Syrian Arab Republic

Lattakia - Tishreen University

Department of Communication and electrical engineering

5^{th}: Network Programming project

الجمهورية العربية السورية

اللاذقية- جامعة تشرين

كلية الهندسة الكهربائية والميكانيكية

قسم هندسة الاتصالات والالكترونيات

السنة الخامسة : مشروع برمجة شبكات

- عنوان البحث باللغة الأجنبية :

REST WebServer to give student marks

- المجال العلمي للبحث :

برمجة الشبكات باستخدام لغة البايثون

- أسماء الباحثين و مرتبتهم العلمية :

١. ريما محمد قاسم 2269 - طالبة في كلية الهندسة الميكانيكية و الكهربائية
٢. سامي علي سليمان 2563 - طالب في كلية الهندسة الميكانيكية و الكهربائية
٣. نغم طلال مصطفى 2183 - طالبة في كلية الهندسة الميكانيكية و الكهربائية

2021

Research Summary:

• We will talk in the search in general about RESTfuL WebServer technology, where we will learn about its definition and then on:

1. Its key elements:
   (Resources, Request Verbs , Request Headers, Response Body, Response Status codes)

   A full explanation of each of them

2. Different methods (POST, GET, PUT, DELETE)
3. Its s Architectur and the importance of its use
4. Its principles and Constraints

   (RESTFul Client-Server, Stateless, Cache, Layered System, Interface/Uniform Contract)

• Then later we will get to know specifically about flask http technology, where we will talk about:

1. Basics of http . protocol

2. Its diverse methods

3. We will then move to the flask curricula and support the previous curricula with educational programming examples

• Then we will finally move to the main code learned from the theoretical content and we will list the steps for creating the code and we will explain the instructions in it with an explanation

• Where as an additional work the article was uploaded to the GITHUB platform

ملخص البحث :

- سنتحدث في البحث بشكل عام عن تقنية RESTfuL WebServer حيث سنتعرف على تعريفها و من ثم على:

1. عناصره المفتاحية :
(Resources, Request Verbs , Request Headers, Response Body, Response Status codes)
وشرح وافي عن كل منها
2. المناهج المختلفة (POST,GET,PUT,DELETE)
3. بنيتها و أهمية استخدامها
4. مبادئه و قيوده

(RESTFul Client-Server, Stateless, Cache, Layered System, Interface/Uniform Contract)

- ثم فيما بعد سنتعرف بشكل مخصص على تقنية flask http حيث سنتحدث عن:

1. أساسيات بروتوكول http
2. مناهجه المتنوعة
3. سوف ننتقل بعدها لمناهج flask و سندعم المناهج السابقة بأمثلة برمجية تعليمية

- ثم سوف سننتقل في النهاية للكود الرئيسي المتعلم من المحتوى النظري و سنقوم بسرد خطوات إنشاء الكود و سنشرح التعليمات الموجودة فيه مع توضيح
- حيث كعمل إضافي تم رفع المقال إلى منصة GITHUB

كلمات مفتاحية (key words) :

(restfull webserver, flask ,http, GET , POST , PUT , DELETE , CACHE, SERVER-CLIENT)

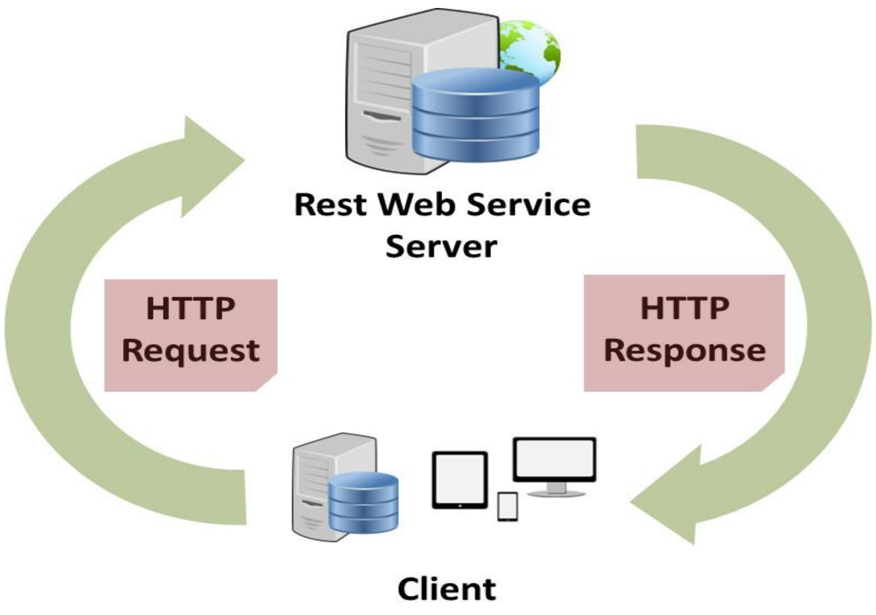
- **an introduction**

REST or RESTful API design (Representational State Transfer) is designed to take advantage of existing protocols. While REST can be used over nearly any protocol, it usually takes advantage of HTTP when used for Web APIs. This means that developers do not need to install libraries or additional software in order to take advantage of a REST API design. REST API Design was defined by Dr. Roy Fielding in his 2000 doctorate dissertation. It is notable for its incredible layer of flexibility. Since data is not tied to methods and resources, REST has the ability to handle multiple types of calls, return different data formats and even change structurally with the correct implementation of hypermedia.

This freedom and flexibility inherent in REST API design allow you to build an API that meets your needs while also meeting the needs of very diverse customers. Unlike SOAP, REST is not constrained to XML, but instead can return XML, JSON, YAML or any other format depending on what the client requests. And unlike RPC, users aren't required to know procedure names or specific parameters in a specific order.

