Advanced CRUD Database Project

Introduction

As part of my job as system administrator I have a responsibility of the daily management of our Cinema software “Vista”. As I administrate the different level of databases one of my struggles is the management of a rational database which requires separating the data. Which in term requires information to be stored in different areas of the program. This can be time consuming as requires lots of opening, closing different windows to manage a user or item.

To solve this issue, I want to create a much simpler program that allows me to access and view all information as a User Management system minimizing the closing of windows.

System Overview

Backend

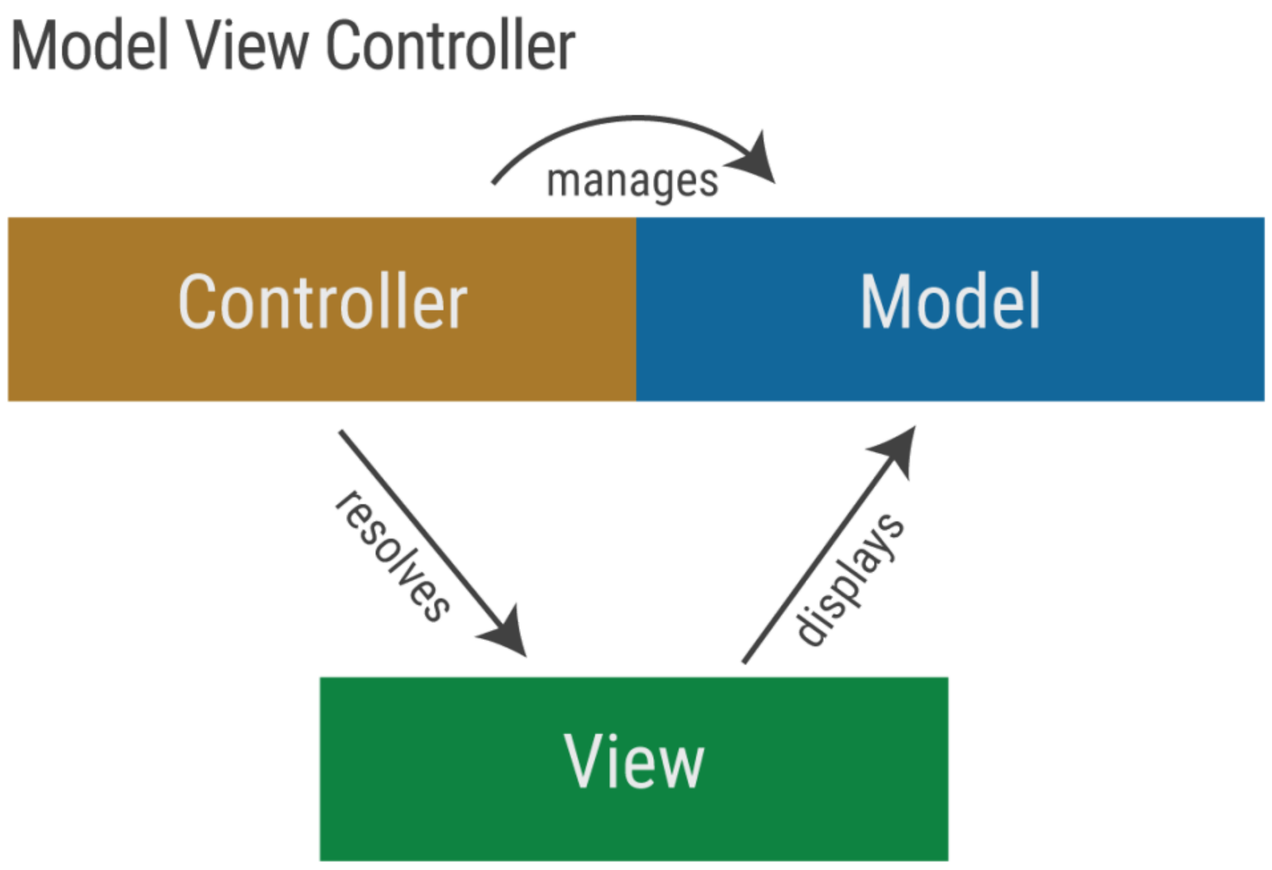
To support me with my assignment I required to installed a few dependencies to help achieve such tasks.

