CIS007-3 Comparative Integrated Systems – AY 24/25

Student Name: Ruchi Bajracharya Student Number: 2133112

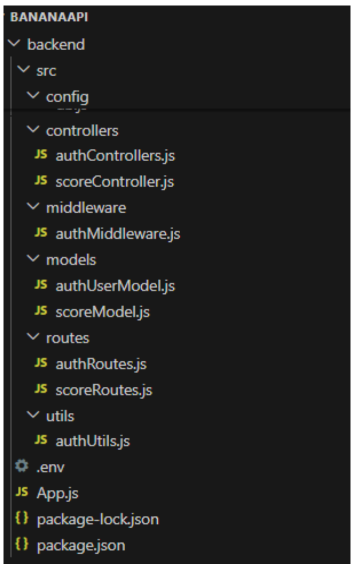
Student Signature: Ruchi Date: November 2024 (Week 8)

TABLE OF CONTENTS

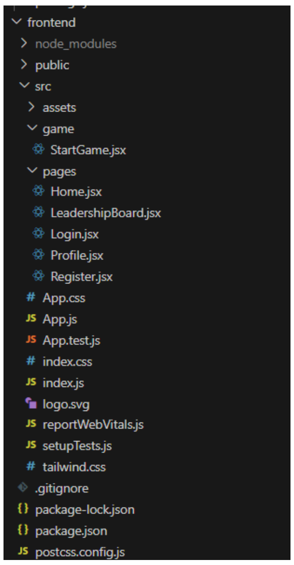
1. Software Architecture

The project follows a **modular component-based architecture**, where each component has a specific responsibility. The structure is designed to ensure separation of scalability, and maintainability.

a. Project Structure



```
BANANAAPI
∨ backend
 ∨ src
  ∨ config
  ∨ controllers
    authControllers.js
    scoreController.js
  ∨ middleware
    authMiddleware.js
  ∨ models
    authUserModel.js
    scoreModel.js
  ∨ routes
    authRoutes.js
    scoreRoutes.js
  ∨ utils
    authUtils.js
 .env
 App.js
 package-lock.json
 package.json
```

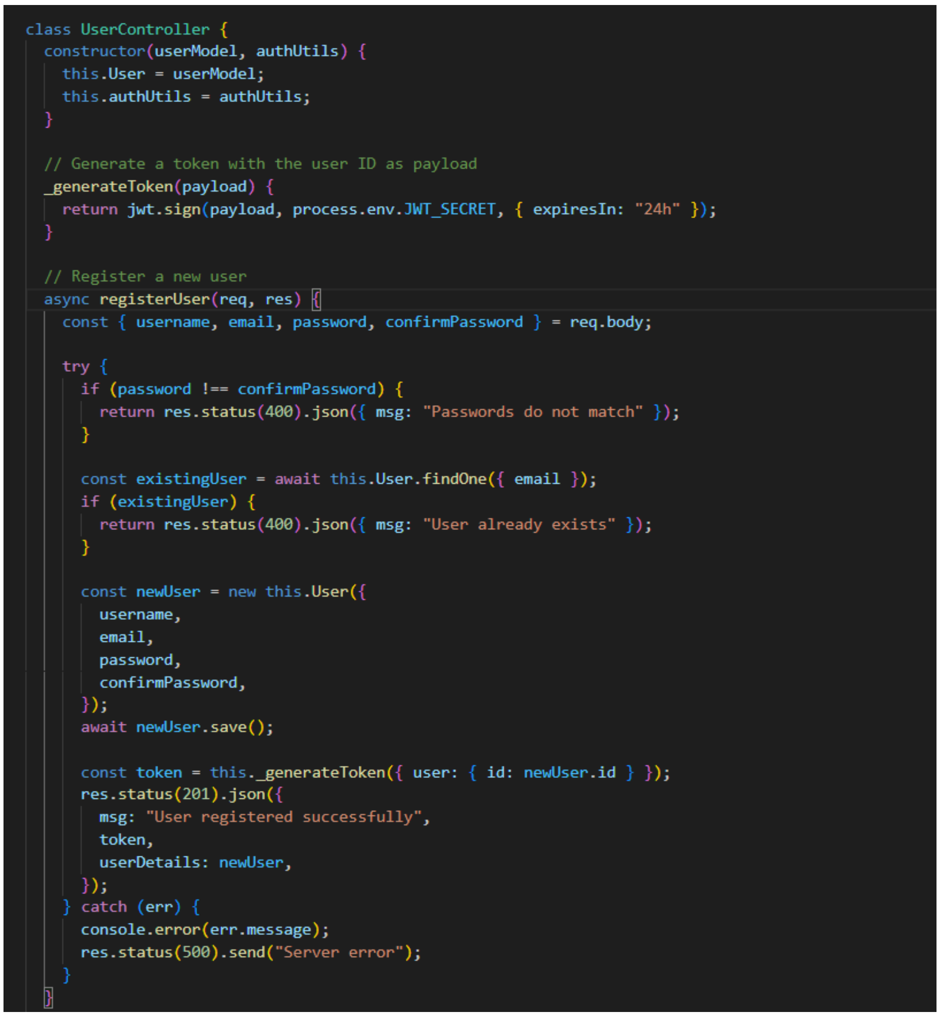


```
∨ frontend
 > node_modules
 > public
 ∨ src
  > assets
  ∨ game
    StartGame.jsx
  ∨ pages
    Home.jsx
    LeadershipBoard.jsx
    Login.jsx
    Profile.jsx
    Register.jsx
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
  reportWebVitals.js
  setupTests.js
  tailwind.css
 .gitignore
 package-lock.json
 package.json
 postcss.config.js
```

In this project, the code is organized into a modular component-based architecture, where each part of the project has a clear responsibility. The backend is split into folders like controllers, models, routes, and utils, each serving a specific role. For example, controllers handle the business logic, models define how data is structured in the database, and routes map user requests to the appropriate controller. This structure helps in organizing the project, making it scalable, and ensuring that different parts of the project are easy to maintain and update without affecting others.

