
Laborprotokoll

Distributed Computing GK9.3

"Cloud-Datenmanagement"

Systemtechnik Labor
5BHIT 2017/18

Nicolaus Rotter

Note:
Betreuer: BORM

Version 0.2
Begonnen am 24. Januar 2018
Beendet am 24. Januar 2018

Inhaltsverzeichnis

1	Einführung	1
1.1	Ziele	1
1.2	Voraussetzungen	1
1.3	Aufgabenstellung	1
2	Ergebnisse	2
2.1	Django-REST-Framework Tutorial	2
2.1.1	1: Projekt Setup	2
2.1.2	Sync Database und Superuser	2
2.2	Registration and Login with Spring Boot, Spring Security, Spring Data JPA, and HSQL	2
2.2.1	Neues Projekt	2
2.2.2	Dependencies	2
2.2.3	User	3
2.2.4	Role	3
2.2.5	Testing	3
2.3	NodeJS	5
2.3.1	Passport	5
2.3.2	Projektstruktur	5
2.3.3	Installation der Packages	6
2.3.4	Erläuterung der Packages	6
2.3.5	Application Setup	7
2.4	Autorefreshing mit nodemon	8
2.5	Aufsetzen von MongoDB	8
2.6	mit DB Verbinden	10
2.7	Routes	10
2.8	Views	11
2.9	User Model	12
2.10	Passport Config	12
2.11	Handling von Login/Registrierung	12
2.11.1	require	13
2.11.2	serialize/deserialize des Users	13

2.11.3 Strategy und zugehörige Funktion für Login und Register	13
2.12 Routen anpassen	14
2.13 Serverstart	14
2.14 Register	15
2.15 Persistenz	16

1 Einführung

Diese Übung zeigt die Anwendung von mobilen Diensten.

1.1 Ziele

Das Ziel dieser Übung ist eine Webanbindung zur Benutzeranmeldung umzusetzen. Dabei soll sich ein Benutzer registrieren und am System anmelden können.

Die Kommunikation zwischen Client und Service soll mit Hilfe einer REST Schnittstelle umgesetzt werden.

1.2 Voraussetzungen

- Grundlagen einer höheren Programmiersprache
- Verständnis über relationale Datenbanken und dessen Anbindung mittels ODBC oder ORM-Frameworks
- Verständnis von Restful Webservices

1.3 Aufgabenstellung

Es ist ein Webservice zu implementieren, welches eine einfache Benutzerverwaltung implementiert. Dabei soll die Webapplikation mit den Endpunkten /register und /login erreichbar sein.

Registrierung Diese soll mit einem Namen, einer eMail-Adresse als BenutzerID und einem Passwort erfolgen. Dabei soll noch auf keine besonderen Sicherheitsmerkmale Wert gelegt werden. Bei einer erfolgreichen Registrierung (alle Elemente entsprechend eingegeben) wird der Benutzer in eine Datenbanktabelle abgelegt.

Login Der Benutzer soll sich mit seiner ID und seinem Passwort entsprechend authentifizieren können. Bei einem erfolgreichen Login soll eine einfache Willkommensnachricht angezeigt werden.

Die erfolgreiche Implementierung soll mit entsprechenden Testfällen (Acceptance-Tests bez. aller funktionaler Anforderungen mittels Unit-Tests) dokumentiert werden. Verwenden Sie auf jeden Fall ein gängiges Build-Management-Tool (z.B. Maven oder Gradle). Dabei ist zu beachten, dass ein einfaches Deployment möglich ist (auch Datenbank mit z.B. file-based DBMS).

2 Ergebnisse

2.1 Django-REST-Framework Tutorial

Ich habe mit zur Bearbeitung dieser Übung die Django-REST-Schnittstelle ausgesucht.

2.1.1 1: Projekt Setup

Zuerst muss ein neues Django Projekt erstellt und die Quickstart app aufgerufen werden. Das Tutorial gibt hierbei vor, wie man über die Commandline ein Projekt erstellt und weitere Schritte. Diese haben jedoch nicht funktioniert, da das Command zum Erstellen eines neuen Projekts : **django-admin.py startproject tutorial** . keine Auswirkung hatte.

Lösung: Also wurde händisch über PyCharm ein neues Django Projekt angelegt und die Ordnerstruktur später manuell erweitert.

2.1.2 Sync Database und Superuser

Anschließend wird die Database über das Command **python manage.py migrate** zum ersten mal Angeglichen und ein neuer Superuser wird mit **python manage.py migrate** angelegt.

Dieser hat den Username **admin** und das Passwort **password123**.

2.2 Registration and Login with Spring Boot, Spring Security, Spring Data JPA, and HSQL

Aufgrund des zu großen Aufwandes bei der vorigen Version mit dem Django-REST Framework, wurde die Aufgabe nun mit Spring , JSP und HSQL gelöst. Dazu wurde folgendes Tutorial verwendet: <https://hellokoding.com/registration-and-login-example-with-spring-security-spring-boot-spring-data-jpa-hsql-jsp/>

2.2.1 Neues Projekt

Zuerst wurde einfach das Projekt aus dem Git-Repository geclont.

