

Web-basierte Systeme – Übung

06: Node.js, MongoDB und Aufgabe 5

Wintersemester 2026

Arne Vogel, Maxim Ritter von Onciul



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Übersicht

Node.js

 Überblick

 Non-blocking I/O

 Module

 Events

 Express

MongoDB

 mongoose

Virtualisierung & Docker

 OpenStack

Aufgabe 5

Node.js

Übersicht

Node.js

 Überblick

 Non-blocking I/O

 Module

 Events

 Express

MongoDB

 mongoose

Virtualisierung & Docker

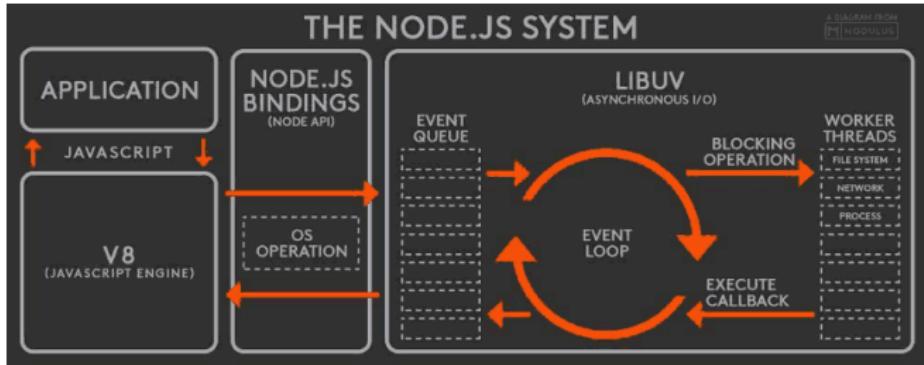
 OpenStack

Aufgabe 5

Node.js: Überblick

- Node.js ist eine **serverseitige** Laufzeitumgebung für JavaScript
- **Node.js = V8 + Event Loop + Async. I/O API**
- **npm**: Paketmanager für Node.js
- Alle I/O Operationen **non-blocking** möglich

Node.js: Event Loop



- Ein Thread führt JS des Entwicklers + Event Loop aus
- Anfragen werden direkt vom OS oder von Worker Threads bearbeitet

Quelle: <https://www.timcosta.io/the-node-js-event-loop/>

Node.js: non-blocking I/O (Übersicht)

- Alle I/O Operation haben **synchrone** und **asynchrone** Versionen
 - Synchron: Direkter Aufruf mit Argumenten
 - Asynchron: Zusätzliches **Callback** Argument
- Beispiel aus der Node.js API:
 - `fs.readFileSync(path[, options])`
 - `fs.readFile(path[, options], callback)`
- Callback Funktionen
 - Konvention: Error Objekt ist **immer** der erste Parameter
 - Bsp. für `readFile`: `function(err, data) {...}`
 - Je nach API Call mehr/andere Parameter möglich!
- **Promises**-basierte API
 - Gleiche API-Aufrufe, aber Promise wird returnt
 - z.B. `fs.promises`

Node.js: non-blocking I/O (Beispiel)

Synchrones Lesen einer Datei:

```
1 const fs = require('fs');
2 const data = fs.readFileSync('file.md'); // blocks here
3 console.log(data);
4 moreWork(); //will run after console.log
```

Asynchrones Lesen einer Datei:

```
1 const fs = require('fs');
2 fs.readFile('file.md', (err, data) => { // does NOT block here
3     if (err) throw err;
4     console.log(data);
5 });
6 moreWork(); //will run before console.log
7
```

Node.js Module

- Node.js bietet viele **Module**, welche unterschiedliche Funktionen implementieren, Beispiele (built-in):
 - Datei- und Netzwerkzugriff: fs und net
 - Netzwerkprotokolle: http, dns, ...
 - Kryptographie: crypto, tls
 - Kompression: zlib
 - ...
- Quasi **Libraries** für Node.js
- Hunderttausende zusätzliche Pakete in **npm**
- Module werden mit dem Schlüsselwort **require** eingebunden

```
1 var net = require('net');
2 var client = new net.Socket();
3 client.connect(1337, '127.0.0.1', function() {
4     client.write('Hello, server! Love, Client.');
5 });
```

