# GymFlow

Gym class management system

# Application Overview

GymFlow is gym management application that allows for Staff members to create add-to, edit and remove customers/classes/Instructors, it also allows for viewing of classes via day, time with the added ability to see the number of students in the class as well as the instructor of said class.

# Slide Overview

- Login Functionality
- Interacting code files
- Login route explanation
- Security Considerations
- Challenges
- Solution
- Solution Implementation
- (Application Demo Video)

# Code Chosen Login Functionality: loginRoute & userModel js files

```
import dotenv from 'dotenv'
import express from 'express'
import jwt from 'jsonwebtoken'
// import { login } from '../controllers/authController.js';
// import { createInstructor } from '../controllers/createInstructorController.js';
// import { createInstructorValidationRules, validate } from '../middleware/loginValidation'; // Ensure path is correct
import { loginLimiter, createInstructorLimiter } from '../middleware/loginRateLimiter.js';
import { User } from '../models/userModel.js';

const loginRoutes = express.Router();

// Route for logging in with rate limiting
loginRoutes.post('/', loginLimiter, async (req, res) => {
    const { email, password } = req.body;

    try {
        const user = await User.findOne({ email }); // Find the user by email
        if (!user) {
            return res.status(400).json({ message: 'Invalid email or password' }); // User not found
        }

        const isMatch = await user.comparePassword(password); // Compare passwords

        if (!isMatch) {
            return res.status(400).json({ message: 'Invalid email or password' }); // Password does not match
        }

        const token = jwt.sign(
            { id: user._id, email: user.email, master: user.master }, // Payload
            process.env.JWT_SECRET,
            { expiresIn: '1h' } // Token expiration
        );

        res.json({ token }); // Return the token
    } catch (error) {
        console.error('Login error:', error);
        res.status(500).json({ message: 'Server error' }); // Handle errors
    }
})

// // Route for creating a new instructor with validation, rate limiting
// router.post('/instructors',
//     createInstructorValidationRules(), // Applies validation rules
//     validate, // Middleware to handle validation errors
//     createInstructorLimiter, // Applies rate limiting
//     createInstructor // Handles the request
// );

export default loginRoutes;
```

```
const userSchema = new mongoose.Schema({
    email: {
        type: String,
        required: true,
        unique: true,
        match: [/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/, 'Please enter a valid email address']
    },
    password: {
        type: String,
        required: true
    },
    master: {
        type: Boolean,
        required: true,
        default: false
    }
});

// hash pword before saving
userSchema.pre('save', async function(next) {
    if (!this.isModified('password')) return next();

    try {
        const salt = await bcrypt.genSalt(10);
        this.password = await bcrypt.hash(this.password, salt);
        next();
    } catch (error) {
        next(error);
    }
});

// compares the hashed pwords
userSchema.methods.comparePassword = async function(candidatePassword) {
    return await bcrypt.compare(candidatePassword, this.password);
};

const User = mongoose.model('User', userSchema);

export { User };
```

# Network Request

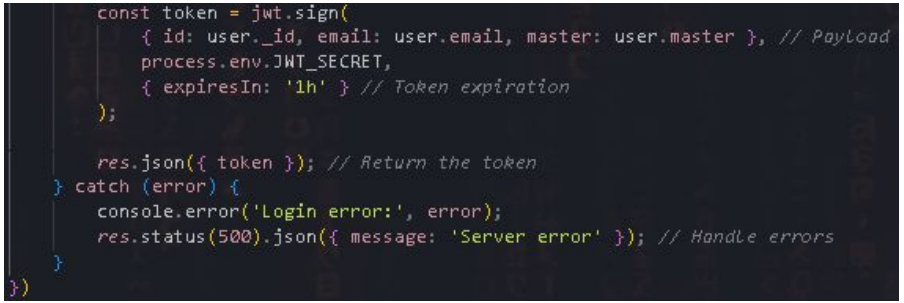
The login route (‘loginRoutes.post(‘/ ’) ’) handles POST requests to the ‘/login’ endpoint.
Customer will send a network request via this endpoint where their email/password are in the request body
Any validation, authorization and authentication information required will also be sent in the body.

```
loginRoutes.post('/', loginLimiter, async (req, res) => {
    const { email, password } = req.body;

    try {
        const user = await User.findOne({ email }); // Find the user by email
        if (!user) {
            return res.status(400).json({ message: 'Invalid email or password' }); // User not found
        }

        const isMatch = await user.comparePassword(password); // Compare passwords

        if (!isMatch) {
            return res.status(400).json({ message: 'Invalid email or password' }); // Password does not match
        }
```



```
        const token = jwt.sign(
            { id: user._id, email: user.email, master: user.master }, // Payload
            process.env.JWT_SECRET,
            { expiresIn: '1h' } // Token expiration
        );

        res.json({ token }); // Return the token
    } catch (error) {
        console.error('Login error:', error);
        res.status(500).json({ message: 'Server error' }); // Handle errors
    }
})
```

# Database Operation

- Find user:

```
try {
    const user = await User.findOne({ email }); // Find the user by email
```

This will query the database for a user email that matches the email passed in the request body.

