

SOCIAL MEDIA

Eksamensprojekt forår 2021

Indledning	3
Problemformulering	3
Metodeovervejelser	4
Research	5
Nodemailer	5
Analyse	5
Bcrypt og salt	6
Passport.js	7
Konstruktion	8
Express engine	8
Interpolation	10
Conditionals	11
Passport og Bcrypt	11
Mail verificering	13
Oprettelse af brugere	15
Yaddas og replies	15
Hashtags	18
Følgere	20
Konfigurationsmulighed	21
Evaluerings af process og produkt	23
Fremvisning af yaddas	23
Konklusion	24
Reference	25

Indledning

I det moderne samfund spiller de sociale medier en stor rolle i hverdagen. Det er her, mange mennesker i dag skaber og holder kontakten med hinanden. På baggrund af det har vi som en del af vores anden semester eksamen fået til opgave at udvikle vores eget sociale medie i form af en slags forum. I vores sociale medie skal man have muligheden for at skrive med andre brugere af mediet og følge dem, der interesserer en, og på den måde skabe bekendtskaber, der giver mening for den enkelte bruger. Derudover skal man selvfølgelig have mulighed for at udtrykke sig gennem billeder, fx med avatars og til svar på beskeder, samt andre funktionaliteter, som vi vil komme nærmere ind på længere inde i rapporten.

Problemformulering

“Hvordan kan vi lave en SoMe website, hvor man kan kommunikere med hinanden, registrere sig og følge en bruger ved hjælp af web-interface Express og MongoDB, samt vælge mellem mørk og lyst tema?”

Metodeovervejelser

Til at udvikle vores projekt har vi benyttet os af metoder, vi har lært indtil videre på uddannelsesforløbet - mere specifikt har vi brugt Node og Express som vores framework, samt MongoDB og Mongoose skemaer til at behandle og opbevare vores data, primært brugerdata i dette tilfælde. Vi har også brugt Pug til vores views i stedet for basic HTML, fordi det har vi delvist vænnet os til i forrige projekter og fordi, at Pug giver os mulighed for at bruge JavaScript inde på vores dokumenter i vores views, hvilket ikke er noget, man normalt kan gøre i samme grad med ren HTML. Til autentificering af brugere har vi benyttet os af Passport middleware. Vi benytter os af Passport sessions og local strategy.

Til at samarbejde har vi brugt Visual Studio Code og dens live sessioner via et plugin kaldet Live Share. Her kan vi aktivt rette og skrive kode, så vi alle kan følge med i, hvad der sker, og så slipper vi samtidig for merge problemer med Git.

Vi bruger dog stadig Git. Både som en sikkerhed til backup og så i de tilfælde, hvor vi sidder alene og arbejder med koden.

Vores metoder har vi valgt udelukkende på baggrund af, hvad vi i forvejen har kendskab til, for at optimere vores tid og effektivitet i løbet af projekt-ugerne. Det har været med til at give os mere overskud til at virkelig fokusere på vores funktionalitet med hensyn til vores kode m.m. Dermed har vi oven i købet formået at drage inspiration fra nutidens sociale medier og fået udarbejdet et layout, som vi er tilfredse med.

Vi valgte tidligt i processen at fokusere på at få de features som var nye i det her projekt i forhold til brugere: login med passport og mail verificering, på plads først. Vi lagde en tidsplan for hvad der skulle laves først og det har gjort det muligt for os at have et mere stabilt ståsted i forhold til den videreudvikling af de andre features og testning.

Research

For en god start på vores projekt har vi sammen gennemgået projektbeskrivelsen nøje, og derefter tænkt på de nuværende mest populære sociale medier for at få inspiration til vores eget sociale medie. Vores mål har bl.a. været et simpelt og meget brugervenligt design, da vores medies fokus ligger på at sende beskeder til hinanden og ikke så meget andet, overordnet set. Bagved ligger der meget research omkring autorisering og godkendelse af brugere.

Nodemailer

I løbet af vores research til dette projekt har vi ud over de værktøjer, vi normalt arbejder med, også opdaget nye moduler og andre værktøjer, som har været rigtig gode til lige netop dette projekt. Bl.a. har vi været inde over et modul til Node kaldet Nodemailer, som kan bruges til at sende emails via SMTP protokollen.

Modulet kan bruges sammen med Ethereal Email, som er en dummy email service hvor man kan oprette en gratis bruger til at teste med. Reelt set fungerer det ikke som en normal email, men som fx bruges en enkelt gang til at sende et verificerings-link til en bruger til vores sociale medie. Nodemailer og Ethereal Email er altså ikke noget, hvor man sender beskeder til rigtige, fungerende emails, men i stedet for benytter de sig af "dummy" emails. De er udarbejdet som test- og udviklingsværktøjer, og det kan man også se, da vi fx skal benytte et reference-link, vi får tilsendt gennem disse værktøjer direkte til vores terminal, når vi forsøger at oprette en ny bruger i vores projekt.

Analyse

Vi har som udgangspunkt ladet os meget inspirere af fx Twitter, når det kommer til den måde, vores beskeder opbygges på. Det betyder, at det - i hvert fald i teorien - er lettere for brugere at tilgå og sætte sig ind i, idet det ligner noget, de lidt har set før. Her har vi også haft fat i designprincipper inden for affordances, primært pattern

affordance, som vi har lært om i forrige semester. Det er netop et princip, der tager udgangspunkt i design, som man kender lidt i forvejen, for at gøre brugeroplevelsen lettere og bedre.

