Flask Framework

Unit - 4

Objectives

- Understand the basics of Flask and web development with Python
- Set up Flask in a virtual environment and configure app settings
- Create routes, URLs, and handle HTTP requests
- Work with templates, static, and media files
- Process form data and render it dynamically
- Connect Flask apps to SQLite3 / MySQL databases
- Handle errors, exceptions, and show flash messages
- Send emails using Flask-Mail
- Implement user authentication & authorization with Flask-Login
- Deploy Flask applications to a web server

Introduction to Flask

What is Web Development?

- Web development involves creating and maintaining websites and web applications.
- It includes both frontend (UI) and backend (server logic, databases).
- Python is widely used for backend development because of its simplicity and readability.

Why Use Flask?

- Simple and minimal setup required.
- Great for beginners and small-to-medium applications.
- Supports RESTful APIs easily.
- Highly extensible with plugins and libraries.

Advantages of Flask & Conclusion

- Lightweight, modular, and easy to learn.
- Excellent for rapid prototyping.
- Supports extensions like authentication, forms, and ORM.
- Flask is a great entry point for learning web development with Python.
- Start small and scale as your project grows.

Setting Up Flask in a Virtual Environment

What is a Virtual Environment?

- A virtual environment (venv) is an isolated Python workspace.
- It allows you to manage dependencies for each project separately.
- Prevents version conflicts between different projects.

Why Use a Virtual Environment?

- Keeps each project’s dependencies separate.
- Avoids compatibility issues between packages.
- Makes project sharing and deployment easier.
- Analogy: Like having separate toolboxes for different projects.

Creating a Virtual Environment

- Use Python’s built-in venv module.
- Commands:
  - python -m venv venv
- This creates a folder named ‘venv’ that holds the environment files.

Activating the Virtual Environment

- Activate to start using it.
- Commands:
  - venv\Scripts\activate   # Windows
  - source venv/bin/activate   # Mac/Linux
- You’ll see (venv) appear before your terminal prompt.

Installing Flask in Virtual Environment

- Use Python’s package manager (pip) to install Flask.
- Commands:
  - pip install Flask
- Flask can be used in a virtual environment for isolated dependencies.

Verifying Installation

- You can confirm Flask installation by running:
  - pip show Flask
- Or open Python shell and import Flask:
  - python
  - >>> import flask
  - >>> flask.__version__

Deactivating the Virtual Environment

- When finished, deactivate using:
  - deactivate
- This returns you to the system-wide Python environment.
- Virtual environments make development clean, organized, and reproducible.

Creating a Flask Application

Introduction

- Flask is a lightweight Python framework used to build web applications.
- You can start a Flask app with just a few lines of code.
- Routing connects URLs (like '/' or '/home') to Python functions that handle user requests.

Your First Flask App

```
from flask import Flask
app = Flask(__name__)

@app.route('/’)
def home():
        return 'Hello, Flask!’

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

Explanation of Key Components

- from flask import Flask → Imports the Flask class.
- app = Flask(__name__) → Creates the Flask application instance.
- @app.route('/') → Defines a route (URL) for your app.
- def hello(): → Function that returns what is shown on the page.
- app.run(debug=True) → Starts the Flask development server.

Running the Application

- Save your file as app.py.
- In terminal, set the environment variable and run:
  - set FLASK_APP=app.py # Windows
  - export FLASK_APP=app.py # Mac/Linux
  - flask run
- Visit http://127.0.0.1:5000 in your browser to see your app!

Routing in Flask

Understanding Routing

What is Routing?

- Routing is the process of mapping URLs (web addresses) to functions in your Flask application.
- Each URL (like /home, /about, /login) is connected to a specific view function that runs when a user visits that address.
- Each route represents a different page or endpoint of your app.

How Routing Works in Flask

- Routing maps URLs to specific functions called 'view functions'.
- Flask uses Python decorators (@app.route()) to define routes.
- When a request comes to the server, Flask checks which route matches the URL and executes the associated function.

Understanding Routing

Importance of Routing

- Defines the structure and navigation of your web app.
- Connects user actions (clicks, form submissions) to server-side logic.
- Makes the app modular and easier to maintain.

Example:

```
@app.route('/about’)
def about():
return 'This is the About Page'
```

Using Multiple Routes

- You can define multiple routes to handle different pages.
- Example:

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


        @app.route('/contact')
def contact():
        return 'Contact Us at
```

Explanation:

- / → URL for homepage
- /about → URL for the about page
- Each route is linked to a specific function that returns content (like HTML or text)

Dynamic Routing

- Flask routes can also accept dynamic parameters, which makes pages more flexible.
- Flask allows dynamic URLs using placeholders.

