Blockchain identity verification for class polls

Fadil Amiruddin, Sanjay Mohan Kumar

Abstract

This project develops a blockchain-based system to prevent cheating in in-class polls by ensuring secure and private user verification. By incorporating Zero-Knowledge Proofs and blockchain technology, the system provides a tamper-proof and decentralized method for validating user identities, thus preserving the integrity of the educational assessment process and ensuring fair evaluations of student performance.

Contents

1. Introduction - The Problem

In recent years, the integrity of in-class polling systems in educational environments has been compromised by the prevalence of cheating, which skews assessment outcomes and misrepresents student performance. Traditional verification methods often fall short, as they are susceptible to manipulation and fail to offer a secure means of authenticating user identities. To address this issue, our project proposes the development of a blockchain-based verification system that utilizes the principles of Zero-Knowledge Proofs (ZKPs) to ensure both the security and privacy of user data. By leveraging the immutability and decentralized validation capabilities of blockchain technology, our solution aims to establish a robust framework that prevents unauthorized access and ensures that each vote in class polls is accurately recorded and remains unaltered. This initiative not only enhances the credibility of polling results but also reinforces the fairness and accuracy of the educational assessment process.

1.1 Importance of addressing this issue

Addressing the problem of cheating in in-class polls is crucial for maintaining the integrity of educational assessments. Accurate and fair polling ensures that student performance is evaluated correctly, which is fundamental to both academic credibility and the effectiveness of the educational system. Implementing a secure verification system thus protects against biases and enhances the reliability of the outcomes, fostering a more honest and equitable educational environment.

2. Application of Blockchain technology

2.1 Blockchain interaction with dock.io

The Dock blockchain is purpose-built to support decentralized identity and the issuance, management, and verification of digital credentials securely and privately. It operates as an immutable registry for decentralized identifiers (DIDs), which are unique, verifiable identifiers that do not require a centralized registry. These DIDs are used to validate the credentials' issuance without needing to reveal the identity of the issuer or the individual. This system enhances trust and privacy as the verifier can simply check the authenticity of the credential against the DID listed on the Dock blockchain without needing to directly interact with the personal data of the individual.

The use of Zero-Knowledge Proofs (ZKPs) plays a crucial role in further securing these interactions. ZKPs allow a user to prove possession of certain information or qualifications without revealing the information itself. This capability is key in contexts where privacy is paramount—such as in verifying age or residency without disclosing specific details like the birth date or home address. By allowing for selective disclosure, Dock not only ensures compliance with stringent privacy laws but also mitigates the risk of personal data being compromised. This technology integration positions Dock as a robust platform for managing digital identities in a secure and user-centric manner.

On the blockchain, the verification process incorporates on-chain elements to ensure security and trust without compromising privacy. Key components stored on-chain include Decentralized Identifiers (DIDs) for credential issuers, which are linked to their public keys and verification methods, credential schemas that outline the structure and standards for the credentials issued, and revocation registries that track the status of issued credentials, allowing issuers to revoke them if necessary. This infrastructure supports the verification of credentials' authenticity by allowing any participant in the network to independently confirm the validity of the issuer and the credential status directly via the blockchain, ensuring that the actual credentials and personal user data remain off-chain to protect privacy.

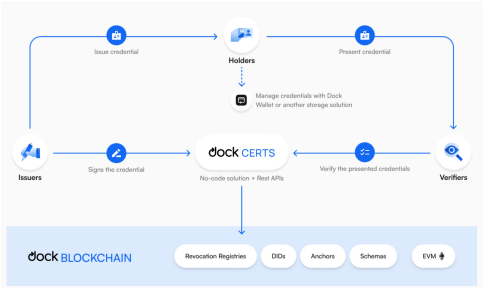

Figure 1: Interactions between users, versifiers, and the blockchain

2.2 Our Process

Our process using the blockchain to verify users with this technology works as follows. From our application, when the user (a class professor in our case) wants to create a new class and add students, they create an "Organization Profile" which issues a new DID and puts it on the chain. To add students they then issue credentials to each student one by one, and which will anchor it to the blockchain by publishing a hash of the credential to issue onto the chain which can be referenced later to verify. The user can choose to add this to their Dock wallet for easier verification as needed. When the user would like to verify a user for some action, they create a verification request to which the student then presents their credential which if it was the correct one issued, would then verify successfully.

3. Related works - Proof of Personhood (PoP)

Proof of Personhood (PoP) protocols represent a groundbreaking approach in the domain of digital identity verification, offering solutions that ensure each participant in a network is a unique, verifiable individual without sacrificing their privacy. This paradigm is detailed comprehensively in a publication by Paradigm Research, which explores various methodologies employed by PoP protocols, such as the use of biometric data, cryptographic techniques, and even real-time gatherings to authenticate personhood. These methodologies are critical for mitigating sybil attacks, where a single entity could present multiple identities, and are essential for the integrity of systems requiring assured individual participation, such as voting platforms or digital economies.