On the frontend, the code is also organized into components like Home.jsx, Login.jsx, and Profile.jsx, which are separate files handling different pages. The App.js file links everything together, while styles are handled in separate CSS files. This approach helps with maintaining clean and organized code, allowing easy updates and improvements.

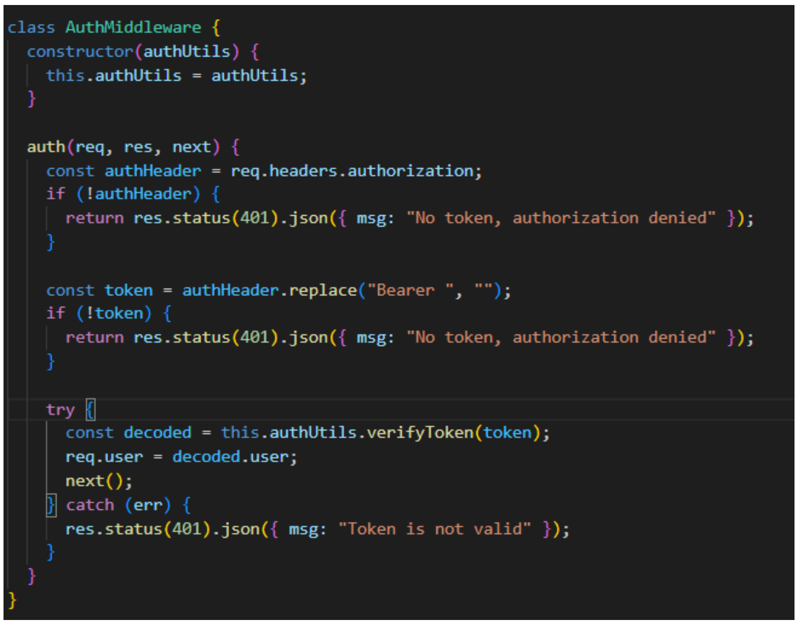
b. Core Components



```
class UserController {
  constructor(userModel, authUtils) {
    this.User = userModel;
    this.authUtils = authUtils;
  }

  // Generate a token with the user ID as payload
  _generateToken(payload) {
    return jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: "24h" });
  }

  // Register a new user
  async registerUser(req, res) {
    const { username, email, password, confirmPassword } = req.body;

    try {
      if (password !== confirmPassword) {
        return res.status(400).json({ msg: "Passwords do not match" });
      }

      const existingUser = await this.User.findOne({ email });
      if (existingUser) {
        return res.status(400).json({ msg: "User already exists" });
      }

      const newUser = new this.User({
        username,
        email,
        password,
        confirmPassword,
      });
      await newUser.save();

      const token = this._generateToken({ user: { id: newUser.id } });
      res.status(201).json({
        msg: "User registered successfully",
        token,
        userDetails: newUser,
      });
    } catch (err) {
      console.error(err.message);
      res.status(500).send("Server error");
    }
  }
```

The UserController class encapsulates all logic related to user registration and login. Methods like registerUser and _generateToken are part of this class, ensuring that user creation and token generation logic are managed in one place. This ensures that data (like user details) is only accessed or modified through methods provided by the class, offering a controlled way of interacting with the data.

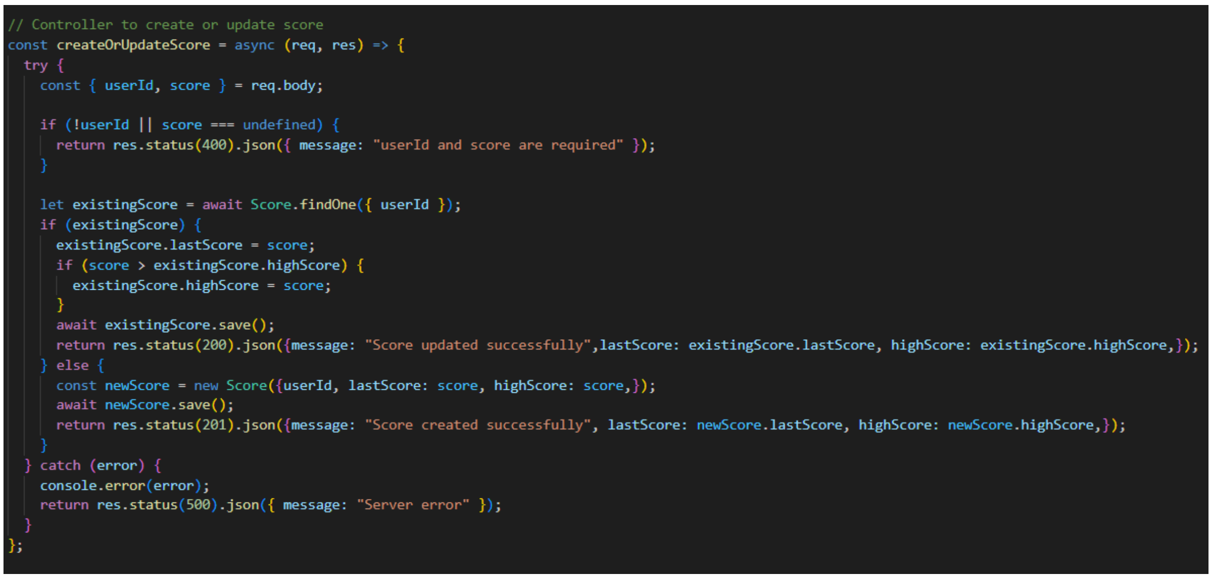
Abstraction is the concept of hiding the complex implementation details and showing only the necessary parts to the user. This helps in simplifying interactions with the system by reducing complexity.



```javascript
class AuthMiddleware {
  constructor(authUtils) {
    this.authUtils = authUtils;
  }

  auth(req, res, next) {
    const authHeader = req.headers.authorization;
    if (!authHeader) {
      return res.status(401).json({ msg: "No token, authorization denied" });
    }

    const token = authHeader.replace("Bearer ", "");
    if (!token) {
      return res.status(401).json({ msg: "No token, authorization denied" });
    }

    try {
      const decoded = this.authUtils.verifyToken(token);
      req.user = decoded.user;
      next();
    } catch (err) {
      res.status(401).json({ msg: "Token is not valid" });
    }
  }
}
```

The AuthMiddleware class abstracts the process of token validation. Other parts of the application don't need to know the internals of how the token is verified or how JWT works. They just use the auth method to ensure the request is authenticated. This hides the complexity of the token validation process and provides a simple interface for other components.