2.2.2 Dependencies

im xml file muss zuerst als **parent** das 'spring-boot-starter-parent' artifact gesetzt werden.

Als **dependencies** wurden: 'starter-web', 'starter-data-jpa', 'starter-security', 'hsqldb', 'starter-tomcat', 'tomcat-embed-jasper', 'jstl' benutzt. als **plugin** wird 'maven-plugin' eingebaut. Beispiel einer dependency Syntax:

```
1 <dependency>
  <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
```

2.2.3 User

Diese Klasse erstellt eine 'User'-Tabelle mit getter und setter Methoden verbindet sie über **@Table** mit der Entität ohne Entität würde default Klassenname genommen werden. Attribute: 'id', 'username', 'password', 'passwordConfirm', 'role'. Es wird per Entitäten die 'id' als Primary Key gesetzt (**@Id**) und bei jedem User eine automatisch noch nicht benutzte Id erstellt (**@GeneratedValue**). Ausserdem wird ein Set mit den usern und ihren Rollen erstellt.

Code Snippets: Tabelle:

```
1  @Entity
2  @Table(name = "user")
3  public class User {
4      private Long id;
5      private String username;
6      private String password;
7      private String passwordConfirm;
8      private Set<Role> roles;
9  }
```

Automatisch generierter Primary Key:

```
1  @Id
2  @GeneratedValue(strategy = GenerationType.AUTO)
3  public Long getId() {
4      return id;
5  }
```

Set mit usern und Rollen als Tabelle:

```
1  @ManyToMany
2  @JoinTable(name = "user_role", joinColumns = @JoinColumn(name = "user_id"), inverseJoinColumns =
3      @JoinColumn(name = "role_id"))
4  public Set<Role> getRoles() {
5      return roles;
6  }
```

2.2.4 Role

Hier wird eine Tabelle 'Rolle' mit den Attributen: 'is', 'name', 'users' und zugehörigen getter und setter Methoden erstellt. Auch hier wird 'id' als Primary Key gesetzt und automatisch generiert. Die **@ManyToMany** Entität erstellt eine M zu N Relation.

@JoinColumn verbindet 2 Tabellen über foreign-key.

2.2.5 Testing

Zuletzt wurde noch versucht über Junit zu testen. Es wurde eine Testklasse **LibraryTest.java** erstellt. Beim Ausführen kommt jedoch folgende Fehlermeldung bezüglich Junit zur Vorschein:

Lösungsansatz: das **pom.xml** mit einem junit dependency erweitern. Also wurde der Code Snippet von Mavenrepository für junit in das xml eingefügt.

Snippet:

```
1  <!-- https://mvnrepository.com/artifact/org.junit/com.springsource.junit -->
2  <dependency>
3      <groupId>org.junit</groupId>
```

```
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ auth ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\Schule\5. Jahrgang\SYT\SYT Repository\GK9_3\GK93\target\test-classes
[INFO] -----
[ERROR] COMPILATION ERROR :
[INFO] -----
[ERROR] /D:/Schule/5. Jahrgang/SYT/SYT Repository/GK9_3/GK93/src/test/java/LibraryTest.java:[1,17] package org.junit does not exist
[ERROR] /D:/Schule/5. Jahrgang/SYT/SYT Repository/GK9_3/GK93/src/test/java/LibraryTest.java:[2,24] package org.junit does not exist
[ERROR] /D:/Schule/5. Jahrgang/SYT/SYT Repository/GK9_3/GK93/src/test/java/LibraryTest.java:[11,6] cannot find symbol
    symbol:   class Test
    location: class LibraryTest
[ERROR] /D:/Schule/5. Jahrgang/SYT/SYT Repository/GK9_3/GK93/src/test/java/LibraryTest.java:[13,9] cannot find symbol
    symbol:   method assertTrue(java.lang.String,boolean)
    location: class LibraryTest
[INFO] 4 errors
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 3.185 s
[INFO] Finished at: 2017-10-19T10:39:10+02:00
[INFO] Final Memory: 27M/277M
```

```
5 |     <artifactId>com.springsource.junit</artifactId>
   |     <version>3.8.2</version>
   | </dependency>
```

Leider ist auch dieser Ansatz keine Lösung, daher wurde die 'LibraryTest'-Klasse auskommentiert damit der Error nicht mehr auftritt. Es kann jedoch so auch nicht getestet werden.

2.3 NodeJS

Nächster Versuch: Webapplikation per NodeJS

Die Projektstruktur wurde von folgendem Git-Repo geclont: **Easy-Node-JS-authentication**

2.3.1 Passport

Das **Passport** Package von Node.js kann in Webapplikationen eingebaut werden, um eine einfache Authentifizierungsmöglichkeit in diesen per **username** und **password** zu implementieren. Daher wird dieses auch hier verwendet.