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

- The value in the URL is passed to the function as an argument.
  - Visiting /user/Alice → “Hello, Alice!”
  - Visiting /user/Bob → “Hello, Bob!”

Redirecting

- You can use redirect() to send users to a different route.
- Flask automatically shows a 404 page when no route matches.
- Example:

```
from flask import redirect

@app.route('/old-page’)


def old_page():

    return redirect(url_for('home'))
```

Application Settings

What Are Application Settings?

- Application settings (or configurations) define how your Flask app behaves.
- They control things like:
  - Debug mode
  - Database connection
  - Secret keys
  - Email server settings
  - File upload limits, etc.
- Configuration defines app behavior, database connection, and secret keys.

Why Use Application Settings

- To keep your app organized and secure.
- To easily switch between development, testing, and production environments.
- To separate configuration from your main code, so you don’t hardcode sensitive values (like passwords or API keys).

Configuration Methods

- Flask allows developers to define app settings in **three main ways**, depending on the **size, complexity, and security** needs of the project.
- Configuration Methods:
  - Direct
  - Config File
  - Environment Variables

Configuration Methods – Direct Assignment

- This is the simplest and most common approach when starting with Flask.
- You define your configuration directly in your main application file (app.py).
- You use app.config as a dictionary to store key–value pairs.
- Quick to write and easy for small apps or class assignments.
- Best for beginners, small demos, or short-term projects.

Configuration Methods – Direct Assignment

Example Code:

app.py

```
app = Flask(__name__)
app.config['DEBUG'] = True
app.config['SECRET_KEY'] = 'mysecret'
```

Configuration Methods – Config file

- For cleaner and more scalable apps, Flask encourages using a separate **configuration file**.
- You define a Config class that holds all settings.
- The Flask app loads it using from_object().
- Secure values (like passwords) can come from environment variables.
- Best for medium–large projects, or any app with databases, mail, or login features.

Limitations:

- Slightly more setup for small apps.
- Requires separate file management.

Configuration Methods – Config file

config.py

```
import os

class Config:
    DEBUG = True
    SECRET_KEY =
os.getenv('SECRET_KEY', 'devkey')
    SQLALCHEMY_DATABASE_URI =
'sqlite:///site.db'
```

app.py

```
from flask import Flask
from config import Config

app = Flask(__name__)
app.config.from_object(Config)
```

Configuration Methods – Environment Variables

- Environment variables store configuration outside your code — ideal for security and deployment.
- Best for production environments, servers, or cloud platforms (Heroku, Render, etc.)

Limitations:

- Harder for beginners to set up locally.
- Requires knowing how to manage system environments.

Configuration Methods – Environment Variables

In Terminal

```
export SECRET_KEY='supersecret'

export FLASK_ENV='production'
```

app.py

```
import os

from flask import Flask


app = Flask(__name__)

app.config['SECRET_KEY'] =

os.environ.get('SECRET_KEY')

app.config['DEBUG'] =

os.environ.get('FLASK_ENV') == 'development'
```

Comparison of Configuration Methods

Method	Where Configuration Lives	Security Level	Ease of Use	Use Case	Example
Direct	Inside main Flask file	Low (hardcoded values)	Easy	Small demos, learning	app.config['DEBUG']=True
Separate config.py	In dedicated file	Medium (can pull from env)	Moderate	Medium to large projects	app.config.from_object(Config)
Environment Variables	In OS or deployment platform	High (values hidden from code)	Requires setup	Production / deployment	os.getenv('SECRET_KEY')

URL Building

What is URL Building?

- URL building means generating URLs dynamically inside your Flask app using the url_for() function.
- Instead of hardcoding URLs, Flask builds them automatically based on the view function name.