Further, the Paradigm Research article discusses the implications of PoP protocols beyond mere identity verification, suggesting their potential to fundamentally transform online interaction norms and security frameworks. By ensuring that each digital identity corresponds to a single human user, PoP protocols can enhance the authenticity and accountability of online actions, which is increasingly vital in areas like social media, financial transactions, and peer-to-peer services. This shift towards more secure and privacy-preserving verification mechanisms indicates a significant evolution in handling digital identities, potentially offering a more balanced approach between user privacy and systemic security. For professionals interested in the technical and societal impacts of these protocols, the full text offers in-depth insights and can be found on Paradigm Research's website here.

4. Challenges in our Project

1. Interactions between our site and the blockchain

    - First level bulleted list. This is the 1st item
    - First level bulleted list. Itemize creates bulleted lists, and description creates unnumbered lists.

2. Working with the Dock Certs API (in trial mode)
   Due to financial restrictions with the API, our model only represents a proof of concept of this type of user identification technology

    - Limit on API calls; In a full-scale model this could be a challenging issue to work around with multiple professors and numerous students per class.
    - Credentials could only be issued via email (our implementation of issuing the credentials) to the account holder's email; In a traditional model, we should be able to send emails to anyone's email ID.

3. Verification isn't instant; Since this is being done on the blockchain, even a single device (for one student) the verification process typically takes anywhere from 5 to 15 seconds. With multiple students and multiple classrooms operating at the same time, this process might take a long time to verify each and every student.

5. Code Documentation

The next few pages serves as a comprehensive site map for our website, delineating the structure and arrangement of its various pages and sections. Additionally, it discusses the interaction between the front-end and back-end code. It's important to note that JavaScript code for CSS-related functionalities is not the primary focus of this project.

To better understand the context of this report, it's essential to grasp three key components: Replit, Replit DB, and Flask.

1. **Replit**: Replit is an online integrated development environment (IDE) that allows users to write, test, and deploy code directly from their web browser. It provides a convenient platform for collaborative coding, offering features like real-time collaboration, version control, and seamless deployment.

2. **Replit DB**: Replit DB is a simple and lightweight database service provided by Replit. It enables developers to store and retrieve data persistently within their Replit projects. Replit DB is particularly useful for small to medium-scale applications where a full-fledged database management system might be excessive.

3. **Flask**: Flask is a lightweight and versatile web framework for Python. It simplifies the process of building web applications by providing tools and libraries for tasks such as routing, request handling, and template rendering. Flask is known for its simplicity and flexibility, making it an excellent choice for developing web applications of various scales and complexities.

Moreover, it's pivotal to acknowledge that Replit, the platform serving as the host for our code, has recently discontinued its provision of free deployment services. However, fret not, as I shall furnish you with meticulous guidance on how to navigate and scrutinize the code. For the interim, an alternative avenue is available for temporary deployment to facilitate testing endeavors. With each iteration, a distinct URL will be generated, which we shall temporarily denote as "picard.replit.dev" for easy reference.Furthermore, a critical aspect of this project entails the transmission of user identification via email for the purpose of class enrollment. This vital identification data is seamlessly conveyed through electronic mail. To streamline the testing process and ensure an efficient workflow, all pertinent email communications are channeled to a designated account. For access to this dedicated email repository, please navigate to the following web address: Proton MailUpon reaching the Proton Mail platform, you will be prompted to input credentials to gain entry. Please utilize the provided login details for seamless access to the account:

```
Email Address: Quizzy499@proton.me
Password: 12345678
```

By adhering to these instructions, you will be equipped with the requisite tools and resources to effectively engage with the project's components. This meticulous approach not only ensures ease of access but also fosters an environment conducive to comprehensive testing and evaluation.

6. Home

This is the main landing page of the website, featuring a simple yet effective design. It comprises two prominent buttons: "Student Login" and "Teacher Login". These buttons serve as gateways to the core functionalities of the site. For teachers, they provide access to content creation, editing, class management, and other administrative tasks. On the student side, these buttons grant access to quizzes created by teachers, facilitating seamless learning experiences.

6.1 Relevant code

On this page, there are essentially two types of relevant JavaScript functions that are triggered depending on whether the "Teacher Login" or "Student Login" buttons are clicked.

```
<button type="button" class="modal-action" onclick="window.location.href =
    '/teacher-login'">Teacher Login</button>

<button type="button" class="modal-action" onclick="window.location.href =
    '/student-login'">Student Login</button>
```

Listing 1: sending signals back to flask

URL
picard.replit.dev/teacher-login

These buttons utilize the window.location.href property to send a signal to the Flask server (Python code) to load and execute code in the specific app route corresponding to either teacher or student login.

