ProManage

Collaboration: **u5653630** and u5614928 and

May 21, 2025

Contents

1 Introduction

This report presents the implementation of **ProManage**, a project management web application, using technologies such as HTML, CSS, and SQLAlchemy. This was a collaboration project between two people. My section involved designing, implementing, and testing the front-end components before handing the finalised front-end code over to my teammate (u5614928), who integrated the backend logic to provide a fully functional web page.

2 Design

Prior to development a structure diagram was designed to break the problem down into manageable units. The structure diagram highlights how both front-end and back-end logic will be implemented, providing a high-level view for program functionality. This not only reduces the complexity of the problem but also allows the diagram to be used as a reference point when implementing the solution.

2.1 Structure Diagram

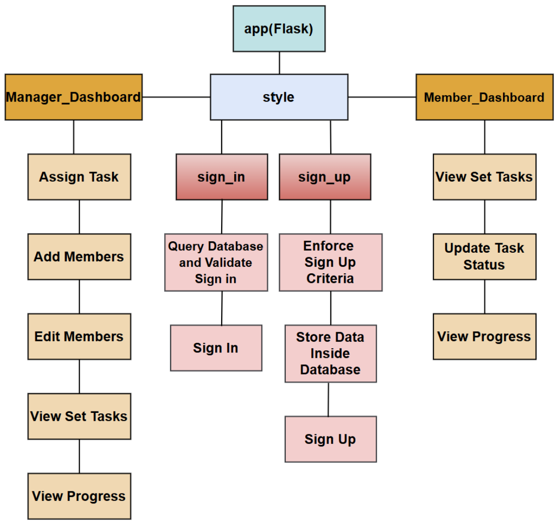


Figure 1: Structure Diagram

2.2 Colour Classification

Colour	Script	Purpose
🟦 Turquoise	`app.py` (Flask)	Core backend file that handles routing, logic, session control, and integrates form and database operations.
🟦 Blue	`style.css`	Defines the visual layout and aesthetic styling of all front-end components including form elements, dashboards, and tables.
🟧 Orange	Dashboard Templates (`manager_dashboard.html`, `member_dashboard.html`)	Role-based interfaces that allow task management, progress tracking, and member interaction depending on user type.
🟥 Red	Authentication Templates (`sign_in.html`, `sign_up.html`)	User authentication pages where credentials are entered, validated, and passed to backend for processing.

Table 1: Colour classification for the structure diagram

3 Flowcharts

Flowcharts were included as part of the report, in order to highlight the integration between front-end and back-end functionality in the web page.

3.1 Authentication Flowcharts

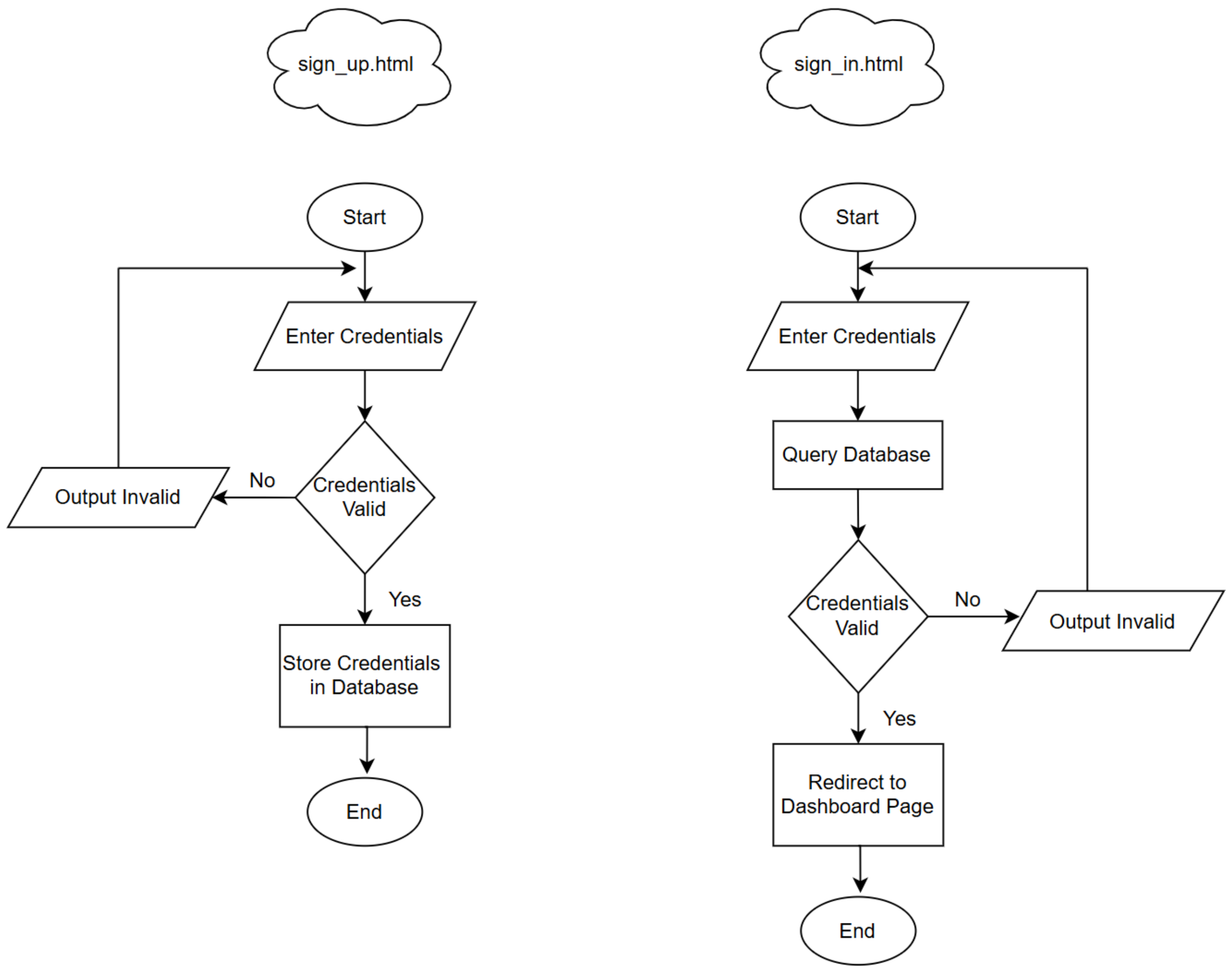


Figure 2: Authentication Flowcharts

To use the website, users must first register before accessing the platform, regardless of their role. To sign up, they will have to pass through strict validation checks ensuring their passwords meet the set criteria, before they are hashed and securely stored in the database as outlined in Figure 2. The user can then enter the website by successfully inputting the valid credentials on the sign-in page. The website checks this by querying the database to see if a record exists matching the entered credentials. In case of a match, the user is then redirected to the corresponding dashboard based on their role. If not, an alert error is shown, prompting the user to try again.

3.2 Dashboard Flowcharts

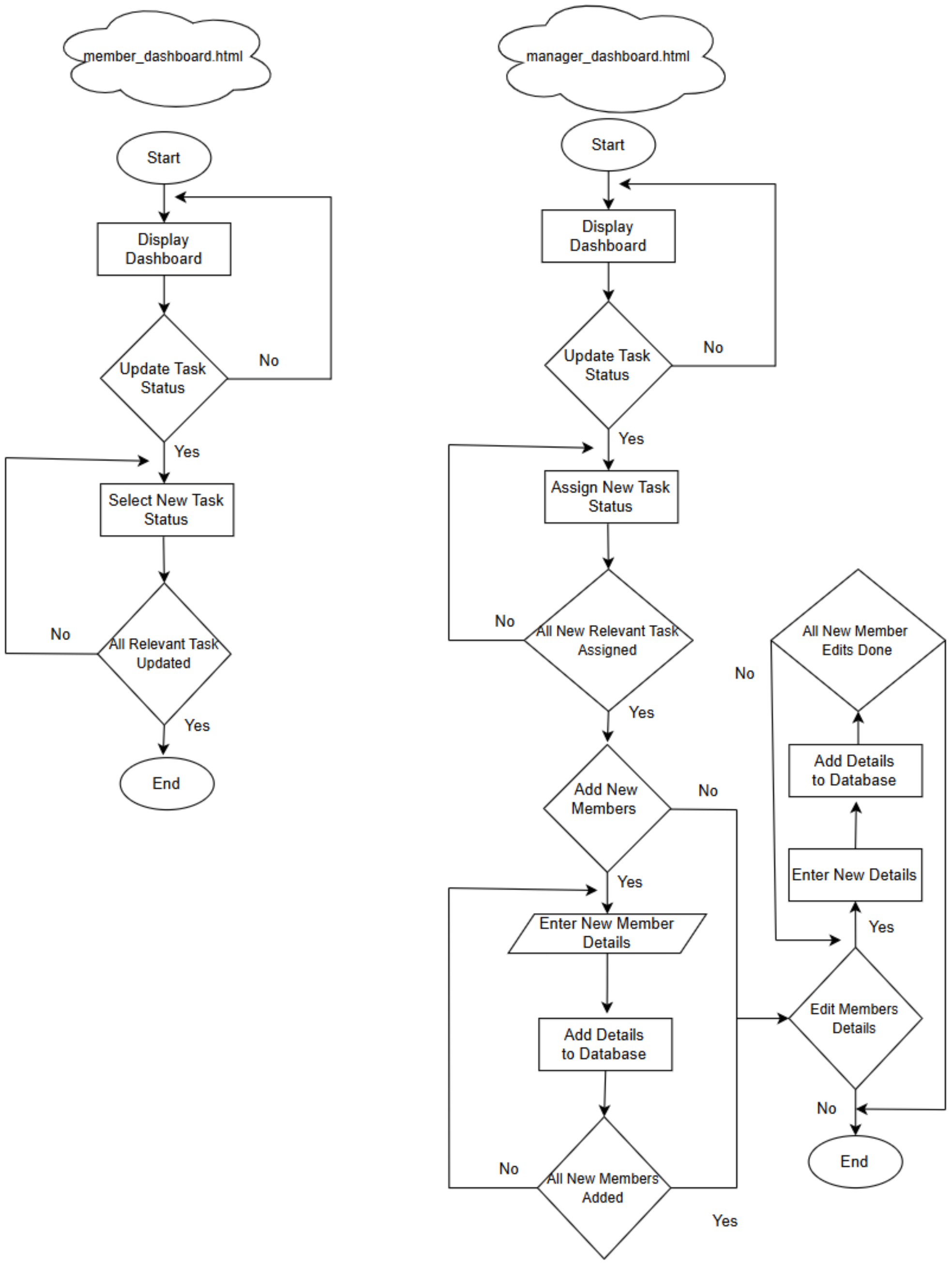


Figure 3: Dashboard Flowcharts

Flowcharts in Figure 3 above, simplify the actual coded implementation and displays only the functionalities that the user will most likely use, neglecting edge cases such as invalid input, unexpected navigation behaviour, or malicious attempts to exploit the website.

The flowchart on the left outlines the typical workflow for a team member, with the primary decision point being whether to update the status of their assigned tasks. The flowchart on the right, on the other hand, represents the logic and decisions handled by the team manager — including assigning tasks to members, adding new members, and editing the member list.

4 Final Product

This section shows the final coded implementation of the front-end, highlighting the core logic and aspects of error handling. The focus will be on comparing the user interface, while detailed error handling will be discussed in depth later in the testing section of the report.