- Example:

```
from flask import url_for

@app.route('/about')
def about():
   return "About Page"

@app.route('/')
def index():
   about_url = url_for('about')
   return f"<a href='{about_url}'>Go to About</a>“
```

Why Use url_for()

- Makes your code flexible and maintainable.
- If you rename a route or change its path, you don’t need to update URLs everywhere.
- Prevents broken links when restructuring the app.
- url_for() builds URLs dynamically using view function names.
- Prevents hardcoding paths → safer and easier maintenance.
- Supports dynamic routes and query parameters.
- Works seamlessly in both Python code and HTML templates.

HTTP Methods

What Are HTTP Methods?

- HTTP (HyperText Transfer Protocol) is the foundation of communication on the web.
- When a client (like a browser) sends a request to a server (like a Flask app), it uses an HTTP method to tell the server what kind of action it wants to perform.
- HTTP Methods describe **“what you want the server to do”**.
- **Example:**
  - When you open a webpage → **GET request**
  - When you submit a form → **POST request**

HTTP Methods in Flask

- Flask routes can handle multiple HTTP methods through the methods parameter in the @app.route() decorator.
- Common HTTP Methods supported in Flask:
  - **GET** – Retrieve Data
  - **POST** – Send Data to Server
  - **PUT** – Update Existing Data
  - **DELETE** – Remove Data
  - **PATCH** – Modify Part of Data

HTTP Methods - GET

- Used to request data from the server.
- Default method for most routes.
- Parameters are sent in the URL (as query strings).
- Example:

```
@app.route('/search’)
def search():
    query = request.args.get('q')
    return f"You searched for: {query}"
```

HTTP Methods – POST

- Used to submit data, usually from forms.
- Data is sent in the request body, not visible in the URL.
- Commonly used for login forms, registrations, and uploads.

- Example:

```
@app.route('/submit', methods=['POST'])
def submit():
    name = request.form['name']
    return f"Hello, {name}! Form submitted successfully."
```

HTTP Methods – PUT

- Used to update or replace existing information on the server.
- Not as commonly used in basic Flask apps, but important in APIs.

- Example:

```
@app.route('/update/<int:id>', methods=['PUT'])
def update_user(id):
    return f"User with ID {id} has been updated!"
```

HTTP Methods – DELETE

- Used to delete data from the server.

- Example:

```
@app.route('/delete/<int:id>', methods=['DELETE'])
def delete_user(id):
    return f"User with ID {id} deleted!"
```

Combining GET and POST in One Route

- You can allow multiple HTTP methods for a single route
- When the user opens the page → GET request
- When the user submits the form → POST request

- Example:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        return f"Welcome, {username}!"
    return render_template('login.html')
```

Templates and Jinja2

What is a Template in Flask?

- A template in Flask is an HTML file that can display dynamic data sent from your Python code.
- Instead of writing static HTML pages, Flask uses templates to combine HTML with Python expressions.
- In short templates allows Flask to generate dynamic web pages.

Why Templates Are Needed

- Static HTML pages can’t display user data (like names, scores, or search results).
- Templates make web pages interactive, personalized, and data-driven.
- Example:
  - Instead of showing the same message to everyone, you can show “Welcome, Alice” or “Welcome, Bob” dynamically using templates.

Jinja2 Template Engine

- A template engine is a tool that helps you create dynamic HTML pages by mixing static HTML with dynamic data coming from your Python code (or any backend language).
- A template engine allows you to write HTML pages that change based on data.
- Flask uses Jinja2 as its built-in template engine.
- Jinja2 is lightweight, fast, and secure.
- Template files are stored in a folder named templates/ by default.

Jinja2 Template Engine

- Jinja2 lets you write Python-like code inside HTML — safely and cleanly.
- You can use:
  - Variables → to display dynamic data
  - Loops → to display lists or tables
  - Conditions → to show or hide parts of a page

How Templates Work in Flask

1. You create HTML files in a special folder named templates/.
2. Flask looks in this folder automatically when rendering templates.
3. You use the render_template() function to send data from your Flask app to the HTML file.

Template Example

- **Directory Sturucture:**

