

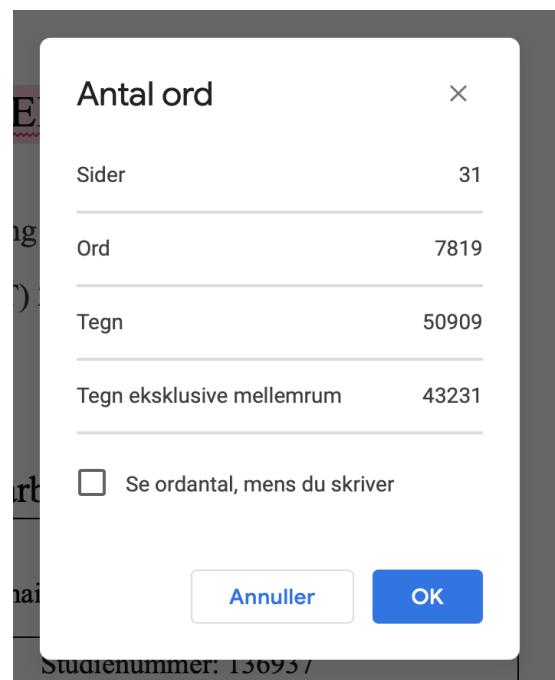
SOTELL 2021

Programmering og udvikling af små systemer samt databaser

HA(IT) 2. semester

Udarbejdet af

Mikkel R. Pedersen	Studienummer: 408449 Email: mipe20ag@student.cbs.dk
Metin Mogensen	Studienummer: 136937 Email: memo19ad@student.cbs.dk
Caroline Langholz	Studienummer: Email: cala20af@student.cbs.dk



KRAVSPECIFIKATIONER

Ja	1. App'en skal tillade en bruger at oprette en profil
Ja	2. App'en skal tillade en bruger at slette sin egen profil
Ja	3. App'en skal tillade en bruger at opdatere sin egen profil
Ja	4. App'en skal tillade brugeren at logge ind
Ja	5. App'en skal tillade at hvis en bruger er logget ind kan de forblive logget ind.
Nej	6. App'en skal gøre det muligt for en bruger at vælge like eller dislike for en foreslægt profil
Nej	7. App'en skal give brugeren en notifikation, såfremt begge profiler har liket hinanden
Ja	8. App'en skal gøre det muligt for en bruger at logge ud
Ja	9. App'en skal kunne vise en liste over en aktuels brugers matches
Ja	10. App'en skal kunne vise en fuld profil for et potentiel match
Ja	11. App'en skal give brugeren mulighed for at fjerne et match igen
Ja	12. App'en skal give admin mulighed for at opdatere en brugerprofil
Ja	13. App'en skal give admin mulighed for at slette en bruger
Nej	14. En admin skal have mulighed for at se brugsstatistikker af appen - Hvor mange brugere systemet har samt hvor mange matches systemet har
Ja	15. App'en skal indeholde en matching algoritme, som præsenterer brugeren for eventuelt interessante profiler. Hvordan I laver denne algoritme er op til jer. I kan matche folk efter by, højde, interesser såsom hvor mange katte de har. Med andre ord - I bestemmer selv. Vi giver point for implementering og performance af algoritmen (begrundet med O-notation). Argumenter også gerne for hvorfor det er en god måde at matche folk på.
Nej	16. En bruger skal kunne sætte et køn samt aldersramme han/hun leder efter. Dertil skal en bruger kun blive vist profiler som opfylder dette filter.

INDHOLDSFORTEGNELSE

Programmering og udvikling af små systemer samt databaser	0
HA(IT) 2. semester	0
KRAVSPECIFIKATIONER	1
INDHOLDSFORTEGNELSE	2
INTRODUKTION	3
Udviklingsværktøjer:	3
Programudvikling:	4
Moduler	4
Dependencies	4
KRAVSPECIFIKATION OG ARKITEKTUR	5
Kravspecifikationer	5
Klassediagram	15
Applikationens klasser	15
STORAGE	16
SQL	17
Dataopbygningsform	17
SQL statements	18
DDL & DML	18
BACKEND	19
Algoritme	19
Serverless computing	21
Tedious	21
FRONTEND	21
Vue.js	22
ROUTER	22
Axios	23
Alternativt til Axios	24
HTTP-requests	24
Promises	25
TESTING	26
Postman	27
Unit Testing	27
PROCESEVALUERING	27
LocalStorage og tokens	28
Frontend	28
KONKLUSION	29
LITTERATURLISTE	29

INTRODUKTION

Vi har til dette års sommereksamen på HA(IT) 2. semester i programmering af systemer og små databaser udarbejdet dating-applikationen SOTELL.

Opgavens grundlæggende udgangspunkt har været fra et programmeringsmæssigt perspektiv. Dybdegående forretningsmæssige overvejelser har dog samtidig spillet en relevant rolle for udviklingsprocessen, og de valg der er taget er derfor også inkorporeret i vores gennemgang af løsningsovervejelser. Vi havde ikke til sinde udelukkende at udvikle en datingapplikation, men gerne også et koncept der bar på mere end dette.

SOTELL er en applikation, der kan matche beboere på hoteller med hinanden. Idebasen for appen er hovedsageligt dating, men appen skulle potentielt kunne skaleres ud til at blive en social platform for den ensomme rejsende, der mangler en at spille et spil kort eller snuppe en drink i baren med. På SOTELL kan man på stående fod udelukkende matche ud fra alder samt køn. Man kan dog specificere hvilket hotel man befinner sig på, og en videreudvikling af applikationen skulle gerne medføre en filtreringsmekanisme, der gør netop hotelmatching muligt.

Applikationen lever op til samtlige 12/16 krav, som eksaminator for dette års eksamen i “udvikling af små systemer samt databaser” på HA (IT) har forelagt. Igennem rapporten kortlægges der for læser, hvilke løsninger vi har implementeret for at imødekomme kravspecifikationerne til produktet. Udarbejdelsen af SOTELL har været en læringsprocess for alle i gruppen og visse løsninger afspejler netop dette. Derfor er der ydermere inkluderet alternative løsningsovervejelser, til de krav, der var for tidskrævende til at udarbejde indenfor de givne rammer. Vi har lagt mange timer i projektet, og applikationen er løbende blevet testet. Samtidig er der blevet lagt stort fokus på at overholde en god kodestik, en overskuelig struktur og tilføjet kommentarer til de dele, hvor der menes at det var nødvendigt for at give læser og eventuelt investor en bedre grundforståelse for projektets helhed. Alt dette både mener og håber vi at opgavebesvarelsen afspejler.

Udviklingsværktøjer:

Applikationen er udviklet i Node.JS, hvor vi så vidt som muligt har prøvet at følge MVC-frameworket. MVC er oprindeligt en arkitektonisk tilgang, der opdeler en applikation i 3 logiske komponenter: Model, View og Controllers. Da vi har skulle inkorporere brugen af Azure funktioner i denne eksamsopgave, har den helt klassiske MVC struktur ikke kunnet lade sig gøre. Azure funktioner medfører, at vi ikke har kunnet separere modellerne i deres egen mappe. Dette skyldes, at hver Azure funktion indeholder 3 komponenter, der skal ligge isoleret i en “funktionsmappe”.

“Visual Studio Code” er blevet brugt som kildekode-editor, hvorfra det er muligt at tilgå forskellige programmeringsredskaber. Applikationen er opdelt i “three-tier-architecture” formen, for at give et bedre overblik, samt et overskueligt workflow. Three-tier-architecture gør, at der kan opdateres, samt arbejdes isoleret på enkelte dele, uden at der forårsages generelle komplikationer. Frontenden er udarbejdet i Vue.js, da dette er et simpelt, dog ekstremt brugbart redskab til applikationer som disse. Vue.js gav os ydermere muligheden for at inkorporere en mere objektorienteret tilgang til udarbejdelsen af applikationen.

For at bibeholde et godt overblik over databasen, er programmet Datagrip blevet brugt. Datagrip gør arbejdet med databasen simpelt og overskueligt.

Da denne opgave er udarbejdet i en gruppe på 3, har vi brugt GitHub til løbende at kunne dele processen med hinanden, samt VSC udvidelsen “Live share” til at kode i realtid sammen.

Programudvikling:

Moduler

I applikationen har vi brugt forskellige moduler. Moduler er selvstændige funktionelle enheder, der kan bruges på tværs af forskellige projekter i Node.js. Moduler giver programmører en større frihed, da vi kan tilføje funktionaliteter til vores programmer, der ville være langt mere komplicerede ellers at tilføje. Alternativet til at inkludere moduler ville være selv at skulle skrive kodebasen, der ligger til grunds for udførelsen af denne handling. Brugen af moduler gør ydermere en fremtidig skalering af projektet simplere, da det giver koden en simplicitet, der gør det nemmere at overtage som nyankommen programmør.

