

Projekt To Do

Indledning

Indholdsfortegnelse

Indledning	1
Indholdsfortegnelse	1
Kort resume	1
Problemformulering	2
Problemstilling	2
Problemformulering	2
Afgrænsning	2
Metodeovervejelser	2
Teknologier	2
Research	3
Database diagram for mongoDB	3
Mongoose	3
Analyse	3
Mongo database i moonmodeler	3
Konstruktion	5
Sådan her ser løsningen ud	5
Evaluering af proces	5
Konklusion	5
Referencer	5

Kort resume

I denne rapport gennemgår vi udviklingen af en to-do liste app udviklet i node.JS med express og EJS.

Problemformulering

Problemstilling

Der skal udvikles en webapplikation der fungerer som en To Do liste, med lagring af data via database. Databasen skal kunne tilknyttes brugere således at applikationen indlæser specifikke brugers data fra databasen, ved brug af applikationen. Det er kun brugere der er oprettet, som har adgang til data i appen.

Applikationen skal indeholde følgende elementer:

- Historik over gennemførte/slettede opgaver
- Bruger login
- Bruger oprettelse med password kryptering
- Bruger godkendelse af admin/validerings løsning
- Deaktivering af bruger
- Oprettelse af liste
- Titel for listen
- Tekst felt for beskrivelse af listen
- Liste punkter for hver opgave
- Start dato for opgaver
- Deadline for opgaver
- Gennemførsels markering af opgaver med evt. dato
- Eksportere data via JSON

Problemformulering

Hvordan udvikler vi en funktionel to-do liste webapplikation hvor brugere kan oprette sig og gemme deres data på en sikker måde, uden risiko for datatab, samt muligheden for at vedligeholde deres planlagte opgaver?

Afgrænsning

Projektbeskrivelsen lyder at vi har mulighed for at vælge en løsningsmodel at arbejde ud fra. Vi afgrænser os derfor fra at løse opgaven med PHP og MySQL/MariaDB, da vores ønske er at udvide vore kompetencer inden for Node.JS, Express og Mongo samt eventuelle view-engines.

Metodeovervejelser

Teknologier

Vi benytter følgende teknologier til at løse følgende problemstillinger:

- HTML5, bruger vi til at bygge frontend skelettet for hele webapplikationen
- CSS3/SASS/BOOTSTRAP, bruger vi til at designe udseendet på webapplikationen

- EJS (Embedded JavaScript), bruger vi som framework til at bibeholde HTML syntaks for læsbarhedens skyld, men med henblik på at undgå redundant kode grundet dets mulighed for at inkludere templates udviklet som partials.
- Node.JS, bruger vi til at få adgang til og udvikle på en lokal webserver, der arbejder asynkront og giver adgang til mange forskellige moduler, så vi undgår at skrive alt kode fra bunden af.
- Express, bruger vi som framework til Node.js, for at få adgang til vores view engine.
- Express generator, bruger vi til at oprette en MVC struktur med server koden serveret.

Research

Database diagram for mongoDB

Vi undersøgte og ledte efter om hvorvidt der fandtes nogle værktøjer som kunne hjælpe os med at oprette et overskueligt database diagram, lignende et ER-diagram, således at vi kunne danne os en visuel præsentation af hvordan en database skulle se ud. Til det har vi fundet et program ved navn Moon Modeller¹, som har en trial periode på 14 dage, hvor i vi kan oprette tabeller, dokumenter, referencer og relationer samt kardinaliteter.

Mongoose populate

Den store udfordring ved brugen af mongoose, var for os at vi ikke har brugt mongoDB eller mongoose, til at joine tabeller/kollektioner før. Derfor voldte det os en del problemer, trods uanede mængder af research via forums og video guides på youtube.

Vi kunne konkludere at Mongoose har en mere kraftfuld metode i sig, kaldt Populate() som gør det muligt at tilføje flere referencer og dokumenter til andre kollektioner. Sammenlignet med SQL, vil det svare til en Left Outer Join. Brugen af populate() synes at skulle fungere ud fra en søgningsmetode (find(), eller findOne()), efter et givent ObjectId som refererer til den kollektion id'en skal fanges fra, efterfulgt af en eller flere values som skal indsættes i stedet for denne ObjectId på det pågældende fundne object. Efterfølgende skal populate bruge en metode "exec", som eksekverer handlingen og derved aktivt erstatter ObjectId med hele objektet, som denne _id referer til. Såfremt dette fejler vil den printe en fejlkode. Se eksemplet² herunder.

```
List.findOne({ title: 'First list' }).populate('user').exec(function (err, list) {
  if (err) return handleError(err);
  console.log('The name for the user of this list is ', list.user.name);
  // prints "The name for the user of this list is kenneth"
});
```

