**Tanvir Ahamed**

**20222440510**

**Computer Science**

**15 Novermber 2024**

Web Application Development



### Abstract

## This project presents the development of a **Travel Guide Web Application** designed to assist users in exploring global travel destinations and managing their trip details effectively. The application is built using **Python Flask** as the server-side framework, with **HTML**, **CSS**, and **JavaScript** used to design the front end. The web app integrates a **MySQL database** to store and manage user profiles, blog posts, and other relevant travel-related information. The system supports essential CRUD (Create, Read, Update, Delete) operations, enabling users to create accounts, update profiles, and interact with blog posts related to various destinations. User-centric design principles are implemented to enhance usability, and multimedia elements such as images and videos are incorporated to improve the overall user experience. The application also features a secure authentication system for users and administrators, ensuring data privacy and a smooth user experience. The project aims to demonstrate the integration of front-end and back-end technologies, offering a seamless web experience for travel enthusiasts.

## Introduction

The purpose of this project is to design and develop a complete web application using modern web technologies. The web application is built using **Python Flask** as the server-side framework and incorporates **JavaScript** to enhance user experience with dynamic design elements. The primary goal is to create a fully functional, interactive website that integrates a backend server, a database for managing user data, and a responsive, user-friendly front-end.

### Project Objective

The objective of this project is to develop a web application that showcases key skills in web development, including:

* **Server-Side Logic**: Implementing a server-side framework (Flask) to manage HTTP requests, interact with databases, and handle user interactions.
* **Dynamic Functionality**: Leveraging JavaScript/jQuery to add dynamic features, such as real-time form validation, content updates, and responsive UI.
* **Database Integration**: Using a SQL-based database (MySQL) to manage user data and implement CRUD operations, enabling users to create, read, update, and delete data.
* **User-Centered Design**: Designing a clean, intuitive interface with a focus on improving user experience (UX).
* **Multimedia Elements**: Incorporating relevant multimedia content, such as images and videos, to enhance the visual appeal and functionality of the website.

### Frameworks and Technologies Used

**Server-Side Framework**: **Flask** (Python)

* + Flask is used to manage the backend logic of the web application. It handles user requests, interacts with the database, and ensures smooth communication between the server and client.

**JavaScript/jQuery**:

* + JavaScript is employed to make the site dynamic. jQuery is used for DOM manipulation, handling asynchronous data, and providing interactive features like real-time form validation and AJAX-based content updates.

**Database**: **MySQL**

* + MySQL is used to manage the database, storing user data and other dynamic content. The application utilizes SQL queries to interact with the database and perform CRUD operations.

**HTML/CSS**:

* + HTML and CSS are utilized for the structure and styling of the application, ensuring it is visually appealing and accessible across various devices.

### Key Features and Functionality

* **User Profile Management**: Users can update their username, email, and password through a secure, user-friendly interface.
* **Post Management**: Users can view, create, and edit blog posts, ensuring seamless content management within the platform.
* **Database Integration**: The app allows for storing, retrieving, and updating user and post data from a MySQL database.
* **Interactive Design**: The application makes use of JavaScript/jQuery to provide dynamic interactions such as form validation and real-time feedback.

### Methodology

The development of this web application follows a structured approach, focusing on key milestones such as:

1. **Designing the user interface**: Ensuring the application is intuitive and visually appealing.
2. **Setting up the backend**: Developing the server-side logic using Flask and integrating MySQL to handle user data and application content.
3. **Implementing dynamic features**: Using JavaScript/jQuery to enhance the interactivity and responsiveness of the website.
4. **Testing and evaluation**: Conducting usability tests to ensure the website works as intended and provides a positive user experience.

This report will detail the processes, challenges, and solutions encountered during the development of this web application. It will also explore how the project meets the requirements outlined in the course guidelines, including server-side functionality, dynamic features, database integration, and user-centered design.

### Code Foldering

### File Descriptions:

**main.py**:

* 1. Contains the Flask app and routes to handle requests.
  2. Runs the web application.

**dump.sql**:

* 1. Includes SQL statements for creating and populating the database (tables, sample data).

**.env**:

* 1. Stores environment variables like SECRET\_KEY, database credentials, email configuration, etc.

**Requirements.txt**:

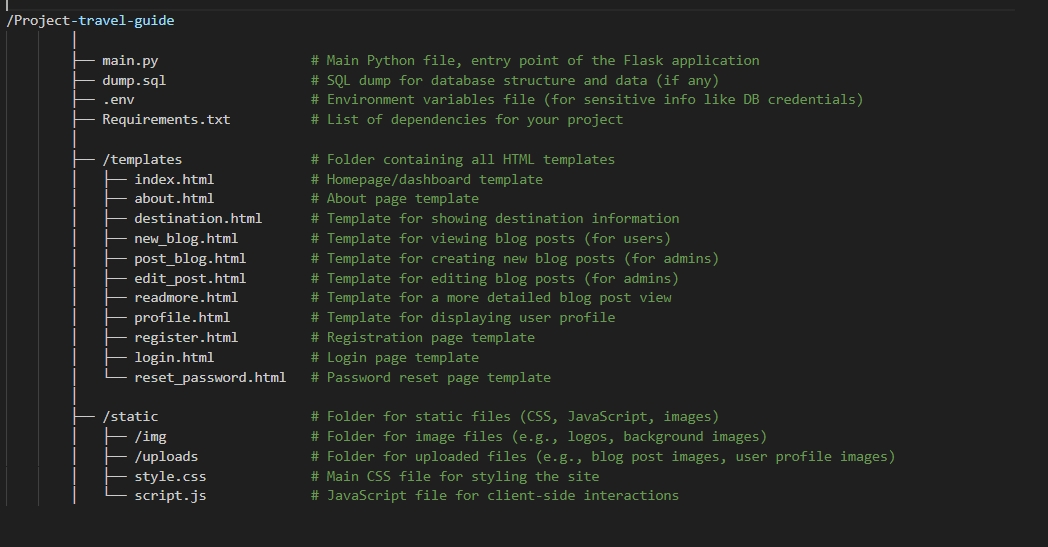
* 1. Lists all the dependencies like **Flask, SQLAlchemy, Flask-WTF**, etc., which are required to run your app.

**/templates**:

* 1. This directory holds all HTML files that are rendered by Flask views.

**/static**:

* 1. Contains all static assets such as images, CSS, and JavaScript files.

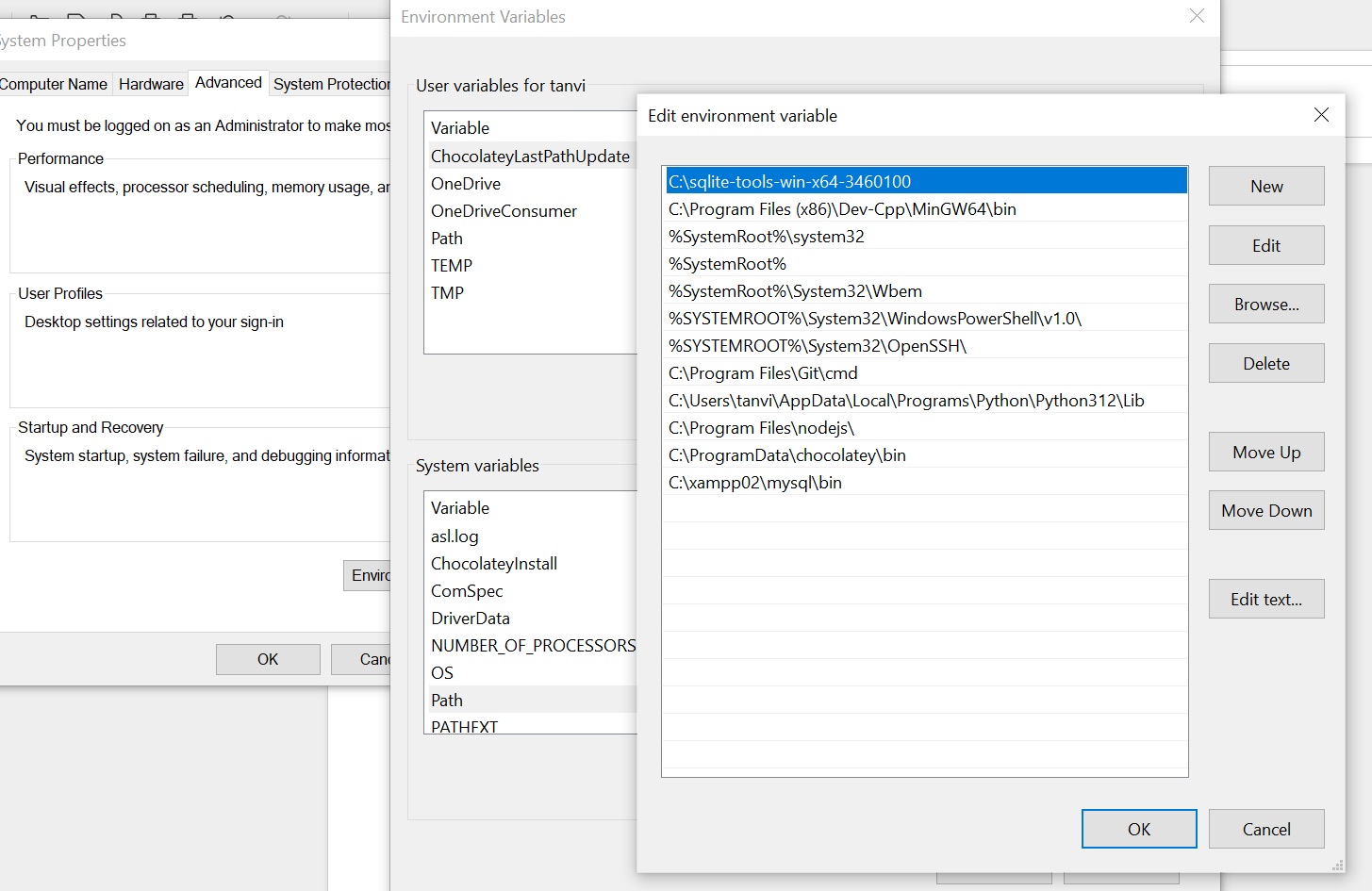


### 1. ****Set Up Your Virtual Environment****

If you haven't already, it's a good idea to create a virtual environment to isolate your project dependencies.

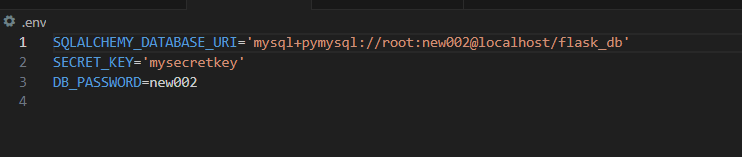
**Steps:**

1. **Install Python**
2. **Setup environment**



1. **Open cmd**
2. Navigate to your project folder (cd /path/to/Project-travel-guide)
3. **pip install -r requirements.txt**

### 2. ****Set Up Your Environment Variables****



We have to make sure to load these environment variables in main.py app. We can use Python's os library or use python-dotenv to load them.