Dependencies

Dependencies bruger vi til at køre vores applikation. I denne opgave har vi brugt følgende:



```
{  
  "name": "progeksamens2021",  
  "version": "1.0.0",  
  "description": "",  
  "scripts": {  
    "start": "webpack-dev-server --open",  
    "build": "webpack --mode production"  
  },  
  "dependencies": {  
    "axios": "^0.21.1",  
    "core-js": "3.11.1",  
    "cors": "^2.8.5",  
    "express": "4.17.1",  
    "jsonwebtoken": "8.5.1",  
    "parse": "3.2.0",  
    "production": "0.0.2",  
    "tedious": "11.0.7",  
    "vee-validate": "2.2.8",  
    "vue": "2.6.10",  
    "vue-router": "3.1.3",  
    "vue-server-renderer": "2.6.12",  
    "vuex": "3.1.2"  
  },  
  "devDependencies": {  
    "@babel/cli": "7.13.16",  
    "@babel/preset-env": "7.14.0",  
    "@vue/cli-plugin-router": "4.5.12",  
    "babel-core": "6.26.0",  
    "babel-loader": "7.1.5",  
    "babel-preset-env": "1.6.1",  
    "babel-preset-stage-0": "6.24.1",  
    "babel-preset-vue": "2.0.2",  
    "css-loader": "3.3.2",  
    "html-webpack-plugin": "3.2.0",  
    "path": "0.12.7",  
    "vue-loader": "14.2.3",  
    "vue-template-compiler": "2.6.10",  
    "webpack": "4.46.0",  
    "webpack-cli": "3.3.10",  
    "webpack-dev-server": "3.11.2"  
  }  
}
```

KRAVSPECIFIKATION OG ARKITEKTUR

Læser vil i dette afsnit få en detaljeret gennemgang af udførelsen af kravspecifikationerne stillet til denne opgaven. Da visse kravspecifikationer har tendens til at ligne hinanden i kodestrukturen, gennemgåes denne tekniske tendens i det første krav, hvor dette bliver relevant. Herefter refereres der til denne gennemgang. Vi er af den opfattelse, at det på denne måde bliver holdt til et niveau, hvor læser bliver præsenteret for det der gør det givne krav unikt fra de andre, og at gennemgangen dermed ikke bliver for ensartet, og vigtige detaljer ikke bliver udeladt grundet tekniske gengangere.

Kravspecifikationer

Krav (1) - App'en skal tillade en bruger at oprette en profil

E-mail	Password	Fornavn
Efternavn	Alder	Køn
Hjem foretrækker du	Hotel	Register
Allerede en bruger? Login		

En bruger tilgår vores forside, for at oprette en profil. Da vi arbejder efter 3-tier-architecture modellen, er brugerens første interaktion med programmet gennem vores Vue.js komponenter. Brugeren tilgår endpointet: [./http://localhost:8080/#/](http://localhost:8080/#/). Herfra bliver vedkommende bedt om at udfylde en registreringsform med de givne input, der passer i vores datamodel.

Brugeren indtaster sine oplysninger, og ved hjælp af axios fetches dataen og sendes som et HTTP request til den tilhørende Azure funktions endpoint.. I Azure functions endpointet startes databasen vba. en asynkron funktion. På denne måde, kan vi sikre, at postmetoden udelukkende køres, hvis databasen er connected.

Herefter bliver requestets body - dataen den respektive bruger har indtastet (se bilag 1) - lavet til objektet payload.

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    try{
        await db.startDb(); //start db forbindelse
    } catch (error) {
        console.log("Error connecting to the database", error.message);
    }
    switch(req.method) {
        case 'GET':
            | await get(context, req);
            break;
        case 'POST':
            await post(context, req);
            break;
        default:
            context.res = {
                status: 200,
                body: "please get or post"
            };
            break;
    }
}

async function post(context, req){
    try{
        let payload = req.body;
        await db.insert(payload);
        context.res = {
            status: 200,
        }
    } catch(error){
        context.res = {
            status: 400,
            body: error.message
        }
    }
}
```

Den asynkrone funktion kalder efterfølgende på “insert” funktionen i Db.js, hvor payload bliver brugt som parameter i funktionen insert. I Db.js har vi defineret en SQL query som følger:

```
'INSERT INTO [user] (firstName, lastName, gender, email, password, age, hotel, preferredGender)
VALUES(@firstName, @lastName, @gender, @email, @password, @age, @hotel, @preferredGender)'
```

Ved brug af requestobjektet fra Tediouspakken sender vi et promise med vores query videre til vores database. Her kan dette promise enten blive rejectet eller resovlet.

Vi laver først et objekt med det SQL statement vi gerne vil have executed. Der defineres et “new Request”, hvor vi indsætter SQL statementet som et parameter i en callback funktion, der bliver kaldt, når requested er færdigkørt.

```
return new Promise((resolve, reject) => {
  const sql = 'INSERT INTO [user] (firstName,
  const request = new Request(sql,(err) => {
    if(err){
      reject(err)
      console.log(err)
    }
  });
});
```

Hvis det fejler, console logges fejlen som “err”, og det givne promise er dermed blevet rejected. Hvis ikke går funktionen videre til næste step, hvor den tilføjer et input parameter til vores request.

Ved tilføjelsen af inputparametrer specificeres navn, type og value. For alle inputparametrer gælder, at navnet skal være identisk med SQL statementdefinitionerne (se bilag 2). Disse definitioner skal videre være identiske med betegnelserne i databasen. Type specificerer, hvilken datatype inputtet skal sendes som til databasen. Sidst har vi det parameter, som payloaden skal indeholde. Herefter køres requestet færdigt. Efterfølgende køres execSql metoden, der executer SQL'en repræsenteret i requestet.

Hvis dette lykkedes, altså en bruger bliver oprettet i systemet, responderer den med en status 200, hvis ikke går den videre til vores catch og responderer med en status 400 samt fejlen. Sidst tager vores Register.vue så dette statusparameter ind og giver den respektive bruger en besked om, at vedkommende nu er blevet oprettet i vores database.

```
request.on("requestCompleted",(row) => { // 
  console.log("User inserted", row);
  resolve("user Inserted", row)
})
connection.execSql(request);

});
```

module.exports.insert = insert;

```
request.on("requestCompleted",(row) => { // 
  console.log("User inserted", row);
  resolve("user Inserted", row)
})
connection.execSql(request);
```

```
async function post(context, req){
  try{
    let payload = req.body;
    await db.insert(payload);
    context.res = {
      status: 200,
    }
  } catch(error){
    context.res = {
      status: 400,
      body: error.message
    }
  }
}

//redirect logic
if (response.status == 200) {
  alert("Du er blevet oprettet i vores database, klik OK!")
  this.$router.push({ path : '/login' });
}
```

Krav (2) - App'en skal tillade en bruger at slette sin egen profil

E-mail	Password	Delete
--------	----------	--------

Vil du alligevel ikke slette din profil? [Tilbage til forsiden](#)

Når en bruger er registreret og logget ind befinder vedkommende sig på “homepagen”. Her kan brugeren vælge “Slet min profil”, hvilket fører dem til endpointet: <http://localhost:8080/#/deleteUser>. Herefter skal email og adgangskode indtastes i formen, hvorefter vi vba. axios connecter til Azure funktionsendpointet: <http://localhost:7071/api/deleteUser>. Næst eksekverer vores Azure funktion, der kalder på deleteUser() funktionen i Db.js, hvor logikken for deletefunktionen er defineret. (Se github for yderligere dokumentation).

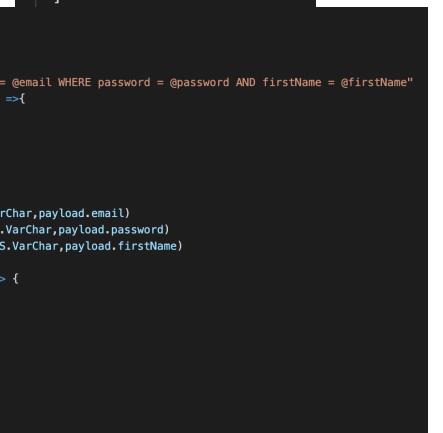
Siden registrerer, hvem der skal slettes ved at vedkommende manuelt indtaster deres oplysninger. En anden løsning kunne være, at siden tager fat i det payload vores JWT token indeholder, da den indeholder præcist det samme, som brugeren af applikationen på nuværende tidspunkt selv skal indtaste. Man kan argumentere for, at der er en vis sikkerhedsbrist i, at en bruger teknisk set kan slette andre brugere af systemet. Dog skal vedkommende kende til den bruger, der ønskes slettets email samt password, og derfor så vi det som en holdbar løsning for nu.