Modularity refers to the practice of breaking down a large system into smaller, independent modules that can be developed, tested, and maintained separately. Each module has a well-defined responsibility.

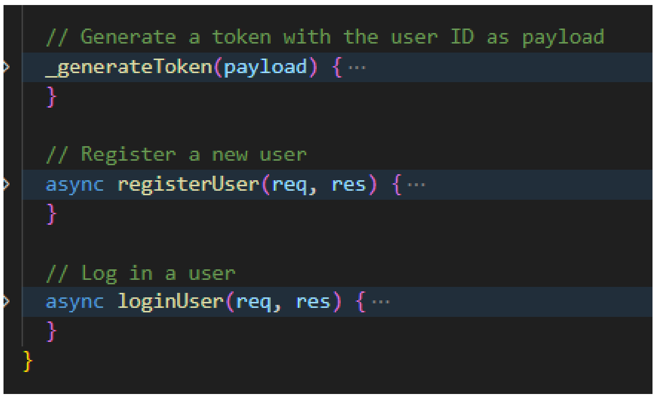


```javascript
// Controller to create or update score
const createOrUpdateScore = async (req, res) => {
  try {
    const { userId, score } = req.body;

    if (!userId || score === undefined) {
      return res.status(400).json({ message: "userId and score are required" });
    }

    let existingScore = await Score.findOne({ userId });
    if (existingScore) {
      existingScore.lastScore = score;
      if (score > existingScore.highScore) {
        existingScore.highScore = score;
      }
      await existingScore.save();
      return res.status(200).json({message: "Score updated successfully",lastScore: existingScore.lastScore, highScore: existingScore.highScore,});
    } else {
      const newScore = new Score({userId, lastScore: score, highScore: score,});
      await newScore.save();
      return res.status(201).json({message: "Score created successfully", lastScore: newScore.lastScore, highScore: newScore.highScore,});
    }
  } catch (error) {
    console.error(error);
    return res.status(500).json({ message: "Server error" });
  }
};
```

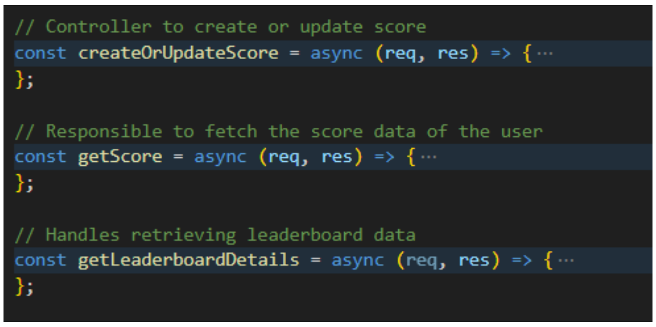
The scoreController.js file is modular and is solely responsible for managing scores. It has methods like createOrUpdateScore that handle the logic of creating or updating a score record, ensuring the logic is encapsulated in one module. This modularity makes it easy to manage score-related actions independently of other parts of the application, such as user management.

Single Responsibility Principle (SRP)

The **Single Responsibility Principle** states that a class should have only one reason to change, meaning it should only have one job or responsibility.



```
    // Generate a token with the user ID as payload
    _generateToken(payload) {···
    }

    // Register a new user
    async registerUser(req, res) {···
    }

    // Log in a user
    async loginUser(req, res) {···
    }
}
```



```
// Controller to create or update score
const createOrUpdateScore = async (req, res) => {···
};

// Responsible to fetch the score data of the user
const getScore = async (req, res) => {···
};

// Handles retrieving leaderboard data
const getLeaderboardDetails = async (req, res) => {···
};
```

c. Libraries and Packages

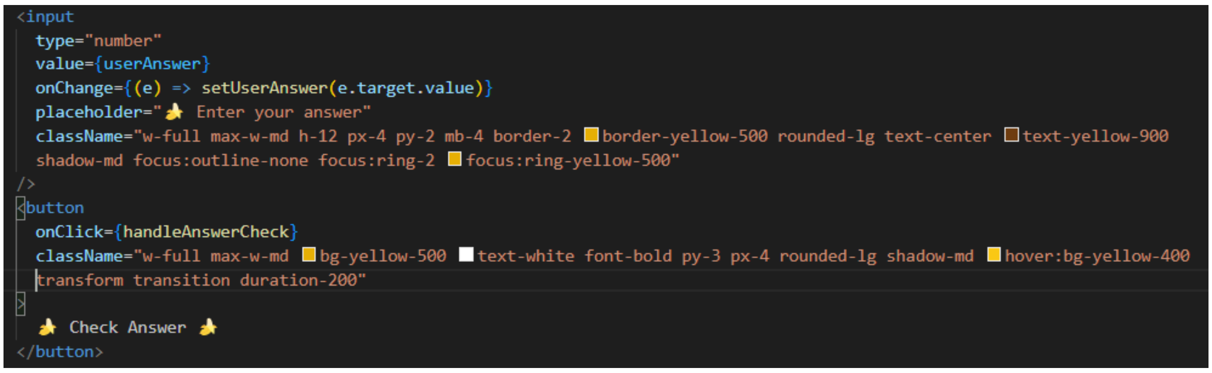
The application uses the following libraries and packages:

- **Express**: A web framework used for routing and handling HTTP requests.
- **Mongoose**: A MongoDB Object Data Modeling (ODM) library that simplifies interaction with the database.
- **JWT (jsonwebtoken)**: Used for creating and verifying JSON Web Tokens for secure user authentication.
- **Bcryptjs**: A library for hashing passwords securely.
- **dotenv**: Used for loading environment variables from a .env file for secure configuration management.

