Web-Anwendungen: MongoDB

Web-Engineering II

Prof. Dr. Sebastian von Klinski

MongoDB

MongoDB

- Dokumentenorientierte NoSQL-Datenbank
  - In 2018 als Open-Source freigegeben
  - Communitiy Edition kostenlos nutzbar
  - In C++ geschrieben
- Dokumente werden in „Collections“ gespeichert
- Häufig für JSON-Dokumente verwendet
- No-SQL
  - Keine Abfragesprache wie SQL, um Daten über mehrere Collections hinweg abzurufen
  - Stattdessen Suche in Dokumenten
  - Ist Nachteil und Vorteil zugleich
  - SQL-Datenbanken müssen für tabellenübergreifende Abfragen als monolithische Anwendung umgesetzt werden
  - MongoDB kann beliebig skaliert werden, weil es diese Beschränkung nicht gibt

MongoDB

- Abfragen
  - Abfrage von Feldern und ganzen Dokumententeilen
  - Verwendung von regulären Ausdrücken
  - Suche in spezifischen Feldern oder über das ganze Dokument hinweg
  - Auf Abfragen wird ein Cursor zurückgeliefert, mit dem durch die Ergebnisliste iteriert werden kann
- Schritte für Nutzung mit Node.js
  - Installation des Servers
  - Installation von Client-Modul/Driver: „npm install mongodb“
    - Installiert nur die Client-Software von MongoDB, nicht die Datenbank!
- Hinweise zur Installation von MongoDB
  - Unter Windows muss mongod.exe gestartet werden, nicht mongo.exe
  - Vor dem Starten muss das Verzeichnis „/data/db“ angelegt werden!
  - Wenn das Datenverzeichnis in der Root-Directory nicht angelegt ist, stoppt der Server mit dem Fehler-Code 100

MongoDB: Beispiel für Zugriff aus Node.js

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost/TestDB';
const client = new MongoClient(url, {useUnifiedTopology: true});

client.connect().then((client)=>{
    var db = client.db('TestDB')
    db.collection('Users').find().toArray(function (err, result) {
        if (err) throw err
                console.log(result);
        var cursor = db.collection('User').find();
        cursor.each(function(err,doc){
                console.log(doc);
        });
   })
})
```

```
Item 1
{
  _id: 5e5bc320ee0459255c5bba22,
  id: 1,
  userID: 'admin',
  userName: 'Manfred'
}
```

Datenbankverbindung

- Bei der Umsetzung von Datenbankanbindungen sollte die Connection gecacht werden
  - Öffnen von Datenbankverbindungen langsam und bindet Ressourcen
  - Pro Anwendung sollte Datenbankverbindung nur einmal geöffnet werden
- Wiederverwendung von geöffneter Verbindung
  - In Node.js werden Module nur einmal geladen!
  - Mit „require“ wird immer die gleiche Modulinstanz geladen
  - Daher: Öffnen der Datenbankverbindung in Modul auslagern
  - Datenbankmodul sollte Datenbank öffnen und Client-Objekt exportieren

Datenbankverbindung

- Zum Öffnen von Datenbank werden Angaben wie URL und Port benötigt
- Wenn Datenbank Authentifizierung erfordert, müssen auch User-ID und Passwort hinterlegt werden
- Diese Daten sollten nie im Programm-Code sein!
  - URL und Port können in der Entwicklungsphase bei den einzelnen Entwicklern unterschiedlich sein
  - Schneller Wechsel zwischen Entwicklungs-, Test- und Produktivumgebung muss möglich sein
  - Kritische Daten wie User-ID und Passwort sind im Programm-Code nicht sicher

Datenbank-Connection