- NodeJS
- ejs
- Express
- Mongoose
- Morgan
- Nodemon
- Body-parse
- dot env

Project Structure

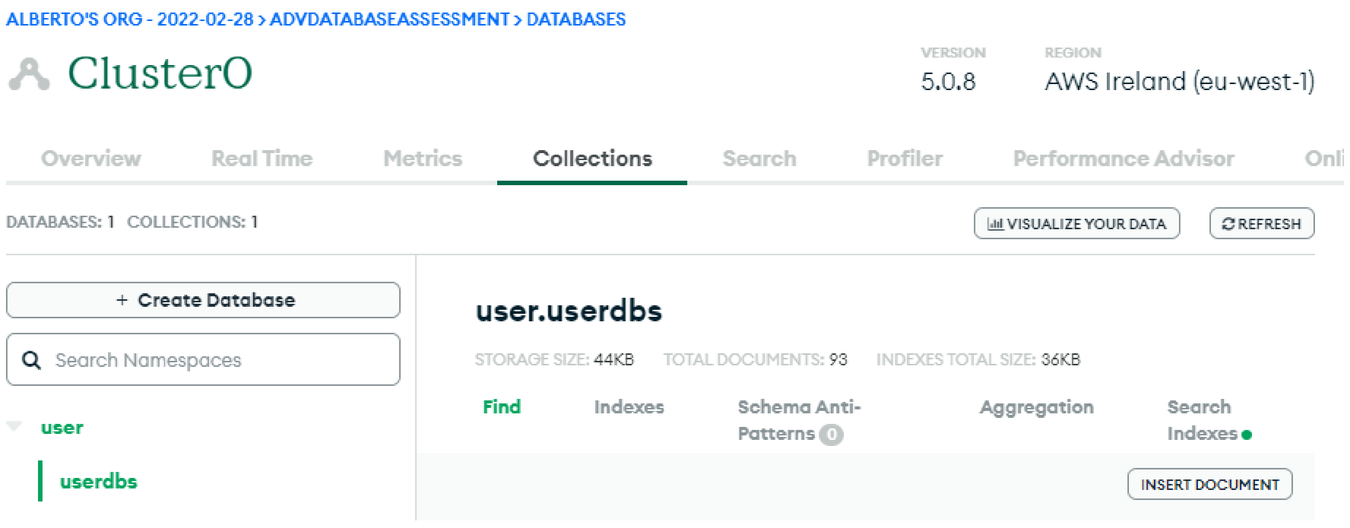
Model - View - Controller Framework

Using the MVC structure we are able to separate the application into three logical components: the model, the view and the controller. These components are built to handle specific developments of the application. This structure is is frequently used as an industry-standard web development framework to create scalable projects.



Model

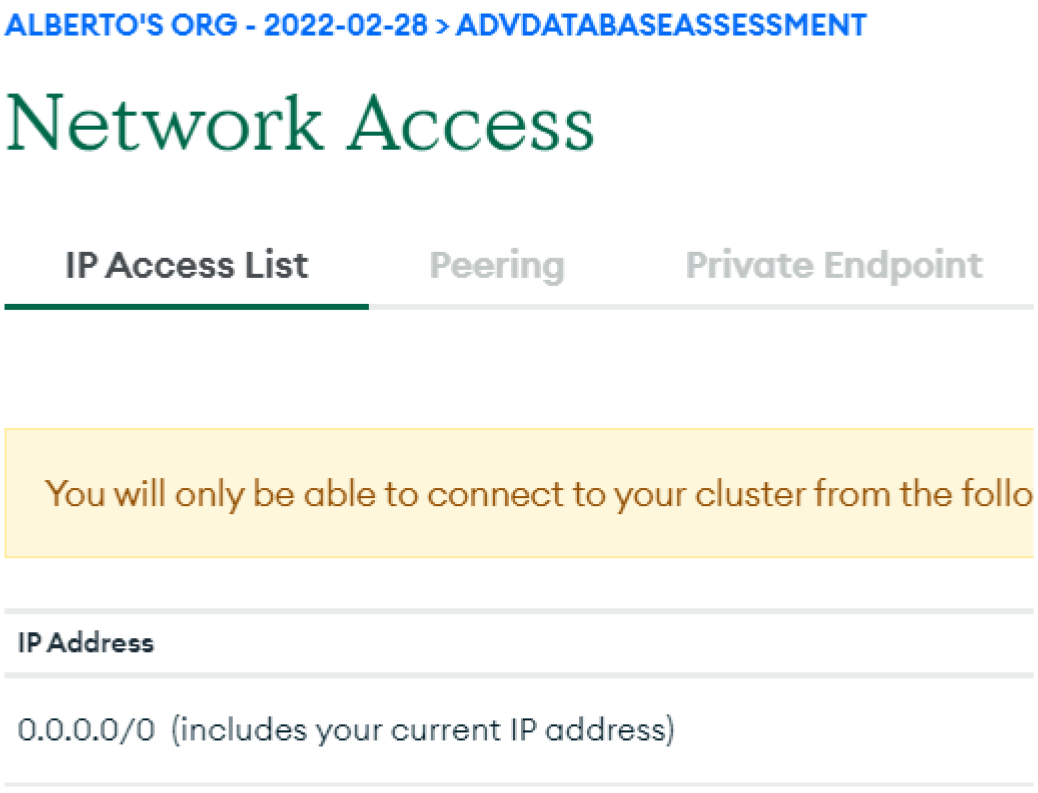
The model looks at the data, for this project MongoDB is the data storage client where we create, write, edit and delete to.



Connecting the the application is handled by the `Controller`. One of the reason as why MongoDB is a suitable application is due to the ability to connect directly as a host, through suitable providers. This format of availability is used via MongoDB Atlas. A more simple tool to use, where the UI is more friendlier to the eye is MongoDB Compass.

As this project is based of Atlas, to allow access to the database we require to adjust another security feature that MongoDB provides. Network Access. We this can be adjusted in the security/network section and can add that only specific IP Address can access. Suitable when working on a private project.

As we require access to this data from several. unknown, IP Address' we can set the ip to 0.0.0.0/0. As long as you have the admin and password to this database then its accessible from anywhere.

