Programeringsmanual for Password Manager

Dette dokumentet skal lære deg koden og hvordan gå videre med prosjektet. Filer er skrevet i rekkefølgen brukeren møter på de forskjellige scriptene.

Electron applikasjonen

Det er noen store endringer som må skje i denne applikasjonen for at programmet skal virke som tiltenkt, da den mangler et «preload-script». Dette scriptet tilrettelegger for kommunikasjon mellom «rendererscriptet» (rendererscriptet er det som kjører som vanlig linket javascript) og «mainscriptet» (mainscriptet er hovedscriptet som kjører Electron applikasjonen) dette konseptet heter «Inter-Process Comunication» eller IPC. Sånn det er satt opp nå virker ikke IPC i programmet, men det er ikke veldig vanskelig å rette opp i.

Her er stegene:

- Lag en preload.js fil
- Uncomment scriptet i main.js, som loader preloadscriptet
- I preload.js skriv ned to funksjoner som er save og open files
    - De skal ta signal fra renderer og påkalle noen funksjoner som er i main.js
- Endre måten rendererscriptet påkaller disse save og open funksjonene
    - Det står: ipcrenderer.send() dette må bli: window.apiName.Function() bytt ut apiName og Function til det du kaller det i preload.js

De forskjellige scriptene er dokumentert i rekkefølgen som de er skrevet i så viss funksjon oliver kommer for funksjon trygve vil funksjon oliver komme først i dette dokumentet.

For å kjøre Electron applikasjonen må du ha følgene:

- Kode fra https://github.com/TheCyberiousPizzerious/pwdmanager
- Modulene som koden trenger
    - Fra root cd in i src/
    - Kjør `npm install electron –save-dev sqlite3`

Nå har du alle modulene du trenger for å kjøre applikasjonen. Du kan se i package.json alle npm kommandoene som er definert. Dette ser du under delen som heter «scripts» for å teste er du interessert i den som heter «start». Du kan endre dette til «electron .» men «electron-forge start» virker også det tar bare lengre tid. For å starte er kommandoen «npm start» dette burde starte prosjektet og etter en stund burde det dukke opp et vindu.

Main.js

```
const { app, BrowserWindow, ipcMain, dialog } = require('electron');
const path = require('node:path');
const sqlite3 = require('sqlite3').verbose(); // Require things dont know if i need these here
//const path = require('path')
```

Først importeres node modulene som brukes i programmet.

```
  // Some build things from the site
  if (require('electron-squirrel-startup')) {
    app.quit();
  }
```

Denne delen av koden har noe med å «builde» programmet å gjøre.

```
function createWindow () {
  const mainWindow = new BrowserWindow({
    // Height controls
    show: true,
    width: 810,
    height: 600,
    minWidth: 800,
    minHeight: 500,
    //Options
    webPreferences: {
      nodeIntegration: true, //Allows render scripts to use modules
      contextIsolation: false //Allows to use modules outside of preload and main
      //preload: path.join(__dirname, 'preload.js') //Preload script loader
    }
  })
  mainWindow.loadFile('src/index.html'); // HTML file to load
  //mainWindow.webContents.openDevTools(); // Auto open devtools
};
app.on('ready', createWindow); // When everything is loaded make the window
```

Dette er koden til selve Electronapplikasjonen. La oss gå gjennom koden bit for bit:

Const mainwindow = new BrowserWindow({}) – Denne linjen bare definerer et instans med nytt vindu. Inne i BrowserWindow ser vi mange innstillinger som definerer vinduet. Først definerer vi høyde og bredde, så minimum verdier av de to. Dette for å kunne jukse litt med cssen og ikke trenge at det skal være alt for dynamisk.

Det neste er webPreferences dette er viktige innstillinger for hvordan IPC skal foregå eller om det skal være nødvendig i det hele tatt. En av de vanlige grunnene til at man ønsker å kommunisere med main processen er for å få tilgang til Node moduler og Electron funksjoner. Dette er fordi det er «main.js» som blir kjørt av node, og derfor vanligvis i andre applikasjoner den eneste filen med tilgang til node moduler. Derimot i Electron kan man gi de andre js filene som ikke er kjørt av node tilgang til node moduler ved innstillingene «nodeIntegration: true» og «contextIsolation: false». ContextIsolation er den moderne versjonen og jeg møtte på noen problemer med å bruke node moduler i andre filer uten nodeIntegration. Disse to funksjonene gjør i teorien at man ikke trenger et «preload» script. Dessverre virker ikke programmet helt som forventet, og renderer filene snakker ikke med main funksjonen.

