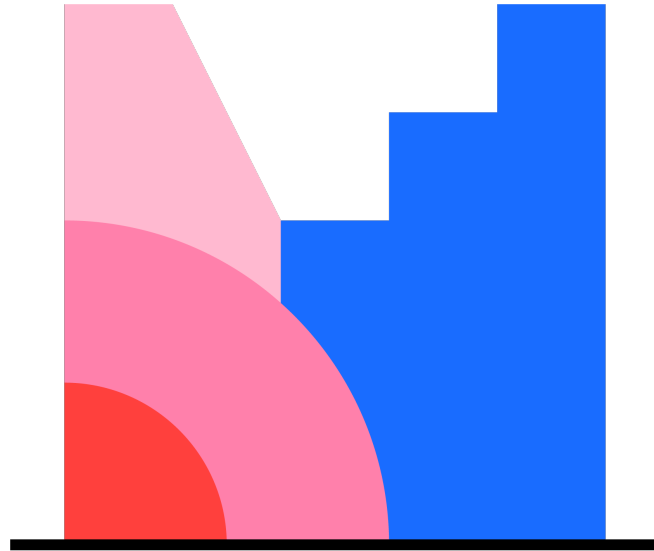













# Blockwoche: Web Programming Lab 🚀

# Recap Client-Side JavaScript I



**Mentimeter**

# Onboarding – Programm Blockwoche

Montag1 	Dienstag  	Mittwoch  	Donnerstag   	Freitag   
Architekturansätze von Web Anwendungen  JavaScript Sprachkonzepte I	Client-Side- JavaScript I	Angular	Angular	Offline- & Progressive Web Apps
JavaScript Sprachkonzepte II	Client-Side- JavaScript II Frameworks & Typescript	Angular	<b>Server-Side- JavaScript</b>	Authentication @ Web Apps

# Agenda

- **Was ist Node.js?**
- **HTTP APIs mit Node.js HTTP Module & Express**
- **Persistieren mit MongoDB**
- **API Integration Testing mit Supertest und Jest (*evtl. Selbststudium*)**

# Repository für heute Nachmittag

👉 <https://github.com/web-programming-lab/nodejs-intro>

# Server-Side JavaScript

Intro Node.js Konzepte



# Was ist Node.js?

*"Node.js is a platform built on **Chrome's JavaScript runtime** for easily building **fast, scalable network applications**. Node.js uses an **event-driven, non-blocking I/O model** that makes it lightweight and efficient, perfect for **data-intensive real-time applications** that run across distributed devices." – [nodejs.org](https://nodejs.org)*

# Node.js – Was ist Node.js?

- **Node.js** ist eine serverseitige Plattform in der Softwareentwicklung zum Betrieb von Netzerkanwendungen. Insbesondere lassen sich Webserver damit realisieren.
- **Node.js** wird in der JavaScript-Laufzeitumgebung „[V8](#)“ ausgeführt, die ursprünglich für [Google Chrome](#) entwickelt wurde.
- **Node.js** bietet eine ressourcensparende Architektur, die eine besonders große Anzahl gleichzeitig bestehender Netzwerkverbindungen ermöglicht.
- JavaScript gibt per se eine [ereignisgesteuerte Architektur](#) vor.
- Diese hat im Serverbetrieb den Vorteil, pro bestehender Verbindung weniger [Arbeitsspeicher](#) zu verbrauchen als bei vergleichbaren Anwendungen, die für jede geöffnete Verbindung einen eigenen [Thread](#) starten.
- **Node.js** wird mit besonderem Fokus auf die Performance entwickelt. So kommt nonblocking I/O statt standardmäßigem blockierendem I/O zum Einsatz.

