



# Ansätze zur Skalierung von Webapplikationen mit nodejs

2025, Michael Reichart  
GFU Cyrus AG, Köln

Wir bilden weiter.

# Was bedeutet Skalierung?

- + Skalierung bezeichnet die Fähigkeit eines Systems, wachsende Anforderungen zu bewältigen - sei es durch mehr Benutzer, höheres Datenvolumen oder komplexere Anfragen.
- + Das Ziel ist es, die Performance und Verfügbarkeit aufrechtzuerhalten, wenn die Last steigt.
- + Es gibt zwei **Hauptaspekte**:
  - + **Performance**-Skalierung: Das System bleibt schnell und responsiv
  - + **Kapazitäts**-Skalierung: Das System kann mehr gleichzeitige Anfragen verarbeiten

# Vertikale Skalierung (Scale Up)

- + Bei der vertikalen Skalierung wird die **Hardware** des Servers aufgerüstet - mehr CPU, RAM oder bessere Festplatten.
- + **Vorteile:**
  - Einfach umzusetzen (Server upgraden)
  - Keine Anpassungen am Code notwendig
  - Keine Komplexität durch Verteilung
- + **Nachteile:**
  - Physische Grenzen der Hardware
  - Teuer bei High-End-Hardware
  - Single Point of Failure bleibt bestehen
  - Downtime beim Upgrade
- + Anwendungsfall: Geeignet als erste Maßnahme oder wenn die Anwendung nicht für Verteilung ausgelegt ist.

+

# Horizontale Skalierung (Scale Out)

- + Hier werden **mehrere Server-Instanzen** parallel betrieben, die sich die Last teilen.
- + **Vorteile:**
  - Nahezu unbegrenzt skalierbar
  - Bessere Ausfallsicherheit (Redundanz)
  - Kostengünstiger (Commodity-Hardware)
  - Kein Downtime beim Skalieren
- + **Nachteile:**
  - Komplexere Architektur erforderlich
  - Load Balancing notwendig
  - Session-Management herausfordernd
  - Mehr Infrastruktur zu verwalten

# Horizontale Skalierung mit Cluster-Modulen für Multi-Core-Nutzung

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);
  // Worker für jeden CPU-Kern erstellen
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
    cluster.on('exit', (worker, code, signal) => {
      console.log(`Worker ${worker.process.pid} died`);
      cluster.fork(); // Neuen Worker starten
    });
  }
} else {
  // Worker erstellen HTTP-Server http.createServer((req, res)
  => {
    res.writeHead(200);
    res.end('Hello World\n');
  }).listen(8000);
  console.log(`Worker ${process.pid} started`);
}
```

# Load Balancing

- + Ein Load Balancer verteilt eingehende Anfragen auf mehrere Server-Instanzen.
- + Strategien:
  - + *Round Robin*: Anfragen werden reihum verteilt
  - + *Least Connections*: Server mit wenigsten aktiven Verbindungen bekommt die Anfrage
  - + *IP Hash*: Basierend auf Client-IP für Session-Persistenz
  - + *Weighted*: Server mit höherer Kapazität bekommen mehr Last
  - +

# Beispiel mit nginx als Load Balancer

```
upstream node_backend {
    least_conn;
    server localhost:3000;
    server localhost:3001;
    server localhost:3002;
    server localhost:3003;
}
server {
    listen 80;
    location / {
        proxy_pass http://node_backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header
        Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

# Container-Orchestrierung

## (z.B. Kubernetes, Docker, Docker-Swarm ...)

- + Container-Orchestrierung ist die **automatisierte Verwaltung, Skalierung und Vernetzung von Containern** über mehrere Server hinweg.
- + Statt Container manuell zu starten, zu überwachen und zu verwalten, übernimmt ein Orchestrierungs-System diese Aufgaben intelligent und automatisch.
- + Container packen deine Node.js-Anwendung zusammen mit allen Abhängigkeiten in eine isolierte, portable Einheit:
- + Vorteile von Containern:
  - Identisches Verhalten in Entwicklung, Test und Produktion
  - Schneller Start (Sekunden statt Minuten)
  - Weniger Ressourcenverbrauch als VMs
  - Einfaches Deployment und Rollback
  - Isolation zwischen Anwendungen

# Datenbank-Skalierung: Read Replicas

- + **Lesezugriffe** werden auf mehrere Datenbank-Kopien **verteilt**
- + **Schreibzugriffe** gehen auf den **Master**.

## 4. Datenbank-Skalierung

```
//const { Pool } = require('pg');
// Master für Schreibzugriffe
const masterPool = new Pool({
  host: 'master-db.example.com', database: 'myapp', max: 20
});
// Replicas für Lesezugriffe
const replicaPools =
[
  new Pool({
    host: 'replica1.example.com', database: 'myapp'
  }),
  new Pool({
    host: 'replica2.example.com', database: 'myapp'
  })
];
// Funktion zur Auswahl eines Replica
function getReplicaPool() {
  return replicaPools[Math.floor(Math.random() * replicaPools.length)];
}
// Verwendung async function getUser(id) {
  const pool = getReplicaPool();
  return await pool.query('SELECT * FROM users WHERE id = $1', [id]);
}
async function createUser(userData) {
  return await masterPool.query('INSERT INTO users (name, email) VALUES ($1, $2)',
    [userData.name, userData.email] );
}
```