Den nest tingen vi ser i browserWindow funksjonen er mainWindow.loadFile() her skriver man bare ruten til filen som skal vises når man starter programmet. Dette er da src/index.html/ siden den ser etter filer fra root. Det siste vi ser er app.on() som ligger utenfor, den sier bare at når programmet har lastet ferdig, så skal vinduet lastes slik at ingenting blir utelatt.

```
ipcMain.on('open-file', async (event, result) => {
  try {
    const result = await dialog.showOpenDialog(mainWindow, {
      title: 'Select a File',
      defaultPath: '~/password.db',
      buttonLabel: 'Open',
      properties: ['openFile'], // Specifies that it is only one file that should be selected
      filters: [{ name: 'SQLite Databases', extensions: ['db'] }] // Only.db files can be opened
    });

    if (!result.canceled && result.filePaths.length > 0) { // if not canceled and making sure only one file has been selected
      event.sender.send('file-selected', result.filePaths[0]) // Send message and file selected
      console.log(result.filePath)
    } else {
      event.sender.send('open-canceled'); // Just sends something to send something
    }
  } catch (err) {
    event.sender.send(err)
    console.error(err); // if err show err
  };
});

ipcMain.on('save-file', async (event, result) => { // Waits for 'save-file'
  try {
    const result = await dialog.showSaveDialog(mainWindow, { // wait for user to select
      title: 'Select File Location',
      defaultPath: app.getPath('documents'),
      buttonLabel: 'Save',
      filters: [{name: 'Database', extensions: ['db']}]
    })
    if (!result.canceled) { // if not canceled send the locations and thins
      event.sender.send('location-chosen', result.filePath);
    } else {
      event.sender.send('open-canceled')
    }
  } catch (err) {
    event.sender.send(err)
    console.error(err)
  }
})
```

De to største funksjonene i main programmet er to «listeners» som bare skal vente på signaler fra enten preload eller renderer. Det virker dessverre ikke med å sende fra renderer sånn det ser ut. Dialog.showOpenDialog() er en Electronfunksjon som åpner et ferdiglaget dialogvindu for å velge filer. Koden er ganske selvforklart og ser kanskje skummel ut, men det og ta med seg herfra er at du selv bestemmer navnet på det du sender og venter på, så ipcMain.on('RAAAAAAAAAA', () => {console.log(jegvirker)}) hadde vært en gyldig ting å vente på, så lenge preload eller renderer sender «RAAAAAAAAAA».

Renderer.js

Dette er koden som må inn i Electron applikasjonen:

Rendererscripts som de heter i Electron er javscript filene som vanligvis hadde kjørt vanlig javascript kode. De kan kommunisere med mainscriptet gjennom preload.js scripts. Dette skriptet er for index.html det første du møter på som bruker.

```
// Getting all the nesecarry elements
const opendbBtn = document.getElementById('opendb-btn');
const errM = document.getElementById("err-p")

//Importing Node modules
const ipcRenderer = require("electron");

opendbBtn.addEventListener('click', async () => {
  ipcRenderer.send('open-file'); //Send request
  ipcRenderer.on('file-selected',(event, filePath) => {
    const dbPath = filePath; // Getting the first item in a array since the function returns a array
    console.log("Selected file: ", dbPath);
    //Check if the file selected is a .db file
    if (dbPath && dbPath.toLowerCase().endsWith('.db')) { // Checks if it is valid
      console.log(dbPath, " checked and it is .db");
      localStorage.setItem("openDBPath", dbPath); // Store for later extraction
      window.href = 'opendb.html'; // Change window to open the db
    } else {
      console.log('Auda, The file selected is not a valid file, make sure that files end in .db');
      errM.innerHTML = "The file selected is not a valid file. Make sure that the file is a .db file";
    }
  });
  ipcRenderer.on('save-canceled', (err) => {
    console.error(err);
  })
});
```

