YaddaYaddaYadda

PBWeb - IBA Erhvervsakademi Kolding
Exam project 2nd semester 2021

Morten Højrup Kristensen og Nadia Skau

Indholdsfortegnelse

Indledning

Vi skal skabe et socialt medie i en webapplikation kaldet ‘YaddaYaddaYadda’. Her skal brugerne kunne kommunikere via korte beskeder (Yaddas). En Yadda skal kunne indeholde ét billede, tekst og tags. Der skal have mulighed for at besvare hinandens Yaddas.

Sitet skal give mulighed for at registrering af en ny bruger. Når denne formular er udfyldt, skal brugeren modtage en e-mail med link til endelig godkendelse af profilen.
En brugerprofil skal indeholde en e-mailadresse, navn, avatar og password. Disse oplysninger skal gemmes i en database, hvor passwordet naturligvis skal være hashet.

På sitet skal man se en tidslinje med Yaddas med den nyeste først. Som default skal alle Yaddas vises. Brugeren skal have mulighed for at følge andre brugere samt se en tidslinje med Yaddas fra netop disse brugere. Brugeren skal have mulighed for se hele tråden, når der er besvarelser på en Yadda, og dette skal vises i hierarkisk korrekt orden.
Der skal ydermere være mulighed for at se tidslinjen ud fra et valgt tag.
Der skal også være mulighed for at vælge mellem to temaer i webapplikationen. Dette valg skal kunne ændres og skal gemmes ned i databasen.

Brugeren skal have mulighed for at se, hvem man følger og hvem der følger en.

Som nævnt skal brugerinformation gemmes i en database, og denne dataindtastning skal valideres forinden. Alle Yaddas skal ligeledes valideres og gemmes i en database.
Webapplikationen skal skrives i Node.js med brug af Express. Databasen skal være MongoDB, hvor vi gør brug af modulet Mongoose til håndtering.
Der skal ydermere indtænkes UX-design, således at der er en god brugeroplevelse ved at benytte sitet.

Problemformulering

Hvordan håndterer vi udsendelse af email til en afventende bruger?
Hvordan håndterer vi godkendelse af en afventende bruger via link, der skal indeholde brugerinformation uden at kompromittere datasikkerheden?

Hvordan får vi skabt en god brugeroplevelse?
Hvordan håndterer vi muligheden for to temaer på sitet?

Hvilken hashing algoritme er mest optimal at bruge til at opbevare password?

Hvordan håndterer vi en tråd med Yaddas?

Hvordan håndterer vi filtering i forhold til fulgte brugere samt muligheden for at udvælge et tag?

Hvordan håndterer vi muligheden for at kunne følge brugere?

Hvordan håndterer vi datavalidering af al dataindtastning?

Hvordan håndterer vi opbevaring af et billede i en Mongo database?

Metodeovervejelser

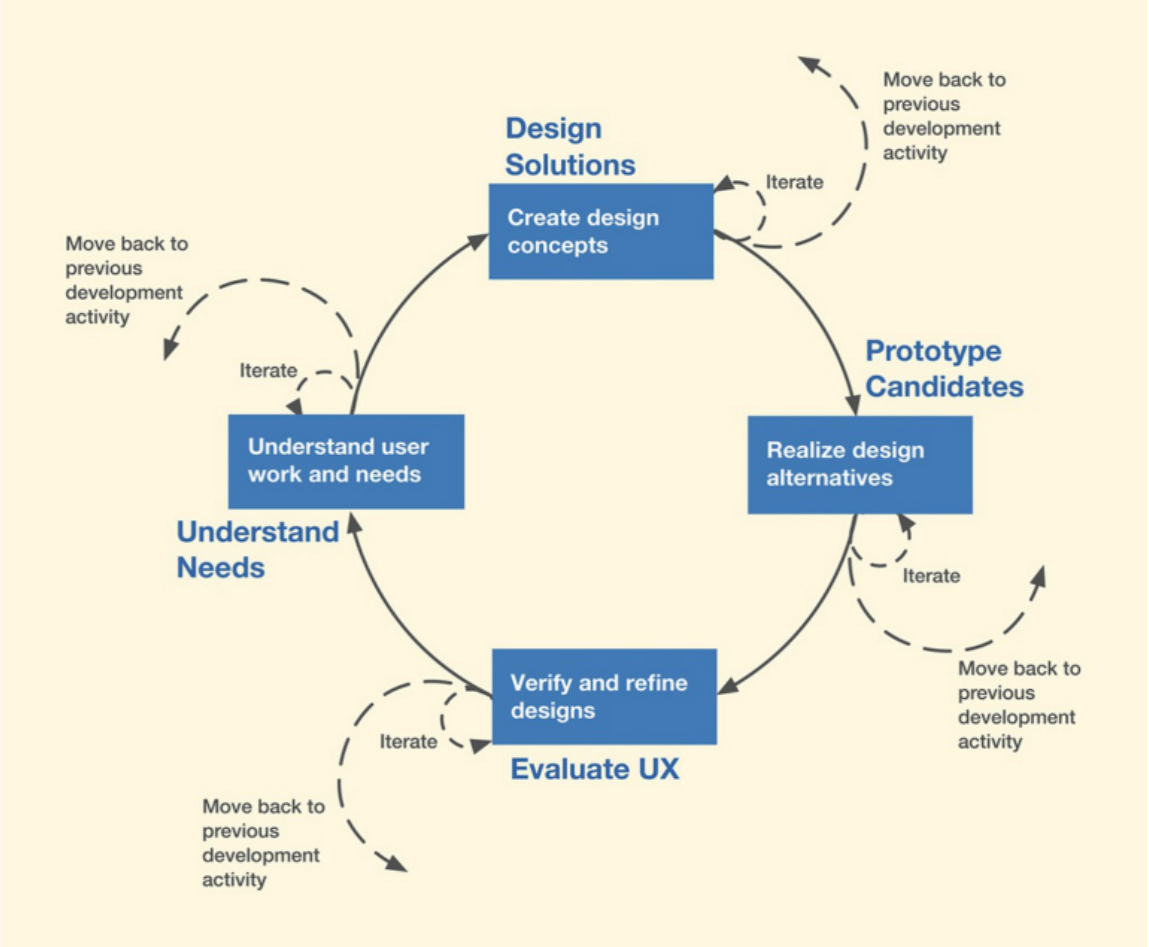
Det var givet i projektbeskrivelsen, at vi skulle benytte Node.js og Express samt MongoDB og Mongoose. Vi har valgt at benytte Pug som view-engine, da vi har arbejdet med det i undervisningen samt tidligere projekter, og det er ligetil at arbejde med.