```
const client = require("mongodb").MongoClient;
const config = require("../config");
let _db;
function initDb(callback) {
    if (_db) {
       return callback(null, _db);
    } client.connect(config.db.connectionString, config.db.connectionOptions, connected); function connected(err, db) {
        if (err) {
            return callback(err);
        }
        console.log("DB initialized - connected to: " + config.db.connectionString.split("@")[1]);
        _db = db;
        return callback(null, _db);
    }
}

function getDb() {
    return _db;
}
module.exports = {
    getDb,
    initDb
};
```

Quelle: https://itnext.io/how-to-share-a-single-database-connection-in-a-node-js-express-js-app-fcad4cbcb1e

Konfigurationen

Konfiguration

- Angaben wie URL und Port sollten in Konfigurationsdatei ausgelagert werden
  - Zum Laden von Konfigurationsdaten ist Nutzung von entsprechenden Modul empfehlenswert (zum Lesen der Konfigurationsdateien)
  - Es sollte möglich sein, beim Start der Anwendung die Konfigurationsdatei auszuwählen → schneller Wechsel zwischen verschiedenen Umgebungen (Entwicklung, Test, Produktiv, etc.)
  - Auswahl der Konfigurationsdatei könnte beispielsweise über Start-Skript in package.json oder über Konsole festgelegt werden
- Mehrere NPM-Module zum Verwalten von Konfigurationen verfügbar
- Beispiel: Modul config-npm sehr gängig

Konfiguration: config - npm

- Installation: ***npm install config***
- Laden und Zusammenfügen von Konfigurationsdateien
- Konfigurationsdateien sind hierarchisch organisiert
- Es können Standardwerte gesetzt werden, die dann für unterschiedliche Umgebungen überschrieben werden.
- Werte können auch überschrieben werden durch
  - Umgebungsvariablen
  - Kommandozeilenparameter
  - Externe Quellen (Datenbanken, Git-Repositories, etc.)

Konfiguration: config - npm

- In der Konfiguration können sowohl zentrale Aspekte wie Datenbankverbindung aber auch Anwendungskonfigurationen hinterlegt werden

```
{
  // Customer module configs
  "Customer": {
    "dbConfig": {
      "host": "localhost",
      "port": 5984,
      "dbName": "customers"
    },
    "credit": {
      "initialLimit": 100,
      // Set low for development
      "initialDays": 1
    }
  }
}
```

Angaben zur Datenbank

Angaben zur Konfiguration der Awendung (Kredit-Limit)

https://www.npmjs.com/package/config

Konfiguration: config - npm

- Konfigurationsdateien werden üblicherweise im Verzeichnis „./config“ gesucht
- Name der Konfigurationsdatei sollte sich an Umgebung orientieren (test, development, stage, production, etc.)

production.json

```
{
   "db": {
      "connectionString": "mongodb://localhost/TestDB“
      "connectionOptions": {
         "useNewUrlParser": true,
         "useUnifiedTopology": true
       }
   }
}
```

https://www.npmjs.com/package/config

Konfiguration: config - npm

- Zugriff auf Konfigurationsangaben

```
var config = require('config');

var connectionString = config.db.connectionString
```

- Falls nicht sicher ist, ob der Wert vorhanden ist

```
if (config.has('dbConfig')) {
  ...
}
```

https://www.npmjs.com/package/config

Konfiguration: config - npm

- Auswahl der Konfiguration durch das Setzen von Umgebungsvariablen (Linux)

```
$ export NODE_ENV=production
$ node my-app.js
```

- Durch Setzen der Variable wird automatisch ./config/production.json geladen
- Aggregieren von Konfigurationsdateien
  - Als Standardkonfiguration wird default.json geladen
  - Gibt es eine spezifische Konfiguration für die Umgebung (z.B. production.json), werden danach diese Werte gelesen und überschreiben die Werte von default.json

https://www.npmjs.com/package/config

Konfiguration: config - npm

- Zunächst werden die Werte von default.json gelesen.
- Anschließend werden die Werte aus production.json ergänzt oder überschrieben

default.json

```
{
    "db": {
        "connectionOptions": "{ useNewUrlParser: true, useUnifiedTopology: true }",
     }
}
```

production.json

```
{
    "db": {
        "connectionString": "mongodb://localhost/TestDB"
    }
}
```

https://www.npmjs.com/package/config

Konfiguration: config - npm

- Laden der Konfigurationsdaten

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

const connectionString = config.get('db.connectionString');
```