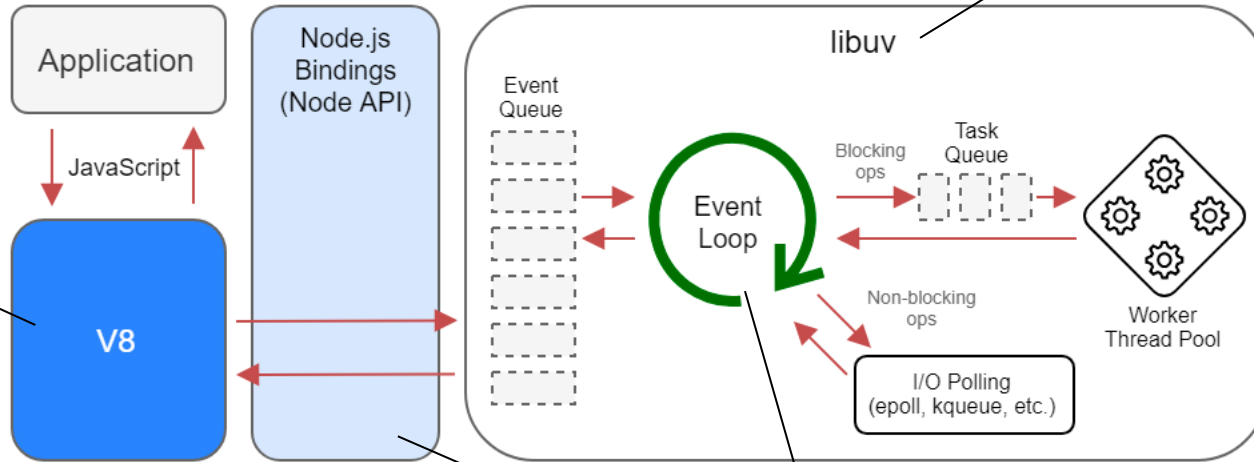


# Node.js – Interne Architektur

## V8

Googles OSS high-performance JavaScript Engine (geschrieben in C++), wird u.A. in Chrome verwendet.

<https://v8.dev/>



## libuv

Multiplattform Support Library mit dem Focus auf async i/o  
Filesystem, dns, network, worker thread pool, polling, etc. <https://libuv.org/>

Beispiele für explizite Blocking Ops:

### > I/O Bound:

- DNS Operationen wie z.B. `dns.lookup`
- File System Operationen, wie z.B. `fs.readFile`

### > CPU Bound:

- Crypto-Operationen (`crypto.randomBytes()`)
- Data compression (zlib)

## Event Loop

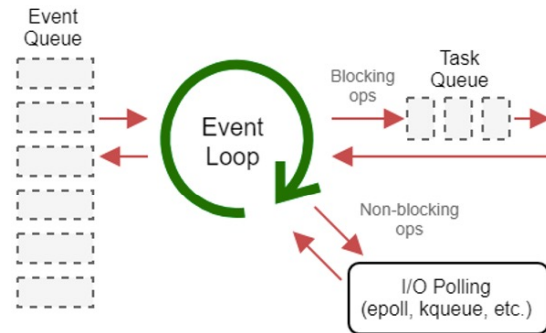
- Single Thread auf Basis libuv
- Semi-infinite, weil er runterfährt, wenn das Programm fertig ist.
- Jeder Node.js Prozess startet einen eigenen Event loop.

## Node API

- Timer, fs, http, crypto, etc.

# Node.js – Der Event Loop

Der Event Loop ist das **zentrale Element** von der Node.js Plattform.



## Warum?

Der Event Loop ermöglicht der Node.js Plattform asynchron sein und kann so per Design ein nicht blockierendes Programmiermodell zur Verfügung stellen.

Der Event Loop läuft in einem einzigen Thread. D.h er kann nur etwas gleichzeitig bearbeiten.

**Wichtig!** Daher sollte man dem Code entsprechend Aufmerksamkeit schenken und sicherstellen, dass nichts diesen einzelnen Thread blockiert – wie z.B. synchrone Netzwerk Calls oder infinite Loops.

# Node.js – Async Pattern

examples/read-file-sync.js

```
const fs = require('fs');  
  
// Liest den Datei-Inhalt synchron  
const data = fs.readFileSync(__filename);  
  
console.log('Data:', data);  
console.log('Done');
```

**Synchron**  
**”Blocking”**  
(Blockiert  
den Event-Loop!)

**Asynchron** via  
Callback-Pattern  
**”Non-Blocking”**

examples/read-file-async.js

```
const fs = require('fs');  
  
fs.readFile(__filename, (error, data) => {  
    console.log('Data: ', data);  
});  
  
console.log('not yet done ;-)');
```

# Node.js – Die Bearbeitung

Der Event Loop prüft kontinuierlich den Call Stack  
funktioniert nach LIFO (Last in, First Out).

```
const bar = () => console.log('bar');  
const baz = () => console.log('baz');  
  
const foo = () => {  
  console.log('foo');  
  bar();  
  baz();  
};  
  
foo();
```

examples/event-loop.js



# Node.js – Die Bearbei

Läuft der Timer ab (die Bearbeitung des Timer Queue verschoben.