Når man har oprettet sig som bruger med en tilhørende avatar og klikker på sit verificerings-link gennem Ethereum, så kan man i vores sociale medie begynde at sende tekstbeskeder, her i projektet kaldet yaddas, til andre brugere. Man kan også vælge, hvem man gerne vil følge fremover. At få yaddas til at fungere har været en af vores primære fokuspunkter igennem hele projektperioden, og vi har startet ud med at skabe en funktionalitet, der poster en yadda, som andre brugere derefter kan se, hvis de følger den bruger, der skriver en yadda.

Derudover har vi gjort det muligt for brugere at tilføje billeder til deres yaddas, da det er en vigtig del af sociale medier, når man egentlig tænker over det. Vi kender nok alle en situation, hvor vi har brugt billeder til at udtrykke os om noget, vi læser. Derfor er det vigtigt, at vores brugere også har den mulighed.

Bcrypt og salt

En af fordelene ved Bcrypt er, at Bcrypt er designet til at være langsom. En Bcrypt hash består af en lang streng og de første 2 dele fortæller om, hvilken algoritme der bliver brugt og den anden om dens cost. Cost beskriver hvor meget kraft det tager at beregne en hash. Bcrypts cost er \$13, og det er regnet som at være virkelig sikkert.

Fordelen kommer ved, at man nogle gange kan opleve at ens database bliver kompromitteret. I databasen har man i mange apps en user collection, der indeholder info omkring brugeren. Her kan der være email, username og adgangskode. Ofte benytter en bruger den samme adgangskode til mange forskellige apps, og derfor er det vigtigt, at man beskytter sine brugere og deres oplysninger.

En af de ting man kan gøre, når man beskytter sine brugere er, at man sørger for, at man gemmer deres adgangskoder som en hash. Det er nemt at hashe en plaintext,

men man kan ikke dehashe den. Derfor vil det være umuligt for en hacker, at få den rigtige adgangskode, når den er blevet hashet ordentligt. Det er ikke nok bare at hashe sin brugers adgangskode med Bcrypt, da en hacker kan lave, hvad der kaldes for en dictionary attack, som er et brute force attack, der bruger en dictionary eller rainbow table. Her har han en liste med adgangskoder, og hvordan de ser ud hashet med Bcrypt. Her kommer salt ind i processen.

Her får man så en adgangskode som er en lang streng af salt og Bcrypt. Man ville dog kunne lave et dictionary attack her også, men fordi at Bcrypt er langsom i forhold til andre hashing algoritmer f eks md5, ville det tage al for lang tid for nogle at skulle lave et dictionary til et dictionary attack. Hvor imod f eks et dictionary attack til md5 ikke ville tage særlig lang tid. Sammen med Bcrypt og med ens salt, gør så at hackeren, ville skulle lave et rainbow table for hver muligt salt sammen med hver mulig adgangskode. Derfor stort set umuligt.

Passport js

Passport er et middleware plugin for Node js. Det er et middleware lavet til at hjælpe en som udvikler med at autentificere request. Man tjekker f eks at man har en user, der benytter ens app, der også matcher en user, man har i appens database.

Passport har over 500 strategier, som man kan vælge at benytte i sin web-app, dette kan være ved hjælp af Oauth eller en lokal strategi.

Når man skal benytte passport til at autentificere en request, skal man her også gøre klar, hvilken strategi, som man ønsker at benytte f eks passport.authenticate('local'), hvis man ønsker at benytte en lokal strategi, hvor man opretter en bruger, der skal lave et username og adgangskode for ens web-app, før de kan logge ind. Når passport så laver en callback tager den username og adgangskode med som arguments. Bliver de valideret, vil den returnere done med den validerede user, og hvis ikke er user returneret som false, her kan man vedhæfte en flash message, der fortæller brugeren om fejl i login processen.

Når man bruger authentication requesten er det normalt kun i login processen at dette er nødvendigt. Det er altså kun her at man sender en brugers info vil blive

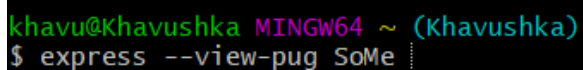
transporteret. Herefter bruger man sessions, hvis altså brugeren er verificeret. Så bliver brugerens id lavet til en session med sessionid, der så bliver gemt i brugerens browser.

Vi benytter os ikke af OpenID og Oauth, da der kan være visse sikkerhedsrisiko. Der kan f.eks. være en risiko, hvis man bruger en Oauth service, der har lavet nogle fejl ved opsætningen, der så tillader en hacker at skaffe sig adgang til et autorisations token eller en kode. Fejl kan også opstå fra klientens egen konfiguration af en webapp. Selvom klienten bruger en velkendt Oauth service, så kan klienten selv opsætte konfigurationen forkert, så der på den måde kommer svagheder for webappens brug af Oauth.