Install python-dotenv if necessary: **pip install python-dotenv**

And in  **main.py** file:

from dotenv import load\_dotenvimport os

load\_dotenv()

SECRET\_KEY = os.getenv('SECRET\_KEY')

### 3. ****Set Up Database****

We need to initialize your MySQL database:

1. Open your MySQL client (XAMPP or any other tool you're using).
2. Create your database using the SQL commands in dump.sql or manually.
3. Modify the DATABASE\_URL in .env to point to your MySQL database, ensuring the username, password, and database name are correct.
4. Set up **SQLAlchemy** in your main.py file for database interaction.

**Example setup for SQLAlchemy in main.py:**

from flask import Flaskfrom flask\_sqlalchemy import SQLAlchemy

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = os.getenv('DATABASE\_URL')

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

### 4. ****Create Database Models****

We need to create the models for the database tables (user-login, blog posts, etc.). Here’s an example for a User and BlogPost model:

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(120), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

password = db.Column(db.String(120), nullable=False)

class BlogPost(db.Model):

id = db.Column(db.Integer, primary\_key=True)

title = db.Column(db.String(100), nullable=False)

content = db.Column(db.Text, nullable=False)

user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

user = db.relationship('User', backref=db.backref('posts', lazy=True))

After defining models, create the tables in the database using:

flask db init

flask db migrate

flask db upgrade

### 5. ****Set Up Routes and Views****

Now, create routes for your application in main.py. Start by setting up basic routes like:

* / (index route, dashboard)
* /login (for user login)
* /register (for user registration)
* /post/<int:post\_id> (to view a blog post)
* /new\_post (for admins to create a new blog post)

**Example:**

@app.route('/')def index():

return render\_template('index.html')

@app.route('/register', methods=['GET', 'POST'])def register():

if request.method == 'POST':

# Handle registration logic here

pass

return render\_template('register.html')

@app.route('/post/<int:post\_id>')def view\_post(post\_id):

post = BlogPost.query.get\_or\_404(post\_id)

return render\_template('readmore.html', post=post)

### 6. ****Set Up Login and Registration****

Implement user authentication using **Flask-Login**. This will allow users to register, log in, and log out.

**Install Flask-Login:**

**pip install flask-login**

**Then, add the necessary code to handle login and registration:**

from flask\_login import LoginManager, UserMixin, login\_user, login\_required, logout\_user

login\_manager = LoginManager(app)

login\_manager.login\_view = 'login'

class User(UserMixin, db.Model):

# Define user model here

@login\_manager.user\_loaderdef load\_user(user\_id):

return User.query.get(int(user\_id))

# Register and Login routes

### 7. ****Implement Blog Post Functionality (Admin and User)****

Create routes and templates for users to create, view, and edit blog posts, as well as for admins to manage posts.

### 8. ****Set Up Email (for OTP, Password Reset)****

Use **Flask-Mail** to send emails. For instance, to send OTPs or password reset links.

**Install Flask-Mail:**

**pip install flask-mail**

**Then, configure it in main.py:**

from flask\_mail import Mail, Message

mail = Mail(app)

@app.route('/send\_email')def send\_email():

msg = Message('Hello', sender='your-email@example.com', recipients=['user@example.com'])

msg.body = 'This is a test email.'

mail.send(msg)

return 'Email sent!'

### 9. ****Develop and Test the Frontend****

Start with the basic HTML pages. Use Bootstrap (or your custom CSS in style.css) to style the pages. Make sure each page is responsive, especially if you want it to be user-friendly on mobile devices.

### 10. ****Testing and Debugging****

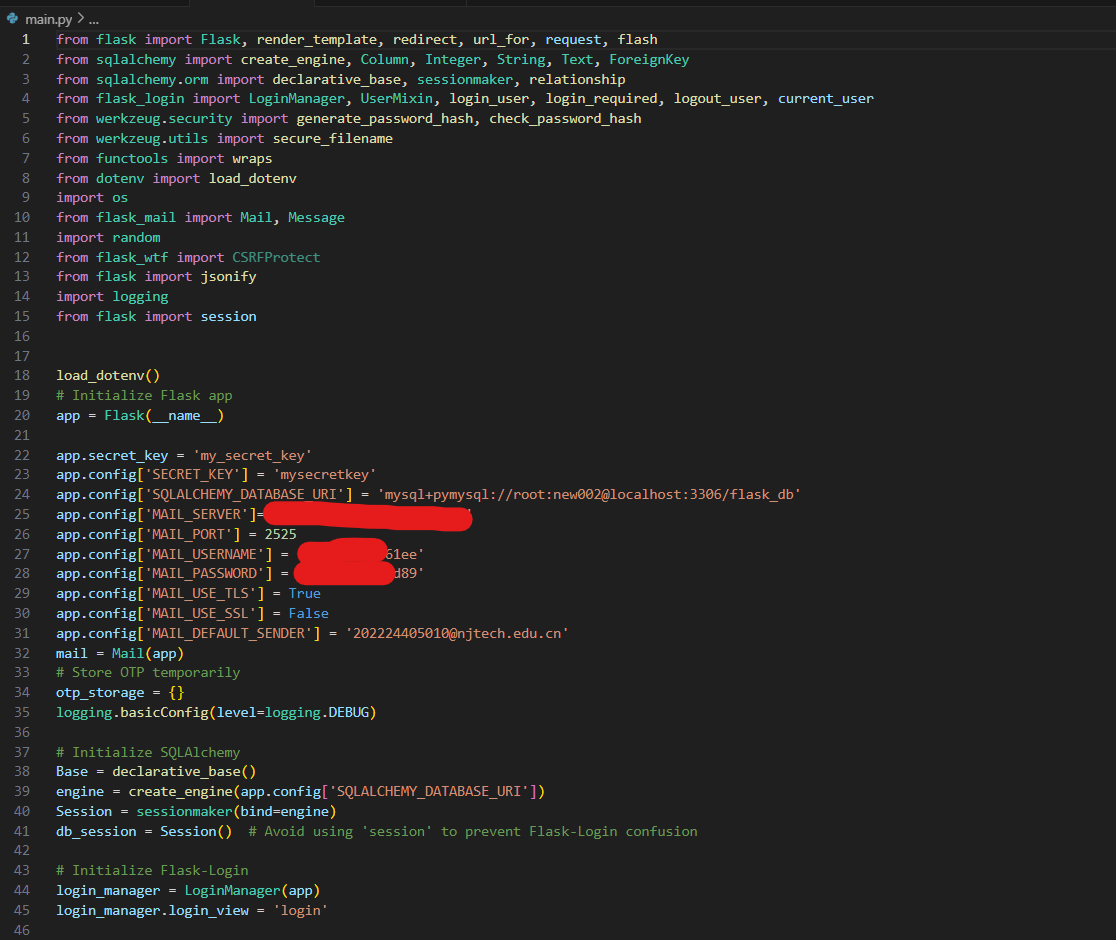
Once you have all routes and functionalities set up, test everything in development mode. Check for errors and debug them as needed.

* Test registration, login, password reset, and OTP functionality.
* Check blog post creation, viewing, and editing.

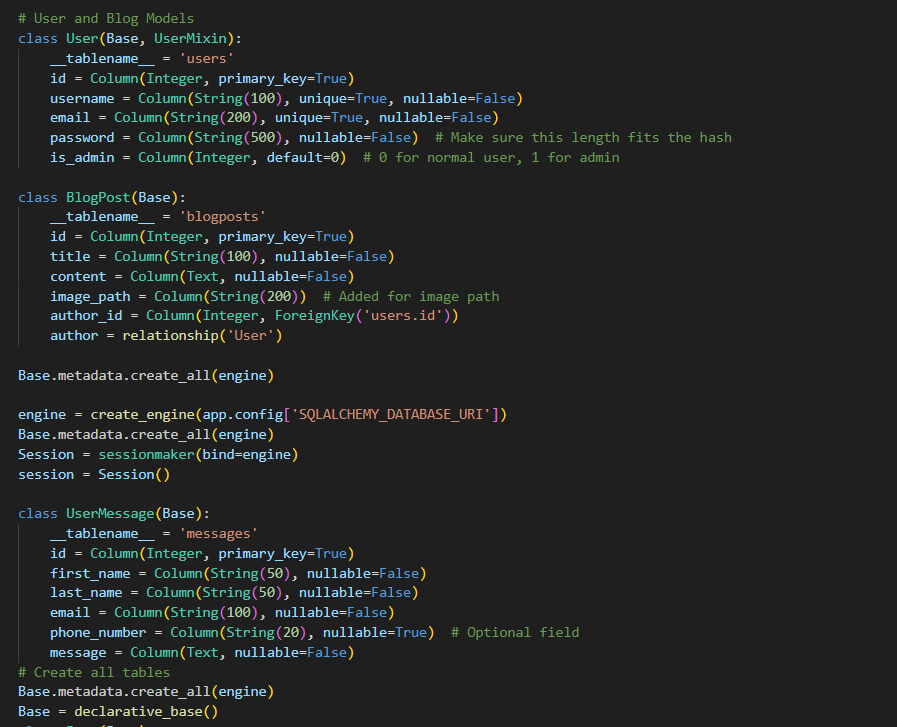
**Now we have the environment so we can start our project step by step.**

1. Set Up the Flask App in **main.py**:

Start by creating basic Flask app setup and configuring your environment variables, database, and email settings. Here in read mark, it is for SMTP server and its password.

**2. Database model setup:**

In the database setup we have to make sure that our database is connected with our main.py. If its not connected with our main.py them it will rise erorr.

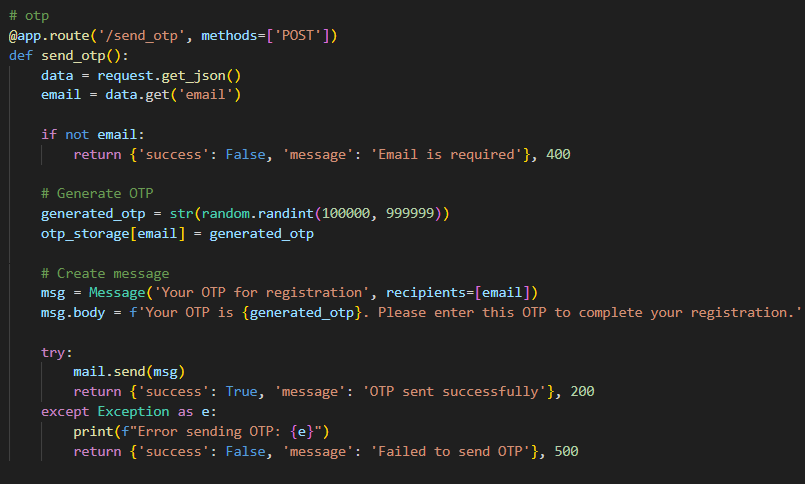


### 3. ****Set Up Login and Registration Routes****

Next, implement the user registration and login functionality. This involves creating the register and login routes. For now, let's keep it simple.