Der Event Loop gibt Priorität dem Call Stack. I aus der Event Queue übernommen.

```
const bar = () => console.log('bar');
const baz = () => console.log('baz');

const foo = () => {
  console.log('foo');
  setTimeout(bar);
  baz();
};

foo();
```

examples/event-loop-queuing-functions.js



# Node.js – Die Bearbeitung des Callstacks (III)

Mit ES2015 wurde das Konzept der **Job Queue** eingeführt, das auch von Promises (async / await) verwendet werden. Man hat so die Möglichkeit das async functions so schnell wie möglich via Job Queue auszuführen. Die Job Queue ist das “Fast Pass” Tickets für asynchrone Funktionen.

event-loop-job-queue.js

```
const bar = () => console.log('bar');

const baz = () => console.log('baz');

const foo = () => {
  console.log('foo');
  setTimeout(bar, 0);
  new Promise((resolve) =>
    resolve('should be right after baz, before bar')
  ).then((resolve) => console.log(resolve));
  baz();
};

foo();
```

# Node.js – HTTP Server bauen

examples/run-a-webserver.js

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

const requestListener = (req, res) => {
  res.end('Hallo Web-Programming-Lab');
};

const server = http.createServer(requestListener);

server.listen(4566, () => console.log("Server started.));
```


HTTP Debug Modus aktivieren:

- **Demo:** NODE\_DEBUG="http" node run-a-webserver.js

# Node.js – Events & Callbacks

Event

Callback



```
setTimeout(() => process.exit(), 4000);  
process.on('exit', () => {  
    console.log('Exit process now.');
```

```
});  
  
console.log('hello web-programming-lab');
```

examples/node-async-example.js



# Node.js – stdin / stdout

Der 'readable' Event wird gefeuert, sobald auf dem Stream Daten gelesen werden können.

```
process.stdin.on('readable', () => {  
  const something = process.stdin.read();  
  if(something) {  
    process.stdout.write(something);  
  }  
});
```

examples/read-from-stdin.js

Event 'readable': [https://nodejs.org/api/stream.html#stream\\_event\\_readable](https://nodejs.org/api/stream.html#stream_event_readable)

Stream stdin: [https://nodejs.org/api/process.html#process\\_process\\_stdin](https://nodejs.org/api/process.html#process_process_stdin)

Stream stdout: [https://nodejs.org/api/process.html#process\\_process\\_stdout](https://nodejs.org/api/process.html#process_process_stdout)

# Node.js – Read Environment Variables

examples/read-env.js

```
console.log(`USER_ID: ${process.env.USER_ID}`);  
console.log(`USER_NAME: ${process.env.USER_NAME}`);
```

Aufruf auf der Commandline

```
USER_ID=123456 USER_NAME=Patrick node read-env.js
```

Unterstützendes npm-Package: [dotenv](#)

Lädt Umgebungsvariablen von einer .env Datei in das `process.env` Objekt.

→ Siehe `examples/dotenv-example`

# Node.js – Promises

```
const { readFile } = require('fs').promises;

console.log(require('fs').promises);

async function readMyFile() {
  const data = await readFile(__filename);
  console.log('Data: ', data);
}

readMyFile();

console.log('not yet done ;-');
```

examples/fs-promisify.js

# Node.js – Modules

## Was ist ein Module?

→ Eine Datei oder ein Ordner, der Code enthält.

Im Node Ökosystem gibt zwei Arten von Module Spezifikationen.

→ [CommonJS](#) (\*.cjs oder \*.js, bisher)

→ [ECMAScript Modules](#) (\*.mjs, “neu”, offizielles [ECMAScript](#) Standard Format)

**Wichtig:** Node.js JavaScript Code standardmässig als CommonJS Module (sofern kein type im package.json angegeben ist).

Dieses Verhalten kann man überschreiben resp. ECMAScript Modules verwenden:

- via “.mjs” Dateiendung
- im package.json der “type” auf module setzen
- via `--input-type` CMD Parameter

# Node.js – CommonJS Modules

Wie verwende ich ein CommonJS Module?


```
exports.myVar = 'This is my var';  
exports.log = (text) => console.log(text);
```

examples/commonjs/output.cjs



Exportieren von Funktionalität als Common.js Module via `exports`

Importieren eines Common.js Modules via `require`