Vi har valgt at arbejde både client-side og server-side.

Vi har valgt at benytte 'passport' (*Passport* n.d.) til autentifikation, da vi gerne ville have yderligere erfaring med dette modul, da det er et kendt og flittigt benyttet modul.

UX-design

Vi har i vores tidligere projekter benyttet UX Design Lifecycle process (Hartson and Pyla n.d.: 30):



I dette projekt går vi ikke ind i alle steps. Dette eksamensprojekt er ikke i samarbejde med en virksomhed/kunde, og derfor indeholder projektbeskrivelsen brugerens behov og krav til applikationen. Vi går ligeledes ikke ind i evaluering af UX, da hovedformålet med dette projekt er læringsområder fra dette semester, og vi vil lægge vores fokus på dette.

Research

Vi har brugt følgende til at researche på at uploade og indlæse billeder i vores database: (‘Upload and Retrieve Image on MongoDB Using Mongoose’ 2020) (*Expressjs/Multer* 2021). Her har vi ikke haft behov for at undersøge andre moduler, da Multer opfyldte vores behov.

Vi har valgt at benytte bcrypt (*Bcrypt* n.d.) til at hashe vores passwords. Bcrypt benytter Blowfish keying-schedule, og det er en hash string, som er bestående af en algoritme identifier (Alg), en cost/work factor (antal rounds), Salt og en Hash.

