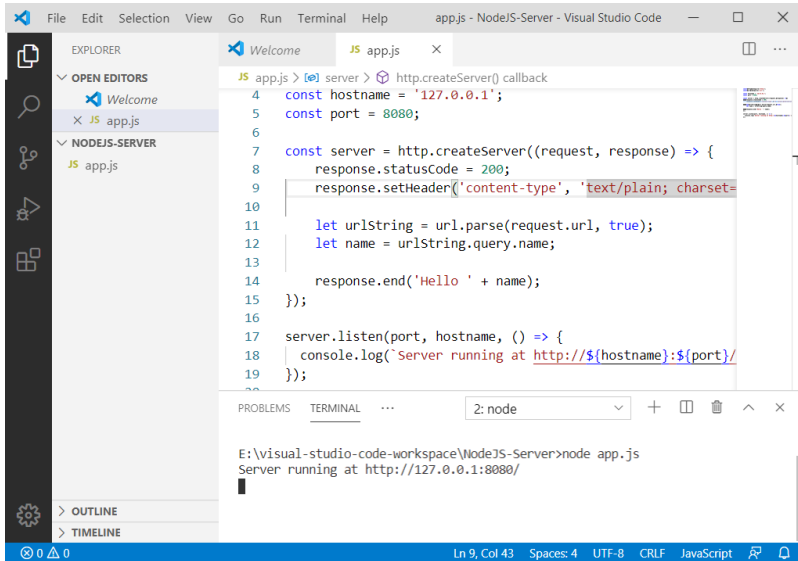
Erforderliche Node.js Standard-Module werden geladen.

Definiert den Server und die Verarbeitung einer Anfrage. Erzeugt eine Antwort (Response).

Server wird gestartet meldet sich beim Start.

Node.js Server

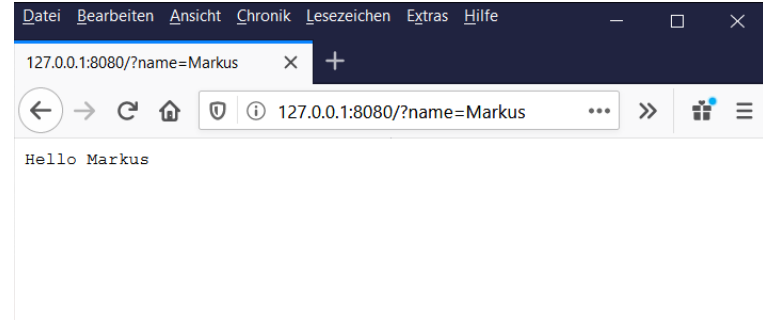
Mit Node.js wird ein einfacher Webserver implementiert, der eine HTML-Seite für einen anfragenden Webbrowser erstellt und zurückliefert.



```
app.js > server > http.createServer() callback
4 const hostname = '127.0.0.1';
5 const port = 8080;
6
7 const server = http.createServer((request, response) => {
8   response.statusCode = 200;
9   response.setHeader('content-type', 'text/plain; charset=
10
11   let urlString = url.parse(request.url, true);
12   let name = urlString.query.name;
13
14   response.end('Hello ' + name);
15 });
16
17 server.listen(port, hostname, () => {
18   console.log('Server running at http://${hostname}:${port}/');
19 });
```

PROBLEMS TERMINAL ... 2: node

E:\visual-studio-code-workspace\NodeJS-Server>node app.js
Server running at http://127.0.0.1:8080/



Web-Engineering

express.js

REST-konformer Umgang mit HTTP Requests

- REST-konforme Web Services nutzen HTTP als einheitliche Schnittstelle, um auf Ressourcen zuzugreifen
- Wie ein Server auf einen HTTP-Request reagiert, ist eine Frage der Implementierung
- Mit der Methode (GET, POST, ...) wird eine Ressource erfragt → Server reagiert
- Mit einem einfachen HTTP-Server ist die Unterscheidung der Methoden umständlich:

REST-konformer Umgang mit HTTP Requests

```
const server = http.createServer((request, response) => {  
  if(request.method == 'GET'){  
    //GET-METHOD HERE  
  }else if(request.method == 'POST'){  
    //POST-METHOD HERE  
  }  
});
```

- Express wird über `npm` als Modul installiert

Kommandozeile

```
C:\Users\...> npm install express  
C:\Users\...> npm install cors
```

← Muss zusätzlich geladen werden.