```

{
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "delete"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}

```

```

// Update User
function updateUser(payload){
  return new Promise((resolve,reject) => {
    const sql = "UPDATE [user] SET email = @email WHERE password = @password AND firstName = @firstName"
    const request = new Request(sql,(err) =>{
      if (err){
        reject(err)
        console.log(err)
      }
    });

    request.addParameter('email',TYPES.VarChar,payload.email)
    request.addParameter('password',TYPES.VarChar,payload.password)
    request.addParameter('firstName',TYPES.VarChar,payload.firstName)

    request.on('requestCompleted',(row) => {
      console.log("User updated", row);
      resolve("user Updated:");
    })
    connection.execSql(request);
  });
}

module.exports.updateUser = updateUser;

```

Krav (3) - App'en skal tillade en bruger at opdatere sin egen profil

Når en bruger er logget ind og befinner sig på “homepagen”, kan vedkommende vælge “Opdater min profil”. Denne funktion fører dem til endpointet: <http://localhost:8080/#/updateUser>. På dette endpoint kan man opdatere sine oplysninger. På nuværende tidspunkt har vi udelukkende lavet det således, at man kan opdatere sin email. Dette kan sagtens ændres, så man ved videreudvikling af applikationen kan ændre på resten af sine oplysninger. Frontenden og backenden taler sammen på samme vis som i kravspecifikation 1 vba. axios. Index.js filen til det givne endpoint er ligeledes tilnærmedsesvist ens. Dog bruges der,



til forskel for krav 1, en PUT metode, i stedet for POST. Hvilket også er specifiseret i funktionens function.JSON fil. PUT metoden er brugt her, da vi dermed udelukkende opdaterer et specifikt felt.

Krav (4) - App'en skal tillade brugeren at logge ind

```
async function post(context, req){
  try{
    let payload = req.body;
    let result = await db.login(payload);
    let token = await db.genToken(payload);
    console.log(token);

    context.res = {
      status: 200,
      body: {token}
    }
  } catch(error){
    context.res = {
      status: 400,
      body: error.message
    }
  }
}
```

```
// LOGIN db.js
function login (payload) {
  return new Promise((resolve, reject) => {
    const sql = 'SELECT * FROM [user] where email = @email AND password = @password';
    const request = new Request(sql,{err,rowCount}=>{
      if (err){
        reject(err)
        console.log(err)
      } else if( rowCount == 0){
        reject({message: "no user found"});
      }
    });

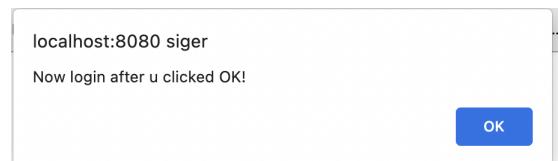
    request.addParameter('email', TYPES.VarChar, payload.email)
    request.addParameter('password', TYPES.VarChar, payload.password)

    request.on('row',(columns) => {
      resolve({message:"you are now logged in"});
    })
    connection.execSql(request)
  });
}
```

E-mail
Password
<input type="button" value="Login"/>
Ingen bruger? Register

Når en bruger har oprettet sig, kan vedkommende fortsætte til loginsiden. Her skal brugeren indtaste sin email og sit password i loginformen. Som tidligere nævnt bliver request bodyen sendt til endpointet: <http://localhost:8080/#/login>. Her fetches dataen fra http://localhost:7071/api/login via axios. Igen defineres objektet ”payload”, som herefter køres igennem loginfunktionen i db.js for at validere de indtastede oplysninger. Også i loginfunktion i db.js har vi defineret en SQL Query, og ved brug af requestobjektet fra Tedium pakken sender den Queryen videre til vores database som enten kan blive rejectet eller resolvet. Først ledes der efter kombinationen af inputtet ”Email” og ”password”, som brugeren har indtastet i vores form. Hvis funktionen finder en bruger med denne kombination, vil den returnere ét row, hvilket vil resolve vores besked ”you are now logged in”.

Hvis brugeren har tastet en kombination, som findes i databasen eksekveres funktionen genToken(), som



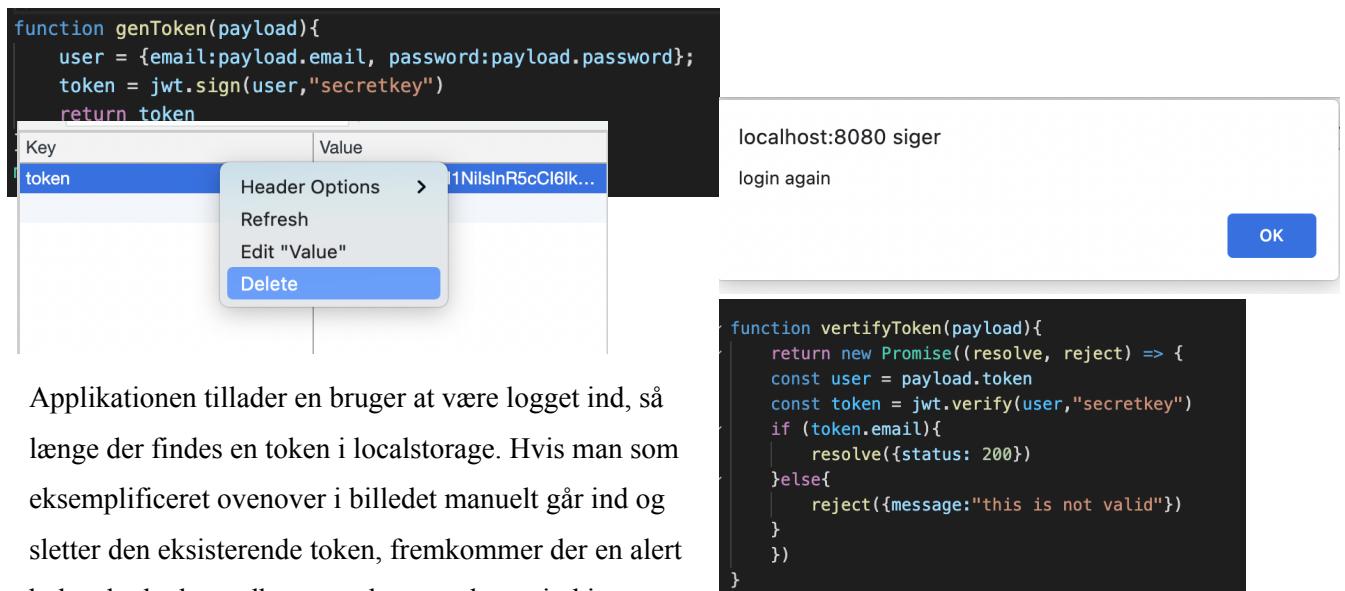
Key	Value
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJlbWFpbCj6InNvcmluQHJlZ2luZGsiLCJwYXNzd29yZC16IjEyMzEyMzIsImInIhdCI6MTYzM0c0NjM30.ufOYokTJSPAl9lPqJIMSp0D18sPVAPZRMlT1 1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJlbWFpbCj6InNvcmluQHJlZ2luZGsiLCJwYXNzd29yZC16IjEyMzEyMzIsImInIhdCI6MTYzM0c0NjM30.ufOYokTJSPAl9lPqJIMSp0D18sPVAPZRMlT1

konverterer vores payload fra req.body til en gyldig jwt token, der derefter gemmes i localstorage, for at brugeren kan forblive logget ind, og dermed

også har en token til fremtidige http requests.

Denne logik vi har gjort brug af, for at sikre det kun er brugere der er logget ind, som kan tilgå bestemte endpoints. Dette er en kombination af funktionen verifyToken(), Azure funktionen Iprotect og en metode i vue, som hedder created(). Azure funktionen sender den token, som er i localstorage til verifyToken(). verifyToken() dekryptere den til det oprindelige payload, og created() kontrollerer om "email" findes i den dekrypterede token.

Krav (5) - App'en skal tillade, at hvis en bruger er logget ind, kan de forblive logget ind.



```
function genToken(payload){  
  user = {email:payload.email, password:payload.password};  
  token = jwt.sign(user,"secretkey")  
  return token  
}
```

```
function verifyToken(payload){  
  return new Promise((resolve, reject) => {  
    const user = payload.token  
    const token = jwt.verify(user,"secretkey")  
    if (token.email){  
      resolve({status: 200})  
    }else{  
      reject({message:"this is not valid"})  
    }  
  })  
}
```

Applikationen tillader en bruger at være logget ind, så længe der findes en token i localstorage. Hvis man som eksemplificerer ovenover i billedet manuelt går ind og sletter den eksisterende token, fremkommer der en alert boks, der beder vedkommende om at logge ind igen.

Løsningen opfylder formålet, men rent teknisk mener vi den i nogen grad er manglende, da den ikke kan kende forskel på om det er email eller password der er forkert, men kun tager hensyn til om kombinationen af inputtet findes i databasen. Ved udarbejdelsen af en markeds klar applikation, kunne man argumentere for, at det ville være fordelagtigt, at brugeren blev oplyst om dette. Det ville højest sandsynligt forbedre brugeroplevelsen, hvilket altid er en markeds mæssig fordel.

Krav (6) - App'en skal gøre det muligt for en bruger at vælge like eller dislike for en foreslået profil

I applikationen skal brugere kunne like & dislike hinanden. Til dette bruger vi to Azure funktioner *like* & *dislike*. Funktionerne ligner register, og bruger POST metoden, da henholdsvis like og

```
async function post(context, req){  
  try{  
    let payload = req.body;  
    await db.insertLike(payload);  
  }  
}
```

dislike skal postes i databasen. Både like og dislike kalder funktionen insertLike(payload) & insertDislike(payload) i db.js. I db.js kører INSERT SQL statements for hver af dem. Værdierne ‘like’ & ‘dislike’ indsættes herefter i likeOrDislike.