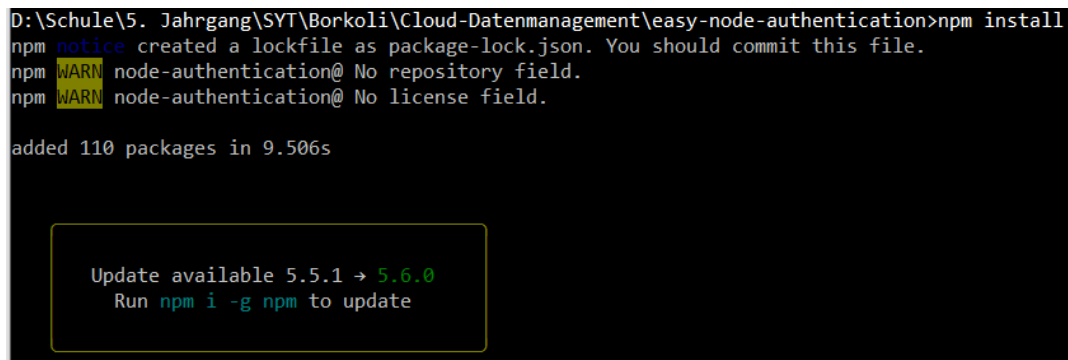
2.3.2 Projektstruktur

- **app**
 - **models**
 - * **user.js**: User-Model
 - **routes.js**: routes der Applikation
- **config**
 - **auth.js**: Hier werden die secret-keys abgespeichert
 - **database.js**: Verbindungs-Settings der Datenbank
 - **passport.js**: Konfiguriert die "passport" Strategien
- **views**:
 - **index.ejs**: Anzeige der Homepage
 - **login.ejs**: Anzeige der Login-page
 - **signup.ejs**: Anzeige der Register-page
 - **profile.ejs**: Anzeige des userprofils
- **package.json**: Beinhaltet npm packages
- **server.js**: Setup der Applikation

2.3.3 Installation der Packages

Zuerst werden alle benötigten Packages installiert.
Folgende Packages sind für uns wichtig:

```
2  "dependencies": {  
4    "express": "~4.14.0",  
6    "ejs": "~2.5.2",  
8    "mongoose": "~4.13.1",  
10   "passport": "~0.3.2",  
12   "passport-local": "~1.0.0",  
14   "connect-flash": "~0.1.1",  
    "bcrypt-nodejs": "latest",  
    "morgan": "~1.7.0",  
    "body-parser": "~1.15.2",  
    "cookie-parser": "~1.4.3",  
    "method-override": "~2.3.6",  
    "express-session": "~1.14.1"  
  }
```



```
D:\Schule\5. Jahrgang\SYT\Borkoli\Cloud-Datenmanagement\easy-node-authentication>npm install  
npm notice created a lockfile as package-lock.json. You should commit this file.  
npm WARN node-authentication@ No repository field.  
npm WARN node-authentication@ No license field.  
  
added 110 packages in 9.506s  
  
Update available 5.5.1 → 5.6.0  
Run npm i -g npm to update
```

Abbildung 1: Installation der Packages

2.3.4 Erläuterung der Packages

- **Express:** Das verwendete Framework
- **Ejs:** JavaScript template engines (in Node.js sind das View-templates)
- **Mongoose:** Object modeling für die MongoDB
- **Passport:** Framework zur Authentifizierung der User
- **Connect-flash:** Erlaubt das Anzeigen von Flash-messages (Error messages usw.)
- **Bcrypt-nodejs:** Um Passwörter zu hashen. (Vorteil gegenüber Bcrypt: einfacher in Windows anzuwenden)

2.3.5 Application Setup

Das Ziel des **server.js**-files ist das bootstrappen der gesamten Applikation. Der Code wird vom Tutorial vorgegeben:

Setup:

Pfade zu Files/Packages setzen, welche später gebraucht werden:

```
1 var express = require('express');
2 var app     = express();
3 var port    = process.env.PORT || 8080; //setzt Port auf 8080
4 var mongoose = require('mongoose');
5 var passport = require('passport');
6 var flash    = require('connect-flash');

8 var morgan    = require('morgan');
9 var cookieParser = require('cookie-parser');
10 var bodyParser = require('body-parser');
11 var session    = require('express-session');
12
13 var configDB = require('./config/database.js');
```

require(): required File/Package im Parameter (sucht im routes.js file nach ihnen)

Konfiguration:

Verbindung mit der Datenbank herstellen:

```
1 mongoose.connect(configDB.url);
```

Setup Express Applikation:

```
1 app.use(morgan('dev')); // log an die Konsole
2 app.use(cookieParser()); // liest cookies fuer Authentifizierung
3 app.use(bodyParser()); // liest auf HTML forms aus

5 app.set('view engine', 'ejs'); // set up ejs

7 // required wegen passport
8 app.use(session({ secret: 'ilovescotchscotchyscotchscotch' }));
9 app.use(passport.initialize());
10 app.use(passport.session());
11 app.use(flash()); // benutzt flash messages
```

Routes:

Lädt die Routen aus routes.js und gibt app und passwort weiter

```
1 require('./app/routes.js')(app, passport);
```

Launch:

Ausführen mit Konsolenausgabe:

```
1 app.listen(port);
2 console.log('The magic happens on port ' + port);
```

Wichtig:

Der Pfad zum "passport" Objekt wird per "var passport = require('passport.js');" gesetzt und dann an config/passport.js zur Konfiguration und anschließend zur app/routes.js übergeben. Erst dadurch wird er bei den Routen benutzt.

