



# WEB701 ASSESSMENT 2

Hamish Drogemuller 13515109

## Contents

1. Web Frameworks .....	2
Purpose of Web Frameworks.....	2
Advantages and Disadvantages:.....	2
Common Framework Features.....	2
Architecture .....	2
Security .....	2
Responsive Design.....	3
2. Framework Comparison.....	6
Backend.....	6
Environment.....	6
Node.....	6
Express .....	6
MongoDB and Mongoose .....	6
Packages.....	6
NPM .....	6
Nodemon .....	6
DotENV.....	6
Concurrently.....	6
Bcrypt .....	7
Voucher-Code-Generator.....	7
Backend.....	8
Database Connection: .....	8
Server: .....	9
MVC:.....	10
Frontend:.....	13
Angular: .....	13
Svelte:.....	14
3. Recommendation of framework.....	15

# 1. Web Frameworks

## Purpose of Web Frameworks

Frameworks are utilised during the web development process to allow for easier building of websites and web applications. Frameworks provide many capabilities such as providing methods for the presentation of information, scripting environments, information pathways and data access. These are usually presented in the form of API's or Application Programming Interfaces.

## Advantages and Disadvantages:

One of the greatest advantages of frameworks is that they are often open source, have large communities and documentation bases and generally provide a wide range of features to assist in faster development. Open source frameworks are extremely efficient as they allow for the reutilisation of code snippets therefore eliminating the need to recode sections of the project. This reutilisation in turn speeds up the completion time of the project, meaning a client gets their application faster and at a lower cost, and the developer can move on to future work.

A major disadvantage to frameworks is that they can be both hard to learn if there is an underlying language the developer is not familiar with, as well as they can be very difficult to add to as they are limited in their scope. This is one of the many reasons it is good to become familiar with a variety of frameworks that can be applied to various projects.

## Common Framework Features

### Architecture

A vast majority of frameworks follow the industry standard architecture of Model-View-Controller also known as the MVC pattern. This pattern allows the developer to separate the logic of their application, providing a much cleaner architecture for any developer working on the project.

### Model

The model layer provides a location to store the business logic, guidelines and functions of the application. Upon receiving user input data from the controller layer the model layer is designed to process that data and tell what the updated interface should show.

### View

The view layer is what is most commonly referred to as the graphical representation of the application, Otherwise known as the Frontend. The view layer takes the user input which is sent to the controller layer which updates itself dependent upon the model layers instruction.

### Controller

This layer is commonly called the translation layer. It takes input data and works as a middleman for the model and view layers. Notifying both layers of any changes that may be required.

## Security

### Input Validation

Input Validation refers to the process of analysing user input to ensure that it is "clean" i.e not malicious and works for the websites expectations of what the input data should be. Input validation is a great process to use for password length, username length and for defining checkbox items. For

this project I will be utilising input validation for signup and login, ensuring the correct data is used and providing an easy way to initiate errors.

#### *Password Hashing*

Password hashing is used to secure user credentials within the database. Hashing takes place with the utilisation of a hashing algorithm that stores a unique encrypted password that cannot be recovered without the hashing algorithm. This process ensures that even if a malicious actor is able to recover our users passwords they won't be usable.

There are many algorithms that can be utilised to hash a password however after analysing the options I have chosen to utilise bcrypt.

#### *Responsive Design*

The majority of frameworks provide a responsive framework that assists in the creation of websites that are more accessible, intuitive and flow better between browsers and browsing devices.

In practice this means that developers can easily tailor how a site will be displayed depending on differing screen sizes. For this project we wish to develop a website that is responsive and one that is accessible across a range of devices. Particular desktop and mobile.