Eigene Node.js Module

- Module **exportieren** JavaScript Objekte
- Zum Beispiel Funktionen:

```
1 // bar.js
2 module.exports.bar = function () {
3     console.log('bar!');
4 }
```

- Eigene Module können dann auch mit **require** verwendet werden:

```
1 // app.js
2 var lib = require('./bar.js');
3 lib.bar();
```

Events

- Mit Node.js lassen sich **Event-getriebene** (event-driven) Anwendungen bauen
- **Events** werden von **Emittern** ausgesendet
- **Listener** sind Funktionen, die bei Events ausgeführt werden
- Dazu müssen Listener mit der `.on()` Methode **registriert** werden

```
1 const EventEmitter = require('events');
2 class MyEmitter extends EventEmitter {}
3
4 const myEmitter = new MyEmitter();
5 myEmitter.on('event', () => {
6   console.log('an event occurred!');
7 });
8 myEmitter.emit('event');
```

Node.js Events: Beispiel Webserver

- Server hört auf Port 8081
- GET Requests werden mit dem String Hello World! beantwortet
- Alle anderen Requests werden ignoriert
- Dokumentation: <https://nodejs.org/api/http.html>

```
1 var http = require('http');
2 var server = http.createServer().listen(8081);
3
4 server.on('request', function (request, response){ //Behandlung des Events 'request'
5     if (request.method==='GET'){
6         response.writeHead(200, {'Content-Type': 'text/html'});
7         response.end("Hello World!\n");
8     }
9 });

```

Node.js: Express

- Meistgenutztes Web-Framework für Node.js
- Bereits als Abhängigkeit des Servers vorgegeben
 - Installierbar mit `npm install` (siehe Übungblatt)
- Antworten auf HTTP Requests werden über **Routen** definiert
- Das `app`-Objekt stellt `.get()`, `.post()`, ... zur Verfügung

```
1 var express = require('express');
2 var app = express();
3
4 // GET method route
5 app.get('/', function (req, res) {
6   res.send('GET request to the homepage')
7 })
8
9 // POST method route
10 app.post('/', function (req, res) {
11   res.send('POST request to the homepage')
12 })
```

Node.js: Express

- Pfade der Route können natürlich angepasst werden

```
1 app.get('/', function (req, res) {  
2   res.send('root')  
3 })  
4  
5 app.get('/about', function (req, res) {  
6   res.send('about')  
7 })  
8  
9 app.get('/users/:userId/', (req, res) => {  
10   res.send(req.params) // /user/vogel => req.params: { "userId": "vogel" }  
11 })
```

- Dabei sind auch reguläre Ausdrücke möglich:

- $/ab?cd \rightarrow acd$ und $abcd$
- $/ab+cd \rightarrow abcd, abbcd, abbbcd, \dots$
- $/ab*cd \rightarrow abcd, abxcd, abRANDOMcd, ab123cd, \dots$
- ...

Node.js: Express

- Zur Erhöhung der Übersichtlichkeit können die Definitionen von Routen in externe JavaScript-Dateien **ausgelagert** werden:

```
1 const index = require('./routes/index.js');
2 const service = require('./routes/service.js');
3
4
5 app.use('/', index);
6 app.use('/service', service);
```

Node.js: express-ws

- Das NPM Paket `express-ws` bietet Endpunkte von WebSockets für Express

```
1 ■ var router = express.Router();
2
3   router.ws('/ws', function(ws, req) {
4     ws.on('message', function(msg) {
5       ws.send("message received!");
6     });
7   });
8
9   app.use("/", router);
```

Node.js: Umgebungsvariablen

- Mit **Umgebungsvariablen** kann man einfach Parameter an Node.js Anwendungen weitergeben
- Die Variable wird in der Bash gesetzt:

```
1 $ TESTVAR="hello" nodejs app.js
```

- Und kann in Node.js ausgelesen werden:

```
1 const testvar = process.env.TESTVAR;
```

