

Projektuppgift

JavaScript-baserad webbutveckling

Projekt

REST-webbtjänst med MongoDB och JavaScript-ramverk

Albin Rönnkvist



Mittuniversitetet

MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. Campus Sundsvall Holmgatan 10, SE-851 70 Sundsvall.

Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.

Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MITTUNIVERSITETET
Avdelningen för informationssystem och -teknologi

Författare: Albin Rönnkvist, alrn1700@student.miun.se
Utbildningsprogram: Webbutveckling, 120 hp
Huvudområde: Datateknik
Termin, år: HT, 2018

Sammanfattning

I detta projekt skapar jag en typ av portfolio som fokuserar på kurser och skolprojekt där utvecklingsmiljön i projektet är JavaScript-baserad.

Rapporten kommer gå igenom två delar, utvecklingen av webbtjänsten och utvecklingen av klient-webbplatserna.

I webbtjänst-delen går jag igenom programsystemet *Node.js* med webbapplikations-ramverket *Express*. I denna del går jag även igenom databashantering med NoSQL-databasen *MongoDB* och *Mongoose* som används för att skapa databasscheman. Jag kommer också visa hur man laddar upp Node.js-applikationen och MongoDB-databasen till en molntjänst.

I klient-delen går jag igenom JavaScript-ramverket *Vue.js* och hur man konsumerar data från ett API med hjälp av HTTP-klienten *axios*.

Innehållsförteckning

Sammanfattning.....	iii
1 Introduktion.....	1
1.1 Bakgrund och problemmotivering.....	1
1.2 Avgränsningar.....	1
1.3 Detaljerad problemformulering.....	1
1.3.1 Översikt.....	1
1.3.2 Webbtjänst.....	1
1.3.3 Klient-webbplatser.....	2
2 Teori.....	3
3 Metod.....	5
3.1.1 Webbtjänst.....	5
3.1.2 Klient-webbplatser.....	5
4 Konstruktion.....	6
4.1 Webbtjänst.....	6
4.2 Klient-webbplatser.....	12
4.2.1 Administrations-webbplats.....	12
4.2.2 Publik webbplats.....	18
5 Resultat.....	19
5.1.1 Webbtjänst.....	19
5.1.2 Klient-webbplatser.....	19
6 Slutsatser.....	20
Källförteckning.....	21

1 Introduktion

1.1 Bakgrund och problemmotivering

Tidigare har JavaScript främst använts i webbläsaren för att skapa dynamiskt innehåll men med introduktionen av NodeJS så har det blivit populärt att använda JavaScript vid utvecklingen på serversidan också. En fördel med detta är att man som utvecklare kan använda samma programmeringsspråk på både server- och klientsidan. Om man kombinerar NodeJS med en NoSQL-databas som till exempel MongoDB så får man även en JavaScript-vänlig datastruktur i JSON-format. Man kan då följa samma datamodell vid utvecklingen av applikationen.

I detta projekt ska jag skapa en portfolio-webbplats med fokus på skolarbeten. Jag kommer använda en JavaScript-baserad utvecklingsmiljö med NodeJS, MongoDB och Vue.js.

1.2 Avgränsningar

Fokus i detta projekt ligger mer på back-end med NoSQL, NodeJS, Express mm men kommer även beröra lite front-end med JavaScript-ramverk. Dock så kommer jag inte lägga något större fokus på användbarhet och den visuella biten.

För att förstå denna rapport så bör man ha grundläggande kunskaper inom HTML, JavaScript, databaser, Git och REST-baserade webbtjänster.

1.3 Detaljerad problemformulering

1.3.1 Översikt

Portfolio-webbplatsen kommer vara uppdelad i två fristående delar:

1. En publik webbplats där besökare endast kan läsa innehåll. Innehållet består av de kurser jag läst och de projekt jag genomfört. Besökaren ska kunna välja kurs från en meny. När besökaren navigerar in på en specifik kurs så ska kursens tillhörande projekt visas med översiktlig information om projektet. Navigerar besökaren in på ett specifikt projekt så ska en utförlig beskrivning visas tillsammans med en länk till webbplatsen samt eventuellt en länk till ett repository.
2. En administrations-webbplats där behöriga användare kan skapa, läsa, uppdatera och radera innehåll.

1.3.2 Webbtjänst

Jag börjar först med att skapa en REST-baserad webbtjänst där det går att utföra CRUD-funktioner(skapa, läsa, uppdatera och radera). Denna webbtjänst gör det

alltså möjligt att lagra och modifiera data i en databas. För att få bättre struktur på min data så kommer jag att skapa databasscheman. Det som ska lagras i databasen är kort information om kurserna och det som ingick i de samt kursens projekt med tillhörande information.

Först lagras då kurserna med följande fält:
id, kursnamn, taggar, sammanfattning, beskrivning och länk kurswebbplats.

Sedan lagras projekten med följande fält:
id, projektnamn, taggar, sammanfattning, beskrivning, länk till webbplats och länk till ett remote repository.

1.3.3 Klient-webbplatser

När jag skapat min webbtjänst med databasscheman och CRUD-funktioner så kan jag kommunicera med den via klient-webbplatser. Båda webbplatsernas gränssnitt kommer skapas med ett JavaScript-ramverk och eventuellt ett CSS-ramverk om tid finns över.

Administrations-webbplats

På administrations-webbplatsen skapar jag ett gränssnitt där det finns funktionalitet för att skapa, läsa, uppdatera och radera innehåll(kurser och projekt). Detta görs mot webbtjänsten som läser in och modifierar data.

Publik webbplats

På den publika klient-webbplatsen ska jag skapa ett gränssnitt där besökare kan läsa innehåll. Jag ansluter då till webbtjänsten och läser ut data från databasen.