Requirements	MEAN Stack	MESN Stack	Differences
Registration	<pre> &lt;div&gt;   &lt;div class="row"&gt;     &lt;div class="col-md-3 fw-normal"&gt;Please login&lt;/div&gt;      &lt;div class="form-floating"&gt;       &lt;input type="email" class="form-control" id="floatingInput" placeholder="Name"&gt;       &lt;label for="floatingInput"&gt;First Name&lt;/label&gt;     &lt;/div&gt;      &lt;div class="form-floating"&gt;       &lt;input type="text" class="form-control" id="floatingInput" placeholder="Last Name"&gt;       &lt;label for="floatingInput"&gt;Last Name&lt;/label&gt;     &lt;/div&gt;      &lt;div class="form-floating"&gt;       &lt;input type="email" class="form-control" id="floatingInput" placeholder="Email"&gt;       &lt;label for="floatingInput"&gt;Email Address&lt;/label&gt;     &lt;/div&gt;      &lt;div class="form-floating"&gt;       &lt;input type="password" class="form-control" id="floatingInput" placeholder="Password"&gt;       &lt;label for="floatingInput"&gt;Password&lt;/label&gt;     &lt;/div&gt;      &lt;div class="form-floating"&gt;       &lt;input type="password" class="form-control" id="floatingInput" placeholder="Confirm Password"&gt;       &lt;label for="floatingInput"&gt;Confirm Password&lt;/label&gt;     &lt;/div&gt;      &lt;div class="form-floating"&gt;       &lt;input type="checkbox" class="form-control" id="floatingInput" placeholder="Remember Me"&gt;       &lt;label for="floatingInput"&gt;Remember Me&lt;/label&gt;     &lt;/div&gt;      &lt;div class="form-floating"&gt;       &lt;input type="button" class="form-control" id="floatingInput" placeholder="Sign In"&gt;       &lt;label for="floatingInput"&gt;Sign In&lt;/label&gt;     &lt;/div&gt;   &lt;/div&gt; &lt;/div&gt; </pre>	<pre> &lt;script&gt;   import { Link, useNavigate } from "svelte-navigator";   import { Form, FormGroup, FormText, Input, Label } from "sveltestrap";   import axios from "axios";    // login states   let emailAddress = "";   let password = "";   let error = "";   const navigate = useNavigate();    function handleSubmit() {     const config = {       headers: { "Access-Control-Allow-Origin": "*" },     };     axios       .post(         "http://localhost:3001/user/login/",         { emailAddress, password },         config       )       .then((res) =&gt; {         console.log(res);         localStorage.setItem("userInfo", JSON.stringify(res.data));         setTimeout(() =&gt; {           window.location.href = "/Product";         }, 2000);       })       .catch((err) =&gt; {         error = err.response.data;       });   } &lt;/script&gt; </pre>	There aren't many relevant differences within the code snippets for each of these projects for Registration. The main difference is the method in which events are handled. MESN uses the bind method whereas MEAN uses event handlers.
Login	<pre> &lt;div class="form-floating"&gt;   &lt;input type="email" class="form-control" id="floatingInput" placeholder="Name"&gt;   &lt;label for="floatingInput"&gt;Email Address&lt;/label&gt; &lt;/div&gt;  &lt;div class="form-floating"&gt;   &lt;input type="password" class="form-control" id="floatingInput" placeholder="Password"&gt;   &lt;label for="floatingInput"&gt;Password&lt;/label&gt; &lt;/div&gt;  &lt;div class="checkbox"&gt;   &lt;input type="checkbox" value="remember-me" /&gt; Remember