2. Event Driven Architectures

In this app, event-driven programming is used to handle various user actions that dictate the flow of the game.

a. Handling User Input with onChange and onClick:

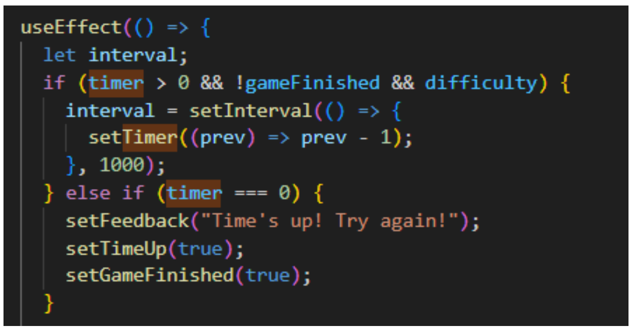


```
<input
  type="number"
  value={userAnswer}
  onChange={(e) => setUserAnswer(e.target.value)}
  placeholder="🍌 Enter your answer"
  className="w-full max-w-md h-12 px-4 py-2 mb-4 border-2 border-yellow-500 rounded-lg text-center text-yellow-900
  shadow-md focus:outline-none focus:ring-2 focus:ring-yellow-500"
/>
<button
  onClick={handleAnswerCheck}
  className="w-full max-w-md bg-yellow-500 text-white font-bold py-3 px-4 rounded-lg shadow-md hover:bg-yellow-400
  transform transition duration-200"
>
  🍌 Check Answer 🍌
</button>
```

This event handler (onChange) updates the state (userAnswer) whenever the user types in the input field. The component reacts to the input changes, and the state gets updated accordingly, which drives the game flow. When the user clicks the "Check Answer" button, the onClick event handler (handleAnswerCheck) is triggered. This function checks if the user's answer is correct and updates the feedback, score, and game status. The event-driven nature of this action is evident, as the game's state changes based on user interaction.

b. Game State Updates Based on Time and User Actions

The game uses several useEffect hooks to handle events such as the countdown timer, changing difficulty, and fetching questions from the API. These are all driven by the state and updated as events happen.

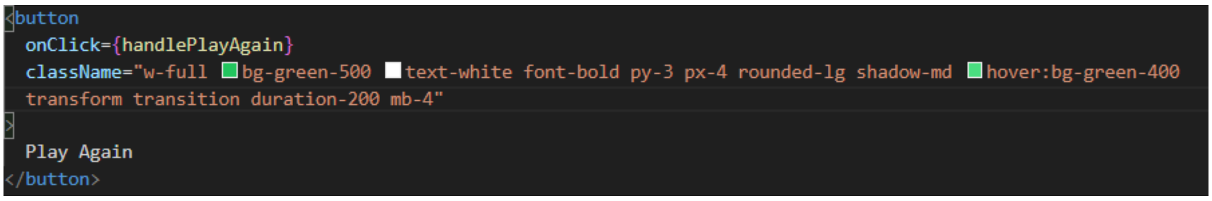


```
useEffect(() => {
  let interval;
  if (timer > 0 && !gameFinished && difficulty) {
    interval = setInterval(() => {
      setTimer((prev) => prev - 1);
    }, 1000);
  } else if (timer === 0) {
    setFeedback("Time's up! Try again!");
    setTimeUp(true);
    setGameFinished(true);
  }
```

This useEffect hook sets up an interval that decreases the timer every second, driving the game's countdown timer. The countdown is an event that triggers the game to end once the timer reaches 0, causing a feedback message ("Time's up! Try again!"), which alters the game state.

c. Handle Game Over and Retry with Onclick

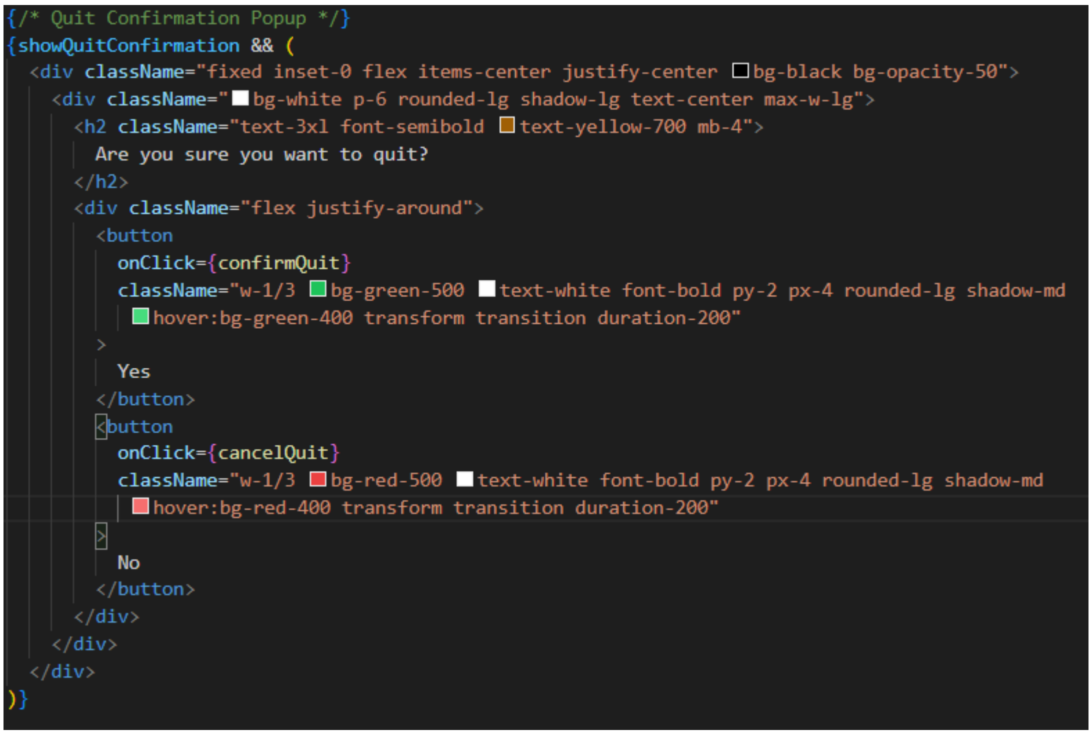
When the player answers the question or the timer runs out, they are presented with options to either retry or quit the game. These are handled by onClick events.



