



INNOVATION. AUTOMATION. ANALYTICS

PROJECT ON

Note Taking Applications - NoteNest

(Code Refactoring and Bug Report Analysis)

Himanshu Agarwal

28th February 2024



About Me

I am a proactive, responsible, and results-oriented professional currently pursuing a bachelor's degree in computer engineering. My interests lie in solving technical issues, conducting research, and innovating new technologies. I thrive in team environments and enjoy connecting with new individuals. With a kind and outgoing nature, I am also a quick learner. Moreover, I excel in working under pressure and possess excellent stress management skills.

I am keen to dive more into the field of Data Science due to its transformative potential across various industries. Firstly, I am intrigued by the prospect of extracting valuable insights from vast amounts of data, which can drive informed decision-making and innovation. Data Science offers a powerful toolkit to uncover patterns, trends, and correlations that can significantly impact businesses and society.

At TCET - Open Source, I've held key roles driving organizational success. As Co-Founder & CEO, I lead project development, manage open-source internships, resolve conflicts, and foster student engagement. In my previous stint as Documentation Team Lead, I provided strategic direction, nurtured team growth, and improved knowledge accessibility.



Link to Project Repo: [NoteNest GitHub Repo](#)

Link to NoteNest: [NoteNest Website](#)



Project Description

NoteNest is a web-based application designed to facilitate note-taking for users. It provides a user-friendly interface for creating, editing, and organizing notes. With features like user authentication, note management, and seamless editing capabilities, **NoteNest** aims to enhance the note-taking experience for individuals across various contexts.

Technologies Used

Frontend:

- Utilized HTML, CSS, and JavaScript for the frontend development.
- Employed Flask's templating system to generate HTML dynamically.

Backend:

- Leveraged Flask, SQLAlchemy, and Python for the backend development.
- Flask provided the web framework, while SQLAlchemy facilitated interactions with the database.
- Utilized Python for server-side logic and routing.

Database Management:

- Implemented database management using SQLAlchemy, which allowed for seamless integration with Flask.
- Hosted the database on PythonAnywhere, a cloud-based platform that supports Python web applications.

Authentication:

- Employed Flask-Login for user authentication within the Flask framework.
- Utilized session-based authentication provided by Flask-Login, which securely manages user sessions.

Deployment:

- Hosted the application on PythonAnywhere, a cloud platform that supports hosting Flask applications.
- Consider exploring other deployment options like AWS, Heroku, or Azure for scalability and additional features.





Key Features

User Authentication:

- Users can securely sign up or log in to access their notes.
- Authentication ensures user privacy and data security.

Note Creation

- Users can easily create new notes, providing titles and content for each note.
- The note creation process is straightforward and intuitive.

Note Editing:

- NoteNest allows users to edit their notes at any time.
- Users can update the content of their notes, ensuring they reflect the latest information.

Note Deletion:

- Users have the flexibility to delete notes they no longer need.
- Deleted notes are removed from the system, helping users maintain a clutter-free note collection.

Bug Description

Initially, the Flask application is configured to handle only POST requests in the **index()** route, and it attempts to retrieve the note from the request arguments using **request.args.get("note")**.

Initial Codebase:

```
home.html x
templates > home.html > ...
Click here to ask Blackbox to help you code faster
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10   <form action="">
11     <input type="text" name="note" placeholder="Enter a note">
12     <button>Add Note</button>
13   </form>
14
15   <ul>
16     {% for note in notes%}
17     <li>{{ note }}</li>
18     {% endfor %}
19   </ul>
20 </body>
21 </html>
```

```
app.py 1 x
app.py > ...
Click here to ask Blackbox to help you code faster
1 from flask import Flask, render_template, request
2
3 app = Flask(__name__)
4
5 notes = []
6 @app.route('/', methods=["POST"])
7 def index():
8     note = request.args.get("note")
9     notes.append(note)
10     return render_template("home.html", notes=notes)
11
12
13 if __name__ == '__main__':
14     app.run(debug=True)
```

Models


Note:

- Represents a note entity in the database.
- Attributes:
 1. **id**: Primary key for the Note table, an auto-incrementing integer.
 2. **data**: String field to store the content of the note, limited to 10,000 characters.
 3. **date**: DateTime field representing the creation date of the note, set to the current time when a new note is created.
 4. **user_id**: Integer field representing the foreign key relationship with the User table. Each note belongs to a single user.

```
class Note(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    data = db.Column(db.String(10000))
    date = db.Column(db.DateTime(timezone=True), default=func.no
```

User:

- Represents a user entity in the database, incorporating UserMixin from Flask-Login.
- Attributes:
 1. **id**: Primary key for the User table, an auto-incrementing integer.
 2. **email**: String field to store the email address of the user, unique across all users.
 3. **password**: String field to store the hashed password of the user.
 4. **first_name**: String field to store the first name of the user.
 5. **notes**: One-to-many relationship with the Note table. Each user can have multiple notes associated with them.



```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(150), unique=True)
    password = db.Column(db.String(150))
    first_name = db.Column(db.String(150))
    notes = db.relationship('Note')
```

These models define the structure of the database tables and the relationships between them, allowing for the storage and retrieval of user and note data within the Flask application. Additionally, UserMixin provides default implementations for user-related functionalities required by Flask-Login, such as user authentication and session management.

Views

Index Route (/):

- Renders the index.html template, which might be a landing page or a simple welcome page.

```
@views.route('/', methods=['GET', 'POST'])
def index():
    return render_template('index.html')
```

Home Route (/home):

- Renders the home.html template, which likely serves as the main interface for users to view and manage their notes.
- Handles both GET and POST requests.
- If it's a POST request, it adds a new note to the database if the note is not empty and then redirects back to the home page.
- If the note is empty, it flashes an error message.

```

@views.route('/home', methods=['GET', 'POST'])
@login_required
def home():
    if request.method == 'POST':
        note = request.form.get('note')

        if len(note.strip()) > 0:
            new_note = Note(data=note, user_id=current_user.id)
            db.session.add(new_note)
            db.session.commit()
            flash('Note added!', category='success')
            return redirect(url_for('views.home'))
        else:
            flash('Note is too short!', category='error')

    return render_template("home.html", user=current_user

```

Edit Note Route (/edit-note/<int:note_id>):

```

@views.route('/edit-note/<int:note_id>', methods=['GET', 'POST'])
@login_required
def edit_note(note_id):
    note = Note.query.get(note_id)

    if current_user.id != note.user_id:
        return render_template("error.html", message="Unauthorized to edit note")

    if request.method == 'POST':
        new_data = request.form.get('note')

        if len(new_data.strip()) > 0:
            note.data = new_data
            db.session.commit()
            flash('Note updated!', category='success')
            return redirect(url_for('views.home')) # Redirect to home page after editing
        else:
            flash('Note is too short!', category='error')

    return render_template("home.html", user=current_user, note=note)

```

- Renders the home.html template with the note to be edited.
- Handles both GET and POST requests.
- If it's a POST request, it updates the existing note with the new data and then redirects back to the home page.
- If the new note is empty, it flashes an error message.
- Checks if the current user has permission to edit the note before allowing the edit.

Delete Note Route (/delete-note):

- Handles the deletion of notes.
- Expects a POST request with JSON data containing the ID of the note to be deleted.
- Deletes the note from the database if it exists and if the current user has permission to delete it.

```
@views.route('/delete-note', methods=['POST'])
def delete_note():
    """
    Handles note deletion.

    Method:
        POST: Deletes the note from the database.


    Returns:
        JSON response.
    """
    note = json.loads(request.data)
    noteId = note['noteId']
    note = Note.query.get(noteId)
    if note:
        if note.user_id == current_user.id:
            db.session.delete(note)
            db.session.commit()

    return jsonify({})
```

Authentications

Login Route (/login):

- This route handles user login functionality.
- When a GET request is received, it renders the login page.
- When a POST request is received (i.e., form submission), it extracts the email and password from the form data.
- It queries the database to find a user with the provided email.



```

@auth.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        user = User.query.filter_by(email=email).first()
        if user:
            if check_password_hash(user.password, password):
                flash('Logged in successfully!', category='success')
                login_user(user, remember=True)
                return redirect(url_for('views.home'))
            else:
                error = 'Incorrect password, try again.'
        else:
            error = 'Email does not exist.'

    return render_template("index.html", user=current_user, error=error)

```

- If a user with the provided email exists, it verifies the password using `check_password_hash`.
- If the password matches, the user is logged in using `login_user`, and a success message is flashed. Then, it redirects the user to the home page.
- If the password doesn't match or the email doesn't exist, it sets an error message to be displayed on the login page.