- Im Programmcode ist weder der Ort der Konfigurationsdatei, noch die Umgebung notwendig
- Wichtig für schnellen Wechsel zwischen den Umgebungen!

https://www.npmjs.com/package/config

Umgebungsvariablen

- Umgebungsvariablen sollten eingesetzt werden
  - Um lokale Einstellungen vorzunehmen (z.B. welche Umgebung verwende ich?)
  - Um sicherheitskritische Angaben zu hinterlegen (z.B. Passwörter)
- Passwörter und andere kritische Angaben sollten nicht im Programm-Code oder in Konfigurationsdateien sein
  - Im Programm-Code sind kritische Daten nicht sicher
  - Für unterschiedliche Entwickler können Angaben wie Passwörter auch unterschiedlich sein

https://www.npmjs.com/package/config

Konfiguration: Umgebungsvariablen setzen

- Über die Konsole (Linux)

```
$ export NODE_ENV=production
$ node my-app.js
```

- Auf der Kommandozeile (Linux)

```
NODE_ENV=stage node myapp.js
```

- Auf der Kommandozeile in Windows

```
$env:NODE_ENV="production"
```

- In Windows können die Umgebungsvariablen im entsprechenden Editor bearbeitet werden

https://www.npmjs.com/package/config

Konfiguration: Umgebungsvariablen setzen

- In package.json

```
"scripts": {
   "start": "node ./bin/www",
   "test": "set NODE_ENV=production&&  node ./bin/www"
 },
```

- Im Skriptcode bevor die Konfiguration geladen wird

```
…
process.env["NODE_CONFIG_DIR"] = __dirname + "/configDir/";
const config = require("config");
…
```

Anmerkung: In diesem Beispiel wird der Standardpfad zum Laden der Konfigurationsdateien geändert.

https://www.npmjs.com/package/config

Mongoose

Mongoose.js

- Zugriff auf Objekte in Datenbank über Datenbank-Client-Objekt in der Regel unkomfortabel
- Daher häufig Verwendung von OR Bridge (Objektrelationale Brücke) für Zugriff auf Datenbank
- Mongoose Bibliothek, um aus JavaScript auf MongoDB zuzugreifen
- Vereinfachung beim Anlegen, Speichern und Abrufen von Objekten in MongoDB
- Vergleichbar zu OR Bridge für relationale Datenbanken wie Hibernate

Quelle: https://mongoosejs.com/

Mongoose.js

- Schritte für Zugriff
  - Mongoose starten, indem Datenbank angebunden wird
  - Mit Mongoose Schema vom Objekt definieren
  - Mit Schema kann Modell angelegt werden (Modell steht für Dokument)
- Beispiel: Definieren und Speichern von Cat-Objekt

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

mongoose.connect('mongodb://localhost:27017/test', {useNewUrlParser: true,
useUnifiedTopology: true});

const Cat = mongoose.model('Cat', { name: String });

const kitty = new Cat({ name: 'Zildjian' });

kitty.save().then(() => console.log('meow'));
```

Quelle: https://mongoosejs.com/

Mongoose.js

- Installation

```
$ npm install mongoose
```

- Datenbankverbindung erstellen

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/TestDB', {useNewUrlParser: true});
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
  console.log(“Conntected”);
});
```

Quelle: https://mongoosejs.com/

Mongoose.js

- Schema erstellen

```
var userSchema = new mongoose.Schema({
    id: Number,
    userID: String,
    userName: String,
});

userSchema.methods.whoAmI = function () {
    var output = this.userID
      ? "My name is " + this.userName
      : "I don't have a name";
    console.log(output);
  }
```

Quelle: https://mongoosejs.com/

Mongoose.js

- Arrays

```
var studentSchema = new mongoose.Schema({
    …
    adressen: [AdresseSchema],
    …
});
```

Quelle: https://mongoosejs.com/

Mongoose.js

- Validatoren

```
var studentSchema = new mongoose.Schema({
    matrikelnummer:{
        type: String,
        required: true,
        unique: true,
        validate: {
            validator: (v: string) => {
                return /^s\d{5,6}$/.test(v);
            },
            message: ‘Matrikelnummer muss das folgende Format haben: s00000'
        }
    },
    userName: String,
});
```