```
my_flask_app/
 ├── app.py
 └── templates/
     └── home.html
```

- **app.py**

```
from flask import Flask,
render_template
app = Flask(__name__)


@app.route('/')
def home():
    name = "Alice"
    return
render_template('home.ht
ml', user_name=name)
```

- **home.html**

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Welcome, {{
user_name }}!</h1>
  </body>
</html>
```

Jinja2 Syntax

Type	Syntax	Description
Variable	{{ variable_name }}	Displays dynamic data
Condition (if)	{% if condition %} ... {% endif %}	Runs conditional logic
Loop (for)	{% for item in list %} ... {% endfor %}	Iterates over data
Comments	{# comment #}	Adds comments (ignored by Jinja2)
Include	{% include 'header.html' %}	Reuses parts of templates

Template Inheritance

- Jinja2 allows you to create a base layout that can be reused across multiple pages (e.g., header, footer, navigation bar).
- This helps avoid repetition and makes updates easier.
- Only the middle section (content) changes for each page.

Template Inheritance – Example

- **base.html**

```
<!DOCTYPE html>
<html>
  <head><title>{% block title %}{% endblock %}</title></head>
  <body>
    <header><h1>My Website</h1></header>
    {% block content %}{% endblock %}
    <footer>© 2025 My Flask App</footer>
  </body>
</html>
```

- **home.html**

```
{% extends "base.html" %}
{% block title %}Home{% endblock %}
{% block content %}
  <p>Welcome to the home page!</p>
{% endblock %}
```

Benefits of Using Templates

- Clean separation of HTML (frontend) and Python (backend) code.
- Easier maintenance and debugging.
- Enables dynamic and interactive websites.
- Reduces repetition through inheritance and includes.

Static Files and Media

What are Static Files and Media Files

- In a web application, not everything is generated dynamically.
- Some files — like images, CSS, JavaScript, or uploaded files — are stored and served as they are.
- These are known as static and media files.

Static Files – Introduction

- Static files are resources that don’t change dynamically.
- They are part of your app’s design and layout — the same for every user.
- They are provided by the developers
- It make your website look good and behave well in the browser.

- **Examples:**
  - CSS files → to style web pages
  - JavaScript files → to add interactivity
  - Images (logos, icons, backgrounds)
  - Fonts

Static Files – Directory Structure

```
my_flask_app/
 ├── app.py
 ├── templates/
 │   └── index.html
 └── static/
     ├── css/
     │   └── style.css
     ├── js/
     │   └── script.js
     └── images/
         └── logo.png
```

How to Use Static Files in Templates

- Flask provides the url_for() function to correctly link static files in HTML templates.
- It automatically builds the correct URL to your static files.
- Prevents hardcoding paths like **/static/...** which might break in deployment.

How to Use Static Files in Templates – Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Flask Site</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
  </head>
  <body>
    <img src="{{ url_for('static', filename='images/logo.png') }}" alt="Logo">
    <script src="{{ url_for('static', filename='js/script.js') }}"></script>
  </body>
</html>
```

Media Files

- Media files are user-uploaded files — they are not part of your project code, but generated while the app runs. Flask provides the url_for() function to correctly link static files in HTML templates.
- It automatically builds the correct URL to your static files.
- Prevents hardcoding paths like **/static/...** which might break in deployment.
- **Examples:**
  - Profile pictures uploaded by users
  - Uploaded documents (PDFs, Word files)
  - Photos, videos, etc.

Media Files – Example

```
from flask import Flask, request,
render_template
import os

app = Flask(__name__)
UPLOAD_FOLDER = 'uploads'
app.config['UPLOAD_FOLDER'] =
UPLOAD_FOLDER
```

```
@app.route('/upload', methods=['GET',
'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['file']

file.save(os.path.join(app.config['UPLOAD
_FOLDER'], file.filename))
        return "File uploaded successfully!"
    return render_template('upload.html')
```

Media Files – Example

- **Upload.html**

```
from flask import Flask, render_template
app = Flask(__name__)


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

- The uploaded files are saved in a folder like:

```
my_flask_app/
└── uploads/
    └── user_uploaded_image.png
```