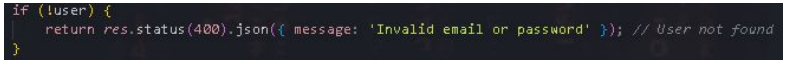
- Comparison of passwords:

```
const isMatch = await user.comparePassword(password); // Compare passwords
```

This will compare the password provided by user and the hashed password stored in the database.

# Conditional Statements

- Checking if the User exists:



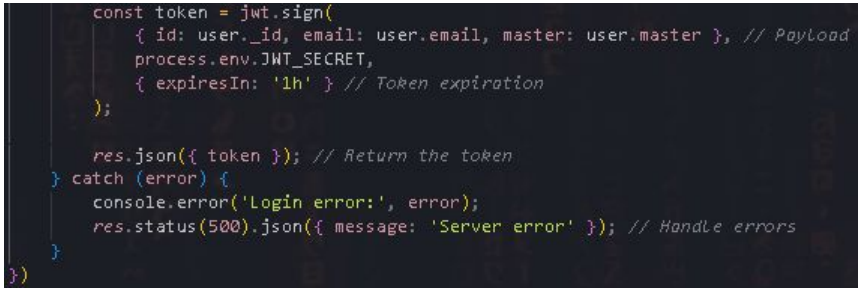

If the user is not found in the database a 400 status error message will be returned (“Invalid email or password”).

- Checking if Password matches:


If the password does not match once again a 400 status error message will be returned (“Invalid email or password”).

# JWT Token Generation

Creation of JWT token for Authentication:



```
        const token = jwt.sign(
            { id: user._id, email: user.email, master: user.master }, // Payload
            process.env.JWT_SECRET,
            { expiresIn: '1h' } // Token expiration
        );

        res.json({ token }); // Return the token
    } catch (error) {
        console.error('Login error:', error);
        res.status(500).json({ message: 'Server error' }); // Handle errors
    }
})
```

If Login succeeds a JWT token containing the user ID, email and if required, master status will be made. The token is then sent back to the customer for authentication required in other requests (if master then allow access to all CRUD (create, read, update, delete) capabilities.

# Security Considerations: Rate Limiter

- ‘windowsMs’ Defines the time period (15 mins) which the time limit applies to.
- ‘max’ specifies the maximum number of requests (5 allowed).
- ‘message’ this is the response sent to the user when the rate limit is exceeded.

**Purpose**: If a user exceeds 5 login attempts within 15mins, they will be temporarily locked out. Mitigating the risks of automated login attacks.

```
// this help limit bruce force attacks on the login route endpoint
import rateLimit from 'express-rate-limit';

const loginLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 5, // limit each IP to 5 requests per window ms
    message: 'Too many login attempts from this IP, please try again later.'
});

const createInstructorLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 10, // limit each IP to 10 requests per window ms
    message: 'Too many requests from this IP, please try again later.'
});

export { loginLimiter, createInstructorLimiter };
```

# Security Considerations: Bcrpyt Hash

- ‘bcrypt.genSalt(10)’ will generate a salt(random number) with 10 rounds of encryption (each round requires computational time to calculate, each added round the computational time doubles).
- ‘bcrypt.hash(this.password, salt)’ Hashes the password using the generated salt.
- ‘comparePassword(candidatePassword)’ This line will compare the plaintext stored in the hash.

```
// hash pword before saving
userSchema.pre('save', async function(next) {
    if (!this.isModified('password')) return next();

    try {
        const salt = await bcrypt.genSalt(10);
        this.password = await bcrypt.hash(this.password, salt);
        next();
    } catch (error) {
        next(error);
    }
});

// compares the hashed pwords
userSchema.methods.comparePassword = async function(candidatePassword) {
    return await bcrypt.compare(candidatePassword, this.password);
};

const User = mongoose.model('User', userSchema);

export { User };
```