```
`INSERT INTO [like] (likeOrDislike, userId, likedUserId)
VALUES ('like', @userId, @likedUserId)`
```

```
`INSERT INTO [like] (likeOrDislike, userId, likedUserId)
VALUES ('dislike', @userId, @likedUserId)`
```

Krav (7) - App'en skal give brugeren en notifikation, såfremt begge profiler har liket hinanden.

Ved tilfældet af at 2 brugere “liker” hinanden, modtager de en notifikation vdr. dette. Rent funktionelt kører der en funktion, der krydschecke de 2 kolonner i like-tabellen. Hvis likeOrDislike = ‘like’ og userId har likedUserId og likedUserId.userId har userId.likedUserId, sendes der en alert om, at der er fundet et match.

likeId	likeOrDislike	userId	likedUserId
1	15 like	59	46
2	16 like	46	59

Krav (8) - App'en skal gøre det muligt for en bruger at logge ud

LogOut

Vil du logge ud? [Logout](#)

Når en bruger vil logge ud, skal vedkommende gå til forsiden og vælge “Logout” knappen. Applikationens logudfunktion læner sig meget op af loginfunktionen. En token bliver givet til brugerens klient ved indlogging og gemmes i local storage. Når brugeren selv trykker logud eksekveres metoden localStorage.removeItem("token"), hvilket fjerner vedkommendes token, og gør at brugeren ikke længere er logget ind, og derfor bliver videresendt til /login.

Krav (9) - App'en skal kunne vise en liste over en aktuels brugers matches

Når en bruger har fået matches, skal brugeren også kunne se en liste over de givne matches. Listen viser firstName og lastName på de matchede brugere. Til at hente dataen, skal der bruges data fra match- & user-tabellen. Fra match-tabellen anvendes inner join til at hente first- og lastname på userId2. Dette gøres ud fra den bestemte userId1 matches.

```
SELECT userId1, userId2, firstname, lastname  
FROM match  
INNER JOIN [user] u ON u.userId = match.userId2  
WHERE userId1 = ?
```

Krav nummer (10): App'en skal kunne vise en fuld profil for et potentiel match

Når en bruger vil se en fuld profil for et potentiel match skrives firstName ind i vores søgeform. Det firstname bliver sendt til vores database, hvor det meste af brugerens metadata bliver filteret fra således, at den bruger der har sendt requestet står tilbage med de 3 koloner der er vist på billedet.

Wellcome to
SOTELL 2021:
Her kan du se dine
matches

FirstName	LastName	MatchId
Mikkel	Rolf	7

Ser du noget, du ikke kan lide?
[Slet Match](#)

Krav nummer (11): App'en skal give brugeren mulighed for at fjerne et match igen

Hvis en bruger har fået et match, men har fortrudt, skal det være muligt at fjerne matchet igen. Funktionen deleteMatch følger funktionen for deleteUser, og der vil derfor ikke gåes mere i dybden med den i dette krav. Dog som det kan ses i SQL statement nedenunder, bliver matchet slettet ud fra matchId. Selve match-tabellen består af userId (1 & 2), som viser id'et for den ene user og id'et for den anden.

```
"Delete FROM [match] where matchId = @matchId"
```

Var I ikke et match alligevel?

Nu har det aldrig været nemmere at slette sit match

Alt du skal er, at skrive matchId'et ind og voila, du kan forstsætte søgningen



ADMIN

ADMIN LOGIN

Lordlangholz@jegerbbbar
.....
ADMIN Login

	WHERE	ORDER BY
1	adminId = 1	email
2	adminId = 2	password
3	adminId = 3	email

Grundet brugen af Azure Functions, har vi lavet en ny trigger til hver adminfunktion. Da azure functions ikke tillader brugen af keywordet “Admin”, er disse kaldet for “Special..xx”.

I databasen er der lavet et særskilt “Admin” table med 3 admins. Ved loginsiden kan der vælges “login som admin”. Her bliver vedkommende videreført til en særskilt login side, der udelukkende tillader de 3 admins oplysninger. I Db.js er denne funktion næsten identisk med den generelle loginfunktion, SQL querien tager dog fra [admin] tablet i stedet for [user] som tidligere.

A screenshot of a database management tool showing the "admin" table structure. The table has four columns: "adminId" (numeric(4)), "adminEmail" (varchar(150)), "adminPassword" (varchar(150)), and "admin_pk" (adminId). It also shows two unique indexes: "admin_admin_id_uindex" on "adminId" and "admin_admin_email_uindex" on "adminEmail".

```
// Admin Login
function adminLogin (payload) {
    return new Promise((resolve, reject) => {
        const sql = 'SELECT * FROM [admin] where email = @email AND password = @password'
        const request = new Request(sql,(err,rowCount) =>{
            if (err){
                reject(err)
                console.log(err)
            } else if( rowCount == 0){
                reject({message: "no user found"});
            }
        });

        request.addParameter('email', TYPES.VarChar, payload.email)
        request.addParameter('password', TYPES.VarChar, payload.password)

        request.on('row',(columns) => {
            resolve(payload.isAdmin);
        })
        connection.execSql(request)

        return "you are now logged in"
    });
}

module.exports.adminLogin = adminLogin;
```

Krav (12) - App'en skal give admin mulighed for at opdatere en brugerprofil

UPDATER SYSTEMETS BRUGERE

VIL DU OPDATERE EN BRUGER? [OPDATER HER](#)

På administratorens homepage kan man tilgå endpointet: <http://localhost:8080/#/SpecialUpdateUser>. Her kan admin isoleret opdatere brugeres forskellige oplysninger. Admins opdaterfunktion fungerer meget på samme måde som brugerens. Dog har vi lavet det således, at der er en trigger til hver funktion. Dette er hovedsageligt implementeret da vi på daværende tidspunkt i udviklingsfasen så dette som den bedste måde at imødekomme denne kravspecifikation. En anderledes måde at gøre det på, ville værekun at have en “SpecialUpdate” funktion, der styrede det hele. Ved en videreudvikling af applikationen, kunne man argumentere for, at dette ville gøre kodebasen både mere overskuelig og intuitiv.

Opdater emallen på en bruger:



Opdater passwordet på en bruger:



Opdater firstName på en bruger:



Opdater LastName på en bruger:



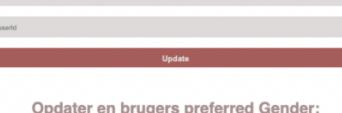
Opdater hvilket hotel en bruger befinder sig på:



Opdater en brugers alder:



Opdater en brugers preferred Gender:



Krav (13) - App'en skal give admin mulighed for at slette en bruger

Delete User

Vil du slette en bruger i systemet? [SLET HER](#)

På admins homepage kan administratoren vælge knappen “Slet her”. Herefter bliver vedkommende ledt videre til <http://localhost:8080/#/SpecialDeleteUser>. Her indtastes et UserId og en email på den pågældende bruger, man gerne vil slette.

Krav (14) - En admin skal have mulighed for at se brugsstatistikker af appen - Hvor mange brugere systemet har, samt hvor mange matches systemet har:

Når admin tilgår sin homepage, kan vedkommende vælge “Se brugerstatistikker”. Her kommer en liste, med de eksisterende brugeres emails samt user id. Brugerne bliver hentet fra databasen vha. et SELECT statement og et GET request. På samme vis som tidligere forklaret, bruger vi axios til at fetcche dataen og displaye den på vores frontend. Email og UserID bliver fremvist, da det er disse parametrer admin skal kende til for at kunne slette en user.

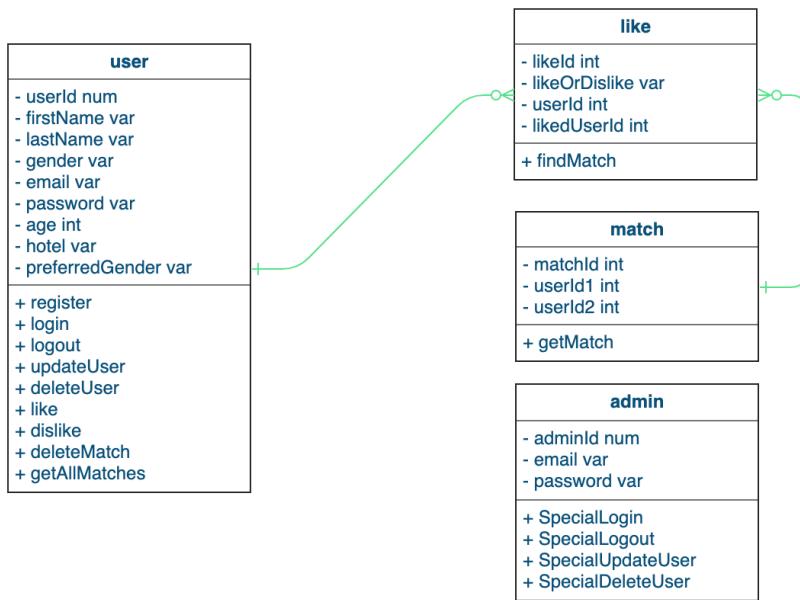
Krav nummer (15): App'en skal indeholde en matching algoritme, som præsenterer brugeren for eventuelt interessante profiler.

Når en bruger benytter sig af “like” funktionen får pågældende vist en række brugere. I stedet for at alle brugere skal hentes, hvor en stor del højest sandsynligt kan fravælges ud fra relevante parametre, benytter vi en matchingalgoritme. Matchingalgoritmen implementeret i vores applikation finder relevante brugere inden for parametrene; hotel og foretrukket køn. En mere dybdegående forklaring af matchingalgoritmen følger i algoritmeafsnittet.

Krav nummer (16): En bruger skal kunne sætte et køn samt aldersramme han/hun leder efter.

Dertil skal en bruger kun blive vist profiler som opfylder dette filter. Når en bruger tilgår vores applikations forside, kan vedkommende sætte en aldersramme for, hvem de ønskes matchet med. I databasen findes disse vba. et SELECT statement, der SELECTER de brugere, hvis alder samt køn passer til de givne parametrer.

Klassediagram



Før den tekniske udførelse af SOTELL begyndte, udarbejdede vi et klassediagram. Helt specifikt er klassediagrammet brugt til at konkretisere vores problemområde, samt skabe en visualisering af applikationens grundmodel. Vi har udviklet en applikation, hvor den respektive bruger kan finde en partner ud fra en meget simpel algoritme, der tager parametrene køn og hotel ind.

Applikationens klasser

Applikationen har fire klasser, hvor userklassen er den mest benyttede klasse. Userklassen indeholder størstedelen af funktionerne i applikationen, men der er endvidere klasser for like, match og admin. Hvis vi designede databasen således, at vi havde en user og en like klasse, ville vi ende ud i et many to many forhold. Da useren kunne like mange usere og mange usere kunne like den givne user. Derfor har vi valgt at benytte os af et decompositiontable, hvor vi tager Primary Key'en fra user og bruger den til at beskrive Primary Key'en i liketabellen. Forholdet mellem user og likes er derfor one to many, fordi en bruger kan like flere potentielle matches, men de potentielle matches kan kun

like den bruger en gang. Dertil er forholdet mellem match og like one to many, fordi at ét match, består af flere likes. Vi vil ikke have et many to many forhold, fordi der derved opnås en masse gentagelser, og 1. normalform er som konsekvens af dette ikke opfyldt.

Systemet har en klasse for admin, hvor en adminbruger bliver hardcoded ind i databasen. Admin har ingen forbindelse eller relation til de øvrige klasser. Et alternativ til dette, ville være at tilføje en isAdmin property til user-klassen, hvor værdien kunne være eksempelvis 0 eller 1. Hvis en user har 1 som værdi i isAdmin, kunne vores loginfunktion godtage dette, og man ville derefter blive videreført til adminHomepage. Denne konstruktion medfører dog højest sandsynligt en unødvendigt kompliceret loginprocess for den gængse bruger, og er derfor ikke implementere som vores endelige løsning.

Vi har valgt at opbygge klassediagrammet således, da dets simplicitet giver et godt kodningsmæssigt udgangspunkt. Det opfylder normalformerne, er intuitivt og forholdsvis overskueligt at implementere. Trods de positive sider af simpliciteten, er vi bevidste om, at der ved en videreudvikling af appen burde tilføjes flere attributter til klasserne. Eksempelvis ville det i en social app som denne, være en fordel, hvis brugeren kunne uploadet et billede af sig selv. Hvis en opskaléringsproces påbegyndes, og dette problem belyses for sent, kan man argumentere for, at det er en klar forretningsmæssig begrænsning, at dette ikke er sket noget tidligere, da det ofte er dyrt og omfattende at ændre i databasen sent i en proces.

STORAGE

I en dating applikation er det nødvendigt at gemme data. Vi har valgt at gøre dette, ved at bruge en normaliseret SQL database, som er kendtegnet ved at data og dens relationer bliver gemt i tabeller. I en SQL database arbejder man med konceptet Primary Key - også kaldet PK. Primary Key'en indeholder information, der er unikt for det row, i den tabel som den tilhører. Ud fra denne Primary Key kan man definere yderligere Foreign Keys - også kaldet FK. Foreign keys kan bruges til, at opbygge et dekomposition table og derved opnå en højere normalform. I de følgende afsnit gennemgås SQL databaser generelt, normalformer og en dybere begrundelse for, hvorfor vi har valgt at opbygge vores database som vi har, vil blive forelagt for læser.

SQL

SQL står for structured query language og er sproget, der anvendes i relationelle databaser. Relationelle databaser strukturerer data i tabeller. Denne databaseform er brugbar i vores applikation,

fordi man kan skabe relationer mellem forskellige tabeller, ved brug af foreign keys. Det tillader at datastrukturen er på en højere normalform. Der findes flere normalformer, hvortil hver har forskellige krav. For SOTELL har målet for datastrukturen været 3. normalform - også defineret som 3NF. For at man opnår 3NF skal 1. - og 2. normalform være opnået¹. Ved 1NF må hver celle kun indeholde en værdi, og hver kolonne skal være unik. Ved 2NF gør man brug af foreign keys til at opdele tabellerne, og en row skal have en unik Primary Key. Hvis en tabel har en sammensat nøgle, skal alle felter, der ikke indgår i nøglen afhænge af den samlede nøgle. Ved 3NF må der ikke findes felter udenfor PK, som er indbyrdes afhængige.

Dataopbygningsform

Vi anvender Microsoft Azure SQL database til opbevaring af dataen. Applikationens database er opbygget ud fra principperne omkring normalisering for at undgå gentagelser af data på tværs af databasen og for at øge hastigheden af INSERT, UPDATE og DELETE. Dette er til fordel for applikationen, da der ved den normaliserede datastruktur eksempelvis ikke er én kolonne med alle likes en user har, men derimod en tabel, der viser hvem en user har liket. Hvis likes var gemt i user-tabellen, med en kolonne der gemte alle likes som en liste, ville et SQL statement der opdaterede et like være således:

```
UPDATE [user] SET likes = 40, 50, 52, 54' WHERE userId = 55
```

Man ville ende ud i, at skulle skrive alle likes for den gældende user, hvilket ville være en yderst kompliceret process, hvis en user eksempelvis har 100 likes. Denne struktur ville ikke følge normalisering, fordi der vil være mere end én værdi i en enkelt celle. Derfor er matches og likes uddelt til sine egne tabeller, så databasen kan være normaliseret. Både match og like benytter sig af foreign keys fra user; user.userId FK.

Databasen godtager flere forskellige datatyper. Til PK benytter vi integers, da man på denne måde, kan autoincrements når der oprettes en ny kolonne. Med dette menes at vores userId (PK) stiger med 1 ved oprettelsen af hver ny bruger, fremfor at en bruger selv skal indsætte et id. Integers er hele tal, og egner sig derfor godt til Id. Ydermere kræver autoincrement en talværdi for at stige hver gang. Til øvrige kolonner bruger vi primært varchar, hvortil vi har defineret en max længde for de indtastede tegn. Dette gøres, for at spare på den mængde af plads, der reserveres til kolonnen i databasen.

¹ Jesper Charles B(2015), [EPub]
Normalisering af database
<https://balslev.io/programmering/database/>

SQL statements

DDL & DML

For at definere og manipulere datastrukturer i SQL tabeller bruges der hhv. DDL og DML statements. DDL står for data definition language og DML står for Data Manipulation Language. Til applikationens funktioner bruges der DML SQL statements. Disse bruges hhv. til at vælge (SELECT), indsætte (INSERT), opdatere (UPDATE) og slette (DELETE) data i databasen. Vi tager derudover også brug af Conditional filtering med (WHERE) klausulen. DDL statements er i denne applikation udelukkende brugt til at oprette databasens tabeller i applikationens tidlige udviklingsfase, og læser vi derfor i følgende afsnit blot få en dybdegående gennemgang af de brugte DML statements og disses tilhørende funktionaliteter.

SELECT bestemmer hvilke columns der skal inkluderes i resultatssættet fra den query der er udformet, FROM bliver brugt til at identificere hvilke(n) tabel dataen i resultatssættet skal trækkes fra, og hvordan tabellerne skal joins.

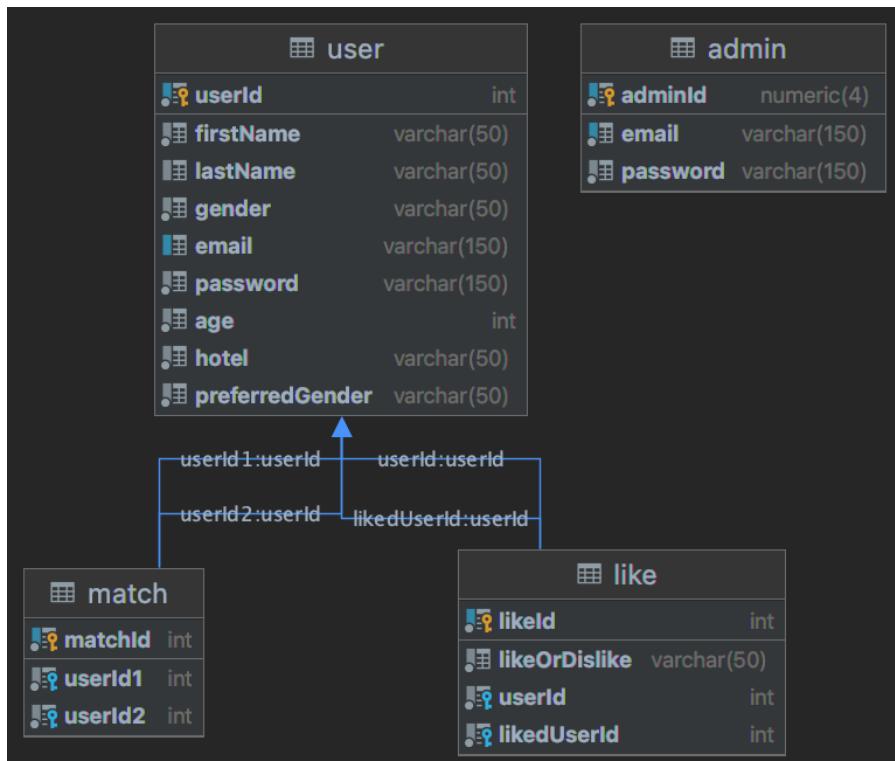
INSERT bruges til at indsætte data. Queryen starter med INSERT efterfulgt at hvor det skal indsættes, hvilke værdier der skal indsættes, og i hvilken rækkefølge de skal indsættes i. Se gennemgangen af Krav (1) for en eksemplificering af dette.

UPDATE bliver brugt til at opdatere et row, som allerede eksisterer. Her er det derfor nødvendigt at bruge WHERE klausulen. I SOTELL applikationen bruges den til at ændre på allerede udfyldte rows. (Se Krav (3) for eksemplificering). Tager vi udgangspunkt i en user fra vores database, kunne man vælge at tillade at nogle af informationer om brugeren forbliver tomme under registreringsprocessen, så der eksempelvis kun bliver indtastet Navn, age, email og password. Det kunne fra et forretningsmæssigt perspektiv være en fordel for applikationen, da det ville kræve en mindre tidsinvestering fra brugeren, og derfor ville brugeren være mere tilbøjelig til at give applikationen en chance, hvor man i andre applikationer før har set, at selve registreringsprocessen tager så lang tid, at det får potentielle brugere af applikationen til at vælge den fra.

DELETE bliver på samme måde som UPDATE brugt til at manipulere et allerede eksisterende row i en tabel. Hvor UPDATE opdaterer et row, sletter DELETE hele rowet for den PK. Da "email" er sat til at være unique i vores database, benyttes den fremfor en PK til at slette rowet. Dette valg er taget, grundet faktummet at vores dekrypterede JWT token ikke indeholder brugerens PK, men andre oplysninger inklusiv deres email. Et alternativ til dette ville være, at JWT tokenen indholdte PK'en og man dermed kunne bruge denne i deleteprocessen.

ERD:

Da vores app lagrer data i en database, har vi udviklet et ERD diagram. “Entity relation diagram” er et strukturelt diagram der bruges til databasedesign. Det klargører de forskellige enheder i systemets scope, samt hvad de interne forhold er. Ydermere er det et stærkt kommunikativt redskab, der sørger for, at alle parter er enige før udarbejdelsen af programmet påbegyndes.



BACKEND

Algoritmer

Applikationens hovedformål er, at brugere kan matche med hinanden. En bruger skal ideelt set, kun blive eksponeret for de brugere, som algoritmen mener passer sammen baseret på forskellige tilkendegivne parametre. For SOTELL skal brugere kunne matche ud fra om hvorvidt de er på samme hotel og deres “kønspræferencer” passer sammen. Til dette kan man bruge comparison og logical operators. If statements tjekker eksempelvis, if users.hotel er det samme for begge parter. Denne

algoritme er den langsomste, fordi den går igennem hver enkelt bruger en af gangen, og tjekker hvilket hotel de bor på.

Udvælgelsen af brugere i vores system, sker ud fra den algoritme Microsoft har valgt passer til vores data i SQL serveren, da vores applikation gør brug af en query, som leder efter de users, der er på samme hotel, og søger det ønskede køn. Algoritmen leder efter users, der er på et bestemt hotel, har et bestemt køn, og foretrækker et bestemt køn. For en kvindelig beboer på Tivoli hotel, der er interesseret i mænd, vil matchingparametrene være således: hotel = Tivoli, gender = female, preferredGender = male og parameteret for userID skal være brugerens eget userID, så man ikke kan matche med sig selv.

