PARULINSTITUTEOF ENGINEERING &TECHNOLOGY
FACULTY OF ENGINEERING & TECHNOLOGY
PARULUNIVERSITY

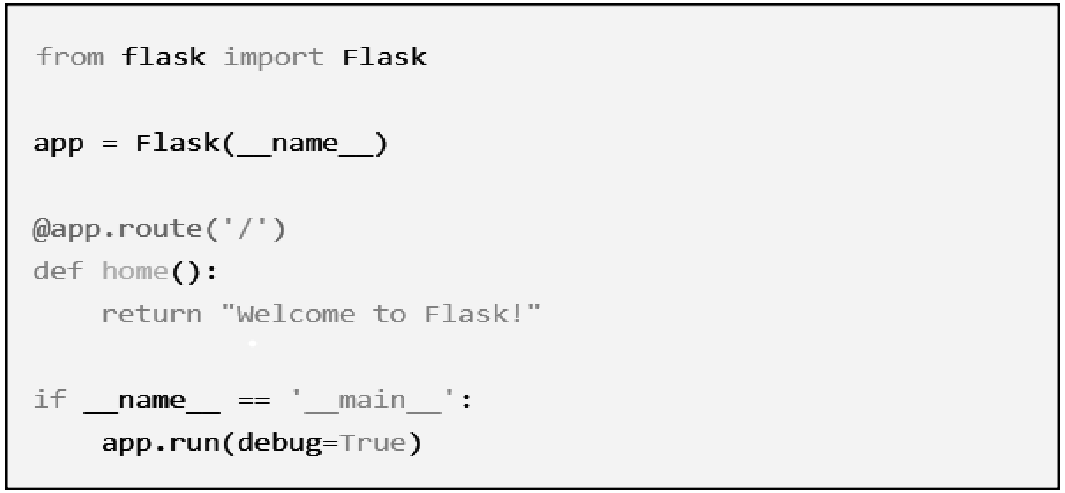
303105257 - Programming in Python with Full Stack Development

Introduction to Flask and Web Development with Python

Introduction to Flask and Web Development with Python

- Flask is a micro-web framework for Python, meaning it is lightweight, modular, and does not impose many dependencies.
- It is designed for developers who prefer flexibility and control over web application development.
- Flask is based on two core components:
- Werkzeug: A WSGI (Web Server Gateway Interface) toolkit that provides request and response handling.
- Jinja2: A templating engine used to render dynamic HTML templates.
- Flask is commonly referred to as "batteries not included" since it provides essential features but leaves additional functionality (like authentication, database handling, etc.) to extensions.

Contd….



```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to Flask!"

if __name__ == '__main__':
    app.run(debug=True)
```

- Flask(__name__): Creates the Flask application.
- @app.route('/'): Defines a route for the home page (/ URL).
- debug=True: Enables debugging features like auto-reloading and error tracking.

Installation in a Virtual Environment

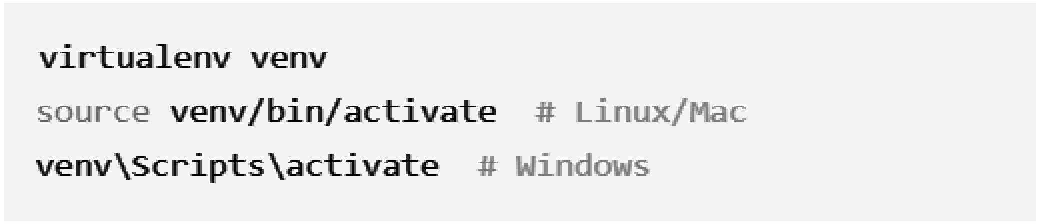
- A virtual environment is an isolated environment for Python projects, ensuring dependencies for one project don't interfere with another.
- Flask is not included in the standard Python library, so it must be installed in the project environment.
- **By creating a virtual environment:**
- You can maintain separate Python libraries for different projects.
- It prevents dependency conflicts between applications.

1. **Install virtualenv:**

```
pip install virtualenv
```

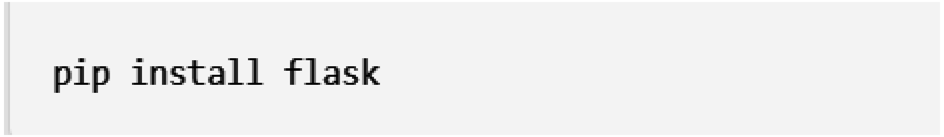
Contd..

- Create and activate a virtual Environment



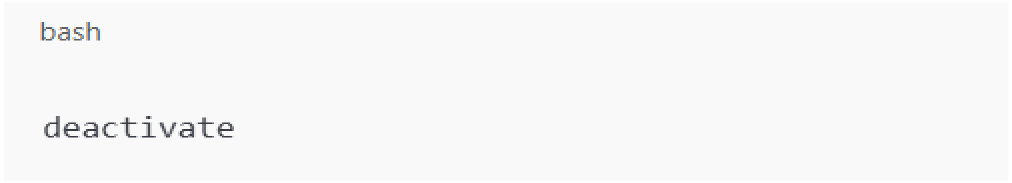
```
virtualenv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate  # Windows
```

- Install Flask



```
pip install flask
```

- Deactivate the virtual environment:



```
bash

deactivate
```

Routing and App Settings

- **Routing**:

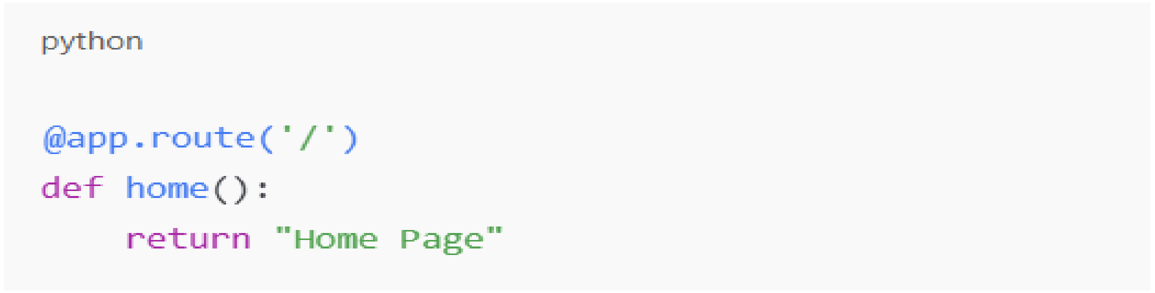
➔ Routing maps URLs to Python functions.

➔ In Flask, the @app.route() decorator is used to define URL endpoints.

➔ Flask supports dynamic routing by using variable placeholders in the URL (e.g., /user/<username>).

Examples:

- **Static Route**



```
python

@app.route('/')
def home():
    return "Home Page"
```

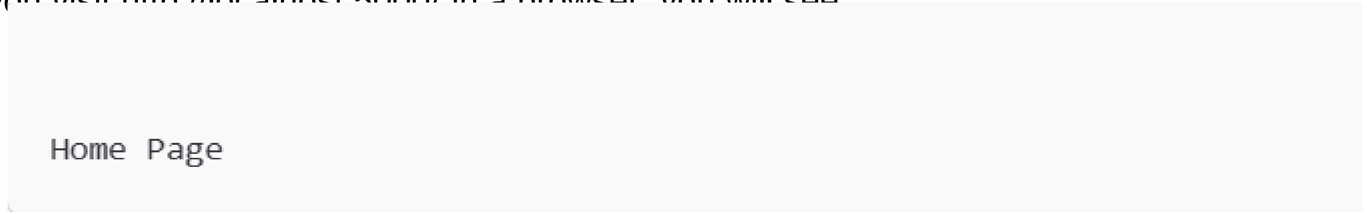
Static Routes:

Explanation:

- **@app.route('/'):**
  - The @app.route decorator in Flask is used to define a URL route for your application.
  - The '/' route represents the root URL of the application (e.g., http://localhost:5000/ or the base of your website).
- **Function home():**
  - The function home() is executed when a user visits the specified route (/).
  - It simply returns the text "Home Page", which will be displayed in the browser when accessing this route.

Example Output:

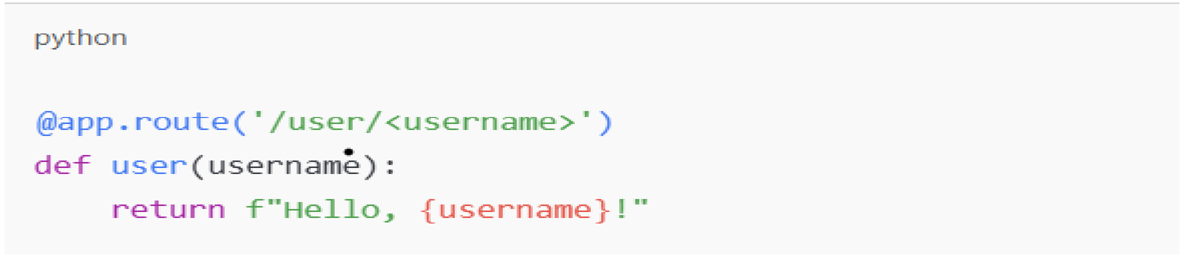
If you visit http://localhost:5000/ in a browser, you will see:



```
Home Page
```

Dynamic Route:

Example:



```python
@app.route('/user/<username>')
def user(username):
    return f"Hello, {username}!"
```

Explanation:

- @app.route('/user/<username>'):
  - The <username> inside the route is a dynamic segment.
  - Flask uses this segment to capture the value entered at the URL and passes it as an argument to the function.
  - For example, if you visit http://localhost:5000/user/John, the value "John" will be passed as the argument username.

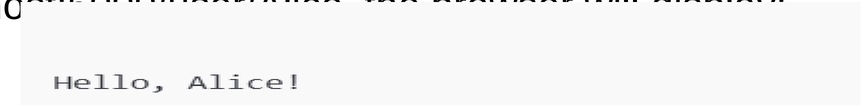
Contd…

- **Function user(username):**
  - This function takes the captured username from the URL and uses it.
  - It returns a personalized greeting like "Hello, John!".