- Auch Standardwerte können definiert werden:

```
1 const testvar = process.env.TESTVAR || 'default_value';
```

MongoDB

Übersicht

Node.js

 Überblick

 Non-blocking I/O

 Module

 Events

 Express

MongoDB

 mongoose

Virtualisierung & Docker

 OpenStack

Aufgabe 5

- MongoDB ist eine quelloffene NoSQL-Datenbank
 - Abgeleitet vom engl. humongous, „gigantisch“
 - NoSQL: **nicht** relational!
 - Keine Festlegung auf Tabellenschemata
 - Stattdessen werden Daten als **Dokumente** abgelegt
- Ablage von **JSON-ähnlichen Objekten**
 - Erweiterung von JSON → Binary JSON (BSON)
 - Vorteile: geringere Speicherbelegung, schneller zu parsen
- Bekannt für **hohe Skalierbarkeit**
- Oft eingesetzt in Webanwendungen
 - MEAN-Stack: MongoDB, Express, Angular, Node.js
 - MERN-Stack: MongoDB, Express, React, Node.js

MongoDB: mongoose ODM

- **Mongoose** ist ein ODM Tool für MongoDB
- ODM: Object Document Mapper
 - Objekt im Code → Dokument in Datenbank
- **Schema:** Mapping von Objekten zu MongoDB
- Schemas werden zu **Models** kompiliert
- Models bieten verschiedene Methoden:
 - Neues Dokument erzeugen: `new Model()`
 - Neues Dokument speichern: `Model.save()`
 - Alle Vorkommen finden: `Model.find()`
 - Einzelnen Eintrag löschen: `Model.deleteOne()`
 - Einzelnen Eintrag ändern: `Model.updateOne()`

MongoDB: mongoose ODM (Beispiel)

```
1 const mongoose = require('mongoose');
2 mongoose.connect('mongodb://localhost:27017/test', {useNewUrlParser: true});
3
4 //new schema
5 var catSchema = new mongoose.Schema({
6   name: String
7 });
8
9
10 //new model:
11 const Cat = mongoose.model('Cat', catSchema);
12
13 //create new cats
14 const cat1 = new Cat({ name: 'Simon' });
15 const cat2 = new Cat({ name: 'Garfield' });
16
17 //save cats
18 cat1.save();
19 cat2.save().then(function(){
20   console.log("cat2 saved!");
21   Cat.find({}, function(err, cats){ //{} is an empty Conditions Object
22     console.log(cats)
23     mongoose.connection.close();
24     console.log("connection closed");
25   });
26});
```

Virtualisierung & Docker

Übersicht

Node.js

 Überblick

 Non-blocking I/O

 Module

 Events

 Express

MongoDB

 mongoose

Virtualisierung & Docker

 OpenStack

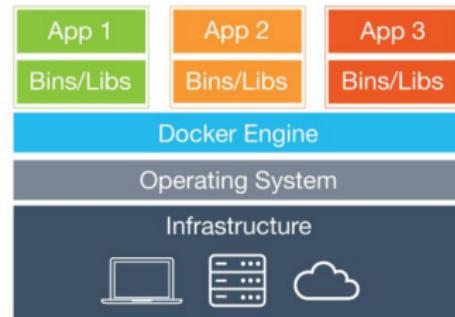
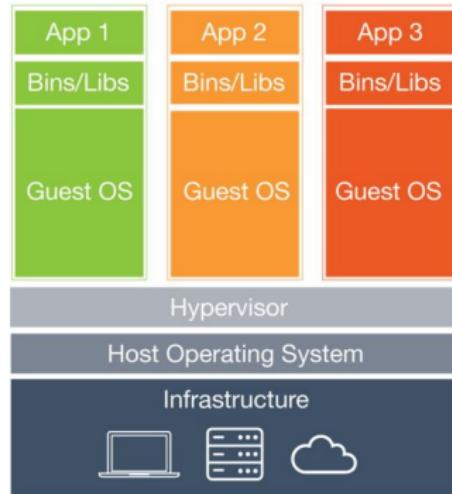
Aufgabe 5