2.4 Autorefreshing mit nodemon

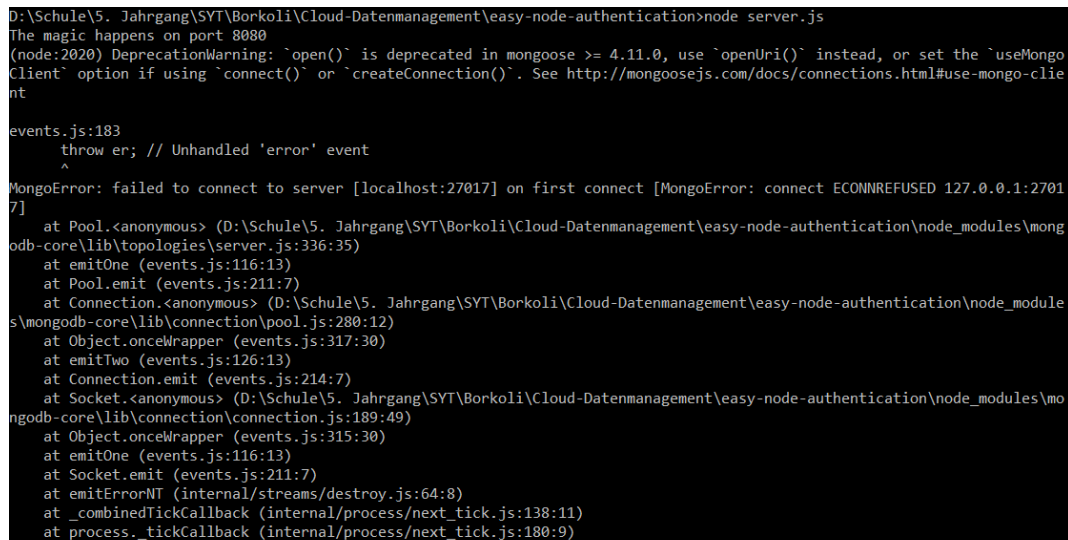
Der Server wird nicht jedes mal wenn ein neuer Datensatz eingefügt wird neu geladen.

Abhilfe: Nodemon: durch den Start des Servers über Nodemon wird immer bei einem neuen Eintrag der Server refreshed und somit aktualisiert.

installation: "npm install -g nodemon"

use: "nodemon server.js"

Hier kommt folgender Error, da noch keine MongoDB Verbindung aufgesetzt wurde.



```
D:\Schule\5. Jahrgang\SYT\Borkoli\Cloud-Datenmanagement\easy-node-authentication>node server.js
The magic happens on port 8080
(node:2020) DeprecationWarning: `open()` is deprecated in mongoose >= 4.11.0, use `openUri()` instead, or set the `useMongoClient` option if using `connect()` or `createConnection()`. See http://mongoosejs.com/docs/connections.html#use-mongo-client
events.js:183
    throw er; // Unhandled 'error' event
    ^
MongoError: failed to connect to server [localhost:27017] on first connect [MongoError: connect ECONNREFUSED 127.0.0.1:27017]
    at Pool.<anonymous> (D:\Schule\5. Jahrgang\SYT\Borkoli\Cloud-Datenmanagement\easy-node-authentication\node_modules\mongodb-core\lib\topologies\server.js:336:35)
    at emitOne (events.js:116:13)
    at Pool.emit (events.js:211:7)
    at Connection.<anonymous> (D:\Schule\5. Jahrgang\SYT\Borkoli\Cloud-Datenmanagement\easy-node-authentication\node_modules\mongodb-core\lib\connection\pool.js:280:12)
    at Object.onceWrapper (events.js:317:30)
    at emitTwo (events.js:126:13)
    at Connection.emit (events.js:214:7)
    at Socket.<anonymous> (D:\Schule\5. Jahrgang\SYT\Borkoli\Cloud-Datenmanagement\easy-node-authentication\node_modules\mongodb-core\lib\connection\connection.js:189:49)
    at Object.onceWrapper (events.js:315:30)
    at emitOne (events.js:116:13)
    at Socket.emit (events.js:211:7)
    at emitErrorNT (internal/streams/destroy.js:64:8)
    at _combinedTickCallback (internal/process/next_tick.js:138:11)
    at process._tickCallback (internal/process/next_tick.js:180:9)
```

Abbildung 2: Error wegen fehlender MongoDB

Folglich wird dies nachgeholt:

2.5 Aufsetzen von MongoDB

um eine Webbasierte MongoDB aufzusetzen wurde folgendes Tutorial verwendet: **Tutorial-MongoDB**

Wie im Tutorial vorgegeben wird zuerst mongoDB per npm installiert. Command: "npm install mongodb --save".

Anschließend wird per Mlab eine MongoDB erstellt. Dafür wählt man den Provider **"Amazon"** und den Typ **"Sandbox"** aus.