me &lt;/div&gt;  &lt;div class="form-floating"&gt;   &lt;input type="button" class="form-control" id="floatingInput" placeholder="Sign In"&gt;   &lt;label for="floatingInput"&gt;Sign In&lt;/label&gt; &lt;/div&gt; </pre>	<pre> &lt;script&gt;   import { Link, useNavigate } from "svelte-navigator";   import { Form, FormGroup, FormText, Input, Label } from "sveltestrap";   import axios from "axios";    // login states   let emailAddress = "";   let password = "";   let error = "";   const navigate = useNavigate();    function handleSubmit() {     const config = {       headers: { "Access-Control-Allow-Origin": "*" },     };     axios       .post(         "http://localhost:3001/user/login/",         { emailAddress, password },         config       )       .then((res) =&gt; {         console.log(res);         localStorage.setItem("userInfo", JSON.stringify(res.data));         setTimeout(() =&gt; {           window.location.href = "/Product";         }, 2000);       })       .catch((err) =&gt; {         error = err.response.data;       });   } &lt;/script&gt; </pre>	No key differences within this code however in my opinion SVELTE provides a much cleaner and easy to follow code snippet.
Administration	<pre> //edit user details const editUser = async (req, res, next) =&gt; {   const { emailaddress } = req.body;   if (!emailaddress) {     return res.status(400).json({       message: "User does not exist"     });   }   const user = await User.findOne({ emailaddress });   if (!user) {     return res.status(400).json({       message: "User does not exist"     });   }   const { firstName, lastName, username, password, isMember, isBeneficiary } = req.body;   if (firstName) {     user.firstName = firstName;   }   if (lastName) {     user.lastName = lastName;   }   if (username) {     user.username = username;   }   if (password) {     user.password = password;   }   if (isMember) {     user.isMember = isMember;   }   if (isBeneficiary) {     user.isBeneficiary = isBeneficiary;   }   await user.save();   res.status(200).json({     success: true,     data: user,     message: "User details updated successfully"   }); } </pre>	<pre> &lt;script&gt;   import { Link, useNavigate } from "svelte-navigator";   import { Form, FormGroup, FormText, Input, Label } from "sveltestrap";   import axios from "axios";    const navigate = useNavigate();   const userInfo = JSON.parse(localStorage.getItem('userInfo'));    //AccountEdit states   let firstName = userInfo.data.firstName;   let lastName = userInfo.data.lastName;   let username = userInfo.data.username;   let emailAddress = userInfo.data.emailAddress;   let password = userInfo.data.password;    function handleEditSubmit () {     const data = {       firstName: firstName,       lastName: lastName,       username: username,       emailAddress: emailAddress,       password: password,     };     axios       .put('http://localhost:3001/user/editUser/', data)       .then((res) =&gt; {         localStorage.setItem('userInfo', JSON.stringify(res.data));         setTimeout(() =&gt; {           window.location.href = "/Account";         }, 2000);       })       .catch((err) =&gt; {         console.log(err);       });   } &lt;/script&gt; </pre>	I found the administration methods to be achieved very similarly as they both utilise standard js.