Quelle: https://mongoosejs.com/

Mongoose.js

- Validatoren

```
var studentSchema = new mongoose.Schema({
    …
    nachname: {
        type: String,
        required: true,
        minlength: 2,
        maxlength: 64
    },
    geburtstag: {
        type: Date,
        required: true,
        min: '1910-7-22',
        max: Date.now()
    },
   …
});
```

Quelle: https://mongoosejs.com/

Mongoose.js

- Standardwerte

```
var studentSchema = new mongoose.Schema({
    …
   registered: {
        type: Date,
        default: Date.now
    },
    …
});
```

Quelle: https://mongoosejs.com/

Mongoose.js

- Anlegen

```
var udoUser = new User({ id: 1, userID: 'udo',userName: 'Udo Mustermann' });

udoUser.save(function (err, result) {
    if (err) return console.error(err);
    result.whoAmI();
  });
```

Mongoose.js

- Für Zugriff auf Objekte wird Modell benötigt
- Modell hat entsprechende Suchfunktionen
- Suche nach allen

```
var User = mongoose.model('User', UserSchema);

User.find(function (err, users) {
  if (err) return console.error(err);
  console.log(users);
})
```

- Suche nach Person, die john heißt und mindestens 18 ist

```
Person.find({ name: 'john', age: { $gte: 18 }});
```

- Gleiche Suche mit Callback

```
MyModel.find({ name: 'john', age: { $gte: 18 }}, function (err, docs) {});
```

Quelle: https://mongoosejs.com/docs/api.html#model_Model.find

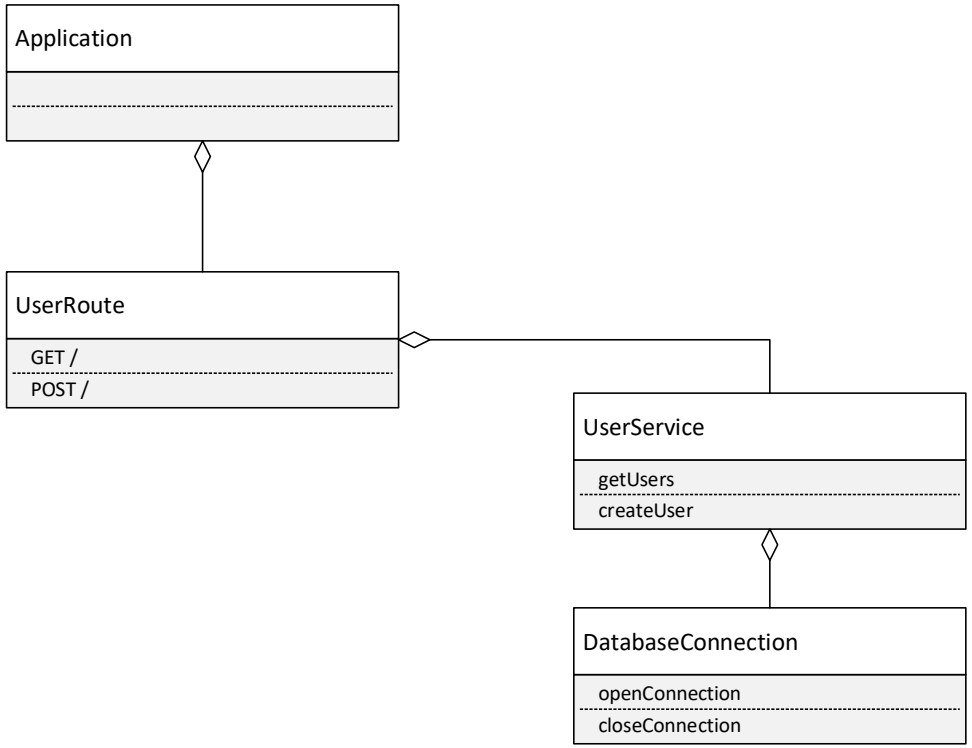
Empfehlung für Anwendungsstruktur

Empfehlungen

- Für REST-Services mit Datenbank ist 3-Schicht-Architektur empfehlenswert
  - Schicht 1: Controller-Schicht mit Express-Routen
  - Schicht 2: Service-Layer mit Anwendungslogik
  - Schicht 3: Persistenz-Layer mit Datenbankzugang (z.B. über Mongoose)
- Ziele dieser Architektur
  - Einfache funktionale Skalierung
  - Austauschbarkeit der Schichten
  - Wiederverwendbarkeit
  - Klare funktionale Modularisierung → bessere Software-Qualität, einfache Einarbeitung

