

WorldDB CMS



Lavet af
Simon Kalmar Riis Mortensen, Christian Kjer Bang og Jonatan Shoshan

Sider: 7,17
Af 17.216 tegn med mellemrum

REPO
<https://github.com/Jonatan-bs/projectmar2020>

Indholdsfortegnelse

Indledning	3
Problemformulering	3
Metodeovervejelser	4
Tidsplan	4
Sketching	4
Git	4
Farveteori	4
Research	4
Analyse	5
Sketching	5
Farver	5
Skrifttyper	6
Færdigt design og proces	6
Konstruktion	9
BACKEND	9
Fundament	9
Models	10
Routers	11
Controllers	11
Delkonklussion	12
FRONTEND	12
Opsætning	12
Event delegering	13
AJAX	13
LazyLoad	14
Evaluering af proces og produkt	15
Konklusion	16

Indledning

Vi vil i denne rapport gennemgå processen af at lave et enkelt CMS, til at håndtere CRUD opgaver i en given database. Der vil også blive sat et kort fokus på selve designprocessen af projektet, men det hovedsagelige fokus vil blive på dette kodnings arbejde.

Vi vil forklare backend delen, med fokus på at lave et API, som en klient kan kommunikere med. Herefter vil vi vise hvordan dette bruges i sammenhæng med frontend.

Produktet kan hentes her:

<https://github.com/Jonatan-bs/projectmar2020>

Ved første start, kør "npm run reStart" i terminalen, for at sætte world databasen op og starte applikationen.

! npm reStart overskriver eventuel eksisterende world database !

Derefter kan applikationen startes ved at køre kommandoen "npm start".

Problemformulering

Hvordan laver vi et CMS der kan udføre CRUD funktioner på de forskellige collections i en givet database, ved brug af mongoose, express og node?

Hvordan sikrer vi at holde os indenfor samme datastruktur, som den eksisterende data i databasen?

Hvordan sørger vi for, at alt manipulation af data, sker ved klienten istedet for på serveren?

Metodeovervejelser

Tidsplan

Vi arbejdede i dette projekt ud fra en tidsplan:

Fredag		Lørdag		Søndag
Problemformulering, tidsplan og begyndelse af kodning		Fortsættelse af kodning		Fortsættelse af kodning
Mandag	Tirsdag	Onsdag	Torsdag	Fredag
Rapport, design	Rapport, design	Rapport, design	Opsamling på alt	Aflevering

Dette var en meget løs tidsplan, som gave os fokuspunkter fra dag til dag, så det var muligt at kunne komme til at få de ting man ville have løst færdig på de dage man ønskede. Disse punkter var også lavet, så det var at der kunne være dage til at lede efter de store fejl man mødte.

Sketching

En god ting at gøre under designprocessen er sketching for at have en nogenlunde idé omkring hvad det er man skal have gjort i løbet af en projekt. Dette var det vi tænkte ville være bedst under arbejdet.

Git

Git blev brugt som et godt middel for gruppearbejde til selve projektet, da det var nemt at komme til at dele koden med hinanden. Dette var perfekt, når det var at alle var isoleret pga COVID-19.

Farveteori

Farveteori er vigtigt i en designprocess og hvis farver ikke er rigtige, så kan ens hjemmeside ikke rigtigt komme til at ende med et godt udseende.

Research

Research i dette projekt var meget limiteret, da der ikke var meget information der skulle findes på internettet. Vi havde allerede fået givet den information, som skulle være gjort brug af i selve projektet, så det gav ikke rigtigt mening at finde information der passede mere til.

Hvis dette projekt skulle have været til en kunde eller være i kontakt med mange andre, så kunne der have været research i forhold til designmæssige opgaver for at forbedre brugeroplevelserne til selve opgaven. Dog, da dette ikke var hovedpunktet for opgaven, så blev dette valgt ikke at blive givet meget fokus og derfor blev der ikke lavet meget research i løbet af opgavens forløb.

Analyse

Hele opgaven gik ikke op i noget designmæssigt, men vi syntes stadigvæk at der skulle gøres et eller andet ved hjemmesidens udseende. Så der blev gjort nogle forskellige ting her og der i løbet af selve opbygningen af hjemmesiden.