Konstruktion

Express engine

Express engine gav os mulighed for at bruge statiske skabelonfiler. Vi har valgt pug, som er en HTML- templating engine, hvilket betyder, at vi kan skrive meget enklere kode, som browseren kan forstå. For det første skal Pug installeres i projektet ved følgende kommando:

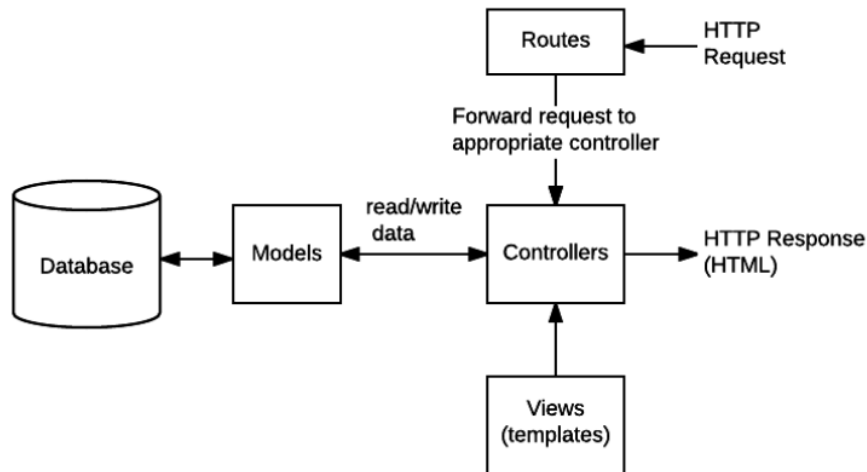


```
khavu@Khavushka MINGW64 ~ (Khavushka)
$ express --view-pug SoMe
```

Vi har navngivet vores projekt for SoMe-projekt. På denne måde har vi med succes bragt Pug i vores projekt.

I vores projekt har vi brugt Mongoose- modeller til at interagere med databasen og brugte et script til at oprette forbindelse mellem. Vi bestemte over, hvilke oplysninger vi vil have vist på vores sider og derefter defineret passende URL'er til returnering af disse ressourcer, derefter oprette ruterne og views for at vise disse sider frem for brugerne.

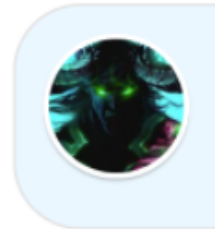
Diagrammet nedenfor viser mappestrukturer som oprettes automatisk efter brugen af kommandoer oppefra.



Controller- funktioner, der adskiller koden til rute anmodninger fra den kode, der faktisk behandler anmodninger. Udover det henter vores controller de ønskede data fra modellerne, som indeholder alle vores schema, opretter en HTML-side, der viser dataene, og returnere dem til brugeren for at se dem i browseren. F.eks.: den funktion nedenfor viser avatar til bruger.

```

exports.lookupAvatar = async function (req, res) {
  let query = req.params.uid;
  let user = await userSchema.findOne({uid: query});
  res.contentType(user.avatar.contentType);
  res.send(user.avatar.data);
};
  
```



Alle vores funktioner har standardformularen til en Express middleware-funktion med argumenter for anmodning og svaret. Koden nedenfor giver et konkret eksempel på, hvordan vi har oprettet ruterne og derefter brugt dem i vores projekt. Ved hjælp af router.get() metoden til at svare på HTTP get-anmodninger med et bestemt sti.

```

/* GET home page. */
router.get('/', function(req, res, next) {
  let user = req.user ? req.user.uid : null;
  res.render('index', {
    title: 'Frontpage',
    user: user
  });
});
  
```

For at kunne bruge ruter-modulet skal man først require().

Rutestien ' / ' i den funktion henviser til index side.

Derefter følger callback tre parameter med til at registrere værdier på bestemte positioner i

URL'en. HTTP - request, HTTP - respons og den næste funktion i middleware kæde. `res.render()` funktion bruges til at gengive en visning og sender den gengivne HTML-streng til klienten i denne funktion tager den titlen med. Views er et standard katalog, som indeholder alle vores pug-filer.

Det sidste ting, der er at tilføje ruterne til middleware kæden og det har vi gjort i vores `app.js` fil.

```
var indexRouter = require('./routes/index');  
var usersRouter = require('./routes/users');
```

Derefter forbinder vi MongoDB-databasen ved hjælp af mongoose. Når vi har lavet forbindelsen med mongoDB i `app.js`, så behøver vi kun at gøre det her og dermed er forbindelse tilgængelig over hele applikationen. Mongoose håndterer alt som schema. Vi har designet schema til en database samling, og har indstillet den datatype, som samlingen skal have. For eksempel, hvis vi skriver et schema for en bruger, så kan vi designe det til at have navn, e-mail, uid osv.

```
const mongoose = require('mongoose');  
  
const userSchema = mongoose.Schema({  
  name: {  
    type: String,  
    required: true  
  },  
  email: {  
    type: String,  
    required: true,  
    unique: true  
  },  
  uid: {  
    type: String,  
    required: true,  
    unique: true  
  }  
});
```

Interpolation

Strenginterpolation er processen med at erstatte en eller flere pladsholdere i en skabelon med en tilsvarende værdi. F.eks.:

```
.comment-avatar-container  
  img.comment-avatar(src="/users/getimage/" + reply.bywhom)  
p.comment-text #{reply.content}  
if (reply.image.data.length > 0)
```

I dette eksempel viser vi yadda beskeden frem på siden. De krøllede parenteser indeholder gyldigt JS-udtryk. Resultatet buffers i output. Til at medtage rå syntaks, har vi brugt et udråbstegn i. F.eks.:

```
p.comment-text !{yadda.content}  
if (yadda.image.data.length > 0)
```

Her har vi valgt at tage hashtags med i yadda-beskeden.