However, there are drawbacks to REST API design. You can lose the ability to maintain state in REST, such as within sessions, and it can be more difficult for newer developers to use. It's also important to understand what makes a REST API RESTful, and why these constraints exist before building your API. After all, if you do not understand why something is designed in the manner it is, you can hinder your efforts without even realizing it.

RESTful Web Services :



Image(1)

What is Restful Web Services?

Restful Web Services is a lightweight, maintainable, and scalable service that is built on the REST architecture. Restful Web Service, expose API from your application in a secure, uniform, stateless manner to the calling client. The calling client can perform predefined operations using the Restful service. The underlying protocol for REST is HTTP. REST stands for REpresentational State Transfer.

In this REST API tutorial, you will learn-

- RESTful Key Elements
- Restful Methods
- Why Restful
- Restful Architecture

- RestFul Principles and Constraints

RESTful Key Elements

REST Web services have really come a long way since its inception. In 2002, the Web consortium had released the definition of WSDL and SOAP web services. This formed the standard of how web services are implemented.

In 2004, the web consortium also released the definition of an additional standard called RESTful. Over the past couple of years, this standard has become quite popular. And is being used by many of the popular websites around the world which include Facebook and Twitter.

REST is a way to access resources which lie in a particular environment. For example, you could have a server that could be hosting important documents or pictures or videos. All of these are an example of resources. If a client, say a web browser needs any of these resources, it has to send a request to the server to access these resources. Now REST services defines a way on how these resources can be accessed.

The key elements of a RESTful implementation are as follows:

1. **Resources** – The first key element is the resource itself. Let assume that a web application on a server has records of several employees. Let's assume the URL of the web application is **http://demo.guru99.com**. Now in order to access an employee record resource via REST services, one can issue the command **http://demo.guru99.com/employee/1** - This command tells the web server to please provide the details of the employee whose employee number is 1.
2. **Request Verbs** - These describe what you want to do with the resource. A browser issues a GET verb to instruct the endpoint it wants to get data. However, there are many other verbs available including things like POST, PUT, and DELETE. So in the case of the example **http://demo.guru99.com/employee/1** , the web browser is actually issuing a GET Verb because it wants to get the details of the employee record.

3. **Request Headers –** These are additional instructions sent with the request. These might define the type of response required or the authorization details.
4. **Request Body** - Data is sent with the request. Data is normally sent in the request when a POST request is made to the REST web services. In a POST call, the client actually tells the REST web services that it wants to add a resource to the server. Hence, the request body would have the details of the resource which is required to be added to the server.
5. **Response Body –** This is the main body of the response. So in our RESTful API example, if we were to query the web server via the request **http://demo.guru99.com/employee/1** , the web server might return an XML document with all the details of the employee in the Response Body.
6. **Response Status codes –** These codes are the general codes which are returned along with the response from the web server. An example is the code 200 which is normally returned if there is no error when returning a response to the client.

Restful Methods

The below diagram shows mostly all the verbs (POST, GET, PUT, and DELETE) and an REST API example of what they would mean.

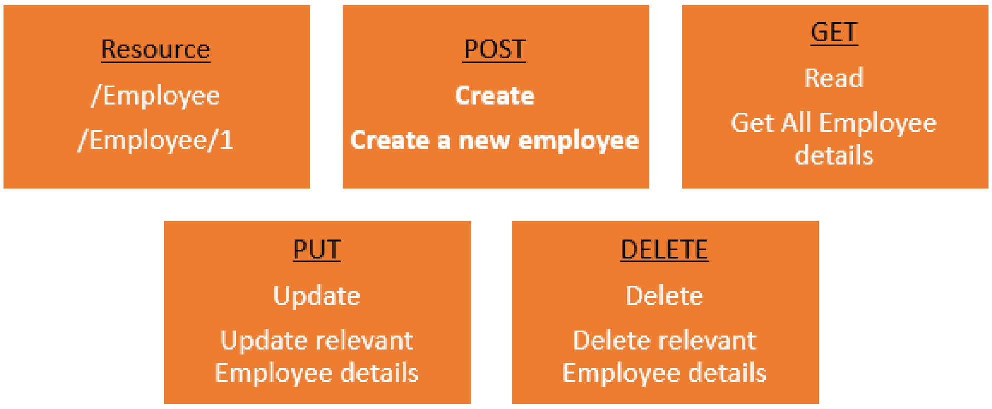
Let's assume that we have a RESTful web service is defined at the location. **http://demo.guru99.com/employee** . When the client makes any request to this web service, it can specify any of the normal HTTP verbs of GET, POST, DELETE and PUT. Below is what would happen If the respective verbs were sent by the client.

1. **POST –** This would be used to create a new employee using the RESTful web service
2. **GET** - This would be used to get a list of all employee using the RESTful web service
3. **PUT** - This would be used to update all employee using the RESTful web service
4. **DELETE** - This would be used to delete all employee using the RESTful services

Let's take a look from a perspective of just a single record. Let's say there was an employee record with the employee number of 1.

The following actions would have their respective meanings.

1. **POST** – This would not be applicable since we are fetching data of employee 1 which is already created.
2. **GET** - This would be used to get the details of the employee with Employee no as 1 using the RESTful web service
3. **PUT** - This would be used to update the details of the employee with Employee no as 1 using the RESTful web service
4. **DELETE** - This is used to delete the details of the employee with Employee no as 1



Image(2)

Why Restful

Restful mostly came into popularity due to the following reasons:

1. Heterogeneous languages and environments – This is one of the fundamental reasons which is the same as we have seen for SOAP as well.

- It enables web applications that are built on various programming languages to communicate with each other
- With the help of Restful services, these web applications can reside on different environments, some could be on Windows, and others could be on Linux.