```
const output = require('./output.cjs');  
output.log('hello web programming lab!');  
console.log(`${output.myVar}`);
```

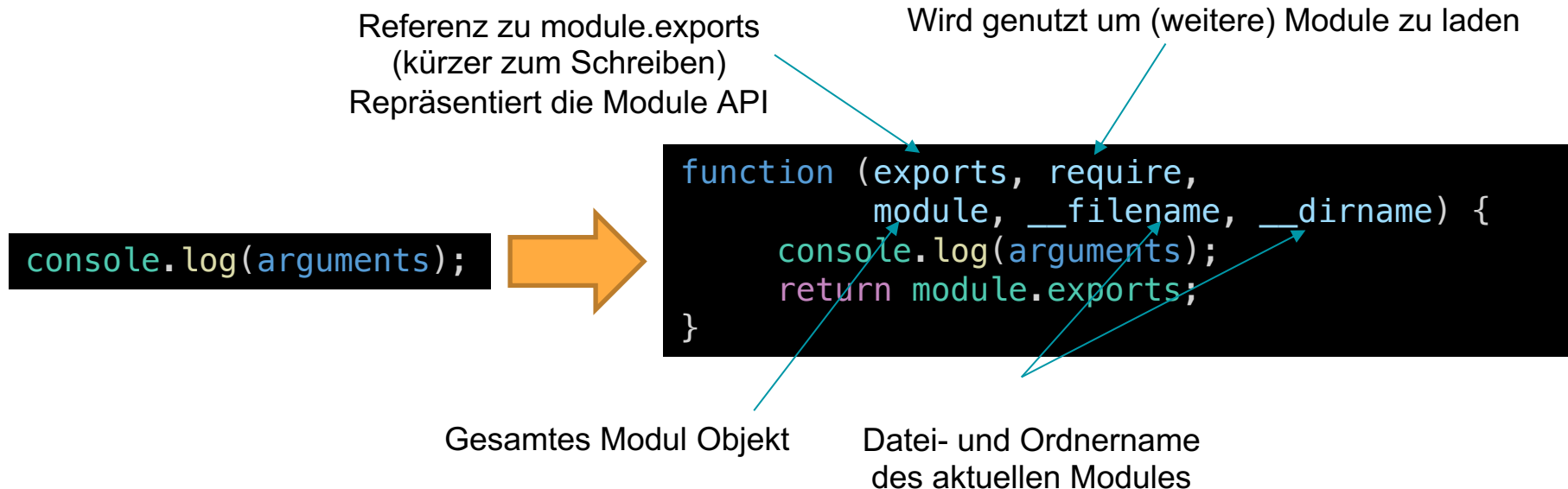
examples/commonjs/consumer.js

# Node.js – CommonJS Modules

## Wie ist ein CommonJS Module aufgebaut?

→ Sämtlicher Code wird als Function gewrappt gemäss der [CommonJS Module Spezifikation](#)

→ Dokumentation: [https://nodejs.org/api/modules.html#modules\\_the\\_module\\_wrapper](https://nodejs.org/api/modules.html#modules_the_module_wrapper)



# Node.js – ECMAScript Module (esm)

Wie verwende ich ein ECMAScript Module?

```
const myVar = 'This is my var';  
const log = (text) => console.log(text);  
  
export {myVar, log};
```

examples/esm/output.mjs

Exportieren von Funktionalität als ESM Module via `export`

Importieren eines ECMAScript Module via `import`

```
import { log, myVar } from './output.mjs';  
  
log('hello web programming lab!');  
log(`${myVar}`);
```

examples/esm/consumer.mjs

# Node.js – Übungsaufgabe I (10')

1. Gib das Folgende auf der Kommandozeile aus:

> "Hallo Web-Programming-Lab nach 4 Sekunden"

> "Hallo Web-Programming-Lab nach 8 Sekunden"

2. Gib das Folgende wiederholend nach 3 Sekunden aus:

> "Hallo Web-Programming-Lab nach 3 Sekunden"

3. Gibt das Folgende wiederholend nach jeder Sekunde aus aber nur 5 Mal:

> "Hallo Web-Programming-Lab"

... anschliessend gib "Done" aus und ermögliche einen Exit.

**Rahmenbedingung:** Kein setTimeout nutzen.

**Datei:** ./exercises/uebung-timers.js



# Node.js – Übungsaufgabe II (10')

- **Schreibe ein CommonJS Modul und ein ECMAScript Module, welches jeweils die Datei kopiert.**
- Kopiert ein File mit der Endung .copy
- **Rahmenbedingungen:** Mit Promises / async await gelöst!
- Interfaces:

```
const filecopy = require('./filecopy.cjs');  
filecopy('filecopy-commonjs.js'); // => filecopy-commonjs.js.copy
```

exercises/filecopy-commonjs.js

```
import {filecopy} from './filecopy.mjs';  
filecopy('filecopy-esm.mjs'); // => uebung-filecopy-esm.mjs
```

exercises/filecopy-esm.mjs

# Server-Side JavaScript

Http Module von Node.js & Express



# Node.js – Nodemon «reload, automatically»

- [Nodemon](#) ist ein Monitor um sämtliche Änderungen in deiner Node.js Applikation zu beobachten
- Bei Änderungen wird die Node.js App (Prozess) neu gestartet.
- Während der Entwicklung nodemon verwenden

`npm install nodemon --save-dev` oder `global npm install nodemon -g`

`npx nodemon myServer.js` ← nodemon als alias aufrufen ohne global installieren zu müssen

