Todo-list project

PBWEB Marts 2021

Nadia Skau

Morten Højrup Kristensen

Indledning

Vi skal udarbejde en webapplikation, der fungerer som en todo-liste. Dette skal gøres ved brugen af Node.js, Express og MongoDB samt Mongoose.

Her skal der være to slags brugere: almindelige og administratorer.
Når en bruger er logget ind, skal man kunne oprette opgaver med eventuel deadline og prioritering. Derudover skal brugeren have mulighed for at sende til todo-liste til en anden bruger i JSON og XML-format. Modtageren skal herefter vælge ønsket format.

Når der oprettes en ny bruger, skal denne godkendes af en administrator for at blive oprettet.

Todo-opgaverne skal aldrig slettes fra databasen for at kunne udgøre historisk data. De skal blot have en dato, der betyder opgivet eller fuldført.

Problemformulering

I vores webapplikation for en todo-liste vil vi se nærmere på følgende problematikker:

Hvordan håndterer vi logind i forhold til, om det er en almindelig bruger
eller en administrator?
Hvordan håndterer vi godkendelsen af en ny bruger?
Hvordan håndterer vi oprettelsen af en todo-opgave?
Hvordan håndterer vi fuldførelsen af en todo-opgave?
Hvordan håndterer vi eksporteringen af en todo-liste i JSON og XML-format?

Metodeovervejelser

I projektet var det givet, 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 arbejder server-side gennem hele applikationen, hvorfor vi er nødt til at have a-elementer med button-elementer indeni for at givet et kald til routeren.

En todo-opgave ville dog typisk være med en checkbox, men denne kræver en eventlistener for interaktion, og det ville altså kræve en kombination af client-side og server-side.

Research

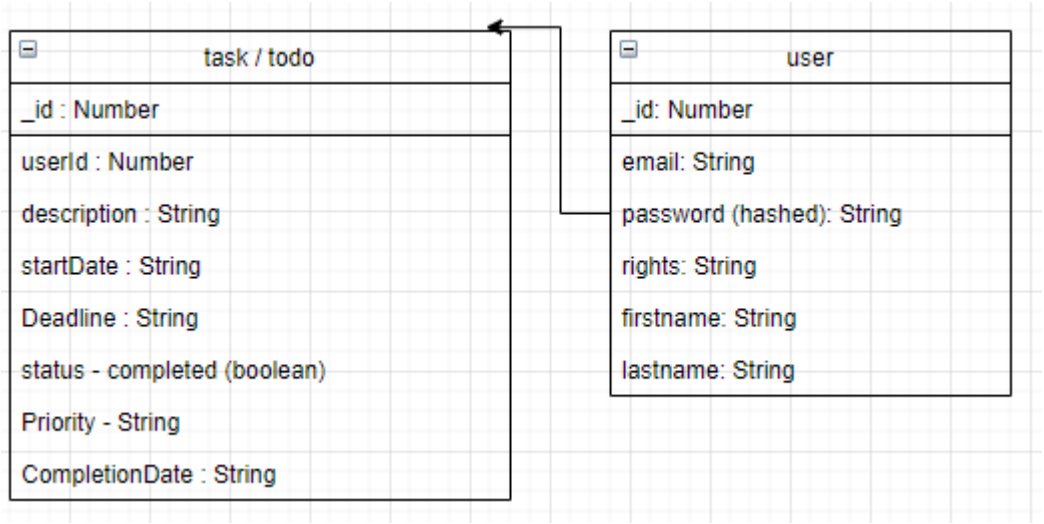
For at kunne tildele rettigheder til vores brugere, undersøgte vi muligheden for at tildele enums, samt en default værdi ved oprettelse af en bruger. Til dette undersøgte vi vores muligheder under SchemaTypes i mongoose's dokumentation: (*Mongoose v5.12.2: SchemaTypes* n.d.)

