DEVOPS PROJECT REPORT

“DOCKERIZING TO-DO-LIST APPLICATION USING PYTHON WITH FLASK AS THE WEB FRAMEWORK”

submitted in partial fulfillment of the requirement for the award of degree of

BACHELOR OF TECHNOLOGY
in
(Computer science and Engineering)

Submitted to

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA , PUNJAB.

JAN-MAY 2024

SUBMITTED BY

NAME : MUGLE SRUTHI
REG NO : 12109334
SECTION : K0309

STUDENT DECLARATION

I , [MUGLE SRUTHI], hereby declare that the report titled "TO-DO-LIST APPLICATION USING PYTHON WITH FLASK AS THE WEB FRAMEWORK" is my own original work. I have not copied or plagiarized any part of this report from any other source. I have acknowledged all sources that I have used in the preparation of this report, either in the text or in the references.

Signed,

[SIGNATURE]

MUGLE SRUTHI

DATE : 20-04-2024

ACKNOWLEDGEMENT

I would like to acknowledge the following people for their help in the preparation of this report:

- My instructor, [**Mrs. Harpreet kaur**], for their guidance and support throughout the course.
- The staff at the Lovely Professional University library for their assistance in finding resources.
- My classmates for their feedback and encouragement.
- **Thank you** for providing me with the opportunity to learn DevOps and for their excellent course materials.

I am grateful for all of the help that I received in completing this report. I hope that it is a valuable contribution to the field of DevOps

Sincerely,

Mugle sruthi

Reg no : 12109334

TABLE OF CONTENTS

INTRODUCTION :

This report aims to provide a comprehensive overview of the process involved in building a To-Do List application using Python with Flask as the web framework and containerizing it with Docker. The To-Do List application serves as a practical example to demonstrate the development of a web application and its deployment within a Docker container.

Overview of Flask and Docker:

- Flask: Flask is a lightweight and flexible web framework for Python. It provides tools, libraries, and technologies to build web applications.
- Docker: Docker is a platform that enables developers to build, ship, and run applications in containers. Containers allow applications to be isolated from the environment they run in, providing consistency across different environments.

- This report serves as a valuable resource for developers interested in learning about web development with Flask and containerization with Docker, providing a practical example of building and deploying a web application.

<u>ABOUT DOCKER :</u>

Docker is a platform that enables developers to package, distribute, and run applications in containers. Containers are lightweight, standalone, and portable units that contain everything needed to run an application, including the code, runtime, libraries, and dependencies.

Docker simplifies the process of building, deploying, and managing applications by abstracting away differences in operating systems and environments. It achieves this through containerization, which isolates applications and their dependencies from the underlying infrastructure, ensuring consistency and reproducibility across different environments.

<u>Key components of Docker include:</u>

- **Docker Engine**: The core component of Docker, responsible for creating and managing containers on a host system. It consists of a daemon process called dockerd, a REST API for interacting with the daemon, and a command-line interface (docker) for managing containers and images.

- **Docker Image:** A read-only template that contains the application code, runtime, libraries, and dependencies. Images are the building blocks of containers and can be created using Dockerfiles or pulled from Docker Hub, a public registry of pre-built Docker images.

- **Docker Container:** A runnable instance of a Docker image. Containers are lightweight, portable, and isolated environments that encapsulate the application and its dependencies. They can be started, stopped, moved, and deleted using Docker commands.

- **Dockerfile:** A text file that contains instructions for building a Docker image. Dockerfiles specify the base image, environment variables, dependencies, and commands needed to set up and configure the application environment.

- **Docker Compose:** A tool for defining and managing multi-container Docker applications. Docker Compose uses YAML files (docker-compose.yml) to define the services, networks, and volumes required by an application, making it easy to orchestrate complex deployments with multiple containers.

Overall, Docker revolutionizes the way applications are developed, deployed, and managed by providing a consistent and efficient platform for containerization. It promotes agility, scalability, and portability, enabling developers to build and deploy applications with ease across different environments.

Building the To-Do List Application:

- **Setup:** The development environment is set up with Python and Flask installed. A basic Flask application is created to handle routes for adding tasks, marking tasks as complete, and rendering HTML templates.