```
[nodemon] 1.19.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node demo-webserver.js`
Hello from server 123
```

- Demo: `npx nodemon examples/run-a-webserver.js`

# Node.js – Als Webserver

Mit `end()` wird die  
Antwort vom Server  
an den Client  
abgeschlossen

[http.IncomingMessage](#)

[http.ServerResponse](#)

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

const requestListener = (req, res) => {
  res.end('Hallo Web-Programming-Lab');
};

const server = http.createServer(requestListener);
server.listen(4566, () => console.log("Server started."));
```

[http.Server](#)

Weitere Infos: <https://nodejs.dev/the-nodejs-http-module>

# Node.js – HTTP-Module

Das HTTP Module von Node.js ist ein **low-level Module**.

> Es gibt z.B. keine (built-in) API um den Body auszulesen (nur via Parsing).

```
http.createServer((r, s) => {  
  console.log(r.method, r.url, r.headers);  
  let body = '';  
  r.on('data', (chunk) => {  
    body += chunk;  
  });  
  r.on('end', () => {  
    console.log(body);  
    s.write('OK');  
  });  
  s.end();  
}).listen(42646);
```

examples/webserver-read-body.js

Es gibt Web Frameworks resp. Middleware, welche die low-level Module wrappen und die **vereinfachte APIs** und **zusätzliche Funktionalitäten** zur Verfügung stellen (wie z.B. Express).

# Node.js – Express I

- [Express](#) ist ein Middleware-Web Framework on top vom Node.js HTTP Module
- “Express bietet eine Thin-Layer-Ebene mit grundlegenden Webanwendungsfunktionen, ohne die bekannten Node.js-Features zu überlagern.”
- Middleware bezeichnet Funktionen, welche Zugriff zu den Request (req) und Response (res) Objekten und den Zugriff auf die nächste Funktion im Request-Response Zyklus (via next Parameter) haben.
- Diese Funktionen können:
  - Code ausführen
  - Änderungen an den Request- und Response-Objekten durchführen
  - Den Request-/Response Zyklus beenden
  - Die nächste Middleware Funktion im Stack aufrufen
- [Express-Anwendungsgenerator](#)
- Alternativen: [koa.js](#), [nestjs](#), [sails.js](#)

# Node.js – Express II

- Eine Express-Anwendung kann die folgenden Middlewaretypen verwenden:
  - [Middleware auf Anwendungsebene](#) (z.B. Request-Interceptor)
  - [Middleware auf Routerebene](#) (via Router bestimmte Abläufe steuern)
  - [Middleware auf Fehlerbehebung](#)
  - [Integrierte Middleware](#) (Serving von statischen Assets)
  - [Middleware anderer Anbieter](#) (3<sup>rd</sup> Party Module wie z.B. cookie-parser)
- Demo (examples / expressjs / router / router-example.js)

# Node.js – Express und Template Engines

- Ohne [Template Engine](#) gibt es Strings zu konkatenieren, falls Content vom Server geteilt wird
- Beispiel Template Engines: Handlebars, Pug, ejs, jsx

```
const express = require('express');

const server = express();

server.set('view engine', 'ejs');

server.get('/', (req, res) => {
  res.render('index');
});

server.listen(4566, () => {
  console.log('Server is listening...');
});
```

```
<body>
  <h1>hello from about</h1>
  <span><%- Math.random() -%></span>
</body>
```