- **String Interpolation (f-string):**
  - The f"Hello, {username}!" syntax is a formatted string literal that dynamically inserts the value of username into the response.

Example Output:

If you visit http://localhost:5000/user/Alice, the browser will display:

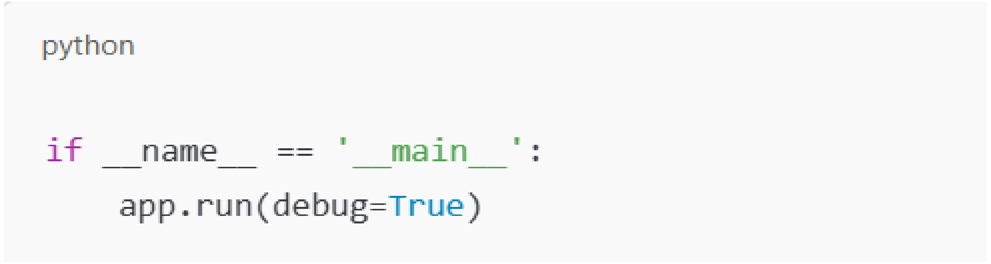


```
Hello, Alice!
```

App Settings:

- Flask provides settings for configuring debugging, secret keys, and database URIs.
- app.run(debug=True) enables the debug mode, which automatically reloads the server on code changes and shows detailed error messages.

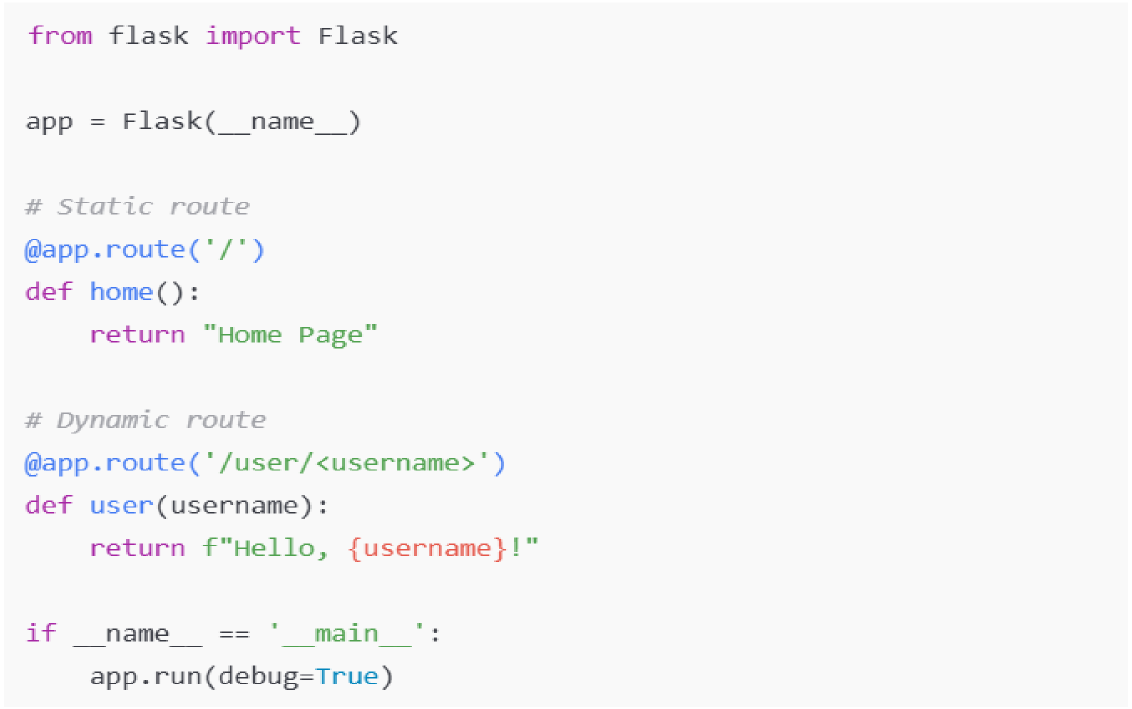
- Example:



```python
if __name__ == '__main__':
    app.run(debug=True)
```

Full Flask Application Example

Here’s how you can combine both static and dynamic routes in a Flask app:



```python
from flask import Flask

app = Flask(__name__)

# Static route
@app.route('/')
def home():
    return "Home Page"

# Dynamic route
@app.route('/user/<username>')
def user(username):
    return f"Hello, {username}!"

if __name__ == '__main__':
    app.run(debug=True)
```

URL Building and HTTP Methods

- **Flask provides tools for handling URL routing and HTTP methods effectively.**
- **URL Building:**

  URL building in Flask involves generating URLs for specific routes dynamically using the url_for() function.

  Why use URL Building?

- Hardcoding URLs can lead to errors, especially when the URL structure changes.
- url_for() dynamically generates URLs based on the function name of a route, making applications easier to maintain.

- **Syntax:**

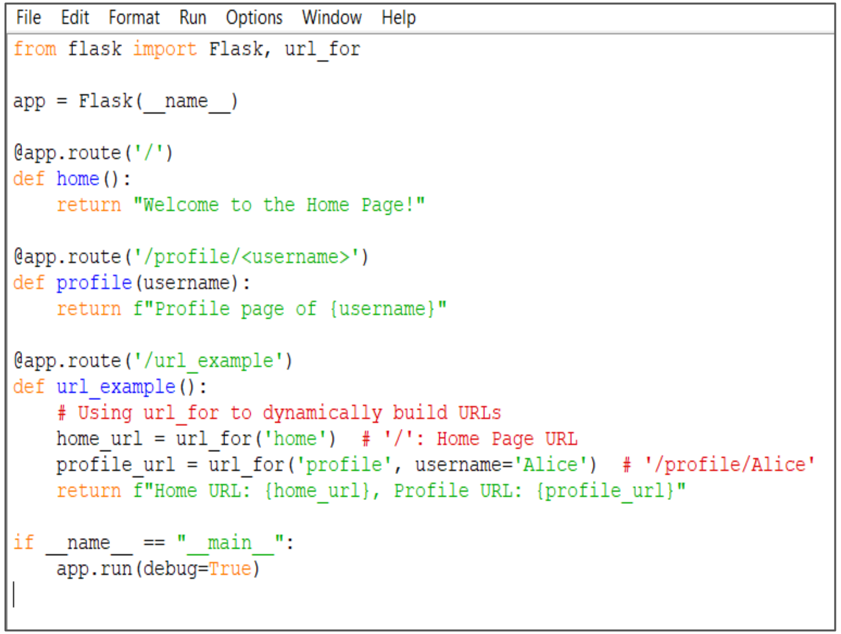
```
url_for(endpoint, **values)
```

  endpoint: The name of the function linked to a route.

  values: Key-value pairs for dynamic segments of the route.

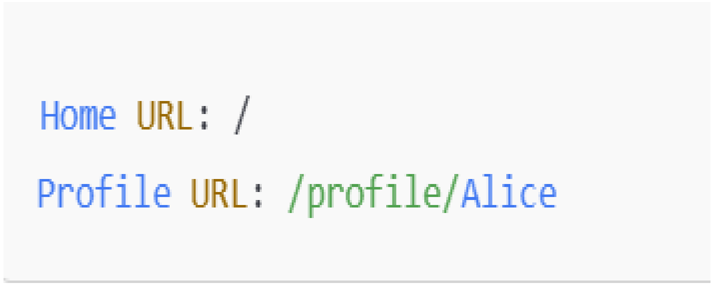
Contd..

Example:



```python
from flask import Flask, url_for

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to the Home Page!"

@app.route('/profile/<username>')
def profile(username):
    return f"Profile page of {username}"

@app.route('/url_example')
def url_example():
    # Using url_for to dynamically build URLs
    home_url = url_for('home')  # '/': Home Page URL
    profile_url = url_for('profile', username='Alice')  # '/profile/Alice'
    return f"Home URL: {home_url}, Profile URL: {profile_url}"

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

Output (Access /url_example):



```
Home URL: /
Profile URL: /profile/Alice
```

Benefits:

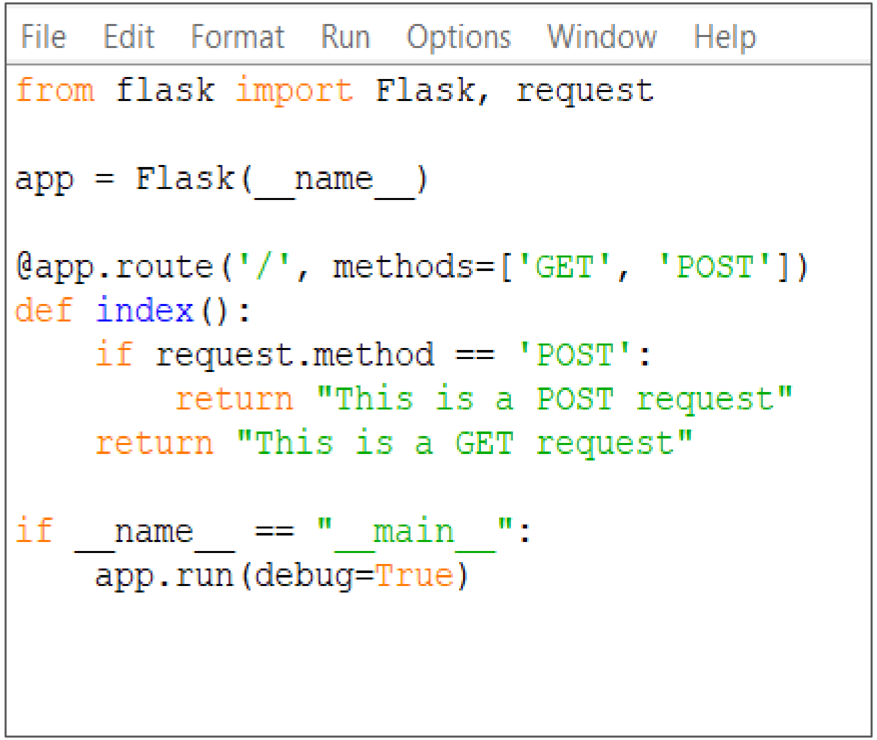
1. Ensures route consistency across the application.
2. Automatically includes query parameters and avoids manual URL assembly.
3. Useful in templates for generating links dynamically.

HTTP Methods

- HTTP methods determine the type of operation performed when a client (a browser) interacts with the server.
- Flask allows handling various HTTP methods (e.g., GET, POST, PUT, DELETE) via the methods argument in the @app.route() decorator.

Method	Purpose	Use Case
GET	Retrieve data from the server (default method).	Fetching a webpage or data.
POST	Send data to the server to create or update resources.	Submitting a form.
PUT	Update an existing resource.	Editing a record
DELETE	Remove a resource from the server.	Deleting a database entry.

Example 1: Handling GET and POST Methods



```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        return "This is a POST request"
    return "This is a GET request"

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

GET Request: Visit http://localhost:5000/ in your browser (default is GET).