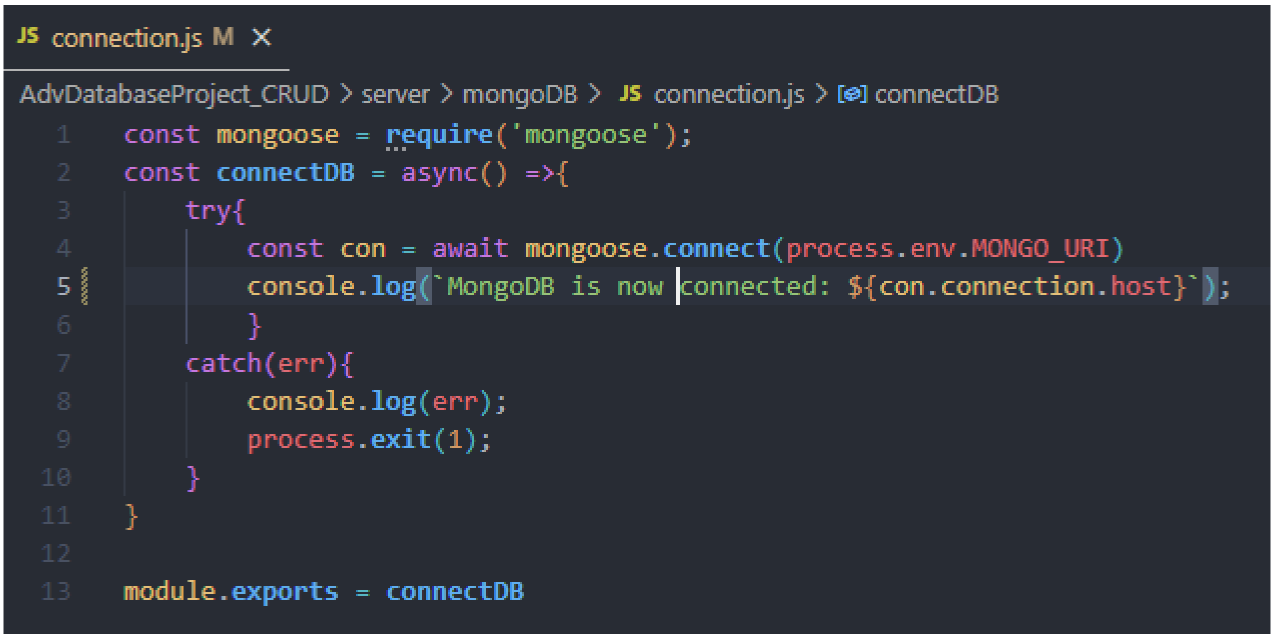

ALBERTO'S ORG - 2022-02-28 > ADVDATABASEASSESSMENT

Network Access

You will only be able to connect to your cluster from the follo

IP Address
0.0.0.0/0 (includes your current IP address)

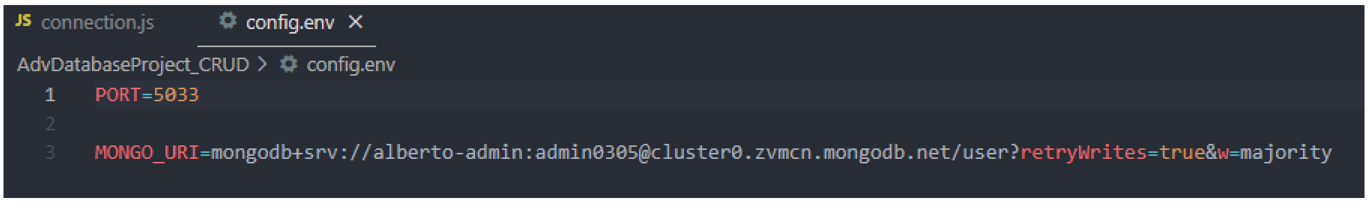
Following through the next step to connecting to MongoDB, from Visual Studios, is to write up the code. The dependencies that is required to access MongoDB is "Mongoose". Below you can see an image of this code where we have successfully connected to MongoDB.


connection.js M
AdvDatabaseProject_CRUD > server > mongoDB > connection.js > connectDB

```
1  const mongoose = require('mongoose');
2  const connectDB = async() =>{
3      try{
4          const con = await mongoose.connect(process.env.MONGO_URI)
5          console.log(`MongoDB is now connected: ${con.connection.host}`);
6          }
7      catch(err){
8          console.log(err);
9          process.exit(1);
10     }
11 }
12
13 module.exports = connectDB
```