- **Functionality:** The application allows users to add tasks to a to-do list, view the list of tasks, and mark tasks as complete.

Implementation: The Flask application is implemented with routes for adding tasks (/add), marking tasks as complete (/complete/<task_id>), and rendering the homepage (/). HTML templates are used for the user interface.

◆ Containerizing the Application with Docker:

- **Dockerfile:** A Dockerfile is created to specify the steps for building a Docker image. It includes instructions to install dependencies, copy the application code, expose ports, and define the command to run the Flask application.

- **Building the Image**: The Docker image is built using the Dockerfile and the docker build command.

- **Running the Container:** A Docker container is launched from the Docker image using the docker run command. Port mapping is configured to expose the Flask application running inside the container to the host machine.

<u>Explanation about the project :</u>

<u>1) IMPORTS:</u>

from flask import Flask, render_template, request, redirect, url_for

- **Flask:** This is the main class of the Flask web framework. It's used to create an instance of the web application.
- **render_template:** This function is used to render HTML templates.
- **request:** This object contains the incoming request data.
- **redirect:** This function is used to redirect the user to another endpoint.
- **url_for:** This function is used to generate URLs for endpoints.

2)**Create Flask App Instance**:

app = Flask(__name__)

This creates an instance of the Flask class. _name_ is a special Python variable that represents the name of the current module.

3)To do List:
todo_list = []
This initializes an empty list to store the todo list items.

4)**Routes**:

- **Index Route**:

```
@app.route('/')
def index():
    return render_template('index.html',
todo_list=todo_list)
```

This route corresponds to the homepage (/). When a user visits this route, the index() function is called, which renders the index.html template and passes the todo_list to it.

Add Route:

python

Copy code

```
@app.route('/add', methods=['POST'])
def add():
    task = request.form['task']
    todo_list.append(task)
    return redirect(url_for('index'))
```

This route handles adding tasks to the todo list. It expects a POST request containing form data with a field named 'task', representing the task to add. It then appends the task to the todo_list and redirects the user back to the homepage.

Complete Route:

python

Copy code

```
@app.route('/complete/<int:task_id>')
def complete(task_id):
    if task_id < len(todo_list):
        todo_list.pop(task_id)
    return redirect(url_for('index'))
```

This route handles completing tasks. It expects the task_id as part of the URL, indicating the index of the task to be completed. If the task_id is valid (i.e., within the range of indices of todo_list), the corresponding task is removed from the todo_list, and the user is redirected back to the homepage.

Main Block:

python

Copy code

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

This block ensures that the Flask app is only run if the script is executed directly (not imported as a module into another script). It starts the Flask application server with debugging enabled.

Overall, this code sets up a simple to-do list application with basic functionalities such as adding tasks, viewing tasks, and marking tasks as complete. The frontend is rendered using HTML templates, and the backend logic is implemented using Flask routes.

EXPLANATION ABOUT HTML CODE:

1)Document Type Declaration (DOCTYPE):

<!DOCTYPE html>

This declaration specifies the document type and version of HTML being used.

2)HTML Element:

<html lang="en">

This is the root element of the HTML document. The lang attribute specifies the language of the document (English in this case).

3)Head Section:

```
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width,
initial-scale=1.0">
   <title>Todo List</title>
</head>
```

The <head> section contains metadata about the document, such as character encoding and viewport settings. The <title> **element** sets the title of the webpage, which appears in the browser's title bar or tab.

4)Body Section:

<body>

The <body> section contains the visible content of the webpage.

5)Heading:

```
<h1>Todo List</h1>
```

This is a level 1 heading that displays the title of the todo list.

6)Form for Adding Tasks:

```
<form action="/add" method="post">
    <input type="text" name="task" placeholder="Enter task">
    <button type="submit">Add Task</button>
</form>
```

This <form> element allows users to input new tasks. It has an action attribute set to /add, which specifies the route to submit the form data to (handled by Flask). The method attribute is set to post, indicating that the form data should be sent as a POST request. Inside the form, there's a text input field (<input>) **named "task" where users can enter their task. The submit button (<button>) submits the form.**

7)List of Tasks:

```
<ul>
    {% for index, task in enumerate(todo_list) %}
        <li>{{ task }} <a href="/complete/{{ index }}">Complete</a></li>
    {% endfor %}
</ul>
```

This <ul> element creates an unordered list to display the tasks.