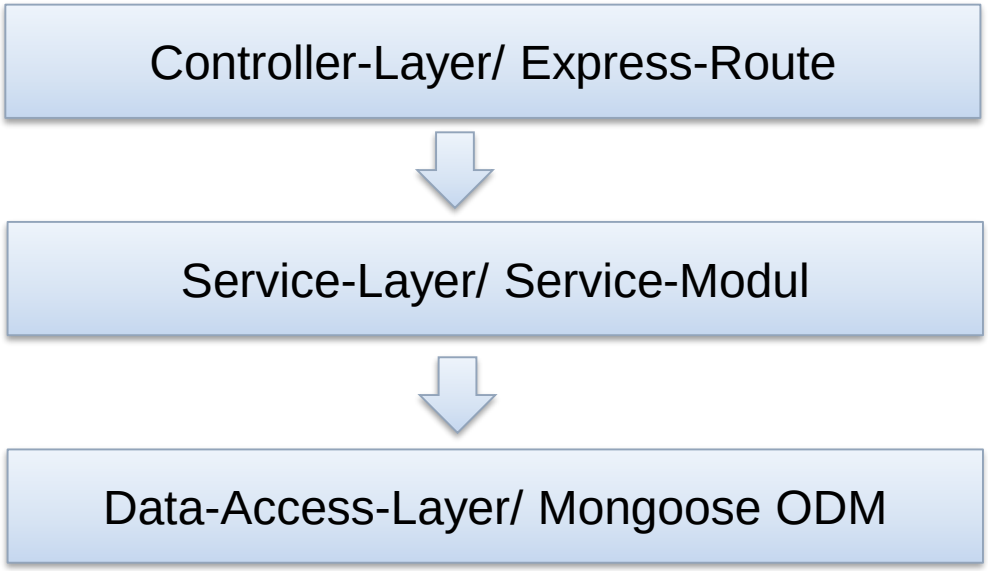
Anwendungsstruktur

- 3-Schicht-Architektur (Route, Service, Datenschicht)
- Application lädt Routen, Routen greifen auf Services zu, Services greifen auf Persistenz-Layer zu



3-Schichtarchitektur

- Jede Schicht kennt nur die Schicht darunter
- Sie sollten jeweils so wenig wie möglich von der Schicht darunter kennen



3-Schicht-Architektur: Aufgabenverteilung

Route (häufig auch bezeichnet Controller)

- Kapselt Zugriff des Web-Servers (eingehende HTTP-Anfragen, schreiben der HTTP-Responses)
- Lesen Anfrage (z.B. Json-Dokumente, URL-Parameter, etc.)
- Geben Anfragen als Objekte an Service-Layer weiter
- Erhalten Antworten vom Service-Layer als Objekte und überführen diese in HTTP-Response-Objekte (Json, etc.)
- Weitere Aufgaben
  - Authentifizierung
  - Token-Handling
- Hinweise
  - Keine Geschäftslogik in Routen!