4.1 Authentication System

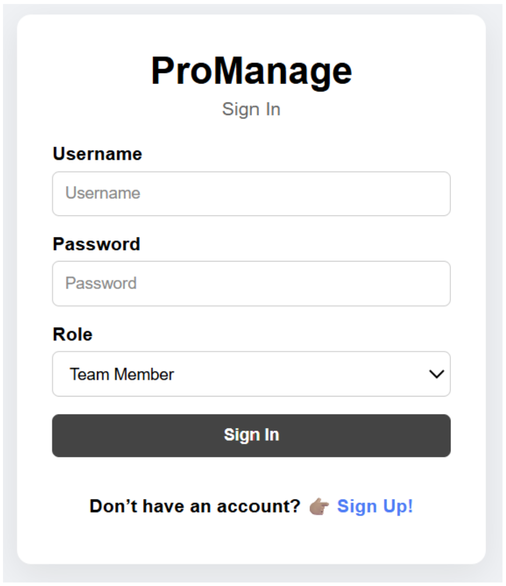


Figure 4: Sign In Page (Neupane, 2024; design inspired)

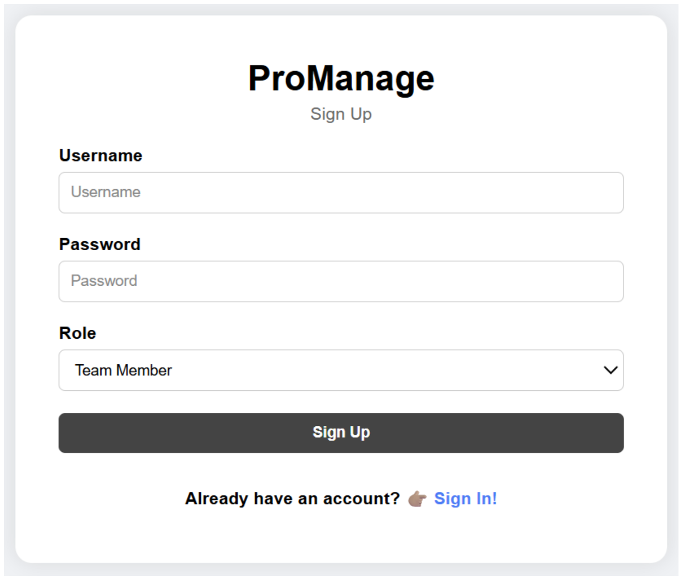
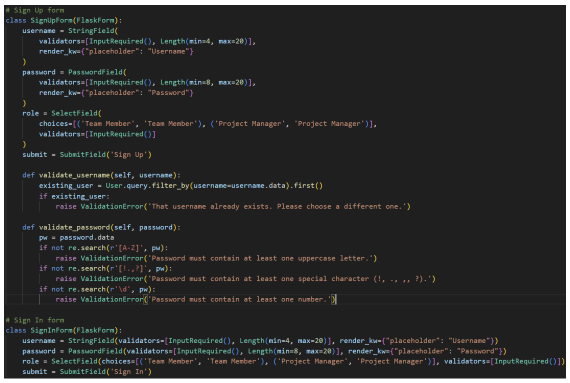
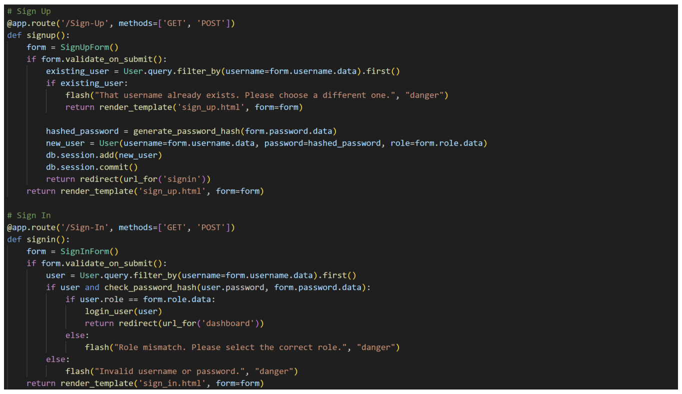


Figure 5: Sign Up Page (Neupane, 2024, design inspired)



```python
# Sign Up form
class SignUpForm(FlaskForm):
    username = StringField(
        validators=[InputRequired(), Length(min=4, max=20)],
        render_kw={"placeholder": "Username"}
    )
    password = PasswordField(
        validators=[InputRequired(), Length(min=8, max=20)],
        render_kw={"placeholder": "Password"}
    )
    role = SelectField(
        choices=[('Team Member', 'Team Member'), ('Project Manager', 'Project Manager')],
        validators=[InputRequired()]
    )
    submit = SubmitField('Sign Up')

    def validate_username(self, username):
        existing_user = User.query.filter_by(username=username.data).first()
        if existing_user:
            raise ValidationError('That username already exists. Please choose a different one.')

    def validate_password(self, password):
        pw = password.data
        if not re.search(r'[A-Z]', pw):
            raise ValidationError('Password must contain at least one uppercase letter.')
        if not re.search(r'[!.,?]', pw):
            raise ValidationError('Password must contain at least one special character (!, ., ,, ?).')
        if not re.search(r'\d', pw):
            raise ValidationError('Password must contain at least one number.')

# Sign In form
class SignInForm(FlaskForm):
    username = StringField(validators=[InputRequired(), Length(min=4, max=20)], render_kw={"placeholder": "Username"})
    password = PasswordField(validators=[InputRequired(), Length(min=8, max=20)], render_kw={"placeholder": "Password"})
    role = SelectField(choices=[('Team Member', 'Team Member'), ('Project Manager', 'Project Manager')], validators=[InputRequired()])
    submit = SubmitField('Sign In')
```

Figure 6: Auth Form Classes

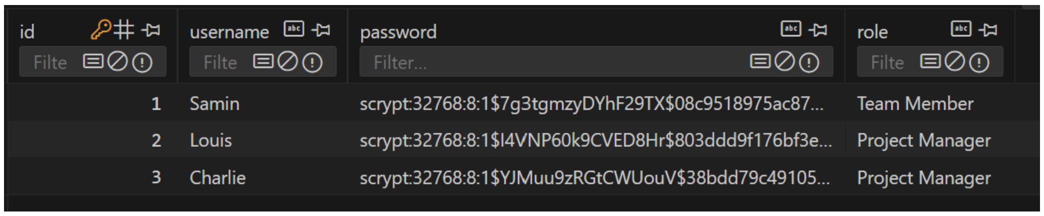


```python
# Sign Up
@app.route('/Sign-Up', methods=['GET', 'POST'])
def signup():
    form = SignUpForm()
    if form.validate_on_submit():
        existing_user = User.query.filter_by(username=form.username.data).first()
        if existing_user:
            flash("That username already exists. Please choose a different one.", "danger")
            return render_template('sign_up.html', form=form)

        hashed_password = generate_password_hash(form.password.data)
        new_user = User(username=form.username.data, password=hashed_password, role=form.role.data)
        db.session.add(new_user)
        db.session.commit()
        return redirect(url_for('signin'))
    return render_template('sign_up.html', form=form)

# Sign In
@app.route('/Sign-In', methods=['GET', 'POST'])
def signin():
    form = SignInForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user and check_password_hash(user.password, form.password.data):
            if user.role == form.role.data:
                login_user(user)
                return redirect(url_for('dashboard'))
            else:
                flash("Role mismatch. Please select the correct role.", "danger")
        else:
            flash("Invalid username or password.", "danger")
    return render_template('sign_in.html', form=form)
```

Figure 7: Auth Route Logic

The final GUI for both the authentication pages was designed with a clean and modern aesthetic to enhance visual appeal and ensure a user-friendly experience. To use the website, users initially have to enter the `http://127.0.0.1:5000`, where they will be greeted with the Sign In page. If they already have an account, they can proceed by entering their login credential. Otherwise, they will have to create one by clicking the "Sign Up" link below to be redirected to the Sign Up page.

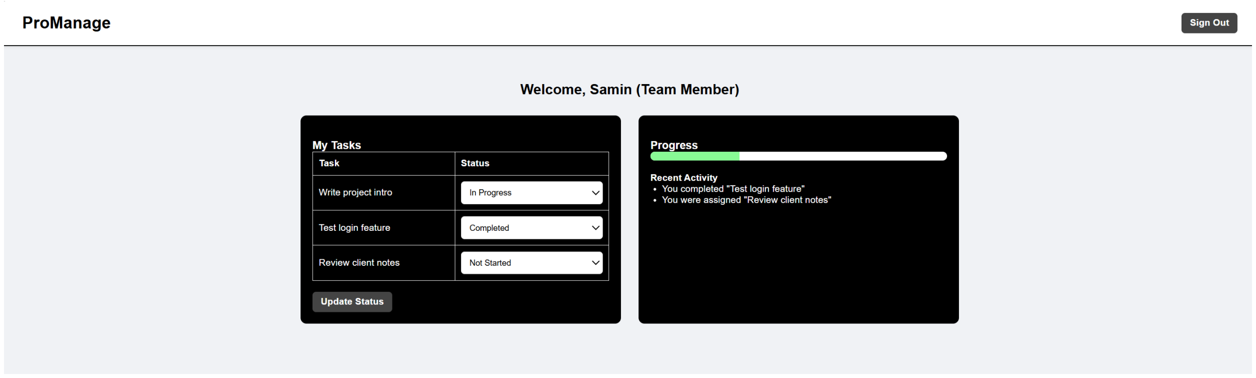
The Sign Up page enables users to select a role and enter a username and password that meet specific validation criteria. Upon successful submission, the account information is safely stored in the database through SQLAlchemy, with passwords hashed to ensure confidentiality. If any field is incomplete or fails validation, the form prevents submission and provides a clear error message to guide the user. As shown in Figure 8, each registered user is recorded in the database along with their role and encrypted credentials. Once registered, users can return to the Sign In page and access the system using their newly created login details.



id	username	password	role
1	Samin	scrypt:32768:8:1$7g3tgmzyDYhF29TX$08c9518975ac87...	Team Member
2	Louis	scrypt:32768:8:1$I4VNP60k9CVED8Hr$803ddd9f176bf3e...	Project Manager
3	Charlie	scrypt:32768:8:1$YJMuu9zRGtCWUouV$38bdd79c49105...	Project Manager

Figure 8: User Records Stored in Database

4.2 Team Dashboard



ProManage

Sign Out

Welcome, Samin (Team Member)

My Tasks

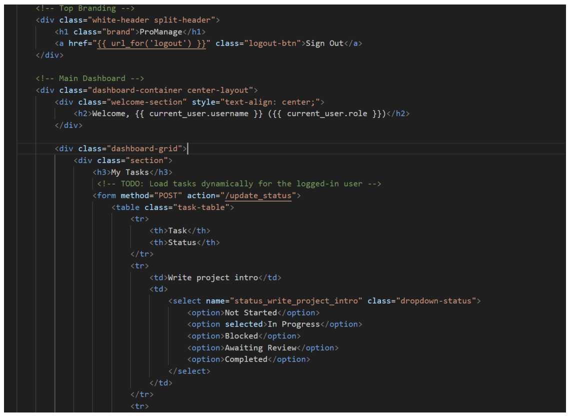
Task	Status
Write project intro	In Progress
Test login feature	Completed
Review client notes	Not Started

Update Status

Progress

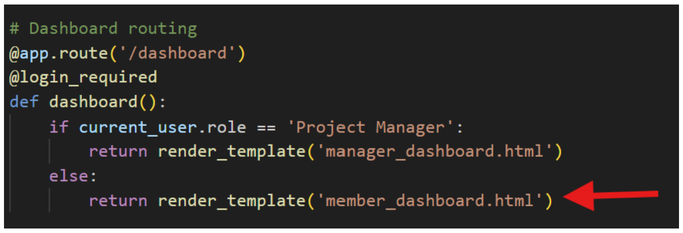
Recent Activity
- You completed "Test login feature"
- You were assigned "Review client notes"