Within the list, a loop is used to iterate over each task in the todo_list. The enumerate() function is used to get both the index and the task itself.

Each task is displayed as a list item (<li>), followed by a link (<a>) labeled "Complete". The link's href attribute points to the /complete/<index> route, where <index> is the index of the task in the todo_list.

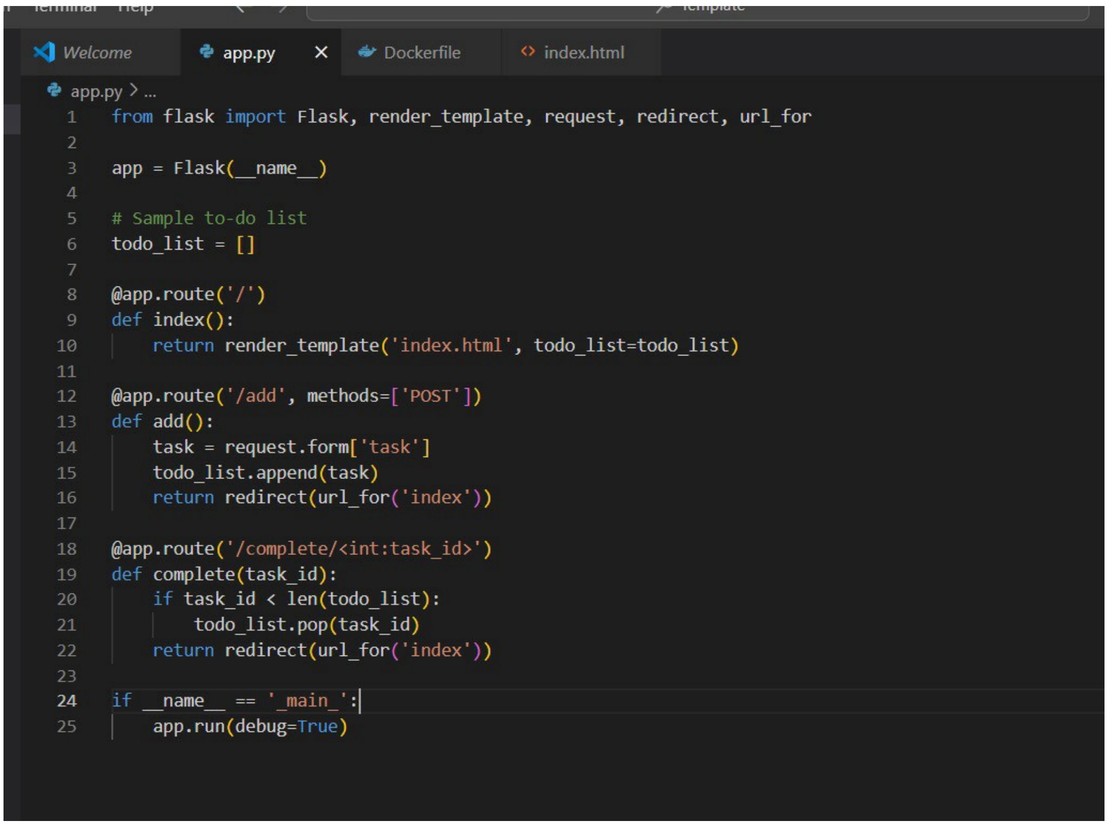
8)Closing Tags:

</body>

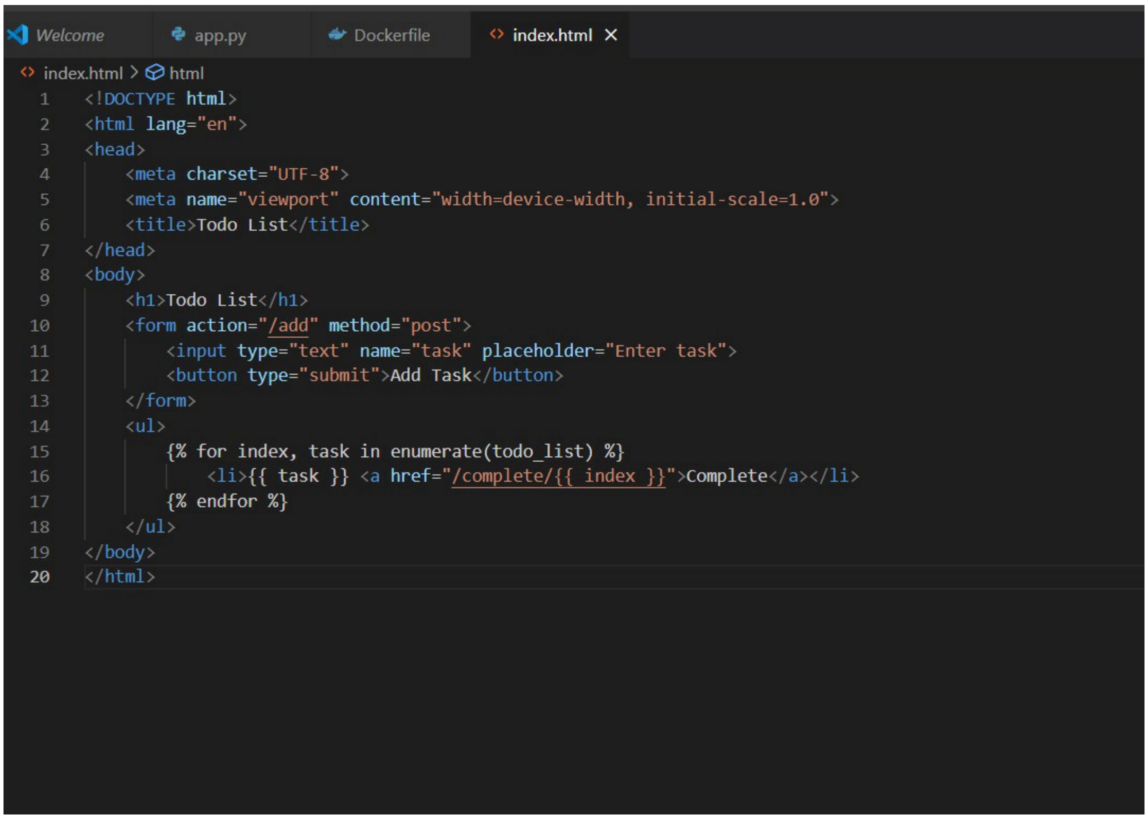
</html>

These closing tags close the <body> and <html> elements, respectively, marking the end of the HTML document.

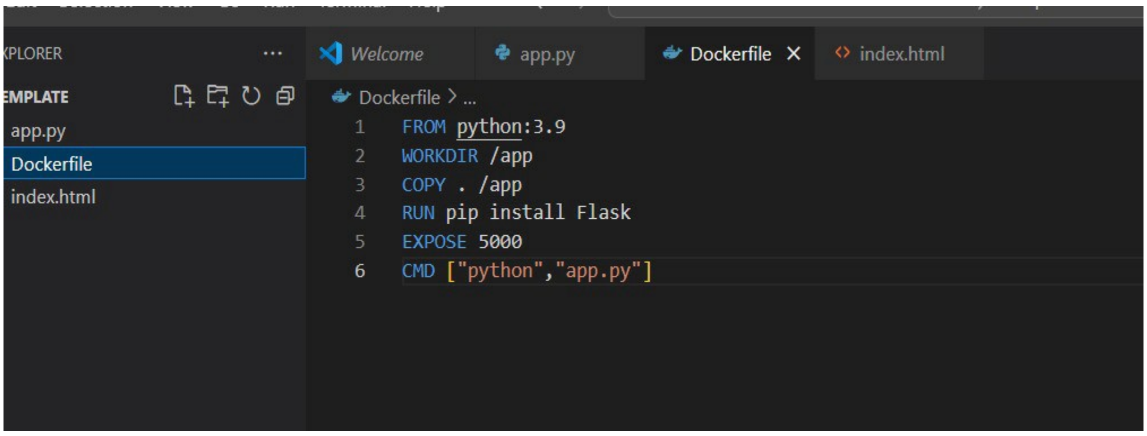
IMPLEMENTATION:



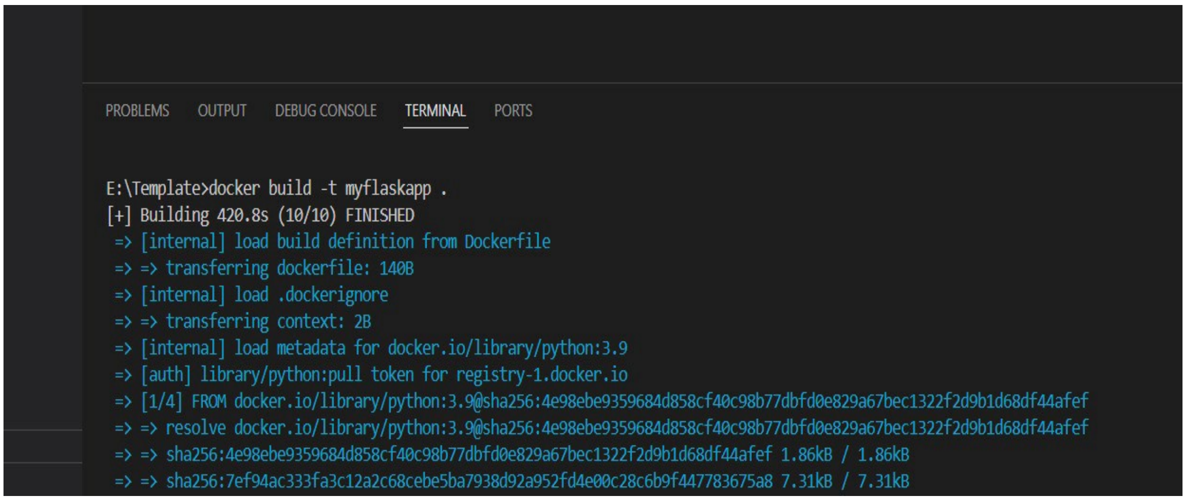
```python
from flask import Flask, render_template, request, redirect, url_for

app = Flask(__name__)

# Sample to-do list
todo_list = []

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

@app.route('/add', methods=['POST'])
def add():
    task = request.form['task']
    todo_list.append(task)
    return redirect(url_for('index'))

@app.route('/complete/<int:task_id>')
def complete(task_id):
    if task_id < len(todo_list):
        todo_list.pop(task_id)
    return redirect(url_for('index'))

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



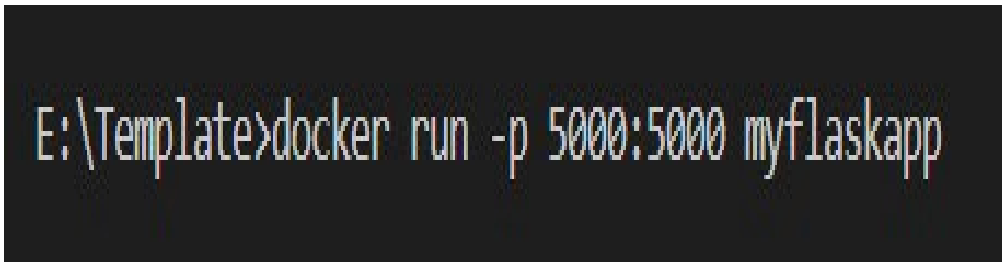
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Todo List</title>
</head>
<body>
    <h1>Todo List</h1>
    <form action="/add" method="post">
        <input type="text" name="task" placeholder="Enter task">
        <button type="submit">Add Task</button>
    </form>
    <ul>
        {% for index, task in enumerate(todo_list) %}
            <li>{{ task }} <a href="/complete/{{ index }}">Complete</a></li>
        {% endfor %}
    </ul>
</body>
</html>
```