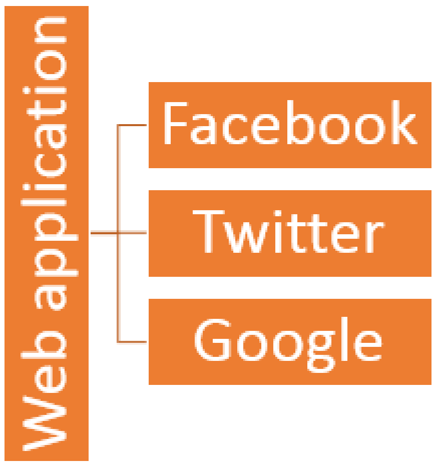
But in the end, no matter what the environment is, the end result should always be the same that they should be able to talk to each other. Restful web services offer this flexibility to applications built on various programming languages and platforms to talk to each other.

The below picture gives an example of a web application which has a requirement to talk to other applications such Facebook, Twitter, and Google.

Now if a client application had to work with sites such as Facebook, Twitter, etc. they would probably have to know what is the language Facebook, Google and Twitter are built on, and also on what platform they are built on.

Based on this, we can write the interfacing code for our web application, but this could prove to be a nightmare.

Facebook, Twitter, and Google expose their functionality in the form of Restful web services. This allows any client application to call these web services via REST.



Image(3)

2. The event of Devices – Nowadays, everything needs to work on Mobile devices, whether it be the mobile device, the notebooks, or even car systems.

   Can you imagine the amount of effort to try and code applications on these devices to talk with normal web applications? Again Restful API's can make this job simpler because as mentioned in point no 1, you really don't need to know what is the underlying layer for the device.

3. Finally is the event of the Cloud – Everything is moving to the cloud. Applications are slowly moving to cloud-based systems such as in Azure or Amazon. Azure and Amazon provide a lot of API's based on the Restful architecture. Hence, applications now need to be developed in such a way that they are made compatible with the Cloud. So since all Cloud-based architectures work on the REST principle, it makes more sense for web services to be programmed on the REST services based architecture to make the best use of Cloud-based services.

Restful Architecture

An application or architecture considered RESTful or REST-style has the following characteristics

1. State and functionality are divided into distributed resources – This means that every resource should be accessible via the normal HTTP commands of GET, POST, PUT, or DELETE. So if someone wanted to get a file from a server, they should be able to issue the GET request and get the file. If they want to put a file on the server, they should be able to either issue the POST or PUT request. And finally, if they wanted to delete a file from the server, they an issue the DELETE request.
2. The architecture is client/server, stateless, layered, and supports caching –

- Client-server is the typical architecture where the server can be the web server hosting the application, and the client can be as simple as the web browser.
- Stateless means that the state of the application is not maintained in REST.

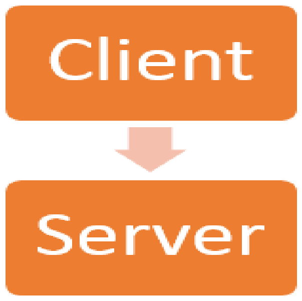
  For example, if you delete a resource from a server using the DELETE command, you cannot expect that delete information to be passed to the next request.

  In order to ensure that the resource is deleted, you would need to issue the GET request. The GET request would be used to first get all the resources on the server. After which one would need to see if the resource was actually deleted.

RESTFul Principles and Constraints

The REST architecture is based on a few characteristics which are elaborated below. Any RESTful web service has to comply with the below characteristics in order for it to be called RESTful. These characteristics are also known as design principles which need to be followed when working with RESTful based services.

1. **RESTFul Client-Server**



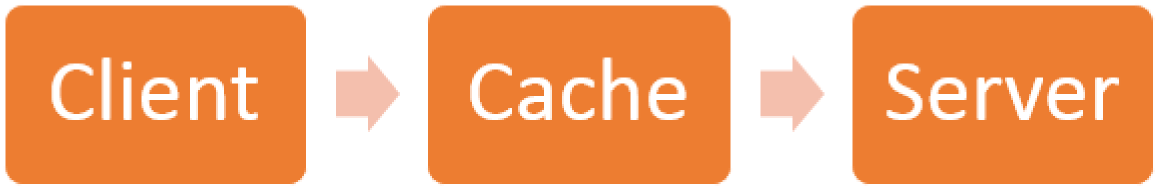
Image(4)

This is the most fundamental requirement of a REST based architecture. It means that the server will have a RESTful web service which would provide the required functionality to the client. The client send's a request to the web service on the server. The server would either reject the request or comply and provide an adequate response to the client.

2. **Stateless**

The concept of stateless means that it's up to the client to ensure that all the required information is provided to the server. This is required so that server can process the response appropriately. The server should not maintain any sort of information between requests from the client. It's a very simple independent question-answer sequence. The client asks a question, the server answers it appropriately. The client will ask another question. The server will not remember the previous question-answer scenario and will need to answer the new question independently.

3. **Cache**



Image(5)

The Cache concept is to help with the problem of stateless which was described in the last point. Since each server client request is independent in nature, sometimes the client might ask the server for the same request again. This is even though it had already asked for it in the past. This request will go to the server, and the server will give a response. This increases the traffic across the network. The cache is a concept implemented on the client to store requests which have already been sent to the server. So if the same request is given by the client, instead of going to the server, it would go to the cache and get the required information. This saves the amount of to and fro network traffic from the client to the server.

4. **Layered System**

The concept of a layered system is that any additional layer such as a middleware layer can be inserted between the client and the actual server hosting the RESTFul web service (The middleware layer is where all the business logic is created. This can be an extra service created with which the client could interact with before it makes a call to the web service.). But the introduction of this layer needs to be transparent so that it does not disturb the interaction between the client and the server.

