WEBLAB Projekt: Synthesizer-Blog

Christopher James Christensen, Fabian Meyer

FS 2018

1. Einleitung

Als Projektarbeit im Fach WEBLAB haben wir uns einen Blog vorgestellt, der mit einem JavaScript-Stack implementiert wird. Technisch gesehen identisch mit einem Foto- oder Reiseblog, wollten wir aber einen Synthesizer-Blog realisieren, da wir uns beide für Synthesizer interessieren.

Die Wahl der Technologie fiel dabei auf Vue.js, ein neues, progressives, OpenSource-JavaScript-Frontend-Framework, dass von Evan You, einem ehemaligen Mitarbeiter von Angular (Google) entwickelt wurde und wird. JavaScript-Stack bedeutet, dass sowohl das Frontend, als auch das Backend mit JavaScript-Technologien realisiert wird. Unser Stack umfasst MongoDB, Express, Vue und Node (MEVN-Stack).

Da Christopher bereits als Backend-Entwickler in einer Firma arbeitet, hat er die Arbeiten des Backends übernommen:

- Server aufsetzen und konfigurieren (Node.js)
- MongoDB Datenbank aufsetzen und konfigurieren
- Middleware aufsetzen und konfigurieren (Express, Axios, etc.)
- REST-Methoden aufsetzen (get, post, put, delete)
- User Authentication aufsetzen und konfigurieren (custom)

Da Fabian als Frontend-Entwickler in einer Firma arbeitet, hat er die Arbeiten des Frontends übernommen:

- Design der Prototypen (von Hand und mit Software)
- Umsetzen des Prototyps
- Styling
- Testing der Umsetzung
- Dokumentation des Projekts

Als Startpunkt für unsere Applikation diente folgender Blog-post, welcher das Aufsetzen eines MEVN-Stacks beschreibt: Aufsetzen eines MEVN-Stacks (link)

2. Requirements

Vorgegebene Anforderungen

- 3 - 5 Seiten Dokumentation
- Vernünftige Architektur
- Full-Stack
- Testing
  - Vue: unit-testing
- Deployment
  - Vue: deployment

Projektbeschreibung

- Blog:
  - Blogposts zum Thema Synthesizer
  - Suche nach Themen
  - Userkontos
- Social-Media-Features (optional)
  - Likes

Stakeholder

- Synthesizer-Fans (Amateur bis Pro)
- Musiker
- Blogger

Muss-Features

- CRUD:
  - create
  - read
  - update
  - delete
- Alle Blogeinträge anzeigen
- Nach Blogeinträgen suchen
- Nach Blogeinträgen filtern
- Blogeinträge lesen
- Konto erstellen

- Blogeintrag erfassen und veröffentlichen
  - Titel setzen
  - Bild hochladen (optional)
  - Text erfassen (vielleicht auch mit Markdown)

Optionale Features (werden evt. nach Projektabgabe noch realisiert)

- Social-Media-Features:
  - Beiträge anderer User kommmentieren
  - Liken
  - Reposten
- Soundcloud API
- Einbinden von GIF's, Emoji's etc.
- Authentifizierung mit OAuth
- Bilder und Videos hochladen, bzw. einlinken

3. Systemspezifikation

Technologien

Stack

Wir bauen unsere Applikation auf einem MEVN Stack. Ein MEVN Stack besteht aus den folgenden Technologien:

- **M**ongoDB: Database-Backend
- **E**xpress.js: Middleware
- **V**ue.js: JavaScript-Frontwend
- **N**ode.js: JS-Runtime-Backend

Testing

Wir verfolgen das "Test First"-Prinzip und verwenden die folgenden Technologien, um unsere Applikation zu testen:

- Komponenten- / Unit-Tests: KarmaJS

Weitere Technologien

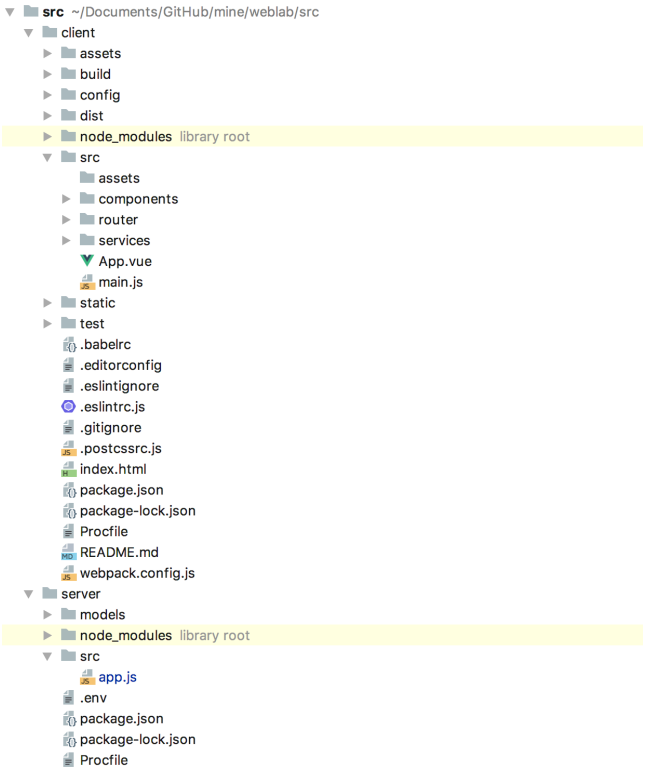
- Axios: Verbindung zwischen Client und Server herstellen.
- JsonWebToken: Token-Based-Authentikation auf dem Server
- Cors: Cross-Origin HTTP Requests