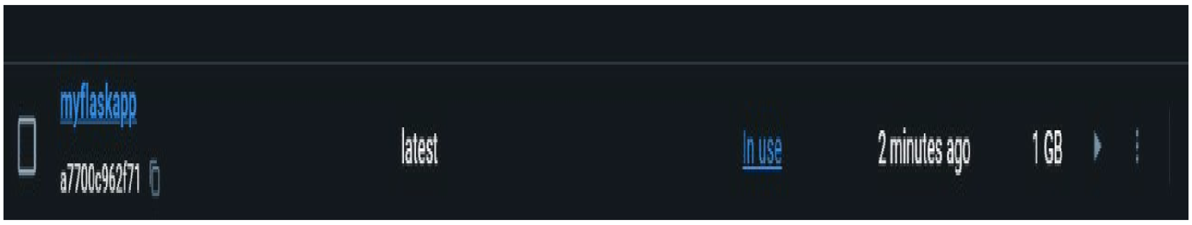
```
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install Flask
EXPOSE 5000
CMD ["python","app.py"]
```



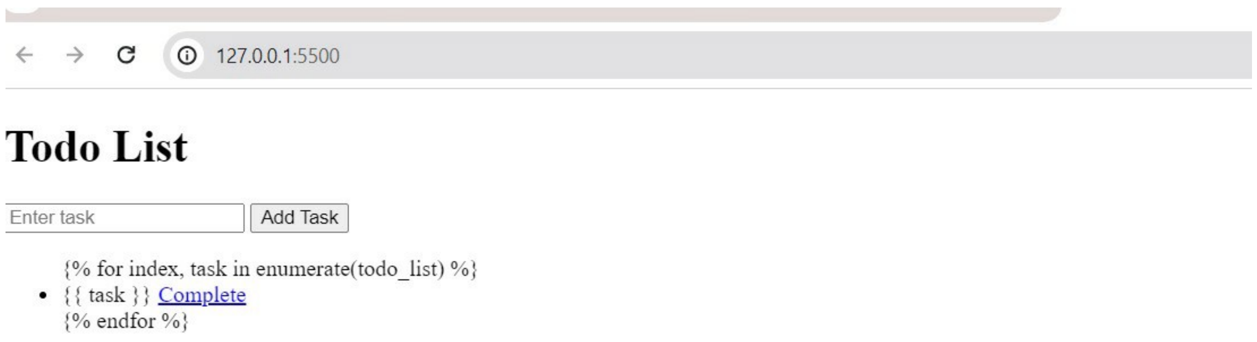
```
E:\Template>docker build -t myflaskapp .
[+] Building 420.8s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 140B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.9
 => [auth] library/python:pull token for registry-1.docker.io
 => [1/4] FROM docker.io/library/python:3.9@sha256:4e98ebe9359684d858cf40c98b77dbfd0e829a67bec1322f2d9b1d68df44afef
 => => resolve docker.io/library/python:3.9@sha256:4e98ebe9359684d858cf40c98b77dbfd0e829a67bec1322f2d9b1d68df44afef
 => => sha256:4e98ebe9359684d858cf40c98b77dbfd0e829a67bec1322f2d9b1d68df44afef 1.86kB / 1.86kB
 => => sha256:7ef94ac333fa3c12a2c68cebe5ba7938d92a952fd4e00c28c6b9f447783675a8 7.31kB / 7.31kB
```



```
E:\Template>docker run -p 5000:5000 myflaskapp
```



```
myflaskapp      latest      In use      2 minutes ago      1 GB
a7700c962f71
```


127.0.0.1:5500

Todo List

```
    {% for index, task in enumerate(todo_list) %}
  • {{ task }} Complete
    {% endfor %}
```

Conclusion:

In conclusion, this report provides a detailed guide on building a To-Do List application using Python with Flask and containerizing it with Docker. By following the steps outlined in this report, developers can create and deploy web applications in a consistent and efficient manner using modern development practices.

Future Directions : Future work may involve enhancing the functionality of the To-Do List application, implementing user authentication, deploying the application to a cloud platform, and exploring advanced Docker features such as Docker Compose for multi-container applications.