Conditionals

Vi har også brugt statements og looping-konstruktioner. Hvis en bruger er logget ind, skal siden vise indholdet, og hvis ikke, så kun forside, registrering og login. Pug gav os mulighed for at forkorte definitionerne af html-markup.

For at kunne opnå dette har vi defineret en simpel skabelon som:

```
header
  nav(class="main-menu")
    ul
      if !user
        li
          a(href='/')
            i(class="fa fa-home fa-lg")
            span.nav-text Home
        li
          a(href='/users/register')
            i(class="fa fa-plus")
            span.nav-text Register
        li
          a(href='/users/login')
            i(class="fa fa-sign-in")
            span.nav-text Login
      if user
        li
          a(href="/feed")
            i(class="fa fa-rss")
            span.nav-text Feed
```

Dette eksempel fremhæver et par vigtige punkter om Pug. For det første er det mellemrum følsomt, hvilket betyder, at Pug bruger indtrykning til at finde ud af, hvilke tags der er indlejret i hinanden.

For det andet har Pug ingen close tags. Dette sparer ganske få tastetryk og giver Pug en ren og letlæselig syntaks.

Passport og Bcrypt

Ved oprettelse af users, samt autorisering af samme, bruger vi i vores web app Passport.js. Passport giver udvikleren en nem måde at lave autorisering af brugere og holde styr på sessioner på en måde, hvor koden bliver så ren som mulig. Autorisering af request sker ved login funktionerne, og herefter holdes der styr på

sessioner, som er tilknyttet den bestemte bruger. Information om sessionerne bliver gemt i brugerens browser.

Her ses hvordan vi i vores app.js starter med at sætte appen til at benytte sig af passport og dens session. Da vi sætter strategi og hashing et andet sted, så require vi det modul.

```
// Passport middleware
app.use(passport.initialize()); // init passport
app.use(passport.session()); // connect passport and sessions
require('./config/passport')(passport);
```

I vores config passport modul require vi passport local strategy og bcrypt. Som sagt, så skal man fortælle hvilken strategi, man vil benytte, når man sætter passport op. Vi vælger at bruge den lokale strategi altså local strategy. I strategien bruger man et verify callback. Her kigger den efter userid og password. Er det ene eller det andet ikke kendt af serveren, så kommer der en fejlmeddelelse ved hjælp af flash.

Fejlmeddelelsen er ikke sigende om det er userid eller password, der ikke er korrekt. For på den måde at gøre en evt hackers arbejde mere besværligt.

Vi har i et andet modul, som er vores authcontroller, opsat at vores brugers password skal hashes med bcrypt. Vi bruger Bcrypt.compare metoden til at matche det indtastede password med det, som den kender til brugeren i databasen.

```
module.exports = function(passport) {
  passport.use(
    new LocalStrategy({
      usernameField: 'uid'
    }, function (uid, password, done) {
      // Match user
      User.findOne({ uid: uid })
        .then(function (user) {
          if (!user) {
            return done(null, false, { message: 'Incorrect user or password' });
          }
          // Match password
          bcrypt.compare(password, user.password, function (err, isMatch) {
            if (err) throw err;
            if (isMatch) {
              return done(null, user);
            } else {
              return done(null, false, { message: 'Incorrect user or password' });
            }
          });
        });
    })
  );
}
```

I billedet fra app.js ser vi, at vi foruden at require passport også require passport.sessions. Når passport skal bruge sessions, så skal den også serialize og deserialize info på brugeren, som den sender til og fra en session.

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

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

Her bruger vi forward authenticated og ensure authenticated. isAuthenticated sørger for, at den request der kommer er autoriseret.

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

Mail verificering

Udgangspunktet for verificering af email-adresser er Node-modulet Nodemailer. Konfigurationen af Nodemailer findes i config/nodemail.js. Her angiver man i variablen transporter de oplysninger, som man skal bruge for at kunne sende en email. Der skal angives en email server med SMTP protokollen og en port. Til vores behov bruger vi Ethereal Email. Der skal også angives brugernavn og password til mail serveren.

```
let transporter = nodemailer.createTransport({
  host: "smtp.ethereal.email",
  port: 587,
  secure: false, // true for 465, false for other ports
  auth: {
    user: 'monserrate.mayert39@ethereal.email',
    pass: 'A6bWgNab6wXJ5URMEQ', // generated ethereal password
  },
});
```

I variablen info angives afsender, modtager, emne og indhold både med tekst og html. Modtagers email, permalink og verification token kommer fra vores authController hvor de er en del af registreringsprocessen. Token og permalink bruges til at lave en url, som er unik for den givne bruger. Når denne url klikkes, udløser det en get request på serveren, som bliver opsnappet af routeren.

```
let info = await transporter.sendMail({
  from: "RainbowWarriors 🌈🐼", // sender address
  to: email, // list of receivers
  subject: "Hello ✓", // Subject line
  text: "Here is your verification token. Please click the link.", // plain text body
  html: `<p>Here is your verification token. Please click the link.</p><br><a href="localhost:3000/users/verify/" + verification_token + "/" + permalink + ">Click me</a>` // html body
});
```