5. **Interface/Uniform Contract**

This is the underlying technique of how RESTful web services should work. RESTful basically works on the HTTP web layer and uses the below key verbs to work with resources on the server

- POST - To create a resource on the server
- GET - To retrieve a resource from the server
- PUT - To change the state of a resource or to update it

- DELETE - To remove or delete a resource from the server

Flask HTTP methods

Flask supports the common HTTP methods, including `GET`, `POST`, `PUT`, `PATCH`, `DELETE` and working with them is extremely simple, allowing us to build URL's and endpoints which only listen for certain HTTP methods.

In this part of the "Learning Flask" series, we're going to build a simple application to demonstrate working with the 5 HTTP methods listed above, along with examples of when and how to use them.

If you're new to programming or new to working on web, getting to know the HTTP methods and understanding the basics of HTTP is extremely valuable, so I'd suggest reading up on it after you're done here.

This guide isn't designed to be an in depth tutorial on HTTP, however it should give you enough information to use them in your Flask applications and hopefully you'll learn something new!

HTTP basics

The "Hyper text transfer protocol" AKA HTTP is the world wide protocol for how we communicate and send/receive information on the web.

It works on a "Request/Response" protocol, where a client makes a request to a web server which then returns a response.

For example, when you clicked on this very URL, your browser (The client) made a request to the Pythonise web server, which internally triggered a series of events to return a response, which is the HTML you're reading now.

Like I said, this is a very basic introduction to HTTP and we're not going to go into too much detail on it in this guide. For more information, the Wikipedia page is a good place to start.

HTTP status codes

When something goes well, it's nice to get a pat on the back, likewise when something doesn't go as planned, it helps to know why so you can put it right.

Feedback is very important and it's no different on the web!

HTTP status codes are used as a feedback system, issued by the server in response to a request from the client to indicate the status of the request.

If the request went well, the server returns a status code to the client to indicate that everything is OK, likewise a bad request or error will result in a different code to tell the client that something didn't go as expected.

If you've spend any time on the web, you've most likely come across these status codes yourself.

Ever clicked on a link and got a `404 NOT FOUND`? That's the web servers way of saying "I got your message, but whatever you're looking for, I don't have it". However rather than saying that it simply returns a `404` status code (Which is much more friendly for machines to understand □)

HTTP status codes are made up of 3 digits that fall into 5 categories, with each category representing a certain class of code.

The first digit is the category and the 5 categories correspond to the following class:

- `1xx` - Informational
- `2xx` - Success
- `3xx` - Redirection
- `4xx` - Client errors
- `5xx` - Server errors

The example of the `404` status code falls under the "client error" category, where the client tried to request something that doesn't exist on the server.

The last 2 digits of the code don't fall under any kind of class or category, but are used to provide more information and context.

Again to use the `404` example, the last 2 digits refer to `NOT FOUND`, giving more context to the type of client error.

A full list of HTTP status codes can be found here at the Wiki.

HTTP methods

The method is the type of action you want the request to perform and is sent from the client to the server on every request.

There are several HTTP methods but we're only going to cover 5 in this article:

- `GET` - Used to fetch the specified resource
- `POST` - Used to create new data at the specified resource
- `PUT` - Used to create new data or replace existing data at the specified resource
- `PATCH` - Used to create new data or update/modify existing data at the specified resource
- `DELETE` - Used to delele existing data at the specified resource

Requesting a URL is an example of a `GET` request, where your browser makes a request for resources at a specified location (the URL) and the server returns some HTML. `GET` requests are "safe" as they aren't able to modify state or data on the server.

An example use case of a `POST` request would be creating a new account on a website or application, whereby the resource doesn't already exist.

Flask HTTP methods

By default, routes created with `@app.route(/example)` only listen and respond to `GET` requests and have to be instructed to listen and respond to other methods using the `methods` keyword & passing it a list of request methods.

Let's explore some common use cases for `GET` requests.

GET requests

Ubiquitously used in Flask applications, the `GET` method is used to return data at a specified resource/location.

Returning text

Possible one of the most simple routes you can write in a flask app simply returns a string:

```
@app.route("/get-text")
def get_text():
    return "some text"
```

Rendering templates

A very common use case for a `GET` request is to return some HTML:

```
from flask import render template

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

Handlinq query strings
No different from either of the 2 previous examples, with the addition of handling a query string in the URL and returning a formatted string to the client:

```
from flask import request

@app.route("/qs")
def qs():

    if request.args:
        req = request.args
        return " ".join(f"{k}: {v} " for k, v in req.items())

    return "No query"
```

Requesting `/qs?name=john&language=python` returns the string `name: john language: python`.

Fetching resources
Again, not really any different from the previous examples, just in this case fetching and returning a resource as a JSON string. We've also created a mock database called `stock`:

```
from flask import make_response, jsonify

stock = {
    "fruit": {
        "apple": 30,
        "banana": 45,
        "cherry": 1000
    }
}

@app.route("/stock")
def get_stock():

    res = make_response(jsonify(stock), 200)

    return res
```

Extending the previous example with some URL variables, ding a lookup and returning a JSON response (Note the use of the `404` if the collection or member is not found):

```
@app.route("/stock/<collection>")
```

```python
def get_collection(collection):

    """ Returns a collection from stock """

    if collection in stock:
        res = make_response(jsonify(stock[collection]), 200)
        return res

    res = res = make_response(jsonify({"error": "Not found"}), 404)

    return res


@app.route("/stock/<collection>/<member>")
def get_member(collection, member):

    """ Returns the qty of the collection member """

    if collection in stock:
        member = stock[collection].get(member)
        if member:
            res = make_response(jsonify(member), 200)
            return res

        res = make_response(jsonify({"error": "Not found"}), 404)
        return res

    res = res = make_response(jsonify({"error": "Not found"}), 404)
    return res
```
In summary, use `GET` requests when you just need to return resources to the client and NOT make any changes to the state/data of your application.

GET and POST

In many cases such as rendering forms, you'll need a route to handle more than just `GET` requests.

Making any request other than `GET` to a route without the `methods` argument and a list of methods will result in a `405 METHOD NOT ALLOWED` HTTP status code, as methods must be declared for the route to respond to.

Adding multiple request methods to a route is done with the following:

```python
@app.route("/example", methods=["METHOD_A", "METHOD_B"])  # GET POST PUT PATCH DELETE
etc..
```
The route will now listen and respond to both methods provided which means we have to put in some control flow to handle each type of request.

Fortunately this is made easy using the `request` object, in particular the `request.method` attribute which returns the method for the current request.

This verbose and silly example illustrates the logic, assuming we want to listen for `GET` and `POST` requests:

```
@app.route("/log-in", methods=["GET", "POST"])
def log_in():

    if request.method == "POST":
        # Attempt the login & do something else
    elif request.method == "GET":
        return render_template("log_in.html")
```

As Flask routes default to `GET` requests, we can remove the `elif` statement and let Flask return the template:

```
@app.route("/log-in", methods=["GET", "POST"])
def log_in():

    if request.method == "POST":
        # Only if the request method is POST
        # attempt the login & do something else

    # Otherwise default to this
    return render_template("log_in.html")
```

A working example can be seen below where we're rendering a template which a user can then use to add a new collection to our `stock` database:

```
@app.route("/add-collection", methods=["GET", "POST"])
def add_collection():

    """
    Renders a template if request method is GET.
    Creates a collection if request method is POST
    and if collection doesn't exist
    """

    if request.method == "POST":

        req = request.form

        collection = req.get("collection")
        member = req.get("member")
        qty = req.get("qty")

        if collection in stock:
            message = "Collection already exists"
            return render_template("add_collection.html", stock=stock, message=message)
        stock[collection] = {member: qty}
        message = "Collection created"

        return render_template("add_collection.html", stock=stock, message=message)

    return render_template("add_collection.html", stock=stock)
```

POST requests

Flask routes listen for `GET` requests by default, so we must implicitly instruct them to listen for anything other than `GET`.

In the example below, we've passed `methods=["POST"]` to the `@app.route()` decorator, meaning this route will ONLY respond to `POST` requests:

```
@app.route("/stock/<collection>", methods=["POST"])
def create_collection(collection):

    """ Creates a new collection if it doesn't exist """

    req = request.get_json()

    if collection in stock:
        res = make_response(jsonify({"error": "Collection already exists"}), 400)
        return res

    stock.update({collection: req})

    res = make_response(jsonify({"message": "Collection created"}), 201)
    return res
```

We're checking to see if the `collection` variable (passed in via the URL) is in our `stock` database and if not, create it, otherwise return a `400 BAD REQUEST` to indicate the resource already exists.

`POST` requests should be used to create NEW resources (New users, devices, posts, articles, datasets etc..)

Again, if you tried to access this resource from the browser, you'd be greeted with a `405 METHOD NOT ALLOWED` status as we've not provided `GET` as one of the available request methods to listen for.

PUT requests

`PUT` requests are similar to `POST` requests but serve a very different purpose.

As we explained earlier, `PUT` should be used to create or replace a resource, meaning if it doesn't exist - create it, however if it does exist, replace it.

Check out the example below:

```
@app.route("/stock/<collection>", methods=["PUT"])
def put_collection(collection):

    """ Replaces or creates a collection """

    req = request.get_json()

    if collection in stock:
        stock[collection] = req
        res = make_response(jsonify({"message": "Collection replaced"}), 200)
        return res

    stock[collection] = req
    res = make_response(jsonify({"message": "Collection created"}), 201)
    return res
```

Since `PUT` requests shouldn't care about the existing data or resource, we've decided to go ahead and create the resource with no regard for any existing data. The only dirrerence in the logic is the HTTP status code.

If the collection DIDN'T exist, we're returning a `201 CREATED` status. Whereas if the collection DID exist, we're returning a `200 OK` to indicate that the collection has been replaced.

It's probably bad design to replace an entire collection using the `PUT` request, but for demonstrational purposes it'll do. A better solution would be to use `PUT` to replace or create a value for one of the members in the collection.

PATCH requests

We're using a `PATCH` request to update OR create a resource in our `stock` database:

```
@app.route("/stock/<collection>", methods=["PATCH"])
def patch_collection(collection):

    """ Updates or creates a collection """

    req = request.get_json()

    if collection in stock:
        for k, v in req.items():
            stock[collection][k] = v

        res = make_response(jsonify({"message": "Collection updated"}), 200)
        return res

    stock[collection] = req

    res = make_response(jsonify({"message": "Collection created"}), 201)
    return res
```

Rather than replace the collection entirely, we're iterating over the keys and values in the request body and updating the values in the collection, only creating new members if they don't exist.

And just like in `PUT`, we're returning a `200` if the collection was updated and a `201` if the collection was created.

Let's cover the last request method in this article, `DELETE`.

Delete requests

Just like it says on the tin, `DELETE` requests should be used to delete a resource.

As per the rest of the examples, we provide `methods=["DELETE"]` in the `@app.route()` decorator, meaning this route will only listen for that specific method.

In the 2 examples below, you'll see we're deleting a collection and deleting individual members from a collection with 2 separate routes:

```
@app.route("/stock/<collection>", methods=["DELETE"])
def delete_collection(collection):

    """ If the collection exists, delete it """

    if collection in stock:
        del stock[collection]
        res = make_response(jsonify({}), 204)
        return res

    res = make_response(jsonify({"error": "Collection not found"}), 404)
    return res

@app.route("/stock/<collection>/<member>", methods=["DELETE"])
def delete_member(collection, member):