Sketching var en vigtig del af selve processen og det at have en minimal prototype for selve siden, så efter at alt kode arbejdet var blevet færdigt, så var det tid til at designe hjemmesiden ud fra sketches, som havde været lavet.

Sketching

Starten af designprocessen gik ud på at tegne forskellige idéer til hvordan selve siden kunne komme til at se ud. Selvom designet kom til at blive ændret ud fra hvad vi havde originalt designet på papir, så var dette en meget vigtig del af processen for at have en hovedsagelig idé.

INDSÆT BILLEDE

Som det kan ses på sketchen, så var idéen at have en sidemenu på hjemmesiden, så man nemt kunne komme til at vælge den collection man ville arbejde med.

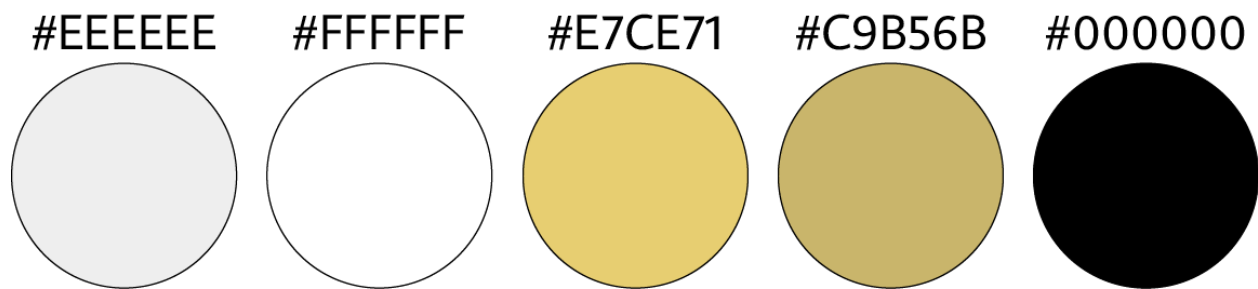
Det vil kunne ses i senere afsnit at vi gik væk fra denne idé, men nogle aspekter blev vedligeholdt i løbet af denne proces.

Farver

I forhold til farver, så var det svært at finde noget, som virkeligt passede ind når det blev designet. De første tanker omkring en primær farve ville være blå, da den giver et meget professionelt udtryk med dens kolde tilstedeværelse, dog virkede den blå farve til at gøre siden kedelig ved dens tilstedeværelse sammen med grå, hvide og sorte nuancer.

Det var derfor besluttet for at give siden noget mere liv i sig at der blev valgt en farve, som hovedsageligt ville kunne bringe en smule glæde til sig. En farve som gul er ofte kategoriseret som en farve der kan gøre folk glade ved bare at kigge på den. Da det også var en varm farve, så gav dette siden en meget mere behagelig følelse for brugeren af hjemmesiden.

Farver vi valgte at bruge på siden var derfor:



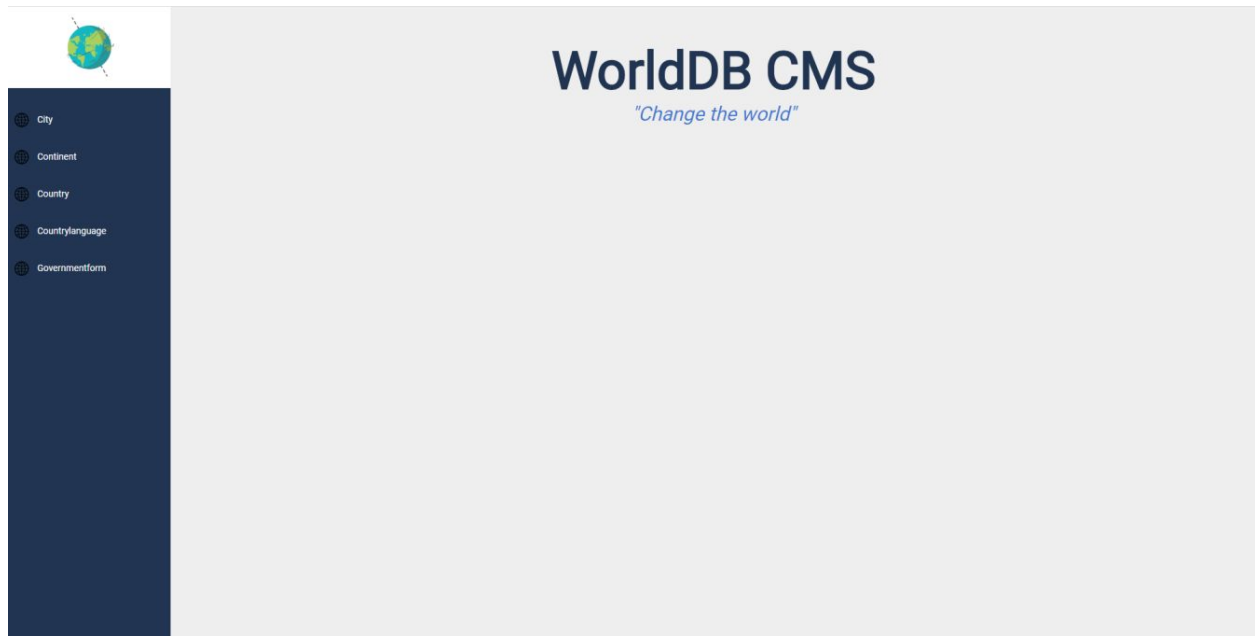
Skrifttyper

I forhold til skrifttyper, så var der to hovedsagelige valg, som ofte blev kigget på i løbet af designarbejdet. Dette var Lucida Sans og Roboto fra Google Fonts. I rapporten her, så har man mulighed for at se Roboto skrifttypen i brug. Det er en skrifttype, som er nemt at læse og det at den også er sans serif gør den meget pæn, hvilket er et plus, når den også er nem at læse.

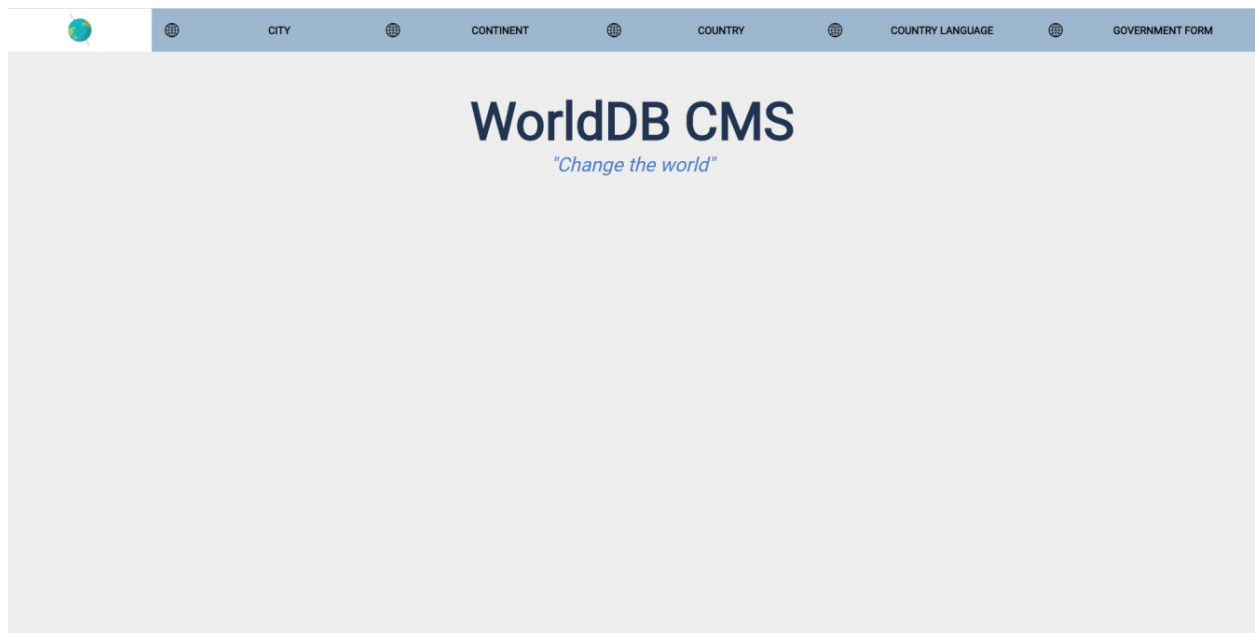
Det var vigtigt at vælge en skrifttype, som var nem at læse på en hjemmeside med en masse information man skal igennem. Dette var grunden til at Lucida Sans blev valgt fra, selvom det var en meget flot skrifttype. Roboto kom op på toppen af piedestalen og vandt denne skrifttype kamp.