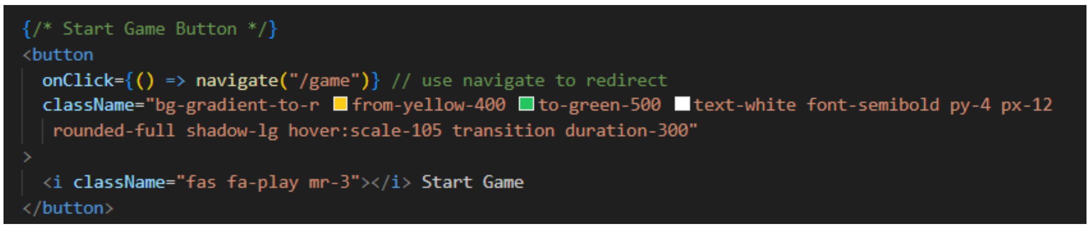
```
<button
  onClick={handlePlayAgain}
  className="w-full bg-green-500 text-white font-bold py-3 px-4 rounded-lg shadow-md hover:bg-green-400
  transform transition duration-200 mb-4"
>
  Play Again
</button>
```

Clicking the "Play Again" button triggers the handlePlayAgain function. This function resets the game state (timer, score, etc.), and fetches a new question. It's an event-driven process, responding to the user's desire to play again.

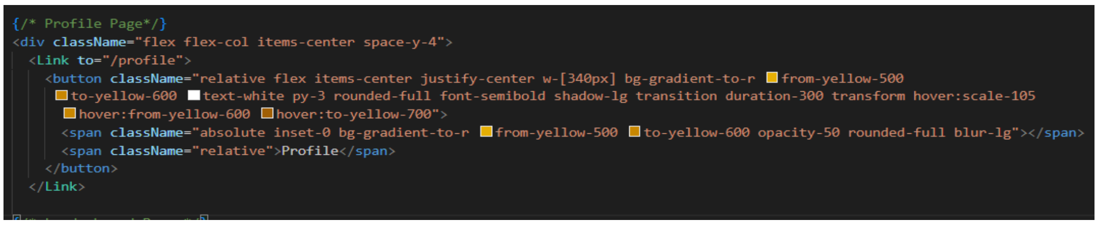
d. Some snippets of other event handlers used on the application:



```
{/* Quit Confirmation Popup */}
{showQuitConfirmation && (
  <div className="fixed inset-0 flex items-center justify-center bg-black bg-opacity-50">
    <div className="bg-white p-6 rounded-lg shadow-lg text-center max-w-lg">
      <h2 className="text-3xl font-semibold text-yellow-700 mb-4">
        Are you sure you want to quit?
      </h2>
      <div className="flex justify-around">
        <button
          onClick={confirmQuit}
          className="w-1/3 bg-green-500 text-white font-bold py-2 px-4 rounded-lg shadow-md
           hover:bg-green-400 transform transition duration-200"
        >
          Yes
        </button>
        <button
          onClick={cancelQuit}
          className="w-1/3 bg-red-500 text-white font-bold py-2 px-4 rounded-lg shadow-md
           hover:bg-red-400 transform transition duration-200"
        >
          No
        </button>
      </div>
    </div>
  </div>
)}
```



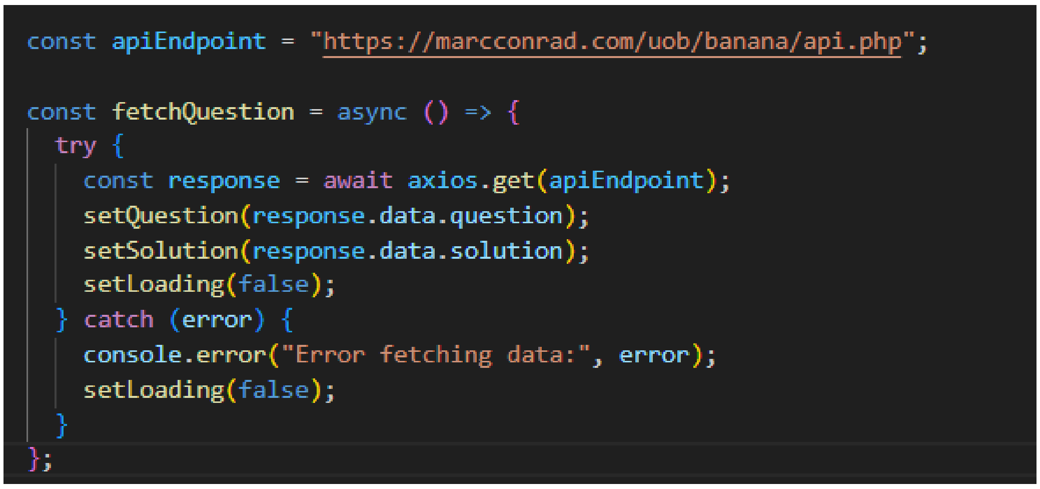
```
{/* Start Game Button */}
<button
  onClick={() => navigate("/game")} // use navigate to redirect
  className="bg-gradient-to-r from-yellow-400 to-green-500 text-white font-semibold py-4 px-12
   rounded-full shadow-lg hover:scale-105 transition duration-300"
>
  <i className="fas fa-play mr-3"></i> Start Game
</button>
```



```
{/* Profile Page*/}
<div className="flex flex-col items-center space-y-4">
  <Link to="/profile">
    <button className="relative flex items-center justify-center w-[340px] bg-gradient-to-r from-yellow-500
     to-yellow-600 text-white py-3 rounded-full font-semibold shadow-lg transition duration-300 transform hover:scale-105
      hover:from-yellow-600 hover:to-yellow-700">
      <span className="absolute inset-0 bg-gradient-to-r from-yellow-500 to-yellow-600 opacity-50 rounded-full blur-lg"></span>
      <span className="relative">Profile</span>
    </button>
  </Link>
```

