Flask interview questions

How to create minimal `Flask` application?

```
from flask import Flask


# Application instantiation
app = Flask(__name__)


# app.route to bind URL to the function
@app.route('/')
def hello_world():
    return 'Hello, World!'
```

How to run `Flask` application?

- We can run `Flask` application using `flask --app <file-name> run` command
- We can also use shortcut `flask run` command provided application instantiation has been done under files with names either `wsgi.py` OR `app.py`

How is the typical `Flask` project layout? [Important question]

- `Flask` follows `M-V-T` architecture (Also known as `Model - View - Template`)
- This is analogous to `MVC` architecture in other web frameworks.
- A `Flask` project layout typically contains multiple sub-applications based on business requirement.
- Each sub-application is basically an isolation of business logic.
- Each sub-application will be created as independent Python package under project root.

- Each sub-application will be written and combined to a `Blueprint` object (`from flask import Blueprint`).
- And finally, the sub-application is registered to the main application (aka orchestrator application) that is available at project root
- Typically application instantiation is done under `main.py` file available at project root (We can name the file anything for that matter).
- File where `app = Flask(__name__)` is the main application file (aka `entrypoint` of project)
- We have a `static` directory under each sub-application containing images, css code and other static files required by respective sub-app.
- We have a `templates` directory under each sub-application containing `HTML` code files required by respective sub-app.
- Typically, we have `venv` folder which is virtual environment.
- At project root level, we also have `README.md` file for project documentation and `requirements.txt` file for dependency management (packages and their versions)
- At project root level, we can also have a `models` package which represents all database related models, schemas and DB manipulation code.
- **Here is the sample structure (for demo purpose)-**

~~~
```
/home/user/Projects/MyFlaskProject       ( PROJECT ROOT )
├── MySubApplication1/                   ( SUB APPLICATION 1 )
|   ├── __init__.py
|   ├├── models/
|   |   ├── basemodel.py
|   |   ├── dal/
|   |   |   ├── __init__.py
|   |   |   └── dml.py
|   |   └── datamodels/
|   |       ├── resource_1.py
|   |       ├── resource_2.py
|   |       └── resource_3.py
|   ├── DDL_database_.sql
|   ├── auth.py                        ( VIEW - BUSINESS LOGIC )
|   ├── views.py                       ( VIEW - BUSINESS LOGIC )
|   ├── templates/                     ( FRONT END CODE )
|   |   ├── base.html
|   |   ├── auth/
~~~

```
│   │   │   ├── login.html
│   │   │   └── register.html
│   │   └── portal/
│   │       ├── create.html
│   │       ├── index.html
│   │       └── update.html
│   └── static/                      ( STATIC FILES - IMAGES & CSS )
│       └── style.css
│       └── logo.png
├── tests/                          ( TEST CASES - UNIT TESTING )
│   ├── test_fixtures.py
│   ├── test_db.sql
│   ├── test_auth.py
│   └── test_create.py
├── venv/                           ( VIRTUALENV )
├── main.py                         ( MAIN APPLICATION FILE )
└── requirements.txt                ( PROJECT DEPENDENCIES )
└── README.md.                      ( PROJECT DOCUMENTATION )
```
```

What is request data validation in `Flask` (OR in web applications in general)?

- We validate request data (aka `payload`, aka `request body`) to ensure the data coming from HTTP request is in expected format.
- We need to perform validation in input data to ensure the HTTP request coming from client application is in desired and agreed format.
- `Pydantic` *is a library that provides data validation and settings management using type annotations.* ***We are using*** `Pydantic` ***in our project.***
- `Pydantic` provides a functionality to define schemas which consist of a set of properties and types to validate a payload.
- There are many more such libraries in Python that provide such data validation features.

What is response data validation in `Flask` (OR in web applications in general)?
```

- We validate response data (aka `payload`, aka `response body`) to ensure the data being sent back to the client application is in expected format.
- We need to perform validation on output data to ensure the HTTP response being sent to the client application is in desired and agreed format.
- *`Pydantic` is a library that provides data validation and settings management using type annotations.* ***We are using `Pydantic` in our project.***
- There are many more such libraries in Python that provide such data validation features.

---

## What is `Pydantic` library?

- **Validation Schemas:** `Pydantic` provides a functionality to define schemas which consist of a set of properties and types to validate a payload.
- *`Pydantic` is a library that provides data validation and settings management using type annotations.* ***We are using `Pydantic` in our project.***
- `Pydantic` will essentially handle the data parsing and validation, among other cool features.

---

## How to configure `Flask` flask application to run on public IP?

```
# Assuming `main.py` is the file where our application is instantiated.
flask --app main run --port 8080 --host 0.0.0.0
```

How to run `Flask` application in `debug` mode? What is `debug` mode?

- In `debug` mode, application restarts automatically with any code change under project root directory.

- We can use `debug` mode only for development purpose.
- In production mode setting `debug` mode on is a bad practice. With `debug=True` attacker can send along some malicious code to run on server side to steal application data.