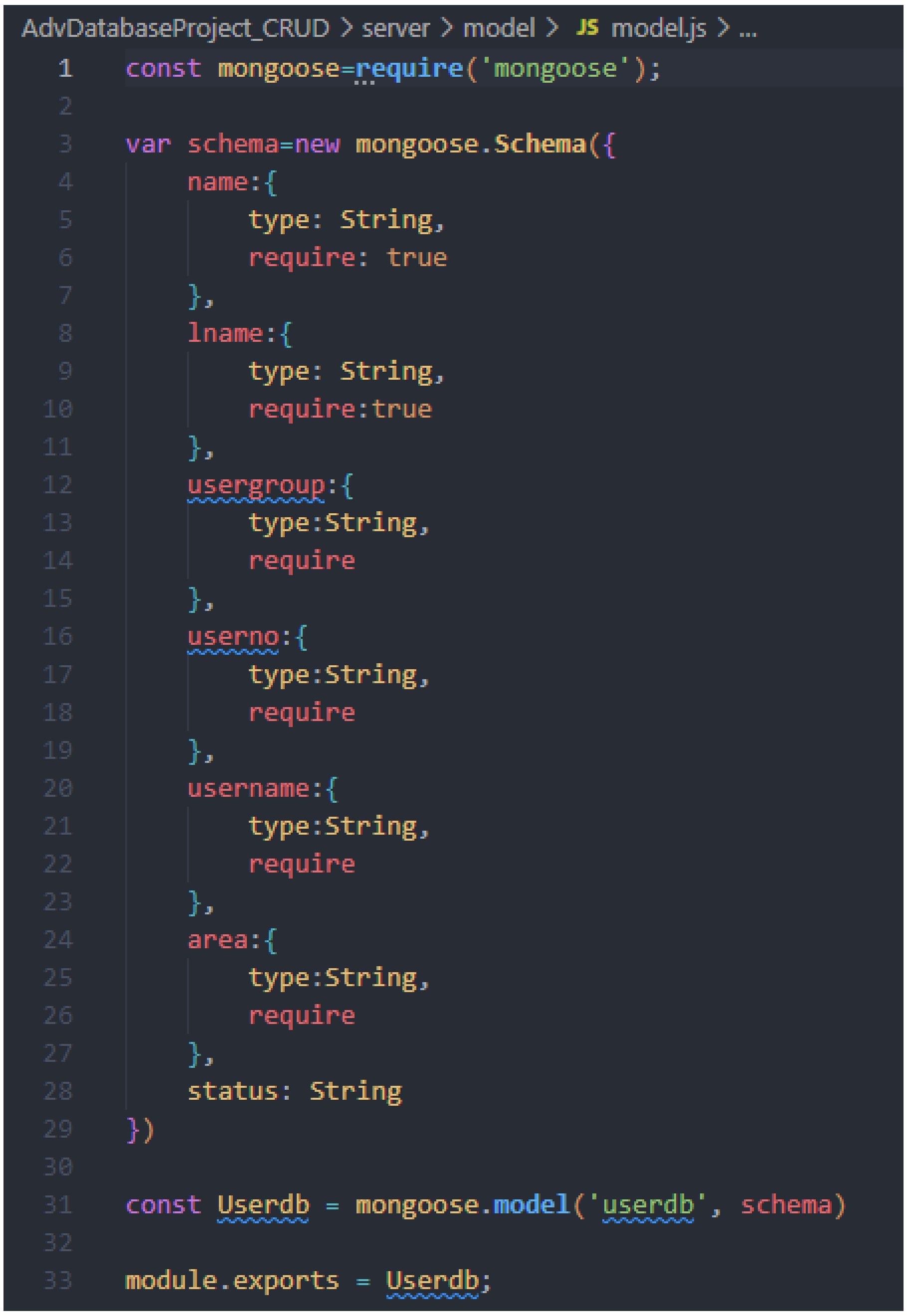
One of the key aspects I always try to incorporate is the log in of information. Line 5 on the code just does that. Once we have successfully connected and found this db it will inform you on the terminal, and if it fails this will also acknowledge in line 8.

Looking at Line 4 of the code, you will see that we are directing this to an env file. Using a dotenv file is a form of security where you would store secret information that is for and stored on your computer. In this case this is where I would hold my private admin name and password for my eyes only. If this was to be a group project, then each individual would require to have their own personal dotenv file due to the sensitivity. (Line 3)



```
1  PORT=5033
2
3  MONGO_URI=mongodb+srv://alberto-admin:admin0305@cluster0.zvmcn.mongodb.net/user?retryWrites=true&w=majority
```

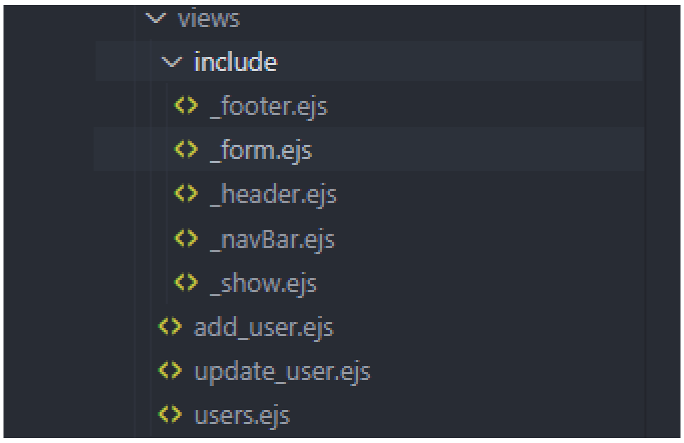
With the MongoDB now connected, the thought process behind the data is how do I wish to present this data. As this plays a vital point for the model. As we will be creating, reading and updating this model. As well as MongoDB is a nonrelational type of database, which mean there are no tables and connectors involved between them like your standard database. the model for MongoDB is called a schema.



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

var schema=new mongoose.Schema({
    name:{
        type: String,
        require: true
    },
    lname:{
        type: String,
        require:true
    },
    usergroup:{
        type:String,
        require
    },
    userno:{
        type:String,
        require
    },
    username:{
        type:String,
        require
    },
    area:{
        type:String,
        require
    },
    status: String
})

const Userdb = mongoose.model('userdb', schema)