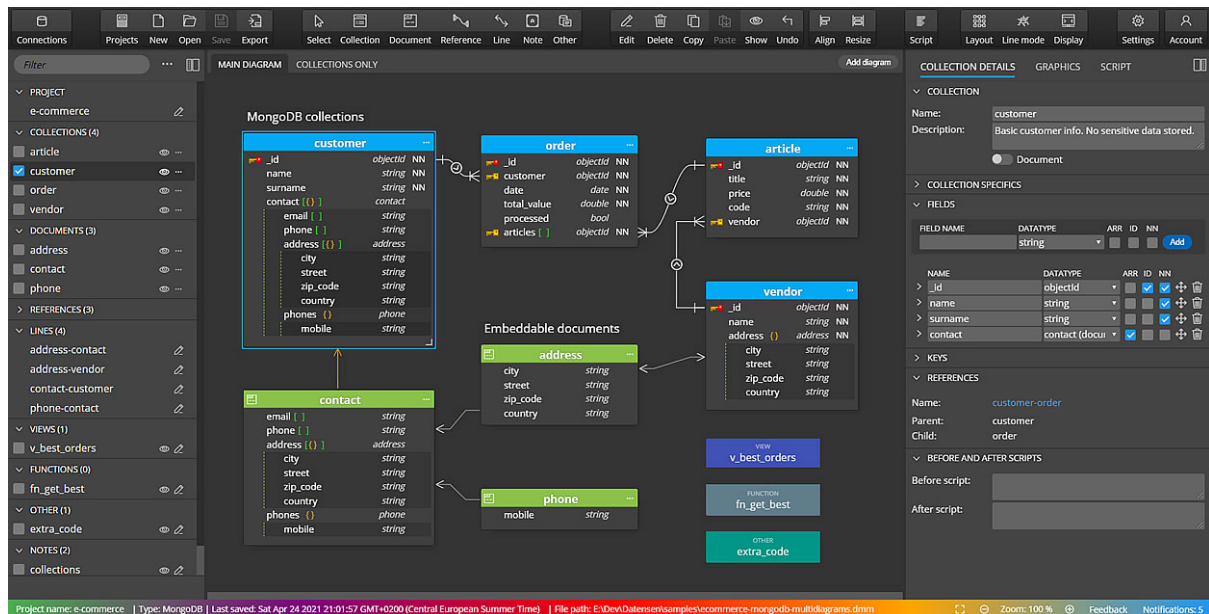
¹ <https://www.datansen.com/data-modeling/moon-modeler-for-databases.html>

² <https://mongoosejs.com/docs/populate.html>

Analyse

Mongo database i moonmodeller

Til at skabe et diagram over vores mongo database for to-do listen, brugte vi vores nyfundne program Moon Modeller som kan vise vores tabeller og deres entities med typer, notnull og keys. Vi demonstrere også hvordan referencerne er forbundet til hinanden, med kardinalitet for mange til mange forhold. Nedenfor ses et screenshot som præsentation af hvordan Moon Modellers kollektioner kunne se ud.




Herunder ses et screenshot af hvordan vores database for to-do list appen er udarbejdet, med en præsentation af hhv. kollektioner med foreign keys der skaber forbindelsen mellem de enkelte kollektioner, samt hvilken kardinalitet der tilhører.

Konstruktion

User MVC

I vores User MVC starter vi med at oprette en model, som skal danne rammen for hvordan vores User class ser ud, der skal instantieres hver gang en ny bruger oprettes, eller logger ind. Vi benytter i dette projekt Mongoose, og derfor oprettes en class via Schema, da det er syntaksen for at oprette indhold til en database.

Til slut i vores User model, eksportere vi Schemaet under navnet 'User', så den kan bruges i vores controllers og routes.

```
models >  User.js > ...  
1   const mongoose = require("mongoose");  
2  
3   //Instantiate new user with mongoose  
4   const UserSchema = new mongoose.Schema({  
5  
6     userName: {  
7       type: String,  
8       required: true  
9     },  
10    // encrypting password through bcrypt  
11    password: {  
12      type: String,  
13      required: true  
14    },  
15    name: {  
16      type: String,  
17      required: false  
18    },  
19    profile: {  
20      type: String,  
21      enum: ['admin', 'regular', 'pending', 'deactivated'],  
22      default: 'pending'  
23    },  
24    lists: [{  
25      type: mongoose.Schema.Types.ObjectId,  
26      ref: 'List'  
27    }]  
28  });  
29  
30  module.exports = new mongoose.model('User', UserSchema);
```