Static VS Media Files

Static

- Provided by the developer
- Does not change over
- Example:
  - CSS
  - JavaScript
  - Static Images

Media

- Provided by the User
- Can change over time
- Example:
  - Uploaded photos, files

Working with Mails

Introduction to Sending Emails

- Emails act as a direct line to the user’s inbox.
- They help your app interact outside the website — for example, sending notifications, confirmations, and updates.
- It’s a key part of user engagement and retention.
- Applications that send well-formatted, timely emails appear more legitimate and professional.
- Adds credibility to your app (users expect confirmation or reset mails).
- Prevents confusion about successful sign-ups or transactions.

Why Work with Mails in Flask?

- Build User Trust & Communication
- Secure User Accounts
- Automate Notifications
- Easy Integration with Flask

Working with Mails in Flask

- Flask itself doesn’t send emails — it uses Flask-Mail, a helper library.
- You can send:
  - Account activation links
  - Password reset emails
  - Welcome messages
  - Automated alerts

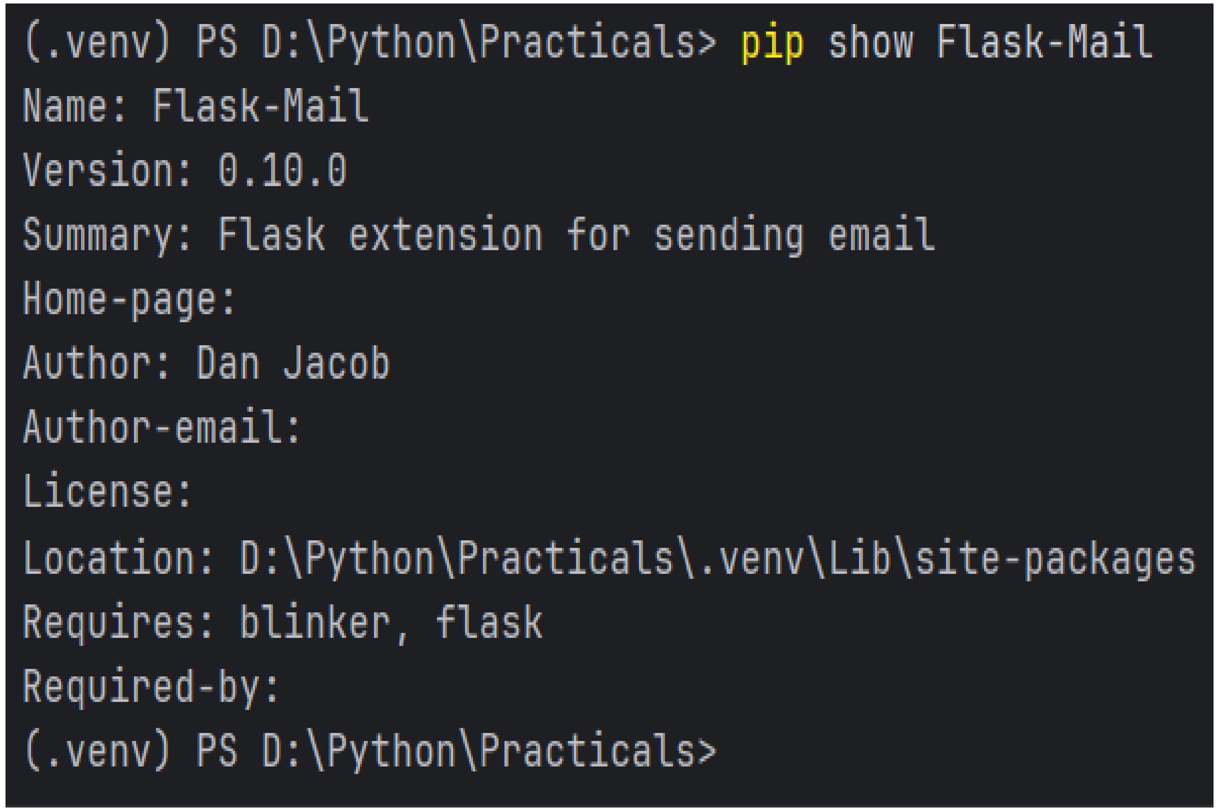
Installing Flask-Mail

- **Installation:**

pip install Flask-Mail

- **Checking installation**

pip show Flask-Mail



```
(.venv) PS D:\Python\Practicals> pip show Flask-Mail
Name: Flask-Mail
Version: 0.10.0
Summary: Flask extension for sending email
Home-page:
Author: Dan Jacob
Author-email:
License:
Location: D:\Python\Practicals\.venv\Lib\site-packages
Requires: blinker, flask
Required-by:
(.venv) PS D:\Python\Practicals>
```