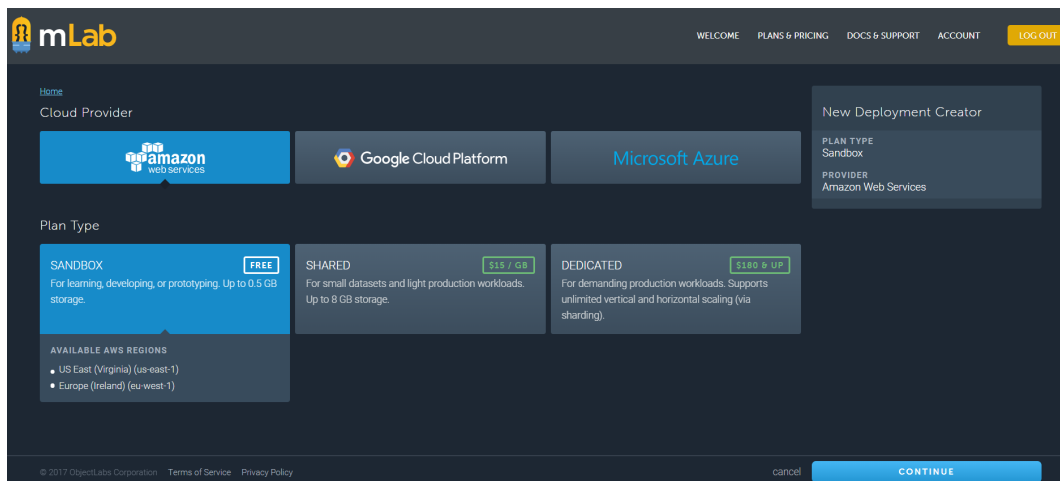


Abbildung 3: Sandbox MongoDB

Die Region wird logischerweise auf **Eu-West** gesetzt

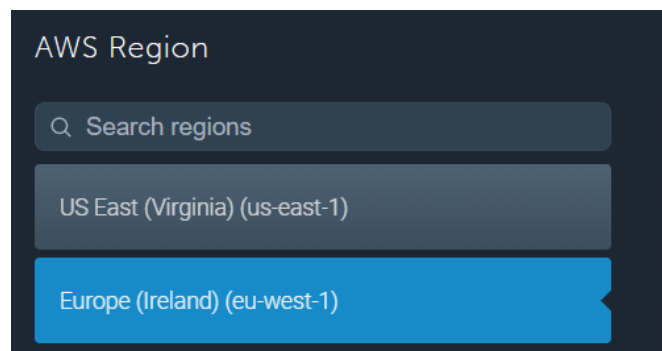


Abbildung 4: Euwest MongoDB

Am Ende werden die Eingaben erneut überprüft:

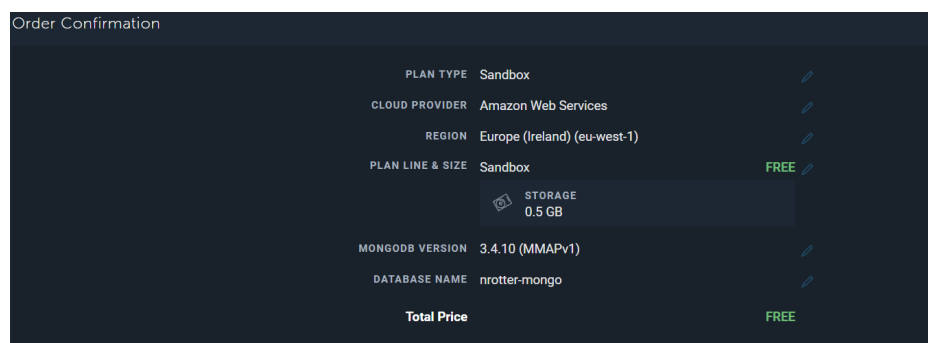


Abbildung 5: Order MongoDB

Zuletzt wird noch ein user mit namen "nrotter" und Passwort "password123" angelegt um später Zugreifen zu können.

2.6 mit DB Verbinden

Nun kann von der Mlab.com Seite eine **URL** zu der MongoDB kopiert werden:

```
mongodb://<dbuser>:<dbpassword>@ds115198.mlab.com:15198/nrotter-mongo
```

"dbuser" und "dbpassword" müssen jedoch noch durch die des gerade erstellten Users ersetzt werden.

```
1 mongodb://nrotter:password123@ds115198.mlab.com:15198/nrotter-mongo
```

Diese URL wird dann im **database.js**-File eingetragen

```
1 // config/database.js
  module.exports = {
3
    'url' : 'mongodb://nrotter:password123@ds115198.mlab.com:15198/nrotter-mongo'
5
  };

```

2.7 Routes

Folgende Routen werden implementiert:

- Homepage (/)
- Login (/login)
- Registrierung (/signup)
- Profil (nach erfolgreichem Login) = Willkommensnachricht

Für diese ist das **routes.js**-File zuständig. Dieses ruft je nach gebraucht die **ejs-Files** im View-Unterverzeichnis auf. Die Methoden dafür sehen folgendermaßen aus:

Homepage:

Zeigt die Homepage, also das index.ejs File

```
2 app.get('/', function(req, res) {
    res.render('index.ejs');
});

```

Loginpage:

Zeigt die Login-Seite an und jegliche flash-Meldungen wenn vorhanden.

```
1 app.get('/login', function(req, res) {
3
    res.render('login.ejs', { message: req.flash('loginMessage') });
});

```

Registerpage:

Zeigt die Registrierungs-Seite an und jegliche flash-Meldungen wenn vorhanden.

```
1 app.get('/signup', function(req, res) {  
2     res.render('signup.ejs', { message: req.flash('signupMessage') });  
4 });
```

Profilpage:

Zeigt die Profil-Seite an, diese ist nur einsehbar, falls man eingeloggt ist um Zugriff von unerwünschten Usern zu verhindern (Überprüfung mit isLoggedIn).

```
1 app.get('/profile', isLoggedIn, function(req, res) {  
2     res.render('profile.ejs', {  
4         user : req.user  
    });  
});
```

Logoutpage:

Ruft req.logout() auf und redirected(verweist) auf die Homepage.

```
1 app.get('/logout', function(req, res) {  
    req.logout();  
3     res.redirect('/');  
    });
```

isLoggedIn Funktion:

Wenn der User eingeloggt ist wird weitergeleitet auf Profilpage, falls nicht auf die Homepage.

```
1 function isLoggedIn(req, res, next) {  
2     if (req.isAuthenticated())  
4         return next();  
6     res.redirect('/');  
    }
```

2.8 Views

Nun werden die Views für die Einzelnen Routen erstellt (Homepage, Loginpage, Registerpage).

Die 3 wurden lediglich aus dem Tutorial kopiert.

2.9 User Model

Das Usermodel ist für das Verbinden eines **Benutzeraccounts** und dessen Email, Passwort und Profil zuständig.

Der Code aus dem Tutorial wurde gekürzt, da Facebook und co. uninteressant für uns sind.

Neuer Code:

```
1 // app/models/user.js
3 var mongoose = require('mongoose');
  var bcrypt = require('bcrypt-nodejs');
5
  var userSchema = mongoose.Schema({
7
9   local      : {
10    email      : String,
11    password   : String,
12  },
13 });
15 //Methoden
17 //Hash erstellen
  userSchema.methods.generateHash = function(password) {
19     return bcrypt.hashSync(password, bcrypt.genSaltSync(8), null);
20  };
21
  //Passwort validieren
23 userSchema.methods.validPassword = function(password) {
24     return bcrypt.compareSync(password, this.local.password);
25 };
27 //Model erstellen und an App weitergeben
  module.exports = mongoose.model('User', userSchema);
```

!!! Das Hashen des Passworts passiert im Usermodel, bevor dieser auf die Datenbank gespeichert wird !!!

2.10 Passport Config

Der Passport wird derzeit in **server.js** erstellt und an **config/passport.js** übergeben. In diesem File werden auch **serializedUser** und **deserializedUser** Funktionen implementiert um den User in der Session zu speichern.

2.11 Handling von Login/Registrierung

Das Handling der Logins und Registrierungen wird im **passport.js**-File abgewickelt.

2.11.1 require

Man braucht 2 requires:

Das Usermodel:

```
1 var User = require('../app/models/user');
```

Die Strategy:

```
1 var LocalStrategy = require('passport-local').Strategy;
```

2.11.2 serialize/deserialize des Users

Passport muss die Möglichkeit haben User zu serialisieren und zu deserialisieren, daher:

Funktion zum serializen des Users:

```
1 passport.serializeUser(function(user, done) {  
    done(null, user.id);  
3 });
```

Funktion zum deserializen des Users:

```
1 passport.deserializeUser(function(id, done) {  
    User.findById(id, function(err, user) {  
3         done(err, user);  
    });  
5 });
```

2.11.3 Strategy und zugehörige Funktion für Login und Register

Hier wird die Logik hinter dem lokalen Login und der lokalen Registrierung gehandelt. Beides besteht aus einer neu angelegten Strategy und einer zugehörigen Funktion.

Login:

Wenn kein User gefunden wurde oder ein falsches Passwort eingegeben wurde, wird eine entsprechende Message zurückgegeben.

Ansonsten wird der User returned und eingeloggt.

Register:

Zuerst Überprüfung ob der User noch nicht eingeloggt ist. Danach abfrage, ob die Userdaten (name, email) schon von einem anderen User verwendet werden, wenn nicht wird der User erstellt.

Wenn der User eingeloggt ist wird die Registrierung ignoriert.

Code dafür wurde aus dem Tutorial übernommen.

