1. Flask is a lightweight, open-source Python web framework designed for building web applications. It differs from other frameworks like Django in its simplicity, flexibility, and minimalistic approach. Flask is suitable for small to medium-sized projects, whereas Django is more feature-rich and opinionated for larger projects.

2. The basic structure of a Flask application typically consists of the following components:

- `app.py` (or a similar file): The entry point of the application, where the Flask instance is created and routes are defined.
- `templates/` directory: This directory contains the HTML templates used for rendering web pages.
- `static/` directory: This directory stores static files like CSS, JavaScript, and images.

Example `app.py`:

python

```
from flask import Flask, render_template

app = Flask(__name__)

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

if __name___== '_main_':
    app.run(debug=True)
```

3. To install Flask and set up a project, follow these steps:

- Install Flask using pip: `pip install flask`
- Create a new directory for your project: `mkdir my_flask_app`
- Initialize a new Python file (e.g., `app.py`) and import Flask: `from flask import Flask`
- Create a Flask instance: `app = Flask(__name__)`

4. Routing in Flask maps URLs to Python functions. The `@app.route` decorator is used to define routes and associate them with view functions.

Example:

python

```
@app.route('/')
def index():
    return 'Hello, World!'

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

5. A template in Flask is an HTML file that can include dynamic content and placeholders. Flask uses the Jinja2 templating engine to render templates.

Example `index.html` template:

html

```
<!DOCTYPE html>
<html>
<head>
    <title>My Flask App</title>
</head>
<body>
```

```
    <h1>Welcome to {{ name }}!</h1>
    <p>This is a dynamic template.</p>
</body>
</html>
```

6. To pass variables from Flask routes to templates, you can use the `render_template` function and provide the variables as keyword arguments.

Example:

```
python
@app.route('/')
def index():
    name = 'Flask App'
    return render_template('index.html', name=name)
```

7. To retrieve form data submitted by users, you can use the `request.form` object in Flask. This object contains the form field values as key-value pairs.

Example:

```
python
from flask import request

@app.route('/login', methods=['POST'])
def login():
    username = request.form['username']
    password = request.form['password']
    # Process the form data
    return 'Login successful!'
```

8. Jinja templates are the templating engine used by Flask. They offer several advantages over traditional HTML, such as:

- Template inheritance: Allows you to create a base template and extend it with child templates, promoting code reuse and maintainability.
- Template filters: Provides built-in filters for modifying data before rendering, such as string formatting, data formatting, and more.
- Template control structures: Allows you to use control structures like loops and conditionals to generate dynamic content.
- Template macros: Enables the creation of reusable macros for common HTML patterns or components.

9. To fetch values from templates and perform arithmetic calculations in Flask, you can use template expressions and built-in filters.

Example:

html

```
<!-- templates/calc.html -->
<h1>Calculation Result</h1>
<p>{{ num1 }} + {{ num2 }} = {{ num1 + num2 }}</p>
```

python

```
@app.route('/calculate')
def calculate():
    num1 = 5
    num2 = 10
    return render_template('calc.html', num1=num1, num2=num2)
```

10. Here are some best practices for organizing and structuring a Flask project:

- Use the `Blueprint` feature to organize routes and templates based on functionality or features.
- Separate configuration settings into a separate file (`config.py`) for easier management and environment-specific settings.
- Use a dedicated directory structure for templates, static files, and other components.
- Leverage Flask extensions for common functionality like database integration, authentication, and caching.
- Use a virtual environment to manage project dependencies and avoid conflicts.
- Implement logging and error handling mechanisms for easier debugging and maintenance.
- Follow coding style guides and conventions, such as PEP 8 for Python code.
- Write unit tests and incorporate automated testing into your development workflow.

Submitted By

Thadikonda Revathi

20HU1A4232

Chebrolu Engineering College