module.exports = Userdb;
```

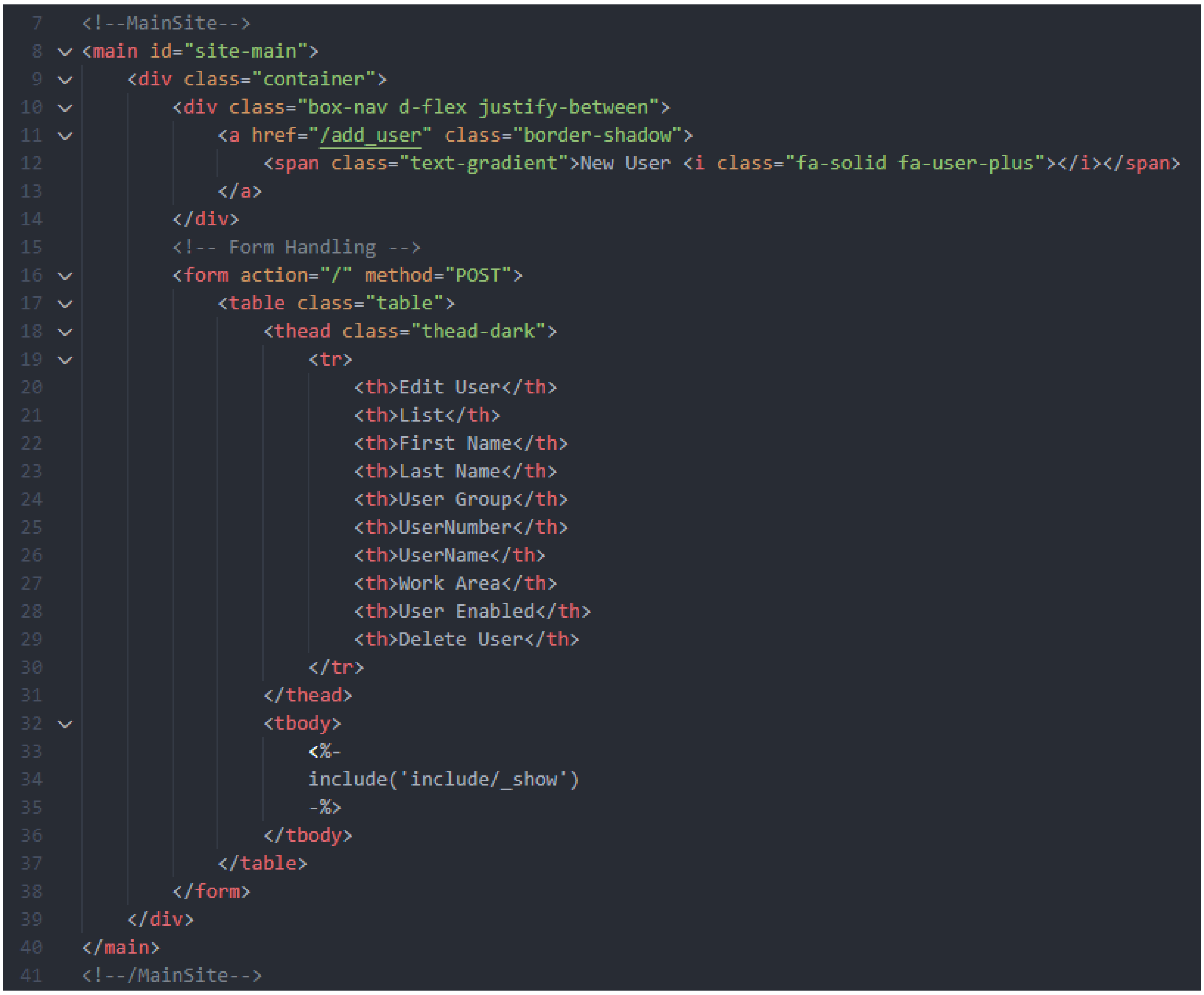
Views

Views is where we store what the user will see, the html files. Instead we are using ejs files as with ejs as with ejs its a simple tool that allows us to markup html format with Javascript to make it more usable or interactive.