**Otp:**

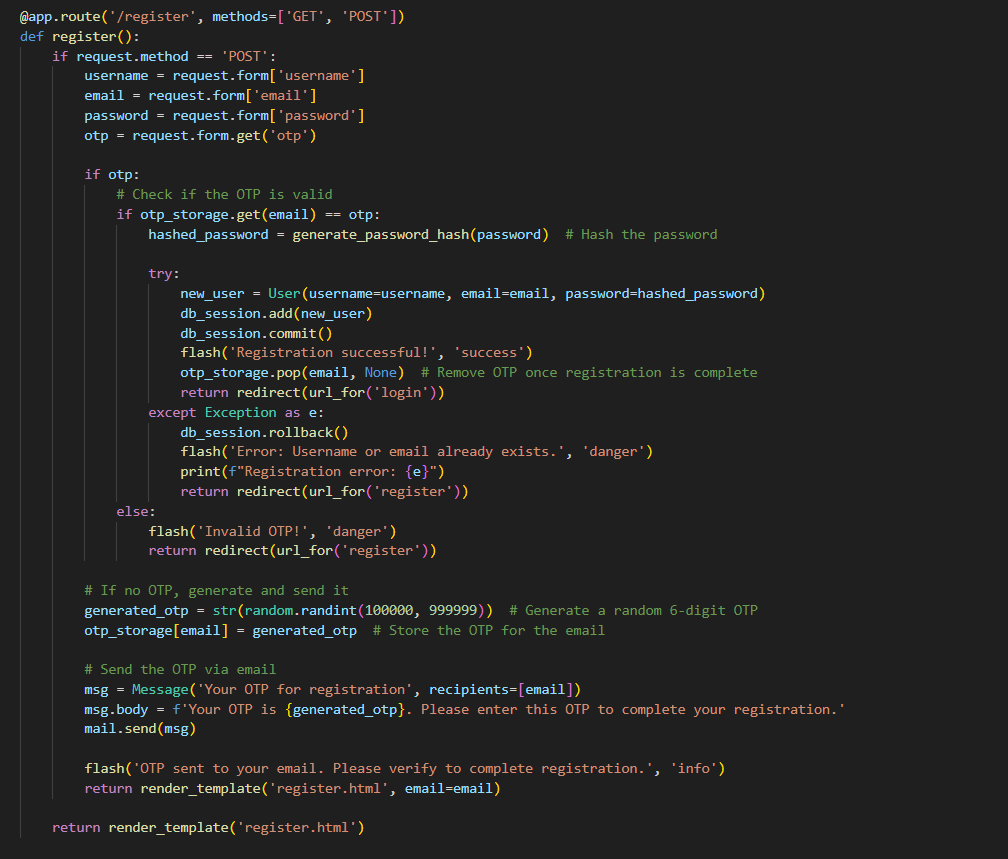
Otp is required for registration so we have make another route for otp



#### Registration Route:

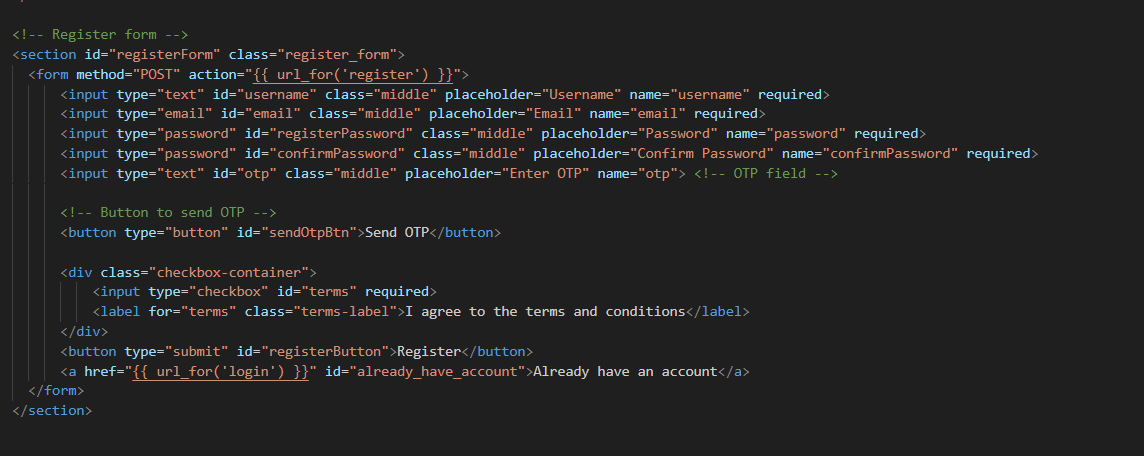
Inside the registration I have defined otp function and it will take user information and the password is hashed\_password that we defined at top

from werkzeug.security import generate\_password\_hash, check\_password\_hash



**User interface for this part:**

For user interface our code file register.html



## AJAX: What is AJAX?

AJAX = ****A****synchronous ****J****avaScript ****A****nd ****X****ML.

AJAX is not a programming language.

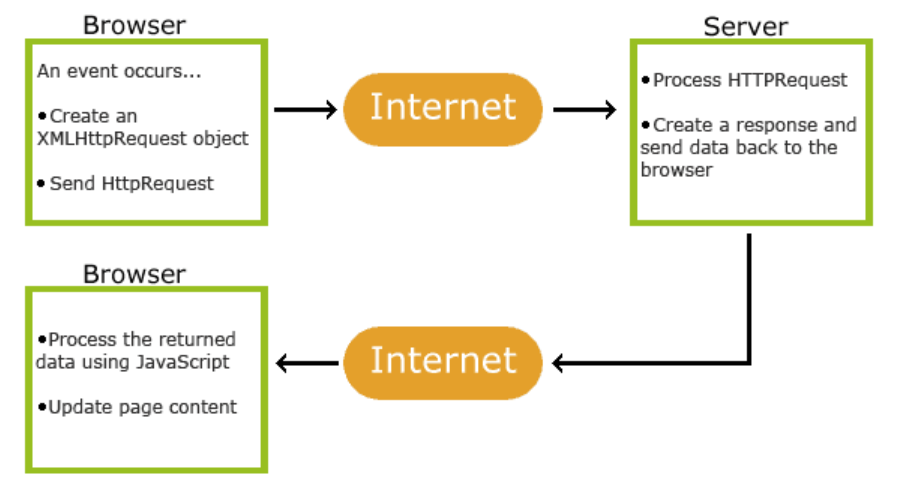
AJAX just uses a combination of:

* A browser built-in XMLHttpRequest object (to request data from a web server)
* JavaScript and HTML DOM (to display or use the data)

AJAX is a misleading name. AJAX applications might use XML to transport data, but it is equally common to transport data as plain text or JSON text.

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

## How AJAX Works



1. An event occurs in a web page (the page is loaded, a button is clicked)

2. An XMLHttpRequest object is created by JavaScript

3. The XMLHttpRequest object sends a request to a web server

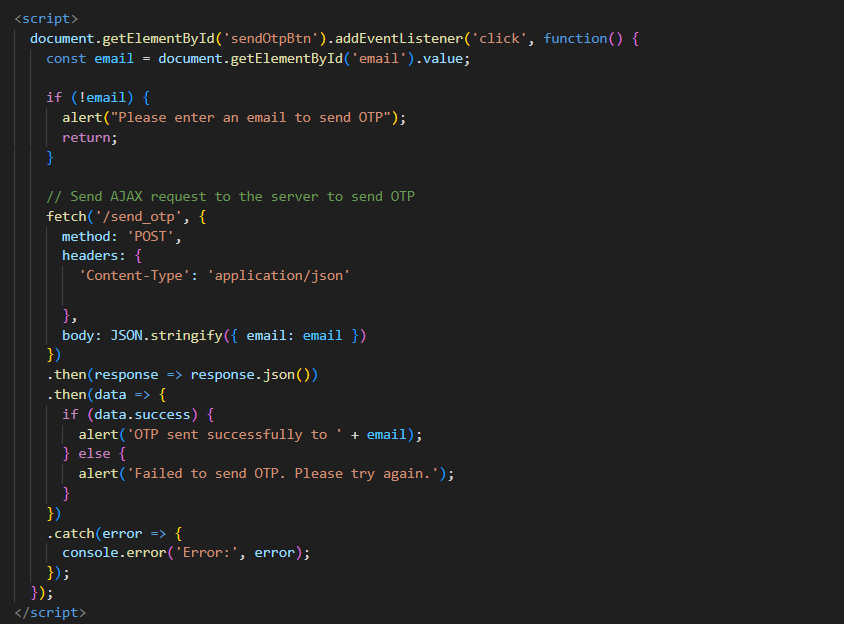
4. The server processes the request

5. The server sends a response back to the web page

6. The response is read by JavaScript

7. Proper action (like page update) is performed by JavaScript

**Why we need Ajax here:** without refreshing to fetching user email then the smtp server will send the otp to user email.

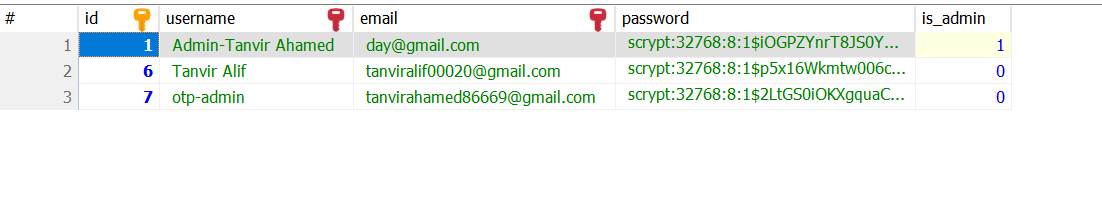


**Log IN:** If user already have an account then it will show the **login.html** page.

**Log in route:**



Now it will check user validation from database, if user found in database then it match user id, email and password. If it is match with the data base then it will give the permission for login if not then it will show Invalid username or password.

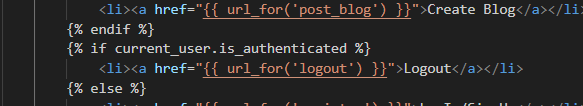
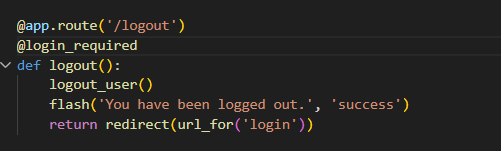


**So how can we control the admin panel?** For the enter the web-page as admin, we already define it at database setup step as normal user is by **default 0 and admin is 1.** We can change it from database. Admin has access to create blog, edit blog and delete blog. Normal user can only read the posted blog but login is required for that.

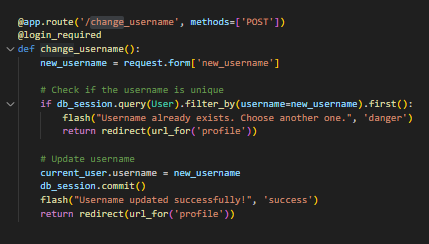
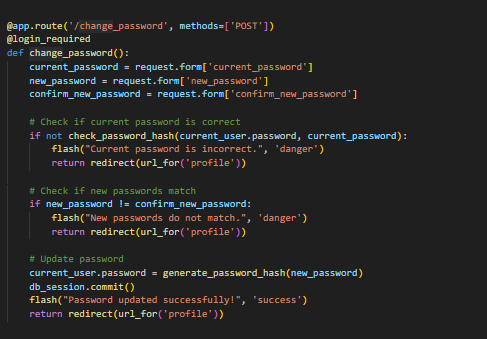
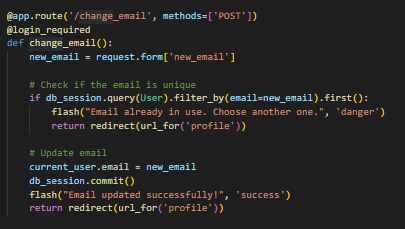
**Why User cannot post anything?** The logic behind this thing is at first its not any social media web-site. In social media everyone can share their opinion and in social media the news can be false and can be true as well. Sometimes people got confused for that. So here all kinds of information is coming form admin or authority. So user will never confused.