3. Interoperability

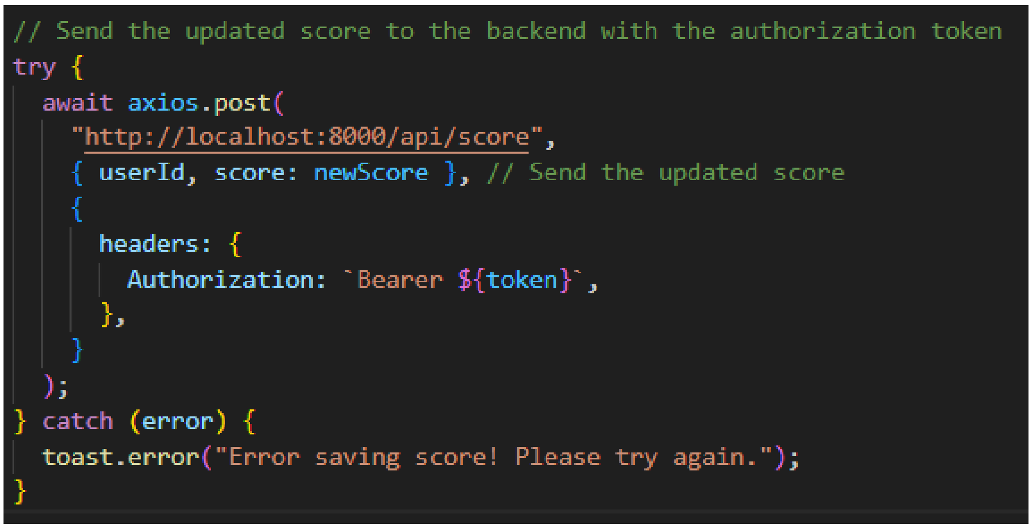
Interoperability is a key feature of this app, as it communicates with an external API to retrieve questions and solutions for the game. The app uses **Axios** to make **GET** requests to the API (https://marcconrad.com/uob/banana/api.php) over the **HTTPS protocol**. Additionally, after a correct answer is provided, the app sends updated score data to a backend server using **POST** requests with a **Bearer token** for authentication. This use of **RESTful APIs** and **HTTP** protocols allows the app to interact with external systems and manage game data effectively.



```
const apiEndpoint = "https://marcconrad.com/uob/banana/api.php";

const fetchQuestion = async () => {
  try {
    const response = await axios.get(apiEndpoint);
    setQuestion(response.data.question);
    setSolution(response.data.solution);
    setLoading(false);
  } catch (error) {
    console.error("Error fetching data:", error);
    setLoading(false);
  }
};
```

This API integration allows the game to **fetch dynamic content** (the question and solution) from an external source.

Additionally, when the player finishes a round, the game sends the score to the backend for storage:



```
// Send the updated score to the backend with the authorization token
try {
  await axios.post(
    "http://localhost:8000/api/score",
    { userId, score: newScore }, // Send the updated score
    {
      headers: {
        Authorization: `Bearer ${token}`,
      },
    }
  );
} catch (error) {
  toast.error("Error saving score! Please try again.");
}
```

This HTTP request sends the score to a backend server, ensuring that the player's progress is saved. The game also uses **authentication tokens** to verify the user's identity before sending this data.

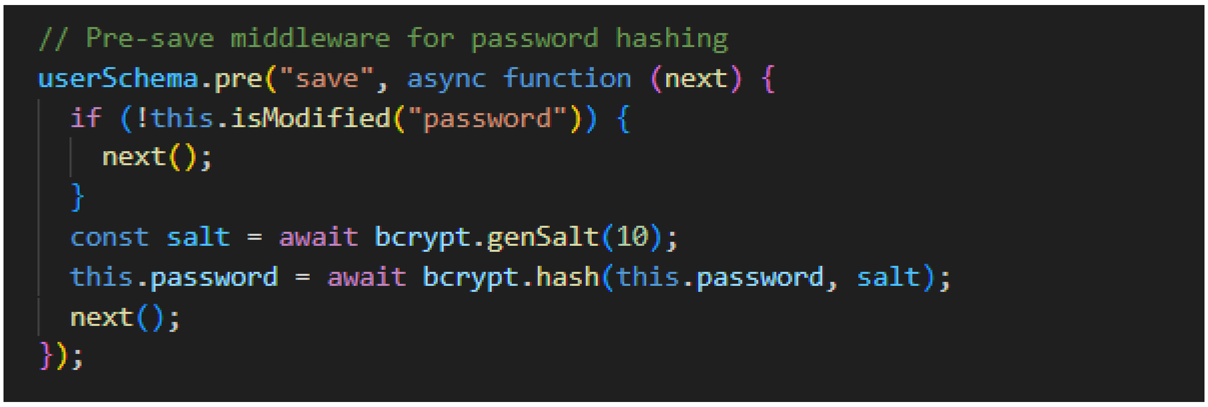
This communication between the frontend and backend, as well as between the game and external APIs, showcases the game's **interoperability**.

4. Virtual Identity

Virtual identity is implemented through a combination of **user authentication**, **password hashing**, and **JWT tokens**. These methods ensure that each user has a unique identity within the system, with their identity stored and validated securely. Here's a detailed explanation of how virtual identity is implemented:

a. Password Hashing:

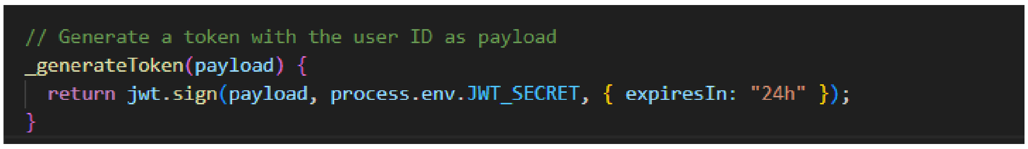
The user's password is stored in a hashed format using **bcryptjs** before being saved to the database. This prevents passwords from being stored in plaintext and ensures they are protected.