Configuring Flask-Mail

Directly Using app.py

```
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_app_password’
```

Setting	Description
MAIL_SERVER	Your mail provider’s server address
MAIL_PORT	Usually 587 (TLS) or 465 (SSL)
MAIL_USE_TLS	Enables encryption
MAIL_USERNAME	Sender’s email address
MAIL_PASSWORD	App password for authentication

Directly Using app.py

Pros:

- Easy to see and edit while learning.
- Good for small projects or prototypes.
- Simple for beginners
- Quick to test functionality

Cons:

- Storing passwords directly in the code is unsafe.
- Harder to manage when deploying to production.
- Mixing configuration with app logic → messy
- Not secure for production (hardcoded secrets)

Using Environment Variables

Step 1: Set environment variables

```
export MAIL_USERNAME='your_email@gmail.com’
export MAIL_PASSWORD='your_app_password'
```

Step 2: Access in flask

```
import os
app.config['MAIL_USERNAME'] = os.getenv('MAIL_USERNAME’)
app.config['MAIL_PASSWORD'] = os.getenv('MAIL_PASSWORD’)
```

Using Environment Variables

Pros:

- Keeps sensitive data out of source code.
- Easy to change credentials without touching the code.
- Required for most production deployments (Heroku, Render, AWS).

Cons:

- Slightly more setup for beginners.

Using .env File

.env File

```
MAIL_USERNAME=your_email@host.com

MAIL_PASSWORD=your_app_password
```

Load with Python

```
from dotenv import load_dotenv

load_dotenv()

app.config['MAIL_USERNAME'] = os.getenv('MAIL_USERNAME’)
```

Comparison of Configuration Approaches

Approach	Pros	Cons	Use Case
Direct in app.py	Easy & quick	Not secure, messy	Learning / small projects
.env File	Structured, secure	Slightly more setup	Real apps / production
Env variables only	Most secure, scalable	Needs setup	Cloud deployment

Sending Mail

- Flask route triggers email function.
- Message created with subject, sender, and recipient.
- Flask-Mail sends via SMTP.

- Example:

```
@app.route("/send_email")
def send_email():
    msg = Message(
        subject="Hello from Flask!",
        sender=app.config['MAIL_USERNAME'],
        recipients=["student@example.com"]
    )
    msg.body = "This is a test email sent from a Flask app."
    mail.send(msg)
    return "Email sent successfully!"
```

Common Email Sending Issues

Problem	Cause	Solution
Auth error	Gmail blocking sign-in	Use App Password
Timeout	Wrong port or encryption	Check MAIL_PORT and TLS
Email not sent	Missing credentials	Verify config
Invalid address	Typo in recipient	Correct it
Bulk limit	Gmail restriction	Use SendGrid or Mailgun

Authenticating and Authorizing Users with Flask-Login

What is Authentication & Authorization?

Authentication

- Who are you?
- It’s the process of verifying a user’s identity.
- The app checks whether you really are who you claim to be.
- Usually done using:
  - Username & password
  - OTP (One-Time Password)
  - Email verification
  - OAuth (Google, GitHub login)

Authorization

- What are you allowed to do?
- Happens after authentication.
- It checks what actions or resources a logged-in user can access.
- Defines permissions and access control:
  - Students can view results but not edit them.
  - Admins can manage users.
  - Guests can only view public pages.

Flask-Login

What is Flask-Login?

- Flask-Login is a Flask extension (add-on library) that helps manage:
- User logins
- Sessions (remembering logged-in users)
- Logout handling
- Route protection (limiting access to logged-in users)

In simple terms

- It's like a "**security manager**" that tracks who is using your website and what pages they can access.