```
views
  include
    _footer.ejs
    _form.ejs
    _header.ejs
    _navBar.ejs
    _show.ejs
  add_user.ejs
  update_user.ejs
  users.ejs
```

The starting home page is the table list of all the staffing within the work area.



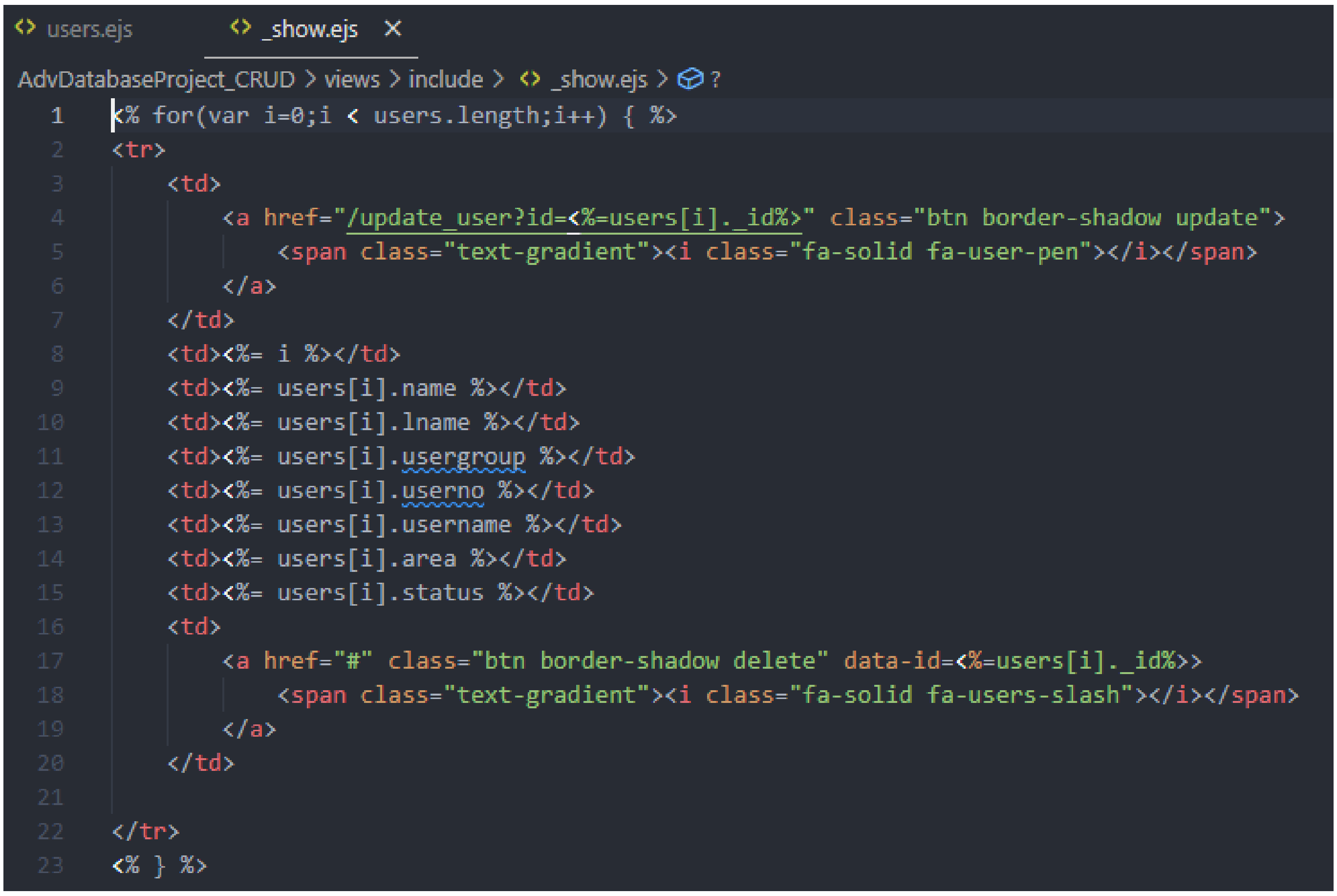
```html
<!--MainSite-->
<main id="site-main">
    <div class="container">
        <div class="box-nav d-flex justify-between">
            <a href="/add_user" class="border-shadow">
                <span class="text-gradient">New User <i class="fa-solid fa-user-plus"></i></span>
            </a>
        </div>
        <!-- Form Handling -->
        <form action="/" method="POST">
            <table class="table">
                <thead class="thead-dark">
                    <tr>
                        <th>Edit User</th>
                        <th>List</th>
                        <th>First Name</th>
                        <th>Last Name</th>
                        <th>User Group</th>
                        <th>UserNumber</th>
                        <th>UserName</th>
                        <th>Work Area</th>
                        <th>User Enabled</th>
                        <th>Delete User</th>
                    </tr>
                </thead>
                <tbody>
                    <%-
                    include('include/_show')
                    -%>
                </tbody>
            </table>
        </form>
    </div>
</main>
<!--/MainSite-->
```

The primary requirement for the front page was the table design, nothing to complicated to structure out, follow up by the design to make the user interface more presentable and easier on the eye.

As the primary reason why we used ejs, I used javascript to pull in the data that we have stored in MongoDB. This can be seen in Lines 33 - 35.

One of the new methods I discovered during my research was how to manage and make my coding more cleaner and easier to manage. Rather than having a whole page of coding where finding a section I want to edit will make it more difficult, we can simplify the page and remove sections and display what we need.

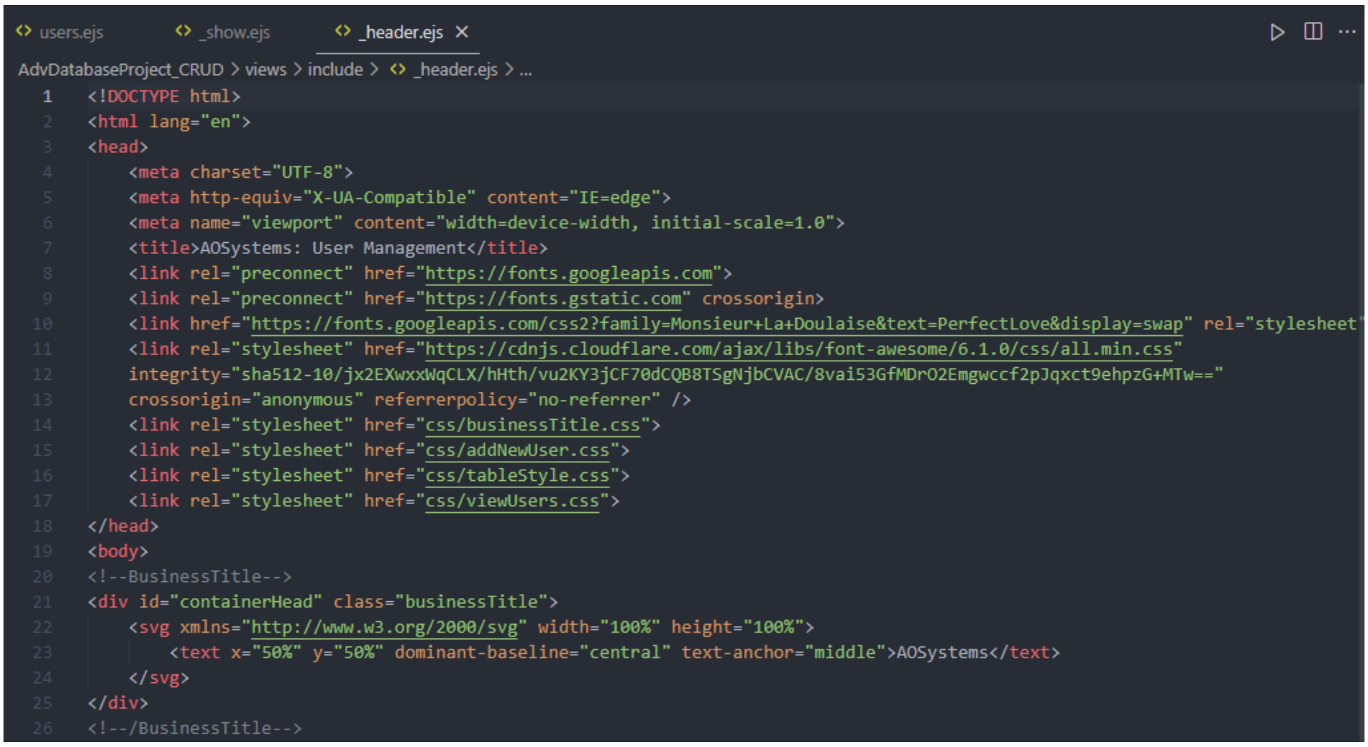
If we look at where the javascript code is saved for Lines 33 - 35;