- **Demo:** exercises / expressjs / ejs / demo.js

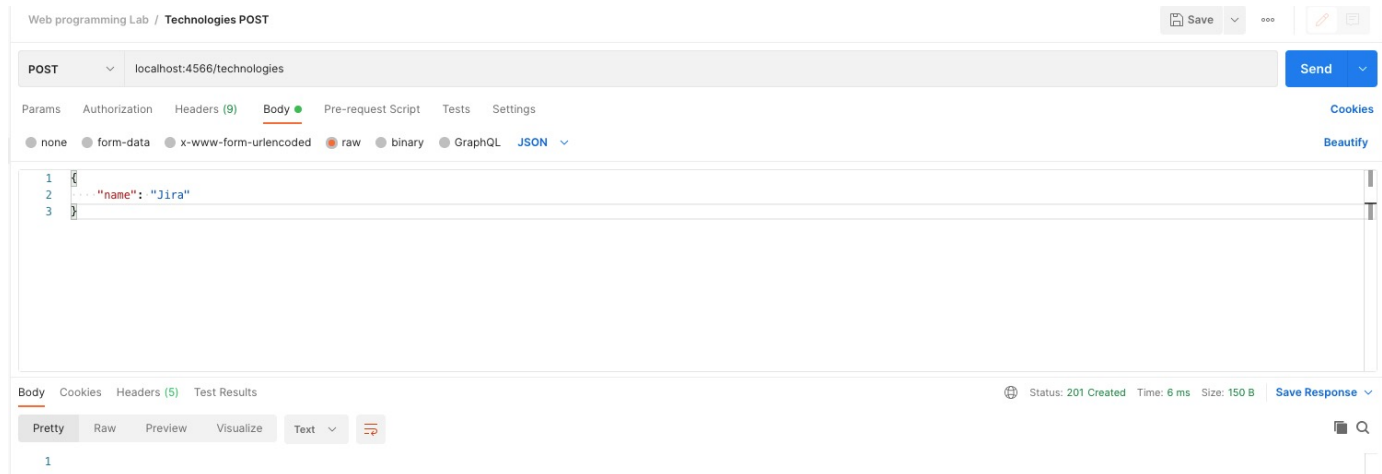


# Node.js – Übungsaufgabe (20')

Node.js App mit den folgenden HTTP APIs erstellen (Speichern Sie die Werte In-Memory, z.B. in einem Array):

- POST /technologies → speicherte eine Technologie in der Liste
- GET /technologies → gibt die Technologien Liste zurück
- GET /technologies/{id} → gibt das Detail einer Technologie zurück
- Verzeichnis exercises / tech-radar verwenden (benötigte Dependencies sind bereits deklariert)

Via [Postman](#) können die APIs getestet werden.



# Server-Side JavaScript

Persistieren mit MongoDB und dem MongoClient



# MongoDB

- Dokumentorientierte NoSQL-Datenbank (in der Programmiersprache C++ geschrieben)
- Datenbank kann mehrere Collections beinhalten
- Eine Collection enthält mehrere Dokumente (eine Collection ist mit einer Tabelle im RDBMS vergleichbar)
- MongoDB Client in NodeJS: `npm install mongodb --save`
- Lokale MongoDB Instanz entweder via Installation oder Docker
- Remote MongoDB Instanz via [cloud.mongodb.com](https://cloud.mongodb.com) (Free Sandbox Cluster)
- Demo: Cluster in [mongodb.com/cloud](https://mongodb.com/cloud)



mongoDB

# MongoDB – Connect & Dokument speichern

```
const MongoClient = require('mongodb').MongoClient;
const url = "mongodb+srv://<<db-user>>:<<pw>>@<<server-url>>/<<your-db>>";

(async function() {
  let client;

  try {
    client = await MongoClient.connect(url);
    console.log('Connected correctly to server');
    const db = client.db('db');
    let r = await db.collection('students').insertOne({text: 'hello web programming lab'});
    console.log(r);
  } catch(err) {
    console.log(err);
  } finally {
    client.close();
  }
})();
```

→ Kurze Demo (examples / mongodb / demo.js)

→ Collection Operationen vom MongoDB Node.js Driver:

<https://mongodb.github.io/node-mongodb-native/api-generated/collection.html>

# Node.js – Übungsaufgabe I (10')

Erstellen Sie sich einen MongoDB Cluster ([cloud.mongodb.com](https://cloud.mongodb.com)) und versuchen Sie ein Dokument zu speichern (mittels `examples/mongodb`).

## Vorgehen

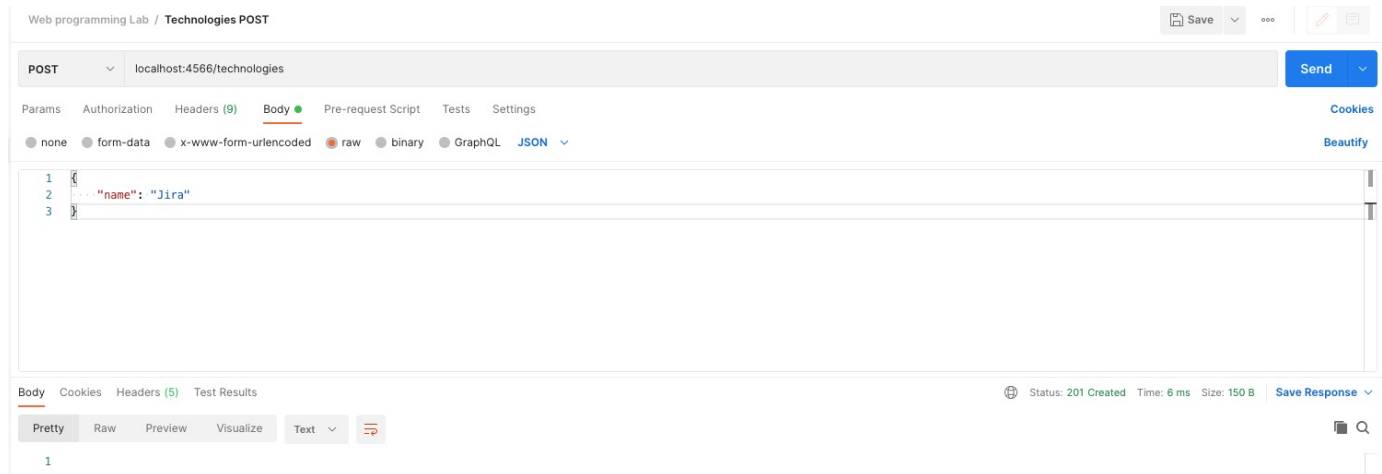
1. Registrieren Sie sich auf <https://cloud.mongodb.com> (free plan, shared db)
2. Erstellen Sie einen MongoDB Cluster
3. Erstellen Sie im Cluster eine Datenbank
4. Fügen Sie einen Benutzer hinzu
5. Fügen Sie ihre aktuelle IP temporär als Zugriffspunkt hinzu
6. Versuchen Sie ein Dokument mittels dem zur Verfügung gestellten Code (`monbgodb / demo.js`) zu speichern