Det første vi gjør, som kan virke overaskende, er at vi bruker «require()», og dette er mulig siden vi har satt contextIsolation til false. Når brukeren trykker på openBtn sender vi en request til main scriptet direkte, og dette påkaller open-file funksjonen i main.js. Dette kan egentlig bli flyttet til opendb.js, slik at det ikke skjer før du har flyttet deg til filen der det skal vises. Dette tror jeg blir både raskere og mer logisk. Main filen returnerer en filePath, som brukeren har definert gjennom et dialog vindu, som dukker opp. Så skjekker koden om filen som er valgt er en «.db» fil. Om det er en .db fil lagrer den det i localstorage, og sender deg til opendb.html, om ikke returnerer den en error. IpcRenderer.on('save-canceled') kommer ikke med en error men heller en melding om at brukeren har kanselert i dialog vinduet det er egentlig ikke nødvendig å printe som console.error().

Createdb.js

Createdb.js er enda et renderer script som inneholder koden for å lage en database, og å definere et table.

```
// Getting in all the elements
const continueBtn = document.getElementById("Continue-button");

// Elements for showing errormessages
let errM = document.getElementById("error-p");

// Getting all the Modules that i need
//const { remote } = require('electron')
const sqlite3 = require('sqlite3')
//const sqlc = require('sqlcipher') // Cirker
const path = require('node:path');
const ipcRenderer = require('electron');
//const { errorMonitor } = require('node:events');
//const { fs } = require('fs')
```

Det første som skjer, er at jeg definerer alt jeg trenger for scriptet som inkluderer et el for error og noen node moduler.

```
// Ask for main save location dialog
function politeMainAsk() {
  ipcRenderer.send('save-file')
  ipcRenderer.on('location-chosen', (event, filePath) => {
  });
};
```

Det neste er en ubrukt funksjon, som spør main om save-file funksjonen. Den vil åpne et dialog vindu for å velge et sted å lagre databasefilen. Funksjonen er ikke brukt siden det ikke virker med ipc mellom renderer og main selv med nodeIntegration true og contextIsolation false.

```
// Kind of the main function with anonymous function
continueBtn.addEventListener('click', () => {
  //Import al needed elements
  let newDBBeskrivelse = document.getElementById("newdb-beskrivelse").value;
  let newdbPassord = document.getElementById("newdb-password").value;
  let passwordCheck = document.getElementById("newdb-password-check").value
  let newDBNavn = document.getElementById("newdb-navn").value;
  console.log(
    "Tingene: \n",
    "Beskrivelse: " + newDBBeskrivelse + ",\n",
    "Navn: " + newDBNavn + ",\n",
    "Password: " + newdbPassord + ",\n",
    "Password check: " + passwordCheck
  )
  console.log("Navnet:" + newDBNavn + "og beskrivelsen:" + newDBBeskrivelse)
  // Check if any required fields are empty
  if (newDBNavn == '' || newdbPassord == '' || passwordCheck == '') {
    errM.innerHTML = "Please fill inn all the required fields";
  } else {
    if (newdbPassord != passwordCheck) {
      errM.innerHTML = "The passwords do not match";
    } else {
      // here ipcRenderer is needed
      const dbPath = path.join(__dirname, newDBNavn + '.db'); // Make filename and path
      //Choose filelocation needed

      const db = new sqlite3.Database(dbPath); // Makes a new instance of sqlite3
      const createTableQuery = // Creating Query
        `
        CREATE TABLE IF NOT EXISTS password (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        Tittel TEXT,
        Brukernavn TEXT,
        Passord TEXT,
        URL TEXT,
        Notes TEXT,
        Dato TEXT,
        Endret TEXT
        );`;
        // SQL command to create the table
      db.run(createTableQuery, (err) => { // for queries that return a single row
        if (err) {
          console.error('Error creating table:', err.message);
        } else {
          console.log('Table created successfully.');
        }
        db.close(); // Close the database connection
      })
    }
  }
});
```

Den siste funksjonen er en veldig stor, men egentlig ikke veldig komplisert. Først defineres alle input elementene, og så skrives de til konsollen. Deretter sjekker den om noen av de nødvendige feltene er tomme, og om de er det, så sender den en error melding. Om feltene er fylt inn sjekker den om passordene er like, og så printer en error om de ikke er. Når passordene er like lager den en ny sql

instans, for så å definere table til databasen. Til slutt kjører den sql kommandoen og lukker sql instansen.