```
SELECT userId, firstName, lastName, gender, age, hotel, preferredGender  
FROM [user]  
WHERE (hotel = ?) AND (gender = ?) AND (preferredGender = ?) AND (userId != ?)
```

Queryen gør brug af “logical operators” AND og NOT. Hvor AND bestemmer de yderligere parametre der skal tages med, sikrer NOT at brugerens egen data ikke bliver taget med som en foreslået profil. I fremtiden kunne det være en forretningsmæssig fordel, hvis queryen blev videreudviklet til også at inkludere et alderssspænd og geografisk placering, som ville gøre det muligt at bruge geolocation.

Algoritmer kører med forskellige effektiviteter. Særligt for en datingapplikation, der er bygget på serverless computing, burde algoritmen være meget effektiv. I takt med at datingapplikationen vokser, bliver matching algoritmens hastighed vigtigere og vigtigere. Algoritmens hastighed er i høj grad afhængig af den SQL engine query'en kører i og kompleksiteten af querien.

Den anvendte matchingalgoritme er relativt simpel, og søger kun ud fra tre parametre.² I hurtigste instans, kan algoritmen køre i $O(\log(n))$ og i langsommeinstans kan den køre $O(n)^3$. Ønskes der at begrænse tiden querien er nødsaget til at køre, kunne man implementere TOP klausulen. Denne begrænser mængden af resultatet, der bliver fundet. Dette vil ikke være relevant for den nuværende algoritme, fordi de brugere der findes af algoritmen, alle passer lige godt. Man kunne argumentere for, at den kunne anvendes og “toppen” af listen, så var de brugere, der bor på samme hotel som dig.

² Aleksei Jegorov(2016), [EPUB]

Estimate time complexity of Java and SQL query.

<https://blog.devgenius.io/estimate-time-complexity-of-java-and-sql-query-afa13a88a981>

³ SQL Server technical eBook series(2018), [EPUB]

<https://info.microsoft.com/rs/157-GQE-382/images/EN-GB-CNTNT-eBook-DBMC-SQL-Server-performance-faster-querying-with-SQL-Server%20%281%29.pdf>

Ulempen ved dette ville dog være, at du så kunne matche med folk fra andre hoteller, og jf. applikationens hovedformål vil dette ikke være ideelt.

Serverless computing

Ved serverless computing er alt hardware til servers styret og kontrolleret af en serviceudbyder. I denne applikations tilfælde benyttes Azure. Til Azure har man et abonnement hvortil man betaler for den beregningskraft man anvender. Serverless computing er smart for en applikation, som skal kunne udvides på et senere tidspunkt. Fra et forretningsperspektiv er serverless computing et oplagt valg, fordi man udelukkende skal skrive koden til applikationen, og ikke have styr på den bagvedliggende infrastruktur. Praktisk fungerer det ved, at Microsoft har datacentre der håndterer og udbyder den server der er nødvendig for at drive applikationen.

Funktionerne i applikationen er event-driven, hvilket vil sige, at de bliver triggered af et request. På denne måde, betaler man kun for den beregningskraft det kræver at køre den funktion. Altså vil det blive dyrere jo mere funktionerne bliver kørt, som på den måde gør applikationen skalerbar. Det kan eksemplificeres ved at kigge på forskellen mellem beregningskraften krævet til at drive SOTELL med 20 brugere, og beregningskraften krævet for at drive tinder med flere millioner brugere.

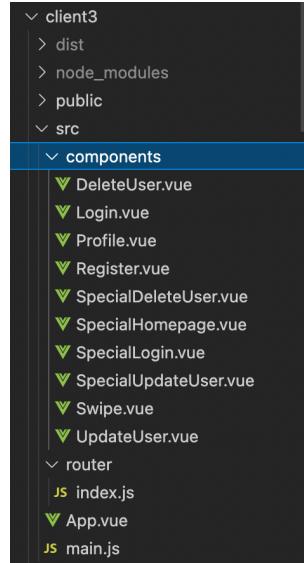
Tedious

Klientens request bliver fetchet til URLen fra vores Azure functions. For at API'et kan tale med vores database er det nødvendigt at implementere TDS protokollen eller Tabular data stream. Vi har i vores applikation benyttet Tedious, som er en Node.js pakke. Dette valg er blandt andet taget, fordi der er et stort fællesskab online, som benytter sig af Tedious, hvilket betyder, at der er mange ressourcer at trække på. Tedious gør det muligt at kommunikere med instanser af Microsoft SQL server⁴.

FRONTEND

Applikationens frontend er blevet udarbejdet i Vue.js' framework samt HTML og CSS. Vue går under betegnelsen et MVVM framework - Model-View-ViewModel - der står i modsætning til MVC frameworkt, som nævnt tidligere. I dette afsnit gennemgås SOTELLS' frontendopbygning, samt de generelle overvejelser vi har gjort os om denne.

⁴Tedious(2021), [dokumentation]
<https://tediousjs.github.io/tedious/>



Vue.js

Vue.js er et javascript framework der bruges til at bygge frontend user interfaces. Vue er forholdsvis simpelt at komme i gang med, og derfor har vi valgt dette framework til udviklingen af vores UI.

Frameworket giver mulighed for at bygge komponenter, der enkapsulerer data i ens javascript og connecter herefter denne til et interaktivt template i HTML. Vue bruges hovedsageligt for dets simplicitet, men også dets mulighed for at kunne opskaleres forholdsvis nemt.

Vi har i vores applikation valgt at dele frontendmappen op således:

Router

“Routing” er en af de metoder i Vue, der muliggør at brugeren af siden kan skifte mellem sider uden at den refresher konstant. Dette simplificerer navigationen i applikation.

```
< import Vue from 'vue'
import Router from 'vue-router'
import Register from '@/components/Register'
import Login from '@/components/Login'
import Profile from '@/components/Profile'
import Swipe from '@/components/Swipe'
import UpdateUser from '@/components/UpdateUser'
import SpecialLogin from '@/components/SpecialLogin'
import SpecialHomepage from '@/components/SpecialHomepage'
import SpecialUpdateUser from '@/components/SpecialUpdateUser'
import SpecialDeleteUser from '@/components/SpecialDeleteUser'
```

Efter oprettelsen af de nødvendige komponenter, konfigureres de routes der skal til, for at kunne navigere mellem komponenterne. Import metoden bruges til, at importere de ønskede komponenter samt vue og Router modulet fra “vue-router module”⁵. Alternativt kunne vue være installeret med vue-cli, hvortil vue-router havde været installeret by default. Dette ville ikke påvirke ydeevnen, men detaljer som disse er altid gode at gøre sig overvejelser omkring, da det bidrager til den gode kodeskik.