- Express Session: Behandelt User-Sessions (express-session)
- BCrypt: Salten und Hashen von Passwörter (bcrypt)
- Connect Mongo: Verbindet Mongo mit Session (connect-mongo)
- Mongoose: Object-Modeling der MongoDB
- SASS: CSS-Präprozessor
- Figma: Sketching- / prototyping-Werkzeug

Allgemeine Struktur

Dependencies

Alle Dependencies werden jeweils in der Client-Applikation und Server-Applikation in einem eigenen `package.json` verwaltet. Die Filestruktur sieht für das Verständnis so aus:



Versionierung

Wir verwenden Git mit Github, um die Zusammenarbeit im Team zu vereinfachen und als Versionierung (https://github.com/christopherchristensen/weblab).

Client

Der Client hat eine `main.js` -Datei als Ausgangspunkt. Dieser lädt die Haupt-Vue-Componente plus den Router. Danach werden alle weiteren Vue-Componenten über den Router geladen:

```
// (router/index.js)
import EditPost from '@/components/EditPost'

Vue.use(Router)

export default new Router({
  mode: 'history',
  routes: [
    {
      path: '/',
      name: 'Posts',
      component: Posts
    },
    {...
```

Die einzelnen Vue-Componenten verfügen über spezifische Services, welche sich mit der axios Api() verbinden. Diese Api sendet alle Requests an den Server.

Server

Mit der Express-Framework werden die Requests empfangen und abgehandelt. Der Server kommuniziert dann meistens mit unserer MongoDB, welcher mit `mongoose` modelliert wurde.

```
// (src/app.js)
app.use(morgan('combined'))
mongoose.connect('mongodb://localhost:27017/synthstagram');

db = mongoose.connection;
db.on('error', console.error.bind(console, "connection error"));
db.once('open', function(callback){
    console.log("Connection Succeeded");
});
```

Morgan wird verwendet, um Request-Informationen zu loggen. Weiter Requests vom Server werden mit Express.js abgehandelt. Hier ein Beispiel wie eine Middleware, um Requests vom Client abzufangen, erstellt wird.

```
const express = require('express')
const bodyParser = require('body-parser')
const cors = require('cors')
const morgan = require('morgan')
const app = express()

app.use(morgan('combined'))
app.use(bodyParser.json())
app.use(cors())

app.listen(process.env.PORT || 8081)
```

Body-Parser wird verwendet, damit die Form-Data als `req.body` verfügbar wird. Cors wird verwendet, um die cross-origin HTTP-Requests zu verwalten. Unten ein Beispiel eines GET-Requests vom Client, um alle Blogposts zu holen.

```
/*
 * Gets all posts and sort by publish date
 */
app.get('/posts', (req, res) => {
    Post.find({}, 'title text publish_date user_id username', function (error, posts
        if (error) { console.log(error) }
        res.send({
            posts: posts
        })
    }).sort({publish_date: -1}); // Last entry first
});
```

Datenbank

Unsere MongoDB-Datenbank kann durch die Express und Axios API angesprochen werden. Die Datenbank wird als folgendes Schema definiert:

UserSchema

Dieses Schema definiert, wie alle relevanten Informationen von jedem individuell registrierten User gespeichert werden sollte.

```
var UserSchema = new Schema({
    email: {
        type: String,
        unique: true,
        required: true,
        trim: true,
        lowercase: true// trim whitespaces
    },
    username: {
        type: String,
        unique: true,
        required: true,
        trim: true
    },
    password: {
        type: String,
        required: true
    },
    passwordResetKey: String,
    passwordKeyExpires: Number,
    createdAt: {
        type: Date,
        required: false
    },
    updatedAt: {
        type: Number,
        required: false
    }
}, { runSettersOnQuery: true })
```

Das Schema bekommt noch einige Pre-Hook's über, um zum Beispiel das createdAt-Datum zu setzen. Dazu erhält sie noch eine statische Methode, um bei der Authentifizierung das mitgegebene Passwort mit dem gehashedten Passwort zu vergleichen.