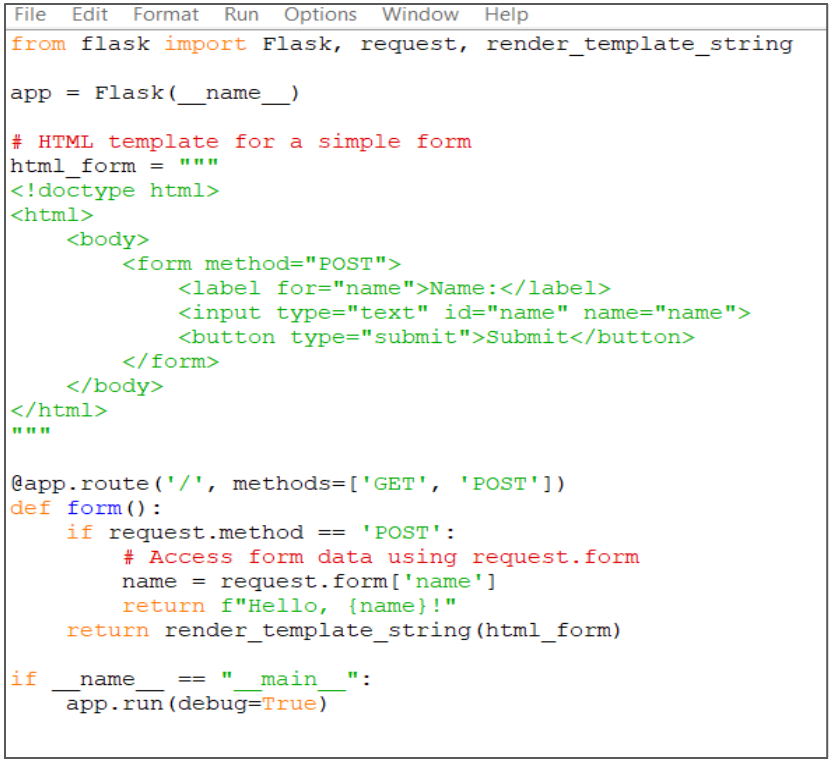
- Output: This is a GET request

POST Request: Use tools like Postman or curl:

```bash
curl -X POST http://localhost:5000/
```

Output: This is a POST request

Example 2: Form Submission with POST

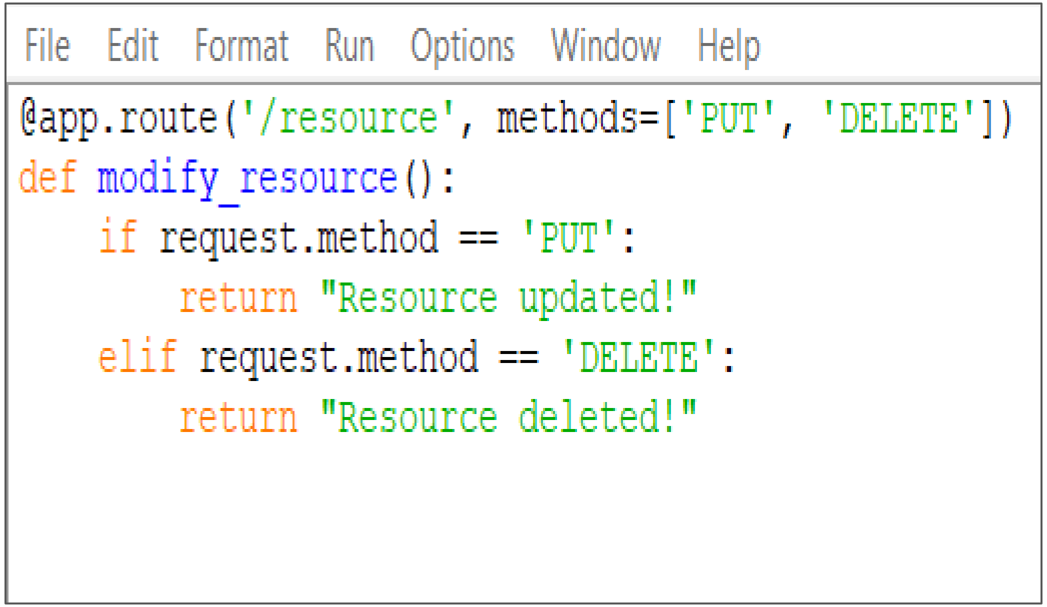


```python
from flask import Flask, request, render_template_string

app = Flask(__name__)

# HTML template for a simple form
html_form = """
<!doctype html>
<html>
    <body>
        <form method="POST">
            <label for="name">Name:</label>
            <input type="text" id="name" name="name">
            <button type="submit">Submit</button>
        </form>
    </body>
</html>
"""

@app.route('/', methods=['GET', 'POST'])
def form():
    if request.method == 'POST':
        # Access form data using request.form
        name = request.form['name']
        return f"Hello, {name}!"
    return render_template_string(html_form)

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

How it Works:

1. On GET request, it displays the form.
2. On POST request, it retrieves the input value from the form and displays a personalized greeting.

Example 3: Using PUT and DELETE



```
@app.route('/resource', methods=['PUT', 'DELETE'])
def modify_resource():
    if request.method == 'PUT':
        return "Resource updated!"
    elif request.method == 'DELETE':
        return "Resource deleted!"
```

PUT Request: Use curl or Postman:

curl -X PUT http://localhost:5000/resource

- Output: Resource updated!

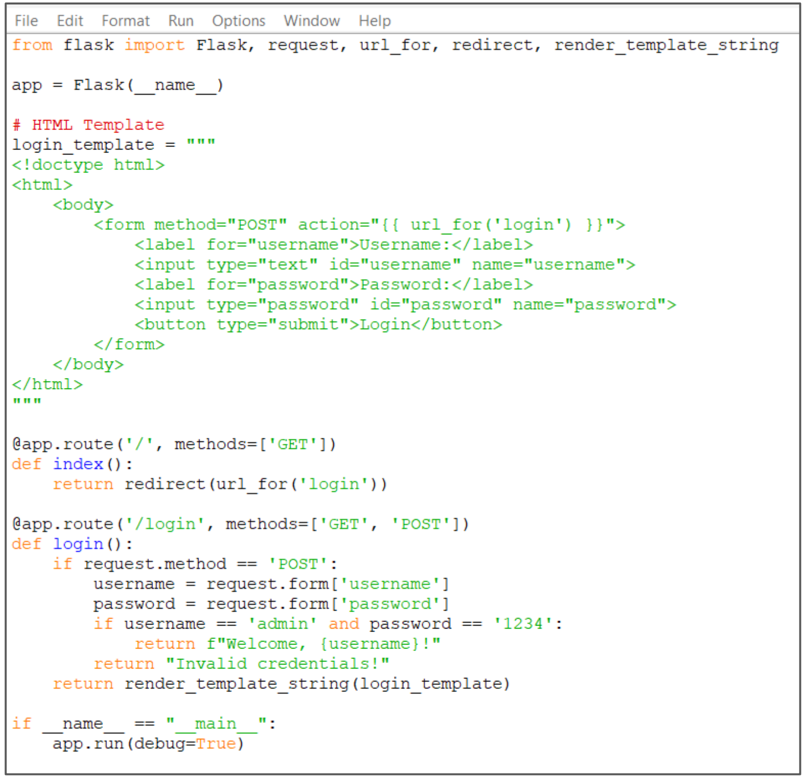
DELETE Request:

curl -X DELETE
http://localhost:5000/resource

- Output: Resource deleted!

Combining URL Building with HTTP Methods

Example: Login Form



```
from flask import Flask, request, url_for, redirect, render_template_string

app = Flask(__name__)

# HTML Template
login_template = """
<!doctype html>
<html>
    <body>
        <form method="POST" action="{{ url_for('login') }}">
            <label for="username">Username:</label>
            <input type="text" id="username" name="username">
            <label for="password">Password:</label>
            <input type="password" id="password" name="password">
            <button type="submit">Login</button>
        </form>
    </body>
</html>
"""

@app.route('/', methods=['GET'])
def index():
    return redirect(url_for('login'))

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        if username == 'admin' and password == '1234':
            return f"Welcome, {username}!"
        return "Invalid credentials!"
    return render_template_string(login_template)

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

1. **The form dynamically generates the action URL using url_for('login').**
2. **Handles both GET (display form) and POST (process login).**

Templates and Static Files

1. Templates in Flask

What Are Templates?

- Templates are **HTML files** used to dynamically generate content for web pages.
- Flask uses **Jinja2**, a templating engine, to embed Python-like expressions into HTML files.
- Templates allow passing data from the server to the client dynamically.