Færdigt design og proces

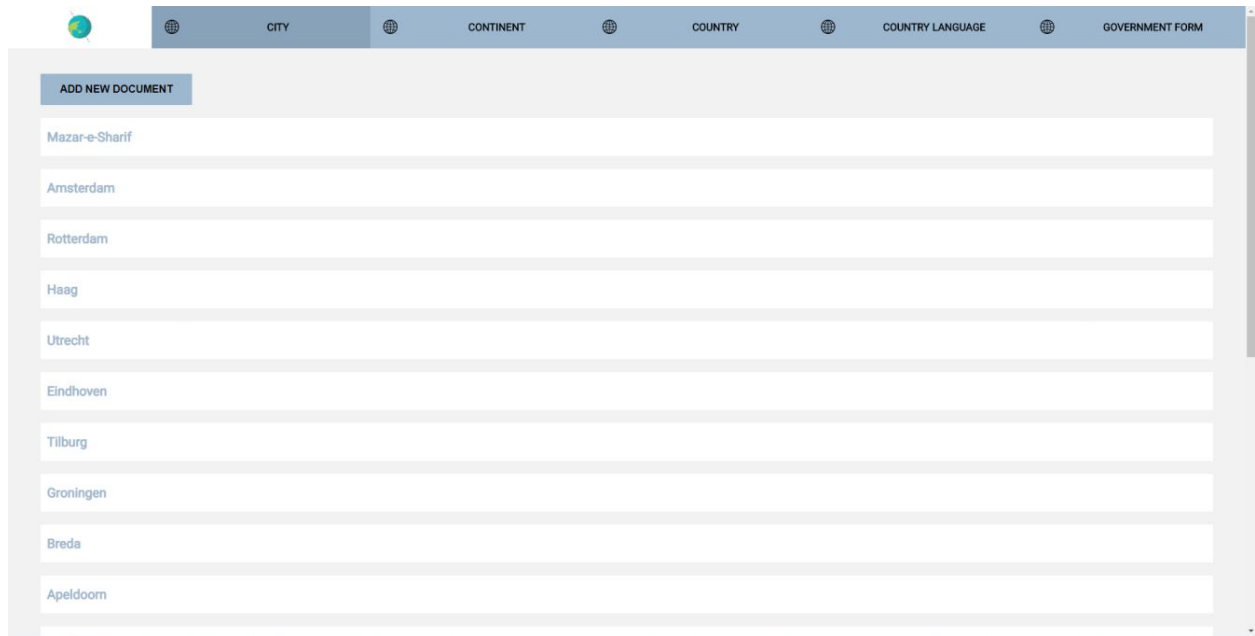
I forhold til selve designet, så var der en idé i starten om at kunne have en menubar i siden, hvilket kan ses på følgende billede:



Det var en fin idé at prøve noget der lod toppen være åbent, men det føltes som om nogle ting manglede eller at det kunne bruge nogle ændringer. Dette startede så ud med at gøre den blå farve lysere og vælge en lidt mere traditionel menubar.