Logout Route (/logout):

```

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('views.index'))

```

- This route handles user logout functionality.
- It ensures that the user is logged in (`@login_required` decorator).
- It logs out the user using `logout_user`.
- After logging out, it redirects the user to the index page.

Sign-up Route (/sign-up):

- This route handles user registration functionality.
- When a GET request is received, it renders the sign-up page.
- When a POST request is received (i.e., form submission), it extracts the email, first name, and passwords from the form data.
- It checks if the provided email already exists in the database.
- It validates the provided email, first name, and passwords according to specified criteria.
- If all validation checks pass, it creates a new user in the database with a hashed password using `generate_password_hash`.
- It logs in the newly created user using `login_user`, flashes a success message, and redirects the user to the home page.
- If any validation check fails or the email already exists, it sets an error message to be displayed on the sign-up page.

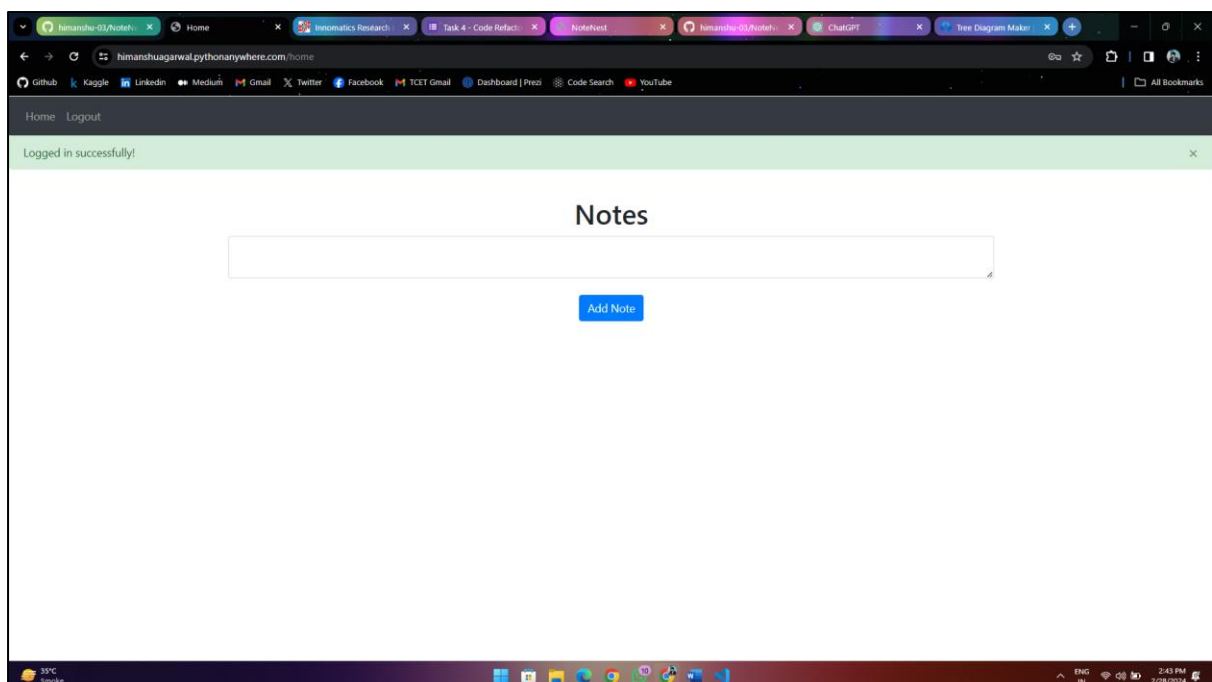
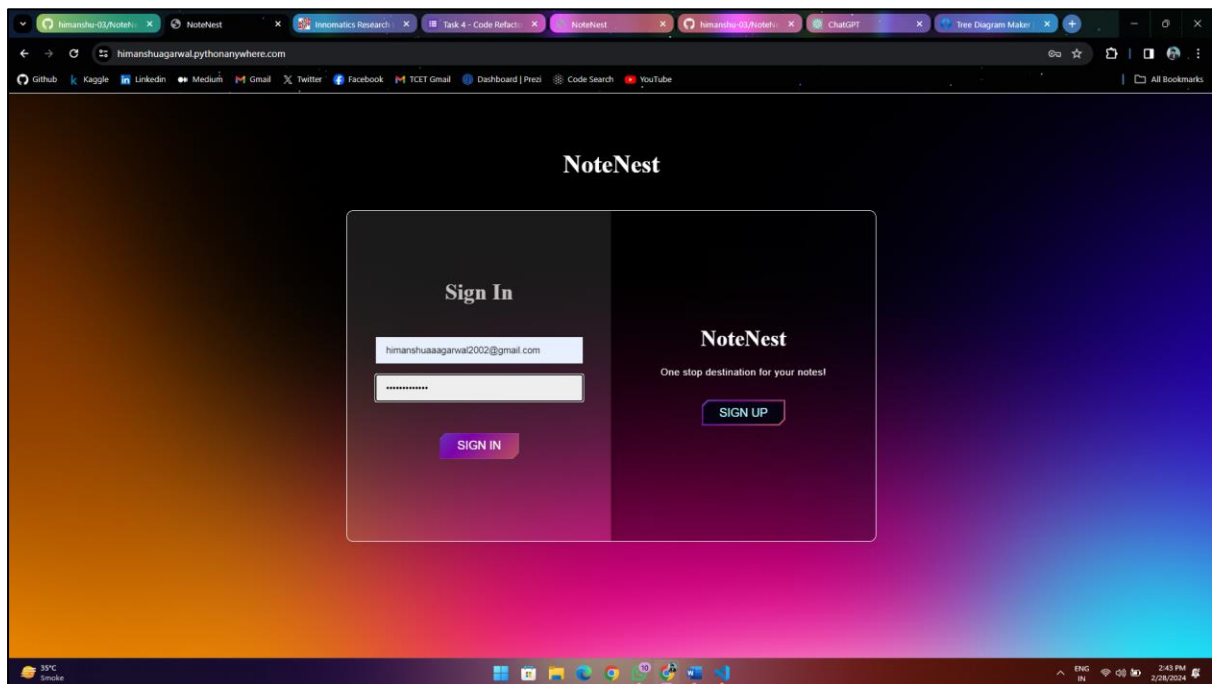
```
@auth.route('/sign-up', methods=['GET', 'POST'])
def sign_up():
    error=None
    if request.method == 'POST':
        email = request.form.get('email')
        first_name = request.form.get('firstName')
        password1 = request.form.get('password1')
        password2 = request.form.get('password2')

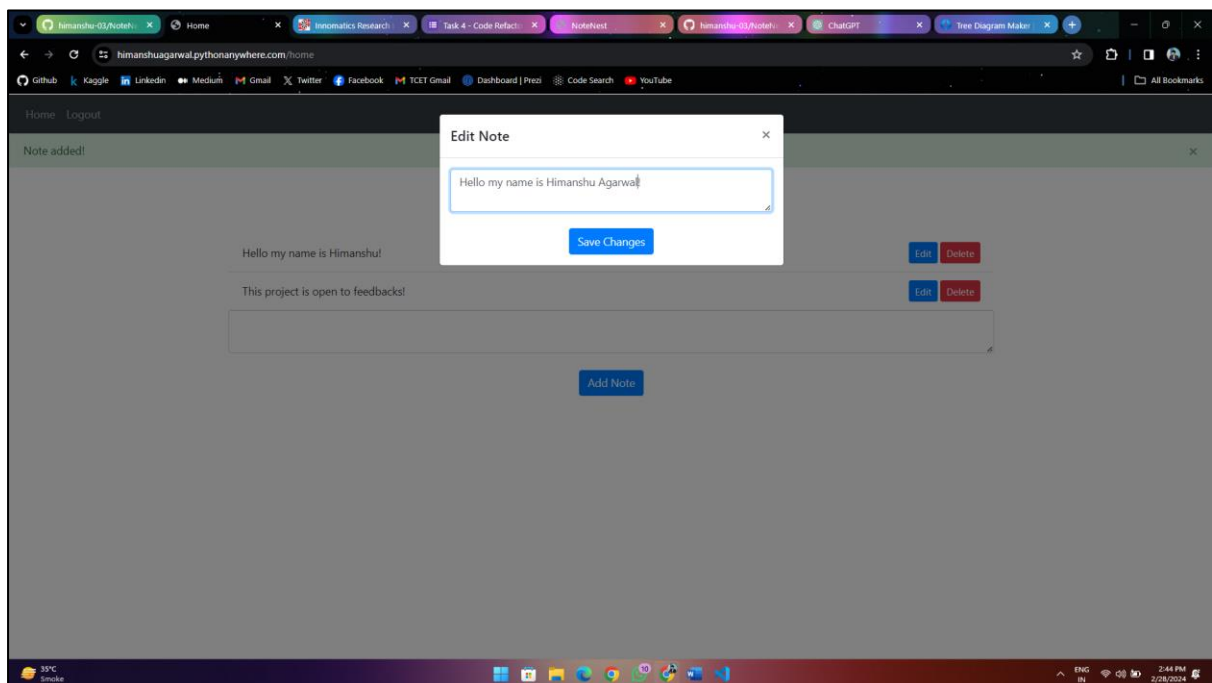
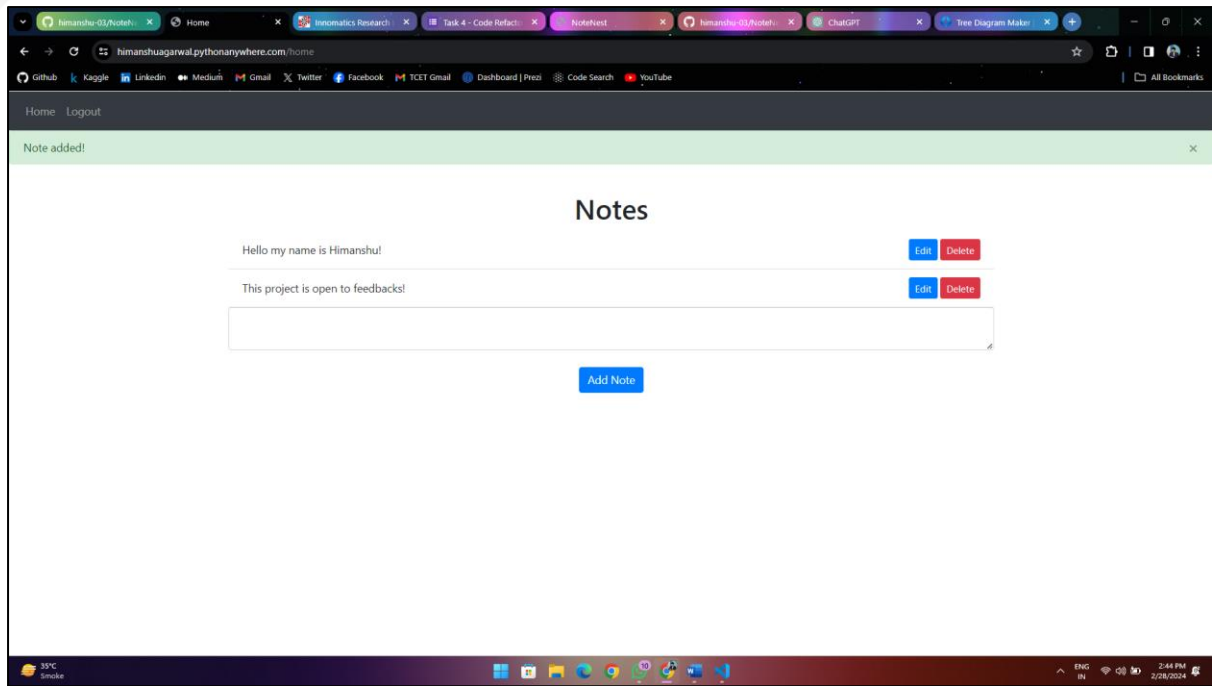