Figure 9: Team Member Dashboard



```
<!-- Top Branding -->
<div class="white-header split-header">
    <h1 class="brand">ProManage</h1>
    <a href="{{ url_for('logout') }}" class="logout-btn">Sign Out</a>
</div>

<!-- Main Dashboard -->
<div class="dashboard-container center-layout">
    <div class="welcome-section" style="text-align: center;">
        <h2>Welcome, {{ current_user.username }} ({{ current_user.role }})</h2>
    </div>

    <div class="dashboard-grid">
        <div class="section">
            <h3>My Tasks</h3>
            <!-- TODO: Load tasks dynamically for the logged-in user -->
            <form method="POST" action="/update_status">
                <table class="task-table">
                    <tr>
                        <th>Task</th>
                        <th>Status</th>
                    </tr>
                    <tr>
                        <td>Write project intro</td>
                        <td>
                            <select name="status_write_project_intro" class="dropdown-status">
                                <option>Not Started</option>
                                <option selected>In Progress</option>
                                <option>Blocked</option>
                                <option>Awaiting Review</option>
                                <option>Completed</option>
                            </select>
                        </td>
                    </tr>
                    <tr>
```

Figure 10: Auth Form Classes

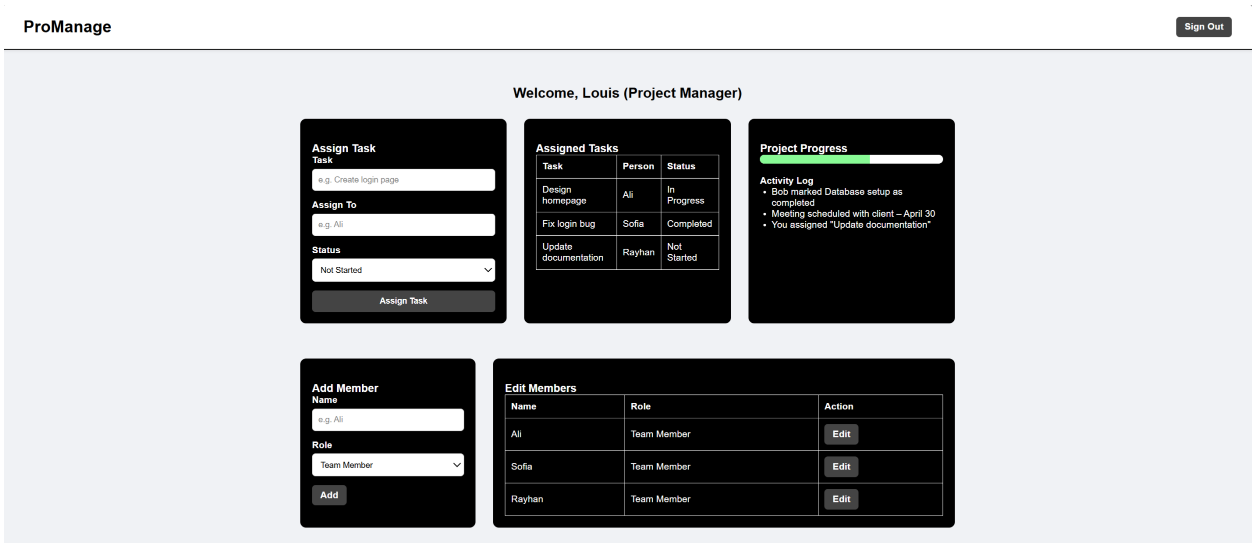


```
# Dashboard routing
@app.route('/dashboard')
@login_required
def dashboard():
    if current_user.role == 'Project Manager':
        return render_template('manager_dashboard.html')
    else:
        return render_template('member_dashboard.html')
```

Figure 11: Auth Route Logic

The Team Member Dashboard provides users with a lightweight design, allowing them to quickly learn to use key features such as tracking assigned tasks and updating their progress. As shown in Figures 6 and 9, tasks are displayed in a table format with dropdowns allowing users to modify the status of each task. This input is handled via form elements embedded in the HTML template (Figure 10), which sends data to the back-end for processing. Access to this page is determined by the user's role, as controlled by the Flask routing logic shown in Figure 11. If the logged-in user is not a project manager, the application automatically renders the member dashboard, ensuring role-based navigation is correctly enforced.

4.3 Manager Dashboard



ProManage

Sign Out

Welcome, Louis (Project Manager)

Assign Task

Task: e.g. Create login page

Assign To: e.g. Ali

Status: Not Started

Assign Task

Assigned Tasks

Task	Person	Status
Design homepage	Ali	In Progress
Fix login bug	Sofia	Completed
Update documentation	Rayhan	Not Started

Project Progress

Activity Log
- Bob marked Database setup as completed
- Meeting scheduled with client – April 30
- You assigned "Update documentation"

Add Member

Name: e.g. Ali

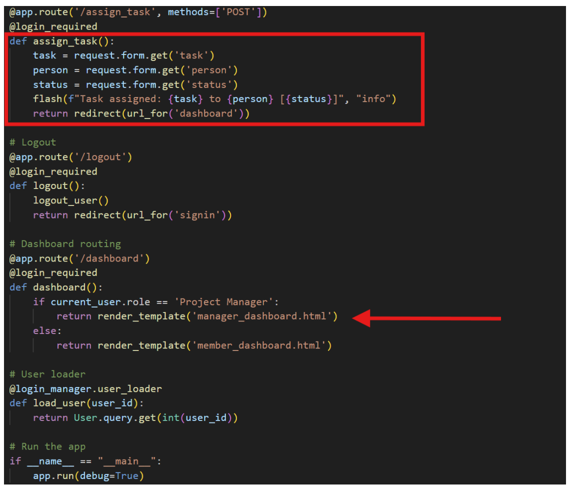
Role: Team Member

Add

Edit Members

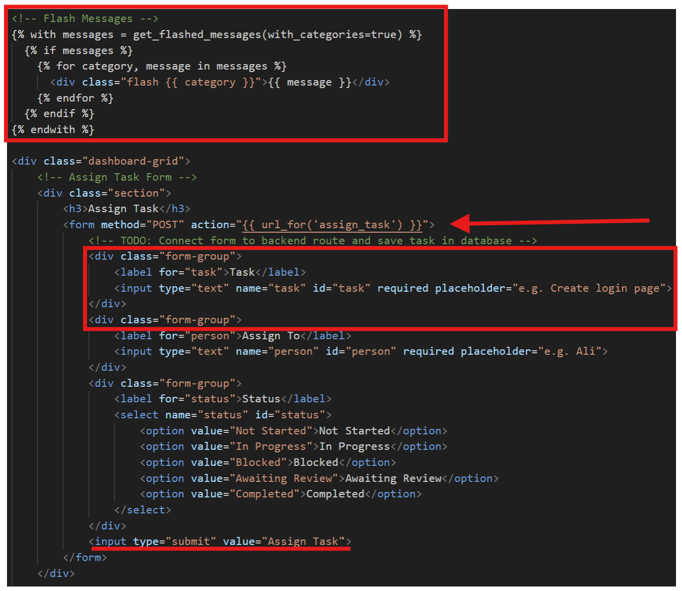
Name	Role	Action
Ali	Team Member	Edit
Sofia	Team Member	Edit
Rayhan	Team Member	Edit

Figure 12: Manager Dashboard



```python
@app.route('/assign_task', methods=['POST'])
@login_required
def assign_task():
    task = request.form.get('task')
    person = request.form.get('person')
    status = request.form.get('status')
    flash(f"Task assigned: {task} to {person} [{status}]", "info")
    return redirect(url_for('dashboard'))

# Logout
@app.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('signin'))

# Dashboard routing
@app.route('/dashboard')
@login_required
def dashboard():
    if current_user.role == 'Project Manager':
        return render_template('manager_dashboard.html')
    else:
        return render_template('member_dashboard.html')

# User loader
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

# Run the app
if __name__ == "__main__":
    app.run(debug=True)
```

Figure 13: Manager Route Logic



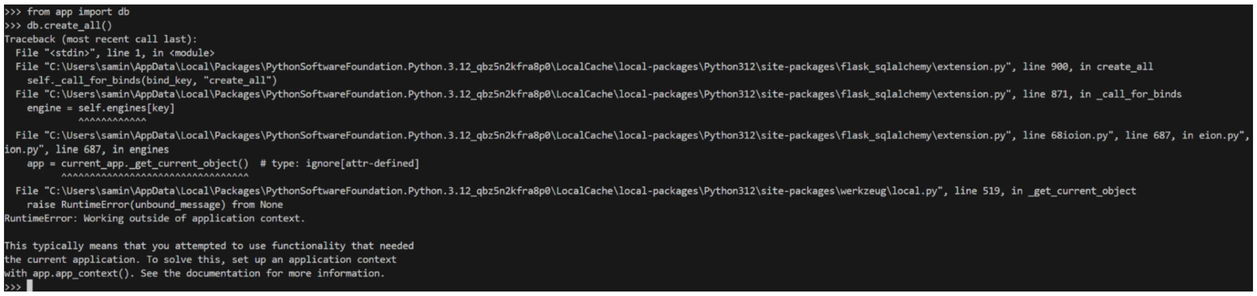
```html
<!-- Flash Messages -->
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    {% for category, message in messages %}
      <div class="flash {{ category }}">{{ message }}</div>
    {% endfor %}
  {% endif %}
{% endwith %}

<div class="dashboard-grid">
    <!-- Assign Task Form -->
    <div class="section">
        <h3>Assign Task</h3>
        <form method="POST" action="{{ url_for('assign_task') }}">
            <!-- TODO: Connect form to backend route and save task in database -->
            <div class="form-group">
                <label for="task">Task</label>
                <input type="text" name="task" id="task" required placeholder="e.g. Create login page">
            </div>
            <div class="form-group">
                <label for="person">Assign To</label>
                <input type="text" name="person" id="person" required placeholder="e.g. Ali">
            </div>
            <div class="form-group">
                <label for="status">Status</label>
                <select name="status" id="status">
                    <option value="Not Started">Not Started</option>
                    <option value="In Progress">In Progress</option>
                    <option value="Blocked">Blocked</option>
                    <option value="Awaiting Review">Awaiting Review</option>
                    <option value="Completed">Completed</option>
                </select>
            </div>
            <input type="submit" value="Assign Task">
        </form>
    </div>
```

Figure 14: Manager Form HTML

The Manager Dashboard offers a more advanced interface compared to the Team Member view, granting the project manager control over task assignment, progress monitoring, and team member management. As shown in Figure 12, managers can assign tasks by specifying a description, assigning it to a team member, and selecting its status. Figure 13 demonstrates how Flask routes handle these actions on the back end, ensuring that only authenticated users with the `"Project Manager"` role access this page. Figure 14 highlights the HTML structure responsible for capturing and processing task assignment inputs, along with the use of flash messages to confirm user actions. While the tasks and members are currently hard-coded, the structure is designed for dynamic integration with a database in future development.