Product Registration	<pre>// add product to database const addProduct = async (req, res, next) =&gt; {   try {     const product = await products.create(req.body);     res.status(201).json({       success: true,       data: product,       message: 'Product added'     });   } catch (error) {     res.status(500).json({       message: 'Invalid product data'     });   } }</pre>	<pre>const handleAddProductSubmit = () =&gt; {   if (     name === ""        description === ""        voucherPrice === ""        imageUrl === ""        category === ""   ) {     setError("Please fill in all the fields");     setLoading(false);   } else {     axios       .post("http://localhost:5001/product/addProduct", {         name,         description,         voucherPrice,         imageUrl,         category,         memberId: userInfo.data._id,         claimedStatus: false,       })       .then((res) =&gt; {         setTimeout(() =&gt; {           window.location.href = "/Product";         }, 2000);       })       .catch((err) =&gt; {         setLoading(false);         setError(err.response.data.message);       });   } };</pre>	Similar to the account administration the add product was very simple, although
Acceptance of tokens	<pre>const voucher_codes = require('voucher-code-generator'); // TODO: Test all routes STATUS:WORKING  const Orders = require("../model/orders"); const Users = require("../model/users");  const createToken = () =&gt; {   const token = voucher_codes.generate({     length: 12,     count: 1,     pattern: '###-###-###-###',   })[0];    return token; }</pre>	<pre>&lt;div class="container"&gt;   &lt;div class="row"&gt;     &lt;div class="col-md-8 offset-md-2"&gt;       &lt;div class="text-center"&gt;Products listed&lt;/div&gt;       &lt;table class="table table-striped"&gt;         &lt;thead&gt;           &lt;tr&gt;             &lt;th scope="col"&gt;ProductID&lt;/th&gt;             &lt;th scope="col"&gt;Name&lt;/th&gt;             &lt;th scope="col"&gt;Voucher Price&lt;/th&gt;             &lt;th scope="col"&gt;Product Claimed&lt;/th&gt;           &lt;/tr&gt;         &lt;/thead&gt;         &lt;tbody&gt;           &lt;tr&gt;             &lt;td&gt;               &lt;div&gt;products&lt;/div&gt;               &lt;div&gt;                 &lt;div&gt;products on product (product.id)&lt;/div&gt;                 &lt;tr&gt;                   &lt;td&gt;product_id&lt;/td&gt;                   &lt;td&gt;product_name&lt;/td&gt;                   &lt;td&gt;product.voucherPrice&lt;/td&gt;                   &lt;td&gt;product.claimedStatus / "Yes" / "No"&lt;/td&gt;                   &lt;td&gt;                     &lt;button variant="primary" onclick=updateClaimedStatus&gt;                       Update Claim/Status                     &lt;/button&gt;                   &lt;/td&gt;                 &lt;/tr&gt;               &lt;/td&gt;             &lt;/tr&gt;           &lt;/tbody&gt;         &lt;/table&gt;       &lt;/div&gt;     &lt;/div&gt;   &lt;/div&gt;   &lt;div class="text-center"&gt;     &lt;div&gt;Products&lt;/div&gt;   &lt;/div&gt; &lt;/div&gt;</pre>	Both of these approaches utilise the same approach but use slightly different methods. Where Svelte uses each block Angular uses the .map method
Database Integration	<pre>&lt;script&gt; import axios from "axios"; import { onMount } from "svelte"; import OrderDetailsModal from "../components/OrderDetailsModal.svelte";  const userInfo = JSON.parse(localStorage.getItem("userInfo")); let tokens = [];  // get user email address from database to update user axios   .post("http://localhost:5001/user/", { email: userInfo.email })   .then((res) =&gt; {     localStorage.setItem("userInfo", JSON.stringify(res));   })   .catch((err) =&gt; {     console.log(err);   });  onMount(async () =&gt; {   tokens = userInfo.data.tokens; }); &lt;/script&gt;</pre>	<pre>&lt;script&gt; import axios from "axios"; import { onMount } from "svelte"; import OrderDetailsModal from "../components/OrderDetailsModal.svelte";  const userInfo = JSON.parse(localStorage.getItem("userInfo")); let tokens = [];  // get user email address from database to update user axios   .post("http://localhost:5001/user/", { email: userInfo.email })   .then((res) =&gt; {     localStorage.setItem("userInfo", JSON.stringify(res));   })   .catch((err) =&gt; {     console.log(err);   });  onMount(async () =&gt; {   tokens = userInfo.data.tokens; }); &lt;/script&gt;</pre>	The axios approach is identical across both frameworks, the library allows us to use api calls to pull product and user information from our mongoose database