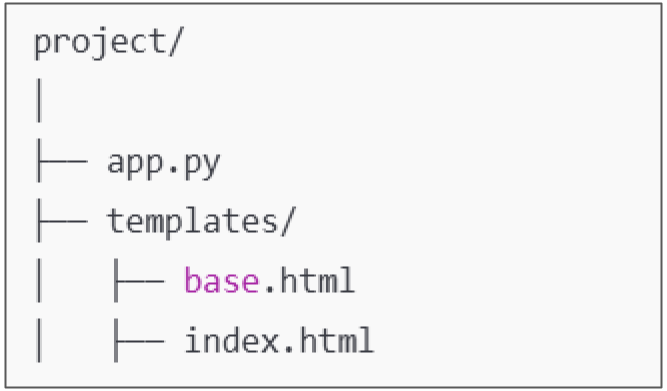
Concepts in Templates

1. **Dynamic Content**: Use placeholders (like {{ variable }}) to insert Python variables into HTML.
2. **Control Statements**: Use Jinja2 syntax for if conditions, for loops, etc.
3. **Template Inheritance** Create a base HTML structure and extend it in child templates to reuse common elements.

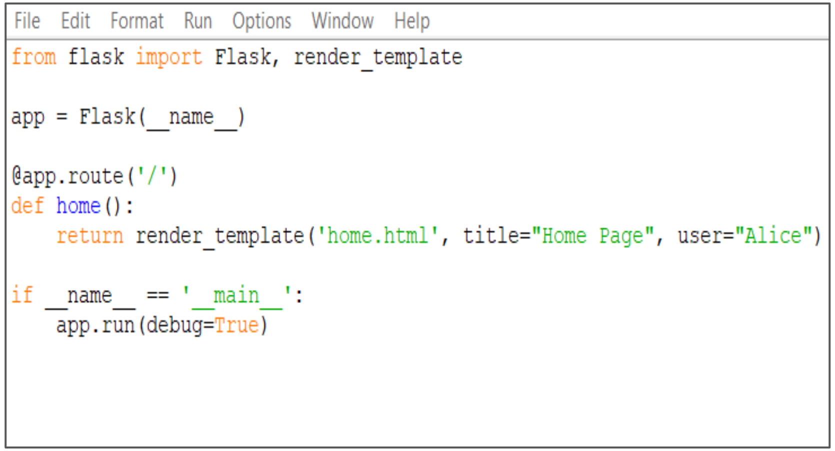
How Templates Work

- Flask looks for templates in a folder named **templates** by default.
- The render_template() function is used to load and render templates with dynamic data.

Example of Using Templates:



```
project/
|
├── app.py
├── templates/
|    ├── base.html
|    ├── index.html
```



```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('home.html', title="Home Page", user="Alice")

if __name__ == '__main__':
    app.run(debug=True)
```

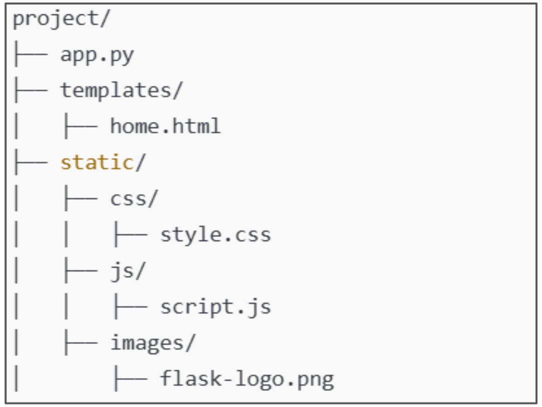
Static Files in Flask

- **Static files** are resources like CSS, JavaScript, images, fonts, or other assets that do not change dynamically.
- Flask serves static files from a folder named **static** by default.

How Static Files Work

- Static files are stored in the **static** folder.
- Use the url_for() function to generate URLs for static files dynamically.

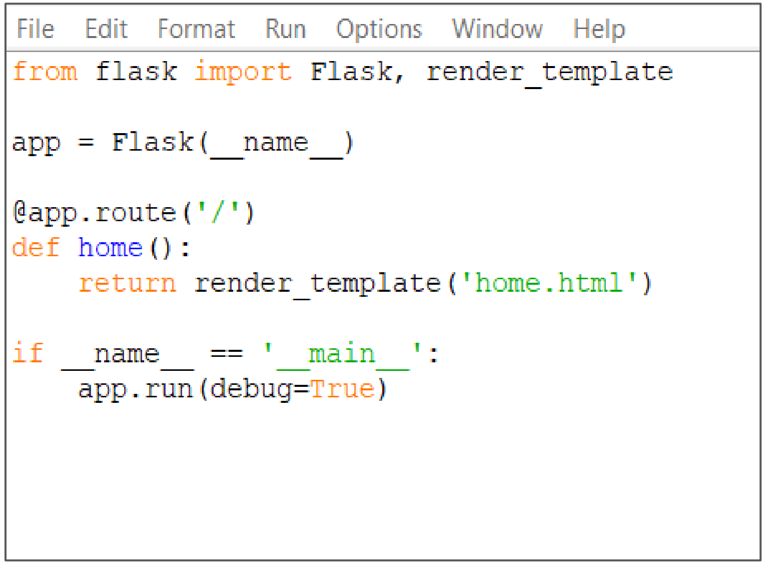
Example: Using static Files:



```
project/
├── app.py
├── templates/
│   ├── home.html
├── static/
│   ├── css/
│   │   ├── style.css
│   ├── js/
│   │   ├── script.js
│   ├── images/
│       ├── flask-logo.png
```

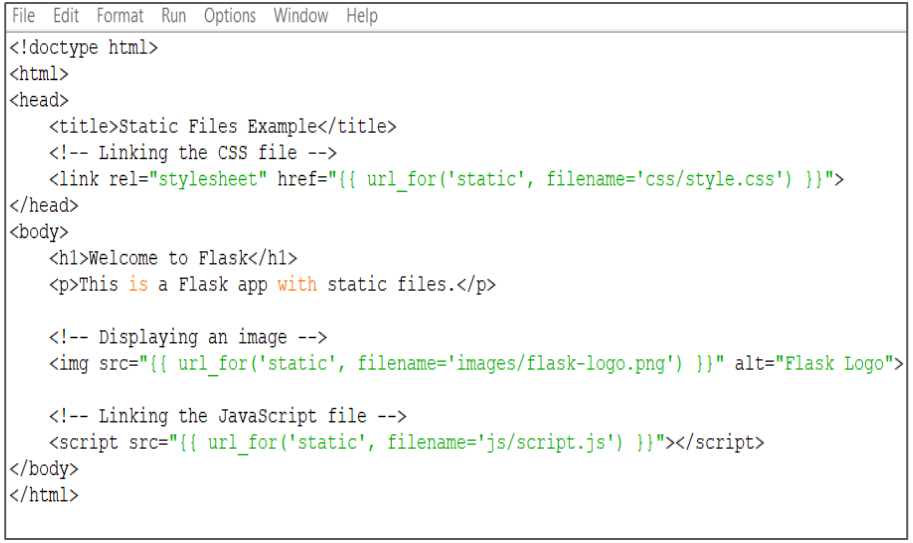
Contd…

Code (app.py):



```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('home.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Template (home.html):



```html
<!doctype html>
<html>
<head>
    <title>Static Files Example</title>
    <!-- Linking the CSS file -->
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
    <h1>Welcome to Flask</h1>
    <p>This is a Flask app with static files.</p>

    <!-- Displaying an image -->
    <img src="{{ url_for('static', filename='images/flask-logo.png') }}" alt="Flask Logo">

    <!-- Linking the JavaScript file -->
    <script src="{{ url_for('static', filename='js/script.js') }}"></script>
</body>
</html>
```

Static files

CSS File (style.css):

```css
body {
    font-family: Arial, sans-serif;
    background-color: #f0f0f0;
    color: #333;
}
h1 {
    color: #007BFF;
}
```

JavaScript File (script.js):

```javascript
document.addEventListener('DOMContentLoaded', function() {
    console.log("Static JavaScript is working!");
});
```

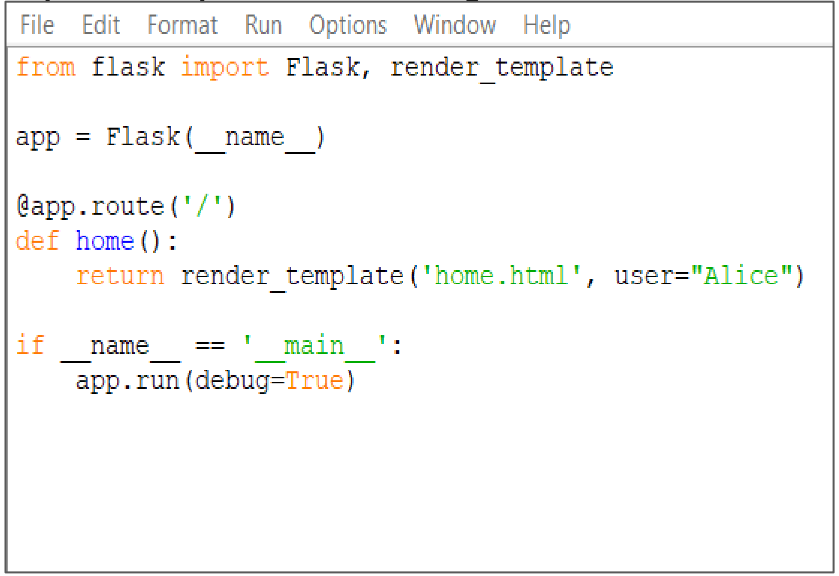
Output:

- The web page displays:
  1. Styled text from `style.css`.
  2. The Flask logo from `images/`.
  3. A console log message from `script.js` in the browser's developer tools.

Combining Templates and Static Files

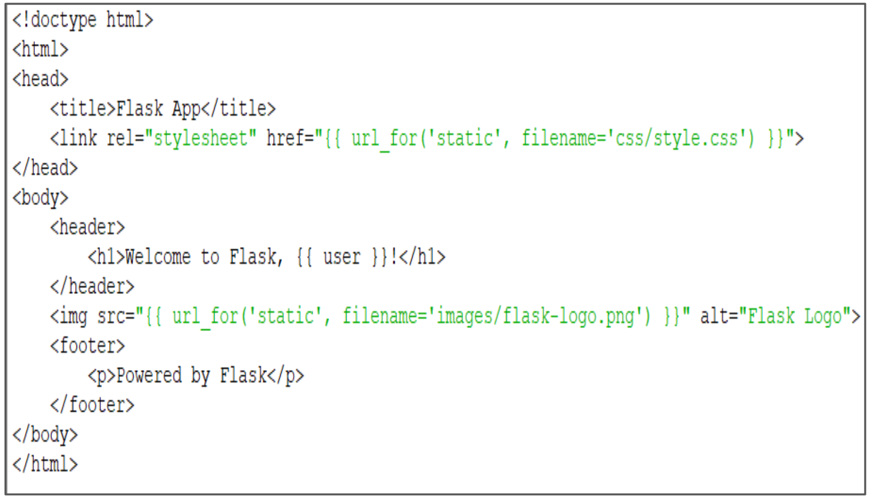
Flask integrates templates and static files seamlessly. Here's how:

Example: A Complete Flask Web Page