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
\__/\/ \____________________/\_____________________________/
 Alg Cost      Salt                        Hash
```

(‘Bcrypt’ 2021)

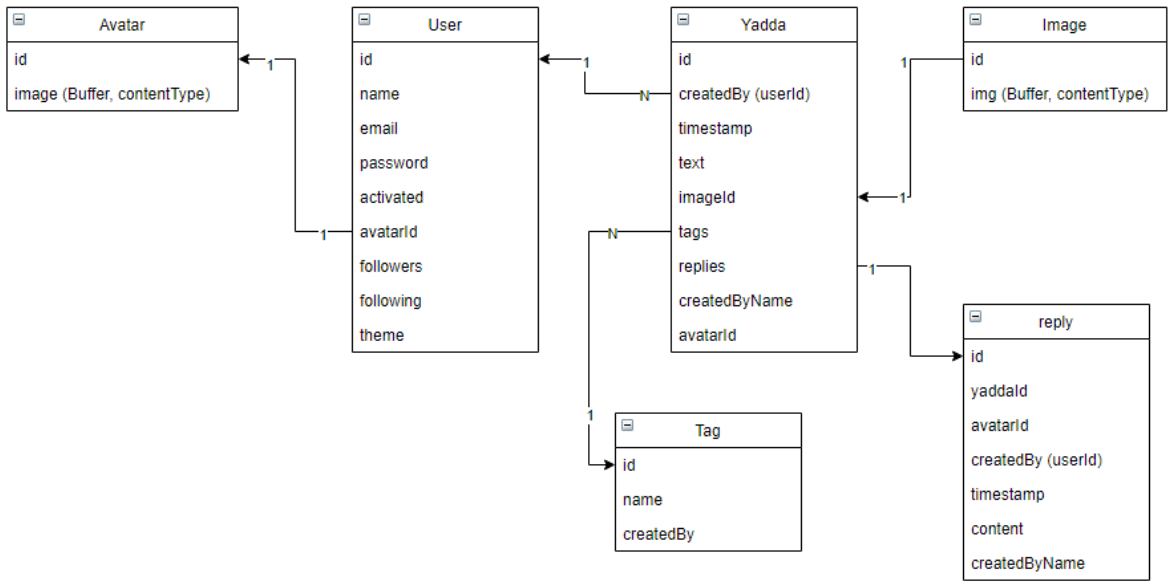
Den er anbefalet af OWASP (*A3:2017-Sensitive Data Exposure | OWASP* n.d.) , og derfor har vi valgt at gå med denne.

I vores research stødte vi også på object-hash (*Object-Hash* n.d.), der giver mulighed for at hashe med forskellige algoritmer bl.a. SHA-1 og MD5. Hverken SHA-1 eller MD5 betegnes som sikre metoder at hashe på (*SHA-1 - Wikipedia* n.d.), da de er mere udsat for length extension attacks og collision.

Analyse

Vi har klarlagt vores design og opbygning af applikationen ved at lave en prototype som det første step. Dette giver et godt overblik både til at opbygge applikationen og style den efterfølgende. Vi har valgt ikke at gå i dybden med UX-metoderne (grundet fokus ligger i de andre fag i dette projekt), og vi har derfor ikke evalueret på prototypen.

Vi har gjort nogle overvejelser i forbindelse med opbygning af databasen, hvor vi har taget udgangspunkt i ER-diagrammet fra projektbeskrivelsen. Vi har justeret en smule:



Vi har valgt at have placeholders (createdByName, avatarID) på Yadda og Reply, da vi har brug for disse i vores indlæsning af dem. De bliver altså aldrig udfyldt i selve databasen og fremgår ikke her, men de er med i vores Schemas, således vi har adgang til dem.

Dette kunne være løst uden disse placeholders ved at gemme disse værdier i enten et særskilt array eller indsætte ekstra objekter ind i arrayet af vores indlæsning af henholdsvis yaddas og replies. Dette ville betyde, at man skulle iterere igennem to arrays eller tage højde for 2D-dimension i indlæsningen af disse.

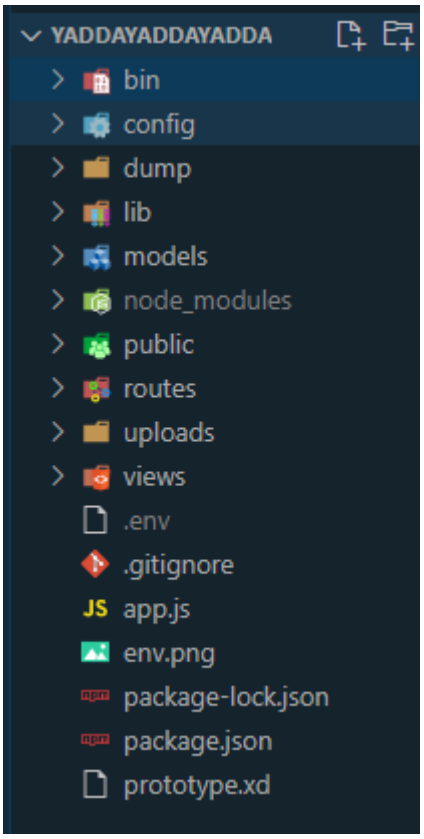
Vi har valgt at benytte 'passport' (*Passport* n.d.) til autentifikation, og dette kræver, at vi åbner forbindelsen til vores database i app.js og altså ikke lukker den igen.
Det ville være at foretrække at lukke databasen efter hver indlæsning mv., men det ødelægger muligheden for at bruge 'req.isAuthenticated()', som vi bruger til at tjekke, hvorvidt brugeren er logget ind. Dokumentationen omkring req.isAuthenticated er i den grad mangelfuld, og det lykkedes os derfor ikke at finde ud af hvorfor, den kræver en åben database forbindelse.
Vi kunne sagtens have løst autentifikationen uden at benytte passport og blot bruge req.session, men vi valgte at gå med passport, da man her nemt kan skifte strategi. Det betyder, at vi altså på et senere tidspunkt nemt kunne skifte fra lokal strategi til Facebook, Google e.l.. Det giver samtidig også mulighed for at have flere strategier, fx at have både mulighed for lokal autentifikation og gennem Google e.l.

Vi har løst det meste af vores indlæsning af data med server-side renderingen, og her kunne vi have løst 'replies'-delen bedre ved at benytte client-side funktion, der indhentede data og appendede dette til siden. Når vi indhenter replies, laver vi en helt ny rendering og brugeren bliver sendt op i toppen og skal scrolle ned. Dette er ikke en optimal brugeroplevelse, og det havde derfor været bedre at tilføje indhold til den eksisterende side.

Konstruktion

Mappestruktur

Vi har bibeholdt den mappestruktur, der bliver generet i et express-projekt. Vi har tilføjet mapperne 'config', 'lib', 'models' og 'uploads'. Config-mappen er til vores passport.js.
I vores lib-mappe har vi vores autentifikationsfunktioner, dato-format, billedhåndtering, e-mailhåndtering og mongooseWrap. Her kunne vi også have placeret vores passport-fil.
Vores models mappe indeholder alle modeller, Schemas og handlers inddelt i undermapper til de specifikke modeller (fx Tag, Image osv.).