    """ If the collection exists and the member exists, delete it """

    if collection in stock:
        if member in stock[collection]:
            del stock[collection][member]
            res = make_response(jsonify({}), 204)
            return res

        res = make_response(jsonify({"error": "Member not found"}), 404)
        return res

    res = make_response(jsonify({"error": "Collection not found"}), 404)
    return res
```

On deletion, we're returning a `204 NO CONTENT` status code to indicate a successful transaction and we have no content to return.

If the resource isn't found, I/e the collection or member doesn't exist, we're returning a `404 NOT FOUND`.

Wrapping up

The code snippets shown here were designed to demonstrate how to work with some of the common request methods in Flask, rather than be examples of how to write a REST API, so take the examples with a pinch of salt.

restfulapi.net is a great place to learn more about API design and using the various request methods, along with the 2 Wikipedia links below.

Just save it as `hello.py` or something similar. Make sure to not call your application `flask.py` because this would conflict with Flask itself.

To run the application you can either use the **flask** command or python's `-m` switch with Flask. Before you can do that you need to tell your terminal the application to work with by exporting the `FLASK_APP` environment variable:

What to do if the Server does not Start

In case the **python -m flask** fails or **flask** does not exist, there are multiple reasons this might be the case. First of all you need to look at the error message.

Old Version of Flask

Versions of Flask older than 0.11 used to have different ways to start the application. In short, the **flask** command did not exist, and neither did **python -m flask**. In that case you have two options: either upgrade to newer Flask versions or have a look at the Development Server docs to see the alternative method for running a server.

Invalid Import Name

The `FLASK_APP` environment variable is the name of the module to import at **flask run**. In case that module is incorrectly named you will get an import error upon start (or if debug is enabled when you navigate to the application). It will tell you what it tried to import and why it failed.

The most common reason is a typo or because you did not actually create an `app` object.

Debug Mode

(Want to just log errors and stack traces? See Application Errors)

The **flask** script is nice to start a local development server, but you would have to restart it manually after each change to your code. That is not very nice and Flask can do better. If you enable debug support the server will reload itself on code changes, and it will also provide you with a helpful debugger if things go wrong.

To enable all development features (including debug mode) you can export the `FLASK_ENV` environment variable and set it to `development` before running the server:

```
$ export FLASK_ENV=development
$ flask run
```

(On Windows you need to use `set` instead of `export`.)

This does the following things:

1. it activates the debugger
2. it activates the automatic reloader
3. it enables the debug mode on the Flask application.

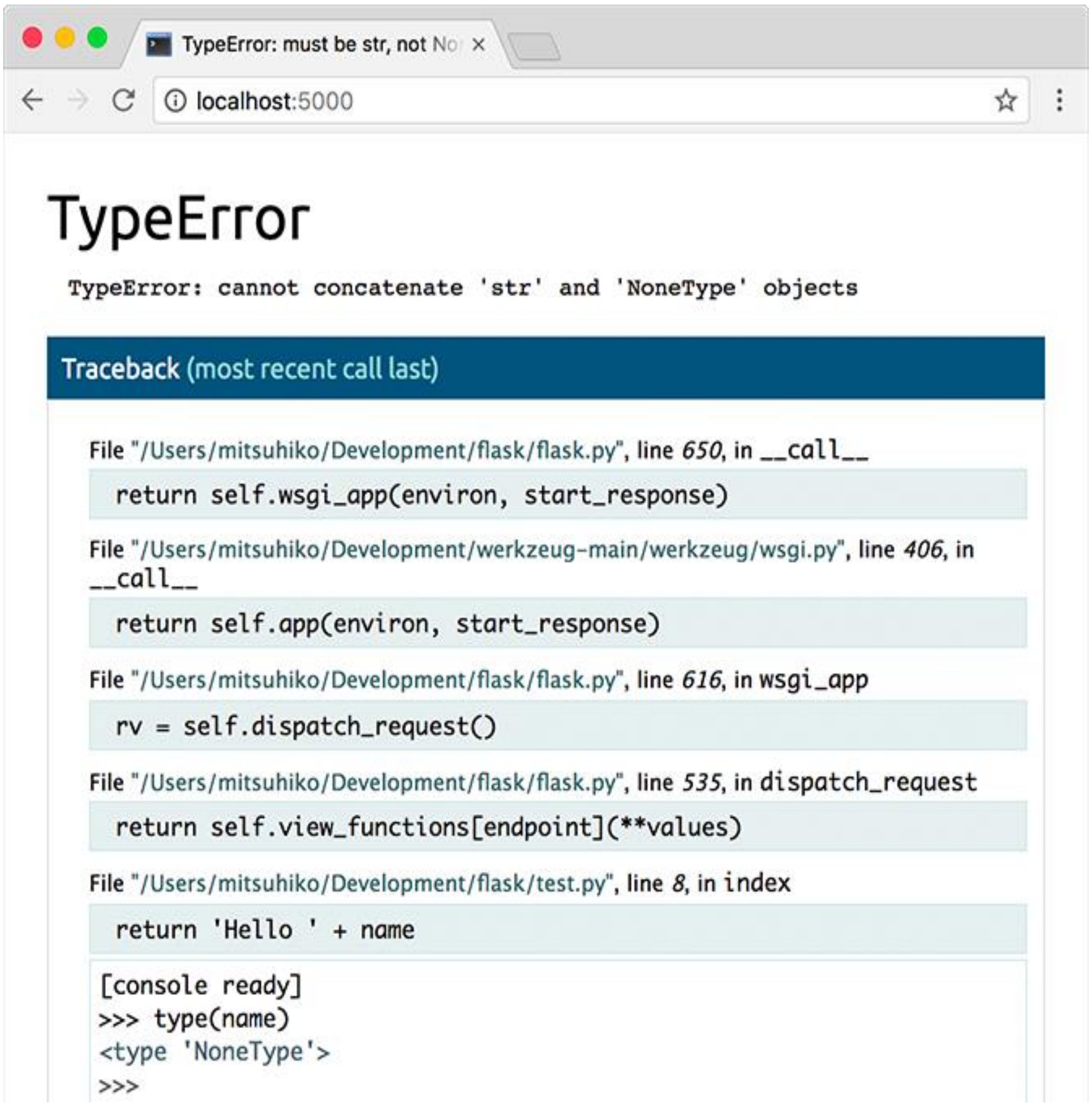
You can also control debug mode separately from the environment by exporting `FLASK_DEBUG=1`.

There are more parameters that are explained in the Development Server docs.

Attention

Even though the interactive debugger does not work in forking environments (which makes it nearly impossible to use on production servers), it still allows the execution of arbitrary code. This makes it a major security risk and therefore it **must never be used on production machines**.

Screenshot of the debugger in action:


TypeError: must be str, not No ×

localhost:5000

TypeError