Working examples were shown in a online demonstration.

## 2. Framework Comparison

After researching a variety of frameworks including a MERN stack, a FastAPI stack and a Django among many others. I decided to branch out and compare a M.E.A.N(Mongo,Express,Angular and Node) Stack and a M.E.S.N(Mongo,Express, Svelte and Node) stack. I chose these two frameworks as I wanted to branch out and analyse how difficult it would be to learn a relatively new framework that is outside of my “Comfort Zone” as a developer.

### Backend

The backend systems of both of these Frameworks consist of MongoDB, ExpressJS and NodeJS. These technologies make for great framework options as they provide exceptional compatibility when integrating with a vast array of differing Front end options. Becoming proficient with these technologies allows for the use of a large variety of code libraries that assists in the development of future projects.

### Environment

#### Node

Node provides us with our backend environment suited for our application, node utilises the JavaScript V8 Engine. Node is a great option for creating scalable network applications.

#### Express

Express is used for the creation of handlers for requests, HTTP and URL paths, also known as routes. It also helps facilitate the creation of application middleware such as port connections and response rendering.

#### MongoDB and Mongoose

MongoDB provides us with the capabilities to develop the Data Layer of our application. We will utilise Mongoose to act as the middleman between our MongoDB and Express.

### Packages

Alongside the application environment we will be utilising a variety of packages to develop our Quill Training application into what is needed.

#### NPM

NPM is the key package for the usage of NodeJS and the management of further packages that we will utilise with Node. NPM boasts one of, if not the, largest online databases for packages known as the NPM registry.

#### Nodemon

Nodemon is a great tool that assists in the development of node based applications, Nodemon will automatically redeploy the application upon saving. Allowing the developer to skip having to close and redeploy the application manually to see edits.

#### DotENV

DotENV is utilised to load environment variables. This allows developers to store environment variables separate to the code.

#### Concurrently

Concurrently is a straightforward package that assist developers by providing functionality to run multiple terminal commands at the same time.

## Bcrypt

As touched on earlier, Bcrypt is what we will be leveraging to hash our user passwords. Improving the security of our application.

## Voucher-Code-Generator

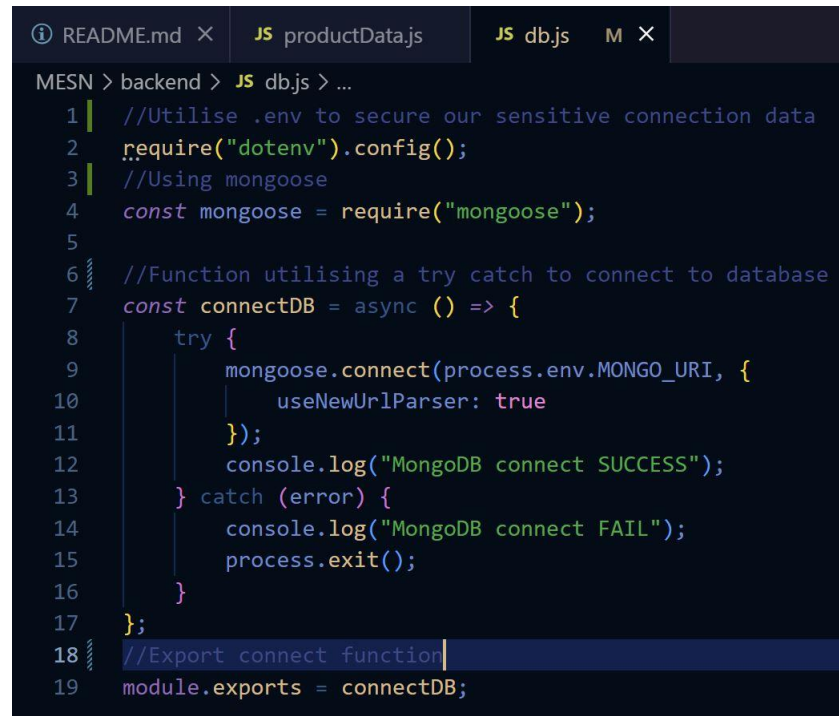
After some experience with JWT I wanted to branch out and discover some alternate options, to this end I decided to utilise a package known as Voucher-Code-Generator that will provide a similar functionality as JWT.