```
export default new Router()
{
  routes: [
    {
      path: '/',
      name: 'Register',
      component: Register
    },
    {
      path: '/login',
      name: 'Login',
      component: Login
    },
    {
      path: '/profile',
      name: 'Profile',
      component: Profile
    },
    {
      path: '/swipe',
      name: 'Swipe',
      component: Swipe
    },
    {
      path: '/updateUser',
      name: 'UpdateUser',
      component: UpdateUser
    },
    {
      path: '/SpecialLogin',
      name: 'SpecialLogin',
      component: SpecialLogin
    },
    {
      path: '/SpecialHomepage',
      name: 'SpecialHomepage',
      component: SpecialHomepage
    },
    {
      path: '/SpecialUpdateUser',
      name: 'SpecialUpdateUser',
      component: SpecialUpdateUser
    },
    {
      path: '/specialDeleteUser',
      name: 'SpecialDeleteUser',
      component: SpecialDeleteUser
    }
  ]
}
```

⁵ vuejs.org(2021) lokaliseret d. 5 maj på: <https://router.vuejs.org/guide/advanced/lazy-loading.html>

Sidst i “routemappen”, konfigurer vi “routes” for at få dem til at fungere. “Routermetoden” tager et array af objekter, der tager hvert components properties:

- path: “the path” til componenten
- name: navnet på componenten
- component: view componenten

VUE routeren har mere avancerede parametrer og metoder, men til denne applikation valgte vi at holde det simpelt. k

Vue er et fordelagtigt værktøj at bruge til applikationer af denne art. Set i perspektiv af en eventuel opskalering af produktet, er det fordelagtigt at vi har brugt vue fra start af. Det ville kræve ekstra timers arbejde at omskrive kodden fra javascript og HTML filer til VUE eller det andet populære frontendframework REACT.

Axios

En af de mest fundamentale opgaver for en webapplikation er at kunne kommunikere med servere igennem HTTP-protokollen. I denne opgave er dette blevet udført vba. Axios-biblioteket i Node.js.

Axios er et HTTP library der bruges til at sende data fra en frontend til en backend eller et API. Det gør det nemt at sende asynkrone HTTP requests til REST endpoints og udføre CRUD operationer. Der er mange nære alternativer til Axios, som f.eks. Ajax, Fetch osv. Vi har valgt at bruge Axios i vores applikation, da det har en simpel syntaks og sparer os for nogle linjer kode.

Da vi har bygget en applikation, hvor der skal manipuleres med data fra en frontend har vi valgt at bruge Axios. Axios kan bruges til POST requests, hvis man ønsker en applikation, der kan gemme brugerinput/data på en server. Når en bruger eksempelvis registrerer sig på vores registreringsendpoint, sendes der et POST request vba. en “post method” gennem Axios til vores server. Axios serialiserer automatisk JavaScript objekter til JSON når det sendes igennem post funktionen. Når requestet er lavet, returnerer axios et promise, der enten resulterer i et responsobjekt eller et “error objekt”. Et promise medfører, at hele programmet ikke bliver sat i stå, indtil det er færdigt med at “fetche” dataen. Tværtimod sender axios dataen “til siden”, prøver at fetche dataen. Tager dét for lang tid, vil det ikke stoppe resten af javascriptet fra at

```
81 |   },
82 |   methods: {
83 |     onsubmit() {
84 |       axios
85 |         .post("http://127.0.0.1:7071/api/register2/", {
86 |           firstName: this.firstName,
87 |           lastName: this.lastName,
88 |           gender: this.gender,
89 |           email: this.email,
90 |           password: this.password,
91 |           age: this.age,
92 |           hotel: this.hotel,
93 |           preferredGender: this.preferredGender
94 |         })
95 |       .then((responce) => {
96 |         console.log(responce)
97 |       })
98 |     }
99 |   }
100 | }
```

fungere. Når requested er succesfuldt, modtager vores .then() callback et responsobjekt med følgende "properties":

- data: payload returneret fra serveren.
 - Som default forventer axios JSON formateret data og parser det tilbage til et JavaScript objekt for os.
- Status: HTTP-koden returneret fra serveren
- statusText: HTTP-statusbeskeden returneret af serveren
- Headers: headers sendes tilbage fra serveren
- Config: den originale request configuration
- request: det aktuelle XMLHttpRequest-objekt

Hvis der er et problem med requestet, bliver promiset rejectet med et error-objekt, der indeholder følgende:

- message: en error besked text
- response: respons objektet (hvis modtaget)
- request: the aktuelle XMLHttpRequest objekt (hvis den kører i en browser)
- config: den originale request configuration.

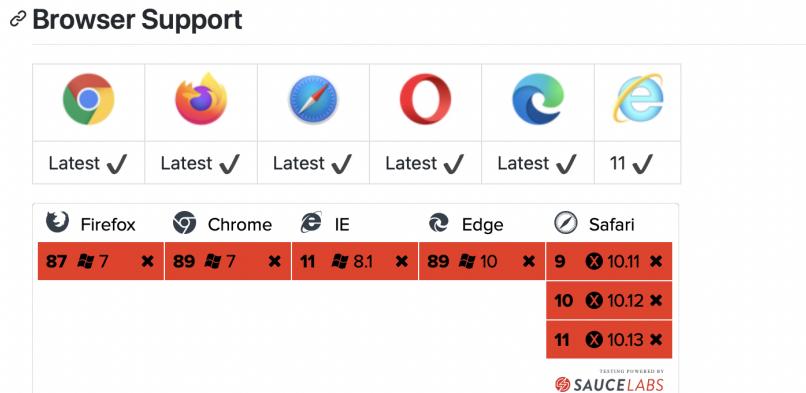
Da vores HTTP request er et promise, bruger vi try/catch til at fange eventuelle fejl.

Ydermere understøttes axios af de fleste browsere, og det er også med denne overvejelse i mente vi har valgt at tage brug Axios

Alternativt til Axios

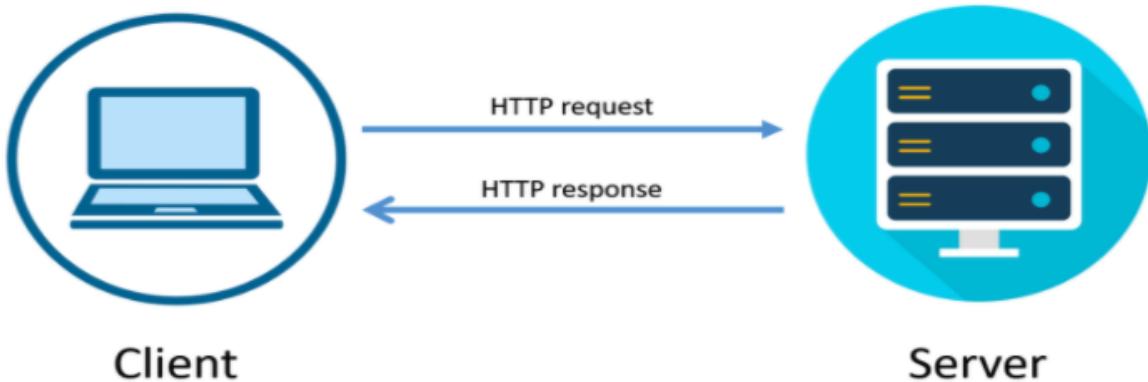
Alternativt til axios kunne vi have brugt fetch() metoden i Javascript eller jQuery AJAX. Ved brug af fetch, skal man igennem flere steps for at få JSON data igennem. Først skal

metoden kaldes og derefter skal .json() metoden kaldes på responset. Vi skulle dermed have ventet på dataen og derefter eksempelvis lavet det til en variabel og kaldt .json() metoden på det.



HTTP-requests

For at udarbejde en applikation der passer til vores investors ønsker, er det nødvendigt at have kendskab til HTTP-requests. I dette afsnit vil de relevante HTTP-requests for vores applikation blive gennemgået.



HTTP står for hypertext transfer protocol er en kommunikationsprotokol, som definere en række forespørgsler en klient kan anmode om for at effektuere en effektiv kommunikation mellem klient og server. http-request dækker over de anmodninger, som klienten kan sende til et request URL hvilket i vores applikation er GET, POST, PUT og DELETE.

GET request bliver benyttet til at modtage data fra en ressource på en server. Det et GET request gør er, at den henter det data, som bliver opfanget af request-URLen. I vores applikation bruger vi det eksempelvis til, at få alle de users en given user har matchet med. Vi bruger det også til at få vist en fuld profil for et givent match.

POST request bliver benyttet til, at sende data til en server. Det der bliver sendt med en post request er arkiveret i request.body for http requestet. I vores applikation benytter vi bl.a. POST request til at registrere brugere og logge ind i applikationen.

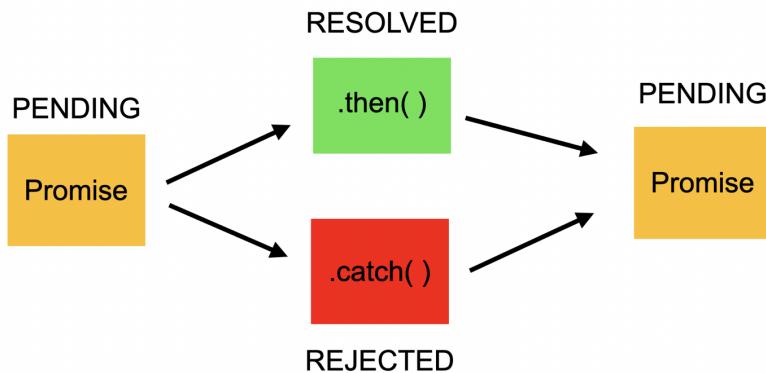
PUT request minder om POST request da de begge sender data, men vi bruger PUT-request til at opdatere data i vores database.

DELETE request er en metode, som bliver brugt til at slette data fra et givent URL. Vi bruger DELETE request til at slette brugerens profil fra databasen, men det kunne også sagtens bruges til at fjerne et match.

Promises

Promises er en måde, hvorpå vi kan håndtere asynkrone operationer i JavaScript. Et promise kan have 2 "outcomes". Enten bliver det resovlet eller rejected. Når vi beder om data fra serveren er det i "pendingmode" indtil vi modtager det. Modtager vi dataen bliver det resovlet, modtager vi den ikke, bliver det rejected. Alternativt til det eksemplificeret i kravspecifikationerne, kunne vi have brugt

“klassiske” callback functions, hvilket også virker fint ved asynkrone operationer. I de givne situationer havde det ikke gjort den store forskel, men skal man tænkte i opskallering af projektet, kan callback funktioner nemt ende ud i en ret uintuitiv kodestruktur, hvis der skal køres flere asynkrone operationer samtidig - også kaldet “callback hell” blandt programmører. Ved brug af promises kan vi ved brug af .then .catch metoden “attatche” callbacks i stedet for at skulle passe dem, og dermed få en meget skarpere og overskueligt kodestruktur.



TESTING

Testing af en applikation er essentielt for at kunne sikre et produkt af høj kvalitet. Vi har løbende testet programmets forskellige funktionaliteter, for at kunne opfange problemer i udviklingsfasen og ikke ende i en situation, hvor vi står med et færdigudviklet program, der skal ændres på. Fra et forretningsmæssigt perspektiv er dette ideelt, da det kan være både dyrt og besværligt at skulle fikse komplikationer senere hen i processen.

Der er mange forskellige måder at teste på. Vi har i vores applikation kørt løbende testing vba. Postman og har console.logget eventuelle fejl, for at vide, hvor problemerne lå. Sidst har vi testet det færdigudviklede produkt med Unittesting.

Postman

Postman er en simpel måde at teste forskellige funktionaliteter i et program. Postman er en populær API klient, der gør det nemt at teste, dele og dokumentere API's. I Postman kan man kreere og

gemme alt fra simple til komplicerede HTTP requests og samtidig læse responset. Især i den første fase af produktet, hvor vi skulle have vores database og http triggers til at tale sammen, var postman meget brugbart. Da vi derved kunne udvikle logikken i backenden uden at have en frontend.

Unit Testing

Under udarbejdelsen af denne applikation har vi lavet en unittest på kravspecifikation 3 - Appen skal tillade en bruger at forblive logget ind. Vi har brugt jest-localstorage-mock modulet, der kører med Jest, hvor man kan “mocke” reelle situationer. Først satte vi vores moduler op, både i jest.package.json, hvor vi indsatte en “setupFile”. Herefter kørte vi en test, der undersøgte om vores applikation formåede at gemme i localStorage.