Why Use Flask-Login Instead of Manual Handling?

- Without Flask-Login, you’d have to:
- Write your own session management code.
- Store and verify login cookies manually.
- Handle what happens when users close their browser or reopen the site.
- Build your own decorators (like @login_required) to protect routes.
- That’s a lot of repetitive, error-prone code.
- Flask-Login simplifies all of this with a few easy functions.

Flask-Login Features

Functionality	Description
Session Management	Keeps track of who is logged in using secure cookies.
Login & Logout	Simplifies logging users in and out of the session.
Remember Me	Lets users stay logged in between visits.
Route Protection	@login_required decorator prevents unauthorized access.
Current User Info	Provides the current_user object anywhere in your app.

Why Use Flask-Login Instead of Manual Handling?

- Without Flask-Login, you'd have to:
- Write your own session management code.
- Store and verify login cookies manually.
- Handle what happens when users close their browser or reopen the site.
- Build your own decorators (like @login_required) to protect routes.
- That's a lot of repetitive, error-prone code.
- Flask-Login simplifies all of this with a few easy functions.

Installing Flask-Login

Installing:

- pip install flask-login

Import it into your app:

- from flask_login import LoginManager

Setting up Flask-Login

```
from flask_login import LoginManager

login_manager = LoginManager()

login_manager.init_app(app)

login_manager.login_view = "login“
```

Explanation:

- LoginManager() → Creates a login handler.
- init_app(app) → Connects it to Flask.
- login_view → Redirects unauthorized users to /login.

Creating a User Model

```
from flask_login import UserMixinclass
User(UserMixin):

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

Explanation:

- UserMixin gives built-in properties like:
  - is_authenticated
  - is_active
  - is_anonymous
  - get_id()
  - The class represents a user record (you can later connect this to a database).

User Loader Function

```
@login_manager.user_loader


def load_user(user_id):
    return User.get(user_id)
```

Explanation:

- Flask-Login calls this function automatically whenever it needs to load the logged-in user.
- user_id comes from the session cookie.
- You typically fetch the user from your database.

Login and Logout

Login

```
@app.route("/login", methods=["POST"])

def login():
    username = request.form["username"]
    password = request.form["password"]
    user = User.get(username)

    if user and user.password == password:
        login_user(user)
        flash("Login successful!", "success")
        return redirect(url_for("dashboard"))
    else:
        flash("Invalid credentials", "error")
        return redirect(url_for("login"))
```

Logout