Opendb.js

Opendb.js er enda et renderer script som skal håndtere åpningen av .db filer

```
openSelected() // Opening the query and printing (not yet tho)
// Node modules import
const app = require('electron');
const path = require('path');
const sqlite3 = require('sqlite3').verbose();

// Importing elements from the DOM
const insertDBRow = document.getElementById("new-row-el")
```

Det første som skjer at den kjører en funksjon som er definert lengre nede, og utenom det henter vi noen moduler samt et element for å printe ut det som står i databasen.

```
function getLocalPath() {
  const storedDBPath = localStorage.getItem("openDBPath");
  return storedDBPath;
};

function openSelected() {
  const storedDBPath = getLocalPath()
  //localStorage.clear() // Hvordan
  const db = new sqlite3.Database(storedDBPath); // New sqlite instance
  const query = `SELECT * FROM password`; // Defining the query
  db.all(query, (err, rows) => { // Actually doing the query
    if (err) {
      console.log(err.message);
      return; // Why return here
    }
    console.log(rows) // Dette er alt den innheneter så trenger noe "for" loop
  });
  db.close()
};
```

Den første funksjonen henter ut pathen vi lagrer i render.js. Den andre funksjonen openSelected er den som kjører queryen i databasen, og den henter kun ut all informasjonen.

```
insertDBRow.addEventListener('click', () => {
  // Shitten skal inn her som er under =>


});

const data = { // Need getLocalPath
  dbPath: '/path/to/your/database.db', // Replace with the actual path to your database
  Tittel: 'Sample Title',
  Brukernavn: 'Sample Username',
  Passord: 'Secret Password',
  URL: 'http://example.com',
  Notater: 'Sample Notes',
  Dato: '2023-11-18',
  Endret: '2023-11-18',
};

console.log(data);

const db = new sqlite3.Database(data.dbPath);
// Insert data into the table
const insertQuery = `INSERT INTO password (Tittel, Brukernavn, Password, URL, Notes, Dato, Endret) VALUES (?, ?, ?, ?, ?, ?, ?)`;

const insertValues = [data.Tittel, data.Brukernavn, data.URL, data.Notes, data.Dato, data.Endret];

db.run(insertQuery, insertValues, function (err) {
  if (err) {
    return `Error inserting data: ${err.message}`;
  }
  return `Data inserted successfully with ID: ${this.lastID}`;
});
db.close();
```

Den siste funksjonen er ikke i en funksjon, dataen som blir definert er bare placeholder data. Så blir det definert et nytt instans med sql. Deretter defineres hva som skal settes inn, og så kjøres alle verdiene i databasen.

Det som du kanskje har merket mangler fra koden er en måte å endre det som står her er speudo kode på hvordan man kan redigere:

redigereBnt.eventlistener(() => {

definere alle input fileds

info = get * from denne id

fyll inn input fieldene med info

ferdigBtn.eventlistener ({

definere alle input fields på nytt så de får den nye infoen

sql query for å lagre

lukk

})

})

Det mangler også for å slette, og her er mer pseudo kode:

sletteBtn.eventlistener({

ny sql instans

for alle el med denne id drop

lukk

})

Serveren

Under er stegene for å kjøre serveren (jeg antar du allerede har koden):

- Installer node `sudo apt install node`
- Installer de nødvendige modulene
- Åpne "server.js" og fjerne konstanten som heter «address»
- Kjør `node server.js`, dette starter serveren

Om du har fulgt alle stegene burde du kunne gå til http://dinip å se nettsiden. Om det ikke virker kan du alltids kjøre den på localhost. Du må enten opprette databaser til «sessions» og «pwdmanager» eller kommentere ut alt i «session storage». Selv om serveren din kjører på localhost burde folk kunne koble seg til netsiden din så lenge porten du hoster på er åpnet.