- Schaffung virtueller Ressourcen auf Basis von physischen
 - Analog zu multitasking Betriebssystemen
 - Virtual Machine Monitor (VMM), Hypervisor
 - Virtuelle Maschine (VM)
- Ziele von Virtualisierung
 - Bessere Ausnutzung existierender Ressourcen
 - Erhöhung von Verlässlichkeit und Sicherheit
 - Höhere Skalierbarkeit von Systemen
 - Zentralisierung von Systemadministration
 - Betrieb von Altsystemen ohne alte Hardware
- Problem: Hoher Ressourcenbedarf → Containervirtualisierung

Containervirtualisierung

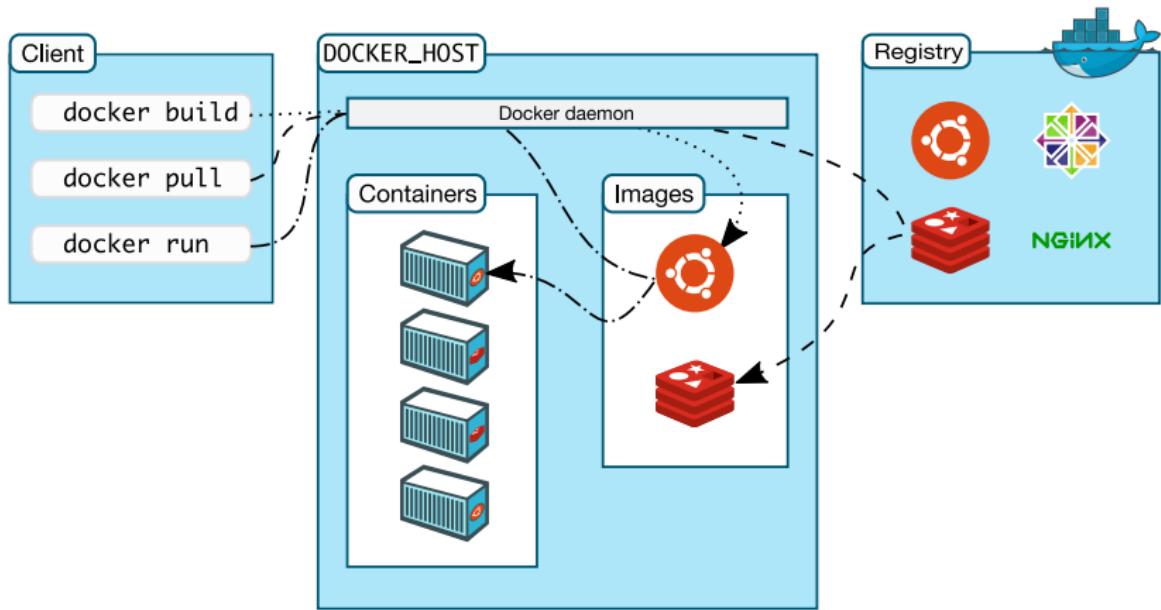
- Gleiche Ziele wie normale Virtualisierung
- Es werden **nur einige** Betriebsmittel isoliert
- Bspw. gemeinsame Nutzung des Betriebssystemkerns
- Bekannteste Implementierung: **Docker**
- **Problem:** Schwache Isolation ggü. normaler Virtualisierung

Virtualisierung vs. Docker



Quelle: <https://www.stratoscale.com/blog/compute/linux-containers-benefits-and-market-trends/>

Docker: Interface



Quelle: <https://docs.docker.com/engine/docker-overview/>

Docker als Deployment Tool

- Durch die **Kapselung** von Anwendung und Bibliotheken eignet sich Docker sehr gut als **Deployment Tool**
- Anwendungen können mit allen Abhängigkeiten gebündelt als Dockercontainer vertrieben werden
- Anwendungen mit **mehreren Komponenten** beinhalten dann mehrere Dockercontainer
- Die Kommunikation zwischen diesen kann mit docker-compose konfiguriert werden