7. Teacher Login

7.1 Relevant code

On this page, there are essentially two types of relevant JavaScript functions that are triggered depending on whether the "Teacher Login" or "Student Login" buttons are clicked.

```
<button type="button" class="modal-action" onclick="window.location.href =
    '/teacher-login'">Teacher Login</button>
```

```
<button type="button" class="modal-action" onclick="window.location.href =
    '/student-login'">Student Login</button>
```

Listing 2: sending signals back to flask

URL
picard.replit.dev/teacher-login

These buttons utilize the window.location.href property to send a signal to the Flask server (Python code) to load and execute code in the specific app route corresponding to either teacher or student login.

8. Teacher Login

As previously discussed, the HTML code activates a Flask route, specifically /teacher-login. The function associated with this route is outlined below:

```
@app.route('/teacher-login')
def teach_login():
    return render_template('teacher_login.html')
```

Listing 3: Flask Code

This function is responsible for loading the `teacher_login.html` file, thereby rendering the Teacher Login page. In future references within this report, when I mention that "performing X action renders Y page," it should be interpreted that such an action is facilitated by a succinct line of code similar to the one demonstrated above, which triggers the display of a new HTML page.

9. User Sign-Up Process

When a user clicks the sign-up button, the following JavaScript code is triggered:

```
$.ajax({
    type: 'POST',
    url: '/new_teach',
    data: JSON.stringify({username: username, email: email, password:
    password}),
    contentType: 'application/json',
    success: function(response) {
        console.log('Data sent successfully', response);
        // Handle the response from the server if needed
    },
    error: function(xhr, status, error) {
```

```
        console.error('Error sending data to server:', error);
        // Handle errors if needed
    }
});

$(document).ready(function() {
    // Placeholder for HTML page for sign-up
    // This part of the code waits for the document to be ready and sets up event listeners
    $('#finish').change(function() {
        // Check if the finish checkbox is checked and the welcome message is visible
        var username = $('#username').val();
        var email = $('#femail').val();
        var password = $('#fpass').val();
    });
});
```

Listing 4: JavaScript code for handling user sign-up

This JavaScript code is responsible for handling the user sign-up process. It performs several tasks:

- **Sending Data to Server:** Upon clicking the sign-up button, an AJAX request is made to the server at the URL `/new_teach`. The request contains the user's provided username, email, and password in JSON format.
- **Handling Success and Errors:** The code includes success and error handling functions to log messages to the console based on the response from the server. This allows for appropriate actions to be taken based on whether the data was sent successfully or if an error occurred.
- **Event Handling:** Additionally, the code sets up an event listener for changes to the 'finish' checkbox. When the checkbox is changed, the code retrieves the values of the username, email, and password fields from the sign-up form.

This JavaScript functionality integrates seamlessly with the HTML sign-up page and facilitates communication with the server-side logic implemented in Flask. Speaking of the server-side logic, when the client sends the sign-up information, it is processed by the following Flask function:

```
@app.route('/new_teach', methods=['POST'])
def new_teach():
    data = request.get_json()

    # Check if the teacher already exists in the database
    existing_teacher = db.get(data['email'])
```

```
7
8       if existing_teacher:
9           # If the teacher already exists, return a message indicating that
10          return jsonify({'message': 'Teacher already exists'})
11      else:
12          # If the teacher doesn't exist, insert a new record into the
    database
13          db[data['email']] = {'username': data['username'], 'email': data['
    email']}
14          # Return a success message
15          return jsonify({'message': 'Teacher added successfully'})
```

Listing 5: Flask function for handling new teacher registration

This Flask function processes the received data, checks if the teacher already exists in the database, and either adds a new record or returns a message indicating that the teacher already exists.

On the dock blockchain wallet side of things, this process would be as simple as getting the credential JSON file that would get sent to the student's email and uploading it as follows:

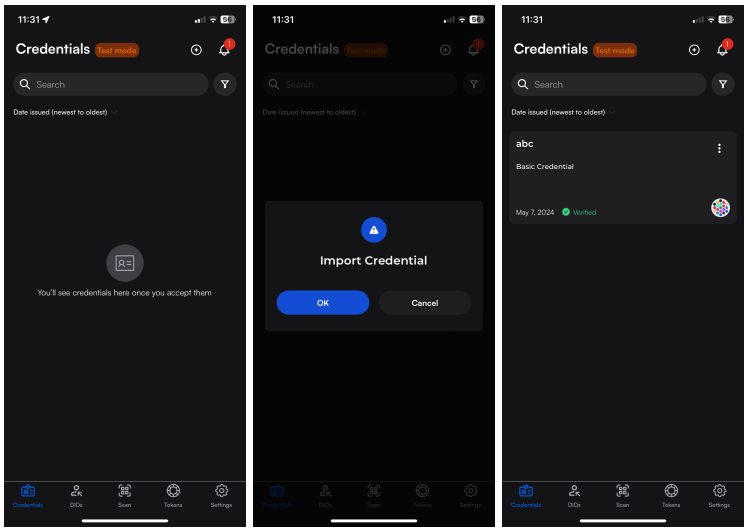

Figure 2: Interactions between users, versifiers, and the blockchain

9.1 Logging In

When a user hits login, the following JavaScript code is executed:

```
1  <script>
2  function validateLogin() {
3      var username = $('#username').val();
4      var password = $('#pass').val();
5
6      $.ajax({
7          url: '/validate_teacher_login',
```

```
        type: 'POST',
        contentType: 'application/json',
        data: JSON.stringify({username: username, password: password}),
        success: function(code) {
            console.log('Data sent successfully', code);
            var newUrl = '/teacher_dashboard?username=' +
    encodeURIComponent(username);
            window.location.href = newUrl; // Redirect to the new page
        },
        error: function(xhr, status, error) {
            console.error('Error:', error);
            // Handle error
        }
    });
}
</script>
```

Listing 6: JavaScript code for validating user login

This JavaScript code is responsible for validating the user's login credentials. It performs the following actions:

- **Data Validation:** Retrieves the entered username and password from the respective input fields.
- **AJAX Request:** Sends an AJAX request to the server-side route `/validate_teacher_login` with the provided username and password in JSON format.
- **Handling Success:** If the login is successful, the code redirects the user to the teacher dashboard page with the username encoded in the URL for future reference.
- **Handling Errors:** If an error occurs during the login process, it is logged to the console for debugging purposes.

It's important to note that for the sake of expediency in this project, user information is passed through the URL. However, in actual deployment scenarios, this method can introduce security vulnerabilities such as cross-site scripting (XSS) attacks. Proper authentication mechanisms and session management should be implemented to ensure secure user authentication.

10. Teacher Account for Quizzy499

10.1 Teacher Dashboard

The teacher dashboard serves as the central interface for teachers to manage their classes and quizzes. It is also the page they are met with after logging in. Below is the HTML code for the teacher dashboard along with relevant backend interactions:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>View Teacher Classes</title>
</head>
<body>
    <h1>View Teacher Classes</h1>
    <p>Click the buttons below:</p>

    <!-- Button to view classes -->
    <button id="viewClassesButton">View Classes</button>

    <!-- Button to add a new class -->
    <button id="goToAddNewClassButton">Add New Class</button>

    <!-- Button to manage classes -->
    <button id="manageClassesButton">Manage Classes</button>

    <!-- Button to manage quizzes -->
    <button id="manageQuizzesButton">Manage Quizzes</button>

    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

    <script>
        // JavaScript functions for button functionality
        // These functions interact with backend Flask routes to perform various actions
    </script>
</body>
</html>
```

Listing 7: HTML code for the teacher dashboard

The HTML code defines the structure of the teacher dashboard, including buttons for viewing classes, adding new classes, managing classes, and managing quizzes. These buttons trigger JavaScript functions that interact with backend Flask routes to handle user actions.

Key aspects of the code include:

- **Integration with Backend:** The buttons on the dashboard trigger JavaScript functions, which make AJAX requests to backend Flask routes to perform actions such as viewing classes, adding new classes, managing classes, and managing quizzes.
- **Dynamic Content Generation:** The dashboard dynamically generates content based on data fetched from the backend. For example, when a teacher clicks the "View Classes"

button, the frontend triggers a request to the backend to retrieve the teacher's classes from the database and display them on the dashboard.

- **User Interaction:** The dashboard provides a user-friendly interface for teachers to interact with the application. Teachers can easily navigate between different functionalities such as class management and quiz creation using the buttons provided.

When a button is pressed, it triggers a specific JavaScript function that interacts with the backend Flask routes to perform the following actions:

- **Add New Class:** Opens a form for the teacher to input details and add a new class to the system.
- **Remove Classes:** Allows the teacher to delete existing classes.
- **Manage Class:** Allows teacher to add/remove a student to a class, as well as create quizzez for the class

These functionalities enhance the teacher's ability to efficiently manage their classes and quizzes within the system.

11. Adding a New Class

The "Add New Class" feature is a crucial part of the system, enabling teachers to dynamically create new classes. This feature streamlines the process by providing a user-friendly interface where teachers can input the name of the class they wish to add.

11.1 Flask Code for Adding a New Class

The Flask code responsible for handling the addition of a new class is encapsulated within the ‘register_class‘ function. This function is associated with the route ‘/g/<username>‘, where ‘<username>‘ represents the unique identifier of the teacher.