Routeren bruger req.params til at få token og permalink variablerne fra url'en, og sætter funktionen verify i gang i controlleren. Verify i authcontroller laver et opslag i databasen med permalink, som svarer til brugernavnet og checker derefter med en if-sætning om det token, som den har fået fra url'en, svarer til det, som står i databasen. Derefter bruger den findOneAndUpdate til at sætte brugerens role i databasen til verified, hvorefter brugeren kan logge ind. Hvis token ikke svarer til det som står i databasen, vil der via flash-connect bliver udskrevet en fejl til brugeren.

```
userSchema.findOne({permalink: permalink}, function (err, user) {
  if (user.verify_token == token) {
    userSchema.findOneAndUpdate({permalink: permalink}, {role: "verified"}, function (err, resp) {
      req.flash('success_msg', 'User email has been verified');
      res.redirect('/users/login');
    });
  } else {
    //console.log('The token was wrong! Reject the user. token should be: ' + user.verify_token);
    req.flash('error', 'The token was wrong!');
    res.redirect('/users/login');
  }
});
```

Oprettelse af brugere

For at oprette en bruger til vores sociale medie benytter vi os først og fremmest af et Mongoose Schema, som indeholder de informationer, der skal bruges, når en ny bruger lægges ind i systemet og databasen. I skemaet bliver inputs fra brugeren blandt andet valideret. Vores schema hedder userSchema, og et udkast af det kan ses nedenfor. Ud over navn, email, brugernavn (uid) og password får dette schema også informationer om brugerens rolle, når de oprettes, samt deres valgte avatar billede, en token til at verify dem som brugere, m.m. Vi har ved mange af felterne benyttet required og unique for at kunne validere brugere ud fra fx email og id. Det gør, at man fx ikke skulle kunne oprette sig flere gange med samme id eller samme email.

```
1  const mongoose = require('mongoose');
2
3
4  const userSchema = mongoose.Schema({
5    name: {
6      type: String,
7      required: true
8    },
9    email: {
10     type: String,
11     required: true,
12     unique: true
13   },
14   uid: {
15     type: String,
16     required: true,
17     unique: true
18   },
19   password: {
20     type: String,
21     required: true
22   },
```

Yaddas og replies

At skrive en yadda-besked kan virke som en simpel process, men der er faktisk en masse processer, der foregår, når man kigger inde i koden. Vi skal både ind og hente informationer og id om brugeren, samt have fat i vores yaddaSchema, som indeholder det, der skal bruges til at skrive en yadda. Hvis alt så er gjort rigtigt, og man ikke har skrevet noget, der ikke er godkendt og valideret af vores schema, så bliver beskeden gemt og posted. Til sidst har vi valgt, at man så bliver redirected til

sit feed/dashboard, hvor man kan se alle sine egne yaddas, samt andre brugeres yaddas.

I nedenstående kodeeksempel kan man se en del af processen i at oprette og poste en yadda. Vi har her også brugt `fs.readFileSync` til at læse og fremvise billeder, som brugerne gerne vil tilføje til deres yaddas. Når et billede skal skrives ud på siden, udløser det en get request, som så derefter kommer igennem routeren til det.

```
exports.postYadda = async function(req, res) {
  let form = new formidable.IncomingForm();
  form.parse(req, async function(err, fields, files) {
    if (err) {console.error(err);}

    let {content} = fields; // fra vores yaddaform
    let yaddareply = req.params.yadda;
    let uid = req.user.uid;

    let yadda = new yaddaSchema({
      bywhom: req.user.uid,
      content,
      replyTo: yaddareply
    });
    if (yadda && yadda.image) {

      yadda.image.data = await fs.readFileSync(files.image.path) // read uploaded image
      yadda.image.contentType = files.image.type;
      //måske {runValidators: true}, som option til save() for schema validering? Men hvor skal den stå?
    }

    yadda.save(function (err) {
      if (err) {
        req.flash(
          'error',
          'Something went wrong! The post was not saved.'
        );
        res.redirect('/yaddaform');
      } else {
        req.flash(
          'success_msg',
          'The post was saved.'
        );
        res.redirect('/feed');
      }
    });
  });
};
```

```
//Til at vise billedet frem på yaddas
router.get('/getImage/:id', ensureAuthenticated, yaddaController.lookupYaddaImage, async function(req, res) {
  let query = req.params.id;
});
```

Til vores yaddas har vi også fokuseret på at gøre det sådan, at man kan svare på en yadda, da det er hele pointen med vores sociale medie - det bliver hurtigt et kedeligt forum, hvis ingen kan svare på andres beskeder. At få vores svar/replies til at virke har været lidt en torn i vores side, men til dette formål har vi arbejdet meget med at give yaddas id'er med mere, som vi så har kunne tage fat i gennem vores kode og

derived på en måde sat vores replies fast på en original post. Rent teknisk sker der det, at når man trykker på reply, som er et link i feed, så kaster den linket til et yadda ID videre til yaddaForm view'et, der derefter tager det videre til form elementet og til sidst routeren, der gemmer det inde i databasen.