Vi havde et behov for at downloade filen direkte i browseren. Da dette nu skulle håndteres over serveren, undersøgte vi muligheden for at sende filen med som et response gennem express. Til dette fandt vi funktionen res.download().
På denne måde kunne vi overholde samme standard, hvad filplacering angår. Den downloadede fil, ville altid blive placeret i brugerens "download" mappe: (*Express 4.x - API Reference* n.d.)

Vi havde behov for at eksportere vores todo-liste som både JSON og XML. Til XML delen researchede vi på forskellige libraries, som kunne konvertere et object til XML.
Her valgte vi at gå med dette modul: (*Easyxml* n.d.)

Analyse

Vi startede med at lave et ER-diagram for at få klarlagt, hvordan vores database skulle konstrueres:



Dette gjorde det overskueligt at lave de efterfølgende Schemas.

Rights og priority har vi lavet som enums med tilhørende objekt, således man ikke har mulighed for at oprette med andre værdier. Det kunne også have været løst med at have disse værdier i hver deres collection.

Vi har derefter klarlagt vores applikation vha. wireframes for at få et overblik over, hvorledes visningen skulle opbygges.

Vores applikation og konstruktion er lykkedes godt i dette projekt, og vi har skabt en overskuelig løsning samt overskuelig mappestruktur.

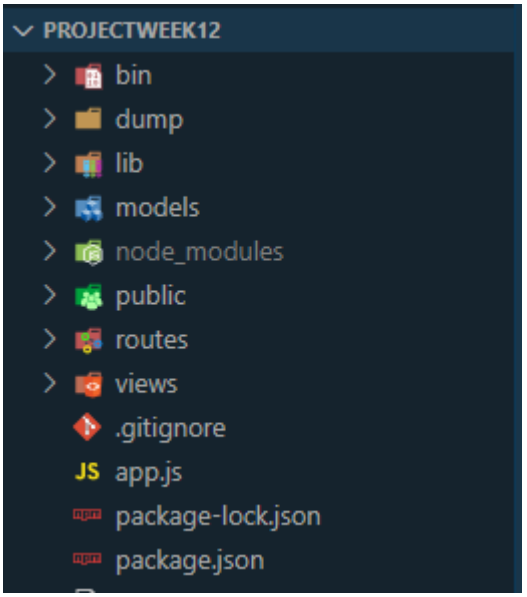
Vi har haft udfordringer med at kombinere client-side/server-side, hvorved vi har undgået at bruge client-side. Dette havde været nyttigt at bruge til at sammenligne passwords samt interaktion på checkboxes.

Vores løsning kunne have været udbygget med, at man tog udgangspunkt i, at applikationen var til en virksomhed. Her skulle man have databasen udbygget med en collection til afdelinger samt en fjerde type bruger: chef/leder, og det her kun var muligt for denne at hente en todo-liste for en tilhørende medarbejder (almindelig bruger).
Vi valgte dog at tage udgangspunkt i, at applikationen er til både private og erhverv, og det er kun en administrator, der kan hente en todo-liste. Dette kunne have været udbygget med, at brugeren selv har mulighed for at eksportere sin todo-liste og dermed sende den til en anden. Projektets beskrivelse krævede dog, at modtageren skulle være logget ind for at kunne modtage en todo-liste, og derfor valgte vi denne løsning.
Vi kunne have optimeret løsningen ved at give brugeren mulighed for sortering af opgaverne fx. i forhold til oprettelsesdato, deadline og prioritet.

Konstruktion

Mappestruktur

Et express-projekt giver os en mappestruktur, og denne har vi bibeholdt.
Herudover har vi vores modeller i 'models' mappen, hvor de har hver deres seperate mappe. Denne indholder en fil til schema og model og en handler-fil. Vores routere ligger i 'routes' mappen, og denne indeholder derfor kun disse fire filer.



I vores 'public'-mappe har vi vores stylesheet og i 'views' har vi vores pug-filer. I vores 'lib' mappe har vi 'date.js' samt 'mongooseWrap.js'.
I 'dump'-mappen ligger der en backup af vores database.