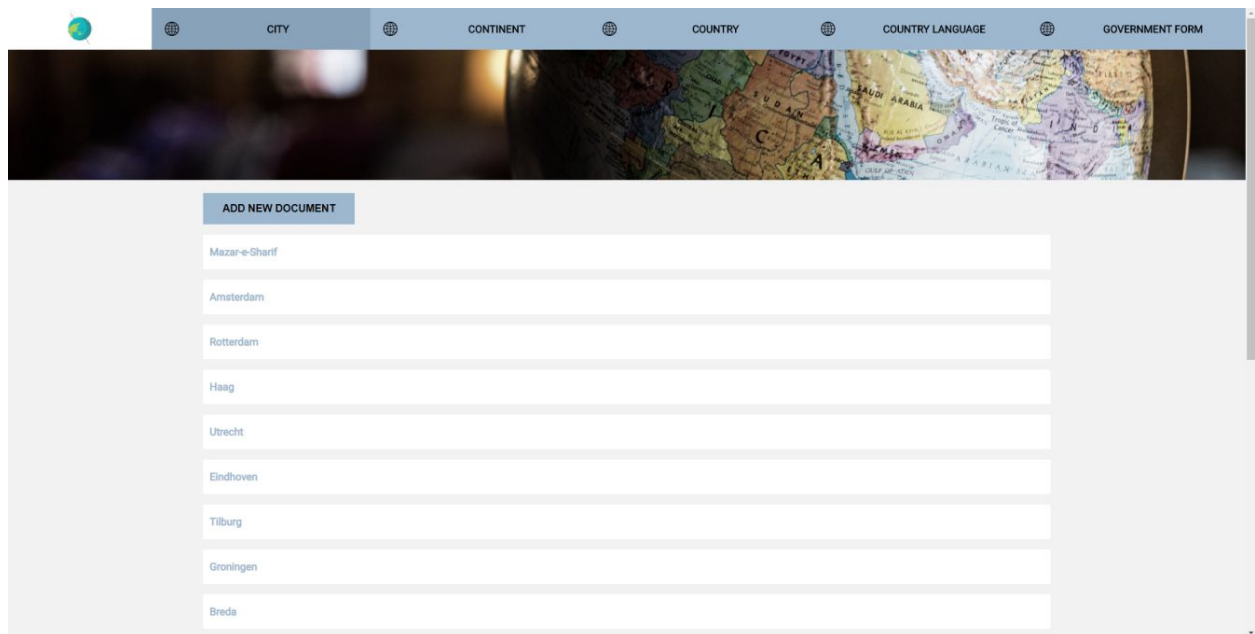


Det føltes lidt bedre i det store billede af alting, så der blev leget lidt med farve, information og designet, men selve siden virkede lidt tom ved ikke rigtigt at gøre brug af nogle billeder. På følgende billede kan dette ses:



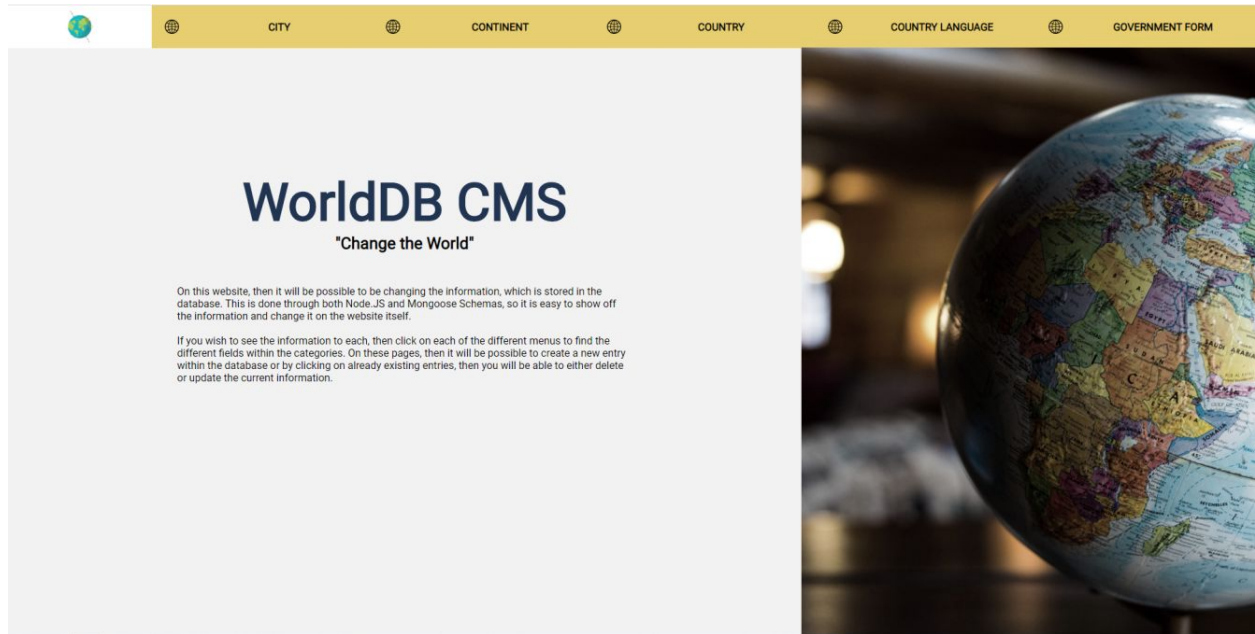
The screenshot shows a web application interface. At the top, there is a navigation bar with a logo on the left and five menu items: CITY, CONTINENT, COUNTRY, COUNTRY LANGUAGE, and GOVERNMENT FORM. Below the navigation bar, there is a section titled "ADD NEW DOCUMENT". Under this section, there is a list of cities: Mazar-e-Sharif, Amsterdam, Rotterdam, Haag, Utrecht, Eindhoven, Tilburg, Groningen, Breda, and Apeldoorn. Each city name is in a separate input field.