2 Teori

JavaScript

JavaScript är ett kraftfullt skriptspråk som initialt skapades för att göra webbssidor mer ”levande”. Det används till exempel ofta till att skapa dynamiskt innehåll och kontrollera vad som ska ske vid olika event som ett musklick eller när besökaren scollar. Skriptspråket används även i andra sammanhang utanför webben så som i Node.js och vissa databaser, som CouchDB och MongoDB. Det som gör JavaScript unikt är möjligheten att ändra HTML och CSS på ett relativt simpelt sätt. Man kan till exempel lägga till och ändra innehåll dynamiskt, kontrollera vad som sker vid olika event, kommunicera med olika servrar, ladda upp och ladda ned filer(t.ex. med AJAX-tekniker). Skriptspråket stöds även av alla de största webbläsarna och är aktiverat som standard. [1]

NoSQL(MongoDB)

NoSQL är en typ av databasdesign som består av teknologier för att lagra och hämta data. Som namnet poängterar så är NoSQL(Not Only SQL) inte baserat på SQL. Det är ett alternativ till de traditionella SQL-baserade relationsdatabaserna där NoSQL-system är distribuerade och icke relationsbaserade databaser, designade för att hantera lagring av stora mängder data. [2] [3]

MongoDB är en dokumentdatabas vilket är en typ av icke-relationsbaserad(NoSQL) databasdesign där man lagrar semistrukturerad data som dokument indelade i samlingar(collections) som fyller ungefär samma syfte som tabeller i relationsdatabaser. Varje dokument är självbeskrivande med metadata som definierar och beskriver datan samt relationen mellan olika dokument och samlingar. Dokumenten kan ha lika eller olika datastrukturer samt unika databasscheman men de behöver nödvändigtvis inte vara beroende av varandra [4][5]. De kan liknas vid en rad(row) i relationsdatabaser.

Denna modell underlättar för programmerare då data ofta representeras i form av JSON-dokument, vilket gör att man kan följa samma datamodell vid utvecklingen av själva applikationen [4].

Node.js med Express.js

Node.js är en open source-plattform där man kör JavaScript på servernivå. Node.js använder JavaScript-motorn V8 som översätter JavaScript till maskinkod som på så sätt kör servern. [6]

Med hjälp av Node.js kan man med JavaScript bygga mer avancerade applikationer som kräver att man har en server som kan skicka och lagra data. En fördel med att använda Node.js på servern är att man som utvecklare kan använda samma programmeringsspråk(Javascript) både på serversidan och på klientsidan. En till fördel med Node.js är pakethanteraren npm eller ”node packet manager” som används för att hämta paket med färdig kod som man kan återanvända i sitt projekt. På så sätt kan man hämta färdiga lösningar och implementera de i applikationen vilket sparar tid. [7]

Express.js är ett exempel på ett sådant paket. Express är ett ramverk som an-

vänds för att skapa webb- och mobilapplikationer på serversidan i Node.js. Med Express kan man bygga single-page, multi-page och hybrid mobil- och webbapplikationer. Man kan även bygga back-end-funktionalitet för webbapplikationer samt API:s. [8] [9]

JavaScript-baserade ramverk

JS-ramverk är verktyg för att snabba upp och effektivisera kodning med JavaScript. Istället för att bygga en webbplats helt från grunden och skapa egen kod till varenda del så kan man med hjälp av ramverk implementera mindre färdigkodade delar direkt i webbplatsen.

I grunden så är JS-ramverk samlingar av JavaScript-kodbibliotek som ger utvecklare färdigskriven kod att använda för vanliga, återkommande funktioner och utvecklarsysslor. Med andra ord, ramverk att bygga webbplatser och applikationer kring. [10]

Vue.js

Vue(eller Vue.js) är ett JavaScript-baserat ramverk med öppen källkod som används för att bygga användargränssnitt och single-page-applikationer.

I grunden är Vue.js ett system för att generera data till dokumentobjektmodellen(DOM) vilket är ett plattform- och språkoberoende gränssnitt som ger programspråk möjligheten att dynamiskt läsa och uppdatera ett dokumentets innehåll, struktur och formatering. Vue.js länkar ihop data med DOM med hjälp av egenskaper och direktiv, man brukar kalla detta för reaktivitet, att datan är reaktiv. Det betyder att när en egenskaps värde ändras i vyn så ändras den även i Vue och vice versa.

Vue.js använder sig mycket av direktiv för att göra bland annat bindningar, if-satser och händelselyssnare. Dessa direktiv inleds med "v-" vilket specificerar att det är ett speciellt attribut angivet av Vue.js och är till för att lägga till ett reaktivt beteende till DOM. [35]

Fördelen med Vue är att ramverket är väldigt lättviktigt samtidigt som det har hög prestanda. Det är även relativt lätt att lära sig med väl strukturerade mallar och välskrivna dokumentation. Vue är också flexibelt och kan hantera applikationer med JSX, ES6, routing och buntning vilket gör det lätt att byta till Vue för utvecklare som har erfarenhet med React, Angular eller andra JavaScript-ramverk eftersom designen är så lik. [36]

3 Metod

3.1.1 Webbtjänst

Data kommer lagras med NoSQL-databasen *MongoDB* och webbtjänsten kommer skapas med *Node.js* och ramverket *Express*. Jag kommer använda *Mongoose* för att skapa databasscheman. För att slippa avsluta och starta om servern efter varje ändring så kommer jag använda mig av *nodemon* som gör detta automatiskt.

REST-webbtjänsten kommer publiceras i molntjänsten *Heroku* som kan hosta *Node.js*-applikationer.

MongoDB-databasen kommer publiceras i databas-molntjänsten *mLab*.

Klient-webbplatserna kommer inte kräva något speciellt webbhotell.

Innan jag publicerar REST-webbtjänsten så kommer jag testa den med verktyget *Advanced REST client*.

3.1.2 Klient-webbplatser

Jag kommer skapa två olika klient-webbplatser, en publik och en för administration. Eftersom webbplatserna kommer vara relativt enkla och små så kommer jag använda *Vue.js* som frontend-ramverk för att skapa gränssnitten. Tillsammans med *Vue.js* kommer jag att använda HTTP-klienten *axios* som är ett populärt tillvägagångssätt för att konsumera data från ett API.

Webbplatsernas struktur och funktionalitet kommer beskrivas i textform. Finns tid över så kommer jag även använda CSS-ramverket *Materialize* för att göra webbplatserna mer visuellt tilltalande.