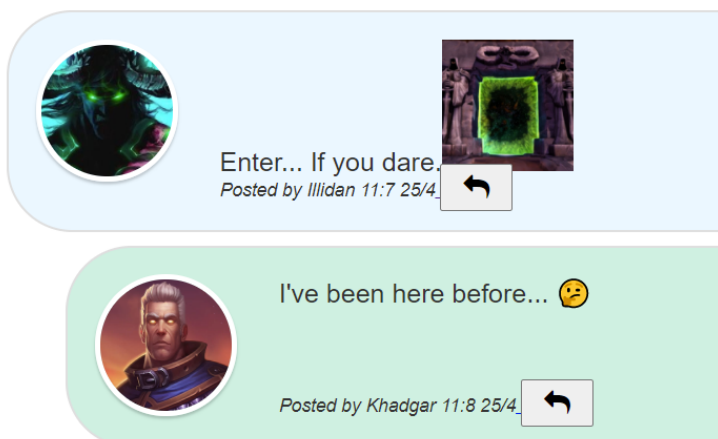
```
router.post('/yaddaForm/:yadda', ensureAuthenticated, async function(req, res) {  
  let yadda = req.params.yadda;  
  await yaddaController.postYadda(req, res);  
});
```

Et reply får noget, som ingen originale yaddas har. Den får udfyldt sin property replyTo. replyTo indeholder et yadda ID fra den yadda, som man svarer på. Det får den med ved hjælp af req.params.

Nedenstående kode fremviser replies, når man trykker på “read more” knappen. Det er her, der findes replies til replies.

```
if(req.params.replies){  
  let replies = req.params.replies;  
  console.log(replies);  
  let replyreply = await yaddaSchema.find({'replyTo': replies});  
  query = {  
    $or: [{ '_id': replies }, { 'replyTo': replies }, { 'replyTo': replyreply } ] // Her bruger vi $or til at vælge den kan have flere conditions  
  };  
}  
let yaddas = await yaddaSchema.find(query).sort({'timestamp': -1}); //til at sortere oplæg  
linkifyHashes(yaddas);  
linkifyHandles(yaddas);  
console.log(yaddas);  
return yaddas;
```

Vi har i øvrigt valgt at sætte en timestamp på vores yaddas, så vi i koden har kunne fået vores app til at sortere beskederne alt efter, hvornår de er skrevet.



Her ses et eksempel på en yadda med billede og en reply til samme yadda uden billede. Ved at trykke på pilen kan man svare på den tilhørende yadda.

Hashtags

Som en del af opgaven skal det være muligt at skrive hashtags inde i sine beskeder/yaddas. På den måde, vil man så kunne trykke på et hashtag, hvis man ønsker at se alle yaddas med et bestemt hashtag. Vi kan starte med at kigge på routeren til hashtags. Vi tager fat i vores link, som er en get request, og den har en url med feed og så hashtaget. Den modtager routeren. Her får den nogle variabler den kan sende med sig videre. Til vores formål fokuserer vi på hashtags, som vi ser den får ved hjælp af # tegnet sammensat med req.params. Videre kan vi kigge på, når vi deklarerer vores yaddas. Her henviser vi til vores yaddaController og en funktion, der hedder getWithHashtag. Når den har været i controlleren kommer den tilbage hertil og renderer så vores feed side med de yaddas, der har det ønskede hashtag.

```
// feed med hashtags
router.get('/feed/:hashtag', ensureAuthenticated, async function(req, res, next) { /
  let user = req.user ? req.user.uid: null; // ? er if for det foran ?
  let uid = req.user.uid;
  let hashtag = '#' + req.params.hashtag;
  let users = await userController.getUsers(req, res);
  let follows = await userController.getFollows(req, res);
  let yaddas = await yaddaController.getWithHashtag(req, res);
  res.render('feed', {
    title: 'The feed',
    user: user,
    users,
    follows,
    yaddas
  });
});
```

Men hvad sker der så inde i controlleren? Inden vi kommer til den funktion, som er refereret i routeren, kan vi tage et kig på to funktioner, der kommer før. Vi har en hashtag funktion og en linkifyHashes funktion. De tager begge yaddas som et argument.

I hashtag laver vi en variabel der hedder regex med en faktisk regex som værdi. subst for et link som en værdi.

variablen txt for at vide den skal tage yaddas og erstatte dem med de forhenværende variabler. Den skal så returnere txt, når den er færdig.

```
// Til Hashtag
const hashTag = function(yaddas){
  const regex = /(\#)(\w+)/g;
  let subst = "<a href='/feed/$2'> $1$2 </a>"; // $
  let txt = yaddas.replace(regex, subst);
  return txt;
};

const linkifyHashes = function(yaddas){
  for (let i = 0; i < yaddas.length; i++) {
    let repltxt = hashTag(yaddas[i].content);
    yaddas[i].content = repltxt;
  }
}
```

Linkify-funktionen tager som sagt yaddas med sig som argument. Så laver den et for loop med et index variabel som får værdien 0. Når index så er mindre end yaddas.length, lægger den index til. Så hver gang, den går det igennem, skal der så ske det, der står efter betingelserne. Den laver så en variabel, der hedder repltxt, der så tager fat i hashtag funktionen fra før. Den tager så yaddas og dens indextal man står ved, og den tager så fat i content. Så for hver regex, den finder i strengen, så tager den og erstatter det streng med linket.