5 Development Challenges and Solutions

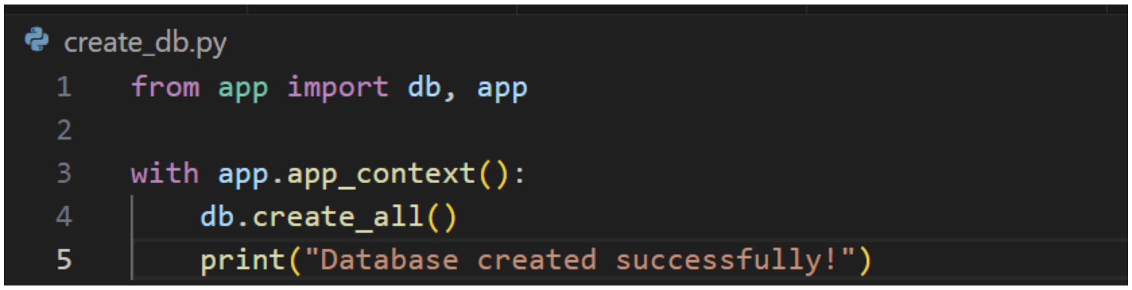
This section covers the challenges faced during the development process of **ProManage**, and how the solutions were implemented to solve the problem.



```
>>> from app import db
>>> db.create_all()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\samin\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\site-packages\flask_sqlalchemy\extension.py", line 900, in create_all
    self._call_for_binds(bind_key, "create_all")
  File "C:\Users\samin\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\site-packages\flask_sqlalchemy\extension.py", line 871, in _call_for_binds
    engine = self.engines[key]
             ^^^^^^^^^^^^

  File "C:\Users\samin\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\site-packages\flask_sqlalchemy\extension.py", line 68ioion.py", line 687, in eion.py",
ion.py", line 687, in engines
    app = current_app._get_current_object()  # type: ignore[attr-defined]
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "C:\Users\samin\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\site-packages\werkzeug\local.py", line 519, in _get_current_object
    raise RuntimeError(unbound_message) from None
RuntimeError: Working outside of application context.

This typically means that you attempted to use functionality that needed
the current application. To solve this, set up an application context
with app.app_context(). See the documentation for more information.
>>> 
```

Figure 15: Database Creation Error

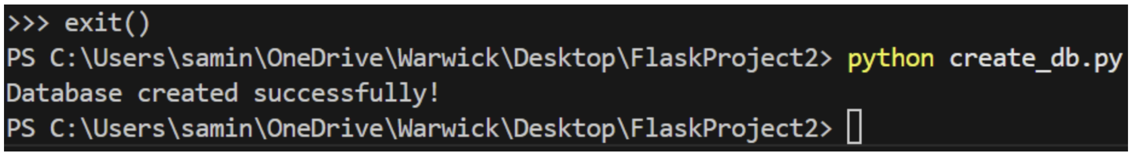
An attempt was made to create the database directly from the terminal using `from app import db` followed by `db.create_all()`. This resulted in a `RuntimeError` due to the absence of an active Flask application context, which is required for proper access to configuration and database binding, as shown in Figure 15.



create_db.py

```python
from app import db, app

with app.app_context():
    db.create_all()
    print("Database created successfully!")
```

Figure 16: Fixing the Error with Context Script (Code with Josh, 2023)



```
>>> exit()
PS C:\Users\samin\OneDrive\Warwick\Desktop\FlaskProject2> python create_db.py
Database created successfully!
PS C:\Users\samin\OneDrive\Warwick\Desktop\FlaskProject2> 
```

Figure 17: Successful Database Creation

To resolve the issue, a separate script named `create_db.py` was created to ensure the database initialisation occurred within the correct Flask application context. This script explicitly wrapped the `db.create_all()` function within `app.app_context()`, allowing the database to be created without errors. When executed through the terminal using `python create_db.py`, the process completed successfully, confirming that the context requirement had been properly handled, as demonstrated in Figures 16 and 17.

6 Testing

This section evaluates the program's robustness and functionality by interacting with the web application as a typical end user. The aim of these tests was to ensure all core front-end features work as expected, including navigation, form validation, role-based access, and visual responsiveness.

Test Case	**Test Type**	**Input**	**Expected Outcome**	**Actual Outcome**	**Pass/Fail**
1. Valid Sign In	Normal	Valid username, password, role	Redirects to correct dashboard	Figure 18,19	Pass
2. Wrong Password	Invalid	Correct username and role, wrong password	Displays `"Invalid username or password"`	Figure 20	Pass
3. Wrong Username	Invalid	Non-existent username	Displays `"Invalid username or password"`	Figure 21	Pass
4. Mismatched Role	Invalid	Valid credentials, wrong role	Displays `"Role mismatch"` error	Figure 22	Pass
5. Weak Sign Up	Invalid	Valid username, weak password	Displays `"Password must contain..."`	Figure 23	Pass
6. Duplicate Username	Invalid	Existing username	Displays `"Username already exists"`	Figure 23	Pass
7. Empty Fields	Erroneous	Blank fields	Displays field validation errors	Figure 24,25	Pass
8. Page Redirect	Normal	Login as manager/member	Loads correct dashboard page	Figure 26,27	Pass
9. Sign Out	Normal	Click Sign Out	Returns to Sign In page	Figure 28	Pass

Table 2: Test Cases

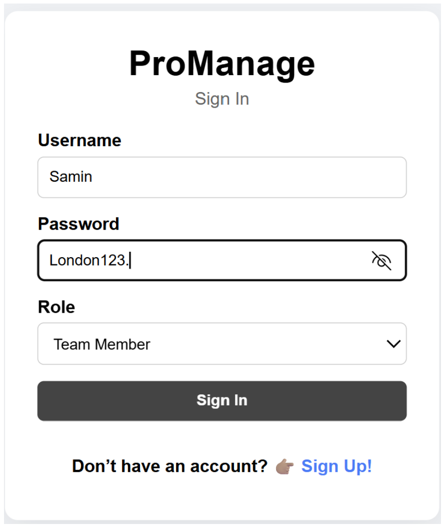
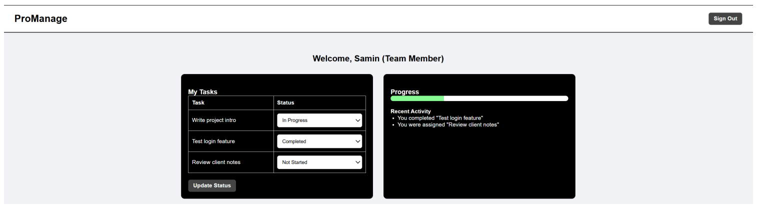

ProManage
Sign In
Username: Samin
Password: London123
Role: Team Member

Figure 18: Valid Sign In Form Submission


ProManage
Welcome, Samin (Team Member)

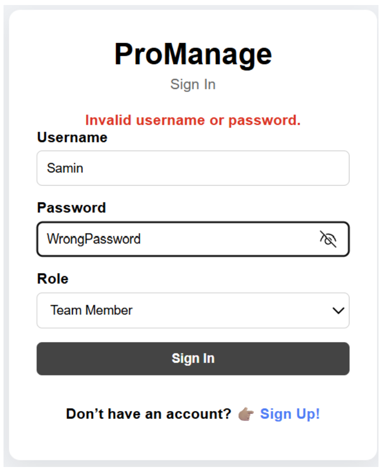
My Tasks

Task	Status
Write project intro	In Progress
Test login feature	Completed
Review client notes	Not Started

Progress

Recent Activity
- You completed "Test login feature"
- You were assigned "Review client notes"

Figure 19: Successful Redirection to Dashboard



ProManage

Sign In

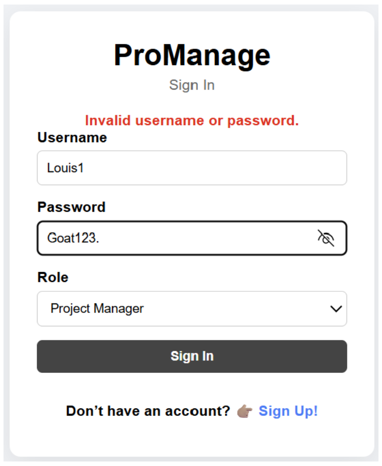
Invalid username or password.

Username: Samin

Password: WrongPassword

Role: Team Member

Figure 20: Error Message for Wrong Password



ProManage

Sign In

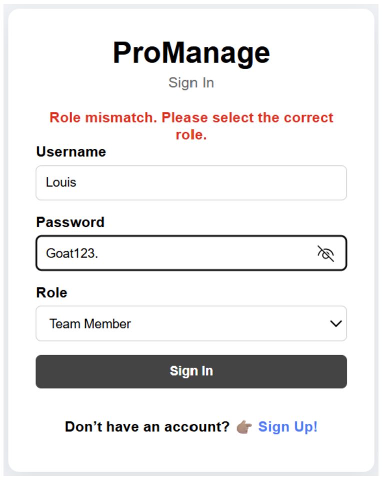
Invalid username or password.

Username: Louis1

Password: Goat123.

Role: Project Manager

Figure 21: Error Message for Wrong Username



ProManage

Sign In

Role mismatch. Please select the correct role.

Username: Louis

Password: Goat123.

Role: Team Member

Figure 22: Error Message for Wrong Role

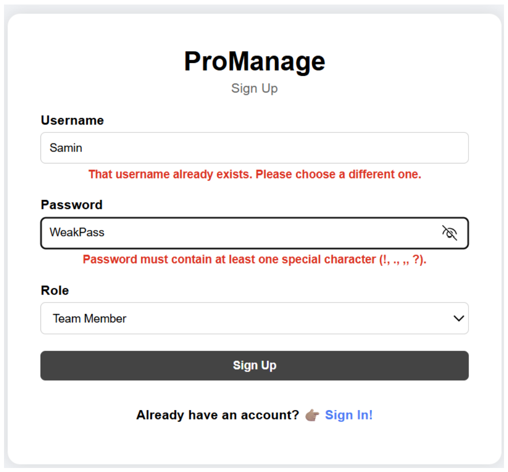

ProManage
Sign Up
Username
Samin
That username already exists. Please choose a different one.
Password
WeakPass
Password must contain at least one special character (!, ., ,, ?).
Role
Team Member
Sign Up
Already have an account? Sign In!

Figure 23: Validation Errors for Duplicate Username and Weak Password

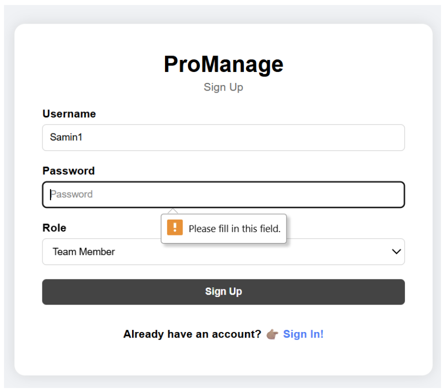


Figure 24: Validation warning for missing password field in the Sign-Up form.

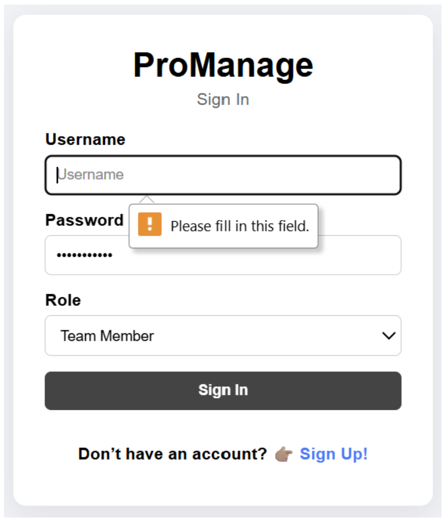
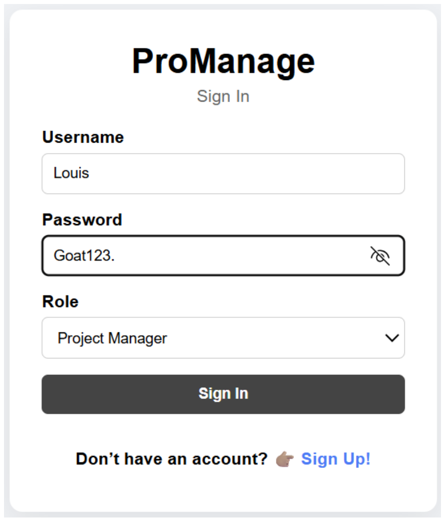


Figure 25: Validation warning for missing password field in the Sign-Up form.



ProManage

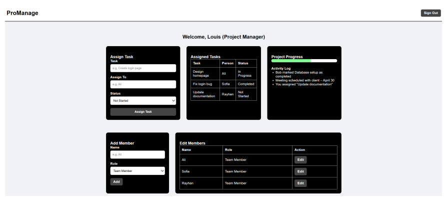
Sign In

Username: Louis

Password: Goat123.