```
TypeError: cannot concatenate 'str' and 'NoneType' objects
```

Traceback (most recent call last)

```
File "/Users/mitsuhiko/Development/flask/flask.py", line 650, in __call__
  return self.wsgi_app(environ, start_response)
File "/Users/mitsuhiko/Development/werkzeug-main/werkzeug/wsgi.py", line 406, in
__call__
  return self.app(environ, start_response)
File "/Users/mitsuhiko/Development/flask/flask.py", line 616, in wsgi_app
  rv = self.dispatch_request()
File "/Users/mitsuhiko/Development/flask/flask.py", line 535, in dispatch_request
  return self.view_functions[endpoint](**values)
File "/Users/mitsuhiko/Development/flask/test.py", line 8, in index
  return 'Hello ' + name
[console ready]
>>> type(name)
<type 'NoneType'>
>>>
```

Image(6)

Methods

Web applications use different HTTP methods when accessing URLs. You should familiarize yourself with the HTTP methods as you work with Flask. By default, a route only answers to `GET` requests. You can use the `methods` argument of the **`route()`** decorator to handle different HTTP methods.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

If `GET` is present, Flask automatically adds support for the `HEAD` method and handles `HEAD` requests according to the HTTP RFC. Likewise, `OPTIONS` is automatically implemented for you.

Static Files

Dynamic web applications also need static files. That's usually where the CSS and JavaScript files are coming from. Ideally your web server is configured to serve them for you, but during development Flask can do that as well. Just create a folder called `static` in your package or next to your module and it will be available at `/static` on the application.

To generate URLs for static files, use the special `'static'` endpoint name:

```
url_for('static', filename='style.css')
```

The file has to be stored on the filesystem as `static/style.css`.

Rendering Templates

Generating HTML from within Python is not fun, and actually pretty cumbersome because you have to do the HTML escaping on your own to keep the application secure. Because of that Flask configures the Jinja2 template engine for you automatically.

To render a template you can use the **`render_template()`** method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Here's a simple example of how to render a template:

```
from flask import render_template

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

Flask will look for templates in the `templates` folder. So if your application is a module, this folder is next to that module, if it's a package it's actually inside your package:

Case 1: a module:

```
/application.py
/templates
    /hello.html
```

Case 2: a package:

```
/application
    /__init__.py
    /templates
        /hello.html
```

For templates you can use the full power of Jinja2 templates. Head over to the official Jinja2 Template Documentation for more information.

Here is an example template:

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

Inside templates you also have access to the **`request`**, **`session`** and **`g`** [1] objects as well as the **`get_flashed_messages()`** function.

Templates are especially useful if inheritance is used. If you want to know how that works, head over to the Template Inheritance pattern documentation. Basically template inheritance makes it possible to keep certain elements on each page (like header, navigation and footer).

Automatic escaping is enabled, so if `name` contains HTML it will be escaped automatically. If you can trust a variable and you know that it will be safe HTML (for example because it came from a module that converts wiki markup to HTML) you can mark it as safe by using the **`Markup`** class or by using the `|safe` filter in the template. Head over to the Jinja 2 documentation for more examples.

Here is a basic introduction to how the **`Markup`** class works:

```
>>> from markupsafe import Markup
>>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
Markup(u'<strong>Hello &lt;blink&gt;hacker&lt;/blink&gt;!</strong>')
>>> Markup.escape('<blink>hacker</blink>')
Markup(u'&lt;blink&gt;hacker&lt;/blink&gt;')
>>> Markup('<em>Marked up</em> &raquo; HTML').striptags()
u'Marked up \xbb HTML'
```

Changelog

1

Unsure what that **g** object is? It's something in which you can store information for your own needs, check the documentation of that object (**g**) and the Using SQLite 3 with Flask for more information.