```
@app.route("/logout")
@login_required

def logout():
    logout_user()
    flash("You have been logged out.", "info")
    return redirect(url_for("login"))
```

SIX PHRASE
Democratizing Skilling
a Veranda Enterprise

Deployment

Deploying a Flask App

- Your app currently runs on your local machine
- Only you can access it there.
- To share it with others or make it available publicly, you need to host it on a server.
- Deployment ensures your Flask app runs 24/7, even when your PC is off.
- Deployment means taking your Flask app (which you built and tested on your computer) and putting it on a web server so that other people can access it over the Internet.
- Deployment = moving your app from localhost → live server (public URL).

Development vs Production

Feature	Development	Production
Environment	Local (your system)	Remote server (Internet)
Purpose	Testing & debugging	Public use
Server	Flask's built-in test server	Production server (like Gunicorn or uWSGI)
Debug Mode	ON	OFF
Security	Low	High (with HTTPS, firewalls, etc.)

Problem:

Flask’s built-in development server (app.run()) is not designed for production — it’s single-threaded and not secure.

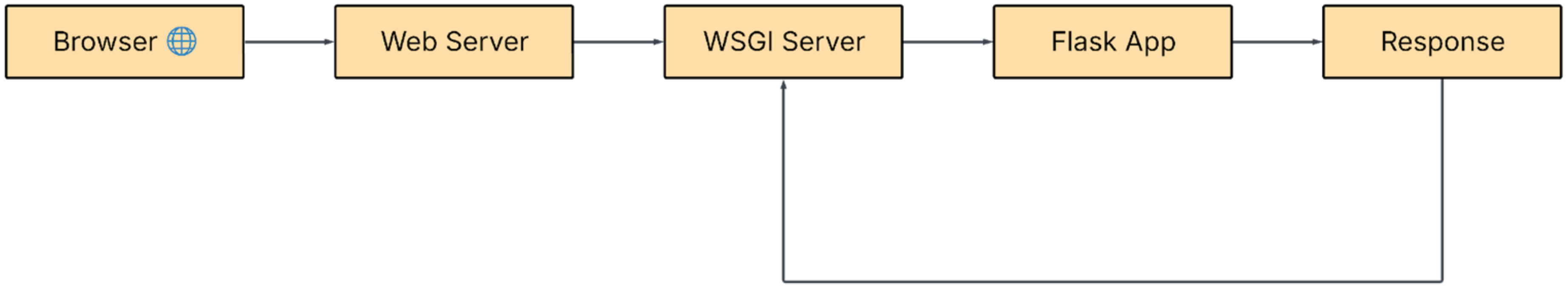
Flask Built-in Server vs. Production Server

- The built-in Flask server (app.run()) is great for development only.
- It is not efficient or secure enough for real-world traffic.
- For deployment, use a production-ready WSGI server like:
  - Gunicorn (Linux/macOS)
  - Waitress (Windows)

WSGI in Development

- Acts as a bridge between your Flask app and the web server
- Passes user requests → Flask app → responses back
- Handles multiple users and requests efficiently

Request Flow:



Common Ways to Deploy a Flask App

Deployment Type	Examples	Features	Best For
Local/Development Server	app.run()	Quick testing; runs locally only	Learning, small demos
Platform as a Service (PaaS)	Render, Railway, Heroku, PythonAnywhere	Easiest option; automatic setup	Beginners, student projects
Virtual Private Server (VPS)	AWS EC2, DigitalOcean, Linode	Full control; need manual setup	Intermediate–advanced users
Cloud Services	Google Cloud Run, Azure App Service, AWS Elastic Beanstalk	Scalable and secure	Large apps, professional use
Container-based Deployment	Docker, Kubernetes	Packages your app with all dependencies	Enterprise or advanced deployment

Common Ways to Deploy a Flask App

1. **Render**

   - Free and beginner-friendly
   - Automatic GitHub deployment
   - Automatically runs Flask apps using Gunicorn (WSGI)
   - Free SSL (HTTPS) and custom domain support

2. **Railway**

   - Simple interface for Flask and Python apps
   - Integrates directly with GitHub
   - Free tier available
   - Easy database integration

Common Ways to Deploy a Flask App

3. **PythonAnywhere**
   - Python-focused hosting
   - Upload your Flask app directly through the web interface
   - No complex server setup required
   - Great for students and small web projects

4. **VPS (Virtual Private Server)**
   - You rent your own small server (e.g., AWS EC2, DigitalOcean)
   - Full control over setup and security
   - Install dependencies manually
   - Usually use Nginx + Gunicorn (WSGI) to serve your app

Common Ways to Deploy a Flask App

5. **Docker & Containers**

- Package your Flask app and all dependencies into a container
- Ensures it runs the same everywhere
- Useful for scaling and production
- Can be hosted on AWS, Google Cloud, or Kubernetes

Common Ways to Deploy a Flask App

5. **Docker & Containers**

- Package your Flask app and all dependencies into a container
- Ensures it runs the same everywhere
- Useful for scaling and production
- Can be hosted on AWS, Google Cloud, or Kubernetes

Deploying on Render

Steps:

1. Push your Flask project to GitHub.
2. Go to https://render.com
3. Create a New Web Service.
4. Connect your GitHub repo.
5. Set Start Command:

```
gunicorn app:app
```

6. Click Deploy — Render will automatically install your dependencies and run your app.

Directory Structure

```
my_flask_app/
│
├── app.py
├── templates/
├── static/
├── requirements.txt
└── Procfile
```

Explanation:

- app.py → Main flask app
- templates/ → Jinja2 HTML templates
- static/ → images, CSS, and JS files
- requirements.txt → list of Python packages
- Procfile → tells hosting service how to run your app

Thank You