Her en en komando du kan kjøre fra katalogen (directory på norsk) hvor server.js ligger.

```

npm install express crypto-js express-session body-parser express-mysql-session

```

Server.js

```
//--------------------------------//
//     * Defining Elements *      //
//--------------------------------//
const express = require("express");
const cjs = require("crypto-js");
const mysql = require('mysql');
const session = require('express-session');
const bodyParser = require('body-parser');
const MySQLStore = require('express-mysql-session')(session);

const app = express();
const port = 8008;//What port the server is running on
const address = "10.0.0.16";//Define the ip address
const key = "#)7avwKsEndQdE2pkv^i";//Random key
```

Som vanlig starter koden med at vi definerer nødvendige moduler. Det er noen andre ting her som også blir definert som: porten serveren skal listene på, adressen serveren skal kjøre på og en key. Denne keyen som blir definert burde egentlig bli generert tilfeldig, men å hardcode keyen burde holde for nå.

```
//--------------------------------//
//        * sql connections *     //
//--------------------------------//

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'root',
  database: ''
});

const sessionConfig = {
  host: 'localhost',
  port: 3306,
  user: 'root',
  password: 'root',
  database: ''
};
```

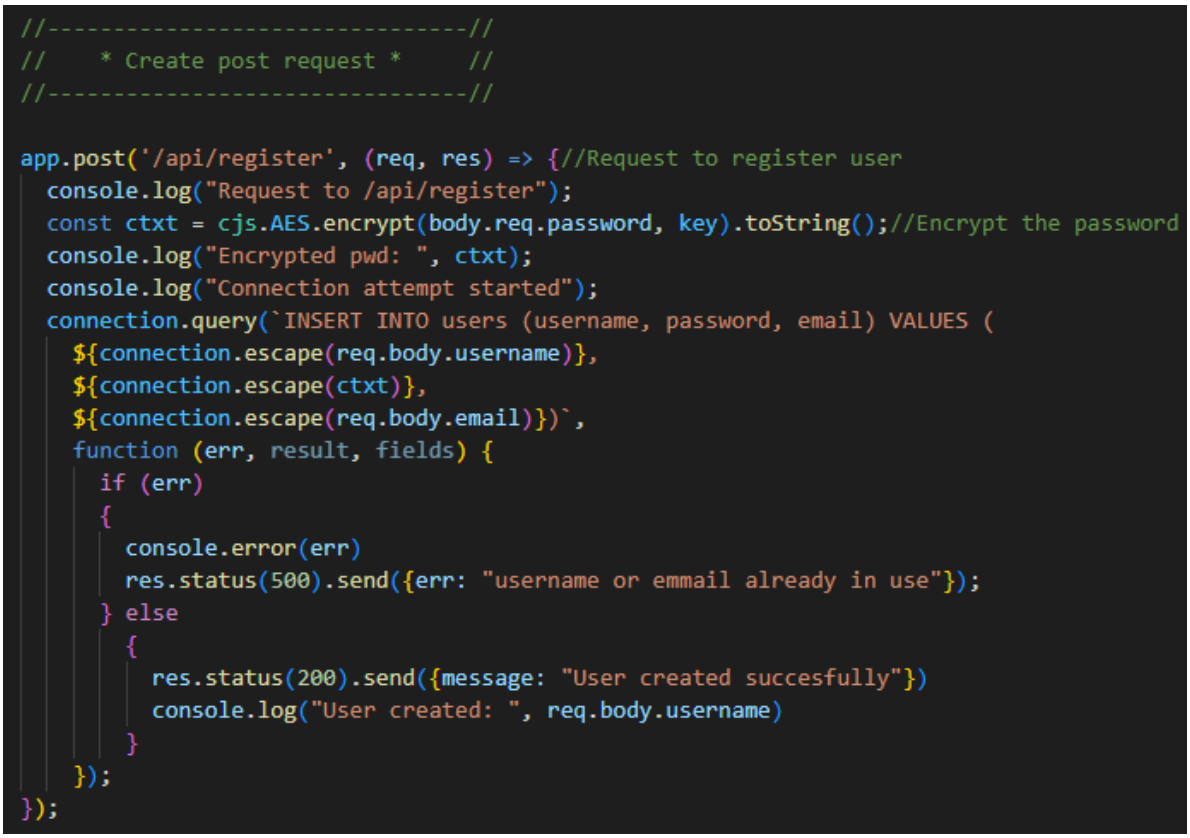
Så defineres begge sql forbindelsene vi trenger. Det konstante objektet som heter «connection» sin database verdi skal være 'pwdmanager' og sessionConfig sin database verdi skal være 'sessions'.

```
//--------------------------------//
//       * Session storage *      //
//--------------------------------//

const storeSession = new MySQLStore(sessionConfig);//New sql instance
connection.connect();//Connect to the database
app.use(bodyParser.json());//For parsing json requests
app.use(session({//sql session info that should be stored
  key: 'session_cookie',
  secret: 'session_secret',
  store: storeSession,
  resave: false,
  saveUninitialized: true,
  cookie: {
    secure: false,
    maxAge: 999999//How long the session will last
  }
}));
```