**Log Out route:**

If user have login page then it should have log out function. So we have also that log out function. But for log out the function is little bit different. The function is inside the profile, so first of all if user is authentic then it will show **profile.html** inside the **index.html** navigation. At that page **profile.html**  user can log out form the web-page. It go back to login page and show message “log out success” More additionally in the profile page there is some other option to change user-name, email and also password.



**Change User-name, Email and Password route:**

This section of the Flask application implements functionality for updating user profile information, including the username, email, and password. The routes are protected by the @login\_required decorator, ensuring that only logged-in users can make these changes.

**Change Username (**/change\_username**)**

* 1. **Description:** This route allows users to change their username.
  2. **Validation:** It checks whether the new username already exists in the database. If the username is already taken, a flash message is shown, and the user is redirected to the profile page.
  3. **Action:** If the username is unique, it updates the current\_user.username and commits the change to the database.

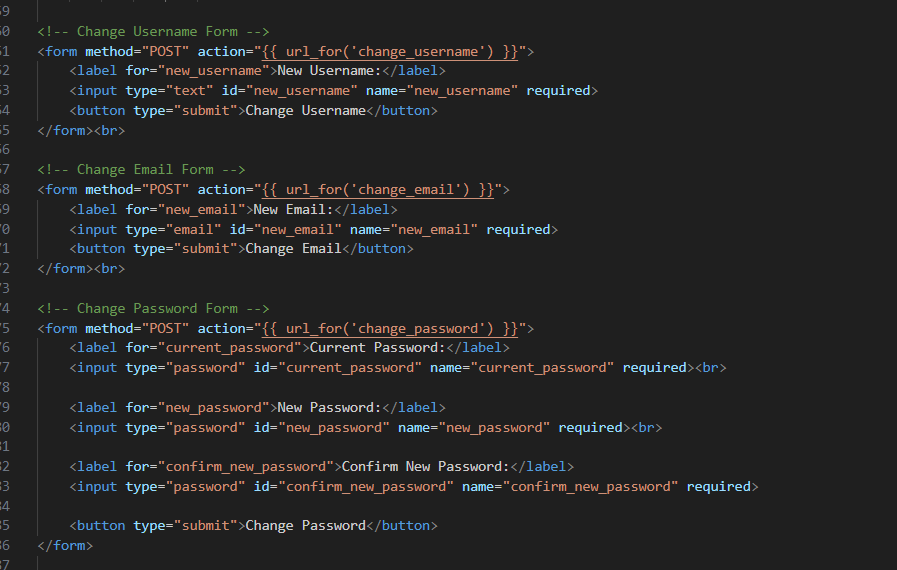
**Change Email (**/change\_email**)**

* 1. **Description:** This route allows users to change their email address.
  2. **Validation:** It ensures the new email is not already in use by checking the database. If the email is already associated with another user, an error message is flashed, and the user is redirected to the profile page.
  3. **Action:** If the email is unique, it updates the current\_user.email and commits the change.

**Change Password (**/change\_password**)**

* 1. **Description:** This route allows users to update their password.
  2. **Validation:** It first verifies that the current password entered by the user matches the one stored in the database. Then, it checks whether the new password and the confirmation match. If either validation fails, the user is informed with a flash message.
  3. **Action:** If validations pass, the password is updated using generate\_password\_hash to securely store the new password.

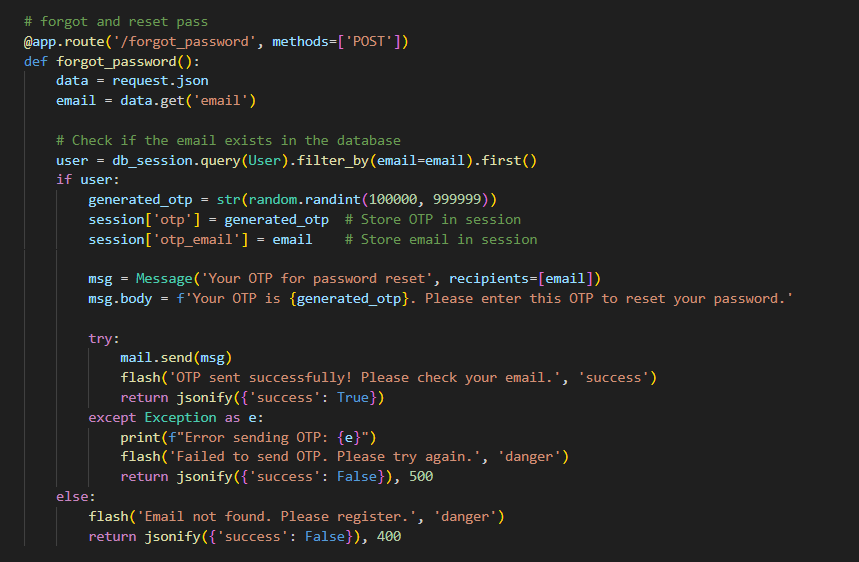
Here, this picture is about **profile.html.** This structure for user input.



**What user will do if he/she forgot password?**

**Forgot password and reset password route:**

**Forgot password:**



This section of the Flask application handles the password reset process for users who have forgotten their passwords. It includes two routes: /forgot\_password for generating and sending the OTP (One-Time Password) and /reset\_password for validating the OTP and resetting the password.

#### 1. ****Forgot Password Route (****/forgot\_password****)****

* **Method:** POST
* **Description:** This route initiates the password reset process by generating an OTP and sending it to the user’s registered email.
* **Flow:**
  + The route accepts an email address from the client in JSON format.
  + It checks if the email exists in the database. If the email is found, a 6-digit OTP is generated and stored in the session along with the user's email.
  + The OTP is sent to the user via email using Flask-Mail.
  + If the OTP is successfully sent, the user receives a success message. If there is an error while sending the OTP, the user is notified of the failure.
  + If the email is not found in the database, an error message is shown, prompting the user to register.

#### 2. ****Reset Password Route (****/reset\_password****)****

* **Method:** GET and POST
* **Description:** This route is responsible for rendering the password reset form and validating the OTP entered by the user.
* **Flow:**
  + When the method is GET, it renders the reset\_password.html template to allow the user to input their email, OTP, and new password.
  + When the method is POST, it retrieves the email, OTP, and new password from the form.
  + The OTP and email stored in the session are compared with the ones provided by the user.
  + If the OTP and email match, the password is updated in the database, and a success message is shown. The OTP and email are then cleared from the session.
  + If there is any error during the database update, a rollback occurs, and the user is notified.
  + If the OTP is invalid, an error message is displayed, and the user is prompted to try again.

### Why Two Routes are Needed?

**Separation of Concerns:**

* + **Forgot Password Route (**/forgot\_password**)**: This route is responsible for generating and sending the OTP. It handles the first step of the password reset process, ensuring that only the user who owns the email address can receive the OTP. This is also where the email verification takes place, ensuring that the email exists in the system before proceeding.
  + **Reset Password Route (**/reset\_password**)**: After the OTP is sent, the second route is required to validate the OTP and allow the user to set a new password. This keeps the process secure by ensuring the user must input the OTP they received, confirming their identity before changing the password.

**Security:**

* + By separating the processes into two distinct routes, the application ensures that:
    - The OTP is only valid for a limited period and can only be used once.
    - The email address and OTP are validated separately, reducing the chance of unauthorized access.
    - The user’s password can only be changed after the OTP is verified, which prevents unauthorized password resets.

**UX Flow:**

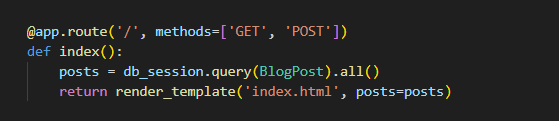
* + Having separate routes makes it easier to manage the user experience. The first route (/forgot\_password) focuses on sending the OTP, while the second route (/reset\_password) focuses on updating the password, ensuring that each route has a clear, defined responsibility and improves the user’s understanding of the process.

**Backend Complete: Time to Move to Frontend and Integrate with main.py**

After successfully completing the backend functionality, including features like user authentication, password reset, and OTP generation, the next step is to move to the frontend. At this stage, we will design the user interface to interact with the backend routes and ensure a seamless user experience. This will involve creating forms for password reset, login, and profile management, as well as connecting the frontend with the backend via route handling. Additionally, we need to integrate the frontend with the main.py to ensure all components communicate correctly, enabling features like user authentication, password reset, and profile updates to work smoothly within the Flask application.

### Description of index.html - Travel Website Homepage

This index.html page is the homepage of a travel website, designed to showcase the platform's features and engage users right from the moment they land on the site. The layout combines a clean, modern aesthetic with interactive elements to create an immersive experience for visitors.



This image describe about connecting with backend file **main.py**.

Here’s a breakdown of the key sections and components included on this page:

1. **Header and Navigation Bar**

* **Logo**: The website logo (Explore) is displayed with a combination of a custom font, highlighting "Explore" to emphasize the travel theme.
* **Navigation Links**:



* + **Home**: Redirects to the homepage (index.html).
  + **About**: Links to the about.html page that provides information about the website and its mission.
  + **Destinations**: Links to the page showcasing travel destinations (destinations.html).
  + **Posted Blog**: A link to view blog posts (new\_post.html).
  + **Create Blog**: Conditional rendering for admin users only, allowing them to create new blog posts (post\_blog.html).
  + **Profile**: Visible for authenticated users, this links to the user’s profile page (profile.html).
  + **LogIn/SignUp**: Visible for non-authenticated users, leading to the login or registration page (register.html).
* **Hamburger Menu**: A responsive navigation option that appears on smaller screens, ensuring the website remains mobile-friendly.

2. **Loading Page**

* A simple loading screen that displays the percentage of content loaded, providing users with feedback while the page is loading.

3. **Hero Section**

* **Text Container**: Features a prominent headline (Lets Explore the World) and a subheading that highlights the adventure and exploration theme of the site.
* **Rotating Earth Animation**: Using a canvas element, the rotating Earth animation creates a visually appealing, dynamic background.
* **Main Text**: Encourages users to discover new destinations, offering an immersive introduction to the site.

4. **Typing Effect (Using Typed.js)**

* The section below the hero features a dynamic typing effect (Typed.js), where the text changes to engage users with phrases related to travel and exploration. This adds a modern touch to the website.

5. **About Section**

* **Trip Planning Prompt**: The heading invites users to explore and plan their trips by reading the blog.
* **Destination Previews**: This section showcases three destinations with images and a "Read more" button:
  + **Nanning, China**
  + **Cox Bazar, Bangladesh**
  + **Vancouver, Canada**
* The images are visually linked to their respective destinations, sparking interest and encouraging users to click for more information.

6. **Video Section**

* A video player embedded from a link, featuring content that highlights "Best in Travel 2024." The video is muted, set to autoplay, and loops, adding a dynamic and engaging element to the page. This section showcases some of the best travel moments, reinforcing the website's mission to inspire travelers.

7. **Feature Section**

* **Dream Itinerary**: A section highlighting the personalized travel itinerary feature. It includes two images representing various travel experiences.
* **Toggle Sections**:
  + **How it Works**: Explains the process of trip requests and expert-crafted itineraries.
  + **Local Experts**: Describes the role of local experts who help plan unique travel experiences.
  + **What’s Included**: Details the key features included in the travel package, such as accommodation, exclusive experiences, concierge services, and transportation.
* These sections offer insights into the travel service, increasing user confidence and interest.