2.12 Routen anpassen

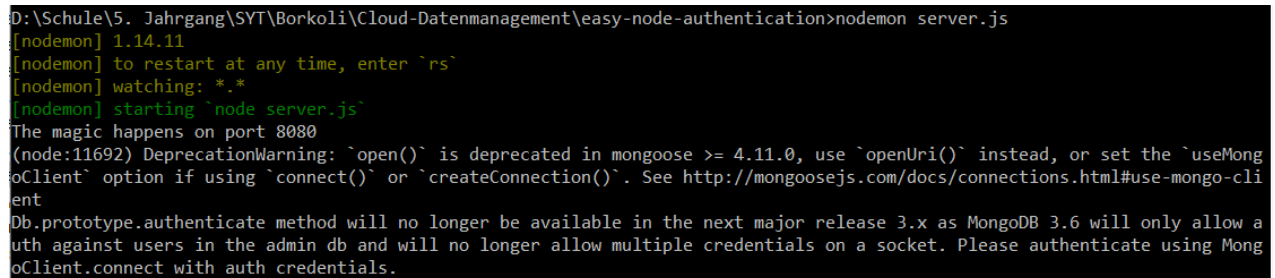
Bei erfolgreicher Registrierung muss nun noch die Weiterleitung in **routes.js**-File hinzugefügt werden.

Hier wird noch zwischen **successRedirect** und **failureRedirect** Unterschieden. Je nach Ergebnis wird der User bei Success zur profilepage und bei Failure zu Homepage weitergeleitet.

```
1 app.post('/signup', passport.authenticate('local-signup', {  
    successRedirect : '/profile',  
3    failureRedirect : '/signup',  
    failureFlash : true  
5 }));
```

2.13 Serverstart

Wenn man nun erneut versucht den Server er **"nodemon server.js"** Command zu starten kommt folgender Konsolenoutput:



```
D:\Schule\5. Jahrgang\SYT\Borkoli\Cloud-Datenmanagement\easy-node-authentication>nodemon server.js  
[nodemon] 1.14.11  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching: *.*  
[nodemon] starting `node server.js`  
The magic happens on port 8080  
(node:11692) DeprecationWarning: `open()` is deprecated in mongoose >= 4.11.0, use `openUri()` instead, or set the `useMongoClient` option if using `connect()` or `createConnection()`. See http://mongoosejs.com/docs/connections.html#use-mongo-client  
Db.prototype.authenticate method will no longer be available in the next major release 3.x as MongoDB 3.6 will only allow a  
uth against users in the admin db and will no longer allow multiple credentials on a socket. Please authenticate using Mong  
oClient.connect with auth credentials.
```

Abbildung 6: Nodemon Serverstart

Nun kann man unter **"localhost:8080"** den Server aufrufen und sieht folgenden Schirm:

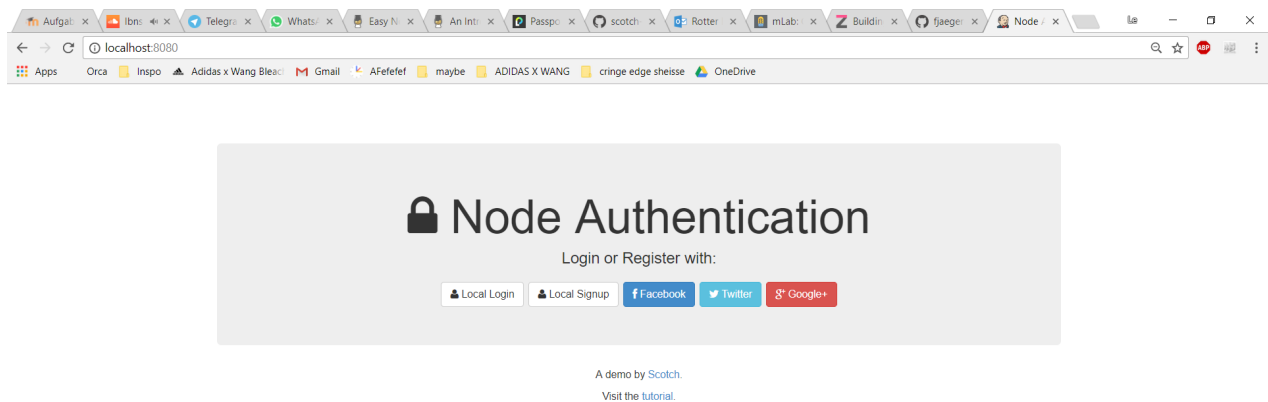


Abbildung 7: Node Authentifizierung

2.14 Register

Nun kann man über **localhost:8080/signup** einen neuen User hinzufügen. Dieser wird dann auch gleich eingeloggt. und auf sein Profil **localhost:8080/profile** weitergeleitet.