Her lager koden en session som bare virker sånn halvveis. Det er noen problemer med at du må logge på igjen noen ganger og koden ville nok gitt mer mening å bare bruke cookies istedenfor sessions

Først defineres en ny sql session instans, deretter kobler koden seg til den nye instansen. Vi parser json requests før vi definerer hva en session skal inneholde.



```
//--------------------------------//
//    * Create post request *     //
//--------------------------------//

app.post('/api/register', (req, res) => {//Request to register user
  console.log("Request to /api/register");
  const ctxt = cjs.AES.encrypt(body.req.password, key).toString();//Encrypt the password
  console.log("Encrypted pwd: ", ctxt);
  console.log("Connection attempt started");
  connection.query(`INSERT INTO users (username, password, email) VALUES (
    ${connection.escape(req.body.username)},
    ${connection.escape(ctxt)},
    ${connection.escape(req.body.email)})`,
    function (err, result, fields) {
      if (err)
      {
        console.error(err)
        res.status(500).send({err: "username or emmail already in use"});
      } else
        {
          res.status(200).send({message: "User created succesfully"})
          console.log("User created: ", req.body.username)
        }
    });
});
```

Det neste i koden er to post funksjoner register og login, først register. Register er det register.js kaller på. La oss gå gjennom koden bit for bit. Først informerer den oss om at det kommer en request til register. Så krypteres passordet før det så lages det en sql query som parser brukernavn, passord og emailen til brukeren. Til slutt defineres en funksjon som egentlig kan være anonym med svarene til de som lagde post requesten.

```
app.post("/api/login", (req, res) => {//Request to login
  console.log("Request to /api/register");
  const ctxt = cjs.AES.encrypt(body.req.password, key).toString();//Encrypt the password
  console.log("Encrypted pwd: ", ctxt);
  console.log("Connection attempt started");
  connection.query(`SELECT * FROM users WHERE username = ${connection.escape(req.body.username)} AND passw
  function (err, result, fields) {
    if (err)
    {
      console.error(err);
      res.status(500).send({err: "Something went wrong"});
    } else
      {
        if (result.lenght > 0)
        {
          console.log(`User ${req.body.username} logged in!`);
          res.status(200).send({message: "Login succes yay!"});
          req.session.logedin = "true";//Create session when logged in
          req.session.username = req.body.username;
          req.session.save();
        } else
          {
            console.error("Wrong pwd or username ", err);
            res.status(401).send({err: "username or password is wrong"});
          }
      }
  });
});
```