REST-konformer Umgang mit HTTP Requests

- Vereinfachte Syntax für den Server-Start:

```
const express = require("express");
const cors = require("cors");
var app = express();
app.use(express.json(), cors());
app.listen(8080, () => {
  console.log("Server running on port 8080");
});
```

Laden und einstellen von Express. Ein **Body-parser** wird benötigt, um den Inhalt von Anfragen bearbeiten zu können. Dieser ist in express integriert in kann durch `express.json()` gesetzt werden. Bei cors handelt es sich um eine middleware für Cross-origin resource sharing.

Mit **listen** wird der Server unter Angabe eines Ports gestartet.

- Express verbessert den Server durch **Deklaratives Routing**:

```
app.METHOD("PATH", HANDLER);
```

REST-konformer Umgang mit HTTP Requests

Deklaratives Routing

- Eine Route wird deklariert und ein entsprechender Handler hinzugefügt, der zur Verarbeitung genutzt wird
- Nested Routing ist generell ebenfalls möglich, dazu werden die Elemente einfach verschachtelt
- Es ist nicht nötig alle geplanten Routen am Anfang in einer Datei anzugeben
- Intuitivere Nutzung im Programmfluss, da es an den Stellen genutzt werden kann, an denen es benötigt wird

REST-konformer Umgang mit HTTP Requests

Die **GET-Methode** wird verwendet, um **lesend auf Ressourcen zuzugreifen**.

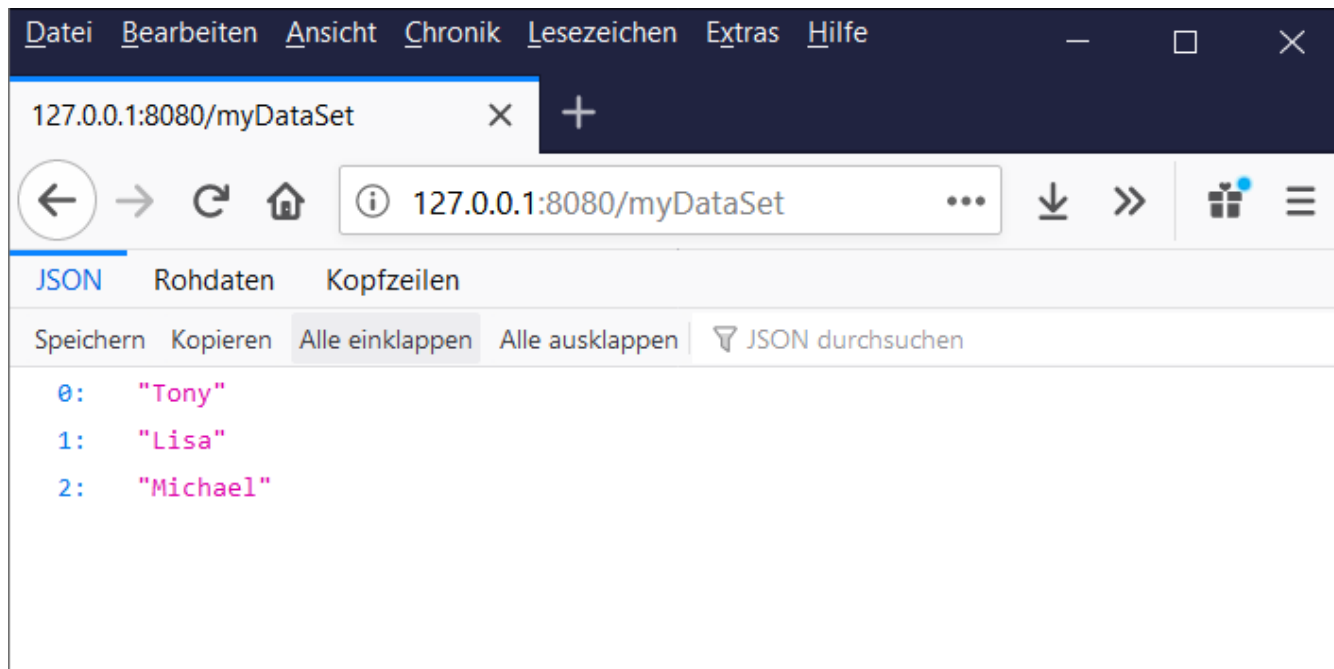
- Ein GET Request beinhaltet keinen Entity Body
- Metadaten im Header des Requests können genutzt werden, um Präferenzen zur Repräsentation einer Ressource mitzuteilen → Content Negotiation
- Der Server reagiert mit einer Response, welche eine Repräsentation der Ressource im Entity Body enthält

```
app.get("/myDataSet", (req, res) => {  
    res.json(  
        ["Tony", "Lisa", "Michael"]  
    );  
});
```

Eine Ressource mit **myDataSet** als **Anfragepfad**, die beim Aufruf der GET-Methode einen kleinen Datensatz mit Namen liefert.