Role: Project Manager

Figure 26: Valid Sign In as Manager



ProManage

Welcome, Louis (Project Manager)

Assign Task

Task: e.g. Create login page

Assign To: e.g. Ali

Status: Not Started

Assigned Tasks

Task	Person	Status
Design homepage	Ali	In Progress
Fix login bug	Sofia	Completed
Update documentation	Rayhan	Not Started

Project Progress

Activity Log
- Bob marked Database setup as completed
- Meeting scheduled with client – April 30
- You assigned "Update documentation"

Add Member

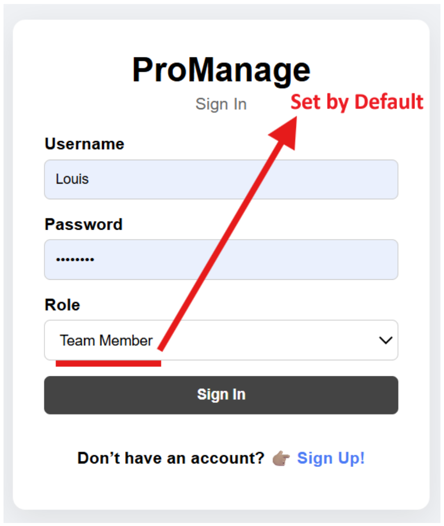
Name: e.g. Ali

Role: Team Member

Edit Members

Name	Role	Action
Ali	Team Member	Edit
Sofia	Team Member	Edit
Rayhan	Team Member	Edit

Figure 27: Successful Redirection to Manager Dashboard



ProManage

Sign In

Set by Default

Username: Louis

Password: ••••••••

Role: Team Member

Figure 28: Signed out successfully

7 Evaluation

Even though the front-end met all the required criteria, it had some limitations. One of them being the lack of 2-factor authentication, which reduced security severely for a modern-based web page storing user credentials. Secondly, it had no forgotten password feature to recover your account, decreasing its usability. This is because it forces users to create a new account in order to continue using the website's services, which can lead to frustration and discourage continued use.

8 Conclusion

The report covered the design, development, and testing of ProManage's front-end functionality and ended with a list of improvements that could be implemented for future iterations. Together with the submitted code, it shows how Flask, HTML, and CSS were used to create a working web application.

9 References

- Code with Josh (2023) *Python Authentication: Create a Secure Login System with Flask.* Published on 13 March 2023. Available at: `https://youtu.be/71EU8gnZqZQ?si=Hp2ES6LEU_uODXWF` (Accessed: 10 May 2025).
- Arpan Neupane (2024) *Python Flask Authentication Tutorial – Learn Flask Login.* Published approximately 11 months ago. Available at: `https://youtu.be/Fr2MxT9M0V4?si=vshmKtpLvsjNOKjz` (Accessed: 10 May 2025).

ProManage Web-based Project Management Application

Student ID: 5614928 (Project Partner: 5653630), May 2025

Software Development and Security (WM145-24), WMG, University of Warwick

Contents

1. Introduction

This project is a task management application with user authentication, role-based access (project manager vs. team member), and activity tracking. This report focuses on the back end, specifically the database components. It explains how tasks, users, and activity logs are managed through Flask and SQLAlchemy.

2. Overview of the Application Architecture

2.1. Front End

The /templates/ folder is front end core, containing HTML files to structure the web pages. Separate HTML files were created for the project manager and team member dashboards, as well as the signing-in and signing-up pages. Python Flask links to the HTML file using render_template(), calling the correct page when the browser accesses a specific route. HTML uses Flask's url_for() to execute other Python routes and their route-specific functions. /static/ holds a CSS file that improves layout and appearance. For instance, it styles flash() messages differently for errors and general information.

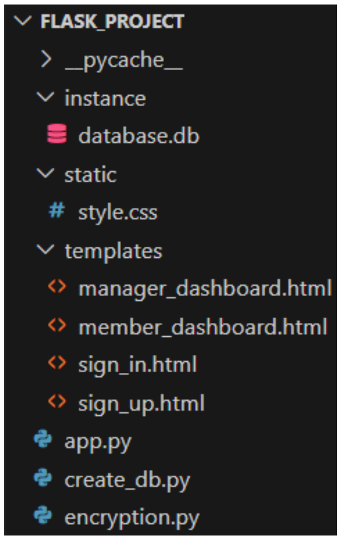


Figure 1: Project files.

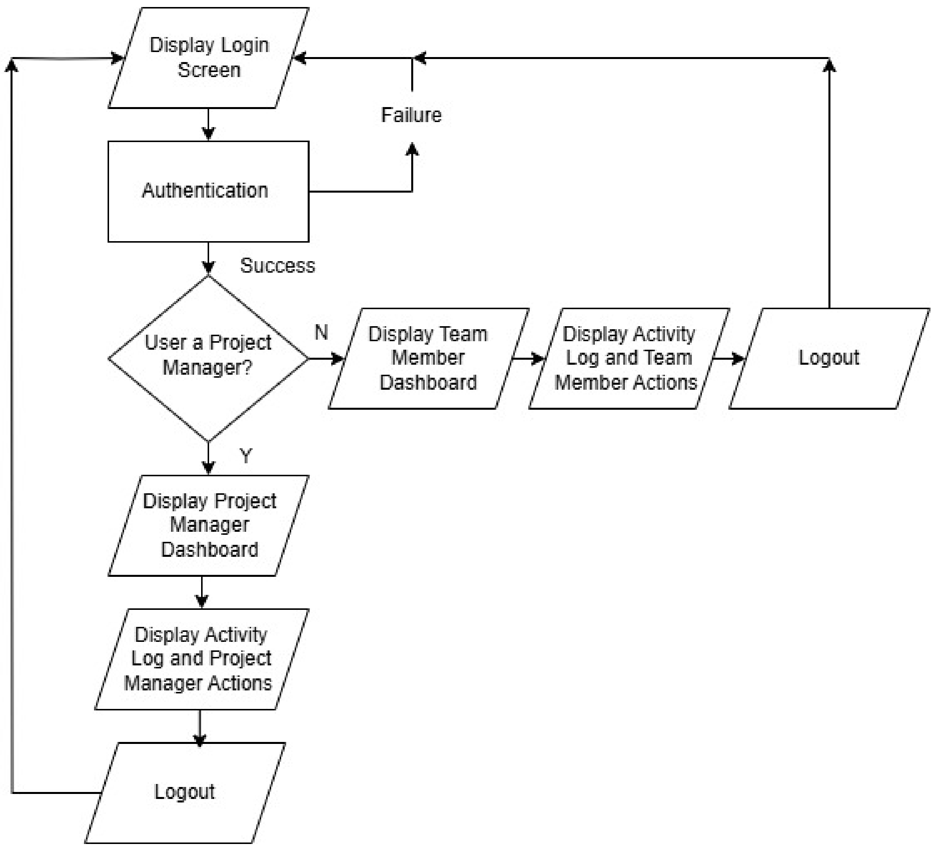

Display Login Screen
Failure
Authentication
Success
User a Project Manager?
N
Display Team Member Dashboard
Display Activity Log and Team Member Actions
Logout
Y
Display Project Manager Dashboard
Display Activity Log and Project Manager Actions
Logout

Figure 2: Main front end logic.

2.2. Back End

The back end is composed of three Python files implemented in Flask, a lightweight web framework for routing between pages and rendering HTML templates.

The main file, app.py, is responsible for:

- Creating ‘app’, the Flask object.
- Defining User, Task, and Activity models (database tables).
- Setting up routes and linking them to HTML templates.

create_db.py generates database.db (the database file) if it does not exist and must be run before app.py when running the application for the first time. It creates two default users: a project manager and team member. encryption.py generates a unique encryption key using the cryptography library. app.py imports this key, storing it in the user’s database row.

The database is created with SQLAlchemy and stored in an SQLite file. SQLAlchemy uses Python classes (models) to represent database tables with attributes such as columns (fields) and foreign keys which define relationships. For example, the User class represents user objects with fields: ID, username, password hash, role, and encryption key. SQLAlchemy uses ORM methods like .add(), .get(), and .commit() to interact with the SQLite database through Python code.

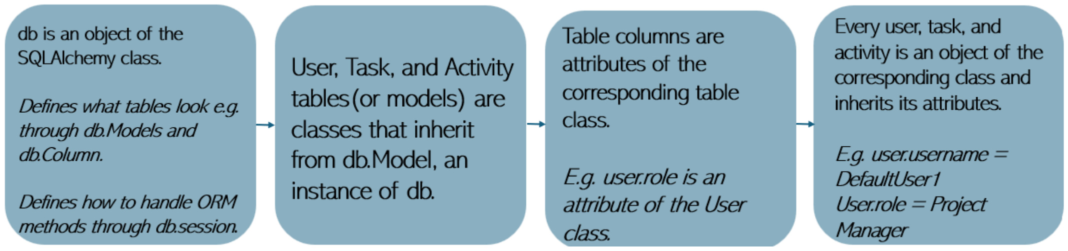

db is an object of the SQLAlchemy class.
Defines what tables look e.g. through db.Models and db.Column.
Defines how to handle ORM methods through db.session.
User, Task, and Activity tables (or models) are classes that inherit from db.Model, an instance of db.
Table columns are attributes of the corresponding table class.
E.g. user.role is an attribute of the User class.
Every user, task, and activity is an object of the corresponding class and inherits its attributes.
E.g. user.username = DefaultUser1
User.role = Project Manager

Figure 3: SQLAlchemy’s database logic summarised.

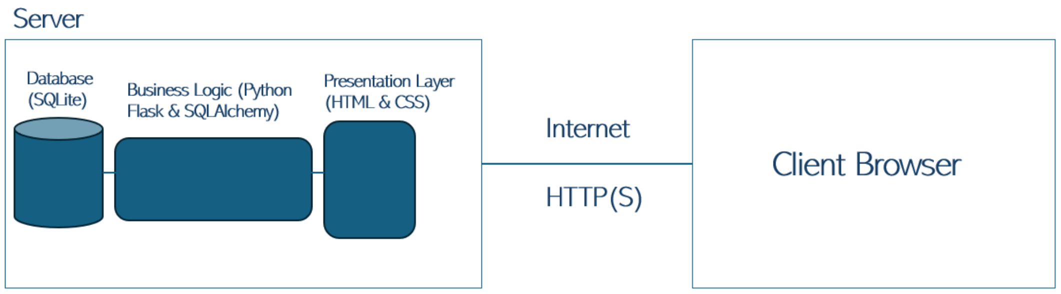

Server
Database (SQLite)
Business Logic (Python Flask & SQLAlchemy)
Presentation Layer (HTML & CSS)
Internet
HTTP(S)
Client Browser

Figure 4: Application architecture diagram.

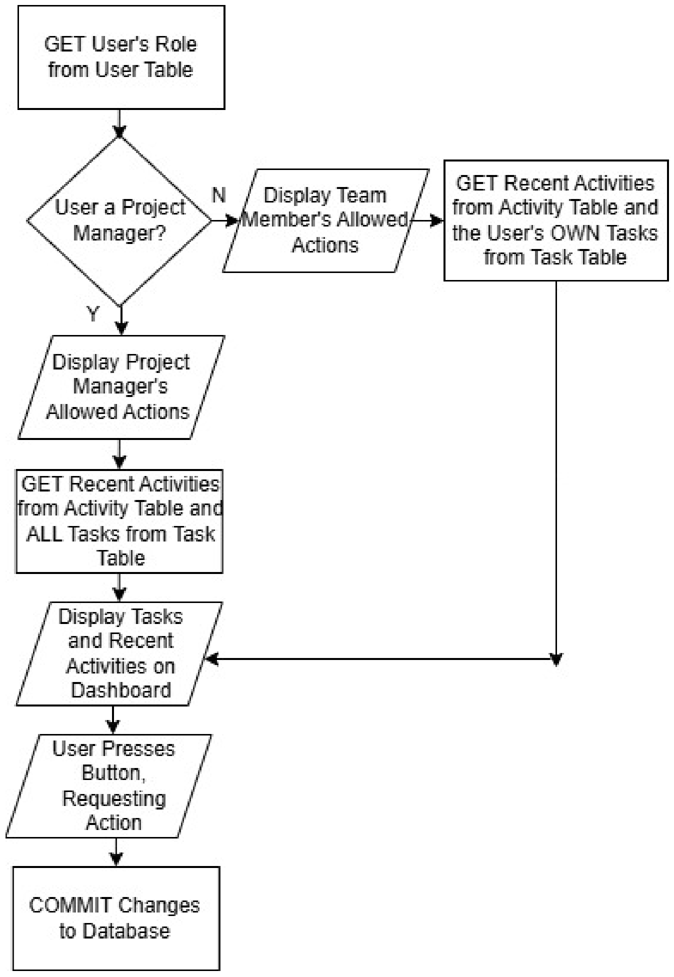
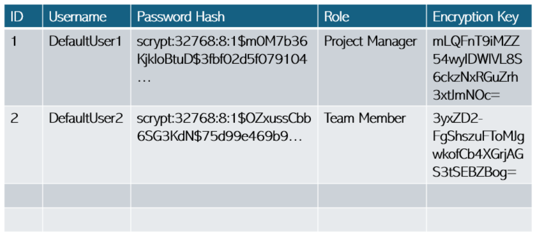

GET User's Role from User Table
User a Project Manager?
N
Y
Display Team Member's Allowed Actions
GET Recent Activities from Activity Table and the User's OWN Tasks from Task Table
Display Project Manager's Allowed Actions
GET Recent Activities from Activity Table and ALL Tasks from Task Table
Display Tasks and Recent Activities on Dashboard
User Presses Button, Requesting Action
COMMIT Changes to Database

Figure 5: Main back end logic.