4 Konstruktion

4.1 Webbtjänst

Initiera npm

Jag börjar med att initiera npm i projektet med kommandot: `”npm init”`. Detta kommando genererar en `package.json`-fil i root-katalogen som används för att ge information till npm som kan identifiera projektet och hantera projektets beroenden(dependencies). [11]

Installera Express

Efter det installerar jag npm-paketet Express som är ramverket jag kommer använda för att skapa min webbtjänst [12]. Detta görs i terminalen med kommandot `”npm install express --save”` som då sparar paketet till projektet. Paketet sparas mer specifikt i en mapp vid namn `node_modules` som ligger i root-katalogen och definieras i dependencies-egenskapen i `package.json`, detta gäller för alla dependencies. [13]

Installera body-parser

Jag behöver även installera middleware-modulen `body-parser` som används för att hantera HTTP POST-data i Express.js. Denna middleware var en del av Express.js tidigare men efter version 4 så måste den installeras separat och detta görs med kommandot: `”npm install body-parser --save”`. [14]

Installera övriga npm-paket

Installera `path`[16] som hjälper till att navigera i filer och kataloger, `mongoose`[17] för att skapa databasscheman, `cors`[33] för att tillåta anrop från olika domäner samt `method-override`[34] som tillåter att man kan använda HTTP-verb såsom PUT och DELETE där klienten inte stöder det.

Importerera dependencies

När alla paket är installerade så kan jag skapa min huvudapplikation som jag döper till `app.js` i root-katalogen. Jag börjar då med att importera alla dependencies(npm-paket) och spara de i variabler som jag kan använda i applikationen. Detta görs med `require`-modulen som tar sökvägen till varje dependency som parameter. Exempel: `const express = require("express");`. [15]

Inkludering och inställningar

När `express`-paketet är inkluderat så skapar jag en ny instans av Express som jag lagrar i en variabel för att på så sätt kunna nå den i resten av koden:

```
var app = express();
```

Efter det gör jag det möjligt för klienten att använda en header(`X-HTTP-Method-Override`) för att ersätta HTTP-metoder. Det görs genom att anropa `app.use()` och modulen `method-override`:

```
app.use(methodOverride('X-HTTP-Method-Override'));
```

Jag använder sedan modulen *body-parser* för att ange att data i applikationen ska returneras i JSON-format:

```
app.use(bodyParser.json());
```

Jag anger även att applikationen ska parse data med Content-Type "application/x-www-form-urlencoded":

```
app.use(bodyParser.urlencoded({extended: false}));
```

Till sist gör jag det möjligt för klienter att skicka alla typer av anrop från olika domäner med hjälp av modulen *cors*:

```
app.use(cors());
```

Testkör applikation

När jag har gjort klart alla inställningar och förberedelser så kan jag testköra applikationen för att se att allt fungerar. För att köra applikationen så måste jag starta servern och det görs med *app.listen()*-metoden. Denna metod binder och lyssnar efter anslutningar på porten som angivs i den första parametern: `app.listen(port, () => {})`. I detta fall är "port" en variabel med värdet 3000 vilket innebär att servern kommer startas på port 3000. [18]

Anslut till databas

När jag vet att applikationen är körbar så kan jag börja konstruera databasen och alla databasscheman. Jag börjar då med att ansluta till MongoDB-databasen med hjälp av modulen *mongoose* och *mongoose.connect()*-metoden:

```
mongoose.connect("mongodb://localhost:27017/courses");
```

Denna databas ligger lokalt på datorn där standardporten är "27017" och det angivna namnet på databasen är "courses". Denna databasanslutning kommer uppdateras senare när databasen ligger på molntjänsten mLab.

För att kunna ansluta till MongoDB-databasen så måste självklart databaservern vara igång och den startar jag genom att öppna terminalen och skriva in kommandot "mongod".

Konstruera databasscheman

När databasanslutningen fungerar så kan jag börja skapa mina databasscheman som bestämmer hur databasen ska vara strukturerad. Varje schema pekar till en MongoDB-samling(collection) och definierar strukturen på dokumenten i den samlingen [19]. Under katalogerna *app*→*models* skapar jag då en JS-fil vid namn *courses* där mina databasscheman ligger.

I denna fil importerar jag först modulen *mongoose* och sedan skapar jag en referens till Schema-klassen:

```
var Schema = mongoose.Schema;
```

Efter det kan jag börja strukturera mina scheman. Det schema jag skapar är för varje kurs och dess projekt / uppgifter (Se Figur 1). Notera även att jag inte lägger till något id-fält, detta behövs inte då MongoDB genererar ett unikt id automatiskt.

```
app ▸ models ▸ JS courses.js ▸ coursesSchema
14 // Schema for courses
15 var coursesSchema = new Schema({
16   courseName: String,
17   tags: String,
18   summary: String,
19   description: String,
20   courseWebsite: String,
21   projects: [{
22     projectName: String,
23     projectTags: String,
24     projectSummary: String,
25     projectDescription: String,
26     projectWebsite: String,
27     projectRepository: String
28   }]
29 });
```

Figur 1.

Exportera databasscheman

När databasschemat är strukturerat så exporterar jag det så det kan användas genom att göra en instans av det. Detta görs med *module.exports* som är ett speciellt objekt som är inkluderat i varje JS-fil som standard i Node.js-applikationer. *module* är en variabel som representerar den nuvarande modulen och *exports* är ett objekt som kommer visas som en modul. Vad som än tilldelas till detta objekt kommer alltså visas som en modul [20]. I detta fall tilldelar jag objektet en mongoose-modell (*mongoose.model*) som används för att hantera skapande och hämtning av dokument från databasen [21]. I denna metod skickar jag med en instans av databasschemat:

```
module.exports = mongoose.model("Courses", coursesSchema);
```

Man kan då inkludera *courses.js*-filen i applikationen vilket gör att man får tillbaka objektet *Courses* som innehåller *coursesSchema*.