```
<% for(var i=0;i < users.length;i++) { %>
<tr>
    <td>
        <a href="/update_user?id=<%=users[i]._id%>" class="btn border-shadow update">
            <span class="text-gradient"><i class="fa-solid fa-user-pen"></i></span>
        </a>
    </td>
    <td><%= i %></td>
    <td><%= users[i].name %></td>
    <td><%= users[i].lname %></td>
    <td><%= users[i].usergroup %></td>
    <td><%= users[i].userno %></td>
    <td><%= users[i].username %></td>
    <td><%= users[i].area %></td>
    <td><%= users[i].status %></td>
    <td>
        <a href="#" class="btn border-shadow delete" data-id=<%=users[i]._id%>>
            <span class="text-gradient"><i class="fa-solid fa-users-slash"></i></span>
        </a>
    </td>

</tr>
<% } %>
```

Has we have this code embedded into the main homepage, this would look a lot messier. Now we can read it clearly when we wish to change or edit.

Another useful use for this separation was separating the header. The header will always need to be available in every page you create for you application/website. The header alone can have a long lists of links, as per the image below.

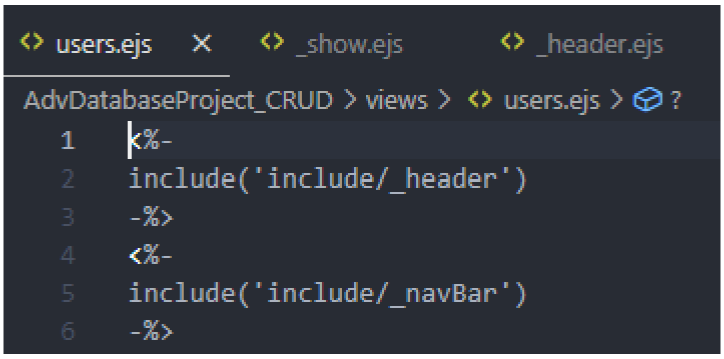


users.ejs _show.ejs _header.ejs

AdvDatabaseProject_CRUD > views > include > _header.ejs > ...

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>AOSystems: User Management</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Monsieur+La+Doulaise&text=PerfectLove&display=swap" rel="stylesheet"
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.1.0/css/all.min.css"
    integrity="sha512-10/jx2EXwxxWqCLX/hHth/vu2KY3jCF70dCQB8TSgNjbCVAC/8vai53GfMDrO2Emgwccf2pJqxct9ehpzG+MTw=="
    crossorigin="anonymous" referrerpolicy="no-referrer" />
    <link rel="stylesheet" href="css/businessTitle.css">
    <link rel="stylesheet" href="css/addNewUser.css">
    <link rel="stylesheet" href="css/tableStyle.css">
    <link rel="stylesheet" href="css/viewUsers.css">
</head>
<body>
<!--BusinessTitle-->
<div id="containerHead" class="businessTitle">
    <svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%">
        <text x="50%" y="50%" dominant-baseline="central" text-anchor="middle">AOSystems</text>
    </svg>
</div>
<!--/BusinessTitle-->
```

Now we can create a new page with a simple javascript header code. And review the code above on a separate area. Any editing of this header page will automatically be updating every page connected, rather than manually adapting them one by one.