8. **About Explore Section**

* A brief paragraph introducing **Explore**, the website's mission, and its focus on offering travelers captivating stories, up-to-date information, and expert guides to make their journeys extraordinary.
* **Call to Action**: Invites users to join the journey of exploration, with a focus on providing valuable travel tips and advice.

9. **Footer Section**

* **Parallax Effect**: The footer features a parallax effect with animated trees, vehicles, and a rotating moto, creating a visually appealing backdrop as users scroll.
* **Developer Info**: Lists the developer of the site, **Tanvir Ahamed,** including affiliation with **Nanjing Tech University in the Computer Science department.**
* **Social Contact Links**: Includes links to social media profiles such as Facebook, Instagram, and GitHub, encouraging users to follow and connect with the developer.
* **Contact Info**: Provides an email address for user inquiries and support.
* **Copyright**: A copyright message with a link to the developer's **GitHub profile.**

**Summary index.html:**

The **index.html** page serves as the main entry point for users visiting the travel website. It combines aesthetic appeal with functionality, offering intuitive navigation and interactive features that guide users through the site’s offerings. From exploring destinations to understanding the services provided, the homepage captures the essence of the platform while making it easy for visitors to engage with the content. The dynamic elements, such as the rotating Earth, video section, and the typing effect, create a lively and engaging atmosphere that reflects the adventurous spirit of travel. For checking the code: **<https://github.com/tanvir-ahamed04/Travel-Website/blob/main/templates/index.html> .**

### ****About Page - Travel Website (about.html)****

**Overview:** The **about.html** page of the Travel Website serves to introduce users to the website's mission, team, and the philosophy behind the platform. The page highlights the goal of encouraging exploration and adventure while providing users with insights into travel and experiences around the world. The content of the page emphasizes the importance of travel as not just visiting destinations, but also as a means of personal growth, cultural immersion, and creating lasting memories.

**About page route(main.py):**



#### ****Key Sections:****

**Introduction to the Site ("About Us"):** The page begins with a welcoming message to users. It explains that "Explore" is a travel website dedicated to helping travelers discover new places, cultures, and experiences. It presents the team behind the website as passionate adventurers, writers, and travel enthusiasts who aim to inspire users with valuable travel knowledge. The message highlights the website's mission of enabling users to travel with confidence by offering tips, itineraries, and cultural insights.

**Mission Statement:**

* 1. The core message of the "Explore" website is that travel is about more than just the destinations—it’s about creating meaningful experiences, immersing oneself in new cultures, and building cherished memories.
  2. The mission focuses on making every journey a unique story that is shared with the world.

**Visuals and Design:**

* 1. The page features a large, stunning image of a travel destination, with an overlay that promotes the beauty of travel and adventure.
  2. The use of purple-themed text boxes creates a visual contrast, drawing attention to key messages such as “Embrace the world’s beauty and adventure.”

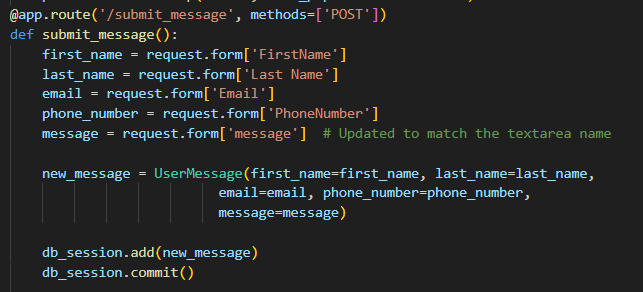
**User Engagement:**

* 1. There is a strong emphasis on connecting with the community of travelers. The section encourages users to “join the journey” and become part of a global network of wanderers.
  2. The website also promotes its social media presence with links to Twitter, Facebook, and Instagram, offering an opportunity for users to engage with the brand and stay updated.

**Contact and Quote Section:**

* 1. A contact section is included for users who wish to get in touch for personalized travel advice or booking services.
  2. The form allows users to request a quote by providing details such as name, email, phone number, and message. This section aims to convert visitors into customers by offering direct interaction with the site’s team.

**Contact with admin:** If user send any message it will store at database. 



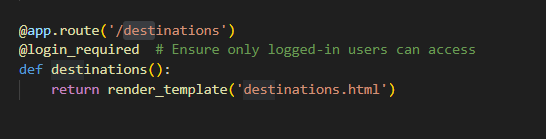
#### ****Key Messages:****

* **Empowerment Through Travel:** The website encourages users to step outside their comfort zone and discover the world’s beauty and adventure.
* **Support for Travelers:** Whether it's advice on destinations, accommodation suggestions, or cultural insights, the platform aims to be the go-to resource for travelers.
* **A Global Community of Travelers:** The site fosters a sense of belonging by connecting like-minded individuals who share a love for exploration.

The **about.html** page is essential in shaping the brand identity of the "Explore" website. It serves to engage users by offering them not only a travel platform but a space to connect with a global community of explorers. The mission and vision of the website are clearly communicated, promoting the idea that travel is an experience that enriches the soul and broadens one’s perspective. Check code: **<https://github.com/tanvir-ahamed04/Travel-Website/blob/main/templates/about.html> .**

**Destination- destination.html:**

In **main.py** we have to define its route for connecting this page.



Here's a breakdown of its structure and functionality:

**Header Section (Navigation Bar):**

* 1. Includes the site logo with a link to the homepage.
  2. Navigation links for "Home," "About," "Destinations," "Posted Blog," and "Profile" (depending on user authentication and admin status).
  3. A hamburger menu for responsive design.

**Loading Page:**

* 1. Displays a loading screen when the page is loading, with a counter that progresses as the page loads.

**Destination Section:**

* 1. A header introducing the "Adventure" theme of the destination page.
  2. A paragraph with a welcome message explaining the purpose of the page: to showcase various travel destinations and inspire users to explore new places.
  3. A card-style section with a "Contract Me" button for users to inquire about travel destinations or services.

**Picture Grid Section:**

* 1. Displays a grid of images, likely representing different travel destinations.
  2. Each image is responsive, and they come from sources like Unsplash, giving the page an aesthetically rich look.

**Contact Section ("Contract me"):**

* 1. A form where users can submit their details (first name, last name, email, phone number, and a message).
  2. The form aims to allow users to request a quote, and it includes social media links to connect with the website owner on Twitter, Facebook, and Instagram.

### Key Features:

* **Dynamic Content:** Uses Jinja templating to include dynamic content like {{ url\_for('index') }}, which generates links based on the routes defined in the Flask application.
* **Responsiveness:** The layout includes responsive classes to ensure it works well on different screen sizes (using wk-desk, wk-ipadp, wk-tab, wk-mobile).
* **Social Media Links:** There are links to the owner's social media profiles, allowing users to engage with the website's creator.

This page would be part of a travel website where users can explore destinations, inquire about travel services, and connect with the website owner. Code check: **<https://github.com/tanvir-ahamed04/Travel-Website/blob/main/templates/destinations.html> .**

**Blog Creation Interface for Admins in the Travel Guide Website**

The **post\_blog.html** page serves as a central interface for website administrators (admins) to create and publish blog posts. This page is restricted to admin users only, ensuring that the blog content shared on the platform is consistent, relevant, and professionally curated. By enabling only admins to create and manage blog posts, the website maintains quality content for users seeking travel insights.



In the route **/post\_blog** it will take input from the user that save it on database. The picture will save on upload directory if upload directory not found it will create that and save the image on it.

### ****Structure and Functionality of**** post\_blog.html

The **post\_blog.html** page consists of structured HTML with integrated CSS and JavaScript to create an intuitive and responsive interface for admins to post blogs. This page includes key sections, such as a loading screen, navigation menu, blog creation form, and a JavaScript-based loading progress animation. Below is a breakdown of each section.

### ****HTML Structure and Components****

#### ****DOCTYPE Declaration and Head Section****

The page starts with the <!DOCTYPE html> declaration, ensuring proper rendering standards. Within the <head> tag:

* The <meta charset="UTF-8"> specifies UTF-8 encoding for handling special characters.
* The <meta name="viewport" content="width=device-width, initial-scale=1.0"> ensures that the page is responsive on all devices.
* The page title is set as "Post a Blog," with an additional <link rel="icon"> pointing to the icon image, enhancing the website’s branding.

#### ****Navigation Header****

The <header> section features the **Navigation Bar**, allowing easy access to key sections.

#### ****Loading Page****

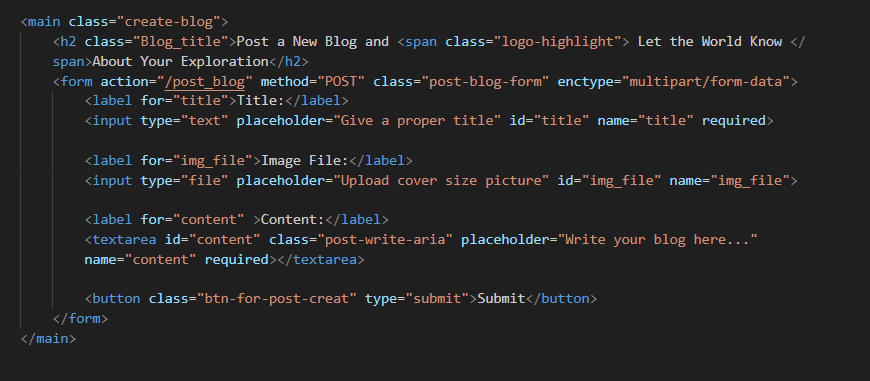
A loading animation is displayed before the main content loads, using a custom CSS animation and JavaScript function.

### . ****Main Content: Blog Creation Form****

The core functionality of the post\_blog.html page lies in the **Blog Creation Form**, which allows admins to input details about the new blog post.

#### ****Form Structure****

**Form Action and Method:** The form sends a POST request to the /post\_blog endpoint, with enctype="multipart/form-data" allowing file uploads.



**Form Fields:** The form includes fields for the blog title, image upload, and content:

* + **Title:** <input type="text" id="title" name="title" required> captures the blog title, which is essential for SEO and user interest.
  + **Image Upload:** <input type="file" id="img\_file" name="img\_file"> allows admins to upload a cover image to enhance the visual appeal of the post.
  + **Content:** <textarea id="content" name="content" required> is a rich text area where the blog content is written.

**Submit Button:** <button class="btn-for-post-creat" type="submit">Submit</button> sends the form data to the server, creating a new blog post upon submission.

### ****Styling and External Links****

* **CSS Integration**: <link rel="stylesheet" href="{{ url\_for('static', filename= 'style.css') }}"> connects the main stylesheet, ensuring consistent styling throughout the page.
* **JavaScript File Link**: <script src="{{ url\_for('static', filename='script.js') }}"></script> connects to external JavaScript, which may include additional scripts for interactive functionality on the website.

The **post\_blog.html** page provides a straightforward and secure interface for admins to create new blog posts. This page is integral to maintaining a high-quality content strategy on the travel website, allowing admins to share travel insights, tips, and stories with users. The loading animation and well-structured form enhance the user experience, while restricting access to admins ensures that only authorized users can publish content.