3. Explanation of Back-End Implemented Features

Back end features involve retrieving and updating table rows. To update rows, the user presses a button, calling a route in app.py that updates the relevant data. For instance, a project manager editing a team member updates the member's row in the User table. Web pages display database data by receiving them as variables from Flask. Flask queries the table and sets the result equal to a variable. It passes the variable to the corresponding HTML template as an argument in render_template(). The client's browser uses GET requests to retrieve data from the Flask server, and POST requests to send.

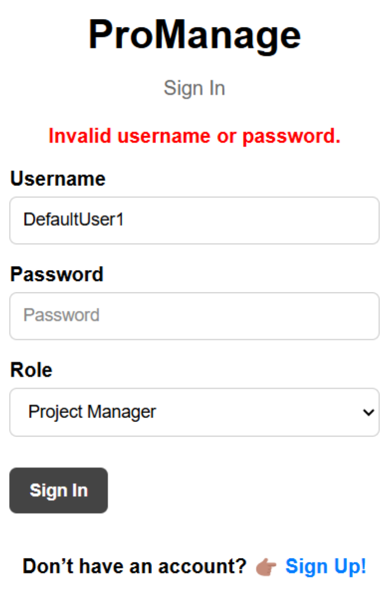


ID	Username	Password Hash	Role	Encryption Key
1	DefaultUser1	scrypt:32768:8:1$m0M7b36 KjkloBtuD$3fbf02d5f079104 …	Project Manager	mLQFnT9iMZZ 54wyIDWIVL8S 6ckzNxRGuZrh 3xtJmNOc=
2	DefaultUser2	scrypt:32768:8:1$OZxussCbb 6SG3KdN$75d99e469b9…	Team Member	3yxZD2- FgShszuFToMJg wkofCb4XGrjAG S3tSEBZBog=

Figure 6: Example of the User table.

3.1. Logging In

When logging in, the user enters their username, password, and role for authentication. The /login route queries the User table using .filter_by(username=form.username.data) and .first() to find the entered username. If found, the inputted password is hashed, compared to the stored password hash, and the role is also compared. If they match, the user is redirected to their dashboard. If not, access is denied.



ProManage

Sign In

Invalid username or password.

Username: DefaultUser1

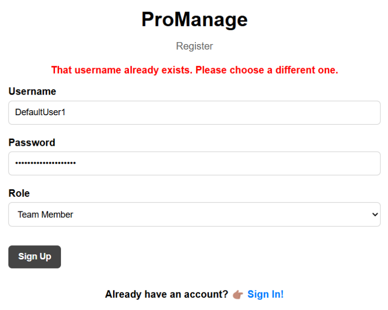
Password: Password

Role: Project Manager

Figure 7: Invalid Password Entered.

3.2. Signing Up

When creating an account, app.py queries the User table to check whether the username exists. If not, the password is hashed, an encryption key is generated, and, along with the username and role, they are all stored in a new row in the table using .add().


ProManage

Register

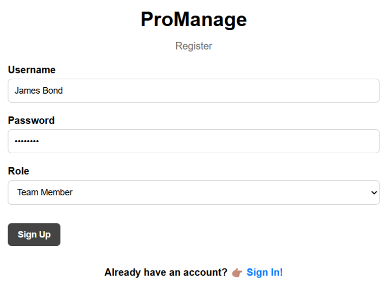
That username already exists. Please choose a different one.

Username: DefaultUser1

Password: ••••••••••••••••••••

Role: Team Member

Figure 8: Signing up with a username currently in use.


ProManage

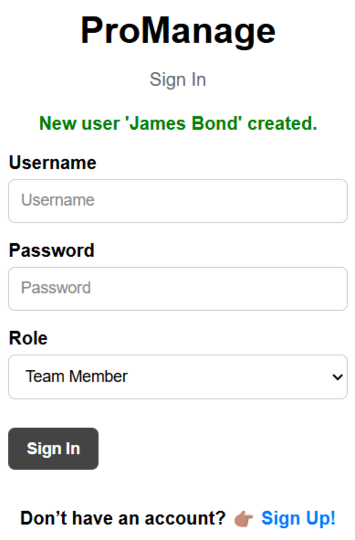
Register

Username: James Bond

Password: ••••••••

Role: Team Member

Figure 9: Signing up as a new user.


ProManage

Sign In

New user 'James Bond' created.

Username: Username

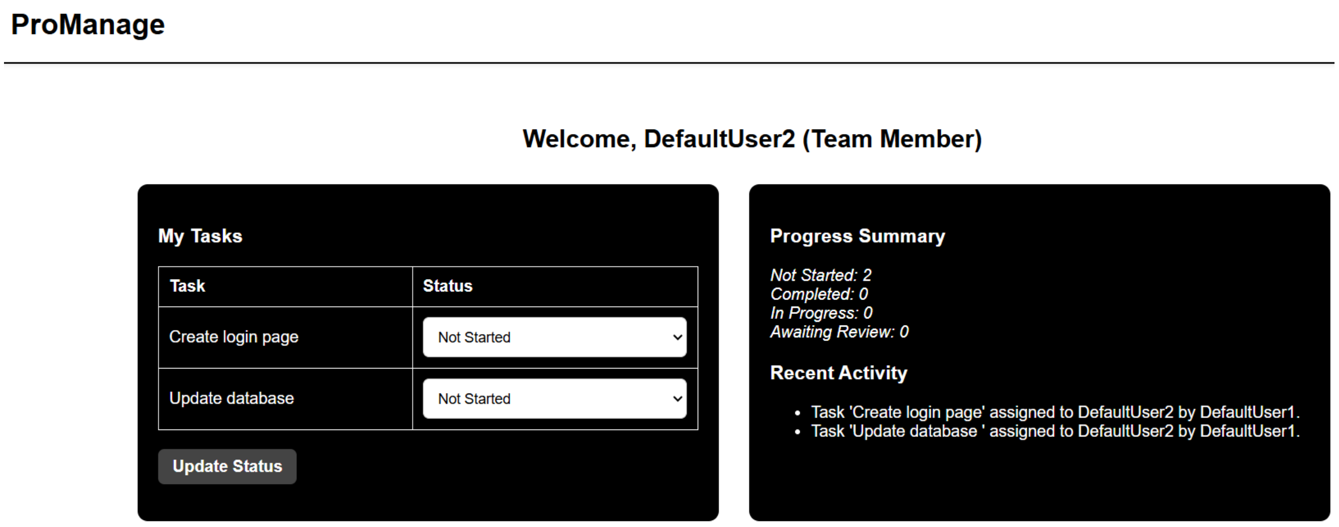
Password: Password

Role: Team Member

Figure 10: The username is not taken so the user was successfully created.

3.3. Team Member Actions

The /dashboard route ensures team members can only see their assigned tasks by querying the Tasks table with their user ID using .filter_by(assigned_to=current_user.id) and .all(). Team members can update task status by pressing 'Update Status'. This sends the task ID and new status to the /update_task route, updating the Task table.



ProManage

Welcome, DefaultUser2 (Team Member)

My Tasks

Task	Status
Create login page	Not Started
Update database	Not Started

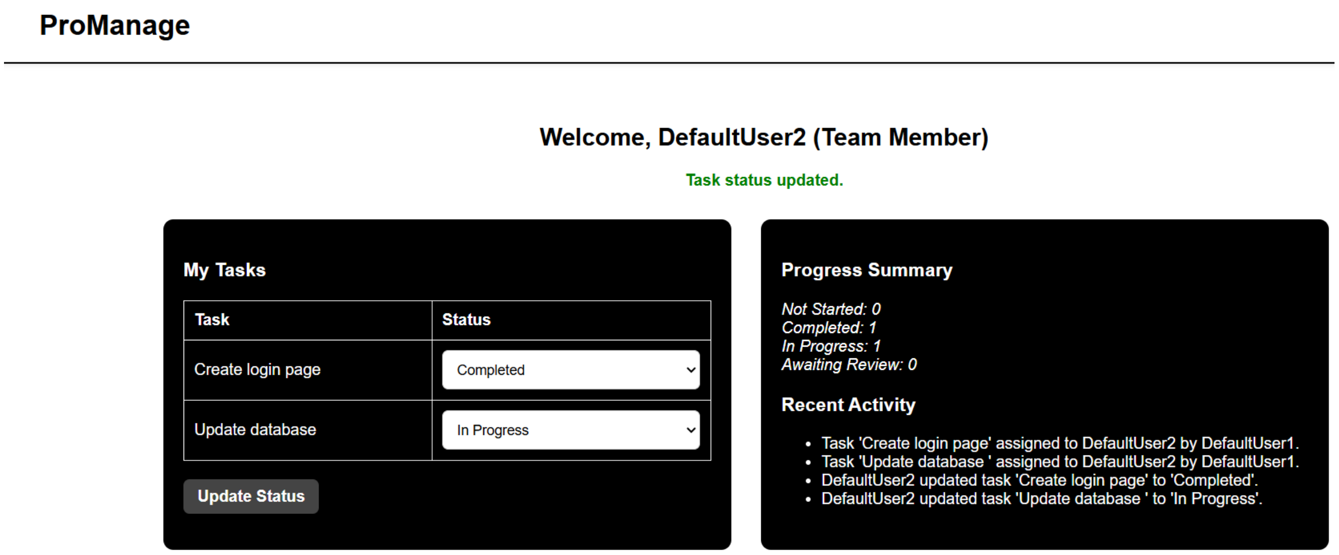
Progress Summary

Not Started: 2
Completed: 0
In Progress: 0
Awaiting Review: 0

Recent Activity

- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.

Figure 11: DefaultUser2's dashboard.



ProManage

Welcome, DefaultUser2 (Team Member)

Task status updated.