users.ejs _show.ejs _header.ejs

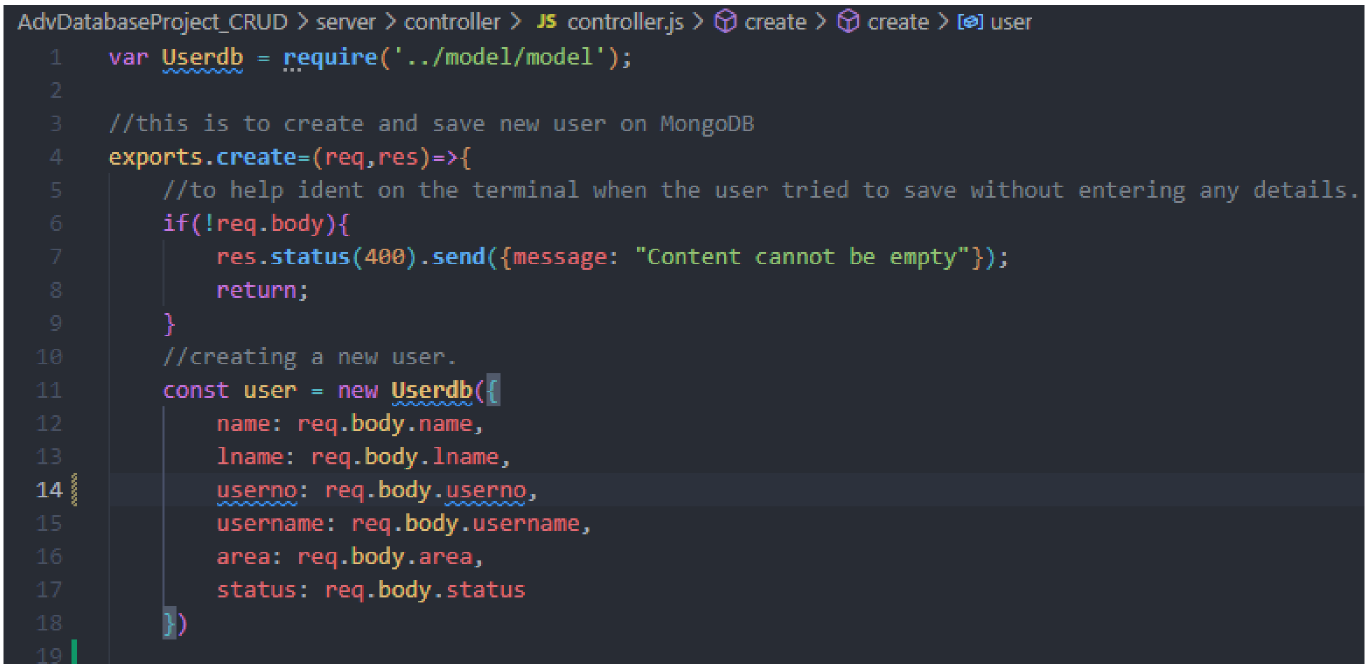
AdvDatabaseProject_CRUD > views > users.ejs > ?

```
<%-
include('include/_header')
-%>
<%-
include('include/_navBar')
-%>
```

Controller

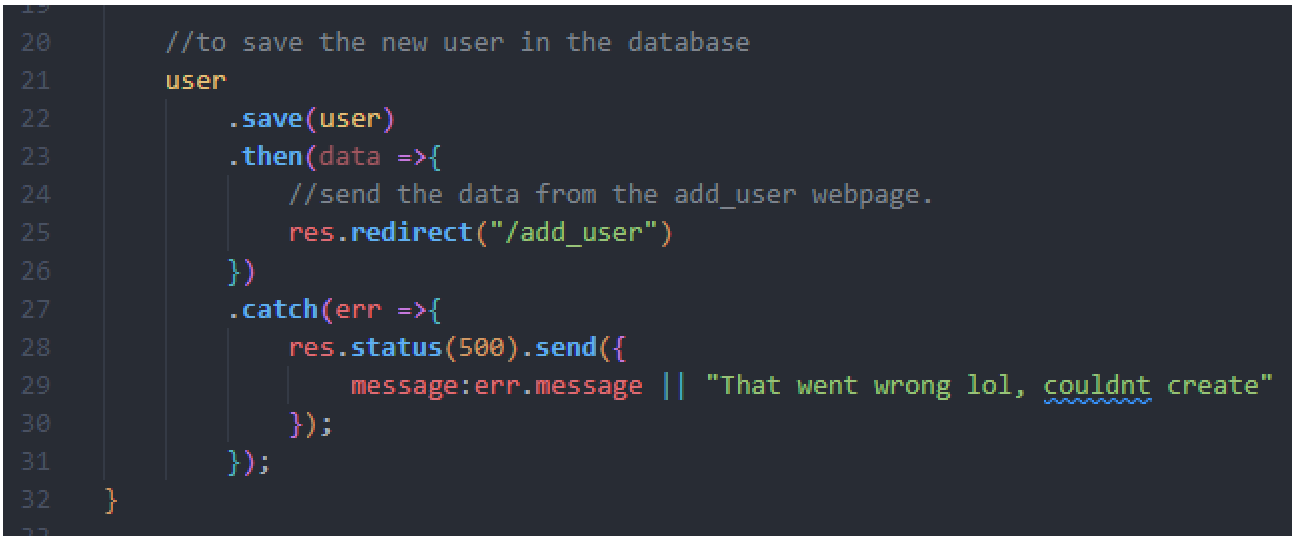
Finally we look at how all this is managed by the controller. The controller is responsible for processing incoming requests, performing the operation of the model. So in my system the controller deals with my whole CRUD application.#

To start the controller requires to bring forward the model that we are using from the model folder



```
AdvDatabaseProject_CRUD > server > controller > JS controller.js > create > create > user
 1  var Userdb = require('../model/model');
 2
 3  //this is to create and save new user on MongoDB
 4  exports.create=(req,res)=>{
 5      //to help ident on the terminal when the user tried to save without entering any details.
 6      if(!req.body){
 7          res.status(400).send({message: "Content cannot be empty"});
 8          return;
 9      }
10      //creating a new user.
11      const user = new Userdb({
12          name: req.body.name,
13          lname: req.body.lname,
14          userno: req.body.userno,
15          username: req.body.username,
16          area: req.body.area,
17          status: req.body.status
18      })
19
```

Line 1 defines that this page requires the information from the `model` from its directory.



```
20      //to save the new user in the database
21      user
22          .save(user)
23          .then(data =>{
24              //send the data from the add_user webpage.
25              res.redirect("/add_user")
26          })
27          .catch(err =>{
28              res.status(500).send({
29                  message:err.message || "That went wrong lol, couldnt create"
30              });
31          });
32  }
```

We can now look at creating, the code is based of the schema from the model and once the fields have been filled then