```
@app.route('/g/<username>', methods=['GET', 'POST'])
def register_class(username):
    if request.method == 'POST':
        # Retrieve the input text from the form
        input_text = request.form['input_text']

        # Generate a unique identifier for the class
        did_creation_response = did.create_did()
        db_key = did_creation_response['did'][9:]

        # Add the new class to the teacher's list of classes
```

```
        db[username]['classes'].append(input_text)

        # Persist the updated list of classes

        print(f"Class '{input_text}' added for teacher '{username}'.")
        return 'Class added successfully.'

    # Serve the form to input the class name
    return render_template('add_new_class.html', username=username)
```

Listing 8: Flask Code for Adding a New Class

This Flask route handles both POST and GET methods. When a teacher submits the form with the class name, the route receives a POST request containing the input text. The function then processes this input by creating a unique identifier for the class and adding it to the teacher's list of classes in the database. Additionally, it prints a confirmation message to the console. If the route receives a GET request, indicating the need to render the form, it serves the HTML template 'add_new_class.html'.

11.2 HTML for Adding a New Class

The HTML template 'add_new_class.html' provides the interface for teachers to input the name of the new class. It consists of a simple form with a text input field and a submit button.

```
<!DOCTYPE html>
<html>
<head>
    <title>Simple Flask Application</title>
    <style>
        /* Relevant CSS: Omitted for brevity. Refer to add_new_class.html for details */
    </style>
</head>
<body style="display: flex; justify-content: center; align-items: center; height: 100vh; margin: 0;">
    <div style="display: flex; flex-direction: column; align-items: center; justify-content: center; height: 100vh;">
        <h1>Enter the name of the class you want to create:</h1>
        <form method="POST">
            <input type="text" class="form__field" name="input_text">
            <input type="submit" class="btn btn--primary btn--inside uppercase" value="Submit">
        </form>
    </div>
</body>
</html>
```

Listing 9: HTML Markup for Adding a New Class

This HTML markup creates a visually appealing and intuitive form for teachers to interact with. Upon submitting the form, the entered class name is sent to the Flask route for processing.

12. Delete Class

The "Delete Class" functionality allows teachers to remove classes from their list. Once a class is deleted, it cannot be recovered, so caution is advised. Below is the Flask code and HTML markup for implementing this feature.

12.1 Flask Code for Deleting a Class

The Flask route `/delete_class` handles the deletion of a class from the teacher's list of classes. It expects a POST request containing the name of the class to be deleted in the request body. The function then searches for the specified class in the teacher's list and removes it if found. If the operation is successful, it returns a JSON object with a status of 1; otherwise, it returns a status of 0 along with an error message.

```
@app.route('/delete_class', methods=['POST'])
def delete_class():
    # Extract the username from the session
    username = session.get('username')

    # Extract the class name from the request JSON
    data = request.json
    class_name = data.get('class_name')

    try:
        # Search for the class in the teacher's list of classes
        for i, sublist in enumerate(db[username]['classes']):
            if str(sublist) == str(class_name):
                break
        else:
            raise ValueError("Class not found in teacher's list")

        # Delete the class from the teacher's list
        del db[username]['classes'][i]

        # Return a success status
        return jsonify({"status": 1})

    except Exception as e:
        # Return an error status along with the error message
        error_message = str(e)
```

```
        return jsonify({"status": 0, "error_message": error_message})
```

Listing 10: Flask Code for Deleting a Class

12.2 HTML Markup for Deleting a Class

The HTML template `delete_class.html` provides a warning message about the irreversibility of class deletion and displays a list of the teacher's classes along with a "Delete" button for each class. When the "Delete" button is clicked, it triggers a JavaScript function to send an AJAX request to the Flask route `/delete_class` with the name of the class to be deleted. Upon successful deletion, the page is reloaded to reflect the updated list of classes.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- CSS and JavaScript imports omitted for brevity -->
</head>
<body>
    <!-- Warning message -->
    <h1 class="warning-message">Warning: Once a class is deleted, it is gone forever!</h1>

    <!-- List of classes with delete buttons -->
    <ul>
        {% for class_name in classes %}
            <li>
                <div>
                    <span class="class-name">{{ class_name[0] }}</span>
                    <button class="button-49" data-class="{{ class_name }}">Delete</button>
                </div>
            </li>
        {% endfor %}
    </ul>

    <!-- JavaScript function to handle class deletion -->
    <script>
        function deleteClass(className) {
            // Send an AJAX request to Flask to delete the class
            fetch('/delete_class', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({ class_name: className })
            })
```

```
            .then(response => {
                if (response.ok) {
                    // Reload the page after successful deletion
                    location.reload();
                } else {
                    console.error('Failed to delete class:', response.
    statusText);
                }
            })
            .catch(error => {
                console.error('Error deleting class:', error);
            });
        }

        // Add event listeners to delete buttons
        document.querySelectorAll('.button-49').forEach(btn => {
            btn.addEventListener('click', function() {
                const className = this.getAttribute('data-class');
                if (confirm('Are you sure you want to delete class ' +
    className + '?')) {
                    deleteClass(className);
                }
            });
        });
    </script>
</body>
</html>
```