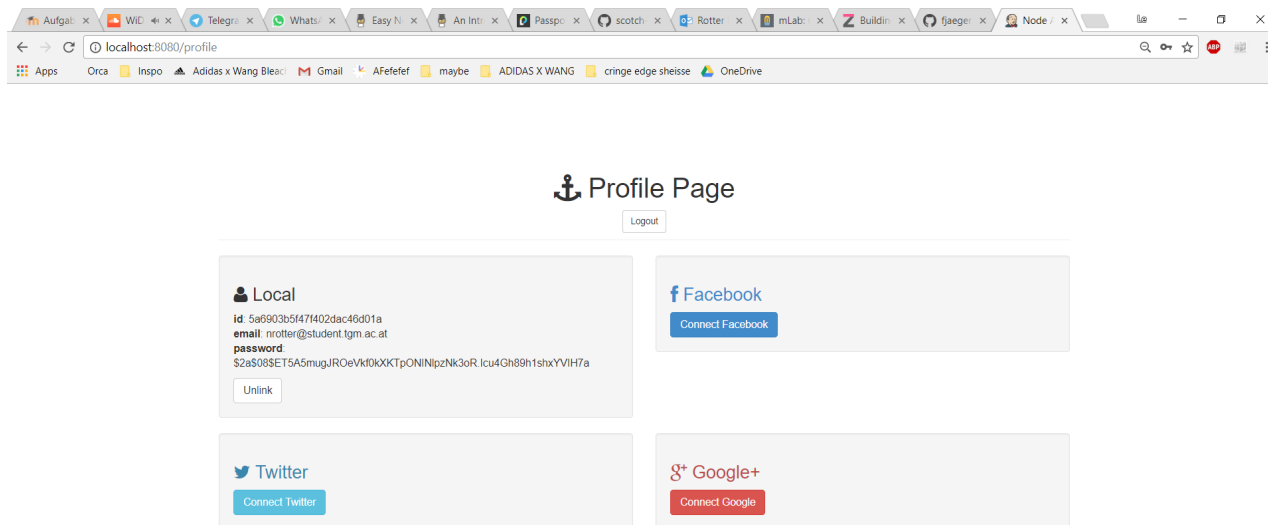


Abbildung 8: userProfile

2.15 Persistenz

Dieser wurde nun auch automatisch in die MongoDB übertragen.

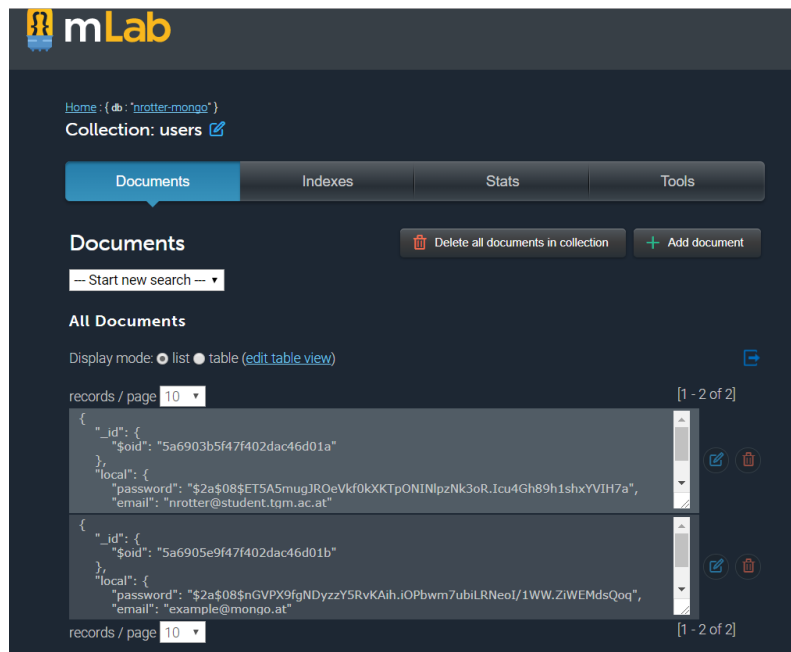


Abbildung 9: userMongo

Serverneustart: Wenn man also nun den Server neu startet und die temporären User gelöscht werden, werden alle in der MongoDB gespeicherten User immer noch vom Server erkannt und man kann sich trotz des Neustarts mit ihnen einloggen.

Beweis für Persistenz: um die Persistenz zu beweisen muss man nur nach einem Serverneustart auf den Kosolenlog achten (falls dieser korrekt implementiert wurde!).

```
D:\Schule\5. Jahrgang\SYT\Borkoli\Cloud-Datenmanagement\easy-node-authentication>nodemon server.js
[nodemon] 1.14.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node server.js`
The magic happens on port 8080
(node:10556) DeprecationWarning: `open()` is deprecated in mongoose >= 4.11.0, use `openUri()` instead, or set the `useMongo
oClient` option if using `connect()` or `createConnection()`. See http://mongoosejs.com/docs/connections.html#use-mongo-cli
ent
Db.prototype.authenticate method will no longer be available in the next major release 3.x as MongoDB 3.6 will only allow a
uth against users in the admin db and will no longer allow multiple credentials on a socket. Please authenticate using Mong
oClient.connect with auth credentials.
GET /login 200 18.369 ms - 1358
POST /login 302 166.995 ms - 60
GET /profile 304 49.529 ms - -
```

Abbildung 10: Beweis Persistenz

Wie man sieht konnte sich der User einloggen und auf sein Profil zugreifen, ohne vorher ein signUp durchzuführen.

Literatur

Tabellenverzeichnis

Listings

Abbildungsverzeichnis

1	Installation der Packages	6
2	Error wegen fehlender MongoDB	8
3	Sandbox MongoDB	9
4	Euwest MongoDB	9
5	Order MongoDB	9
6	Nodemon Serverstart	14
7	Node Authentifizierung	15
8	userProfil	15
9	userMongo	16
10	Beweis Persistenz	16