3-Schicht-Architektur: Aufgabenverteilung

Service

- Kapselt Anwendungslogik
- Zugriff auf Datenschicht/ Datenbank
- Zugriff auf andere Drittanbieter-Services (falls vorhanden)
- Verknüpfungen zwischen Entitäten, Nutzen der anderen Services
- Autorisierung (Prüfung, ob Rechte für Anfrage gegeben sind)
- Weitere Hinweise
  - Sollte keine Request- oder Response-Objekte oder andere HTTP-Aspekte kennen
  - Sollte keine Details des Persistierens oder der Suchen kennen (Art der Datenbank, Suchfunktionen, etc.)
  - Wechsel des Persistenz-Layers muss möglich bleiben

3-Schicht-Architektur: Aufgabenverteilung

Datenschicht/ Persistenz-Layer

- Kapselt Datenbankanbindung und Art des Speicherns (relationale DB, No-SQL-DB, etc.)
- Validierung von Objekten
- Weitere Hinweise
  - Häufig nicht eindeutig zu beantworten ist, wo die Definition der Objekte ist
  - Kann sowohl im Service- als auch Persistenz-Layer sein
  - Heutzutage eher im Persistenz-Layer angesiedelt
  - Mit Mongoose sind Objekte der Persistenz-Layer, da Modell-Objekte CRUD- und Suchfunktionen umgesetzen

Mongoose-Persistenz-Layer

- Mit Mongoose besteht Persistenz-Layer aus den Objektmodellen
- Modul mit Schema und Model
- Nur Modell muss exportiert werden

```
var mongoose = require('mongoose');
…
const UserSchema = new mongoose.Schema({
    id: Number,
    userID: { type: String, unique: true },
    userName: String,
    email: String,
    password: String,
…
const User =  mongoose.model("User", UserSchema);
module.exports = User;
```

Service-Layer

- Service-Layer lädt relevante Objektmodelle und nutzt diese zum Zugriff auf Datenbank

```
…
const User = require("./UserModel");

function getUsers(callback) {

    permission = accessControl.can(requestingUserID).readAny(access.CLASS_USER);
    if (!permission) {
        logger.err("Have no right to read user data");
        return callback("Permission denied", null);
    }

    User.find(function (err, users) {
        if (err) {
            return callback(err, null);
        }
        else {
            return callback(null, users);
        }
    })
}
…
```

Controller-Layer/ Routen

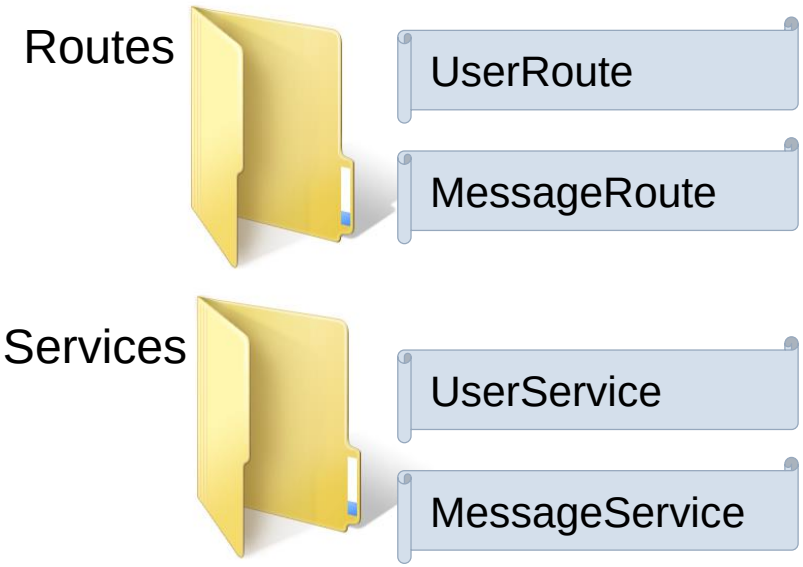
- Route lädt den Service und leitet Anfragen weiter

```
…
var userService = require("./UserService");
var AuthenticationUtils = require('../util/AuthenticationUtils');

router.get('/', AuthenticationUtils.isAuthenticated, function (req, res, next) {
  userService.getUsers(function (err, result) {
    if (result) {
      res.send(Object.values(result));
    }
    else {
      res.send(“Error retrieving users");
    }
  }
  );
});
…
```