Håndtering af brugere/admins

For at kunne håndtere rettighederne på vores brugere, har vi valgt at inkludere "Rights" på vores userSchema. I skemaet har vi defineret rights af type string, og her har vi angivet en enum "Role", og sat Role.PENDING som værende default.
Grunden til dette er, at en nyoprettet bruger altid vil få tildelt "PENDING" som rettigheder. Dermed kan vi håndtere hvilke sider af applikationen, en ny bruger kan tilgå, inden man bliver godkendt.
Vi har valgt at oprette Role som en enum, sådan at brugeren ikke kan oprettes med en forkert Role. Her sikrer vi os, at der udelukkende kan oprettes/tildeles følgende roller: ADMIN, USER & PENDING.

```
const Role = {
  ADMIN: 'ADMIN',
  USER: 'USER',
  PENDING: 'PENDING',
};

const userSchema = mongoose.Schema({
  email: { type: String, unique: true },
  password: String,
  firstname: String,
  lastname: String,
  rights: { type: String, enum: Role, default: Role.PENDING },
});
```

Log-ind: forskellige typer af brugere

Vi bruger req.session til at håndtere log-ind af de forskellige typer af bruger. Her gemmer vi informationen om brugerens rettigheder. Her i vores 'userHandler' i linje 22 sætter vi req.session.role til at være lig med brugerens rettigheder fra databasen.

```
16  exports.comparePassword = async function(plain, userinfo, req){
17    //comparing plaintext (input) to hash value from database
18    const loggedin = await bcrypt.compare(plain, userinfo[0].password);
19    if(loggedin){
20        req.session.authenticated = true;       // set session vars
21        req.session.user = userinfo[0].email;
22        req.session.role =  userinfo[0].rights;
23    }
24
25    return loggedin;
26  }
```

Når en bruger er logget ind, bliver de redirectet til '/todo' og her håndterer vi visningen således:

```
9   router.get('/', async function(req, res, next) {
10    if(req.session.authenticated && req.session.role == 'ADMIN'){ //ADMIN LOGIN
11      let pendingUsers = await adminHandler.readPending(); //read all pending users
12      let users = await adminHandler.readUsers(); //read all regular users
13      res.render('admin', { pendingUsers: pendingUsers, users: users});
14    } else if(req.session.authenticated && req.session.role == 'USER'){ //REGULAR USER LO
15      let userQuery = {email: req.session.user};
16      let user = await userHandler.readUser(req, res, userQuery);
17      let pendingQuery = {$and:
18        [{userID: user[0]._id},
19        {status: false}]
20      };
21      let pendingTasks = await taskHandler.readTask(res, res, pendingQuery);
22      let completedQuery = {$and:
23        [{userID: user[0]._id},
24        {status: true}]
25      };
26      let completedTasks = await taskHandler.readTask(res, res, completedQuery);
27      res.render('todo', { pendingTasks: pendingTasks, completedTasks: completedTasks });
28    } else if(req.session.authenticated && req.session.role == 'PENDING'){
29      res.render('pending', { title: 'Express' });
30    } else {
31      res.redirect('../users/login')
32    }
33  });
34
```

I linje 10, 14 og 28 tjekker vi på henholdsvis req.session.role == 'ADMIN', 'USER' og 'PENDING' og derfra vælger vi, hvilken data der skal indlæses og hvad der skal renderes. Hvis brugeren ikke er logget ind, sender vi dem til '/users/login'.

Opret bruger - dobbelttjek af passwords

Vi kører vores sammenligning af de to indtastede passwords server-side:

```
46  router.post('/createuser', async function(req, res){
47    if(req.body.password == req.body.passwordRepeat){
48      await handler.createUser(req);
49      res.redirect('/users/pending');
50    }
51    else{
52      res.render('createuser', {message: 'Passwords do not match'});
53    }
54
55  })
```

Hvis ikke de matcher hinanden, kører vi et nyt render af siden med en besked.
Dette havde været løst bedre, hvis vi havde tjekket dette client-side og forhindret formularen i at afsende med 'e.preventDefault();'. På denne måde havde vi sparet et kald til routeren samt en rendering, og brugeren ville heller ikke miste sin indtastning i formularen.