# Node.js – Übungsaufgabe II (20')

Ergänzen Sie nun Ihre Node.js App mit Persistenz (der Fokus liegt auf dem Durchstich):

- POST /technologies → persistiert eine Technologie in der MongoDB (falls geklappt Return Code 201)
- GET /technologies → gibt die Technologien aus der DB zurück (falls geklappt Return Code 200)
- GET /technologies/{id} → gibt das Detail der selektierten Technologie zurück
- Verzeichnis exercises/mongodb verwenden (benötigte Dependencies sind bereits deklariert)

Via [Postman](#) können die APIs getestet werden.

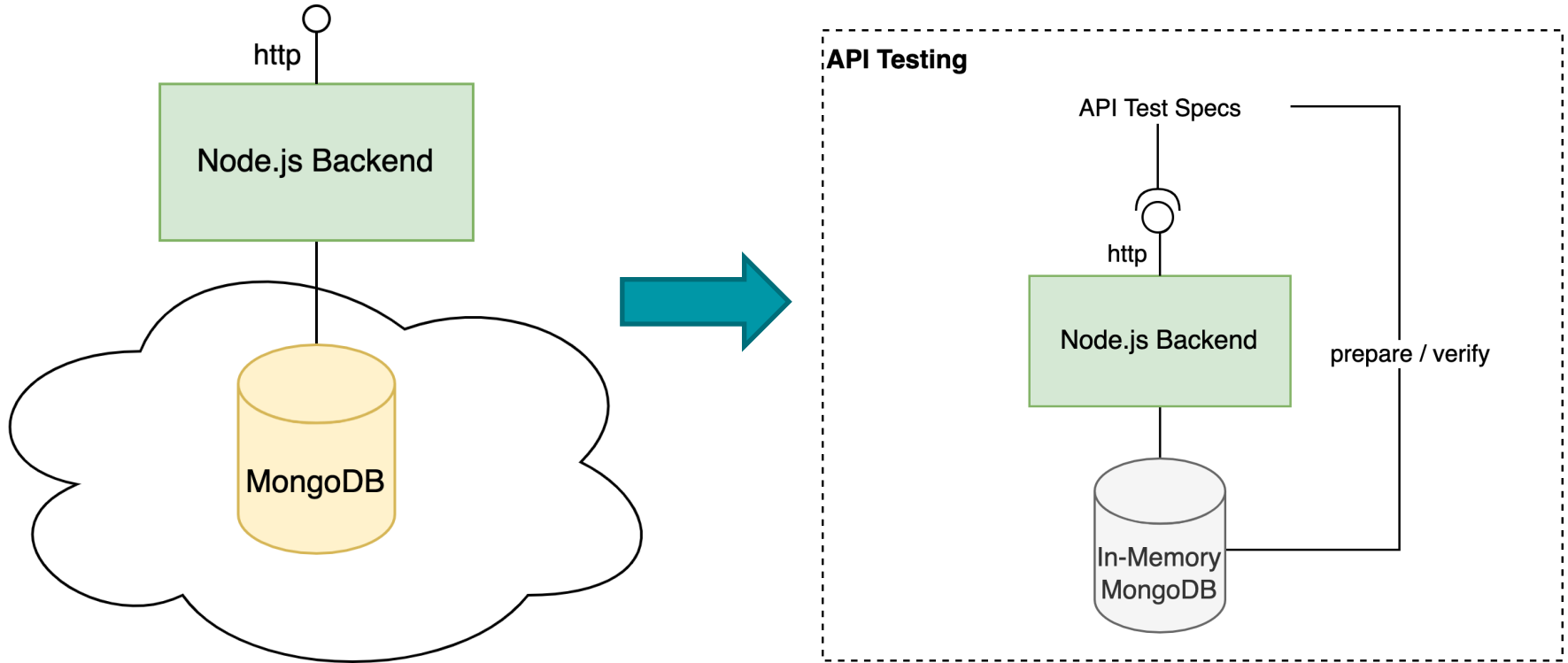


# Server-Side JavaScript

API Integration Testing



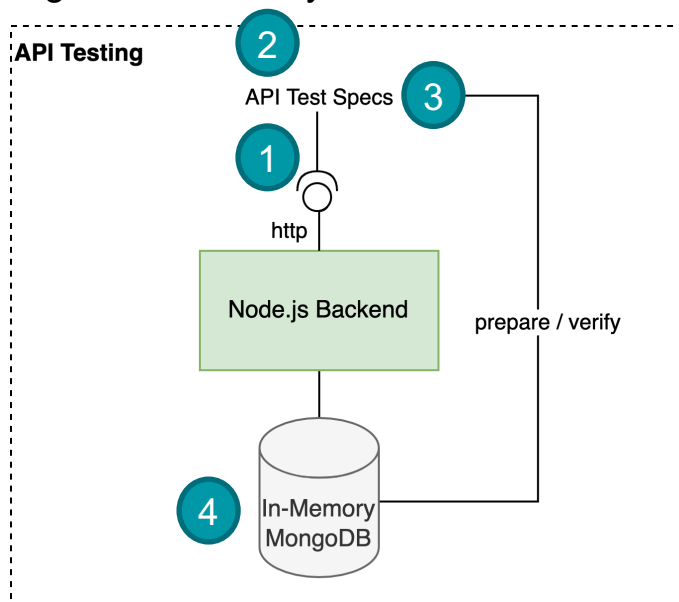
# API Integration Testing Scenario





# API Integration Testing Toolkit

1. [SuperAgent](#) ist eine leichtgewichtige clientseitige HTTP Request Library.
2. [SuperTest](#) ist eine SuperAgent basierte Bibliothek um HTTP Server zu testen mit einer Fluent API.
3. [Jest](#) ist ein JavaScript Testing Framework
4. [@shelf/jest-mongodb](#) ist ein MongoDB in-memory server für Jest



# API Integration Testing Toolkit

1. [SuperAgent](#) ist eine leichtgewichtige clientseitige HTTP Request Library.
2. [SuperTest](#) ist eine SuperAgent getriebene Bibliothek um HTTP Server zu testen mit einer Fluent API.
3. [Jest](#) ist ein JavaScript Testing Framework

```
const request = require('supertest');
const app = require('./app.js');

describe('GET /technologies', () => {
  3 it('responds with 200', (done) => {
    1 request(app)
      .get('/technologies')
      .expect(200, done); 2
  });
  ...
}
```