```
# Assuming `main.py` is the file where our application is instantiated.
flask --app main --debug run
```

How to run `Flask` application from a Python script?

```
from flask import Flask

# Flask` is the class we use to instantiate an application.
app = Flask(__name__)


# `route` decorator allows us to bind function to certain `relative URL path`.
@app.route("/")
def hello_world():
    print(f"{__name__} running")
    return "<p>Hello, World!</p>"


# How to run flask application like we run any Python script.
if __name__ == "__main__":
    app.run(host="localhost", port=8080, debug=True)
```

How to capture `path variable` from URL in `Flask`?

```
from flask import Flask


# Flask` is the class we use to instantiate an application.
app = Flask(__name__)
```

```
@app.route("/user/<username>")
def show_user_profile(username):
    # Show the user profile for that user
    return f"User - {username}"


@app.route("/post/<int:post_id>")
def show_post(post_id):
    # Show the post with the given id, the id is an integer
    return f"Post {post_id}"
```

How to get URL binding associated with each function defined in the `Flask` app? What is use of `url_for` function in `Flask`?

```
from flask import Flask, url_for


# Flask` is the class we use to instantiate an application.
app = Flask(__name__)


@app.route('/')
def index():
    print(f"{__name__} running")
    return 'index'


@app.route('/user/<username>')
def profile(username):
    return f'{username}\'s profile'


with app.test_request_context():
    print(url_for('index'))
    print(url_for('profile', username='John Doe'))
```

How to do URL mapping to any function in `Flask`?

- We use `app.route` decorator and specify “relative URL” binding with the function
- Routing URLs for example -

```
@app.route("/login", methods=["GET", POST", "PATCH", "DELETE", "PUT"]))
def random_method():
    # LOGIC
    return Response('{}')
```

How to capture `Flask` request attributes?

- `Flask` has a global object called `request`. We can import it as -

`from flask import request`

- This global `request` object has all the attributes of HTTP request.
- Few examples -
    - `request.args` to capture query parameters
    - `request.json` to capture JSON input payload coming from HTTP request object.
    - `request.method` to capture type of HTTP method (`GET,` `POST`, `DELETE`,… etc)
    - `request.mimetype` to capture "Content-Type" in request payload.
    - `request.form` to capture form-data if request "Content-Type" is an HTML form-data.
    - `request.url` to capture complete request URL
    - AND many more.

What are HTTP verbs and their use cases?

- Also known as `HTTP request methods`.
- There are many HTTP verbs. But we use `GET`, `POST`, `DELETE`, `PATCH`, `PUT` primarily for writing most of the web applications.
- **Refer mozilla developer document** -

HTTP request methods - HTTP | MDN

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

- Details -

  `GET`

  The `GET` method requests a representation of the specified resource. Requests using `GET` should only retrieve data.

  `HEAD`

  The `HEAD` method asks for a response identical to a `GET` request, but without the response body.

  `POST`

  The `POST` method submits an entity to the specified resource, often causing a change in state or side effects on the server.

  `PUT`

  The `PUT` method replaces all current representations of the target resource with the request payload.

  `DELETE`

  The `DELETE` method deletes the specified resource.

  `CONNECT`

  The `CONNECT` method establishes a tunnel to the server identified by the target resource.

  `OPTIONS`

  The `OPTIONS` method describes the communication options for the target resource.

  `TRACE`

  The `TRACE` method performs a message loop-back test along the path to the target resource.

  `PATCH`

  The `PATCH` method applies partial modifications to a resource.

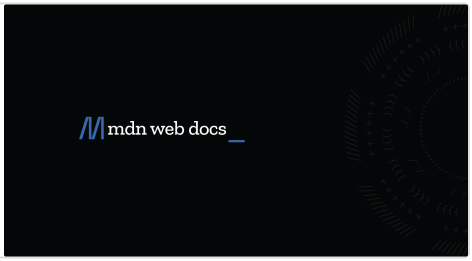
What are HTTP status codes?

- **Refer mozilla developer document -**

HTTP response status codes - HTTP | MDN

This interim response indicates that the client should continue the request or ignore the response if the request is already finished. 101 Switching Protocols This code is sent in response to an request

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status



HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

1. Informational responses (`100` – `199`)
2. Successful responses (`200` – `299`)
3. Redirection messages (`300` – `399`)
4. Client error responses (`400` – `499`)
5. Server error responses (`500` – `599`)

💡 **MOST asked HTTP codes in the interview are -**
200 (`success`), 201 (`created`), 301 (`redirect`), 400 (`bad request`) , 401 (`un-athorized`), 403 (`forbidden`), 404 (`not-found`), 405 (`method not allowed`), 500 (`internal-server error`)

How to render static HTML pages from `Flask` app?

- We use `render_template` function from `flask` module.
- We need to create a directory called `templates` at application level and place our HTML file under it.
- For example -

```
from flask import Flask, render_template

app = Flask(__name__)


@app.route("/hello")
def hello():
    return render_template("hello.html")
```

How to render dynamic HTML pages from flask app?

- `Flask` uses a library called `Jinja2` to produce dynamic HTML pages.
- We can use `render_template` method along with additional keyword parameters as required by the HTML template.
- For example -

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

- HTML template can look like -

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello, World!</h1>
{% endif %}
```

How to embed Python code in html code from `Flask`?