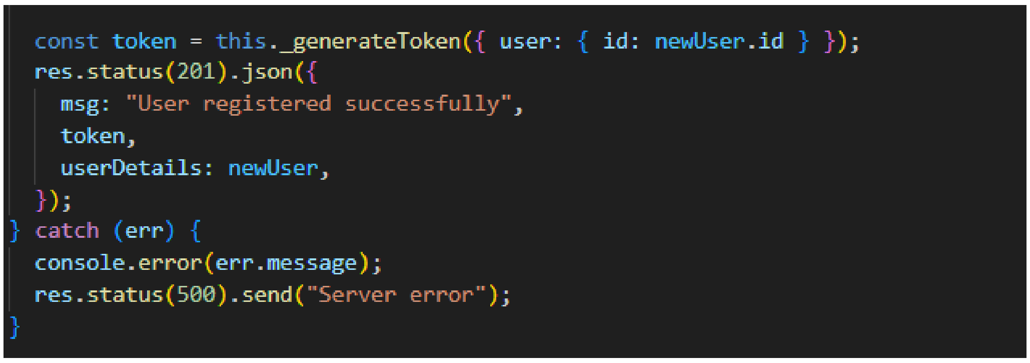
```
// Pre-save middleware for password hashing
userSchema.pre("save", async function (next) {
  if (!this.isModified("password")) {
    next();
  }
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});
```

b. JWT Tokens for Authentication:

When a user registers or logs in, a **JWT token** is generated. This token contains information about the user (e.g., user ID) and is signed using a secret key. The token acts as the user's **virtual identity**. It is returned to the client upon successful login or registration and is used for authenticating the user for future requests. The JWT token is stored on the client-side and sent along with every request that requires authentication. This allows the user to remain logged in across sessions without needing to re-enter credentials. The token has a 24-hour expiration period to enhance security. After it expires, the user must log in again to generate a new token.



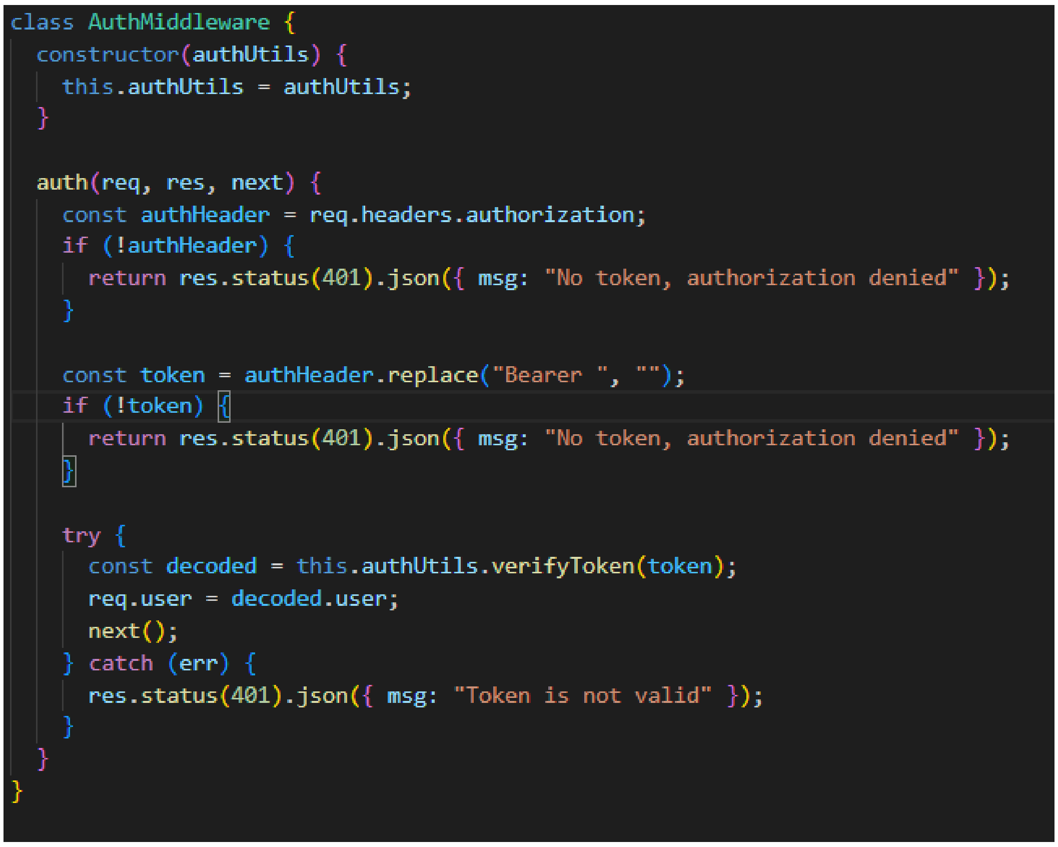
```
// Generate a token with the user ID as payload
_generateToken(payload) {
  return jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: "24h" });
}
```



```
  const token = this._generateToken({ user: { id: newUser.id } });
  res.status(201).json({
    msg: "User registered successfully",
    token,
    userDetails: newUser,
  });
} catch (err) {
  console.error(err.message);
  res.status(500).send("Server error");
}
```

c. Token Validation (Auth Middleware):

To ensure that only authenticated users can access certain routes, the AuthMiddleware verifies the token in the request headers. If the token is valid, the middleware decodes the token to extract the user's information and attaches it to the request object. If the token is missing or invalid, the middleware returns a 401 Unauthorized error, preventing unauthorized access.