Så kan vi så se vores getWithHashtag funktion, der så er refereret i routeren. Egentlig så tager basis versionen bare fat i yaddas. Den har så fået nogle if sætninger inde i sig. Den vi er interesseret i, kigger efter om den har fået tilsendt et hashtag med sig fra linket til routeren med params. Den får den samme variabel hashtag deklareret, som vi har gjort i routeren også. Så får den en regex, hvor den her også får fortalt, at det handler om en new RegExp. Så ved den, at det er en regular expression, der står inde i. Den kigger så på indholdet i content og specifikt efter hashtagget, som den har fået at vide er en regex. Den slår så op i databasen efter yaddas og tager dem ind i linkify-funktionen, hvorefter den til sidst tager yaddas igennem hele den ovennævnte process og returnerer dem. De bliver så vist, når vi i routeren render feed med disse yaddas, der har det bestemte hashtag.

```

exports.getWithHashtag = async function (req, res) {
  let query = {}; //til database
  let subtitle = ' ';
  let yaddareplies = '';
  if(req.params.hashtag){
    let hashtag = '#' + req.params.hashtag;
    let regex = new RegExp(hashtag, "i");
    query = { //når man trykker på en hashtag
      content: regex
    };
    subtitle = 'Hashtag: ' + hashtag;
  }

  if(req.params.replies){
    let replies = req.params.replies;
    console.log(replies);
    let replyreply = await yaddaSchema.find({'replyTo': replies});
    query = {
      $or: [{ '_id': replies }, {'replyTo': replies}, {'replyTo': replyreply}] /
    };
  }
  let yaddas = await yaddaSchema.find(query).sort({timestamp: -1}); //til at se
  linkifyHashes(yaddas);
  linkifyHandles(yaddas);
  console.log(yaddas);
  return yaddas;
}

```

Sådan kan det se ud, når man trykker på et hashtag.



Følgere

Vi har gjort det muligt at følge andre brugere i vores applikation. Vi har i databasen en collection, som hedder follows. Denne består af bruger ID'er, henholdsvis fra den følger (user) og den, der bliver fulgt (follows). Vi har kaldt det netop dette, fordi man

kan læse dokumentet op og det vil umiddelbart give mening: eksempelvis “user: Martin, follows: Monika”.

Selve det interface, som gør det muligt at følge en bruger, findes på undersiden yaddapeople. Umiddelbart når man kommer ind på siden, så vises de brugere, man følger. Man kan så klikke på knappen “Find yadda people” og det fremviser en liste af andre brugere, som man kan følge. Når man trykker “follow” vil den bruger, man har trykket follow på, dukke op på listen af brugere, man følger.

Det, der foregår inde bagved er, at der er to routere til det samme view. Vi har så en variabel (showsfollows) i routeren, som styrer hvilket indhold, der skal vises fra view’et yaddapeople. Med den knap, som man bruger til at trykke enten follow eller unfollow, følger der en url, som svarer til en get request i routeren. Routeren bruger req.params til at lave henholdsvis findOneAndRemove for unfollow eller en save hvis det er follow.

```
router.get('/follow/:uid', ensureAuthenticated, function(req, res, next) {
  var uid = req.user.uid;
  var follows = req.params.uid;
  userController.follow(req, res, next);
});

/* unfollow på yaddaPeople */
router.get('/unfollow/:uid', ensureAuthenticated, function(req, res, next) {
  var uid = req.user.uid;
  var follows = req.params.uid;
  userController.unfollow(req, res, next);
});
```

Konfigurationsmulighed

Vores brugere kan vælge at bruge et af mindst to mulige temaer. Man kan vælge imellem et “lyst” og et “mørkt” tema. Det er også en del af vores opgavebeskrivelse, og det at kunne vælge mellem flere temaer er noget, man ser hos stort set alle sociale medier, så derfor skal og vil vi også gerne have den mulighed for vores brugere. Vi har også indsat sol- og måne-symboler for at illustrere til brugerne, at der kan vælges mellem et lyst eller mørkt farvetema. Vi har oprettet en fil i public-mappen og tilføjet de to knapper på dashboardet.

LocalStorage er en egenskab, der tillader JavaScript at gemme nøgleværdier i webbrowseren, som en cookie, uden udløbsdato. Dette betyder, at de data, der er gemt i browseren, fortsætter med at gemme data, når browseren er lukket. Derfor har det været oplagt at bruge i dette tilfælde.

For en visuel forståelse af, hvordan vi har brugt LocalStorage, kan man se nedenfor:

```
const htmlEl = document.getElementsByTagName('html')[0];
const currentTheme = localStorage.getItem('theme') ? localStorage.getItem('theme') : null;

if (currentTheme) {
  htmlEl.dataset.theme = currentTheme;
}

const toggleTheme = (theme) => {
  htmlEl.dataset.theme = theme;
  localStorage.setItem('theme', theme);
}
```

h2.tilMode Toggle Dark/Light mode

p Click the button to toggle between dark and light mode.

button#moon(onclick="toggleTheme('dark')" class="fa fa-moon-o")

button#sun(onclick="toggleTheme('light')" class="fa fa-sun-o")

Evaluering af process og produkt

Sikkerhedsproblem med Nodemailer

I konfigurationsfilen til nodemailer angives brugernavn og password på den email som bruges til at sende verificerings mails. Der kunne ligeså godt have stået brugernavn og password på en enhver anden email service eksempelvis gmail eller et firmas egen interne email server. Det er et muligt sikkerhedsproblem da password her angives i plaintext. Så hvis nogen udefrakommende får adgang til vores backend eller github så har de hvad der skal til for at logge ind på mail kontoen. I en live situation bør alle passwords være hashet, og det gælder både for backend og i databasen.