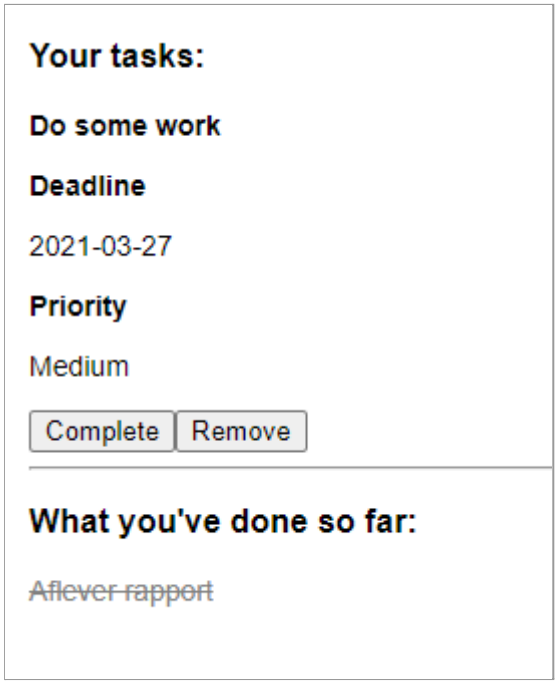
Færdiggør en opgave

I visningen af todo-listen ville vi gerne have haft en checkbox ved hver opgave, der kunne vinges af og opgaven således ville blive opdateret med 'status=true' i databasen.

Her blev vi dog udfordret på client-side/server-side.
Hvis en checkbox skal gøre noget, ville vi være nødt til at bruge eventlisteners og her vidste vi ikke, hvordan vi skulle få client-side til at kalder på server-side og dermed opdatere databasen.

Vi har løst udfordringen ved at have a-elementer med nested button-elementer, der kalder på routeren.

Vores view:



Kaldet til vores router:

```
35  router.get('/complete/:taskID', async function(req, res, next){
36    let query = {_id: req.params.taskID};
37    let updateQuery = {$set:
38      {status: true, completionDate: date.formatedDate()}};
39    await taskHandler.updateTask(req, res, query, updateQuery);
40    res.redirect('/todo');
41  })
42
```

Vores pug-view, der leverer 'taskID' ved klik (se linje 24 og 26):

```
16      h3 Your tasks:
17      each task in pendingTasks
18          div(id=task._id clas="taskContainers")
19              p(class="description") #{task.description}
20              p(class="subtitle") Deadline
21              p #{task.deadline}
22              p(class="subtitle") Priority
23              p #{task.priority}
24              a(href="/todo/complete/" + task._id)
25                  Button Complete
26              a(href="/todo/delete/" + task._id)
27                  Button Remove
```

Eksportering

Når vi eksporterer en todo-liste til XML-format, er vores kode opbygget således:

```
13  router.post('/', async function(req, res, next) {
14    let userEmail = req.body.chosenuser;
15    let format = req.body.format;
16    let user = await userHandler.readUser({email: userEmail});
17    let tasks = await taskHandler.readTask({userID: user[0]._id});
18    tasks = JSON.stringify(tasks);
19    if(format == 'xml'){
20      tasks = JSON.parse(tasks); //needs to get JSON stringified to
21      var serializer = new EasyXml({
22          singularize: true,
23          rootElement: 'todo',
24          dateFormat: 'ISO',
25          manifest: false,
26          unwrapArrays: false
27      });
28
29      let xml = `<?xml version='1.0' encoding='utf-8'?>`; //manifest
30      for (let i = 0; i < tasks.length; i++) { //looping through our
31        xml += serializer.render(tasks[i]); //convert to XML format
32      }
33      xml += `</xml>`
34      tasks = xml;
35
36    }
```

Når vi henter vores array med objekter, er id'et ikke en streng, og derfor kan vi ikke lave det om til XML. Derfor kører vi først en JSON.stringify i linje 18, og i linje 20 kører vi en JSON.parse for at lave det til objekter igen.
Vi fandt ingen moduler, der var i stand til at konvertere et array af objekter til xml, og vi iterer derfor igennem vores array, mens vi opbygger XML'en (se linje 30-33).