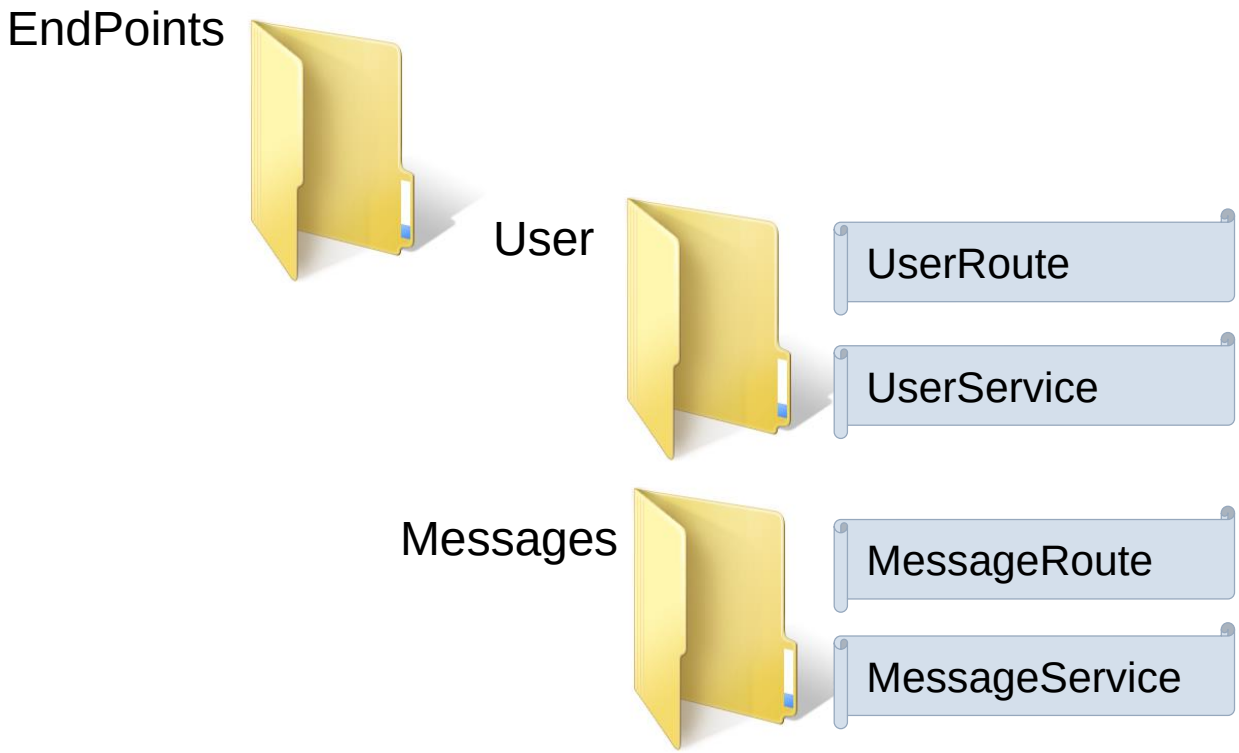
Anwendungsstruktur:

- Häufig werden Verzeichnisse/ Module nach der Art der Datei strukturiert
- Bei großen Projekten nicht gut:
  - Änderungen meist themenorientiert (z.B. Erweiterung der Message-Funktionen, Anpassen der User-Funktionen, etc.)
  - Änderungen erstecken sich dann immer auf zahlreiche Ordner
  - Bei großen Projekten schnell unübersichtlich, Merge-Konflikte wahrscheinlich



Anwendungsstruktur: besser

- Dateien/ Module sind inhaltlich gruppiert



- Hinweis: Routen werden oft als Endpoints bezeichnet (entspricht einer URL mit dem ein Client-System interagiert)

Anwendungsstruktur

- Verzeichnisstruktur sollte Zuständigkeit der Entwickler/innen widerspiegeln
  - Häufig themenorientierte Zuständigkeit (nach Endpoint, Objekten, etc.) → themenorientierte Verzeichnisse
- Zentrale Konfigurationsdatei verwenden
  - Festlegungen von Datenbank-Adresse, Server-Port, etc.
- Ergänzen der Konfiguration über Umgebungsvariablen
  - Z.B. Angabe von Produktiv- oder Entwicklungssystem, Credentials, etc.