Listing 11: HTML Markup for Deleting a Class

13. Add Student

The "Add Student" functionality allows teachers to add students to their classes. When a student is added, an email notification is sent to the teacher, and a credential is created for the student to access the class material. Below is the Flask code and HTML markup for implementing this feature.

13.1 Flask Code for Adding a Student

The Flask route `/AddStudent` handles the addition of a student to a class. It expects a POST request containing the name of the class (`className`) and the name of the student (`studentName`). Upon receiving the request, the function retrieves the student's username (`v`) from the database, creates a credential using the `create_credential` function, and sends an email notification to the teacher with the credential details.

```python
@app.route('/AddStudent', methods=['POST'])
def AddStudent():
    data = request.get_json()
    student_name = data.get('studentName')
    v = db[student_name]['username']
    x = create_credential(db[data.get('className')], student_name, v, data.get('className'))
    send_mail_with_json('fadil.amiruddin1@gmail.com', 'New Student Data', 'New student data added to the database', x)
    return x
```

Listing 12: Flask Code for Adding a Student

13.2 HTML Markup for Managing Classes

The HTML template `Manage_Class_Menu.html` provides a user interface for teachers to manage their classes. It displays a list of classes with buttons to add or remove students. When the "Add Student" button is clicked, an input field is shown where the teacher can enter the student's name. Upon pressing the Enter key, an AJAX request is sent to the Flask route `/AddStudent` to add the student to the class. Similarly, when the "Create Quiz" button is clicked, a request is sent to another route for quiz creation.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- CSS and JavaScript imports omitted for brevity -->
</head>
<body>
    <!-- Container for class management -->
    <div class="container">
        <!-- Title indicating the selected class -->
        <h1 id="classTitle" contenteditable>Select A Class</h1>

        <!-- Button set for selecting a class -->
        <ul class="shadow-button-set" id="buttonSet">
            {% for class_name in classes %}
                <li>
                    <button onclick="addClassToURL('{{ class_name[0] }}')">{{ class_name[0] }}</button>
                </li>
            {% endfor %}
        </ul>

        <!-- Button set for managing students (hidden by default) -->
        <ul class="shadow-button-set" id="studentButtonSet" style="display: none;">
```

```
            <li class="green">
                <button onclick="addStudent()">Add Student</button>
            </li>
            <li class="red">
                <button onclick="removeStudent()">Create Quiz</button>
            </li>
        </ul>
    </div>

    <!-- JavaScript imports and script omitted for brevity -->
</body>
</html>
```

Listing 13: HTML Markup for Managing Classes

This documentation provides insights into how the "Add Student" feature is implemented in Flask and HTML, including the handling of requests and the user interface for managing classes.Before I can talk about the authetication side of this application it is important that we have a understanding on how the backend of th student side of this application works

14. Student Login

The student login feature enables students to log in to the application using their credentials. Upon successful login, students gain access to the quizzes available to them.

14.1 Implementation Details

The login functionality is implemented using Flask routes and HTML forms. When a student submits their login credentials through the login form, the submitted data is processed on the server-side to authenticate the user.

14.1.1 Flask Route

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Process login form data
        # Authenticate user
        # Redirect to appropriate page
    else:
        # Render login page
```

Listing 14: Flask Route for Student Login

14.1.2 HTML Form

The HTML form for the login page includes input fields for username and password. Upon submission, the form sends a POST request to the login route.

```
<form method="POST" action="/login">
    <input type="text" name="username" placeholder="Username" required>
    <input type="password" name="password" placeholder="Password" required>
    <button type="submit">Login</button>
</form>
```

Listing 15: HTML Form for Student Login

15. Student Sign-Up

15.1 Registration Route

The `/signup` route handles the registration of new students. It accepts POST requests containing the student's username, email, and GNumber. The GNumber serves as the primary key, allowing teachers to add students to classes and facilitating the backend's ability to send verification information to the student.

```
@app.route("/signup", methods=['POST'])
def signup():
    if request.method == 'POST':
        # Retrieve student registration data from the request
        data = request.get_json()
        username = data['username']
        email = data['email']
        gnumber = data['gnumber']

        # Check if the GNumber already exists in the database
        if gnumber in db:
            return jsonify({'error': 'GNumber already exists'})

        # Store the student's information in the database
        db[gnumber] = {'username': username, 'email': email}
        return jsonify({'message': 'Registration successful'})
```

Listing 16: Flask Route for Student Registration

The `/signup` route allows students to register by providing their username, email, and GNumber. The GNumber acts as the unique identifier for each student, enabling teachers to add students to classes and allowing the backend to send verification information when needed.