Det føltes tomt, så idéen var derfor at tilføje et billede til hjemmesiden, så det var at det ikke føltes så tomt med det meget lange data bokse.



This screenshot shows the same web application interface as the first one, but with a large, colorful map of the Middle East region added to the header area, below the navigation bar. The map shows countries like Sudan, Saudi Arabia, Iraq, and Iran. The "ADD NEW DOCUMENT" section and the list of cities remain the same as in the first screenshot.

Dette hjalp en smule på denne tanke omkring at der manglede noget, men billede følte som om det ikke rigtigt passede ind i den position og den blå farve var ikke helt rigtig, når man tænkte på hvad det skulle have gjort for siden. Dette var i dette øjeblik, hvor vi valgte at ændre den primære farve til en gul farve og rykke billede positionen fra hvor den i øjeblikket var placeret.



Dette gjorde siden meget mere livlig og det føltes ikke så tungt igen. Med den gule farve og tilstedeværelsen af et billede, så virkede siden til at være meget mere æstetisk og dejlig at være på. Selve processen var lang, men gennem trial and error, så fandt vi frem til hvad vi ønskede os af en hjemmeside.

Konstruktion

BACKEND

Fundament

For at sætte fundamentet op for vores express app, bruger vi express-generator, som generer grund-filer, mappestruktur og kode arkitektur.

For at installere modulerne i den genererede "package.json" fil, bruger vi herefter "npm install". Efter vi har fjernet unødvendige filer og kodelinjer, og installeret 'mongoose', opretter vi en mappe til controllers og models.

Vi har nu en MVC arkitektur at bygge vores applikation ud fra.

Vi opretter derudover to forskellige scripts i "package.json", så vi har følgende:

Start:

starter applikationen

reStart:

Resetter databasen og starter applikationen

devStart:

starter applikationen med nodemon, som sørger for at genstarte serveren, hver gang der lavet ændringer

Før vi går videre til næste felt, sætter vi vores connection op til databasen i vores app fil, ved hjælp af mongoose.

```
//connect to database
mongoose
.connect("mongodb://localhost:27017/world", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useCreateIndex: true
})
.then(res => console.log("connected to database"))
.catch(error => handleError("error connecting to database"));
```

Models

For at holde en fast struktur i vores collections, laver vi ved hjælp af mongoose, skemaer ud fra hver collections data.

Det gøres helt simpelt ved at lave et objekt der indeholder de samme keys, som er i den tilhørende collection og skrive hvilken datatype der bruges.

Vi undlader mongooses standard versionKey og sætter et collection navn, for at undgå mongooses automatiske omskrivning til flertal.

```
const mongoose = require("mongoose");

const userSchema = mongoose.Schema(
{
  oldid: { type: Number, required: true},
  name: { type: String, required: true },
  countrycode: { type: String, required: true },
  district: { type: String, required: true },
  population: { type: Number, required: true }
},
{ collection: "city", versionKey: false }
);

module.exports = mongoose.model("city", userSchema);
```

Disse modeller giver os nogle forskellige muligheder, for at interagere med databasen, som vi gør brug af i controllers afsnittet.

Routers

For at opretholde læseligheden i vores kode, har vi én router fil, der vider sender de forskellige requests til deres respektive controller. (se næste afsnit)

```
/* get Schema*/
router.post("/api/schema/:collection", apiCtrl.getSchema);

/* Retrieve documents from collection */
router.post("/api/:collection", apiCtrl.retrieve);

/* Create document to collection */
router.post("/api/:collection/create", apiCtrl.create);

/* Delete document from collection */
router.post("/api/:collection/:id/delete", apiCtrl.delete);

/* Update document from collection */
router.post("/api/:collection/:id/update", apiCtrl.update);
```

Controllers

I vores controllers, har vi oprettet funktioner til alle CRUD opgaver, samt en til at hente vores mongoose skemaer.

