FLASK

- Flask is a micro web framework written in python
- Flask is backend but uses jinja2 for frontend so is full stack
- It has no database layer, form validation etc, this are used from third parties

Included with flask:

1) Jinja 2 : for frontend (templating)
2) Werkzeug : HTTP and routing
3) Development server and debugger
4) Unit testing support

- Start python flask project : pip install flask
  (inside working directory)

Format :-

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

run as
python script.py

```
@app.route('/')
def hello():          <- (views)
    return "Hello world"

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

it is running on debug mode (development)

can add more than one routes
@app.route('/')
@app.route('/home')

the functions hello() is called view function
flask follows Model template - View pattern

- Passing data from URL

```
@app.route ('/home/<string:name>')
def hello (name):
    return "Hello" + name
```

eg /home/Palash → Hello Palash

29 Tuesday

- Templating

Component responsible for showing data

1) make folder called templates, make files inside it, eg index.

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

30 Wednesday

import as : from flask import Flask, render_

also possible

```
def index():
    return "<h1> Home Page </h1>"
```

Template Inheritance (Blocks)

=> All common things of html, we can put in seperate file eg base.html.

base.html =>

```
<!Doctype html>
<head>
```

for title → {% block head %} {% endblock %}

```
</head>
<body>
```

for body → {% block body %} {% endblock %}

```
</body>
</html>
```

Friday 25

index.html

```
{% extends 'base.html' %}
{% block head %}
<title> Home </title>
{% end block %}
{% block body %}
<H1> Home Page </h1>
{% end block %}
```

Passing data betn python and html.

in script.py

```
@app.route("/")
def home():
    return render_template("home.html", message="Hiiiii")
```

here message is a variable which can be used anywhere in home.html.

22 Tuesday

in home.html ⇒ (done using jinja 2)

```
<title> {{ message }} </title>
<body> Welcome {{message}} </body>
```

eg 2 app.py

```
from flask import Flask, render_template
app = Flask(__name__)
all_posts = [
    {'title': 'Post1',
     'content': 'This is Post 1'},
    {'title': 'Post 2',
     'content': 'This is Post 2'}]
```

23 Wednesday

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

in posts.html

```
{% extends 'base.html' %}

{% block head %}
    <title> Posts </title>
{% endblock %}

{% block body %}
<h1> All Posts </h1>

{% for post in posts %}
    <h2> {{ post.title }} </h2>

    <p> {{ post.content }} </p>

{% endfor %}

{% endblock %}
```

to add conditions (loop, if else etc)

for variables

Friday 18

Eg of Model-Template-View

Model → • Data • Usually from database

Template → • Presentation • Generates HTML
• using Jinja 2

View → • Behaviour • Python function
• Mapped to URL

using json as sample database.

- templates → flashcard.html
- flashcard_db.json
- model.py
- app.py.

app.py

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

@app.route("/flashcards")
def cards():
    card = db[0]
    return render_template("flashcards.html",
                           card = card)
```

flashcards_db.json

```
[
  {
    "question": "What's your age?",
    "answer": "23"
  },
  {
    "question": "What is your name?",
    "answer": "Palash Bajpai"
  }
]
```

model.py

```
import json
def load_db():
    with open("flashcards_db.json") as f:
        return json.load(f)
db = load_db()
```

in flashcards.html

```
<body>
  Question : {{ card.question }} <br>
  Answer : {{ card.answer }}
</body>
```

ERROR HANDLING

this code except index of json data to be shown, if given index not exist it shows error. with 404 response

- abort has to be imported

```
from flask import Flask, render_template , abort

@app.route("/flashcards/<int:index>")
def cards(index):
    try:
        card = db[index]
        return render_template("flashcards.html", card=card)
    except IndexError:
        abort(404)
```

Get card index from user & 404 error

we can get index from URL as parameter
use → try & except to handle error

```
from flask import abort
@app.route("/card/<int:index>")

def card_view(index):
    try:
        card = db[index]
        return render_template("card.html", card = card)
    except IndexError:
        abort(404)
```

Friday 4

Building links for cards (url_for)