REST-konformer Umgang mit HTTP Requests

GET-Methode testweise ausgeführt:

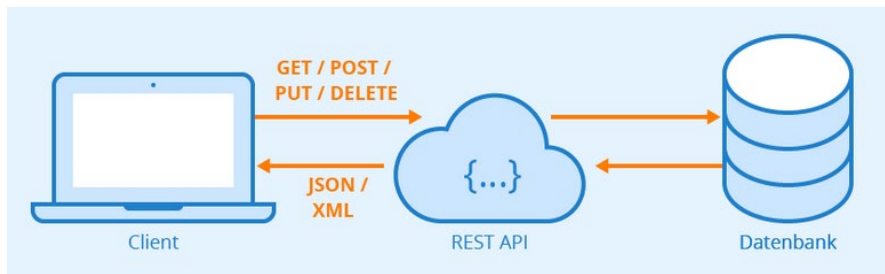


REST-konformer Umgang mit HTTP Requests

Die **PUT-Methode** wird genutzt, um eine Ressource neu zu erstellen oder eine bestehende Ressource zu überschreiben.

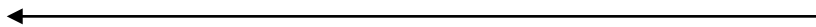
- Der Entity Body des Requests enthält eine Repräsentation der anzulegenden Ressource
- Die Ressource wird vom Server unter der angegebenen URI abgelegt

GET	<ul style="list-style-type: none">• Retrieves a resource• Guaranteed not to cause side-effect (SAFE)• Cacheable
POST	<ul style="list-style-type: none">• Creates a new resource• Unsafe, effect of this verb isn't defined by HTTP
PUT	<ul style="list-style-type: none">• Updates an existing resource• Used for resource creation when client knows URI• Can call N times, same thing will always happen (idempotent)
DELETE	<ul style="list-style-type: none">• Removes a resource• Can call N times, same thing will always happen (idempotent)



REST-konformer Umgang mit HTTP Requests

```
let contracts = [];
```



Platzhalter für die Dokumente.

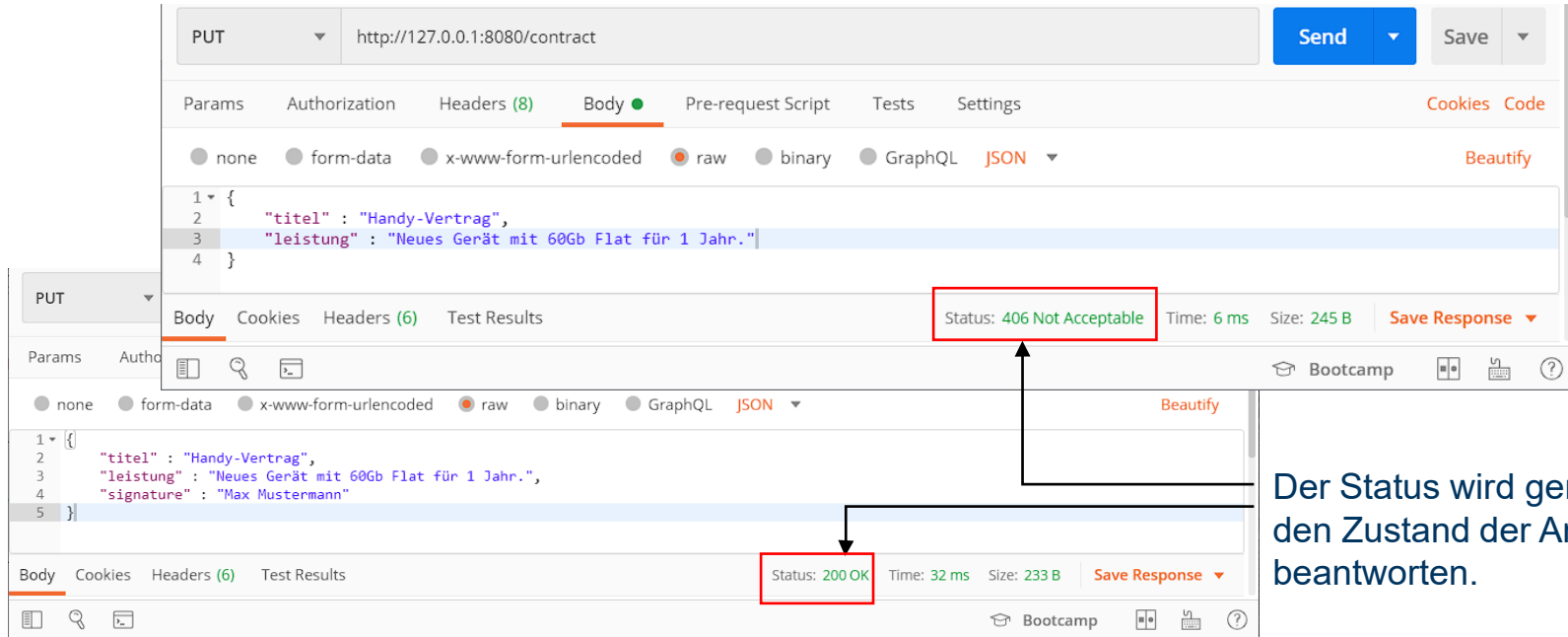
```
app.put("/contract", (req, res) => {  
  let contractTopics = Object.keys(req.body);  
  const check = (topic) => contractTopics.includes(topic);  
  const mustHaveTopics = ["titel", "leistung", "signature"];  
  if(mustHaveTopics.every(check)){  
    contracts.push(req.body);  
    res.status(200); //Ok  
  }else{  
    res.status(406); //Not Acceptable  
  }  
  res.send("Requested contract Ressource.");  
});
```