Vores uploads-mappe er til de billedfiler, der bliver uploadet (både avatars og billeder i Yaddas). Dertil har vi en 'dump'-mappe, hvilket er en kopi af vores database.
Her ses vores mappestruktur:



Godkendelse via e-mail samt URL

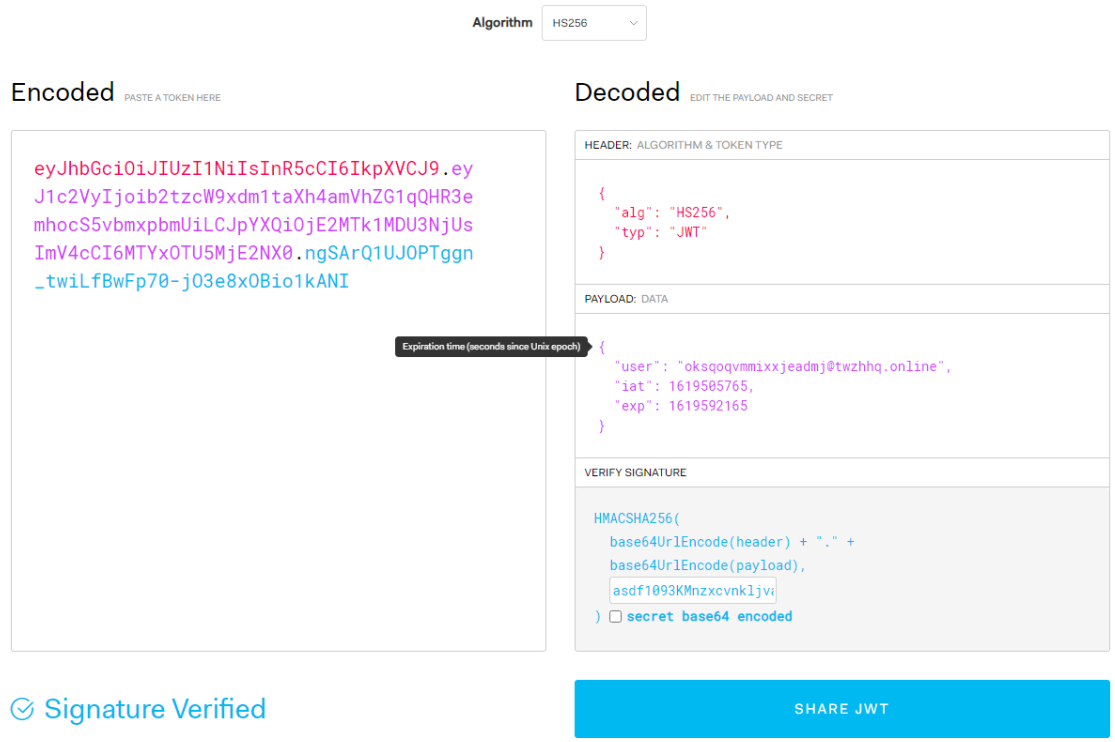
Vi har benyttet modulet 'Nodemailer' (*Nodemailer :: Nodemailer* 2021) til at udsende e-mail fra webapplikationen. I vores mappe 'lib' har vi filen 'mailService.js' liggende, hvor vi har opsætningen til vores mailservice samt funktionen til at udsende e-mail med godkendelseslink.

Med modulet har vi mulighed for at konfigurere et transport objekt (se linje 8 på nedenstående billede), hvilket giver os mulighed for at anvende SMTP[1] til afsendelse af mails. Vi besluttede at anvende Gmail som vores mailservice og oprettede en konto dedikeret til dette projekt.

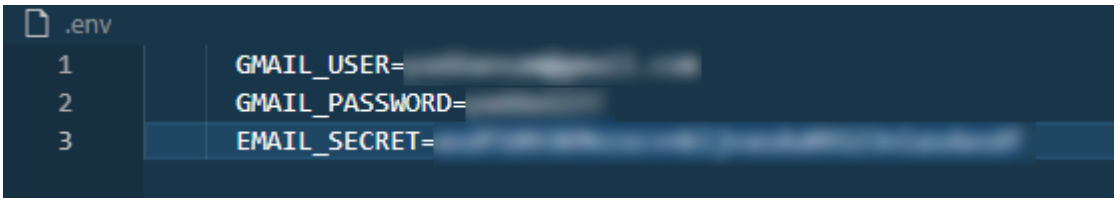
```
1   'use strict';
2   require('dotenv').config();
3   const nodemailer = require('nodemailer');
4   const jwt = require('jsonwebtoken');
5
6   exports.sendEmail = function (email) {
7     // Transporter object - email setup
8     let transporter = nodemailer.createTransport({
9       host: 'smtp.ethereal.email',
10      service: 'Gmail',
11      auth: {
12        user: process.env.GMAIL_USER,
13        pass: process.env.GMAIL_PASSWORD,
14      },
15    });
16    langehk
17
18  //When a user registers
19    jwt.sign(
20      {
21        user: email,
22      },
23      process.env.EMAIL_SECRET,
24      {
25        expiresIn: '1d',
26      },
27      (err, emailToken) => {
28        const url = `http://localhost:3000/users/confirmation/${emailToken}`; //URL for confirmation
29
30        transporter.sendMail({
31          to: email,
32          from: `"YADDA YADDA ADMIN 🤡" ${process.env.GMAIL_USER}`,
33          subject: 'Confirm Email',
34          html: `Please click this email to confirm your email: <a href="${url}">${url}</a>`,
35        });
36      }
37    );
38
39  };
```

Vi benytter modulet 'JSON Web Token' (*JSON Web Token Introduction - Jwt.Io* n.d.) til at oprette et sikkert godkendelseslink. Med dette kan vi udveksle JSON data mellem klienten og serveren på en sikker måde. Denne information kan vi verificere, da den er digitalt signeret. I vores tilfælde har vi benyttet os af HMAC algoritmen, som giver os mulighed for at signere vores JSON object med en secret.Når brugeren klikker på vores link, og serveren får svar retur, bliver token verificeret af serveren vha. vores secret-key.

[1] Simple Mail Transfer Protocol



Vi har benyttet modulet 'dotenv' (*Dotenv* n.d.) til at opbevare vores brugerinformation sikkert og ikke direkte i koden. Filen ligger i roden, men den deles ikke via Git, således informationen heller ikke kan opsnappes her. Nedenstående er vores .env fil:



Opret og visning af kommentarer til en Yadda

Indlæsning i database to gange, så vi kan rendere på "/" og "/:yadda"- kunne måske have været løst smartere
Vi har specifikt valgt ikke at have en underside grundet brugeroplevelse
Når man kommenterer på en tråd, bliver man smidt tilbage til "/" og kommentarer vises ikke - ikke helt optimalt for brugeroplevelse

Passport-local til autentifikation

Vi benytter modulet 'passport' (*Passport* n.d.) med lokal strategi til autentifikation.
I vores tidligere projekter har vi åbnet og lukket databaseforbindelsen ved hver indlæsning/opdatering etc. via 'mongooseWrap.js' filen.
Hvis vi benyttede den måde, når 'passport' skulle autentificere en bruger, så fungerede 'req.isAuthenticated' ikke - den meldte false til trods for, at brugeren var autentificeret.
I linje 15 finder vi brugeren i databasen, og her fungerer det altså ikke at lukke databaseforbindelsen, selvom vi har fået returneret vores bruger.