Neste er en veldig lik den forrige requesten. Først krypterer den passordet som brukeren skrev inn. Så henter den all informasjonen om brukeren hvor passordet matched passordet og bruker navnet matcher brukernavnet. Så er det en funksjon som sender svar til den som sendte requesten med feilkoder. Om alt er riktig så starter den en session med brukernavnet til brukeren som brukernavnet i sessionen.

```
//--------------------------------//
//     * Create get requests *    //
//--------------------------------//

app.get("/", (req, res) => {
  res.sendFile(__dirname + "/index.html");
});

app.get("/download", (req, res) => {
  res.sendFile(__dirname + "/src/html/download.html");
});

app.get("/docs", (req, res) => {
  res.sendFile(__dirname + "/src/html/docs.html");
});

app.get("/faq", (req, res) => {
  res.sendFile(__dirname + "/src/html/faq.html");
});

app.get("/team", (req, res) => {
  res.sendFile(__dirname + "/src/html/team.html");
});

app.get("/logout", (req, res) => {
  res.sendFile(__dirname + "/src/html/logout.html");
  req.session.destroy();
});

app.listen(port, () => {//Listen on this port
  console.log(`Server running at ${port}`);
});

app.use(express.static('src')); // Make so that the user cannot acess anything outside of /src/
```

Her i koden sier vi at om brukeren spør om for eks. /docs skal vi sende dem filen docs.html. Dette gjør vi da for alle sidene vi ønsker at skal være tilgjengelig for brukeren. Så sier vi at serveren skal listene på porten vi spesifiserte tidligere i koden. Så Til slutt med express.static bestemmer vi at brukeren ikke skal kunne nå noen ting utenfor src/.

Register.js

```
const registerBtn = document.getElementById("register-btn");// Define the button element
registerBtn.addEventListener('click', register());//create eventlistener

const errM = document.getElementById("error-el");

async function register() {//defining register function
    let username = document.getElementById("username").value;//
    let password = document.getElementById("password").value;
    let email = document.getElementById("email").value;

    console.log(
        username,
        password,
        email
    );

    if (username == '' || password == '' || email == '')
    {
        errM.innerHTML = "Please fill in all the fields!";
    } else
    {
        const data =
        {
            username: username,
            password: password,
            email: email
        };
        console.log(data);
        const conf =
        {
            method: 'POST',
            headers:
            {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        };
        console.log(conf);
        const req = await fetch('/api/register', conf);
        console.log(res);
        const json = await req.json();
        console.log(json);
        if (response.status == 500)
        {
            console.error("response: ", 500);
        } else if (response.status == 200)
        {
            console.log("response: ", 200, " user created");
            window.location.href="/login"
        } else
        {
            errM.innerHTML = "Something went wrong";
        }
    }
};
```

Først defineres knappen som skal kalle på funksjonen og et DOM element for å printe feilmeldinger. I register funksjonen starter det med at verdiene til de forskjellige input elementene blir definert. Så gjennom en if statement sjekker dem om noen av elementene er tomme om de er printer den en feilmelding om de fylt ut går den videre. I else statementet som kommer etter definerer den hva som skal sendes i et objekt som heter data. Så lages det et konfigurasjons objekt som heter conf, i dette conf objektet spesifiseres det hvordan vi skal sende requesten og hva slags request det er. Så sender den en request til /api/register sammen med conf objektet. Så definerer den svaret i json konstantent. Etter dette er det noen if statements som håndterer hva den skal gjøre om den får de forskjellige svarene fra /api/register.

Login.js

```
const loginBtn = document.getElementById("login-btn");//Get the button for logging in
loginBtn.addEventListener('click', login());//adding eventlistener

errM = document.getElementById("error-el");//element for printing error

async function login() {//defining a login() function
    let username = document.getElementById("username").value;//Getting the values of the input elements
    let password = document.getElementById("password").value;

    if (username == '' || password == '') //Checks if the fields are empty
    {
        errM.innerHTML = "Please fill in all the fields"
    } else //If the fields are not empty
    {
        const data = //Makes a object to send
        {
            username: username,
            password: password
        };
        console.log(data);
        const conf = //Make a config object to send
        {
            method: 'POST',//Specify the way to send
            headers: //Headers so info about what is being sent
            {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data) // Makes the data object to a json object
        };
        console.log(conf);
        const req = await fetch('/api/login', conf);//await response from /api/login
        console.log(req)
        const json = await req.json();// await the req const
        console.log(json)
        if (response.status == 200)//If this message do this
        {
            console.log("User logged in");
            window.location.href = "/download";//Change window location
        }
    }
};
```

Login koden er nesten akkurat det samme som register koden eneste forskjellen er at den sender requsten til et annet sted og sender noe annen informasjon.