Docker: docker-compose

- Mit docker-compose können Anwendungen bestehend aus **mehreren Docker Containern** definiert und ausgeführt werden
- Konfiguration über YAML-Datei docker-compose.yml:

```
1 services:
2   mongodb:
3     restart: always
4     image: mongo:latest
5     ports:
6       - "27017:27017" # this is dangerous! Dont do this in a production system
7
8   node:
9     restart: always
10    build: ./chat_server
11    environment:
12      - DB_URL= mongodb://mongodb/chatDB
13    links:
14      - mongodb
15    command: npm start
```

docker-compose: Wichtige Befehle

- `docker-compose up -d` Initial, um die Container zu instantiiieren.
- `docker-compose start` Pausierte Container wieder starten
- `docker-compose stop` Laufende Container pausieren
- `docker-compose down` Bestehende Container herunterfahren und löschen(!)
- `docker-compose kill` Bestehende Container gewaltsam herunterfahren und löschen(!)

OpenStack

- OpenStack ist eine quelloffene Software, die eine Architektur für das sogenannte Cloud-Computing zur Verfügung stellt
- U.a. schnelles erstellen von **virtuellen Maschinen** (VMs)
- Das RRZE betreibt eine Instanz unter <https://cc.rrze.de>
 - Zugriff: <https://www.idm.fau.de/go/application/cloud>
 - <https://www.anleitungen.rrze.fau.de/serverdienste/cc/compute-cloud-faq/>
- Ablauf für Aufgabe 5:
 - Start einer VM über das Webinterface
 - Zugriff per ssh
 - Kopieren von Dateien mit scp oder rsync

ssh

- ssh ist ein Programm zur sicheren Verbindung zu einem entfernten Rechner
- Allgemein: ssh <user>@<host>
- Beispiel: ssh vogel@i4wbs.cs.fau.de
- rsync
 - Dateien Kopieren
 - rsync *.pdf
vogel@i4wbs.cs.fau.de:/proj/i4wp/extern/lehre/ws25/wbs/

OpenStack: Webinterface

Project

API Access

Compute

Overview

Instances

Images

Key Pairs

Server Groups

Volumes

Container Infra

Network

Object Store

Identity

Overview

Compute

Limit Summary

Compute

Resource	Used	Total
Instances	Used 1 of 10	Used 1 of 10
VCPUs	Used 2 of 20	Used 2 of 20
RAM	Used 4GB of 50GB	Used 4GB of 50GB

Volume

Resource	Used	Total
Volumes	Used 4 of 10	Used 4 of 10
Volume Snapshots	Used 0 of 10	Used 0 of 10
Volume Storage	Used 80GB of 1000GB	Used 80GB of 1000GB

Network

Resource	Used	Total
Floating IPs	Allocated 0 of 1	Allocated 0 of 1
Security Groups	Used 2 of 10	Used 2 of 10
Security Group Rules	Used 9 of 100	Used 9 of 100
Networks	Used 0 of 100	Used 0 of 100