Überprüft das eingehende Dokument zunächst auf Vollständigkeit. Die Arrow function namens `check` führt beim Aufruf von `every` die Prüfung aller Einträge von `contractTopics` durch.

Je nach Prüfungsergebnis wird das Dokument akzeptiert oder verworfen.

REST-konformer Umgang mit HTTP Requests

PUT-Methode testweise ausgeführt (mit Postman):



Der Status wird genutzt, um den Zustand der Anfrage zu beantworten.

REST-konformer Umgang mit HTTP Requests

POST dient dazu, bestehende Ressourcen, um untergeordnete Informationen zu ergänzen.

- Anlegen einer neuen (Sub-)Ressource unterhalb eines per URI identifizierten Eltern-Elements (bsp. Kommentar zu einem Blog-Eintrag)
- Anhängen eines Inhalts an eine Ressource (bsp. Hinzufügen eines neuen Log-Eintrags als Zeile)

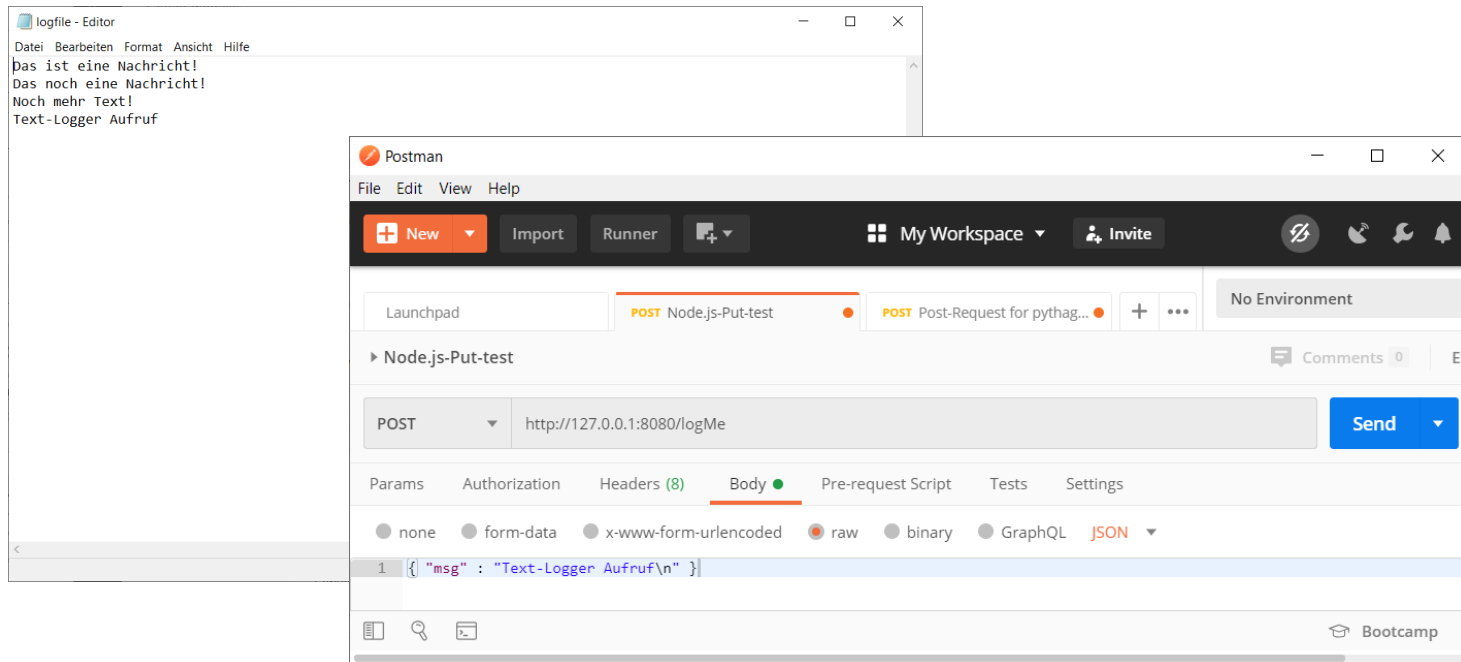
```
const fs = require("fs");  
  
app.post("/logMe", (req, res) => {  
  fs.appendFile('logfile.txt', req.body.msg,  
    function(err) {  
      if (err) throw err;  
      console.log('Saved!');  
    });  
  res.send('POST received! Added Log entry.');
```