16. Quiz Retrieval

To retrieve the quiz data, an AJAX request is made to the server. This request fetches the quiz questions, answer options, and correct answers from the backend.

16.1 Implementation Details

The quiz retrieval functionality involves handling AJAX requests on the server-side and responding with the appropriate quiz data.

16.1.1 Flask Route

```
@app.route('/get_quiz', methods=['GET'])
def get_quiz():
    # Retrieve quiz data from the database
    # Return quiz data as JSON response
```

Listing 17: Flask Route for Quiz Retrieval

16.1.2 AJAX Request

The AJAX request is made from the client-side to the get_quiz route using JavaScript.

```
fetch('/get_quiz')
    .then(response => response.json())
    .then(data => {
        // Process quiz data
    });
```

Listing 18: AJAX Request for Quiz Retrieval

17. Quiz Submission

After attempting a quiz, students can submit their answers for evaluation. The submitted answers are processed on the server-side to calculate the score.

17.1 Implementation Details

The quiz submission functionality involves processing the submitted answers and comparing them against the correct answers stored in the database.

17.1.1 Flask Route

```
@app.route('/submit_quiz', methods=['POST'])
def submit_quiz():
    # Process submitted answers
    # Calculate score
    # Return score to the client
```

Listing 19: Flask Route for Quiz Submission

17.1.2 HTML Form

The HTML form for submitting the quiz includes radio buttons for selecting answers to each question. Upon submission, the form sends a POST request to the submit_quiz route.

```
<form method="POST" action="/submit_quiz">
    <!-- Quiz questions with radio buttons -->
    <button type="submit">Submit</button>
</form>
```

Listing 20: HTML Form for Quiz Submission

18. Quiz Generation

Quiz questions are dynamically generated based on the quiz selected by the student. The quiz data is retrieved from the database and rendered on the quiz page.

```

function generateQuiz() {
          const container = document.getElementById('quiz-container');
          allQuestions.forEach((item, index) => {
              const questionDiv = document.createElement('div');

              questionDiv.className = 'question';
              questionDiv.innerHTML = `<p class ="underlined underlined--
    thick">${index + 1}. ${item.question}</p>`;
              Object.entries(item.answers).forEach(([key, value]) => {
                  const label = document.createElement('label');
                  label.innerHTML = `<input type = "radio"  id ="f-option"
    button class="button-89" role="button"name="question${index}" value="${
    key}"> ${value}<br><br>`;

                  questionDiv.appendChild(label);
              });
              container.appendChild(questionDiv);
          });}
```

Listing 21: Dynamic Quiz Generation

19. Adding and Verifying a student in the class.

Now that we have gathered all the necessary information, let's delve into the process of adding and authenticating students. To illustrate this, let's start by revisiting the teacher sign-in process and logging in as a teacher account you have previously created. Then, proceed to add a class and press enter. Afterward, navigate to the manage class section and press "add student." Enter the GNumber from earlier, and after clicking enter, within a few seconds, the ProtonMail account mentioned earlier should receive an email containing a JSON file.

Next, download that JSON file to the mobile device with the Dock Cert app installed. Now, navigate to the credential section of the Dock Cert app and import the JSON file. At this point, you should have an ID linked to the class.

Whenever a user attempts to take a quiz, a QR code requesting an ID will pop up before they can proceed. To complete the verification process, present the ID along with the name of the class.

Let's break down how this process is implemented in terms of code to gain a better understanding.