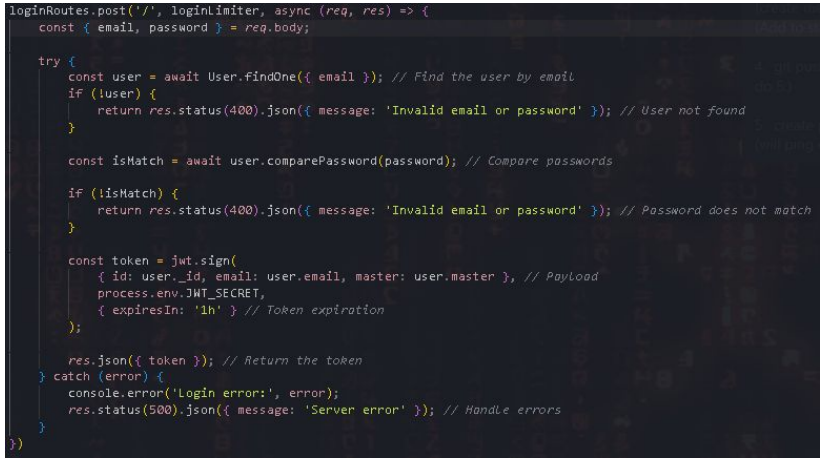
**Purpose:** Even if attackers manage to gain access to the database, they cannot easily retrieve user passwords saved within. As all they will see are hashed values.

```
const userSchema = new mongoose.Schema({
    email: {
        type: String,
        required: true,
        unique: true,
        match: [/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/, 'Please enter a valid email address']
    },
    password: {
        type: String,
        required: true
    },
    master: {
        type: Boolean,
        required: true,
        default: false
    }
});
```

# Security Considerations: JWT Token

- ‘jwt.sign({ id, email, master }, process.env.JWT_SECRET, {expires: ‘1hr’ }) will sign in the payload (user id, email and master status) with a secret key (Token) which is set to expire in 1 hour.
- The token is used for authenticating API requests and managing user sessions securely.

**Purpose:** In essence this token is a way to securely pass information between systems, ensuring that the data is only readable by those who have the secret key and that it is only valid for a certain amount of time.



```
loginRoutes.post('/', loginLimiter, async (req, res) => {
    const { email, password } = req.body;

    try {
        const user = await User.findOne({ email }); // Find the user by email
        if (!user) {
            return res.status(400).json({ message: 'Invalid email or password' }); // User not found
        }

        const isMatch = await user.comparePassword(password); // Compare passwords

        if (!isMatch) {
            return res.status(400).json({ message: 'Invalid email or password' }); // Password does not match
        }

        const token = jwt.sign(
            { id: user._id, email: user.email, master: user.master }, // Payload
            process.env.JWT_SECRET,
            { expiresIn: '1h' } // Token expiration
        );

        res.json({ token }); // Return the token
    } catch (error) {
        console.error('Login error:', error);
        res.status(500).json({ message: 'Server error' }); // Handle errors
    }
})
```

# Challenge encountered during the project

**Scenario:** We initially designed our system with two separate user schemas: one for Gym management personnel (Master) and one for Instructors. Both schemas included:

- **`email`** and **`password`** properties, to facilitate login and authentication
- **`isAdmin`** property (boolean) to indicate whether a user had administrative privileges to perform certain operation (e.g. deleting a customer or class from the system)

## Instructor schema (old)

```
const instructorSchema = new mongoose.Schema({
    firstName: { type: String, required: true },
    lastName: { type: String, required: true },
    age: { type: Number, required: true, min: 18 },
    email: { type: String, required: true, unique: true, match: [/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$
    phone: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    master: { type: Boolean, required: true},
    classes: [{ type: mongoose.ObjectId, ref: 'Class' }]
});
```

## User schema (old)

```
const masterSchema = new mongoose.Schema({
    email: { type: String, required: true, unique: true, match: [/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/,
    password: { type: String, required: true },
    master: { type: Boolean, required: true, default: false }
});
```

# Challenge encountered during the project

**Challenge:** During the coding process, we realised that this approach introduced unnecessary complexity.

- The **`master`** property in the Instructor schema created confusion around the scope of administrative privileges, especially since only the Master role was intended to have such privileges

Hence, the challenge was to redefine the user role management system to accurately reflect intended permissions which would also result in a cleaner codebase

# Challenge encountered during the project

**Solution:** removed the **`master`** property from the Instructor schema completely.

- By focusing administrative privileges exclusively on the Master (subsequently changed to User) schema, we could more clearly define and manage user roles within our application.

**Instructor schema (new)**

```
const instructorSchema = new mongoose.Schema({
    firstName: { type: String, required: true },
    lastName: { type: String, required: true },
    age: { type: Number, required: true, min: 18 },
    email: { type: String, required: true, unique: true, match: [/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/,
    phone: { type: String, required: true, unique: true },
    classes: [{ type: mongoose.ObjectId, ref: 'Class' }]
});
```

# Challenge encountered during the project

To implement the solution, we performed the following steps:

- **Schema update:** updated Mongoose schema to remove the **`master`** property from the Instructor schema. This included updating MongoDB collections and database-related code where necessary.
- **Role-based access control:** revised authorisation logic to ensure that only Users with **`master`** role had access to admin-only function. This included updating middleware and updating validation checks throughout codebase.
- **Code refactoring:** updated codebase to remove any code that had reliance on **`master`** property for Instructors
- **Testing**: tested the application to ensure updated authorisation rules were working as expected

# GymFlow Demo