Efter det läser jag in schemat i applikationen med *require*-modulen:

```
var Courses = require("../app/models/courses.js");
```

Konstruera REST-api

Nu kan jag börja konstruera mitt REST-api för kurserna. Jag behöver skapa funktionalitet för anropen *GET*, *POST*, *PUT* och *DELETE*. Detta görs i Express med så kallad "routing". Routing innebär att man bestämmer hur en applikation ska svara på klient-anrop till en viss sökväg och med specifika HTTP-anropsmetoder även kallade "routing methods". Man definierar en route enligt följande struktur:

```
app.METHOD(PATH, HANDLER)
```

Där *app* är instansen av Express, *METHOD* är en HTTP-anropsmetod (i små bokstäver), *PATH* är en sökväg på servern och *HANDLER* är funktionen som exekveras när ett klient-anrop görs till routen. [22] [23]

- **GET:**

```
app.get("/api/courses", (req, res) => {});
```

För GET-anrop (läsa) använder jag routing-metoden "get" där jag sedan i första parametern anger sökvägen för anropet och i andra parametern en handler vilket är en funktion som tar två parametrar: *req* är det som anropas och *res* är det objekt som skickas tillbaka från anropet.

Detta anrop används för att skicka tillbaka all kurser till klienten. Det görs genom att använda *Courses*-modellen och *find()*-metoden för att läsa ut data från databasen och sedan skicka tillbaka resultatet i JSON-format (se Figur 2).

```
JS app.js ▶ app.get("/api/courses") callback
40 // Find courses-documents in DB and send result
41 Courses.find((err, Courses) => {
42 // If there was an error
43 if(err) {
44 // Send error-message
45 res.send(err);
46 }
47 else {
48 // Convert result to JSON and send
49 res.json(Courses);
50 }
51 });
```

Figur 2.

Innan jag går vidare till nästa routing-metod så testar jag att denna routing-metod fungerar som tänkt. Jag använder mig då av Googles Advanced REST client där jag skickar ett HTTP-anrop med metoden GET till korrekt sökväg på servern (se Figur 3).



Figur 3.

Eftersom jag inte lagt till någon kurs ännu så returnerar detta anrop en tom array men jag vet nu att anropet fungerar då jag inte får tillbaka något felmeddelande.

- **POST:** `app.post("/api/courses/add", (req, res) => {});`. För POST-anrop(skapa) använder jag routing-metoden "post". Jag börjar med att skapa en ny instans av databasschemat *Courses*. Efter det skapar jag ett nytt objekt utifrån det som skickades med i anropet. Här måste jag läsa in alla fält som ska lagras, en för en(samma fält som databasschemat). Efter det lagrar jag objektet i databasen med *save()*-metoden (se Figur 3).

```
JS app.js ▸ app.post("/api/courses/add") callback
67 // New instance of Courses schema
68 var course = new Courses();
69
70 // Create new object
71 course.courseName = req.body.courseName;
72 course.tags = req.body.tags;
73 course.summary = req.body.summary;
74 course.description = req.body.description;
75 course.courseWebsite = req.body.courseWebsite;
76 course.projects = req.body.projects;
77 course.projectName = req.body.projectName;
78 course.projectTags = req.body.projectTags;
79 course.projectSummary = req.body.projectSummary;
80 course.projectDescription = req.body.projectDescription;
81 course.projectWebsite = req.body.projectWebsite;
82 course.projectRepository = req.body.projectRepository;
83
84 // Store new course in database
85 course.save((err) => {
86 // If there was an error
87 if(err) {
88 // Send error message
89 res.send(err);
90 }
91 });
```

Figur 3.

Jag testar sedan detta anrop med Advanced REST client med POST-metoden och en body med alla fält i JSON-format. Jag behöver inte ange ett id då ett unikt sådant genereras automatiskt. Jag kan sedan använda GET-metoden igen för att se om objektet har lagrats korrekt i databasen.

- **DELETE:** `app.delete("/api/courses/delete/:id", (req, res) => {});`

För DELETE-anrop(radera) använder jag routing-metoden "delete". I sökvägen anger jag även ":id" som används för att läsa in id direkt från adressraden i anropet vilket jag sedan kan använda i applikationen för att ta bort en specifik kurs. För att läsa in id:t från adressraden så hämtar jag id-parametern från anropet med följande kod:

```
var deleteId = req.params.id;
```

Sedan använder jag *Courses*-modellen och metoden *deleteOne()* som raderar det första dokumentet som matchar villkoret angivet i den första parametern, i detta fall id:t(deleteId) från adressraden i anropet (se Figur 4). Det betyder att vi raderar en kurs från databasen som matchar det angivna id:t i anropet. Notera även att *deleteOne()*-metoden endast raderar ett dokument åt gången, oavsett villkoret [24].

```
JS app.js ▸ app.delete("/api/courses/delete/:id") callback
104 =   Courses.deleteOne({
105 =     _id: deleteId
106 =   }, (err, Courses) => {
107 =     if(err) {
108 =       res.send(err);
109 =     }
110 =   });
```

Figur 4.

Jag testar även detta anrop med Advanced REST client.

- **PUT:** `app.put("/api/courses/update/:id", (req, res) => {});`. För PUT-anrop (uppdatera) använder jag routing-metoden "put". Jag anger även ":id" i sökvägen på samma sätt som för DELETE-anrop vilket jag sedan hämtar på precis samma sätt:

```
var updateId = req.params.id;
```

Jag använder sedan *Courses*-modellen och *findOne()*-metoden som hittar ett dokument som matchar villkoret i den första parametern, i detta fall `id:t(updateId)` från adressraden i anropet:

```
Courses.findOne({_id: updateId}, (err, course) => {});
```

Om `id:t` från anropet matchar ett dokument id så returneras ett objekt som jag kan ge nya värden och återigen spara i databasen med *save()*-metoden på samma sätt som i POST-anrop (se Figur 3).

Ladda upp databas på mLab

För att ladda upp den lokala MongoDB-databasen till internet så använder jag mig av molntjänsten mLab. I gränssnittet skapar jag en ny databas med gratisalternativet "SANDBOX", sen väljer jag en region där databasen ska ligga och till sist döper jag den nya databasen till "coursesdb".

För att kunna ansluta till databasen så behöver jag en adress till databasen med dess namn samt ett användarnamn och lösenord. Användaren skapar jag i databasen under "Users"-fliken.