```
8  module.exports = function(passport) {
9    passport.use(
10     new LocalStrategy({ usernameField: 'email' }, (email, password, done) => {
11       //Make sure the email is lowercase
12       email = email.toLowerCase();
13
14       // Match user
15       model.User.findOne({
16         email: email
17       }).then(user => {
18         if (!user) { //if the user does not exist
19           return done(null, false, { message: 'That email is not registered' });
20         }
21         if(user.activated == false) { //if the user is not activated
22           return done(null, false, { message: 'Check your email for verification link' });
23         }
24
25         // Match password
26         bcrypt.compare(password, user.password, (err, isMatch) => {
27           if (err) throw err;
28           if (isMatch) {
29             return done(null, user);
30           } else {
31             return done(null, false, { message: 'Username or password incorrect' }); //pass
32           }
33         });
34       });
35     })
36   );
37
38   passport.serializeUser(function(user, done) {
39     done(null, user.id);
40   });
41
42   passport.deserializeUser(function(id, done) {
43     model.User.findById(id, function(err, user) {
44       done(err, user);
45     });
46   });
47 };
```

Vi åbner derfor forbindelsen til databasen i app.js, og den er åben hele tiden.
Vi benytter 'req.isAuthenticated' til at sikre os, brugeren er logget ind og ikke kan tilgå direkte URL'er samt at brugeren ikke kan tilgå irrelevante URL'er, når man er logget ind (fx kan brugeren ikke tilgå logind-siden, når man er logget ind).

Funktionen ‘ensureAuthenticated’ tjekker på, om brugeren er autentificeret og returnerer next(), hvis dette er tilfældet. Ellers skyder vi en fejlmeddelelse ind i req.flash og redirecter til logind-siden. Funktionen ‘forwardAuthenticated’ tjekker ligeledes på, om brugeren er autentificeret og returnerer next(), hvis det ikke er tilfældet. Ellers redirectes brugeren til forsiden med alle Yaddas.