Create

For at kunne lave vores funktioner så dynamiske som muligt, har vi gjort brug af "mongoose.models", som er et objekt der indeholder vores modeller.

Vi finder så ud af hvilken collection vi skal lede efter, gennem vores URL parameter og finder derfra den tilhørende model.

(Det havde her nok givet mere mening, at sende vores collection navn i vores req.body objekt)

```
const collection = req.params.collection;

model = mongoose.models[collection];
```

Ud fra vores model skaber vi en ny instance, som indeholder vores req.body parametre.

```
const newDocument = new model(req.body);
```

Denne nye instance af vores model constructor indeholder nogle forskellige metoder, til at udføre CRUD handlinger og vi kan i dette tilfælde gøre brug af .save, som gemmer vores nye dokument i databasen, hvis dataen matcher vores skema

```
newDocument.save()
```

Delete, update

For at slette eller opdatere et dokument, kan vi bruge den samme tilgang, bare med metoden `deleteOne()` og `updateOne`, istedet for `save()`.

Retrieve

For at hente dokumenter, har vi valgt at give klienten muligheden for at sende nogle json parameter med, for at vælge hvilke felter der skal hentes, hvor mange, sortering og lignende.

```
model.find(query, fields, options)
```

Grunden til dette var, at vi både kan hente et enkelt dokument ud fra et id og alle dokumenter i en collection, ved brug af samme metode.

Dertil skulle vi bruge parameterne `skip` og `limit`, til at lave `lazyLoad` af data senere hen i processen.

getSchema

Denne metode finder en model tilhørende en valgt collection.

Denne model har en `schema.paths` property, der inderholder vores tidligere definerede schema. Da dette schema indeholder navne og datatyper, kan vi bruge det til at generere vores input felter frontend.

```
getSchema(req, res, next) {  
  const collection = req.params.collection;  
  const schema = mongoose.models[collection].schema.paths;  
  res.send(schema);  
}
```

Delkonklusion

Vi har nu et fuld funktionel API til at udføre de nødvendige CRUD funktioner på vores world database.

FRONTEND

Opsætning

Vi har sat vores CMS op, som en `singlePage` hjemmeside, hvilket vil sige at vi kun har en html fil, som ved hjælp af Ajax kald, opdaterer indholdet uden refresh.

Hvores `index.html` fil, indhenter vores style og et JS modul der håndterer opdateringer og events.

Derudover er der en menu, med vores forskellige collections, samt en container div, til at modtage vores dom elementer, som bliver genereret ud fra vores Ajax respons.

Event delegering

Vi har gjort brug af event delegering til at håndtere vores click events. Dette er grundet bedre læselighed og muligheden for at have events på elementer der bliver kreeret dynamisk efter DOM load.

Dette fungerer helt simpelt ved at lytte på alle click og derefter tjekke om det klikkede element matcher vores kriterier.

(Selvom det ikke umiddelbart virker sådan, skulle dette være mindre krævende)

```
document.addEventListener("click", e => {  
  // UPDATE DOCUMENT  
  if (e.target.matches(".update")) {  
    api.update(e);  
  }  
})
```

AJAX

For at snakke sammen med vores API, bruger vi AJAX og mere præcist har vi valgt at bruge fetch.

Vi venter på respons og bygger derfter vores DOM op udfra svaret.

```
fetch("http://localhost:3000/api/" + collection, {  
  method: "post",  
  headers: {  
    "Content-Type": "application/json"  
  },  
  body: JSON.stringify(body)  
})  
  .then(response => response.json())  
  .then(doc => {  
    createDomElms.createEditDocDom(doc[0], collection);  
  })
```

Vi går ikke igennem flere eksempler på dette, dat det er den samme process, for de forskellige kald, med med forskellige funktioner til at håndtere svaret.

LazyLoad

Da en af vores collections i databasen havde over 4000 resultater, har vi valgt at gøre brug af konceptet lazyLoad. Dette fungerer ved at vi kun loader 20 resultater ad gangen og derefter henter flere, hvis brugeren har scrollet ned til det sidst hentede resultat.