```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('home.html', user="Alice")

if __name__ == '__main__':
    app.run(debug=True)
```

`Templates/Home.html`:



```html
<!doctype html>
<html>
<head>
    <title>Flask App</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
    <header>
        <h1>Welcome to Flask, {{ user }}!</h1>
    </header>
    <img src="{{ url_for('static', filename='images/flask-logo.png') }}" alt="Flask Logo">
    <footer>
        <p>Powered by Flask</p>
    </footer>
</body>
</html>
```

Practices for Templates and Static Files

File Organization:

- Keep templates in the templates folder.
- Keep CSS, JS, and images in respective subfolders under static.

Dynamic URLs:

- Always use url_for() for linking static files or routes.

Template Inheritance:

- Use a base template for common layouts like headers and footers.

Minification:

- Minify CSS and JavaScript files to optimize performance.

Flask App with Database Connectivity in Python

Flask can be used to connect to databases like **SQLite, MySQL,** or **PostgreSQL** to store, retrieve, update, and delete data. Flask provides support for database operations using libraries like sqlite3, SQLAlchemy (ORM), or connectors like mysql-connector-python for MySQL.

Steps to Connect Flask with a Database

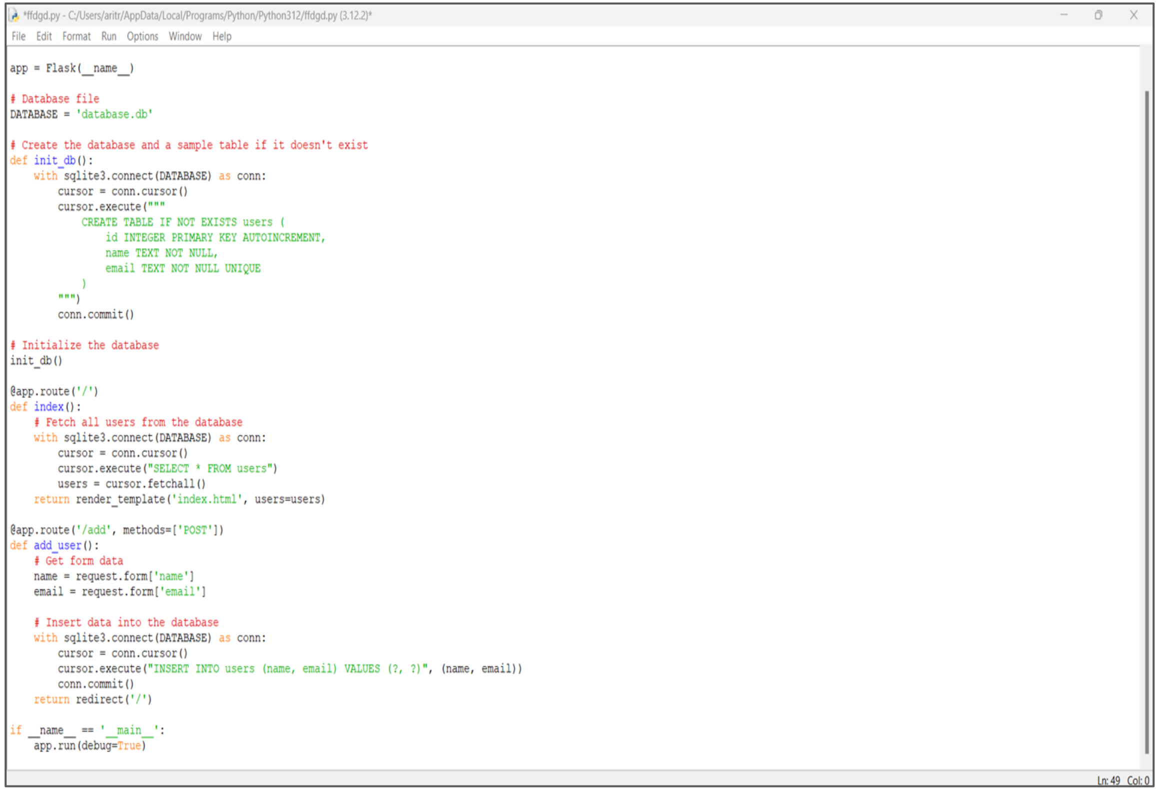
1. **Set Up the Database:**
   - Create a database (e.g., SQLite, MySQL, etc.) and define its schema.
2. **Install Required Libraries**:
   - Use the appropriate database library (sqlite3, mysql-connector-python, or psycopg2 for PostgreSQL).
3. **Configure Database in Flask:**
   - Add database configurations in the Flask app.
4. **Perform CRUD Operations**:
   - Create, read, update, and delete data using SQL queries or an ORM.

Example 1: Flask with SQLite (Basic)

Folder Structure:

```
project/
├── app.py
├── database.db
├── templates/
│   ├── index.html
```

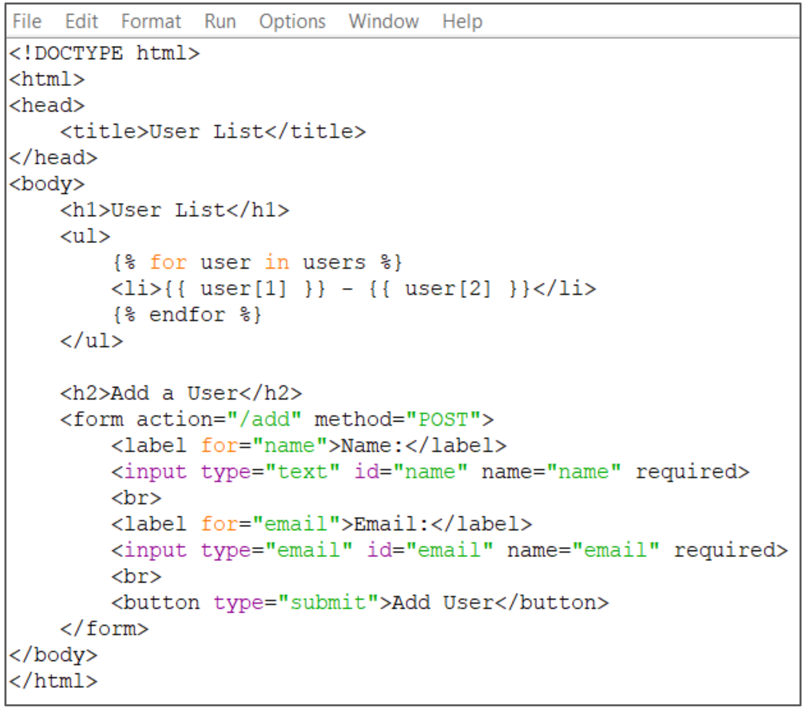
Code Example:`app.py`


ffdgd.py - C:/Users/aritr/AppData/Local/Programs/Python/Python312/ffdgd.py (3.12.2)
File Edit Format Run Options Window Help
Ln: 49 Col: 0

```python
app = Flask(__name__)

# Database file
DATABASE = 'database.db'

# Create the database and a sample table if it doesn't exist
def init_db():
    with sqlite3.connect(DATABASE) as conn:
        cursor = conn.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                email TEXT NOT NULL UNIQUE
            )
        """)
        conn.commit()

# Initialize the database
init_db()

@app.route('/')
def index():
    # Fetch all users from the database
    with sqlite3.connect(DATABASE) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM users")
        users = cursor.fetchall()
    return render_template('index.html', users=users)

@app.route('/add', methods=['POST'])
def add_user():
    # Get form data
    name = request.form['name']
    email = request.form['email']

    # Insert data into the database
    with sqlite3.connect(DATABASE) as conn:
        cursor = conn.cursor()
        cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", (name, email))
        conn.commit()
    return redirect('/')

if __name__ == '__main__':
    app.run(debug=True)
```

Example contd…

index.html:



```
<!DOCTYPE html>
<html>
<head>
    <title>User List</title>
</head>
<body>
    <h1>User List</h1>
    <ul>
        {% for user in users %}
        <li>{{ user[1] }} - {{ user[2] }}</li>
        {% endfor %}
    </ul>