## Backend

### Database Connection:

Our application MongoDB database will be access using our db.js, this code utilises the connection string found in our earlier env file.



```
MESN > backend > JS db.js > ...
1 //Utilise .env to secure our sensitive connection data
2 require("dotenv").config();
3 //Using mongoose
4 const mongoose = require("mongoose");
5
6 //Function utilising a try catch to connect to database
7 const connectDB = async () => {
8   try {
9     mongoose.connect(process.env.MONGO_URI, {
10       useNewUrlParser: true
11     });
12     console.log("MongoDB connect SUCCESS");
13   } catch (error) {
14     console.log("MongoDB connect FAIL");
15     process.exit();
16   }
17 };
18 //Export connect function
19 module.exports = connectDB;
```

Server:

Within the backend of the application is our server javascript file. It utilises express.js to create an instance that can be accessed with app.listen.

A screenshot of a code editor with a dark theme. The editor has several tabs at the top: 'README.md', 'JS productData.js', 'JS db.js', 'M', 'JS server.js' (which is active), and a close button 'X'. The main area shows the content of 'server.js'. The code starts with a terminal prompt 'MESN > backend > JS server.js > ...'. The code itself is a JavaScript file that sets up an Express.js server. It includes imports for 'dotenv', 'express', 'mongoose', 'cors', and an 'error' module. It configures the environment, creates an Express app, sets up CORS, and connects to a database. It then defines several route modules (userRoutes, productRoutes, cartRoutes, orderRoutes, categoryRoutes) and uses them with the Express app. Finally, it sets up a GET endpoint for '/' that responds with 'API is running' and listens on port 5001.

```
1 require("dotenv").config({path: '../.env'});
2 const express = require('express');
3 const mongoose = require('mongoose');
4 const app = express();
5 const connectDB = require('./db')
6 const { notFound, errorHandler } = require('./middlewares/error.middleware')
7 var cors = require('cors')
8
9 app.use(cors())
10
11 // creates express application
12 app.use(express.json());
13
14 // database connect
15 connectDB();
16
17 /* ----- get routes ----- */
18 const userRoutes = require('./routes/userRoutes')
19 const productRoutes = require('./routes/productRoutes')
20 const cartRoutes = require('./routes/cartRoutes')
21 const orderRoutes = require('./routes/orderRoutes')
22 const categoryRoutes = require('./routes/categoryRoutes')
23
24 /* ----- routes ----- */
25 //list of user
26 app.use("/user", userRoutes)
27 //list of product
28 app.use("/product", productRoutes)
29 //list of cart
30 app.use("/cart", cartRoutes)
31 //list of order
32 app.use("/order", orderRoutes);
33 //list of category
34 app.use("/category", categoryRoutes);
35
36 //middleware
37 app.use(notFound);
38 app.use(errorHandler);
39
40 // request and send to check the backend is running
41 app.get('/', (req, res) => {
42   res.send("API is running");
43 })
44 // Use port 5001 and display backend information
45 app.listen(5001, console.log("Server started on PORT 5001"));
46
47
```

MVC:

*Model:*

As touched on earlier, Models are essential for providing information regarding the storing of data within the database.