```
2   exports.ensureAuthenticated = function(req, res, next) {
3         if (req.isAuthenticated()) {
4           return next();
5         }
6         req.flash('error_msg', 'Please log in to view that resource');
7         res.redirect('/users/login');
8
9       };
10
11  exports.forwardAuthenticated = function (req, res, next) {
12        if (!req.isAuthenticated()) {
13          return next();
14        }
15        res.redirect('/');
16    };
17
18
```

Vi bruger disse to funktioner i vores routes:

```
12  //All of our yaddas
13  router.get('/', auth.ensureAuthenticated, async function (req, res) {
14    let tags = await handlerTag.readTags();
15    let avatars = await handlerAvatar.readAvatar();
16    let yaddas = await handler.readYaddas({}); //read all posts
17    let images = await handlerImage.readImages();
18    let following = await handlerUser.findUserwithId(req.session.passport.user);
19    following = following.following;
20    /*Find all users except the loggedin user and not activated users*/
21    let usersQuery = {
22      $and: [{
23          _id: {
```

Her tjekker vi på, om brugeren er autentificeret, inden indlæsningen og rendering af Yaddas forsiden. På nedenstående billede bruger vi ‘forwardAuthenticated’, således en autentificeret bruger ikke kan tilgå logind-siden:

```
16  router.get('/login', forwardAuthenticated, function (req, res, next) {
17    res.render('login', { title: 'Your credentials', loggedin: false, styleSpecific: "login" });
18
19  });
20
```

Upload og indlæsning af billeder

Upload

Vores formularer har attributten 'enctype' sat til 'multipart/form-data' for at være i stand til at uploade en fil (*HTMLFormElement.Enctype - Web APIs | MDN* n.d.). Vi har benyttet os af attributten 'accept' med '.jpg, .jpeg, .png' for at hjælpe brugeren med at vælge de rigtige filtyper (i dette tilfælde billedfiler). Dette er valideret yderligere, således det ikke er muligt at uploade andet end de ønskede filtyper.

Vi benytter modulet 'Multer' (*Expressjs/Multer* 2021) til vores upload af billeder.
Multer fungerer som middleware til håndtering af fil-uploads, der håndterer typen "multipart/form-data". Multer anvendes til at upload billedet til webapplikationens server-side.
Her opretter Multer et objekt, hvilket indeholder et Body og File objekt.
Body objektet indeholder værdier fra tekstfeltet i vores form, mens File objektet opbevarer filerne som bliver sendt med formen.

Her ses vores opsætning til billedhåndteringen:

```
1  var multer = require('multer');
2  var fs = require('fs');
3  var path = require('path');
4  var modelUser = require('../models/user/user');
5
6  //Multer setup - for our image upload
7  var storage = multer.diskStorage({
8      destination: (req, file, cb) => {
9          cb(null, 'uploads')
10     },
11     filename: (req, file, cb) => {
12         cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname))
13     }
14 });
15
16 var upload = multer({ storage: storage });
17
18 exports.storage = storage;
19 exports.upload = upload;
```

I linje 7-14 fortæller vi Multer-modulet, hvor billedfilerne skal gemmes og hvorledes de skal navngives. Vi har valgt at bruge 'Date.now()' i hvert navn, således de får et unikt navn.

I POST-requestet uploader vi vores billede til serveren (se linje 33):

```
33 router.post('/createuser', image.upload.single('avatar'), async function (req, res) {
34
35   if (req.body.password == req.body.passwordRepeat) {
36     await handler.createUser(req);
37     res.redirect('/users/pending');
38   } else {
39     res.render('createuser', { message: 'Passwords do not match' });
40     // TODO - Logger / flash
41   }
42 });
```

Vi kalder først funktionen ‘image.upload.single(‘avatar’), som henviser til vores ‘imageService’, hvor vi som nævnt har opsat lokation og navngivning.
Parameteret i funktionen er navnet på det inputfelt, hvorfra vi ønsker at hente en fil.

Herefter gemmer vi lokationen i vores database (se linje 15):

```
11  exports.createUser = async function (req, res) {
12    let hash = await bcrypt.hash(req.body.password, 10);
13    let avatar = new avatarModel.Avatar({
14      img: {
15        data: fs.readFileSync(path.join('./uploads/' + req.file.filename)),
16        contentType: 'image/png'
17    }
18    });
19    let savedAvatar = await mongooseWrap.saveAndReturn(avatar);
20
21    let user = new model.User({
22      name: req.body.name,
23      email: req.body.email,
24      password: hash,
25      avatarId: savedAvatar.id
26    });
```

Vi gemmer altså først billedfilerne på serveren og gemmer blot fillokation og mimetype i vores database. Nedenfor er vores skema for en Avatar:

```
1  const mongoose = require("mongoose");
2
3  const avatarSchema = mongoose.Schema({
4     img: { data: Buffer, contentType: String }
5  });
6
7  const Avatar = mongoose.model("Avatar", avatarSchema, 'avatar');
8
9  exports.Avatar = Avatar;
10
```

Fillokationen er en Buffer (binær data). Vi benytter os af Buffer, da det giver bedre performance mht. håndtering af billeder, idet opbevaring og indlæsning af binær data kræver mindre.

Indlæsning