Through the admin-exclusive blog creation page, the travel website can continue to grow as a trusted resource for travelers, enriching their journey with insightful, reliable information about destinations worldwide. Check code: <https://github.com/tanvir-ahamed04/Travel-Website/blob/main/templates/post_blog.html> .

**Blog Display Interface for Users in the Travel Guide Website**

If an admin posts a new blog entry, users need to have a way to view that content. This is achieved through the **post\_blog.html** page, which serves as a public interface for displaying blog posts created by the admin. This page dynamically lists all available posts, showing a summary for each post, along with a link to read more if the user is interested in the full content.

### ****Structure and Functionality of**** post\_blog.html

The **post\_blog.html** page is designed to provide a streamlined viewing experience for users. It is composed of key sections: a navigation menu, the blog post list, and individual post details. This structure ensures users can easily access, explore, and read through blog entries created by the admin.

### ****HTML Structure and Components****

#### ****DOCTYPE Declaration and Head Section****

* The document begins with <!DOCTYPE html> to ensure standards-compliant rendering across browsers.
* The <head> section sets essential metadata, such as UTF-8 character encoding and viewport configuration, which ensures responsiveness on all devices.
* A favicon is specified via the <link rel="icon">, creating a branded experience.
* The main stylesheet is linked with <link rel="stylesheet" href="{{ url\_for('static', filename= 'style.css') }}"> for consistent styling across elements.

#### ****Navigation Header****

The <header> section contains the **Navigation Bar**, which includes

### ****Main Content: Blog Post List****

The main purpose of the **post\_blog.html** page is to display a list of blog posts. The content is wrapped in the <section class="post-list"> tag, which contains dynamic elements for each post.

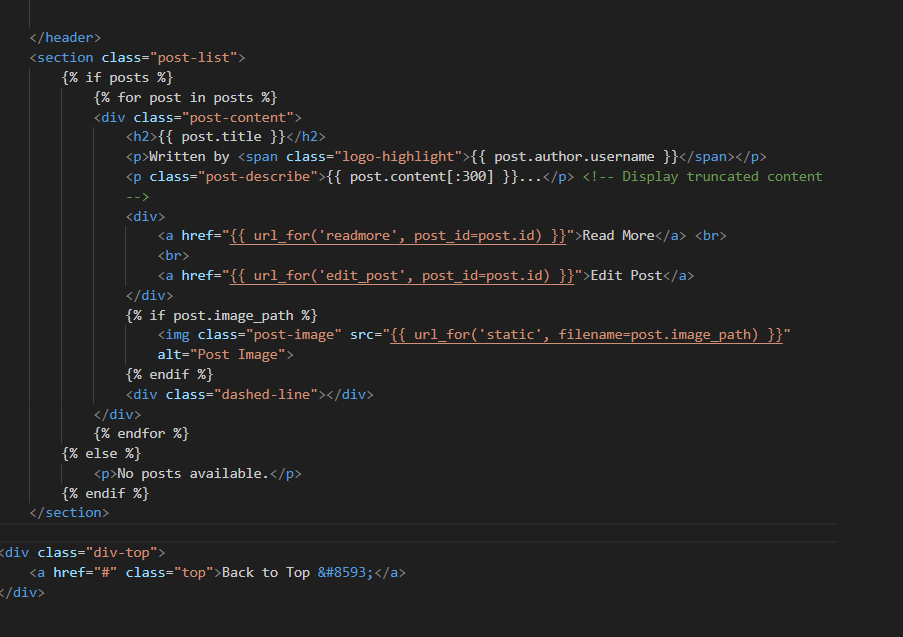
#### ****Dynamic Post Display Using Jinja Templating****

The {% if posts %} block checks if any posts are available. If posts exist, each one is rendered using the {% for post in posts %} loop. This allows the page to display a dynamic number of blog posts based on the content in the database.

#### ****Post Details and Truncated Content****

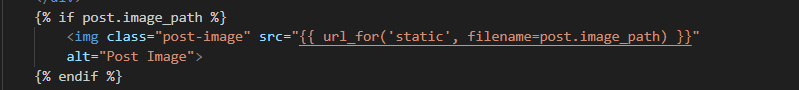
Each blog post is displayed within a <div class="post-content"> structure, including:

* **Title:** Displayed with <h2>{{ post.title }}</h2>, showing the post’s title prominently to attract user attention.
* **Author Name:** <p>Written by <span class="logo-highlight">{{ post.author.username }}</span></p> specifies the author, giving credit to the admin or author.
* **Content Preview:** The <p class="post-describe">{{ post.content[:300] }}...</p> line displays the first 300 characters of the content as a preview, encouraging users to click on "Read More" for full details.



#### ****Interactive Elements****

* **Read More Link:** The <a href="{{ url\_for('readmore', post\_id=post.id) }}">Read More</a> link directs users to a detailed view of the post, allowing users to engage with the full content.
* **Edit Post Link (Admin-Only):** For admin users, the <a href="{{ url\_for('edit\_post', post\_id=post.id) }}">Edit Post</a> link allows editing of posts, adding flexibility in case content updates are needed.
* **Post Image (Optional):** If an image is associated with the post, it is displayed using:



This conditional ensures only posts with images display a visual, enhancing the post’s appearance.

#### ****Section Divider****

Each post ends with <div class="dashed-line"></div>, a styled divider line that separates posts visually, improving readability by clearly distinguishing one post from another.

### ****Back to Top Button****

A "Back to Top" link, <a href="#" class="top">Back to Top &#8593;</a>, is included at the end of the page. This button provides a smooth user experience by allowing users to quickly navigate back to the top of the page, especially helpful on pages with multiple posts.

JavaScript for Interactivity:

<script src="{{ url\_for('static',filename='script.js') }}"></script>

This JavaScript may include additional functionality, such as animations, smooth scrolling, or form validation, enhancing the interactive experience.

The **post\_blog.html** page is essential for delivering content to the end-users, showcasing posts in an appealing and structured manner. The page’s layout and design ensure that users can quickly navigate through and engage with the travel content posted by the admin. The use of Jinja templating makes it dynamic, displaying posts as they are added and allowing for efficient updates without manual modifications.

**Blog Editing and Deletion Interface for Admins on the Travel Guide Website**

What if the admin makes a mistake while posting? In such cases, it’s crucial to have an option to correct or delete the post. The **edit\_post.html** page provides this functionality, allowing the admin to edit existing posts or delete them entirely if needed. This report explains the structure of edit\_post.html and its associated Flask route.

### ****Structure and Functionality of**** edit\_post.html

The **edit\_post.html** page is designed specifically for admins to update or delete posts. It includes a form where the post’s title, content, and optional image can be modified. Additionally, there’s a delete button that allows the admin to remove the post from the database if necessary.

### ****HTML Structure and Components****

#### ****DOCTYPE Declaration and Head Section****

* The document begins with <!DOCTYPE html> to ensure standard HTML rendering across browsers.
* The <head> section sets metadata such as UTF-8 encoding and viewport settings, making the page responsive on all devices.
* The favicon and stylesheet links are set up to enhance branding and consistent styling.

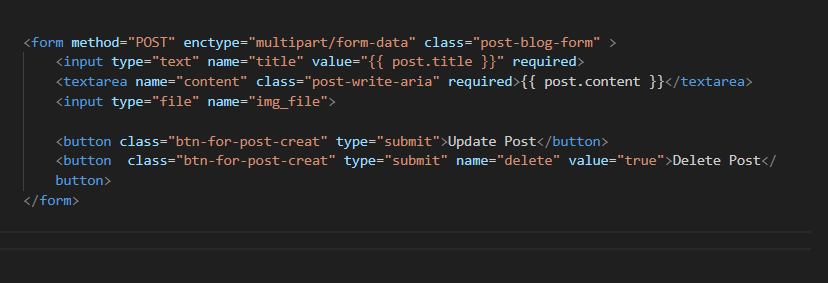
#### ****Navigation Header:**** Similar to other pages

### ****Main Content: Edit Form****

The primary functionality of the **edit\_post.html** page is encapsulated in a form that allows the admin to update or delete a post.

#### ****Form Fields for Editing****

The form contains fields that populate with the current post information, allowing the admin to easily make updates:



* **Title Input:** <input type="text" name="title" value="{{ post.title }}" required> displays the existing title, which can be edited directly.
* **Content Textarea:** <textarea name="content" class="post-write-aria" required>{{ post.content }}</textarea> displays the post content, allowing edits.
* **Image Upload:** <input type="file" name="img\_file"> allows the admin to upload a new image. If a new image is uploaded, it replaces the current one.

#### ****Update and Delete Buttons****

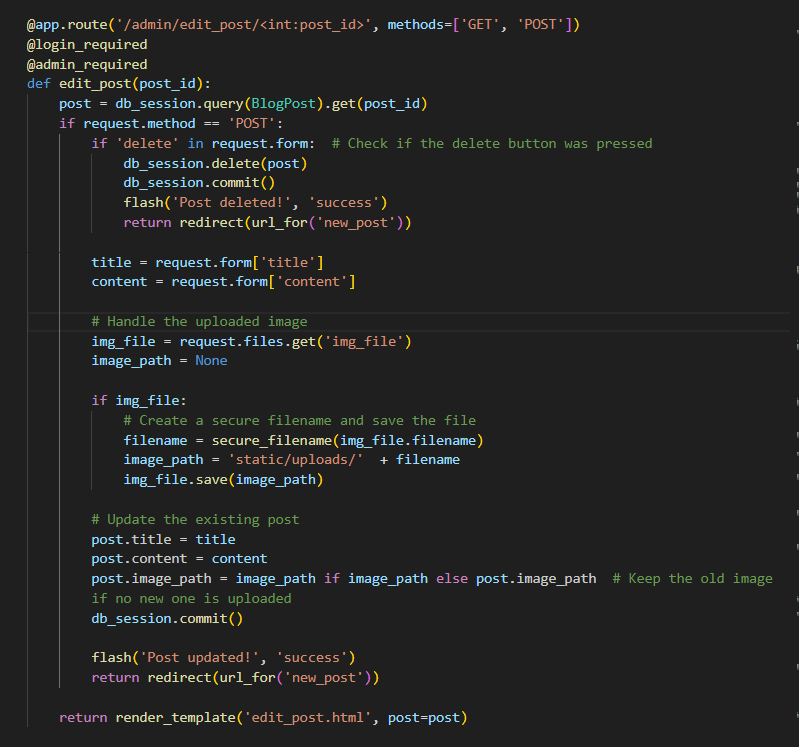
* **Update Button:** <button class="btn-for-post-creat" type="submit">Update Post</button> submits the form to save changes.
* **Delete Button:** <button class="btn-for-post-creat" type="submit" name="delete" value="true">Delete Post</button> allows the admin to delete the post. When pressed, it triggers deletion in the backend.

### 5. ****Loading Animation****

The page includes a loading screen, which simulates progress until the main content is ready. This is managed by JavaScript, which increments the loading percentage until reaching 100%, enhancing user experience.

### ****Flask Route:**** /admin/edit\_post/<int:post\_id>

The backend logic for the edit and delete functionality is handled by the edit\_post route. Here’s a breakdown of how it works:



#### Route Explanation

1. **GET Request:** When the page is accessed with a GET request, it renders the existing post’s data in the edit form, allowing the admin to review and modify the content.
2. **POST Request for Update:** If the form is submitted without the "delete" flag, the route retrieves updated title and content values. If a new image is uploaded, it replaces the existing image. After updating, changes are saved to the database.
3. **POST Request for Delete:** If the "delete" button is pressed, the form sends a delete flag in the request. The route deletes the post from the database, and the admin is redirected to new\_post with a success message.