Fremvisning af yaddas

Det som endte med at være det mest komplicerede at implementere var den måde som yaddas og svar til yaddas bliver vist frem på feed view'et. Vi endte med en løsning hvor man kan trykke på en "read more" knap som indlæser yaddas som er svar til svar. Det kan man umiddelbart blive ved med hele vejen ned i hierarkiet af svar på svar på svar osv.

Vi ville gerne lave en løsning, hvor der indlæses og udskrives flere niveauer af en tråd, frem for bare originale yaddas og et enkelt niveau af svar. Vi startede med at lave database opslag både for originale yaddas og for et niveau af svar, men fandt ud af at hvis vi indlæste alle yaddas, så kunne vi via if sætninger og each løkker i vores view engine sætte dem i orden. Det endte så med, at hvis vi skulle have mere end 2 niveauer, skulle vi lave endnu en if sætning og endnu en each løkke. Det mente vi, at vi kunne gøre på en mere elegant måde uden så meget ekstra kode. Vi havde ideer om at bruge frontend javascript til at sortere det ved hjælp af DOM'en, men så skulle vi indlæse alle yaddas af brugere, som man følger, inklusiv billeder og det ville tage længere tid at indlæse. En anden ide vi havde, var at indlæse alle de relevante yaddas og derefter lægge de yaddas, som er svar, ind i et nested object og derefter lægge deres svar ind i endnu et lag af nested objects. På den måde så

kunne man indlæse en fuld tråd af beskeder ad gangen i vores view engine, fordi den orden, de skal ligge i, er givet på forhånd.

Konklusion

Vi har lokalt oprettet en SoMe-skabelon og derefter testet og bekræftet, at det kører ved hjælp af Node. Vi har opnået en løsning på problemformuleringen, som vi er meget tilfredse med, både mht. funktionaliteten og vores brugervenlige design.

Sociale netværk styrer meget i denne verden. De har vores mest værdifulde skatte: Tid og opmærksomhed. Folk bruger mere end 1 time om dagen på sociale mediekkanaler. Det mest attraktive ved de sociale medier i dag er, at det at have en profil på et socialt netværk er den nemmeste måde at holde kontakten med venner i udlandet eller familier, der er hundredvis af kilometer væk. I det vores projekt har haft fokus på at kunne kommunikere med dem, man følger, så kan vores sociale medie også tænkes at passe ind i under denne norm - at brugerne inddrages i et socialt netværk, som ellers kan være svært at holde fysisk ved lige, når alle har en travl hverdag.

Vi har tænkt på brugeroplevelsen igennem hele forløbet og nået frem til et meget overskueligt design som bare kan lige præcis det, det skal - fungere som et netværk for kommunikation.

Reference

Appendix E. Images with MongoDB and Node (n.d.) available from
<<http://dkexit.eu/webdev/site/ape.html>> [24 May 2021]

Express, Installation, and Best Practice (n.d.) available from
<<http://dkexit.eu/webdev/site/ch20s04.html>> [25 May 2021]

Fra Ekspres Og Mops Til HTML (n.d.) available from
<<https://blog.pchudzik.com/201801/express-pug/>> [28 April 2021]

Javascript - Minlength Validator Not Working in Mongoose (n.d.) available from
<<https://stackoverflow.com/questions/58831178/minlength-validator-not-working-in-mongoose>> [24 May 2021]

Javascript - Mongoose Save vs Insert vs Create - Stack Overflow (n.d.) available from
<<https://stackoverflow.com/questions/38290684/mongoose-save-vs-insert-vs-create/38291042>> [24 May 2021]

JavaScript Comparison and Logical Operators (n.d.) available from
<https://www.w3schools.com/js/js_comparisons.asp> [24 May 2021]

Mongoose - MongoDB Schema: Like a Comment OR a Post - Stack Overflow (n.d.) available from
<<https://stackoverflow.com/questions/51793672/mongodb-schema-like-a-comment-or-a-post>> [24 May 2021]

'Mongoose Relationships Tutorial' (2018) [22 June 2018] available from
<<https://vegibit.com/mongoose-relationships-tutorial/>> [24 May 2021]

Mongoose v5.12.10: Query Population (n.d.) available from
<<https://mongoosejs.com/docs/populate.html>> [24 May 2021]

Mongoose v5.12.10: SchemaTypes (n.d.) available from
<<https://mongoosejs.com/docs/schematypes.html#string-validators>> [24 May 2021]

Nodejs | Authentication Using Passportjs and Passport-Local-Mongoose - GeeksforGeeks (n.d.) available from
<<https://www.geeksforgeeks.org/nodejs-authentication-using-passportjs-and-passport-local-mongoose/>> [25 May 2021]

Node.js - How to Reference Another Schema in My Mongoose Schema? (n.d.) available from
<<https://stackoverflow.com/questions/29078753/how-to-reference-another-schema-in-my-mongoose-schema>> [24 May 2021]

Passport (n.d.) available from <<https://www.npmjs.com/package/passport>> [22 April 2021]

Passport-Local (n.d.) available from
<<http://www.passportjs.org/packages/passport-local/>> [25 May 2021]

RegExr: Learn, Build, & Test RegEx (n.d.) available from <<https://regexr.com/>> [24 May 2021]

User Authentication with Passport.Js (n.d.) available from
<<https://mherman.org/blog/user-authentication-with-passport-dot-js/>> [25 May 2021]

<https://stackoverflow.com/questions/28847491/verification-email-with-token-in-passport-js>