examples / api-testing / test.js

# API Integration Testing Toolkit

1. [SuperAgent](#) ist eine leichtgewichtige clientseitige HTTP Request Library.
2. [SuperTest](#) ist eine SuperAgent getriebene Bibliothek um HTTP Server zu testen mit einer Fluent API.
3. [Jest](#) ist ein JavaScript Testing Framework

```
it('responds with one technology', async () => {  
  const response = await request(app).get('/technologies');  
  
  expect(response.body).toBeDefined();  
  const technology = response.body[0];  
  expect(technology).toBeDefined();  
  expect(technology.name).toBe('ArgoCD');  
});
```

examples / api-testing / test.js

# API Integration Testing Toolkit

1. [SuperAgent](#) ist eine leichte
2. [SuperTest](#) ist eine Super
3. [Jest](#) ist ein JavaScript Test
4. [@shelf/jest-mongodb](#) ist

```
beforeAll(async () => {
  connection = await MongoClient.connect(global.__MONGO_URI__, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  });
  db = await connection.db('techradar');

  technologies = db.collection('technologies');
});
```

```
it('responds with one technology', async () => {
  const mockTechnology = {_id: 'some-user-id', name: 'ArgoCD'};
  4 await technologies.insertOne(mockTechnology);

  const response = await request(app).get('/technologies').expect(200);

  3 expect(response.body).toBeDefined();
  expect(response.body).toEqual([mockTechnology]);
  return response;
});
```

# Node.js – Übungsaufgabe (20')

Teste Sie nun Ihre Node.js App mit mindestens einem Test pro Verb mit dem vorgestellten API Integration Testing Toolkit – SuperTest und Jest.