        user = User.query.filter_by(email=email).first()
        if user:
            error = 'Email already exists.'
        elif len(email) < 4:
            error = 'Email must be greater than 3 characters.'
        elif len(first_name) < 2:
            error = 'First name must be greater than 1 character.'
        elif password1 != password2:
            error = 'Passwords don\'t match.'
        elif len(password1) < 7:
            error = 'Password must be at least 7 characters.'
        else:
            new_user = User(email=email, first_name=first_name, password=generate_password_hash(
                password1, method='sha256'))
            db.session.add(new_user)
            db.session.commit()
            login_user(new_user, remember=True)
            flash('Account created!', category='success')
            return redirect(url_for('views.home'))

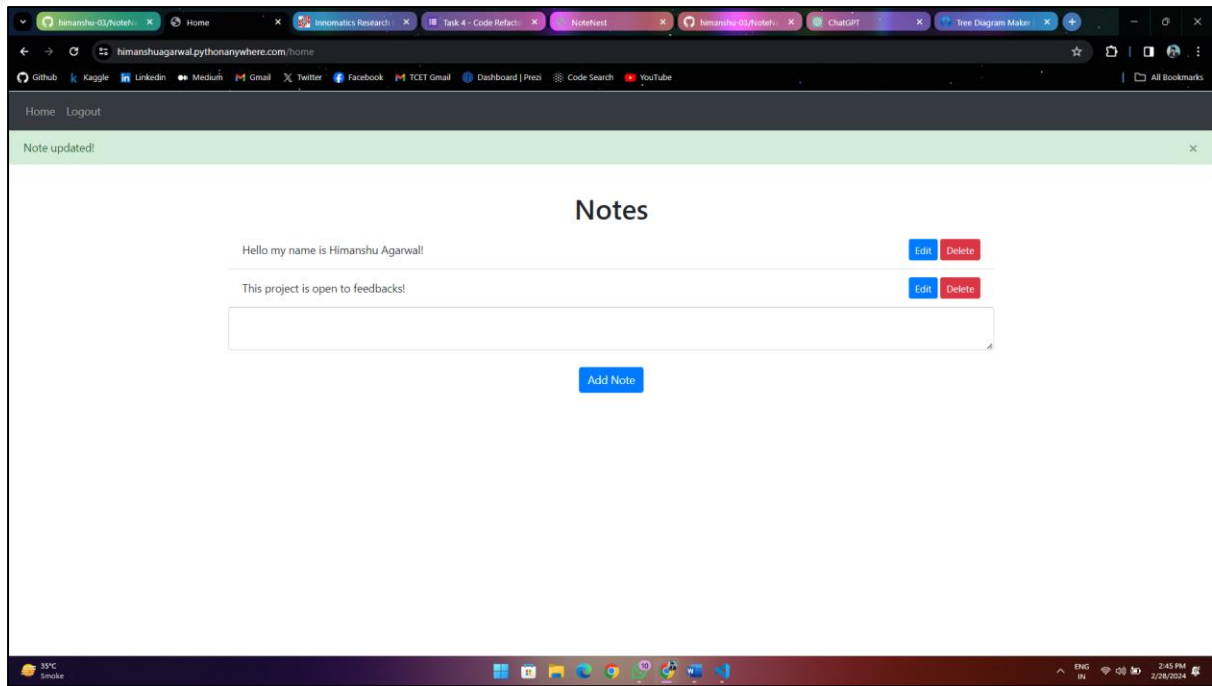
    return render_template("index.html", user=current_user, error=error)
```

These functions together provide the necessary functionality for user authentication and registration within your Flask application, ensuring secure access to protected resources and smooth user experiences during login and sign-up processes.

Final Output







Conclusion

The Flask Notes Application provides a simple yet effective solution for managing notes in a web environment. It leverages Flask and its extensions to implement user authentication and note management functionalities seamlessly. With its intuitive interface and robust backend architecture, the application offers users a convenient platform for organizing their thoughts and ideas.