→ creating a link to Home page

note

Sunday 6

home page has view function hello.

inside card.html.

```
→ <body>
    Home Page
    <a href = "{{ url_for('hello') }}"> Home </a>
```

name of view function

creating next button to change cards.

card.html

```
<body>
  <button>
    <a href "{{ url_for ('card_view', inden =index+1)}}">
      Next card
    </a>
      , </button>
```

in main page:

```
@app.route ('/card/<int: index>')
def card_view(inden):
    try:
        card = db [inden]
        return render_template ("card.html", card =card , index =inden)
    except IndexError:
        abort (404)
```

Use of if-else

→ When we reach to last card, send us to start page

card.html

```
<button>
{% if index < max_index %}
    <a href="{{ url_for('card_view', index = index+1) }}">
        Next Card </a>
{% else %}
    <a href="{{ url_for('card_view', index = 0) }}">
        Start over </a>
{% endif %}
</button>
```

in flashcard.py or main.py (from which card.html is called)

```
@app.route('/card/<int:index>')
def card_view(index):
    card = db[index]
    return render_template('card.html', card=card, index=index,
                           max_index = len(db) - 1)
```

Serving REST API

API - it is an interface through which one program or website talks to another. They are used to share data and services.

REST (Representational state transfer) describes the general rules for how data & services are represented through the API so that other program will be able to correctly request & receive data and services that API makes available

⇒ we can't directly return json (since it is list now), we need to jsonify it in main.py

```
@app.route('/api/cardlist')
def api_cards():
    return jsonify(db)
```

→ this api can be used by other functions.

FORMS

by default only GET method is available in route, so to use post we need to include it.

```
@app.route('/add_card', methods = ['GET', 'POST'])
```

to access value of form, use its name value

```
request.form['question']
```

name value

⇒ add_card.html

```
<form method = "post">
  <p>
    Question:
    <input type = "text" name = "question">
  </p>
  <p> Answer:
    <input type = "text" name = "answer">
  </p>
  <button type = "submit"> Add Card </button>
</form>
```

⇒ model.py ← add fn to save updated db.

```
import json
def load_db():
    with open("cards_db.json") as f:
        return json.load(f)
def save_db():
    with open("cards_db.json", 'w') as f:
        return json.dump(db, f)
db = load_db()
```

⇒ main.py

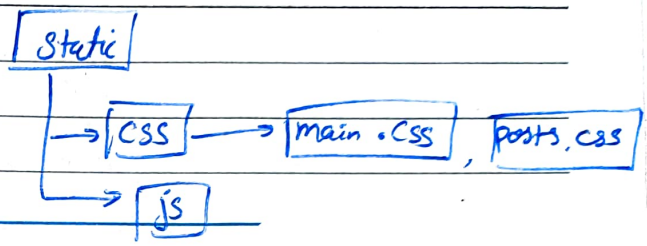
```
@app.route('/add_card', methods=['GET', 'POST'])
def add_card():
    if request.method == 'POST':
        card = {"question": request.form['question'],
                "answer": request.form['answer']}
        db.append(card)
        savedb()
        return redirect(url_for('card_view', index = len(db)-1))

    else
        return render_template('add_card.html')
```

Styling

To add css & js files

- make a folder static



add link in 2 ways.

1) <link rel = "stylesheet" href = "static/css/main.css">

2) <link rel = "stylesheet"
href = "{{ url_for('static', filename = 'style.css')}}">

add it in head.

⇒ CSS file is like normal css
⇒ to add main.css to all together, add it is base.html.
⇒ can add inline
styles or internal style

- Note
if you make change in CSS & it does not change in browser so either open in incognito mode or refresh cache using fn + F5.
It happens due to cache.

Deploy to heroku

→ in heroku
make new app → qna-cards.
install heroku cli

→ in app folder → in cmd line

- git init
- heroku login
- heroku git:remote -a qna-cards (app name in heroku)
- pip install gunicorn

• Make 2 file - requirement.txt (empty)
- Procfile
web: gunicorn main:app
(main: for main.py file)

- pip freeze > requirements.txt
- git add .
- git commit -m "first commit"
- git push heroku master