For at lave denne funktion har vi gjort brug JS hoisting.

```
let loadmore = loadMoreConst(collection);  
loadmore();
```

Her returnerer loadMoreConst() en function, som stadig har nogle variabler i sit scope, fra dens constructor funktion.

(se loadMoreConst() på næste side)

Vi sætter en scroll event på vores document objekt, som tjekker om vi er i bunden af dokumentet og i så fald, laver et nyt AJAX kald til at hente dokumenter. Denne gang med parametern skip, som skipper de første 20 dokumenter vi allerede har hentet. Herefter lægger vi 20 til vores skip variabel og venter på brugeren igen scroller til bunden af dokumentet, for at se de næste 20 resultater.

```
// Load Extra documents  
function loadMoreConst(collection) {  
  let skip = 0;  
  let extras = false;  
  let loadTwenty = () => {  
    document.removeEventListener("scroll", lazyload);  
  
    lazyload = () => {  
      if (bottomOfDiv()) {  
        loadTwenty();  
      }  
    };  
  };  
  
  let body = {  
    query: {},  
    fields: "",  
    options: { limit: 20, skip: skip }  
  };  
  
  fetch("http://localhost:3000/api/" + collection, {  
    method: "post",
```

```

headers: {
  "Content-Type": "application/json"
},
body: JSON.stringify(body)
})
.then(response => response.json())
.then(docs => {
  createDomElms.createDocumentsDom(docs[0], collection, extras);
  extras = true;
  if (docs[0].length === 20) {
    document.addEventListener("scroll", lazyload);

    skip += 20;
    if (bottomOfDiv()) {
      loadTwenty();
    }
  }
})
.catch(err => {
  console.log(err);
});
};
return loadTwenty;
}

```

Evaluering af proces og produkt

Det kan nok roligt blive sagt at situationen med COVID-19 (Coronavirus) i Danmark gjorde at projektarbejde ikke kunne gøres helt på samme måde som i tidligere situationer. Dette gjorde at noget arbejde blev gjort uden rigtigt at kunne komme til at snakke ordentligt sammen med gruppen. Dette gjorde også at det kunne være svært at finde ud af helt hvor meget folk arbejdede i løbet af projektet, men selvom dette skete, så lod projektet til at ende ud med et færdigt produkt.

Det var svært at arbejde uden så meget kommunikation, men det lod til at selve projektet endte på en ordentlig nok måde med et produkt, som kan kaldes godt i øjnene på alle medlemmerne i gruppen. I forhold til tidsplanen så lod det til at vi fulgte den ret godt, da kodnings arbejdet var færdigt allerede søndag og designet var færdigt om onsdagen. Det sidste der manglede var rapporten og som det nok kan ses, så er den blevet færdig.

Selve hjemmesiden ser godt ud og alle funktionerne, som vi havde planlagt var på hjemmesiden, hvor der havde været tanker om måske at kunne have lavet en søgefunktion for det land, sprog, by og andet man havde lyst til at finde. Dog valgte vi ikke at have det med til sidst.

Så, alt i alt vi kunne have haft mulighed for at tale mere sammen, hvis vi havde været sammen personligt, hvilket var svært med den nuværende situation. På den måde ville det være nemmere at være fuldkommen enig omkring alting der foregik i projektet, dog var vi meget tilfredse med selve det færdige produkt.

Konklusion

Vi kan konkludere, at MongoDB, express og node, skaber et fundament, til hurtigt og nemt at opsætte et rest API.

Dette gøres endda endnu nemmere ved brug af mongoose.

For at holde en statisk data struktur i databasen, giver mongoose også muligheden for at bruge schemaer og derved opsætte nogle kriterier for hvilken data der kan komme ind i databasen.

Ved kun at sende den rene data fra databasen, kan vi bruge API'et i sammenspil med en hjemmeside, som kan håndtere dataen og opdatere DOMMEN ud fra dette. Dette giver eksempelvis muligheden for at lave en singlePage hjemmeside.

Ved designet, så lavede vi et design, som kunne være nemt at bruge for en bruger, da menuvalgene er nemme og ved brug af farver, så er det nemt for brugeren at finde ud af hvad der kan gøres noget ved på hjemmesiden.