    <h2>Add a User</h2>
    <form action="/add" method="POST">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <br>
        <button type="submit">Add User</button>
    </form>
</body>
</html>
```

Explanation of Example 1

1. **Database Initialization**:
   - The init_db() function ensures the database and the users table are created if they don't already exist.
   - The database file is database.db.
2. **Routes**:
   - **/**:
     - Fetches all users from the database and displays them on the webpage.
     - Executes an SQL SELECT query to retrieve the data.
   - **/add**:
     - Handles the form submission for adding a new user.
     - Inserts the data into the database using an SQL INSERT query.
     - Redirects back to the home page.
3. **Templates**:
   - The index.html template dynamically displays all users fetched from the database using Jinja2 syntax ({% for user in users %}).
4. **Database Operations**:

Advantages of Using Databases in Flask Apps

Data Persistence:

- Stores data that persists beyond the application's runtime.

Dynamic Content:

- Dynamically generates web pages based on database content.

Secure Storage:

- Stores user information securely with encryption.

Handling Exceptions and Errors in Flask

Flask provides mechanisms to gracefully handle errors and exceptions. This is critical for building robust applications and providing user-friendly feedback.

Concepts

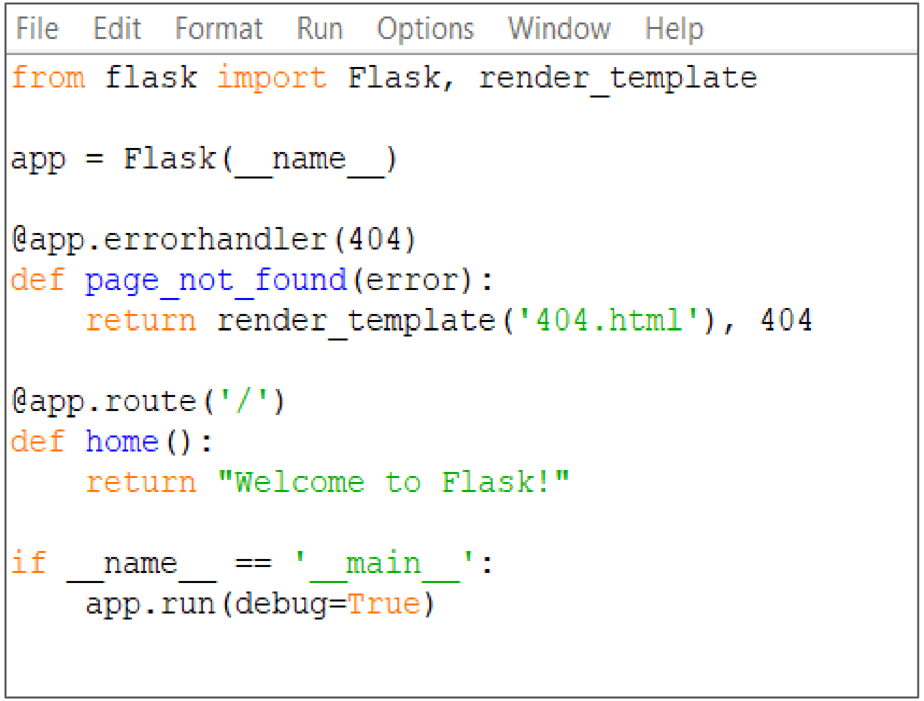
1. **HTTP Error Handlers**: Handle specific HTTP errors (e.g., `404`, `500`).

1. **Custom Exception Handling**: Define and handle your application-specific exceptions.

1. **Logging**: Capture exceptions and log them for debugging.

Example: handling a 404 error.

404 error:



```
from flask import Flask, render_template

app = Flask(__name__)

@app.errorhandler(404)
def page_not_found(error):
    return render_template('404.html'), 404

@app.route('/')
def home():
    return "Welcome to Flask!"

if __name__ == '__main__':
    app.run(debug=True)
```

Explanation:

- @app.errorhandler(404): Captures any 404 Not Found errors.
- render_template('404.html'): displays a custom 404 error page.
- debug=True: Shows detailed error pages during development but should be disabled in production.

Flash Messages in Flask

Flash messages are a way to show temporary notifications to users, such as success, error, or warning messages.

Key Functions

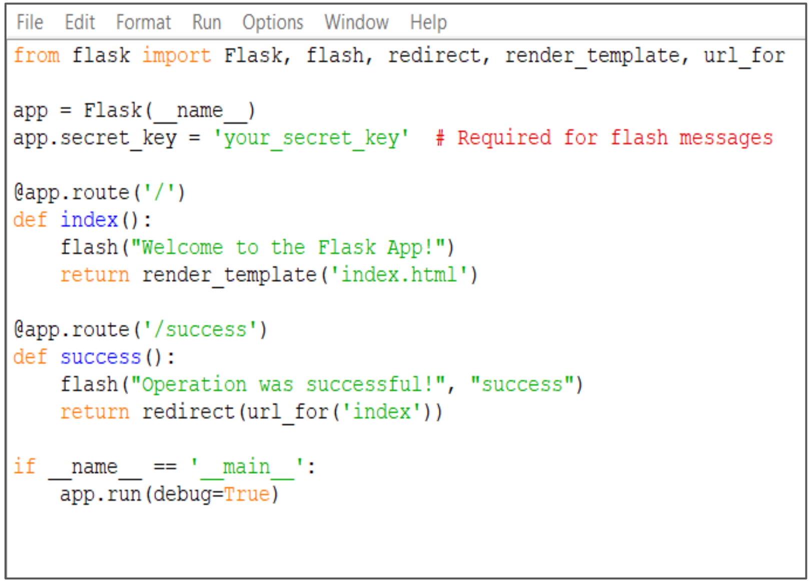
- flask.flash(message): Creates a flash message.
- flask.get_flashed_messages(): Retrieves and displays flash messages.

Steps to Use Flash Messages

1. Enable sessions in Flask by setting a SECRET_KEY.
2. Use flash() to create a message.
3. Display messages in the template using get_flashed_messages().

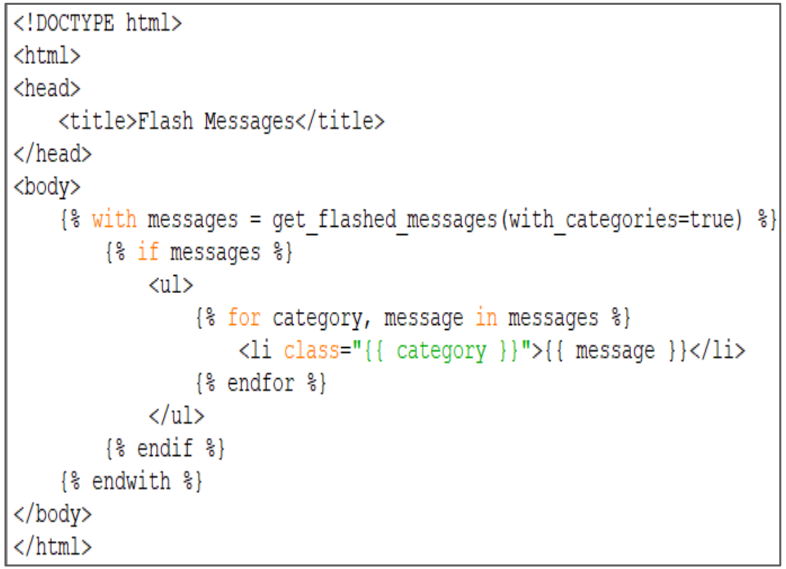
Example: Flash Messages

Flash Message



```python
from flask import Flask, flash, redirect, render_template, url_for

app = Flask(__name__)
app.secret_key = 'your_secret_key'  # Required for flash messages

@app.route('/')
def index():
    flash("Welcome to the Flask App!")
    return render_template('index.html')

@app.route('/success')
def success():
    flash("Operation was successful!", "success")
    return redirect(url_for('index'))

if __name__ == '__main__':
    app.run(debug=True)
```

HTML Template (index.html):



```html
<!DOCTYPE html>
<html>
<head>
    <title>Flash Messages</title>
</head>
<body>
    {% with messages = get_flashed_messages(with_categories=true) %}
        {% if messages %}
            <ul>
                {% for category, message in messages %}
                    <li class="{{ category }}">{{ message }}</li>
                {% endfor %}
            </ul>
        {% endif %}
    {% endwith %}
</body>
</html>
```

Explanation:

- flash(message, category): Creates a flash message with an optional category (e.g., success, error).
- get_flashed_messages(): Retrieves all flash messages and their categories for display in the template.

Working with Mails in Flask

Flask provides support for sending emails using the Flask-Mail extension. This is useful for features like account activation, password reset, or notifications.

Installation

pip install Flask-Mail

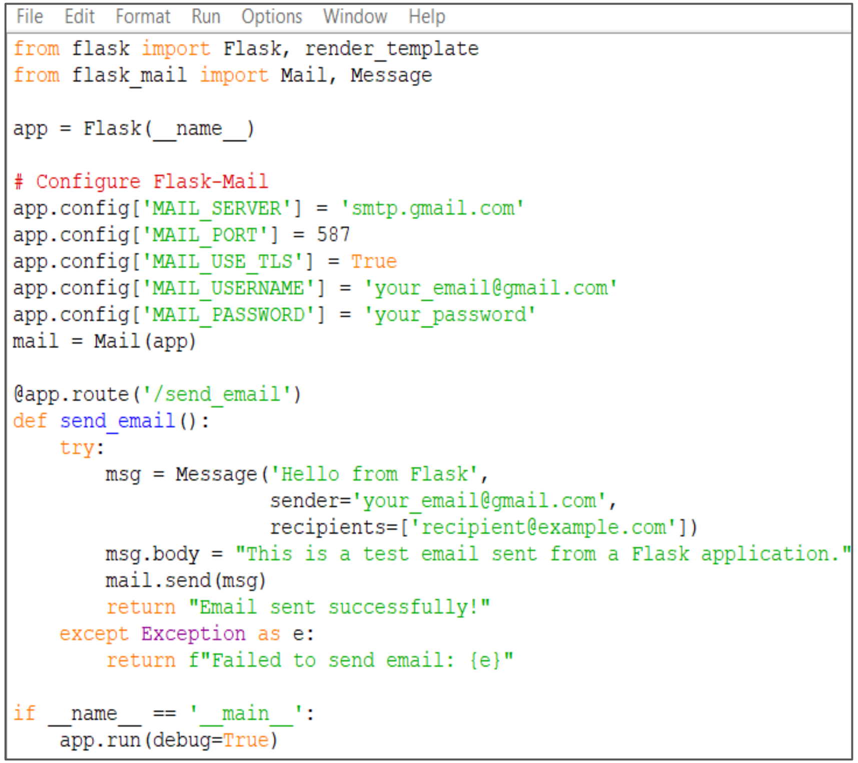
Setting Up Flask-Mail

1. **Configuration**: Add mail server details to your Flask app.

1. **Create a Mail Instance:** Use the Mail class from Flask-Mail.

1. **Send Emails**: Use the send method to send emails.

Example: Sending Emails