När användaren är skapad så genererar mLab en URI för att ansluta till databasen:

```
"mongodb://<dbuser>:<dbpassword>@ds161620.mlab.com:61620/coursesdb"
```

blir då mitt URI där jag lägger till mitt skapta användarnamn och lösenord.

Denna sträng läggs till i *mongoose-connect()*-metoden som vi gick igenom under "Anslut till databas"-sektionen ovan.

Ladda upp applikation på Heroku

Till sist laddar jag upp Node.js-applikationen till molntjänsten Heroku.

Jag börjar då med att ladda ned Heroku CLI där jag sedan skapar ett konto som jag loggar in med i terminalen.

Efter det skapar jag en Heroku-fil med namnet Procfile i root-katalogen vilket talar om för Heroku vilken typ av server den ska köra och vilka kommandon den måste köra för att starta applikationen. I denna fil skriver jag sedan in följande information: "web: node app.js". "Web" innebär att den ska köra servern

som en webbserver och ”node app.js” är kommandot för att köra Node.js-applikationen.

Efter det initierar jag git i projektet med kommandot ”git init” och efter det skriver jag in kommandot ”heroku create” som skapar ett namn för applikationen.

Efter det lägger jag till alla ändringar till Heroku med de vanliga git-kommandona ”git add .” och ”git commit” för att sedan pusha ändringarna till servern med kommandot ”git push heroku master”.

För att öppna applikationen så kör jag kommandot ”heroku open” vilket öppnar en ny flik med URL:en till den publicerade webbtjänsten(<https://albincourses.herokuapp.com/>).

Jag kan nu använda denna URL till att göra anrop till webbtjänsten från klient-webbplatserna, till exempel ”<https://albincourses.herokuapp.com/api/courses>” för GET-anrop.

[27]

Länkar

Repository: <https://github.com/albinronnkvist/coursesWebService/tree/master>

4.2 Klient-webbplatser

4.2.1 Administrations-webbplats

Översikt

Två tabbar:

1. *Start-tabbar* där användare kan välja en kurs i en <select>-meny. När användaren har valt en kurs så visas kursens information i en sektion under menyn. I samma sektion finns det två knappar där användaren antingen kan redigera eller radera kursen. Vid klick på radera så raderas kursen från databasen och sidan laddas om. Annars om användaren klickar på redigera så visas ett popup-fönster med ett formulär som är ifyllt med kursens information. Här kan användare göra ändringar och skicka formuläret som då uppdaterar kursen i databasen och laddar om sidan. Användaren kan även välja att avbryta uppdatering vilket stänger popup-fönstret.
2. *Skapa-tabbar* där användaren kan skapa en ny kurs. Här visas ett formulär där användaren kan fylla i information om kursen och sedan skicka vilket lagrar den nya kursen i databasen och laddar om sidan.

Skapa projekt

Jag börjar med att skapa ett nytt projekt för administrations-webbplatsen där jag initierar git för versionshantering och npm för att kunna installera Gulp vilket jag kommer använda för att automatisera utvecklarsysslor. Gulp är ej nödvändig för denna lösning så jag kommer inte gå igenom det i denna rapport, det är endast till för att underlätta utvecklingen.

Inkludera Vue.js

Jag använder Vue.js som ramverk för att skapa gränssnittet.

Jag börjar då med att skapa en ny HTML-fil där jag inkluderar Vue.js via ett in-

nehållsleveransnätverk(CDN) i en <script>-tagg:

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

Det finns två olika versioner, produktversionen och utvecklarserversionen. Nu i utvecklingsfasen använder jag mig av utvecklarserversionen då den innehåller hjälpmedel för felsökning. När jag sedan ska publicera webbplatsen så kommer jag använda produktversionen eftersom den är optimerad för storlek och prestanda.

Inkludera Axios

Jag inkluderar Axios via ett CDN på samma sätt som för Vue:

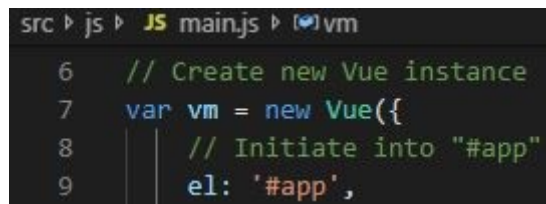
```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Skapa ny JS-fil för Vue.js-kod

När Vue.js är inkluderat i projektet så skapar jag en ny JavaScript-fil där all min Vue.js-kod kommer ligga, även denna fil inkluderas i HTML-filen. Här gör jag bland annat mina anrop med hjälp av ett JavaScript-bibliotek som heter Axios vilket underlättar utformningen av dessa anrop.

Anrop till webbtjänst:

new Vue({}): Jag börjar med att skapa en ny instans av Vue.js som jag initierar i elementet med id "app" (se Figur 5). På HTML-sidan eller i den så kallade "vyn", alltså det som används kommer att se, lägger jag till ett element med id "app". Det är i detta element som Vue.js kommer generera all kod, det är själva Vue.js-projektet.



```
src ▸ js ▸ JS main.js ▸ vm
6 // Create new Vue instance
7 var vm = new Vue({
8   // Initiate into "#app"
9   el: '#app',
```

Figur 5.

Data(): När Vue.js-instansen är skapad så lägger jag till alla egenskaper(properties) i Vue.js `data()`-funktion (se Figur 6.). Vue konverterar egenskaperna till getters/setters för att göra dessa "reaktiva" [30]. När värdena för dessa egenskaper ändras så "reagerar" vyn på det och uppdateras för att matcha de nya värdena. Dock så uppdateras inte egenskaper i vyn automatiskt om de inte existerade vid instansieringen. Därför måste jag sätta ett initialt värde på egenskaperna [28]. I detta fall är mina egenskaper objekt med ett initialt värde av null, vilket innebär att objekten finns men att de inte har någon typ eller värde [29].