### ****JavaScript for Interactivity****

The loading animation script runs upon page load, providing a smooth user experience by displaying progress until the page fully loads.

The **edit\_post.html** page offers a simple and functional interface for admins to manage blog posts. Its structured layout and clear update/delete options allow the admin to efficiently modify or remove content, ensuring that the website’s content remains accurate and up-to-date.

This route and page structure empower admins with easy-to-use tools, supporting their role in maintaining the quality and relevance of the travel content displayed on the website.

### ****Detailed View: Read More Feature for Blog Posts after 700 words****

When users browse through all the blog posts on the website in the new\_post.html page, each post is displayed in a truncated format, showing only the first 300 words to give users an overview without overwhelming the page layout. However, if a user is interested in reading the entire content of a particular blog post, the **Read More** feature allows them to access the full details. This is managed through a separate template called readmore.html, which provides a dedicated page to showcase the complete content of a selected post.

### ****1. Functionality Overview****

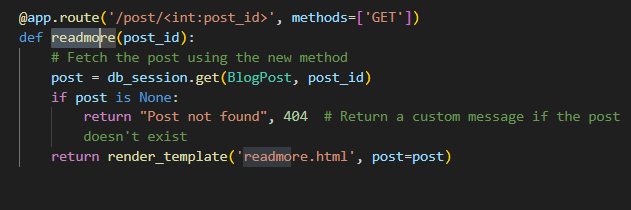
The **Read More** functionality enables users to:

* View the entire content of a blog post by navigating to a dedicated page.
* Access the post title, author’s name, full content, and any associated images.

This feature provides a seamless reading experience, allowing users to switch from a summarized view to an in-depth reading mode.

### ****2. Backend Route Configuration****

The route responsible for fetching and displaying the full content of a specific post is defined as follows:



**Explanation of the Route:**

**Endpoint:** /post/<int:post\_id>

* + This dynamic URL takes post\_id as a parameter, allowing it to handle requests for any specific post in the database by fetching it via its unique identifier.

**Method:** GET

* + The route is configured to respond to GET requests only, ensuring it is used for displaying information rather than modifying it.

**Fetching the Post:**

* + The route retrieves the post with the specified post\_id from the BlogPost table using db\_session.get(). If the post does not exist, it returns a 404 error with a custom message, "Post not found."

**Rendering the Template:**

* + If the post is successfully fetched, it renders the readmore.html template, passing the post data to it for display.

### ****3. Template Structure in**** readmore.html

The readmore.html template is designed to display the full content of a blog post, including the title, author, body content, and image (if available). Below is the structure of the readmore.html template:



**Explanation of the Template Structure:**

**Title (**<h2>{{ post.title }}</h2>**)**: Displays the title of the post, making it the most prominent element on the page for easy identification by the reader.

**Author Information (**<p>Written by <span class="logo-highlight">{{ post.author.username }}</span></p>**)**: Shows the author’s name, providing attribution and adding a personal touch.

**Full Content (**<p>{{ post.content }}</p>**)**: Renders the full text of the blog post, allowing readers to access the entire article instead of the truncated preview shown on new\_post.html.

**Image Display (**{% if post.image\_path %}**)**: Checks if the post has an associated image path. If an image exists, it is displayed below the content, enhancing the visual appeal and providing context to the article.

### ****4. User Experience and Benefits****

**Increased Engagement**: By showing a “Read More” option in new\_post.html, users can choose to view full posts, encouraging deeper engagement and prolonged time on the website.

**Optimized Layout**: Displaying posts in a summarized format in new\_post.html keeps the layout clean and user-friendly. Only interested users click through to readmore.html to access the full post, allowing for efficient use of screen space on the main blog page.

**Improved Navigation**: The /post/<int:post\_id> route facilitates a straightforward transition from the main blog page to the detailed view, enhancing user navigation and ease of access.

### ****5. Summary of Code and Database Integration****

This feature integrates smoothly with the backend database:

* The route retrieves posts using SQLAlchemy’s db\_session.get() function, ensuring secure and efficient data fetching.
* Only posts with valid IDs are displayed, with a fallback 404 error message for non-existent posts, enhancing the robustness of the application.

By maintaining this separation of views—summary in new\_post.html and full content in readmore.html—the website provides users with a streamlined, efficient, and interactive experience that caters to diverse reading preferences.

**Profile Function: profile.html**

In user profile, user can change users name, e-mail and password. For that at first we have to define the a new route for profile.html in our main.py file. And also in that page we have some fuction as changing user details from database.

#### Routes Overview

**Change Username Route (**/change\_username**)**:

* 1. **Purpose**: This route allows the logged-in user to change their username.
  2. **Methods**: POST

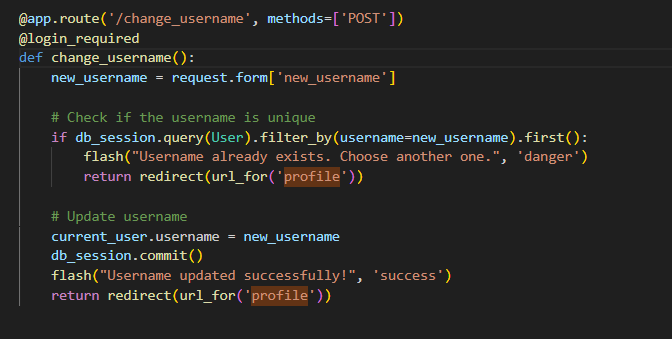
**Validation**:

* + 1. The route checks whether the new username already exists in the database using the filter\_by method. If the username exists, a flash message is shown, and the user is redirected back to their profile page.
    2. If the username is unique, it updates the current\_user.username attribute and commits the change to the database.

**Flash Messages**:

* + 1. "Username already exists. Choose another one." if the username is not unique.
    2. "Username updated successfully!" if the update is successful.

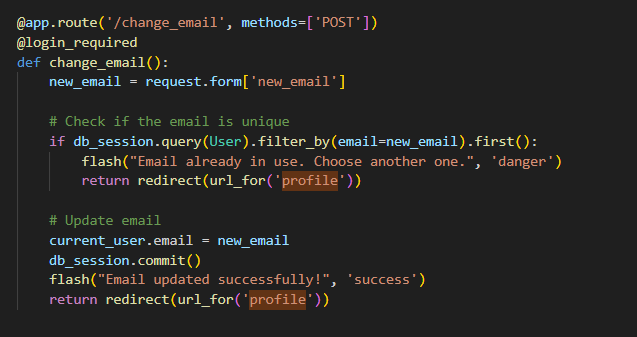
**Code Implementation**:



**Change Email Route (**/change\_email**)**:

* **Purpose**: This route allows the logged-in user to change their email address.
* **Methods**: POST
* **Validation**:
  + The route checks whether the new email already exists in the database using the filter\_by method. If the email exists, a flash message is displayed, and the user is redirected back to the profile page.
  + If the email is unique, it updates the current\_user.email attribute and commits the change to the database.
* **Flash Messages**:
  + "Email already in use. Choose another one." if the email is not unique.
  + "Email updated successfully!" if the update is successful.

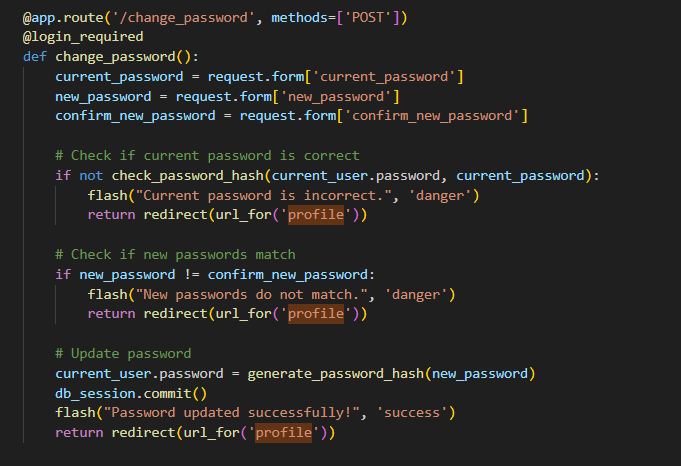
**Code Implementation**:



**Change Password Route (**/change\_password**)**:

* **Purpose**: This route allows the logged-in user to change their password.
* **Methods**: POST
* **Validation**:
  + The route first checks if the provided current\_password matches the stored password using check\_password\_hash. If it doesn't match, a flash message is displayed, and the user is redirected back to their profile.
  + It then verifies if the new\_password matches the confirm\_new\_password. If they don't match, another flash message is shown.
  + If both checks pass, it updates the current\_user.password with the hashed version of the new password using generate\_password\_hash and commits the change to the database.
* **Flash Messages**:
  + "Current password is incorrect." if the current password is incorrect.
  + "New passwords do not match." if the new passwords don't match.
  + "Password updated successfully!" if the update is successful.

**Code Implementation**:



### Explanation

**Login Required**: All three routes are protected with the @login\_required decorator, ensuring that the user must be authenticated before accessing them. If the user is not logged in, they will be redirected to the login page.

**Database Interaction**:

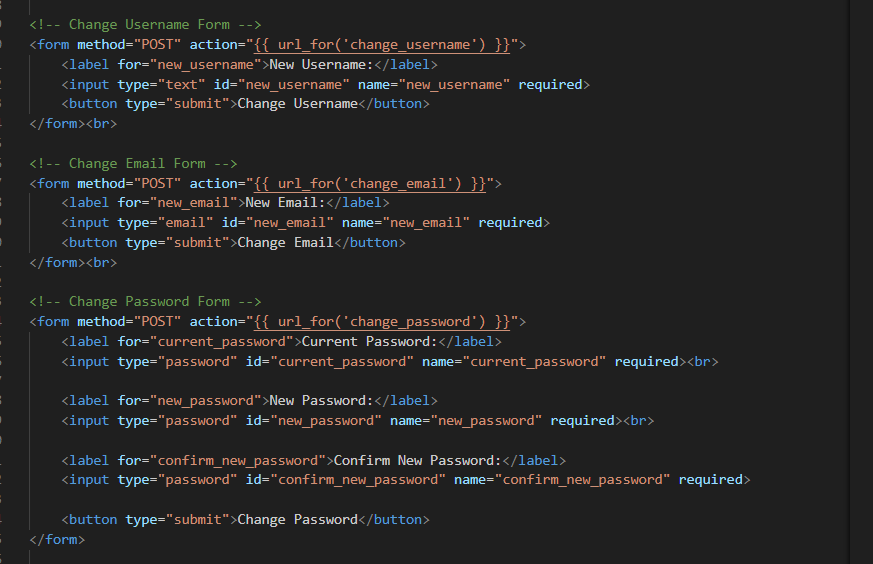
* 1. The routes interact with the database using SQLAlchemy queries to check for unique usernames and emails and to update the user's information.
  2. For username and email changes, it uses the db\_session.query(User).filter\_by() method to check for duplicates.
  3. For the password change, it uses the check\_password\_hash and generate\_password\_hash functions from werkzeug.security to validate and update the user's password securely.