```python
from flask import Flask, render_template
from flask_mail import Mail, Message

app = Flask(__name__)

# Configure Flask-Mail
app.config['MAIL_SERVER'] = 'smtp.gmail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = 'your_email@gmail.com'
app.config['MAIL_PASSWORD'] = 'your_password'
mail = Mail(app)

@app.route('/send_email')
def send_email():
    try:
        msg = Message('Hello from Flask',
                      sender='your_email@gmail.com',
                      recipients=['recipient@example.com'])
        msg.body = "This is a test email sent from a Flask application."
        mail.send(msg)
        return "Email sent successfully!"
    except Exception as e:
        return f"Failed to send email: {e}"

if __name__ == '__main__':
    app.run(debug=True)
```

Explanation:

1. **Mail Configuration**:
   - MAIL_SERVER: The SMTP server (e.g., smtp.gmail.com for Gmail).
   - MAIL_PORT: The port for the SMTP server (587 for TLS).
   - MAIL_USE_TLS: Enables Transport Layer Security.
   - MAIL_USERNAME and MAIL_PASSWORD: Your email credentials.
2. **Message Object**:
   - sender: Email address sending the email.
   - recipients: List of recipient email addresses.
   - body: The email content.

Contd…

Summary :-

Feature	Description	Example
Handling Exceptions	Use @app.errorhandler for HTTP errors and try...except for application errors.	Handle 404 or log errors using logging.
Flash Messages	Use flash() for temporary notifications; display using get_flashed_messages().	Notify users about successful actions or errors.
Sending Emails	Configure Flask-Mail to send emails via SMTP. Supports plain text and HTML emails.	Send account activation links, password resets, or notifications.

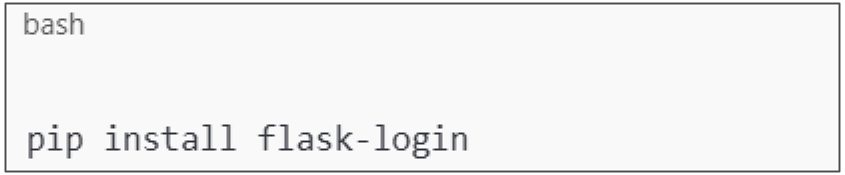
Authenticating and Authorizing Users with Flask-Login

Authentication (verifying the identity of a user) and authorization (granting access based on roles) are fundamental for secure web applications. Flask provides the Flask-Login extension to manage these functionalities.

Features of Flask-Login

- **User Session Management:** Handles login/logout and session persistence.
- **User Loader:** Keeps track of the currently logged-in user.
- **Login Required Decorator**: Protects routes from unauthorized access.

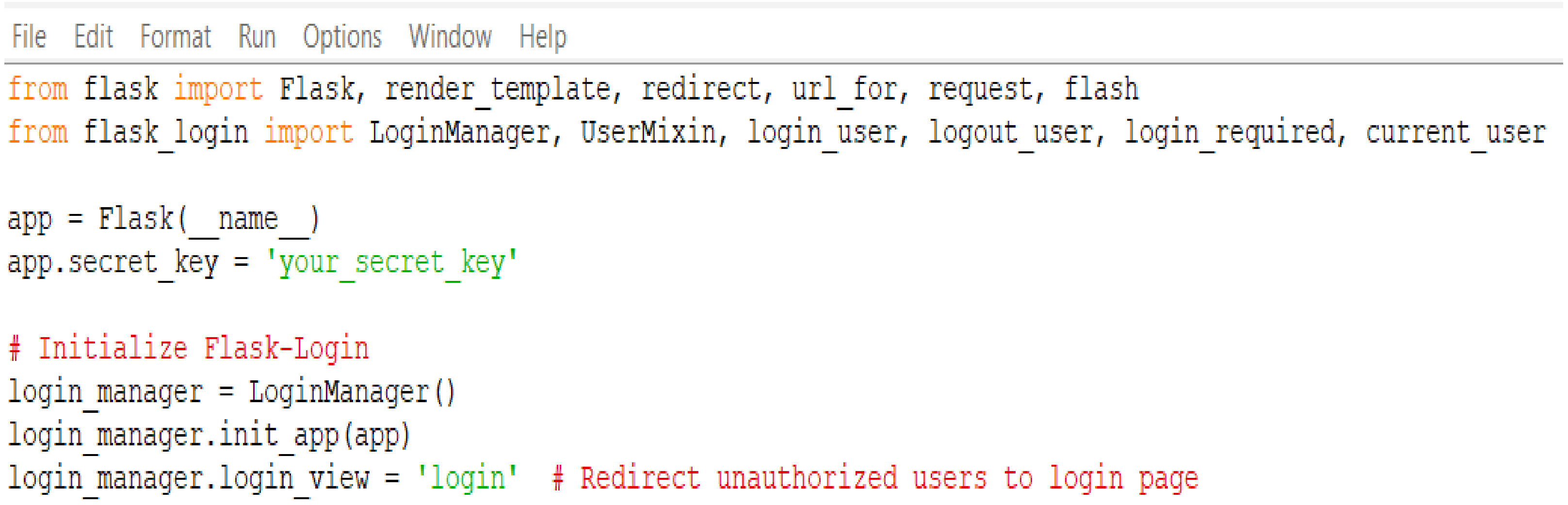
Installation:



```bash
pip install flask-login
```

Example: Authenticating Users

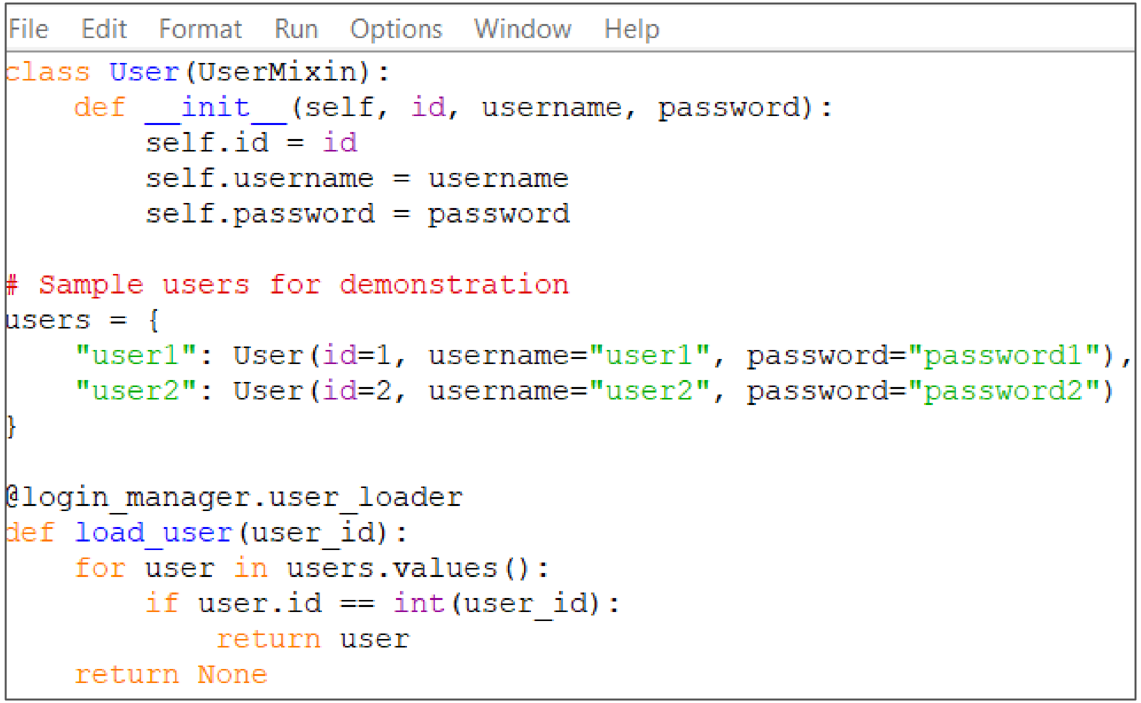
Step 1: Set up the Flask Application:



```python
from flask import Flask, render_template, redirect, url_for, request, flash
from flask_login import LoginManager, UserMixin, login_user, logout_user, login_required, current_user

app = Flask(__name__)
app.secret_key = 'your_secret_key'

# Initialize Flask-Login
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = 'login'  # Redirect unauthorized users to login page
```

Example Contd…

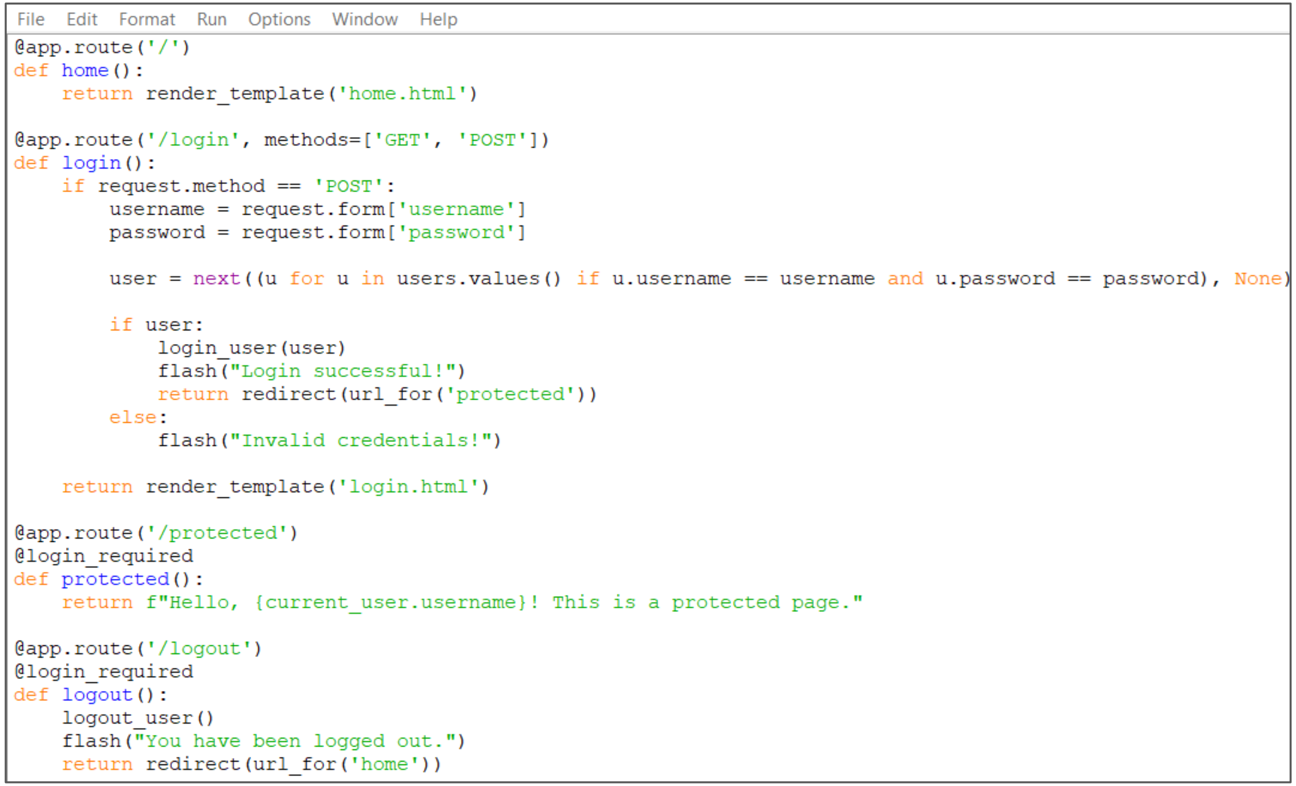
Step 2: Create the User Model