```
src ▾ js ▾ JS main.js ▾ vm ▾ data
13 data () {
14   return {
15     // GET courses
16     courses: null,
17     specificCourse: null,
18     selectedCourseId: null,
19
20     // POST courses & projects
21     postCourseName: null,
22     postTags: null,
23     postSummary: null,
24     postDescription: null,
25     postCourseWebsite: null,
26     postProjectName: null,
27     postProjectTags: null,
28     postProjectSummary: null,
29     postProjectDescription: null,
30     postProjectWebsite: null,
31     postProjectRepository: null,
32
33     // PUT courses & projects
34     putCourseName: null,
35     putTags: null,
36     putSummary: null,
37     putDescription: null,
38     putCourseWebsite: null,
39     putProjectName: null,
40     putProjectTags: null,
41     putProjectSummary: null,
42     putProjectDescription: null,
43     putProjectWebsite: null,
44     putProjectRepository: null
45   }
46 }
```

Figur 6.

mounted(): Efter det så använder jag *mounted()*-funktionen som körs så fort applikationen har renderat klart / lagts till i DOM. Jag använder denna funktion för att läsa in metoder och egenskaper direkt vid inladdning av applikationen (se Figur 7).

```
src ▾ js ▾ JS main.js ▾ vm ▾ mounted
48 mounted () {
49   this.getAllCourses();
50 }
```

Figur 7.

Methods-objektet: Jag definierar sedan alla metoder i *methods*-objektet: *getAllCourses()*, *getSpecificCourse(id)*, *addCourse()*, *updateCourse(id)*, *setFormData()* och *deleteCourse(id)*. Jag går igenom dessa metoder mer specifikt i nästa sektion.

Metoder:

- **GET**

- getAllCourses():**

- I denna metod använder jag axios med *axios.get()*-funktionen som tar url:en till webbtjänsten som parameter. *then()*-funktionen talar sedan om för axios vad som ska hända när vi har fått tillbaka ett svar från webb-

tjänsten. Denna funktion tar emot en anonym funktion som parameter med ett svar från anropet. I den anonyma funktionen kan jag sedan använda datan som jag fick tillbaka i svaret från webbtjänsten för att lägga till värden till mina egenskaper, i detta fall ett JSON-objekt med kurser och projekt som jag lägger till i egenskapen "courses" (se Figur 8).

```
src > js > JS main.js > vm > getAllCourses
23 // Get all courses, store in "course"
24 getAllCourses() {
25   var self = this;
26   axios
27     .get('https://albincourses.herokuapp.com/api/courses')
28     .then(response => {
29       self.courses = response.data;
30     })
31     .catch(error => {
32       console.log(error.response);
33     });
34 },
```

Figur 8.

Jag använder sedan egenskapen `courses` för att läsa in dess värden i `vyn`. Jag läser in kursnamnen i en `<select>`-meny med hjälp av `v-for`-direktivet som genererar en lista med element från en array:

```
<option v-bind:value="course._id" v-for="course in courses">
```

`</option>`. "courses" är arrayen med data och "course" är ett alias för det specifika array-element som itereras över [31].

Med hjälp av `v-bind`-direktivet så sätter jag `<option>` value-attributet till kursernas id (`course_id`).

Jag använder sedan `v-model`-direktivet i `<select>`-menyn som lagrar det valda `<option>`-elementets värde i egenskapen `selectedCourseId`. I `select`-menyn använder jag även `v-on`-direktivet eller `@change` som är en förkortning av `v-on:change`:

```
<select v-model="selectedCourseId" @change="getSpecificCourse(selectedCourseId)"
```

`>`. `V-on` är ett direktiv som används för att lyssna efter händelser i DOM och `change` är en händelselyssnare som i detta fall lyssnar efter ändringar av val i `<select>`-menyn. Denna händelselyssnare kör metoden `getSpecificCourse()` som tar id:t från egenskapen `selectedCourseId` som parameter [32]. Detta id används sedan för att göra ett nytt anrop i `getSpecificCourse()`-metoden där jag endast hämtar en kurs och dess projekt.

getSpecificCourse(id):

I denna metod jag gör ett nytt GET-anrop med `axios` fast med "id" som en extra Url-parameter vilket pekar på en specifik kurs:

```
'https://albincourses.herokuapp.com/api/courses/' + id.
```

Svaret från webbtjänsten lagras i egenskapen `specificCourse`. Jag kan sedan loopa igenom värdena i detta objekt genom att endast ange objektets namn och de värden jag vill skriva ut. Eftersom detta objekt har värdet `null` innan användaren har valt en kurs så anger jag att information endast ska skrivas ut om en kurs är vald, dvs om objektet inte är tomt

utan det har fått ett värde som inte är null. För detta använder jag *v-if*- och *v-else*-direktiven som skriver ut olika information beroende på om *specificCourse*-egenskapen har värdet null eller om den har fått ett värde:

```
v-if="specificCourse != null".
```

- **POST**

- addCourse():**

- I denna metod använder jag *axios.post()*-funktionen som tar emot 3 parametrar ('url', {data} och {config}).

- Url är helt enkelt Url:en till webbtjänsten med "add" som en extra Url-parameter:

- `'https://albincourses.herokuapp.com/api/courses/add'`.

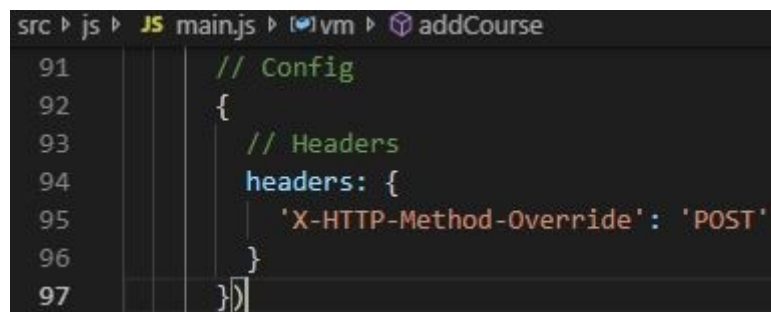
- I data-parametern anger jag all data som användaren skickar med i anropet, i detta fall ska alla värden från *courses*-objektet i Figur 3 skickas med. All data initieras i POST-egenskaperna (*postCourseName*, *postTags*, *postSummary*... etc.) med värdet null i *data()*-funktionen (se Figur 6). Jag binder sedan dessa egenskaper till ett formulär med *v-model*-direktivet där formulär-elementen automatiskt uppdaterar egenskaperna:

- `<input type="text" v-model="postProjectName">`.

- När användaren sedan skickar formuläret så anropas *addCourse()*-funktionen med *v-on*-direktivet (@) med *submit* som händelselyssnare och *.prevent*-modifieraren som hindrar *submit*-händelselyssnaren från att ladda om sidan:

- `<form @submit.prevent="addCourse">`.

- I config anger jag headern X-HTTP-Method-Override med metoden POST. Gör jag inte detta så skickar klienten ett GET-anrop, men med headern så ersätts GET av POST så att det blir ett korrekt POST-anrop (se Figur 9).