20. Dockcert API Integration

The Dockcert API is utilized for creating digital credentials for students. Here's how it works:

1. **API Authentication**: The application authenticates with the Dockcert API using an API key. This key is stored securely and used to authorize requests made to the API.

```
    # API key stored securely
    api_key = "your_dockcert_api_key"
    headers = {
        'Authorization': f'Bearer {api_key}',
        'Content-Type': 'application/json'
    }

```

Listing 22: API Authentication

2. **Creating Credentials**: When a new student signs up or is added to a class by a teacher, the application generates a credential object containing relevant information such as the student's GNumber, full name, and the name of the class. This information is necessary for issuing the credential.

```
    def create_credential(issuer_did, gnumber, name, classname):
        credential = {
            "anchor": True,
            "distribute": True,
            "recipientEmail": "recipient@example.com",
            "credential": {
                "type": ["BasicCredential"],
                "name": name,
                "subject": {
                    "id": gnumber,
                    "FullName": name,
                    "CourseName": classname
                },
                "issuer": issuer_did
            }
        }

        response = requests.post(f"{dockcert.BASE_URL}/credentials",
    headers=headers, json=credential)
        return response.json()

```

Listing 23: Creating Credentials

3. **Credential Generation**: The application sends a POST request to the Dockcert API endpoint responsible for creating credentials. This request includes the credential object as JSON data.

4. **Response Handling**: Upon receiving the request, the Dockcert API processes the information and generates a digital credential for the student. The API returns a response containing the newly created credential, typically in JSON format.

5. **Credential Storage**: The application may store the generated credential securely for future reference. This could involve storing it in a database or associating it with the student's account.

21. Mailchimp API Integration

The Mailchimp API is used for sending emails with JSON attachments. Here's how it's integrated into the application:

1. **Mailchimp Account Setup**: The application's backend is configured to interact with the Mailchimp API. This involves creating an account on the Mailchimp platform and obtaining an API key, which is used to authenticate requests.

```
    # Mailchimp API key stored securely
    api_key = "your_mailchimp_api_key"

```

Listing 24: Mailchimp Account Setup

2. **Sending Emails**: When certain events occur, such as a student signing up or a teacher adding a student to a class, the application triggers an email notification. The email contains important information or attachments (e.g., JSON files containing digital credentials).

```
    def send_mail_with_json(to, subject, body, json_data):
        # Convert the JSON data to a string
        json_string = json.dumps(json_data)

        # Create the email
        encoded_json = base64.b64encode(json_string.encode())
        attachment = mt.Attachment(filename="data.json", content=
    encoded_json)
        mail = mt.Mail(
            sender=mt.Address(email="sender@example.com", name="Sender
     Name"),
            to=[mt.Address(email=to)],
            subject=subject,
            html=body,
            attachments=[attachment],
            category="Integration Test",
        )

        # Send the email
        client = mt.MailtrapClient(token="your_mailchimp_api_key")
        client.send(mail)

```

Listing 25: Sending Emails

3. **API Authorization**: The request to the Mailchimp API includes the API key obtained

during account setup. This key serves as a form of authentication, allowing the application to send emails on behalf of the configured Mailchimp account.

4. **Email Delivery**: Upon receiving the request, the Mailchimp API processes the email and attachments, ensuring they meet any formatting requirements. It then sends the email to the specified recipient(s) using the configured Mailchimp account.

By integrating both the Dockcert API and the Mailchimp API into the application, it becomes possible to automate the process of creating digital credentials for students and sending them via email, streamlining administrative tasks for teachers and improving the user experience for students.

22. Verification Process

The verification process involves checking if a student has a valid digital credential issued through the Dockcert API. Here's how it works:

1. **QR Code Authentication**: When a student attempts to access a quiz, a QR code requesting their ID is displayed. The student must present their ID, which is typically a digital credential obtained through the Dockcert API.

2. **Credential Verification**: Upon receiving the student's ID, the application verifies its authenticity by querying the Dockcert API. This involves sending a request to the API endpoint responsible for retrieving credential information.

3. **API Request**: The application sends an HTTP request to the Dockcert API endpoint, passing the student's ID as a parameter. The API responds with the credential information associated with the provided ID.

4. **Response Handling**: The application receives the API response containing the credential information. It parses the response to extract relevant data such as the student's name, class, and any additional details.

5. **Access Granted**: If the credential is valid and the student's information matches the expected criteria, access to the quiz is granted. The student can proceed to take the quiz without any further authentication steps.

23. Retrieving Verified Users

To retrieve a list of verified users who have scanned a given QR code and possess valid digital credentials, the application can utilize subprocess in Python to execute a command that interacts with the Dockcert API. Here's an example Python code snippet demonstrating this:

```
import subprocess

command = '''
    API_KEY="your_dockcert_api_key"

    curl 'https://api-testnet.dock.io/proof-templates/d0155104-c2d0-4fe2-b5e8-f79ddb570047/history?offset=0&limit=64' \
      -H 'accept: application/json' \
      -H "DOCK-API-TOKEN: $API_KEY" | jq -r '.[] | .presentation.credentials[].name'
    '''

# Execute the command
result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
print(result.stdout)
```

Listing 26: Retrieving Verified Users

This code snippet executes a command using the subprocess module to query the Dockcert API for the history of scanned QR codes and retrieve the names of verified users. The result is then printed to the console for further processing or display.