```
class User(UserMixin):
    def __init__(self, id, username, password):
        self.id = id
        self.username = username
        self.password = password

# Sample users for demonstration
users = {
    "user1": User(id=1, username="user1", password="password1"),
    "user2": User(id=2, username="user2", password="password2")
}

@login_manager.user_loader
def load_user(user_id):
    for user in users.values():
        if user.id == int(user_id):
            return user
    return None
```

Contd..

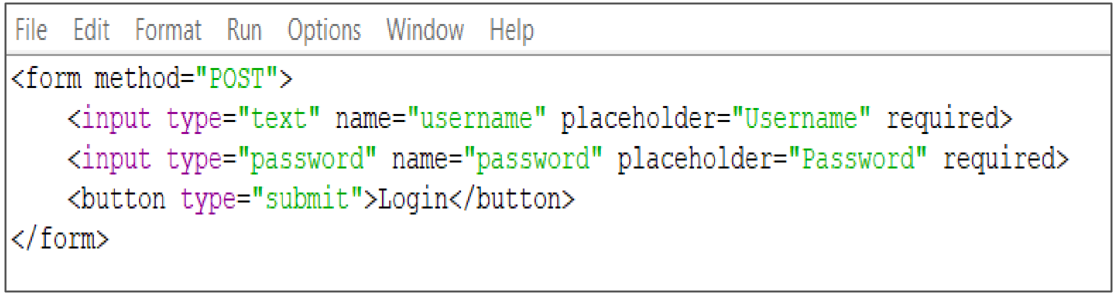
Step 3: Define Routes:



```
@app.route('/')
def home():
    return render_template('home.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        user = next((u for u in users.values() if u.username == username and u.password == password), None)

        if user:
            login_user(user)
            flash("Login successful!")
            return redirect(url_for('protected'))
        else:
            flash("Invalid credentials!")

    return render_template('login.html')

@app.route('/protected')
@login_required
def protected():
    return f"Hello, {current_user.username}! This is a protected page."

@app.route('/logout')
@login_required
def logout():
    logout_user()
    flash("You have been logged out.")
    return redirect(url_for('home'))
```

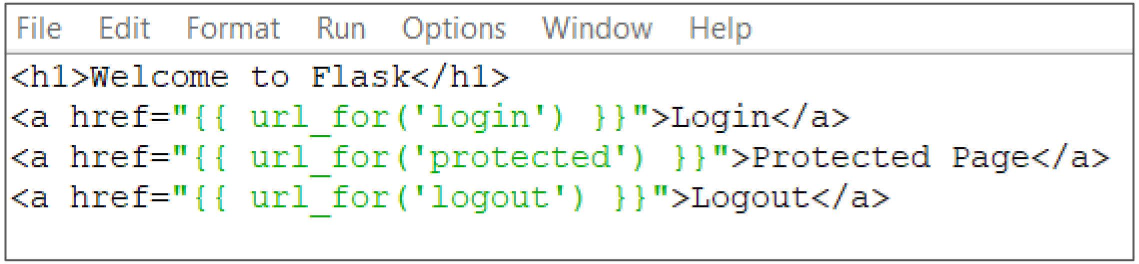
Step 4: Templates

Login.html:



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

home.html:



```
<h1>Welcome to Flask</h1>
<a href="{{ url_for('login') }}">Login</a>
<a href="{{ url_for('protected') }}">Protected Page</a>
<a href="{{ url_for('logout') }}">Logout</a>
```

Deploying a Flask Application to a Web Server

Deployment makes your application accessible to users over the internet. Common steps involve using production-grade servers and services.

Steps for Deployment

1. Use a Production WSGI Server

- Flask's built-in development server (app.run()) is not suitable for production.
- Use **Gunicorn** (Linux) or **Waitress** (Windows).

Example (Linux with Gunicorn):

```
pip install gunicorn
```

```
gunicorn -w 4 -b 0.0.0.0:8000 app:app
```

- -w 4: Number of worker processes.
- -b: Bind to a specific address and port.

Contd…

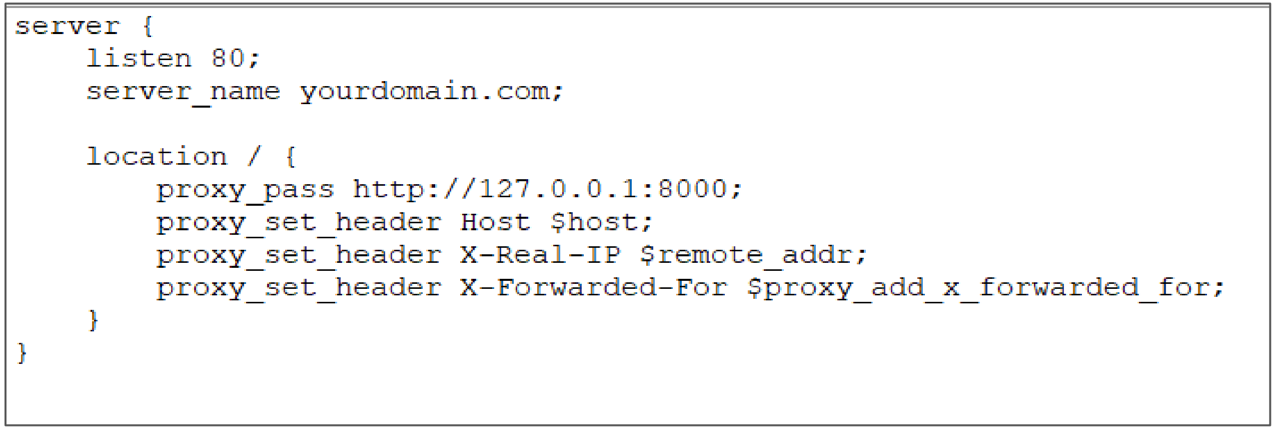
2. Use a Reverse Proxy:

Configure **Nginx** or **Apache** as a reverse proxy to forward requests to the Flask application.

Example (Nginx):

1. **Install Nginx:** sudo apt install nginx

2. **Configure a server block:**

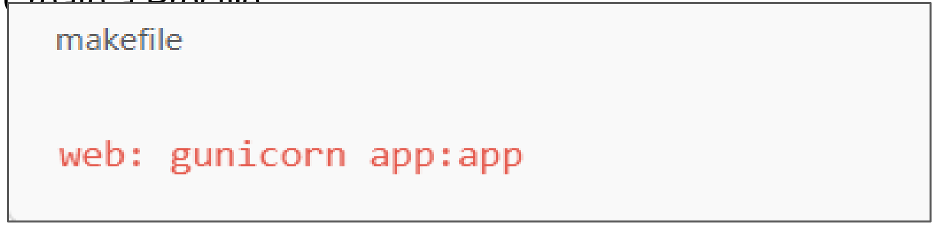


```
server {
    listen 80;
    server_name yourdomain.com;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Contd…

3. Deployment Platforms

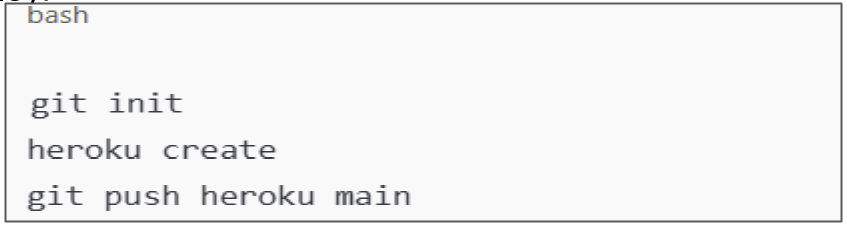
- **Heroku**:
    1. Install Heroku CLI.
    2. Create a Procfile:



makefile

```makefile
web: gunicorn app:app
```

    1. Deploy:



bash

```bash
git init
heroku create
git push heroku main
```

Contd…

- **AWS Elastic Beanstalk:** Use Flask with a pre-configured Python environment.
- **Docker:** Containerize your application and deploy to cloud platforms.

4. Serve Static Files

- Ensure that static files (CSS, JavaScript) are properly served by the web server.
- Place static assets in a /static folder in your Flask project.

Feature	Description
Flask-Login	Simplifies authentication, session management, and protecting routes.
Deployment	It involves using production-grade WSGI servers and reverse proxies.
Platforms	Heroku, AWS, Docker, and other cloud services provide easy deployment.