**Flash Messages**: Flash messages are used to provide feedback to the user about the success or failure of the operation. These messages are displayed on the profile page using the get\_flashed\_messages() function in the profile.html template.

**Redirection**: After each update (username, email, or password), the user is redirected back to their profile page, where they can see the changes or any error messages.

### Usage in the Profile Template (profile.html)

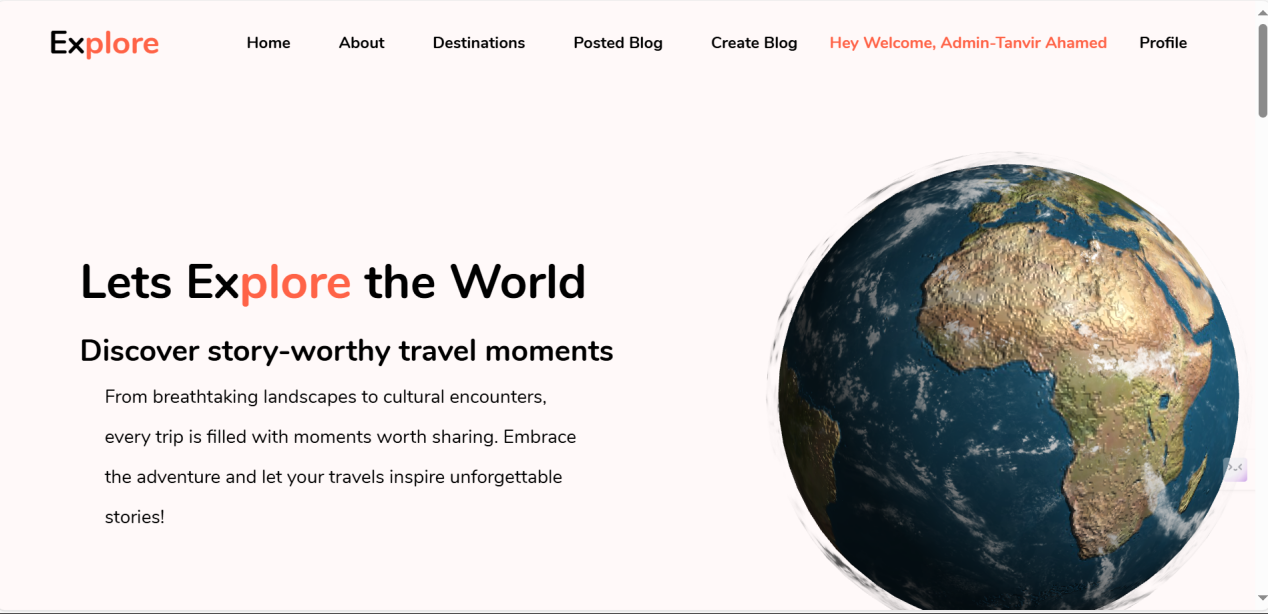
The profile page will have forms for updating the username, email, and password. Each form sends a POST request to the corresponding route. For example, for changing the username



These routes allow users to update their personal information (username, email, and password) in a secure and efficient manner. They incorporate validation checks, flash messages for user feedback, and ensure that only authenticated users can make changes to their profiles. The use of db\_session.commit() ensures that the changes are committed to the database, while the login\_required decorator maintains user session integrity.

### **CSS and Styling (style.css)**

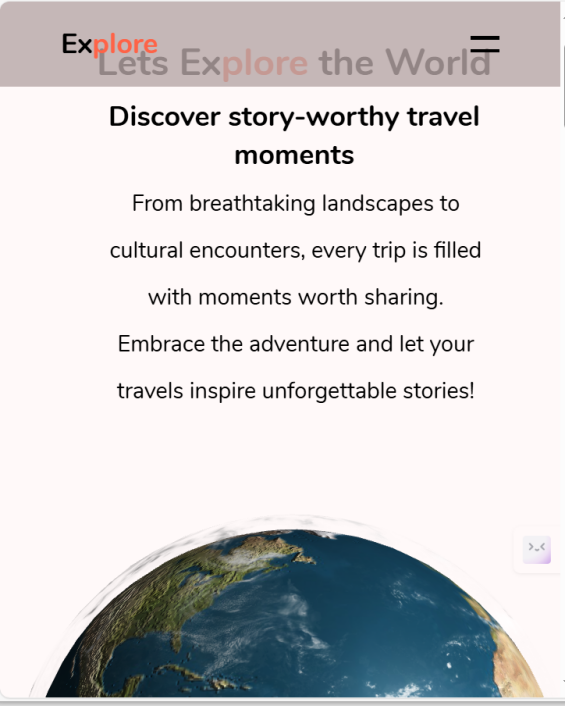
CSS played a crucial role in the design and presentation of the web application. It was used to style the user interface, ensuring that the website was not only functional but also visually appealing and user-friendly. The application’s layout and design were created with responsiveness in mind, ensuring a seamless experience across various devices.



**Layout and Structure**: CSS was employed to define the overall structure of each page, including the positioning of elements, navigation bars, and content areas. Flexbox and Grid layouts were used to create a flexible, adaptable design that could adjust based on the screen size.

**User Interface Design**: Through the use of CSS, the application was styled to ensure a clean, modern, and user-friendly interface. Font styles, color schemes, button designs, and form input elements were carefully chosen to enhance the overall aesthetic and ensure accessibility.

**Responsiveness**: Media queries were used to ensure the web application’s layout was responsive. These queries allowed the design to adapt to different screen sizes, ensuring the website remained usable and visually consistent on mobile, tablet, and desktop devices. These two pictures are showing that the web-page is responsive.

**Interaction and Feedback**: CSS was also used for creating interactive elements such as hover effects on buttons, form validation feedback, and smooth transitions between pages and elements. This helped enhance the user experience by providing visual cues and making the application feel more dynamic.

By using CSS effectively, the project not only met the technical requirements but also prioritized user experience, offering a clean, functional, and visually appealing website.

**Java script (script.js, Three.js, app.js)**

### ****app.js: Earth Animation and Scene Creation (Three.js)****

The **app.js** file is responsible for creating a 3D animation of the Earth using **Three.js**. Below is a breakdown of the core features:

**Scene Setup**: The scene is created using THREE.Scene(), and a THREE.PerspectiveCamera is added to define the viewing perspective of the 3D world.

**Renderer Setup**: The THREE.WebGLRenderer is initialized to handle the rendering of the scene to the canvas, with anti-aliasing and transparency enabled. It automatically adjusts the canvas size based on the window dimensions.

**Earth Creation**:

* 1. The Earth is represented by a **SphereGeometry** object with a texture applied to it using a THREE.MeshPhongMaterial. The texture files (earthmap1k.jpg and earthbump.jpg) are loaded using the THREE.TextureLoader().
  2. The Earth's surface includes bump mapping to create a more realistic 3D effect.

**Lighting**:

* 1. An **ambient light** is added to illuminate the scene globally, providing softer lighting.
  2. A **point light** is positioned in space to cast focused light onto the Earth and clouds.

**Cloud Layer**:

* 1. Another **SphereGeometry** is used to create a cloud layer around the Earth. A semi-transparent texture (earthCloud.png) is applied to simulate cloud coverage, and it rotates to create the illusion of movement.

**Animation**:

* 1. The animation function (animate()) is called recursively using requestAnimationFrame(). This updates the Earth's and cloud's rotation, creating a continuous spinning effect.

**Rendering**: The render() function renders the scene from the perspective of the camera, making the 3D animation visible on the canvas.

### ****script.js: Navigation and Interactivity****

This file handles various interactive features of the website:

**Navigation Menu**:

* 1. A hamburger icon (.hamburger) is used to toggle the visibility of the navigation links (.nav-links). When clicked, the class active is toggled to show or hide the menu.

**Typed.js (Typing Effect)**:

* 1. The Typed.js library is used to create a typing animation on elements with the class .typing. It types out messages like "Where will your next adventure take you?" and "Embark on a journey to uncover the world's hidden gems!", adding an engaging and dynamic effect to the webpage.
  2. The typing effect has customizable options for typing speed, backspacing, and looping.

**Scroll Behavior**:

* 1. The scroll event listener monitors the user's scroll position. When the user scrolls past 50px, the header's color changes to white, and the scrolled class is added to the header. This effect provides visual feedback to users as they scroll through the page.

**Collapsible Content Sections**:

* 1. Three content sections (howItWorksContent, localExpertsContent, and whatsIncludedContent) can be toggled by clicking their respective buttons. The toggleContent() function hides or shows the content when a button is clicked, allowing users to expand or collapse sections as needed.

### ****General Observations****:

* **Interactive Design**: The combination of **Three.js** (for 3D Earth animation) and **JavaScript/jQuery** (for dynamic elements) creates an interactive, visually engaging website. The Earth animation adds a unique and captivating feature to the site.
* **Navigation and UX**: The hamburger menu, collapsible content sections, and smooth scroll effects help improve the user experience by providing an intuitive, responsive interface.

This combination of dynamic visual effects (3D Earth), smooth UI interactivity (typed text, collapsible sections), and enhanced navigation makes the website both functional and engaging.

**Three.js**

**It** is a JavaScript library that simplifies the process of creating 3D graphics in a web browser. It leverages WebGL to render interactive 3D content without requiring external plugins. Three.js abstracts much of the complexity of WebGL, offering developers an easier way to create 3D environments, models, and animations. The library supports various features such as scene creation, geometries, lighting, shadows, camera control, and animations, which help in building immersive 3D experiences directly on the web.

**Conclusion**

Now that we have developed and tested the web application, the project is complete. The web application integrates key components, including a server-side framework (Flask), dynamic user interfaces enhanced with JavaScript/jQuery, and a robust database integration for managing user data and posts. The application is designed with a user-centered approach, providing an intuitive experience and interactive features such as dynamic navigation, real-time content updates, and 3D Earth visualization using Three.js.

The website's clean and responsive design, backed by the backend server, ensures smooth user interactions and functionality across different devices and screen sizes. The incorporation of multimedia elements like images, videos, and interactive 3D visuals further enhances the user experience, making the platform more engaging for travelers.

**How to Run the Application:**

To run the application locally, follow these steps:

1. **Clone the Repository:** Clone the project repository to your local machine using Git:

**git clone [https://github.com/tanvir-ahamed04/Travel-Website.git]**

1. **Install Dependencies:** Ensure you have Python and pip installed. Then, install the required Python libraries by running:

**pip install -r requirements.txt**

1. **Set Up the Database:**

* Ensure MySQL is running on your machine.
* Import the provided dump.sql file into your MySQL database to set up the necessary tables:

**mysql -u [username] -p < dump.sql**

1. **Configure Environment Variables:** Create a .env file with the following variables:

FLASK\_APP=main.py

FLASK\_ENV=development

MAIL\_SERVER=sandbox.smtp.mailtrap.io

MAIL\_PORT=2525

MAIL\_USERNAME=your-username

MAIL\_PASSWORD=your-password

MAIL\_USE\_TLS=True

MAIL\_USE\_SSL=False

If its defined inside the main.py then its not necessary.

1. **Run the Flask Application:** Start the Flask development server by running:

python main.py

The application should now be accessible in your browser at http://localhost:5000.

By following these steps, we can successfully run the application locally and explore its features. The project showcases the integration of front-end and back-end technologies to create a fully functional, dynamic web application with interactive and engaging elements.