```
test('gemmer den i localStorage', () => {
  const KEY = 'foo',
    VALUE = 'bar';
  dispatch(action.update(KEY, VALUE));
  expect(localStorage.setItem).toHaveBeenCalledWith(KEY, VALUE);
  expect(localStorage.__STORE__[KEY]).toBe(VALUE);
  expect(Object.keys(localStorage.__STORE__).length).toBe(1);
});
```

Vi valgte lige netop denne test, da det er en vigtig del af vores applikation, at en bruger kan gemme oplysninger i localStorage for at forblive logget ind. Alternativt kunne man have decideret tjekket om den gemte en token.

PROCESEVALUERING

Ved udviklingen af SOTELL har vi gennemgået mange løsningsovervejelser, og er stødt på mange problemstillinger, vi metodisk har fået løst. Vi har prøvet så vidt som muligt at inkorporere både udfordringer og løsninger løbende i vores rapport, da vi synes dette på en fin måde fremlægger applikationens process for læseren. Næstskrevede er dog nogle generelle problemer vi alle nikker genkendende til, har været særligt udfordrende men samtidig utroligt læringsrige.

LocalStorage og tokens

Til at starte med gemte vi ingenting i LocalStorage, når en bruger loggede ind. Dette problematiserede processen, og gjorde det umuligt for en bruger at forblive logget ind. Herefter gemte vi brugerens email i LocalStorage. Senere indså vi dog, at dette heller ikke var ideelt, og tildelte efterfølgende

brugeren en token ved inlogging. Dog blev brugeren ikke “vertificeret”, hvis de gik til et andet endpoint, end det de loggede ind på, hvilket ville sige, at alle kunne tilgå alle endpoints. Sidste step i udviklingen af denne feature, var derfor at sørge for, at brugeren kunne forblive logget ind, ved at verificere om hvorvidt, der befandt sig en token i LocalStorage eller ej. Ingen i gruppen havde arbejdet med tokens før, og dette var derfor utroligt givende for den indlæringsmæssige process.

Frontend

I den første lange del af projektet var vores frontend udelukkende baseret i HTML. Vi havde lavet utroligt enkle forms, der sendte http requests til vores endpoints vba. fetch. Dette var udelukkende fordi, vi fandt det simpelt og overskueligt, og vi sidste år havde lært at bruge expressserveren, hvortil vi havde tilkoblet simple HTML sider. Da vi kom lidt længere i processen ville vi gerne udvikle et frontend, der så mere professionelt ud. Ingen af os havde haft med frontendframeworks at gøre før, og vi tog derfor et kursus på CodeCademy i Vue.js, hvilket viste sig at være utroligt givende. Det har klart forbedret brugeroplevelsen af vores applikation, og gjort os til bredere favnende programmører.

Azure portalen

Vi stiftede først bekendtskab med Azureplatformen, da vi blev undervist i det i foråret. Her havde ingen i gruppen nogen erfaring med brugen af denne og de kvaliteter Azure kan tilbyde, og der blev derfor brugt mange timer på, at sætte sig ind i programmet. Vi var alle godt indforståede med hvordan opbyggelsen af et API, som vi lærte i sidste semester, foregik, så at skulle kanaliserer den viden over i et nyt projekt som dette, hvor der kan drages på erfaringer, men man samtidig skal være omstillingsparat var meget interessant.

SQL

Tidligere programmeringsopgaver der har krævet en database er fra alle i gruppen blevet udarbejdet i Mongo.db. Alt fra SQL statements til hele kodestrukturen var derfor ny og tog en del tid at forstå og sætte sig ind i. Efterfølgende kan vi dog konkludere, at vi alle er meget positivt stemt overfor videre at begive sig ud i, at arbejde med SQL på egen hånd.

KONKLUSION

Vi har udviklet applikationen SOTELL. Opgaven er blevet grebet an fra et programmeringsmæssigt perspektiv, men løbende har vi haft forretningsmæssige overvejelser med. Vores håb om at skabe mere end bare en dating-app er blevet opfyldt, og med SOTELL appen er “Hotelmatching” blevet gjort til en ting. Appen er konstrueret med henblik på at kunne skaleres ud, og den løser dermed vores ønske om, potentielt at kunne give ensomme rejsende bedre muligheder for at møde folk. Applikationen lever op til 12 ud af de 16 kravspecifikationer, hvoraf flere af dem hovedsageligt mangler grundet et tidsmæssigt problem. Vi har her fået specificeret hvad, hvordan og hvorfor vi ville implementere dem. Ydermere har vi skrevet en rapport, der har kortlagt udviklingsprocessen af applikationen og dermed givet læser samt eventuelle investorer et unikt indblik i, ikke mindst hvad SOTELL er blevet til, men også hvad den kunne blive til. Vi har lagt meget energi i både programmet og rapporten og er utroligt tilfredse med slutresultatet. Alt dette håber vi også er en opfattelse læser sidder tilbage med.

LITTERATURLISTE

Jesper Charles B(2015), [EPub]

Normalisering af database

<https://balslev.io/programmering/database/>

SQL Server technical eBook series(2018), [EPUB]

<https://info.microsoft.com/rs/157-GQE-382/images/EN-GB-CNTNT-eBook-DBMC-SQL-Server-performance-faster-querying-with-SQL-Server%20%281%29.pdf>

Aleksei Jegorov(2016), [EPUB]

Estimate time complexity of Java and SQL query.

<https://blog.devgenius.io/estimate-time-complexity-of-java-and-sql-query-afa13a88a981>

https://www.plus2net.com/sql_tutorial/sql_like.php

vuejs.org(2021) lokaliseret d. 5 maj på: <https://router.vuejs.org/guide/advanced/lazy-loading.html>