Aufgabe 5

Übersicht

Node.js

 Überblick

 Non-blocking I/O

 Module

 Events

 Express

MongoDB

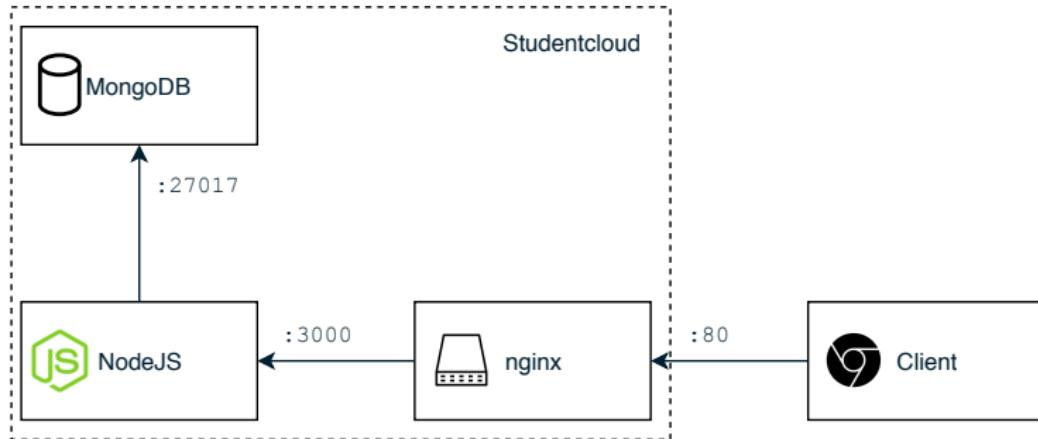
 mongoose

Virtualisierung & Docker

 OpenStack

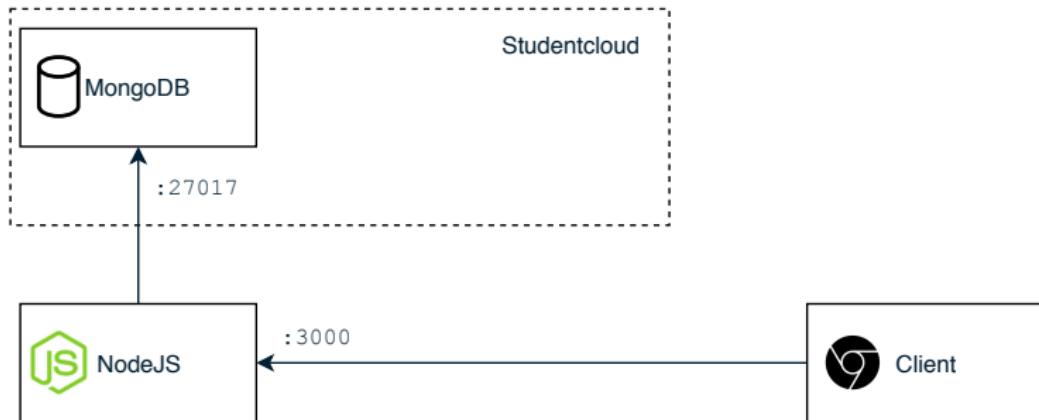
Aufgabe 5

Aufgabe 5: Endgültiges Deployment



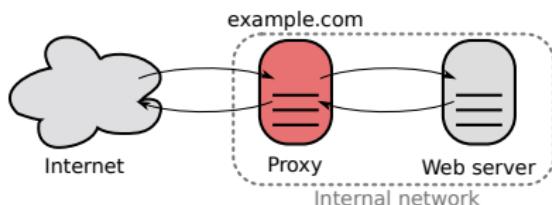
- `ssh -L 80:localhost:80 <user>@<ip>`
 - `http://localhost:80` im Browser
- Implementierung des Chatservers, der bisher vorgegeben war
- Komponenten: Node.js (mit Express), nginx, MongoDB
- Jede Komponente läuft in **Docker Container**
- Docker Container laufen auf **Virtueller Maschine (VM)**
- Eine VM pro Team in der **OpenStack**

Aufgabe 5: Deployment während Entwicklung



- `ssh -L 27017:localhost:27017 <user>@<ip>`
 - `mongodb://localhost:27017` für die Verbindung zur MongoDB
- Während der Entwicklung muss Servercode laufend angepasst werden
- Nur die MongoDB soll in der OpenStack ausgeführt werden

- nginx ist eine bekannte offene Webserver-Implementierung
- Hier: Einsatz als **Reverse Proxy**
- Gründe für einen Reverse Proxy:
 - Verstecken der Existenz/Charakteristik des eigentlichen Servers
 - Einsatz von Caching zur Entlastung der eigentlichen Server
 - Lastverteilung auf mehrere Server ohne Konfiguration des Clients
 - Hinzufügen von TLS Verschlüsselung
 - HTTP Authentifizierung
 - ...



Quelle: https://de.wikipedia.org/wiki/Reverse_Proxy

Quellen

- <http://expressjs.com/en/guide/routing.html>
- <https://www.npmjs.com/package/express-ws>
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling

Promise

```
1 const myPromise = new Promise((resolve, reject) => {
2     setTimeout(() => {
3         resolve('foo');
4     }, 300);
5 });
6 myPromise
7     .then(handleResolvedA)
8     .then(handleResolvedB)
9     .then(handleResolvedC)
10    .catch(handleRejectedAny);
```