```
src > js > JS main.js > | vm > addCourse
91 // Config
92 {
93   // Headers
94   headers: {
95     'X-HTTP-Method-Override': 'POST'
96   }
97 }
```

Figur 9.

- **PUT**

- **updateCourse(id):**

- Denna metod fungerar nästan precis likadant som *addCourse()*. Jag använder *axios.put()*-funktionen som tar emot samma parametrar som *axios.post()*, ('url', {data} och {config}).

- Url till webbtjänsten med "update" och "id" som extra Url-parametrar:
`'https://albincourses.herokuapp.com/api/courses/update/' + id.`

- Datan som ska skickas med initieras i PUT-egenskaperna (*putCourseName*, *putTags*, *putSummary*... etc.) i *data()*-funktionen (se Figur 6) och binds till ett formulär med *v-model*-direktivet.

- För att användaren ska slippa skriva om all information om kursen i formuläret så läser jag in de redan befintliga värdena från databasen till formulär-elementen. Så när användaren klickar på länken för att uppdatera så kommer uppdaterings-formuläret vara ifyllt med kursens och projektens data. Detta görs med ett klick-event som kör metoden *setFormData()*:

- I *setFormData()*-metoden ger jag PUT-egenskaperna värden som jag hämtar från *specificCourse*-objektet på följande vis:

- `self.putCourseName = self.specificCourse.courseName;`

- Eftersom dessa egenskaper är bundna till formuläret med *v-model*-direktivet så kommer formulärets element vara ifyllda med egenskapernas värden.

- I config anger jag headern X-HTTP-Method-Override med metoden PUT:

- `'X-HTTP-Method-Override': 'PUT'.`

- **DELETE**

- **deleteCourse(id):**

- Funktionen tar emot ett id som parameter. Id:t är den specifika kursens id och skickas med från händelselyssnaren:

- `<a @click.prevent="deleteCourse(specificCourse._id)">.`

- Jag använder sedan *axios.delete()*-funktionen med Url:en till webbtjänsten som innehåller två extra Url-parametrar, "delete" och "id" vilket pekar på en specifik kurs:

- `'https://albincourses.herokuapp.com/api/courses/delete/' + id.`

- Jag anger även headern X-HTTP-Method-Override i config-parametern fast nu med metoden DELETE:

- `'X-HTTP-Method-Override': 'DELETE'.`

Länkar

Webbplats: <https://albinronnkvist.se/dt162g/projekt/admin/>

Repository: <https://github.com/albinronnkvist/coursesAdminWebsite>

4.2.2 Publik webbplats

Översikt

Användare kan välja en kurs i en <select>-meny. När användaren har valt en kurs så visas kursens information i en sektion under menyn.

Skapa projekt

Den publika webbplatsen är ungefär likadan som administrations-webbplatsen fast den använder endast funktionaliteten för att läsa data. Den enda skillnaden jag gör är då att ta bort all funktionalitet och innehåll som berör POST-, PUT- och DELETE-anrop.

Länkar

Webbplats: <https://albinronnkvist.se/dt162g/projekt/public/>

Repository: <https://github.com/albinronnkvist/coursesPublicWebsite>

5 Resultat

Jag har med hjälp av en JavaScript-baserad utvecklingsmiljö skapat en webbtjänst och två klient-webbplatser för att läsa och administrera kurser och projekt.

5.1.1 Webbtjänst

Webbtjänsten är REST-baserad där det går att utföra CRUD-funktioner (skapa, läsa, uppdatera och radera). Webbtjänsten är skapad med Node.js och Express samt övriga moduler. Data lagras i NoSQL-databasen MongoDB med databas-scheman skapade med Mongoose.

Webbtjänsten är publicerad i molntjänsten Heroku och databasen i molntjänsten mLab.

5.1.2 Klient-webbplatser

Jag har skapat två klient-webbplatser, en publik och en för administration. Båda webbplatsernas gränssnitt är skapade med JavaScript-ramverket Vue.js och CSS-ramverket Materialize. Webbplatserna konsumerar webbtjänsten med hjälp av HTTP-klienten axios.

Administrations-webbplats

På administrations-webbplatsen finns funktionalitet för att skapa, läsa, uppdatera och radera innehåll (kurser och projekt).

Publik webbplats

På den publika webbplatsen finns funktionalitet för att läsa innehåll.

6 Slutsatser

Jag tycker att själva utvecklingen i projektet har rullat på ganska bra och jag har uppnått det jag planerade i problemmotiveringen. Jag känner också att jag fått en bra grund för att fortsätta lära mig mer om JavaScript-lösningar med Node.js, Express, NoSQL och JS-ramverk. Sen finns det självklart så mycket mer att lära sig och vidareutveckla.

Jag vill främst lära mig mer om JavaScript-ramverk eftersom det känns som om jag kunde löst det på ett snyggare sätt med till exempel komponenter och templates men eftersom det är så små webbplatser med lite funktionalitet så går det ändå att förstå hur Vue-applikationen är uppbyggd.

Jag hade tänkt lägga till säkerhet med ett inloggningssystem och restriktioner gällande HTTP-anrop men jag kände att jag inte skulle hinna och rapporten är redan lång som den är. Det är något som kan vidareutvecklas i projektet och något som skulle vara nyttigt eller till och med nödvändigt att lära sig.