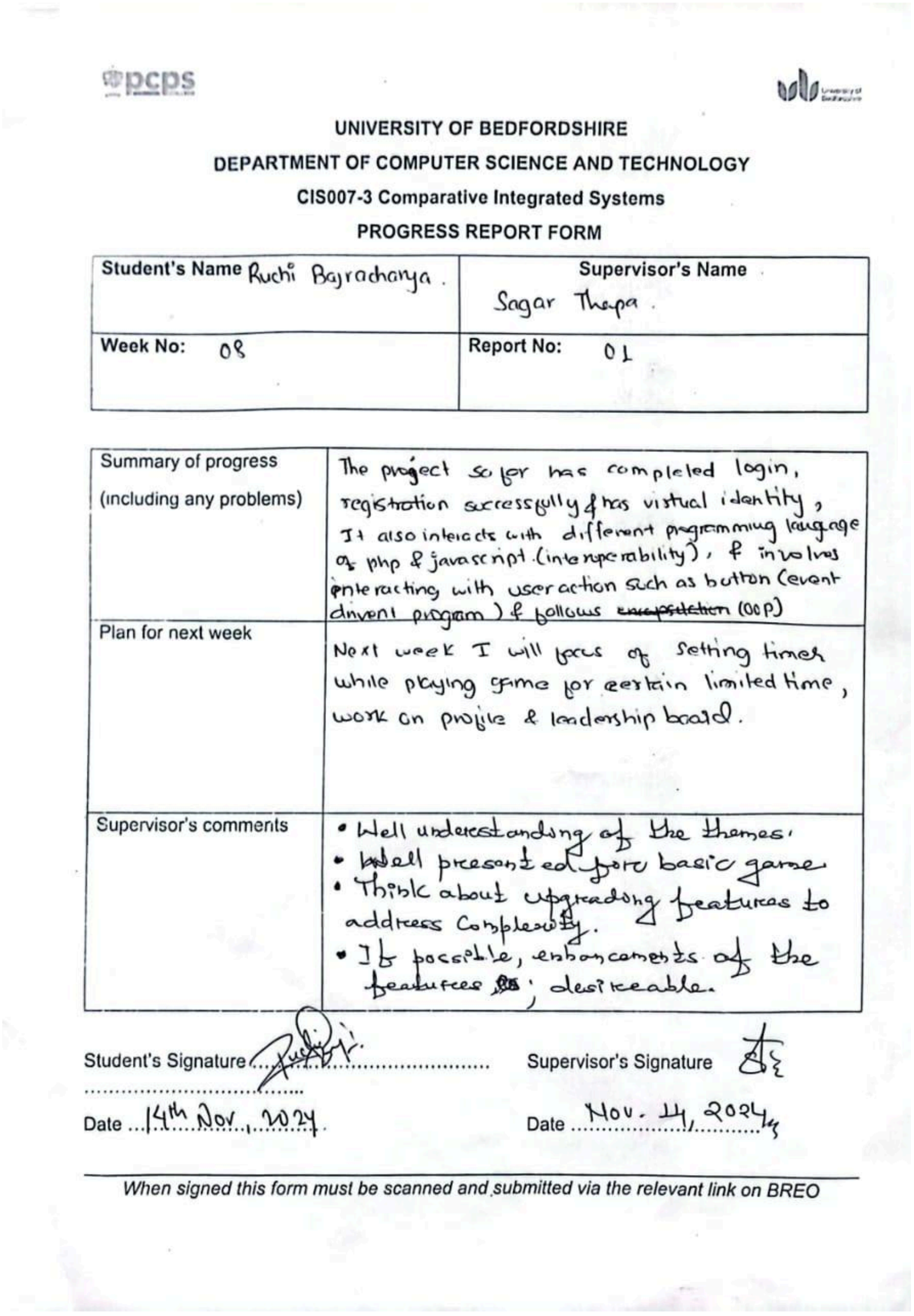


```
class AuthMiddleware {
  constructor(authUtils) {
    this.authUtils = authUtils;
  }

  auth(req, res, next) {
    const authHeader = req.headers.authorization;
    if (!authHeader) {
      return res.status(401).json({ msg: "No token, authorization denied" });
    }

    const token = authHeader.replace("Bearer ", "");
    if (!token) {
      return res.status(401).json({ msg: "No token, authorization denied" });
    }

    try {
      const decoded = this.authUtils.verifyToken(token);
      req.user = decoded.user;
      next();
    } catch (err) {
      res.status(401).json({ msg: "Token is not valid" });
    }
  }
}
```

5. Any other interesting features:

The Banana Game app includes several engaging and user-centric features to enhance the gaming experience. One notable feature is the difficulty levels, which allow users to select from three levels—easy, medium, and hard—each with unique time constraints and chances to guess the correct answer. Additionally, the app integrates a dynamic leaderboard that updates in real-time to showcase the top players and their scores, motivating users to improve their performance. The use of a secure login and registration system ensures player data is protected, while the countdown timer adds excitement and urgency to the gameplay. These features make the Banana Game not only fun and challenging but also a secure and interactive platform for users.

6. Week 8 Progress Report Evidence



UNIVERSITY OF BEDFORDSHIRE

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

CIS007-3 Comparative Integrated Systems

PROGRESS REPORT FORM

Student's Name Ruchi Bajracharya.	Supervisor's Name Sagar Thapa.
Week No: 08	Report No: 01

Summary of progress (including any problems)	The project so far has completed login, registration successfully & has virtual identity, It also interacts with different programming language of php & javascript (interoperability), & involves interacting with user action such as button (event driven program) & follows ~~encapsulation~~ (OOP)
Plan for next week	Next week I will focus of setting timer while playing game for certain limited time, work on profile & leadership board.
Supervisor's comments	• Well understanding of the themes. • Well presented for basic game. • Think about upgrading features to address complexity. • If possible, enhancements of the features; desireable.

Student's Signature [illegible]

Date 14th Nov, 2024.

Supervisor's Signature [illegible]

Date Nov. 14, 2024

When signed this form must be scanned and submitted via the relevant link on BREO