```
13              each avatar in avatars
14                  if(yadda.avatarId == avatar.id)
15                      img(class="avatarYadda" src="" + avatar.img.data.toString('base64'), alt="")
```

Ovenfor ses vores indlæsning af billedet i vores pug fil.
Her er det nødvendigt at fortælle den, at det er base64 samt benytte metoden ‘toString(‘base64’) for at pug kan indlæse URL’en på billedfilen samt mimetypen.

Validering

Vi forsøgte at bruge modulet express-validator (*Getting Started · Express-Validator* n.d.) til at validere vores dataindtastning, men det fungerer ikke med formularer, der har enctype=”multipart/form-data”. Metoderne kører ikke på inputfelterne, og fejl bliver derfor ikke fanget.

Ved at benytte modulet ‘Mongoose’ til at kommunikere med databasen, kan vi også forebygge NoSQL-injektion. I Mongoose opretter vi Schema på hver model, og her er password fx sat til ‘String’. Det betyder, at hvis en hacker forsøger at indtaste et objekt, vil det blive konverteret til en streng, og hackingen vil mislykkedes.

Vi har forhindret brugerne i at kunne uploade andet end .jpg, .jpeg og .png ved at have en funktion, der filtrerer på filtyper:

```
18  var upload = multer({
19      storage: storage,
20      limits: {
21          fileSize: 1000000,
22      },
23      fileFilter : function(req, file, cb) {
24
25      const filetypes = /jpeg|jpg|png/;
26      // Check ext
27      const extname = filetypes.test(path.extname(file.originalname).toLowerCase());
28      // Check mime
29      const mimetype = filetypes.test(file.mimetype);
30
31      if (mimetype && extname) {
32          return cb(null, true);
33      } else {
34          req.flash('error', 'Wrong format!');
35          return cb(null, false);
36      }
37      }
38  })
```

I linje 29 har vi defineret filtyperne, og i linje 27 og linje 29 tester vi på extensionname og mimetype. Hvis de matcher det tilladte, så returnerer vi true og ellers bliver filen ikke uploadet. På denne måde forhindrer vi, at der kan uploades skadelige filer.

Tema

Vi har valgt at gemme tema-valg på brugeren i databasen, således valget bliver husket (og ikke bliver glemt i en ny session). Vi har lavet en toggle-knap med client-side kode, der håndterer at skifte stylesheet samt opdatere databasen.

Vi starter med at hente vores bruger vha fetch GET og indlæse deres tema-valg:

```
10  //Fetching our user
11  async function getUser(url = '/users/theme'){
12    const user = await fetch(url, {
13      method:'GET'})
14      return user.json();
15  }
16
```

```
29  //On page load
30  async function init(){
31    //let theme;
32    let checked = document.getElementById('switch');
33
34    //Calling the fetch to check for theme option
35    getUser().then(user => {
36      if(user.theme == 0){
37        checked.checked = false;
38        document
39        .getElementById('pagestyle')
40        .setAttribute('href', 'stylesheets/styleColor.css');
41      }
42      else {
43        checked.checked = true;
44        document
45        .getElementById('pagestyle')
46        .setAttribute('href', 'stylesheets/style.css');
47      }
48      userid = user._id;
49    });
50  }
51
```

I linje 35 kalder vi vores funktion, der henter brugerdata fra databasen:

```
21  router.get('/theme', ensureAuthenticated, async function (req, res, next) {
22    let user = await handler.findUserwithId(req.session.passport.user);
23    res.json(user);
24  });
25
```

Herfra styrer vi så, toggle-knappen samt stylesheetet efter tema-valget (linje 35-47).

Når brugeren klikker på toggle-knappen, sender vi et POST-request, der opdaterer valget i databasen. Nedenstående er vores fetch POST:

```
17    //Sending theme option
18    async function updateUserTheme(body){
19
20      await fetch('/users/theme', {
21        method:'POST',
22        body: JSON.stringify(body),
23        headers: {"Content-type": "application/json"},
24        redirect: 'follow'
25      }).then(theme => theme.json())
26      .then(text => console.log(text))
27      }
28
```

Den kalder vi her i linje 66, hvorefter vi opdaterer siden med det nye valg i linje 68:

```
52    async function changeTheme() {
53
54      let checked = document.getElementById('switch').checked;
55      let theme;
56      if(checked)
57      {
58        theme = 1;
59      } else {
60        theme = 0;
61      }
62
63      //Body for the POST request
64      let body = {"theme": theme};
65      //Calling our fetch POST to update user theme in MongoDB
66      updateUserTheme(body);
67      //Refreshing our page
68      window.location.href = '/';
69    }
```

Her ses vores POST-request i routeren:

```
26  router.post('/theme', async function (req, res, next) {
27    await handler.updateUser(req, res, {_id: req.session.passport.user}, {$set: req.body});
28  });
```

Toggle-knappen lytter vha. eventlistener på 'onchange', og denne er client-side. Vi har kombineret client-side og server-side ved at benytte fetch, hvor vi laver requests, der både indhenter og opdaterer data fra/i vores database. Dette gør, at vores toggle-knap bliver dynamisk samt gemmer valget for brugeren.

Evaluering

Vi har i vores projekt valgt at samarbejde om opbygningen af webapplikationen. Dette har vi gjort gennem pair-programming. Dette hjælper os til at være mere opmærksom på fejl i koden samt vi får løst vores udfordringer hurtigere. Dertil kommer der, at vi begge gerne vil lære det hele, da det netop er et skoleprojekt. Vi har ligeledes udarbejdet rapporten i fællesskab.

Vi har benyttet et Kanban board til projektstyringen (*Nadiaskau/Yaddayaddayadda/Kanban* n.d.). Her har vi haft tre kolonner 'todo', 'in progress' og 'done', hvor vi har oprettet alle opgaver i 'todo' og så efterfølgende flyttet dem i passende kolonner. Derudover vi har haft en kolonne med vores tidsplan, som vi løbende har justeret i forhold til fremskridt.
Dette har været en effektiv projektstyring for både at huske alle arbejdsopgaver/features samt holde overblik over, om vi overholdt tidsplanen.

Konklusion

Vi har lavet en webapplikation, der håndterer godkendelse af nye brugere via email og et godkendelseslink heri. Hertil har vi brugt modulerne 'nodemailer' og 'JSON Web Token', hvortil sidstnævnte sørger for, vi ikke kompromitterer datasikkerheden i vores godkendelseslink.

Vi har skabt et brugervenligt design i en prototype, som vi har efterfulgt i vores webapplikation. Temavalget gemmes i databasen, således det huskes fra gang til gang.
Der er en god brugeroplevelse med at skifte tidslinjer, poste en Yadda og følge andre brugere. Sidstnævnte huskes i databasen - både hvem man følger og hvem man følges af. Dette er der ydermere mulighed for at få en liste over.
Som nævnt i vores analyse kunne 'replies' funktionen have været løst bedre, da denne på nuværende tidspunkt forringer brugeroplevelsen ved at lave en ny rendering hver gang.

Vi har givet brugeren mulighed for tre forskellige tidslinjer: alle Yaddas, Yaddas fra fulgte brugere og Yaddas med et specifict tag. Her kan brugeren vælge mellem default og followed i toppen, og der er mulighed for at klikke på tags ude i venstre side.
Her laver vi et nyt request og nye indlæsninger i databasen, der sørger for vores filtrering.

Vi har benyttet bcrypt-modulet til at hashe vores passwords, da dette er et modul, som OWASP (*A3:2017-Sensitive Data Exposure | OWASP* n.d.) anbefaler.

Vi fandt ikke nogle moduler, der kan håndtere datavalidering fra en multipart/form-data, og vi kunne derfor ikke gøre brug af disse til datavalidering. Vi lader derfor selve formularen stå for en stor del af valideringen og derudover har vi selv tilføjet nødvendig validering i koden.

I vores research fandt vi ud af, det ikke er muligt at opbevare billedet i en Mongo database. Derfor uploades billederne på webserveren, mens vi gemmer metadata om billedet i databasen.

Vi har skabt en webapplikation med de nødvendige funktioner, der overall har en god brugeroplevelse. Applikationen kunne med fordel videreudvikles til at have en brugerprofil samt muligheden for at indsætte reklamer.

Referencer

A3:2017-Sensitive Data Exposure | OWASP (n.d.) available from <https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure.html> [25 May 2021]

'Bcrypt' (2021) in *Wikipedia* [online] available from <https://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1021887394> [7 May 2021]

Bcrypt (n.d.) available from <https://www.npmjs.com/package/bcrypt> [7 May 2021]

Dotenv (n.d.) available from <https://www.npmjs.com/package/dotenv> [27 April 2021]

Expressjs/Multer (2021) JavaScript. expressjs. available from <https://github.com/expressjs/multer> [29 April 2021]

Getting Started · Express-Validator (n.d.) available from <https://express-validator.github.io/index.html> [5 May 2021]

Hartson, R. and Pyla, P. (n.d.) *The UX Book*. Second Edition. vol. 2019. Morgan Kaufmann Publishers

HTMLFormElement.Enctype - Web APIs | MDN (n.d.) available from <https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/enctype> [3 May 2021]

JSON Web Token Introduction - Jwt.Io (n.d.) available from <https://jwt.io/introduction> [27 April 2021]

Nadiaskau/Yaddayaddayadda/Kanban (n.d.) available from <https://github.com/nadiaskau/yaddayaddayadda/projects/1> [25 May 2021]

Nodemailer :: Nodemailer (2021) available from <https://nodemailer.com/about/> [26 April 2021]

Object-Hash (n.d.) available from <https://www.npmjs.com/package/object-hash> [7 May 2021]

Passport (n.d.) available from <https://www.npmjs.com/package/passport> [7 May 2021]

SHA-1 - Wikipedia (n.d.) available from <https://en.m.wikipedia.org/wiki/SHA-1> [10 May 2021]

'Upload and Retrieve Image on MongoDB Using Mongoose' (2020) [19 May 2020] available from <https://www.geeksforgeeks.org/upload-and-retrieve-image-on-mongodb-using-mongoose/> [29 April 2021]