Tyvärr fick jag lägga en del tid på problem som uppstod på grund av Materialize och ramverkets sätt att hantera dynamisk data i bland annat `<select>`-element och `<textarea>`-element. Även fast designen inte var prioriterad så hade det varit kul om allting såg någorlunda stilrent ut men nu fick jag göra några egna lösningar i sista minut som inte blev helt perfekt.

Något som kanske inte spelar så jättestor roll i ett så litet projekt men som jag ändå kunde gjort bättre är att planera databasens fält bättre. Nu lagrar jag vissa fält som inte ens används på webbplatserna vilket tar upp onödig plats i databasen och tillförde överflödigt jobb vid utvecklingen.

Källförteckning

- [1] JavaScript.info, "An introduction to JavaScript"
<http://javascript.info/intro> Hämtad 2019-01-01
- [2] SearchDataManagement, "NoSQL (Not Only SQL database)"
<https://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL> Publicerad 2017-03. Hämtad 2019-01-01
- [3] spring, "Understanding NoSQL"
<https://spring.io/understanding/NoSQL> Hämtad 2019-01-01
- [4] aws, "What Is a Document Database?"
<https://aws.amazon.com/nosql/document/> Hämtad 2019-01-01
- [5] Wikipedia, "Document-oriented database" https://en.wikipedia.org/wiki/Document-oriented_database Publicerad 2018-09-20. Hämtad 2019-01-01
- [6] tjejdkodar, "Vad är Node.js" <http://www.tjejdkodar.se/blogg/vad-ar-node-js/> Publicerad 2018-05-06. Hämtad 2019-01-01
- [7] Cerpus, "Node.js – Hur fungerar det?" <https://cerpus.se/artiklar/nodejs-hur-fungerar-det/> Hämtad 2019-01-01
- [8] Express, "Fast, unopinionated, minimalist web framework for Node.js"
<https://expressjs.com/> Hämtad 2019-01-01
- [9] Upwork, "Express.js: A Server-Side JavaScript Framework"
<https://www.upwork.com/hiring/development/express-js-a-server-side-javascript-framework/> Hämtad 2019-01-01
- [10] Skillcrush, "What Is a JavaScript Framework? Here's Everything You Need to Know" <https://skillcrush.com/2018/07/23/what-is-a-javascript-framework/> Publicerad 2018-08-15. Hämtad 2019-01-01
- [11] nodejitsu, "What is the file `package.json`?"
<https://docs.nodejitsu.com/articles/getting-started/npm/what-is-the-file-package-json/> Publicerad 2011-08-26. Hämtad 2019-01-01
- [12] npm, "express" <https://www.npmjs.com/package/express> Hämtad 2019-01-01

- [13] Nodestsource, "The Basics of Package.json in Node.js and npm"
<https://nodesource.com/blog/the-basics-of-package-json-in-node-js-and-npm/> Publicerad 2017-05-03. Hämtad 2019-01-01
- [14] npm, "body-parser" <https://www.npmjs.com/package/body-parser> Hämtad 2019-01-01
- [15] freeCodeCamp, "Requiring modules in Node.js: Everything you need to know" <https://medium.freecodecamp.org/requiring-modules-in-node-js-everything-you-need-to-know-e7fbd119be8> Publicerad 2017-05-19. Hämtad 2019-01-01
- [16] Node.js, "Path" <https://nodejs.org/api/path.html> Hämtad 2018-01-01
- [17] mongoose, "elegant mongodb modeling for node.js" <https://mongoosejs.com/> Hämtad 2019-01-01
- [18] Express, "app.listen" <https://expressjs.com/en/api.html#app.listen> Hämtad 2019-01-01
- [19] mongoose, "Schemas" <https://mongoosejs.com/docs/guide.html> Hämtad 2019-01-02
- [20] TutorialTeacher, "Export Module in Node.js" <http://www.tutorialsteacher.com/nodejs/nodejs-module-exports> Hämtad 2019-01-02
- [21] mongoose, "Models" <https://mongoosejs.com/docs/4.x/docs/models.html> Hämtad 2019-01-02
- [22] Express, "Basic routing" <https://expressjs.com/en/starter/basic-routing.html> Hämtad 2019-01-03
- [23] Express, "Routing" <https://expressjs.com/en/guide/routing.html> Hämtad 2019-01-03
- [24] mongoose, "Model.deleteOne()" https://mongoosejs.com/docs/api.html#model_Model.deleteOne Hämtad 2019-01-04
- [25] Express, "app.all(path, callback [,callback ...])" <https://expressjs.com/en/api.html#app.all> Hämtad 2019-01-06
- [26] Dev.Opera, "DOM Access Control Using Cross-Origin Resource Sharing" <https://dev.opera.com/articles/dom-access-control-using-cors/> Publicerad 2011-12-14. Hämtad 2019-01-06

- [27] Appdividend, "How To Deploy Nodejs App To Heroku"
<https://appdividend.com/2018/04/14/how-to-deploy-nodejs-app-to-heroku/> Publicerad 2018-12-29. Hämtad 2019-01-09
- [28] Vue.js, "The Vue Instance" <https://vuejs.org/v2/guide/instance.html>
Hämtad 2019-01-10
- [29] MDN web docs, "null"
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null Publicerad 2018-01-10. Hämtad 2019-01-10
- [30] Vue.js, "Options / Data" <https://vuejs.org/v2/api/#Options-Data> Hämtad
2019-01-10
- [31] Vue.js, "List Rendering" <https://vuejs.org/v2/guide/list.html> Hämtad
2019-01-10
- [32] Vue.js, "Methods and Event Handling"
<https://v1.vuejs.org/guide/events.html> Hämtad 2019-01-10
- [33] npm, "cors" <https://www.npmjs.com/package/cors> Hämtad 2019-01-10
- [34] npm, "method-override" <https://www.npmjs.com/package/method-override>
<https://www.npmjs.com/package/method-override> Hämtad 2019-01-10
- [35] Vue, "Introduction" <https://vuejs.org/v2/guide/> Hämtad 2019-01-20
- [36] agriya, "Pros and Cons of Vue.js framework",
<https://www.agriya.com/blog/pros-and-cons-of-vue-js-framework/> Pu-
blicerad 2017-05-15. Hämtad 2019-01-20