**Sharding:** Daten werden auf mehrere Datenbanken aufgeteilt (z.B. nach User-ID-Bereichen).

```
// Einfaches Sharding-Beispiel
class DatabaseSharding {
  constructor(shards) {
    this.shards = shards; // Array von DB-Pools
  }
  getShardForUser(userId) {
    const shardIndex = userId % this.shards.length;
    return this.shards[shardIndex];
  }
  async getUserData(userId) {
    const shard = this.getShardForUser(userId);
    return await shard.query('SELECT * FROM users
                              WHERE id = $1', [userId]);
  }
}
```

# Caching-Strategien

- + Caching speichert häufig abgerufene Daten temporär im schnellen Speicher (z.B. RAM via Redis)
- + Sie werden nicht mehr jedes Mal neu aus der langsamen Datenbank geladen.
- + Die Datenbankabfragen werden reduziert, die Response-Zeit verkürzt und die Gesamtlast des Systems (deutlich) gesenkt.

# Caching Beispiel mit Redis

```
const redis = require('redis');
const client = redis.createClient();
async function getCachedData(key, fetchFunction, ttl = 3600) {
  // Erst im Cache suchen
  const cached = await client.get(key);
  if (cached) {
    return JSON.parse(cached);
  }
  // Bei Cache-Miss: Daten holen und cachen
  const data = await fetchFunction();
  await client.setEx(key, ttl, JSON.stringify(data));
  return data;
}
// Verwendung
app.get('/api/user/:id', async (req, res) => {
  const userId = req.params.id;
  const userData = await getCachedData(
    `user:${userId}`,
    () => database.getUserById(userId),
    1800 // 30 Minuten TTL );
  res.json(userData);
});
```

# Microservices-Architektur

- + Anstatt einer monolithischen Anwendung werden Funktionen in eigenständige Services aufgeteilt.
- + Services können unabhängig skaliert werden
- + Technologie-Stack pro Service wählbar
- + Einfacheres Deployment einzelner Komponenten
- + Bessere Fehler-Isolation

# Web-Service für eine Bestell-Komponente

```
// User Service (Port 3001)
const express = require('express');
const app = express();

app.get('/users/:id', async (req, res) => {
  const user = await userDatabase.findById(req.params.id);
  res.json(user);
});

app.listen(3001);

// Order Service (Port 3002)
const orderApp = express();

orderApp.post('/orders', async (req, res) => {
  // Kommunikation mit User Service
  const userResponse = await fetch(`http://user-service:3001/users/${req.body.userId}`);
  const user = await userResponse.json();

  // Order erstellen
  const order = await createOrder(user, req.body);
  res.json(order);
});

orderApp.listen(3002);
```

# Message Queues & Asynchrone Verarbeitung

- + Message Queues ermöglichen es, zeitaufwändige Aufgaben (wie E-Mail-Versand oder Bildverarbeitung) asynchron im Hintergrund zu verarbeiten.
- + Die Aufgabe wird in eine Warteschlange gelegt und später von separaten Worker-Prozessen abgearbeitet.
- + Der User bekommt eine Response, **während die eigentliche Arbeit parallel erledigt** wird.

# Beispiel mit RabbitMQ

```
const amqp = require('amqplib');

// Producer (API Server)
async function queueTask(taskData) {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();
  const queue = 'tasks';

  await channel.assertQueue(queue, { durable: true });
  channel.sendToQueue(queue, Buffer.from(JSON.stringify(taskData)), {
    persistent: true
  });

  console.log('Task queued:', taskData);
  await channel.close();
  await connection.close();
}

// Consumer (Worker)
async function processQueue() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();
  const queue = 'tasks';

  await channel.assertQueue(queue, { durable: true });
  channel.prefetch(1);

  channel.consume(queue, async (msg) => {
    const task = JSON.parse(msg.content.toString());
    console.log('Processing task:', task);

    // Task verarbeiten
    await performHeavyTask(task);

    channel.ack(msg);
  });
}
```

# Content Delivery Networks (CDN)

- + Statische Assets werden über geografisch verteilte Server ausgeliefert:
- + Reduzierte Latenz für Endnutzer
- + Entlastung der Origin-Server
- + DDoS-Schutz

# Konfiguration für CDN-Integration

```
app.use(express.static('public', {
  maxAge: '1d',
  setHeaders: (res, path) => {
    if (
      path.endsWith('.js')
      || path.endsWith('.css')) {
      res.setHeader(
        'Cache-Control',
        'public,
        max-age=31536000');
    }
  }
}));

// In HTML-Templates
// <script src="https://cdn.example.com/app.js"></script>
```

# Best Practises für Skalierungen

## + **Ein guter Anfang:**

- + Cluster-Modul für Multi-Core-Systeme
- + Caching für häufig abgerufene Daten
- + Datenbankabfragen mit Indizes optimieren

## + **Bei wachsender Last:**

- + Load Balancer
- + horizontale Skalierung mit Container-Orchestrierung (Kubernetes, Docker Swarm)
- + Database Read Replicas

## + **Für große Systeme:**

- + Microservices-Architektur
- + Message Queues für asynchrone Verarbeitung
- + Auto-Scaling basierend auf Metriken
- + Die richtige Skalierungsstrategie hängt von deinen spezifischen Anforderungen, Budget und technischen Möglichkeiten ab. Oft ist eine Kombination verschiedener Ansätze am effektivsten.
- +