My Tasks

Task	Status
Create login page	Completed
Update database	In Progress

Progress Summary

Not Started: 0
Completed: 1
In Progress: 1
Awaiting Review: 0

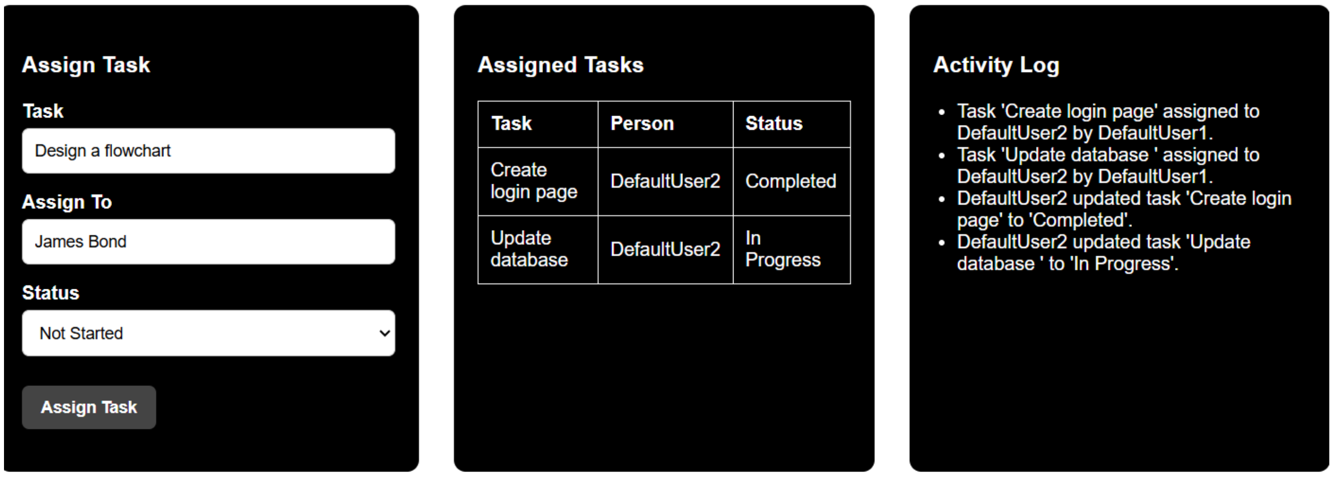
Recent Activity

- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.
- DefaultUser2 updated task 'Create login page' to 'Completed'.
- DefaultUser2 updated task 'Update database ' to 'In Progress'.

Figure 12: DefaultUser2 updated their tasks.

3.4. Project Manager Actions

Project managers can assign tasks by submitting a form to the /assign_task route, including a task description, assignee's name, and status. The User table is queried to confirm the assignee exists. If so, the new task is added to the Task table by .add().



Assign Task

Task: Design a flowchart

Assign To: James Bond

Status: Not Started

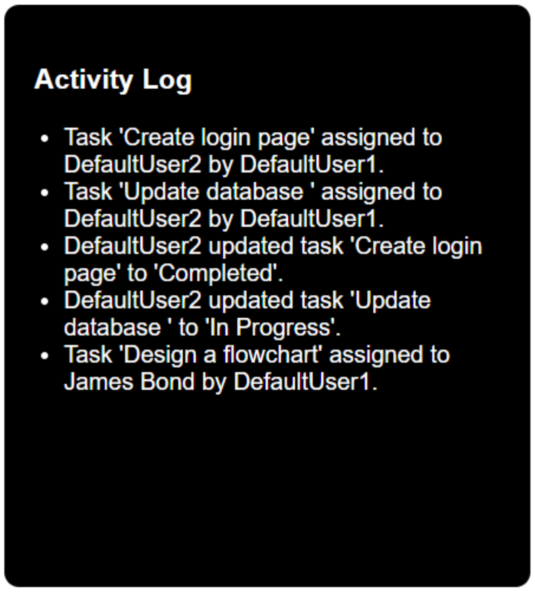
Assigned Tasks

Task	Person	Status
Create login page	DefaultUser2	Completed
Update database	DefaultUser2	In Progress

Activity Log

- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.
- DefaultUser2 updated task 'Create login page' to 'Completed'.
- DefaultUser2 updated task 'Update database ' to 'In Progress'.

Figure 13: Submitting the task form by pressing 'Assign Task'.

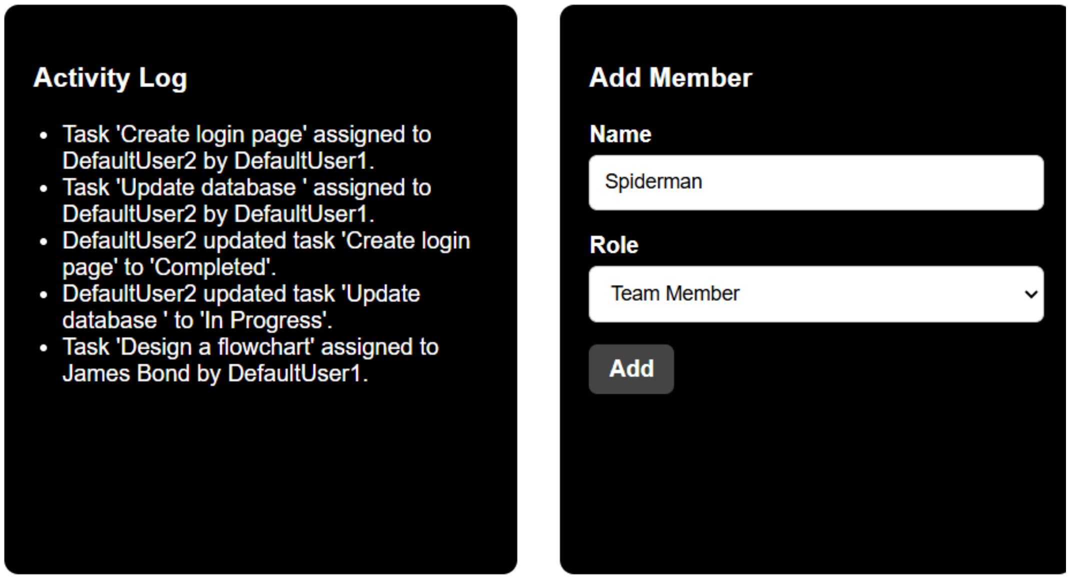


Activity Log

- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.
- DefaultUser2 updated task 'Create login page' to 'Completed'.
- DefaultUser2 updated task 'Update database ' to 'In Progress'.
- Task 'Design a flowchart' assigned to James Bond by DefaultUser1.

Figure 14: Task successfully assigned to James Bond.

Managers can add, edit, and delete team members. To add, a form is submitted and a new User object is created with .add(). To edit, the form sends the user ID, and app.py uses .get() to find and update the user's record in the User table. To delete, the user ID is sent, their tasks are removed, and the user is deleted with .delete().



Activity Log

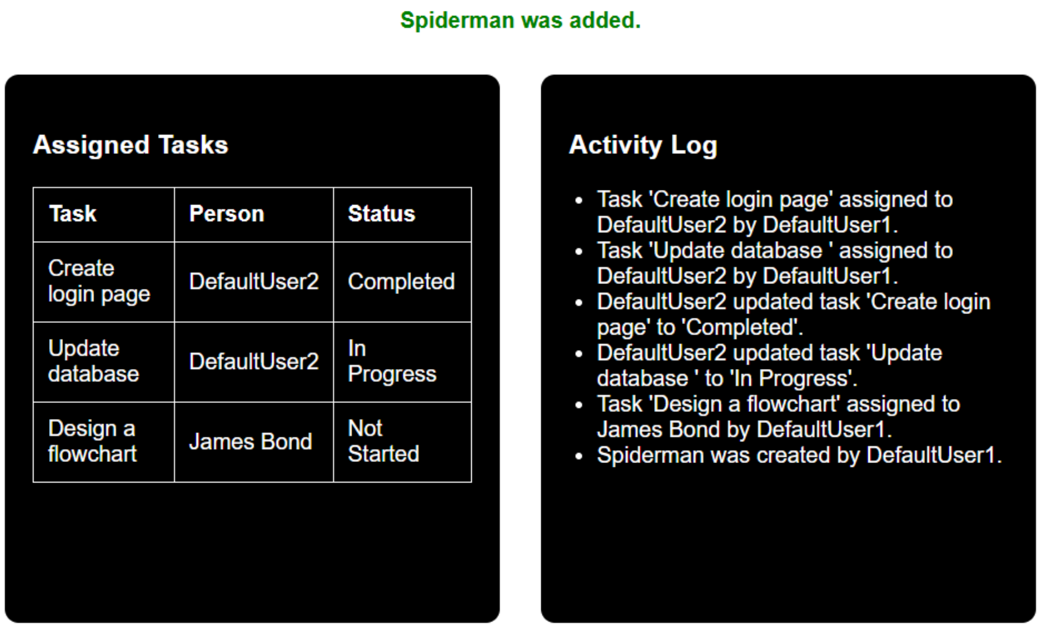
- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.
- DefaultUser2 updated task 'Create login page' to 'Completed'.
- DefaultUser2 updated task 'Update database ' to 'In Progress'.
- Task 'Design a flowchart' assigned to James Bond by DefaultUser1.

Add Member

Name: Spiderman

Role: Team Member

Figure 15: Adding a new user.



Spiderman was added.

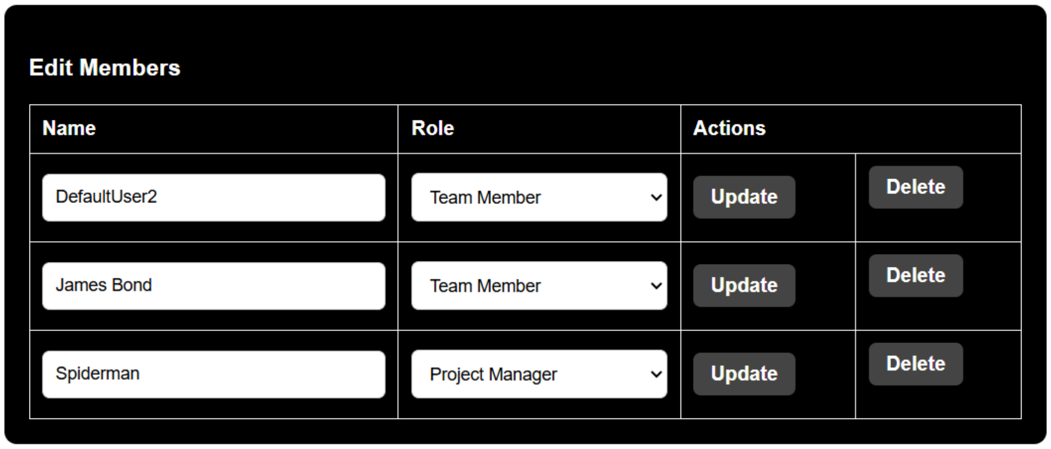
Assigned Tasks

Task	Person	Status
Create login page	DefaultUser2	Completed
Update database	DefaultUser2	In Progress
Design a flowchart	James Bond	Not Started

Activity Log

- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.
- DefaultUser2 updated task 'Create login page' to 'Completed'.
- DefaultUser2 updated task 'Update database ' to 'In Progress'.
- Task 'Design a flowchart' assigned to James Bond by DefaultUser1.
- Spiderman was created by DefaultUser1.

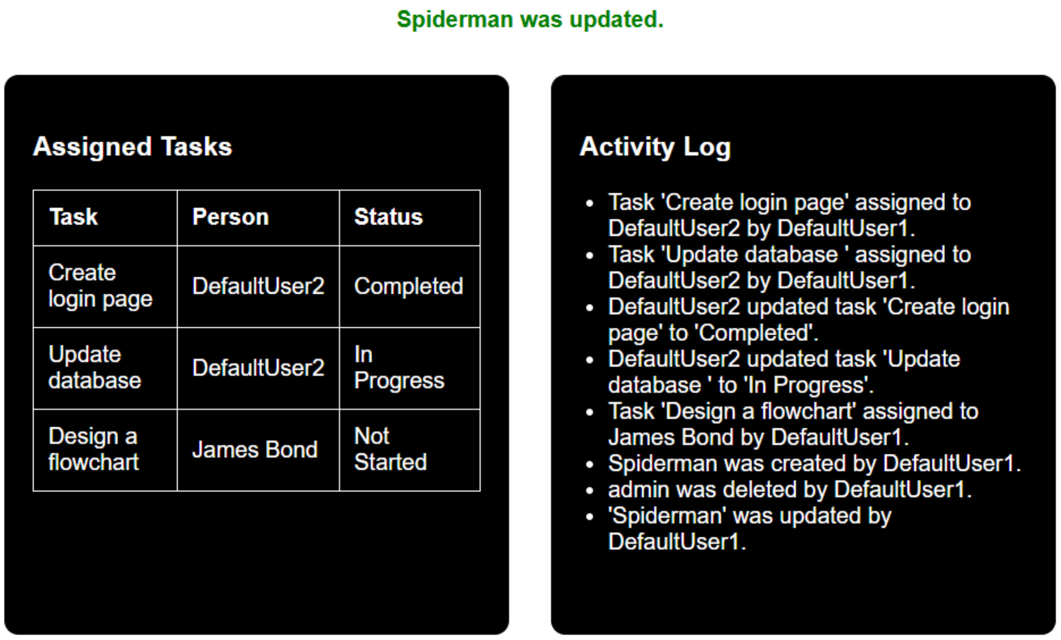
Figure 16: New user was successfully added.