Das FileSystem Modul (fs) wird für Lese- und Schreibfunktionen benötigt.

Eine POST-Methode. Schreibt Text in eine Log-Datei namens logfile.txt, welche sich lokal im Server-System befindet.

REST-konformer Umgang mit HTTP Requests

POST-Methode testweise ausgeführt:



REST-konformer Umgang mit HTTP Requests

- Mittels `DELETE` wird die an der URI des Requests befindliche Ressource gelöscht
- Der Request benötigt keinen Entity Body
- Die Server Response kann als Entity Body eine Statusmeldung zurückgeben – muss dies aber nicht

//Dateisystem: blog/articles/01/cat.jpg

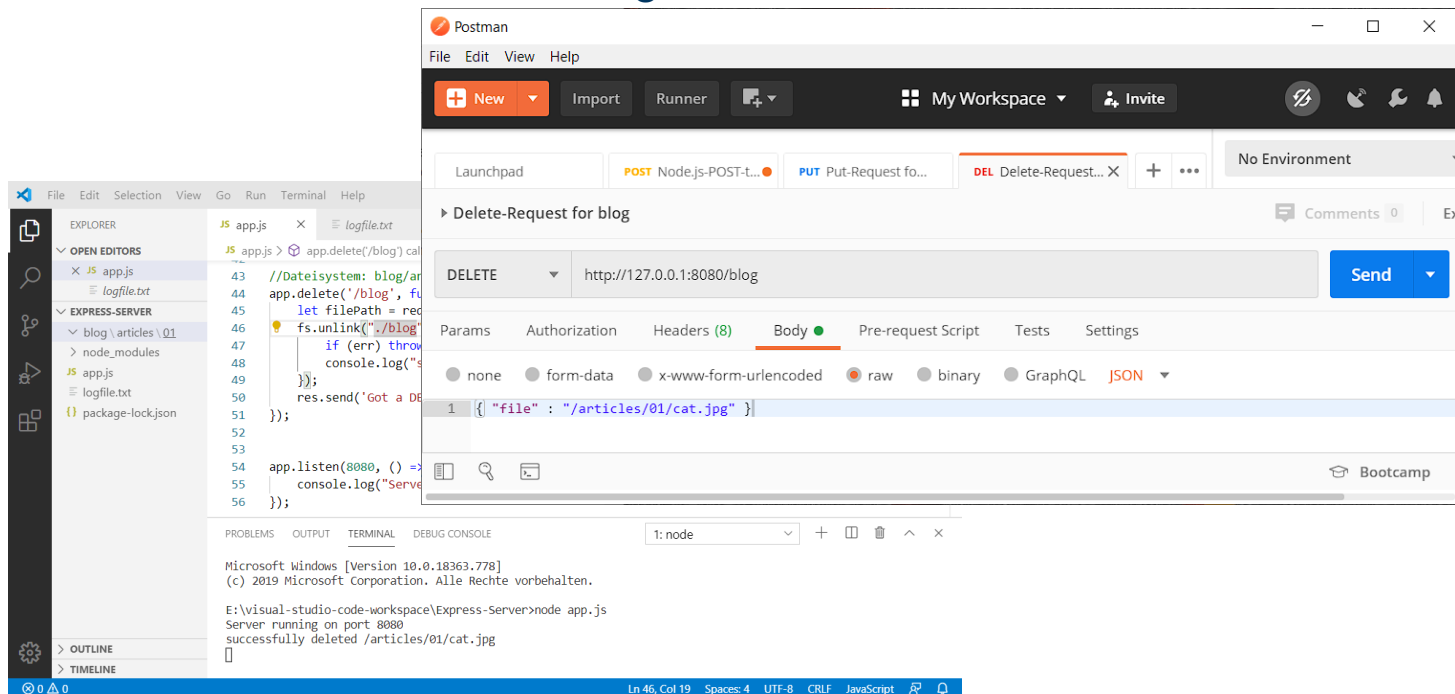
```
app.delete('/blog', (req, res) => {  
  let filePath = req.body.file;  
  fs.unlink("./blog" + filePath, (err) => {  
    if (err) throw err;  
    console.log("successfully deleted " + filePath);  
  });  
  res.send('Got a DELETE request at /blog');  
});
```