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

// defines the Users schema
const userSchema = new mongoose.Schema({
  firstName: {
    type: String,
    required: [true, "First name is required"],
  },
  lastName: {
    type: String,
    required: [true, "Last name is required"],
  },
  userName: {
    type: String,
    required: [true, "User name is required"],
  },
  emailAddress: {
    trim: true,
    type: String,
    required: [true, "Email address is required"],
    unique: true,
  },
  password: {
    type: String,
    required: [true, "Password is required"],
    minLength: [8, "Password must be at least 8 characters long"],
  },
  isMember: {
    type: Boolean,
    required: [true, "Is this user a member?"],
    default: false,
  },
  isBeneficiary: {
    type: Boolean,
    required: [true, "Is this user a beneficiary?"],
    default: false,
  },
  vouchers: {
    type: Number,
    required: [false, "Vouchers"],
  },
  tokens: {
    type: [String],
    required: [false, "Tokens"],
    default: undefined,
  }
},{
  timestamp:true,//this will check when the user is created and updated
```

### Controller:

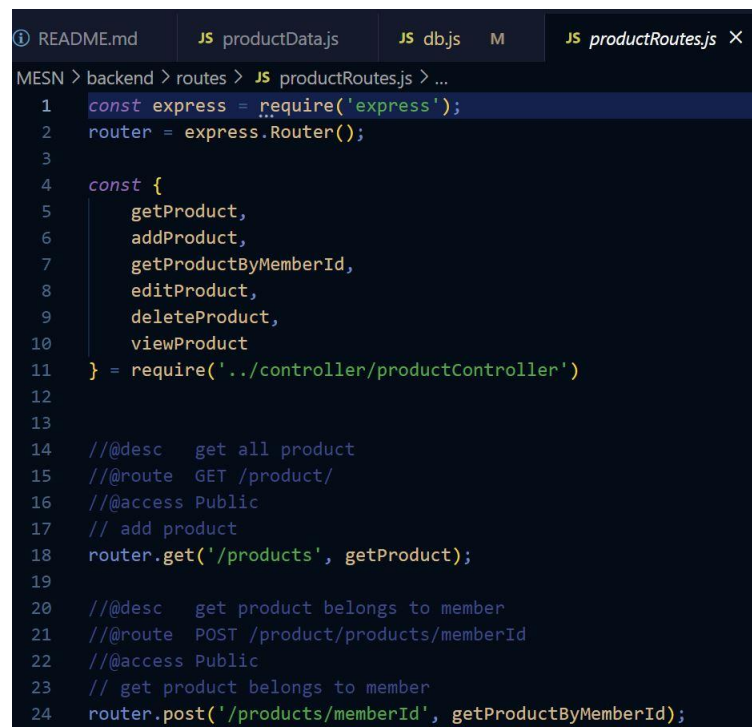
Our controller files will receive our user inputs and then instruct our Model and view files what to do.

```
//edit user details
const editUser = async (req, res, next) => {
  const { emailAddress } = req.body;
  try {
    const user = await Users.findOne({ emailAddress: emailAddress });
    if (!user) {
      return res.status(400).json({
        message: 'User does not exist'
      });
    }
    const { firstName, lastName, userName, password, isMember, isBeneficiary } = req.body;
    if (firstName) {
      user.firstName = firstName;
    }
    if (lastName) {
      user.lastName = lastName;
    }
    if (userName) {
      user.userName = userName;
    }
    if (password) {
      user.password = password;
    }
    if (isMember) {
      user.isMember = isMember;
    }
    if (isBeneficiary) {
      user.isBeneficiary = isBeneficiary;
    }
    await user.save();
    res.status(200).json({
      success: true,
      data: user,
      message: 'User details updated successfully'
    });
  } catch (error) {
    res.status(500).json({
      message: 'Invalid user data'
    });
  }
}
```

### Routes:

Routes represent a map through our system, they are what determines the path a request data is sent down.

HTTP requests are made by functions prompted by user input, this data will then return response data to the application frontend.



```
MESN > backend > routes > JS productRoutes.js > ...
1  const express = require('express');
2  router = express.Router();
3
4  const {
5    getProduct,
6    addProduct,
7    getProductByMemberId,
8    editProduct,
9    deleteProduct,
10   viewProduct
11 } = require('../controller/productController')
12
13
14 // @desc  get all product
15 // @route  GET /product/
16 // @access Public
17 // add product
18 router.get('/products', getProduct);
19
20 // @desc  get product belongs to member
21 // @route  POST /product/products/memberId
22 // @access Public
23 // get product belongs to member
24 router.post('/products/memberId', getProductByMemberId);
```

## Frontend:

For the frontend of this application we have compared Angular and Svelte. Both of these options provide great features for the development of a frontend. Below is the signup component for each stack, a comparison will be made after.

## Angular:

```
Go to component
<body class="text-center">

  <main class="form-signin">
    <form>
      <h1 class="h3 mb-3 fw-normal">Signup</h1>

      <div class="form-floating">
        <input type="first-name" class="form-control" id="floatingInput" placeholder="FirstName">
        <label for="floatingInput">First Name</label>
      </div>
      <div class="form-floating">
        <input type="last-name" class="form-control" id="floatingInput" placeholder="LastName">
        <label for="floatingInput">Last Name</label>
      </div>
      <div class="form-floating">
        <input type="email" class="form-control" id="floatingInput" placeholder="name@example.com">
        <label for="floatingInput">Email address</label>
      </div>
      <div class="form-floating">
        <input type="password" class="form-control" id="floatingPassword" placeholder="Password">
        <label for="floatingPassword">Password</label>
      </div>
      <div class="form-floating">
        <input type="confirm-password" class="form-control" id="floatingPassword" placeholder="ConfirmPassword">
        <label for="floatingPassword">Confirm Password</label>
      </div>

      <button class="w-100 btn btn-lg btn-primary" type="submit">Register</button>
      <a class="have-account text-muted" routerLink="../login">Already have an account?</a>
    </form>
  </main>
</body>
```

## Svelte:

```
<script>
import { Link, useNavigate } from "svelte-navigator";
import { Form, FormGroup, FormText, Input, Label } from "sveltetrapp";
import axios from "axios";

//signup states
let firstName = "";
let lastName = "";
let userName = "";
let emailAddress = "";
let password = "";
let isMember = false;
let isBeneficiary = false;
let error = "";

const navigate = useNavigate();

function handleSubmit() {
  const config = {
    headers: { "Access-Control-Allow-Origin": "*" },
  };
  if (isMember && isBeneficiary) {
    setError("Please select only one of the checkbox");
  }
  if (!isMember && !isBeneficiary) {
    setError("Please select at least one of the checkbox");
  }
  if (isMember && !isBeneficiary) {
    axios
      .post("http://localhost:5001/user/register", {
        firstName,
        lastName,
        userName,
        emailAddress,
        password,
        isMember: true,
        isBeneficiary: false,
      }, config)
      .then(() => {
        navigate("/");
      })
      .catch((error) => {
        console.log(error);
      });
  }
}
```

Svelte, Similarly to Angular (And by extension, React and Vue) utilises the idea of components as reusable aspects of our code. Svelte's Syntax is very similar to standard HTML and JavaScript. This lead to a much shallower learning curve when learning Svelte. The one big difference I found is that svelte is primarily a UI framework compared to Angular which could be described as more of a full application framework.

### 3. Recommendation of framework

Both frameworks covered are relatively similar. Offering similar functionality as well as being open source which gives them a massive bonus in terms of community and information available. Due to both backends being comprised of the same technology (MongoDB, Express and Node) I will detail the differences of the frontend technology.

Angular was released in 2010 as Angular JS 1 and has since gone through many changes since then to become the Angular 12 we utilise today. Some reasons that may draw developers to Angular is that it has Google's long term support which means it will be supported for many years to come, meaning once its learnt you have a great technology that you can upskill on for the future. Alongside these are;

- Stability
- Scalability
- Large community
- Provides ready made components
- Has a wide range of tech industry backers

Svelte is a newer framework than angular, having been released in November of 2016. Svelte focuses on allowing developers to create applications with less code, Svelte's main draw is that instead of utilising large, complex libraries it will run code through a compiler that in turn emits smaller, faster more optimized code bundles. This factor greatly reduces the strain of the application on systems. Some other reasons for using Svelte are;

- Simplicity
- A more User Interface focused framework
- "Out of the box" Animations and style

After comparing both frameworks and their pros and cons I have decided that Svelte alongside the other backend components will be utilised for the final application. I chose this because of Svelte's minimal styling and fast compiling.