Edit Members

Name	Role	Actions	
DefaultUser2	Team Member	Update	Delete
James Bond	Team Member	Update	Delete
Spiderman	Project Manager	Update	Delete

Figure 17: Changing Spiderman to a project leader by pressing 'Update'.



Spiderman was updated.

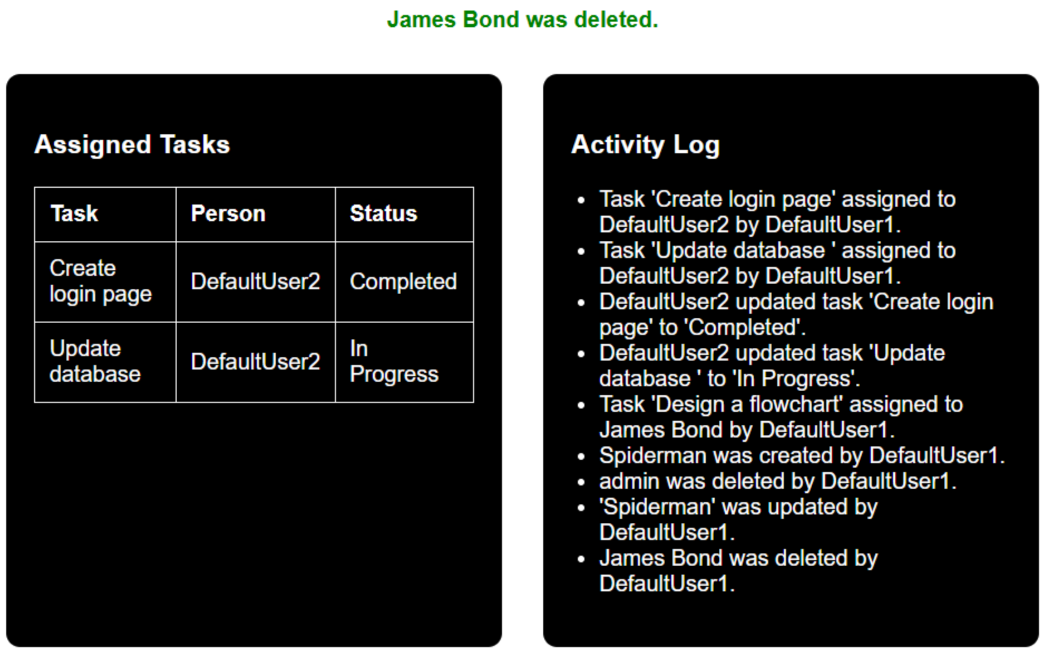
Assigned Tasks

Task	Person	Status
Create login page	DefaultUser2	Completed
Update database	DefaultUser2	In Progress
Design a flowchart	James Bond	Not Started

Activity Log

- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.
- DefaultUser2 updated task 'Create login page' to 'Completed'.
- DefaultUser2 updated task 'Update database ' to 'In Progress'.
- Task 'Design a flowchart' assigned to James Bond by DefaultUser1.
- Spiderman was created by DefaultUser1.
- admin was deleted by DefaultUser1.
- 'Spiderman' was updated by DefaultUser1.

Figure 18: Spiderman was successfully edited.



James Bond was deleted.

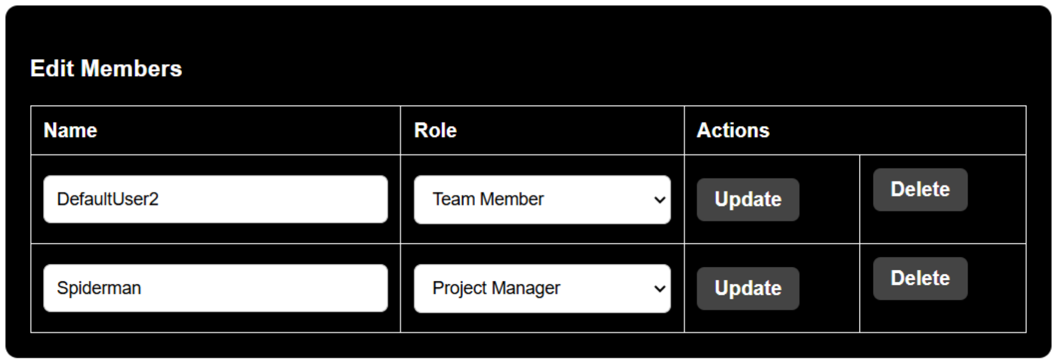
Assigned Tasks

Task	Person	Status
Create login page	DefaultUser2	Completed
Update database	DefaultUser2	In Progress

Activity Log

- Task 'Create login page' assigned to DefaultUser2 by DefaultUser1.
- Task 'Update database ' assigned to DefaultUser2 by DefaultUser1.
- DefaultUser2 updated task 'Create login page' to 'Completed'.
- DefaultUser2 updated task 'Update database ' to 'In Progress'.
- Task 'Design a flowchart' assigned to James Bond by DefaultUser1.
- Spiderman was created by DefaultUser1.
- admin was deleted by DefaultUser1.
- 'Spiderman' was updated by DefaultUser1.
- James Bond was deleted by DefaultUser1.

Figure 19: Deleting James Bond.



Edit Members

Name	Role	Actions	
DefaultUser2	Team Member	Update	Delete
Spiderman	Project Manager	Update	Delete

Figure 20: James Bond no longer appears as a user.

3.5. Activity Logs

All users can see recent activity in their Activity Log. The /dashboard route passes the HTML page a query of the full Activity table through .all(), displaying the latest activities on the page.

4. Design Justifications and Security Decisions

The back and front ends are both coded in app.py for simplicity and efficiency. HTML methods (e.g. GET, POST) and database interactions are handled in the same file as the HTML template rendering. This allows changes to be made faster by avoiding switching between files and reducing the need to configure imports from other files.

The Task table uses the foreign key 'user_id', linked to the User table's ID field. This simplifies assigning and tracking tasks. For example, when a team member updates a task's status, their user ID is searched up in the Task table to find the corresponding task entry.

Sessions hold the current user's information, avoiding the need to login to every page the user visits and manually query the User table. Flask's LoginManager handles the session. After logging in, the /login route calls login_user(user), setting Flask's current_user to the current user's row in the User table. current_user holds the user's attributes e.g. current_user.id, current_user.role, etc. This data is stored on the client's browser as a session cookie. When the user navigates through pages, their data is saved in the session. Since current_user behaves like a global variable, it can be accessed in any function or HTML template to identify and retrieve the current user's details.

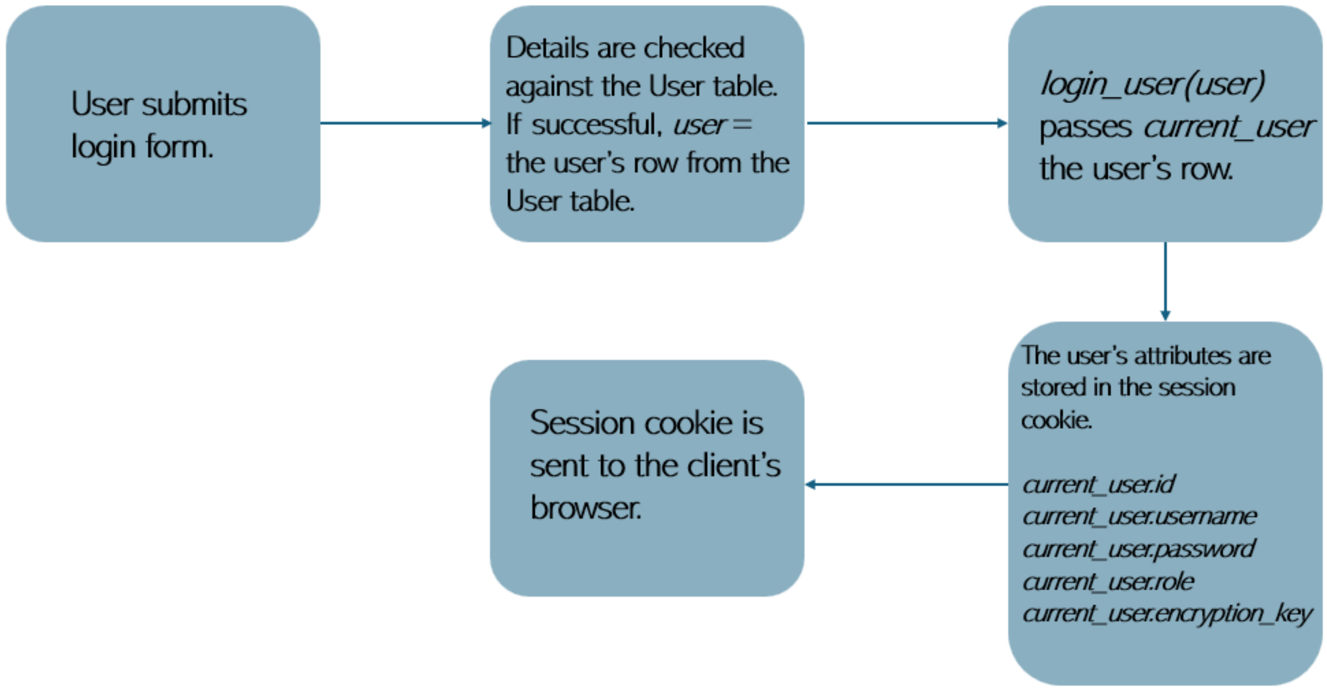

User submits login form.
Details are checked against the User table. If successful, *user* = the user's row from the User table.
login_user(user) passes *current_user* the user's row.
The user's attributes are stored in the session cookie.
current_user.id
current_user.username
current_user.password
current_user.role
current_user.encryption_key
Session cookie is sent to the client's browser.

Figure 21: Flask sessions summarised.

In real life, HTTPS encrypts all data between browser and server, protecting user data like credentials from interception. Every user has a unique encryption key stored in their User table row. When a team member submits a file, it would be securely sent via HTTPS and encrypted on the server using current_user.encryption_key before being stored in the File table.

Role-based access control prevents team members from actions that only project managers are allowed. Routes which affect other users like /edit_member ensure current_user.role = 'Project manager' before accessing the database. /dashboard ensures team members can only see their tasks while managers can see all tasks.

Passwords are stored as hashes with generate_password_hash(). When logging in, the password is compared to the hash in the User table with check_password_hash(). These functions use salting to prevent rainbow table attacks.

Flask's @login_required decorator ensures only authenticated users can access certain routes, preventing unauthorised access to pages and actions.

5. Development Process and Issues Encountered

Back-end Creation Steps:

- Database and tables had to be created first since all routes access them.
- Authentication with /register and /login.
- /dashboard (core route for all users).
- Routes for specific actions: /assign_task,/update_task, /edit_member, etc.

Deleting a user as project manager caused errors because of the foreign key linking the User and Task tables. Every task is assigned to a user via user.id. When trying to delete a user with assigned tasks, the database threw an error as it violated the relationship. This was resolved by deleting all the user's tasks before deleting the user.

Activity Log session issues were solved by replacing the concept with an Activity database table. User activities were only stored in the user session, but because sessions are user-specific, different users could not see each other's actions. An Activity table allows all users to see activities because they are stored as strings inside Activity's message field and are retrieved through ORM.