Efter vores User model, opretter vi en controller, som skal håndtere de funktioner og metoder der skal bruges under et request, for at løse requestet inden responset sendes retur til routen. Vores eksempel herunder indeholder flere metoder, både til at fange alle brugere, samt at oprette ny bruger og at logge en bruger ind ved at validere input op mod database kollektionen Users. Der bruges desuden nodemodulet bcrypt, for at hashe brugerens password inden oprettelse i databasen. Når der logges ind bruges bcrypt også her, til at sammenligne det indtastede password med det gemte i databasen.

```
controllers > userController.js > ...
1  const bcrypt = require("bcrypt");
2  const mongoUtil = require("../models/mongoUtil");
3  const User = require("../models/User");
4
5  module.exports = {
6    // query for users and sorting parameters
7    getUser: async function (query, sort) {
8      const db = await mongoUtil.mongoConnect(); // connect
9      const users = await User.find(query, null, sort); // find and read users collection, sorted
10     db.close(); // close connection, so we don't have multiple open or leave a hole
11     return users;
12   },
13   postUser: async function (req) {
14     const db = await mongoUtil.mongoConnect(); // connect
15     bcrypt.hash(req.body.password, 10, function (err, hash) {
16       // Create user only if Hash is successful
17       let user = new User({
18         // create object in schema-format
19         userName: req.body.username,
20         name: req.body.name,
21         password: hash
22       });
23
24       // create method is a static method from mongoose that uses the Create method from http methods (C.R.U.D.)
25       User.create(user, function (error, savedDocument) {
26         console.log(savedDocument); //! remove when publishing
27         if (error) console.log(error);
28         db.close();
29       });
30     });
31   },
32 },
33 loginUser: async function (req, res) {
34   console.log("in loginUser");
35   const db = await mongoUtil.mongoConnect();
36   const inputUser = req.body.username;
37   const inputPwd = req.body.password;
38
39   const user = await User.find({ userName: inputUser });
40   if (user) {
41     console.log("found user");
42
43     const isValidated = await bcrypt.compare(inputPwd, user[0].password); //Password validation
44
45     if (isValidated) {
46       db.close();
47       return user[0];
48     }
49   }
50 },
51 };
52
```

Til slut i vores MVC for users, opretter vi en route, som skal håndtere requestet og sende det ud til hhv. Vores userController, hvori vi danner en route til både at vise og logge en bruger ind, samt en route til at oprette en bruger, og en route til at præsentere lister ved login og ved oprettelse af lister, samt en administrator side der render brugere med pending status, så admin kan se disse og tage stilling til hvilken status de skal have, før brugeren kan logge ind.

```
routes > users.js > ...
1  var express = require('express');
2  var router = express.Router();
3  const userController = require('../controllers/userController');
4  const todoController = require('../controllers/todoController');
5
6  /* GET users listing. */
7  router.get('/', async function(req, res, next) {
8
9      const users = await userController.getUser({});
10     console.log(users);
11     const user = await userController.loginUser({});
12     console.log(user);
13 });
14
15 // render index page upon signup
16 // router.get('/signup', async function(req, res, next) {
17
18 //     res.render('signup', { title: 'Sign Up' });
19 // });
20
21 // register user in db
22 router.post('/signup', async function(req, res, next) {
23
24     const user = await userController.postUser(req, res);
25
26     res.render('index', { title: 'Signed up' });
27 });
28
29 // render dashboard page upon login
30 // router.get('/login', function(req, res) {
31 //     res.render('dashboard', { title: 'Express' });
32 // });
33
34 // log in user using controller for validating input
35 router.post('/login', async function(req, res, next) {
36
37     let user = await userController.loginUser(req, res);
38     let lists = await todoController.getList(req, res);
39     console.log(lists);
40
41     res.render('dashboard', {
42         title: 'Express',
43         user: user,
44         lists: lists
45     });
46 });
47
48 router.get('/admin', async function(req, res, next) {
49
50     const users = await userController.getUser({profile: 'pending'});
51
52     //TODO: Render a list with users, and possibility to change profile from pending to other options.
53     res.render('adminUserPendingList', {title: 'User Administration', users: users });
54 });
55
56 router.post('/updateuser', async function(req, res, next) {
57
58
59
60 });
61
62
63
64 module.exports = router;
```

Evaluering af proces

Gruppedynamikken fungerer upåklageligt, da vi alle 3 er meget engageret og interesseret i at arbejde og udvikle vores kompetencer inden for programmerings feltet. Vi har derfor haft et godt samarbejde både med opgavefordeling samt fælles opgaver, hvor vi løbende har brainstormet og live coded foran hinanden, med henblik på at problemløse i fællesskab. Projektet i sig selv har været en udfordring grundet vores manglende kendskab til at udarbejde en løsning med mongoDB hvortil der skal skabes forbindelser mellem kollektioner såfremt der er behov for foreign keys, og så har projektet også været tidspresset grundet fremrykningen af kick-off, hvortil at weekenden ikke var en mulighed for gruppen at arbejde i. Taget det hele i betragtning, så har vi indfriet vores alles forventninger, ved at udvikle en løsning der kan præsentere tankegangen bag hele projektet, samt hvordan det er tiltænkt at blive løst såfremt vi havde mere tid til det.

Konklusion

Vores kompetencer inden for programmering med Node.JS og Express, samt brugen af MVC, er forbedret betydelig, grundet vores fokus på at prioritere det fremfor designet af opgaven. Vi har i fællesskab opnået bedre forståelse for brugen af routing og controllers, samt brugen af EJS som view-engine. Vi har dog ikke nået at løse opgaven i dens fulde helhed, da vi stødte panden mod muren da det galt kollektionsforbindelser via foreign keys. Vi er dog blevet skarpere på at have en mere struktureret tilgang til fremtidige opgaver, for at undgå at ende i en endeløs bug-spiral.

Referencer