Authentikation

Wir verwenden für die Authentikation das Package `jsonwebtoken`. Der Server erstellt beim registrieren oder beim einloggen eines Users einen Token aus einem token_secret (Geheimniss) und einem payload (mit User-Informationen).

```
// 1. PAYLOAD
const payload = {
    email: user.email,
    username: user.username,
    id: user._id
}

// 2. PAYLOAD MIT SECRET SIGNIEREN
let token = jwt.sign(payload, token_secret);

// 3. TOKEN ZUM CLIENT SENDEN
res.status(200).send({
    token,
    username: user.username,
});
```

Dieser Token wird dann dem Client zurückgesendet. Der Client speichert den Token dann in seinem LocalStorage und sendet ihn bei jedem Request mit. So kann der Server überprüft den Token mit der folgenden Middleware:

```
// 4. JEDER REQUEST DER IM CODE NACH DIESER MIDDLEWARE KOMMT BRAUCHT AUTHENTICATION
app.use((req, res, next) => {
    var token = req.headers['x-access-token'] || req.body.token || req.params.token
    if (token) {
        jwt.verify(token, token_secret, function (err, decoded) {
            if (!err) {
                req.decoded = decoded;
                next();
            } else {
                res.status(403).send('Invalid token');
            }
        })
    } else {
        res.status(403).send('Authorization failed! Please provide a valid token');
    }
})
```

Authentifizierung

Die Authentifizierung wird mithilfe der Bibliothek BCrypt realisiert, die die Passwörter der User salted und hasht und so die Speicherung als Klartext verhindert. Wir benutzen auch Mongoose, da mithilfe von Mongoose das Datenbank-Schema definiert wird und bereits dort einige Sicherheitsaspekte, wie die Verschlüsselung von Passwörter angeordnet werden können.

Es wurde nach der folgenden Vorgehensweise vorgegangen: password authentication.

```
UserSchema.pre(save, function(next) {
    var user = this;

    // only hash the password if it has been modified (or is new)
    if (!user.isModified('password')) return next();

    // generate a salt
    bcrypt.genSalt(SALT_WORK_FACTOR, function(err, salt) {
        if (err) return next(err);

        // hash the password using our new salt
        bcrypt.hash(user.password, salt, function(err, hash) {
            if (err) return next(err);

            // override the cleartext password with the hashed one
            user.password = hash;
            next();
        });
    });

});

UserSchema.methods.comparePassword = function(candidatePassword, cb) {
    bcrypt.compare(candidatePassword, this.password, function(err, isMatch) {
        if (err) return cb(err);
        cb(null, isMatch);
    });
};
```

Um die Übersicht über alle User-Sessions zu behalten, verwenden wir die Bibliothek "Express Session". Mithilfe dieser Middleware kann das Login und Logout einer App realisiert werden.

```
//use sessions for tracking logins
app.use(session({
  secret: 'our super secret',
  resave: true,
  saveUninitialized: false
}));
```

Sobald der User auf dem Server eingeloggt ist, wird auf der Client-Seite ein Token im lokalen Speicher des Browsers gespeichert. Anhand dieses Tokens erkennt Vue, welche Komponenten gerendert werden sollen und welche nicht.

Funktionsweise des Vue-Frontends

Direktiven

Wie Angular auch, dient Vue der Erstellung von Single-Page-Webanwendungen (SPA). Das Verhalten einer solchen SPA erfolgt durch sogenannte Vue Direktiven. Dies sind spezielle Tokens (in Form von HTML-Elementen) in einer Webseite, die den DOM oder einzelne DOM-Elemente manipulieren können. Als einfaches Beispiel wird hier die Funktionsweise von v-text gezeigt:

```
<div v-text="'hello ' + user.firstName + ' ' + user.lastName"></div>
```

Dies führt dazu, dass der Inhalt des Property "firstName" des Objekts "user" im HTML gerendert wird (String-Interpolation). Mithilfe der Vue-Direktiven kann sämtliches Verhalten einer Webseite oder Webapplikation gesteuert werden, so auch die Navigation auf einer Seite:

```
<a v-bind:href="'/posts/' + post._id">{{ post.title }}</a>
```

Die Direktive v-bind führt durch Klicken dazu, dass der User auf eine neue Seite (Route) weitergeleitet wird.

Single file components

Bei Vue ist es möglich, einen gesamten Komponenten mit nur einem einzelnen File zu erzeugen. Diese Files haben In jedem dieser Files ist also sowohl HTML-, JavaScript-, als auch CSS-Code vorhanden.