- `Flask` uses a library called `Jinja2` to produce dynamic HTML pages.

- We can use `render_template` method along with additional keyword parameters as required by the HTML template.
- `Jinja2` templates have their own syntax rules such as -
  - `{% ... %}` for Statements
  - `{{ ... }}` for Expressions to print to the template output
  - We write `if block` as -

    ```
    {% if True %}
            yay
    {% endif %}
    ```

  - OR `for loop` as -

  ```
  {% for item in sequence -%}
      {{ item }}
  {%- endfor %}
  ```

  - And some more similar rules (**REFER** - https://jinja.palletsprojects.com/en/2.11.x/templates/).

How to maintain project dependencies in any project?

- There are many libraries to create virtual environment.
- One of the popular library is `virtualenv`.
- We create environment using `virtualenv` under each project-root.
- Typically environment directory is named as `venv` (created as `virtualenv -p python3 venv`)
- After every third party package installation, we overwrite `requirements.txt` file with output of `pip freeze`
- Command is - `pip freeze > requirements.txt`
- To install all the packages under `requirements.txt` the command is -

```
pip install -r requirements.txt
```

How to redirect request from one url to other?

- We can use `redirect` function available under `flask` module.
- Right click in `chrome` browser and `inspect elements` to see "network" actions
- We will see one action as `redirection`

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


game = Flask(__name__)


@game.route("/success/<name>")
def success(name):
    return f"<h1> success - {name}</h1>"


@game.route("/login", methods=["POST", "GET"])
def login():
    if request.method == "POST":
        player = request.form.get("player")
        success_url = url_for("success", name=player)
        return redirect(success_url)

    return "<h1>DONE</h1>"


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

How to use global variable to build view count app on each page reload?

```
from datetime import datetime
from flask import Flask

app = Flask(__name__)


@app.route("/")
def welcome():
    return "Welcome to my Flash Cards application!"


@app.route("/date")
def date():
    return "This page was served at " + str(datetime.now())


counter = 0


@app.route("/count_views")
def count_demo():
    global counter
    counter += 1
    return "This page was served " + str(counter) + " times"
```

How to load `Flask` configurations?

- `app` object instantiated using `Flask` class has an attributed called `app.config`
- `app.config` is a dictionary object and can be loaded with all the required configurations.
- We can use these configurations throughout the project.

What is MVC architecture?

- MVC stands for `Model - Control - View`
- It's a generic concept used by many web frameworks for isolation of code.
- `Model` is the directory containing all database related code.

- `Control` contains all the business logic
- `View` contains all the front end related code.
- `Flask` follows `M-V-T` architecture (Also known as `Model - View - Template`)
- For example -

```
project root (swapi)
    model
        character
        film
        species

    view (front end)
        html (building web pages)
        css (styling)
        php
        javascript

    control (business logic)
        student
        pensioner
        saving
```

Attributes of `app` object?

`app.config` - for application configurations

`app.route` - routing URLs (for example, `@app.route("/login", methods=["GET", POST"])`)

`app.get` - defining HTTP GET Methods

`app.post` - defining HTTP POST methods

💡 Like `app.get` you can define rest of the HTTP method and their URLs

---

How to produce `Flask` Response object?

- We can pass status code, Content-Type and other Response attributes along with `Response` object.
- For example -

```
import json
from flask import Flask, Response


# Application instantiation
app = Flask(__name__)


# app.route to bind URL to the function
@app.route('/')
def hello_world():
    response_obj = {
        "status": "success",
        "message": "Hello, world!!!"
    }
    return Response(
        json.dumps(response_obj),
        status=200,
        mimetype="application/json"
    )
```

Why do we use `Flask` Blueprints to create sub-applications?

- A `Flask` project layout typically contains multiple sub-applications based on business requirement.
- Each sub-application is basically an isolation of business logic.
- Each sub-application will be created as independent Python package under project root.
- Each sub-application will be written and combined to a `Blueprint` object (`from flask import Blueprint`).
- And finally, the sub-application is registered to the main application (aka orchestrator application) that is available at project root

- Typically application instantiation is done under `main.py` file available at project root (We can name the file anything for that matter).