Sammenhæng i database

Vores task i databasen referer til tilhørende bruger vha. brugerens id (linje 10):

```
9   const taskSchema = mongoose.Schema({
10      userID: String,
11      description: String,
12      startDate: String,
13      deadline: String,
14      status: {type: Boolean, default: false},
15      priority: {type: String, enum: PRIORITY, default: PRIORITY.LOW},
16      completionDate: String
17  });
18
```

Her har vi blot sat type til at være en String, men dette kunne have været løst ved at bruge 'mongoose.Schema.Types.ObjectId' som type. På nuværende tidspunkt kan det netop være en hvilken som helst streng, på den anden måde ville vi have sikret os, det var et ID, der kom med.

Evaluering af proces

Vi har i vores projekt valgt at samarbejde om opbygningen af vores todo web-applikation. Måden hvorpå vi har gjort dette er gennem pair-programming.
Pair-programming hjælper os til at være mere opmærksom på fejl i koden, samtidig med at vi hurtigere løser vores udfordringer.

Ved projektstart oprettede vi et Kanban board (*Langehk/Projectweek12* n.d.), som havde til hensigt at hjælpe med projektstyringen samt estimering af vores arbejdsopgaver.
Vi startede ud med at identificere alle features vi gerne ville have implementeret, efterfulgt af udarbejdelse af en tidsplan for den kommende uge.

Vi opdelte KanBan boarded med tre kollonner, ("ToDo" - "In Progress" - "Done"), hvor vi udelukkende havde en feature under "todo" ad gangen.
Denne håndtering af arbejdsopgaver gav os en rigtig god gennemsigtighed i projektet, hvor vi kunne løbende se, om vi overholdt tidsplanen.

Ift. strukturen af vores applikation, havde vi fra vores tidligere projekt allerede aftalt en standard for filstrukturen. Denne opbygning valgte vi at nedarvede til dette projekt, for at overholde samme standard, og dermed mindske forvirringen mht. placering af filer/mapper.

Konklusion

Vi har skabt en overskuelig applikation, der håndterer log-ind af tre forskellige slags brugere: afventende, almindelige og administratorer.

En ny bruger bliver som default oprettet som afventende, hvorved de ikke kan foretage sig på noget i applikationen. Administratorerne får automatisk en visning af afventende brugere, som de enkeltvis kan godkende eller afvise. Godkendes brugeren, får denne rettigheder til at kunne oprette og opdatere opgaver samt visning heraf.

Administratoren har også rettighed til at hente todo-lister for samtlige brugere i enten XML- eller JSON format, og denne fil downloades direkte ned på administratorens PC.

Den almindelige bruger har mulighed for at oprette en opgave, og i visningen af opgaverne har brugeren mulighed for at fuldføre opgaven samt slette den.
Fuldføres opgaven bliver den opdateret som fuldført og vises under fuldførte opgaver.

Referencer

Easyxml (n.d.) available from <https://www.npmjs.com/package/easyxml> [24 March 2021]

Express 4.x - API Reference (n.d.) available from <http://expressjs.com/en/api.html#res.download> [24 March 2021]

Langehk/Projectweek12 (n.d.) available from <https://github.com/langehk/projectweek12> [24 March 2021]

Mongoose v5.12.2: SchemaTypes (n.d.) available from <https://mongoosejs.com/docs/schematypes.html> [24 March 2021]