← Datei die gelöscht werden soll wird übermittelt.

} Löscht eine spezifische Ressource im Dateisystem des Servers.

REST-konformer Umgang mit HTTP Requests

DELETE-Methode testweise ausgeführt:



REST-konformer Umgang mit HTTP Requests

- **HEAD** dient dazu, **Metadaten zu einer Ressource abzufragen**: Ein Client kann per HEAD z.B. prüfen, **ob eine Ressource existiert**
 - Der Einsatz von HEAD ist insbesondere dann sinnvoll, wenn das Laden des potenziell großen Entity Body einer Ressource vermieden werden soll
- **OPTIONS** kann genutzt werden, **um die auf einer Ressource für den Client verfügbaren Methoden zu ermitteln**
 - In der **Response** des Servers werden die verfügbaren Methoden **im HTTP Header Allow** aufgezählt
- Der HTTP-Statuscode wird über einen Zahlencode angegeben (bsp. 200 = Ok)
 - Von denen können viele zur Auszeichnung einer Antwort genutzt werden
<https://de.wikipedia.org/wiki/HTTP-Statuscode>

REST-konformer Umgang - Weiterleitungen

- **Weiterleitungen (Routing)** wird verwendet, um Anwendungsendpunkte und deren Antworten bei Clientanforderungen besser zu koordinieren
- Express verwendet für den Abgleich **path-to-regexp**, wodurch auch Reguläre Ausdrücke in der Definition von Pfaden angegeben werden können
- So können dynamische Pfade definiert werden, welche einem Muster folgen
 - Unterschiedliche Pfade sprechen denselben Endpunkt an

Anfrage auf Basis von Zeichenfolgen:

```
app.get('/', ... );  
app.get('/about', ... );  
//statische Anfragen an Ressourcen
```

```
app.get('/ab?cd', ... );  
//abd oder abcd
```

```
app.get('/ab+cd', ... );  
//abcd, abbcd, abbbcd usw.
```

```
app.get('/ab*cd', ... );  
//abcd, abxcd, ab123cd, usw.
```

```
app.get('/ab(cd)?e', ... );  
//abe und abcde
```

Anfrage auf Basis von Regex:

```
app.get(/a/, ... );  
//alles, was ein a enthält
```

```
app.get(/.*fly$/, ... );  
//alles, was mit fly endet
```


REST-konformer Umgang - Weiterleitungen

- Eine REST-Methode kann mehrere Callback-Methoden enthalten und nacheinander bearbeiten
 - Es muss dafür das Objekt `next` angegeben werden
- Um Redundanz und Schreibfehler zu vermeiden, werden verkettete Routenhandler verwendet
 - Strukturiert unterschiedliche Methoden unter einem einheitlichen Pfad

REST-konformer Umgang - Weiterleitungen

Angeben mehrerer Callbacks:

```
app.get('/example/test',  
function (req, res, next) {  
  //Mache etwas hier  
  next();  
}, function (req, res) {  
  //Mehr Code hier  
});
```

Verkettung durch Routenhandler:

```
app.route('/book')  
  .get(function(req, res) {  
    res.send('Get a random book');  
  })  
  .post(function(req, res) {  
    res.send('Add a book');  
  })  
  .put(function(req, res) {  
    res.send('Update the book');  
  });
```