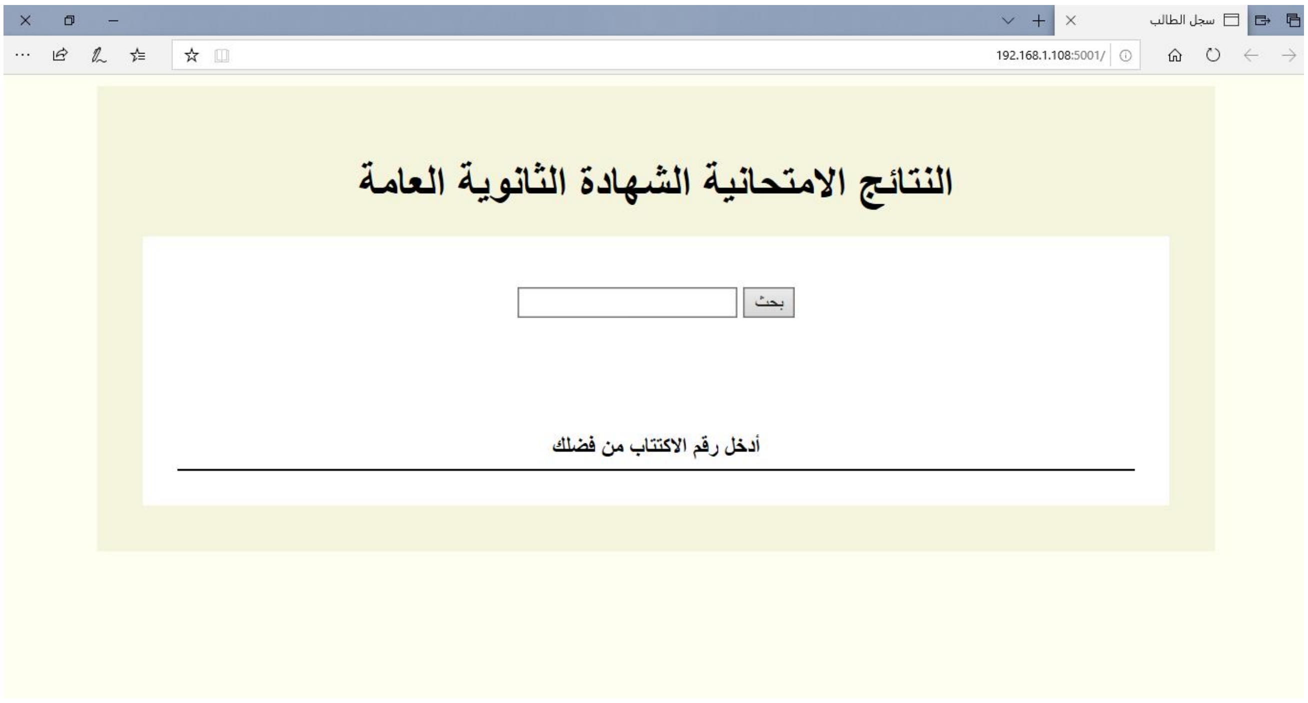
الجزء العملي(الكود):

في البداية قمت بإنشاء صفحات ال HTML حيث لدي صفحتين :

١. index.html :

```
<!DOCTYPE html>
<html lang="en">

<head>
    {% block head %}
    <meta charset="UTF-8">
    <meta http-equiv="expires" content="0">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="apple-mobile-web-app-status-bar-style" content="default">
    <meta http-equiv="Content-Security-Policy" content="default-src * 'self'
'unsafe-inline' 'unsafe-eval' data: gap:">
    <title>{% block title %}{% endblock %} سجل الطالب </title>
    <link rel='stylesheet' href="{{ url_for('static',
filename='css/style.css')}}">
    {% endblock %}
</head>
<main>
    <h1>النتائج الامتحانية الشهادة الثانوية العامة</h1>
    <figure>
        <div>
            <table class="wochenprogramm">
                <colgroup>
                    <col>
                </colgroup>
                <thead>
                    <tr>
                        <br>
                    </tr>
                    <tr>
                        <form method="POST" action="/">
                            <label>
                                <input type="number" name="student_id">
                            </label>
                            <input type="submit" value="بحث">
                        </form>
                    </tr>
                </thead>
                <tr>
                    <td><br><strong>{{message}}</strong><br></td>
                </tr>
            </table>
        </div>
    </figure>
</main>
</body>

</html>
```

حيث الكود السابق يعبر عن الصفحة التالية :



النتائج الامتحانية الشهادة الثانوية العامة

أدخل رقم الاكتتاب من فضلك

ولنشرح هذه الصفحة في البداية هذه الصفحة ذات العنوان (سجل الطالب) حيث هذه الصفحة اكتسبت هذا المظهر من ملف style.css الذي سأستعرضه لاحقا

والكتلة الأولى فيها :

تدعى meta data هي ليست للمستخدم بل تعمل في الخلفية من أجل المتصفح أو اعدادات عامة للصفحة او حتى من أجل محرك البحث بحالتنا مثلا حددنا زمن المحتوى و تناسب العرض مع اجهزة أخرى مثل الموبايل

ومن ثم لدينا العنصر <main> :

يحتوي على العنوان الرئيسي بتنسيق h1

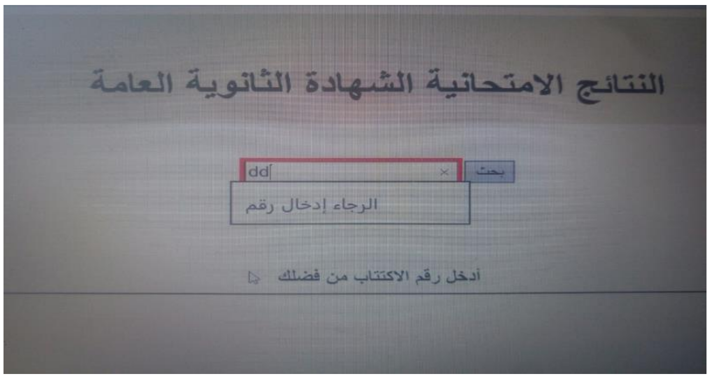
ومن ثم لدينا العنصر figure (باللون الغامق في الصفحة)الذي وضعناه لنخصص مكان من الصفحة و نضع فيه باقي العناصر

ومن ثم العنصر div (باللون الأبيض) الذي هو عبارة عن حاوية عامة للمحتوى

ثم لدينا عنصر الجدول table الذي يحتوي على :

فورم لكي يكون الطالب قادرا على إدخال رقم اكتتابه حيث هذا الرقم سوف يسند إلى المتحول

Student_id ,   ويتم استخدامه في الكود الأساسي ،والدخل سيكون حصرا رقم لأن النوع number



النتائج الامتحانية الشهادة الثانوية العامة

أدخل رقم الاكتتاب من فضلك

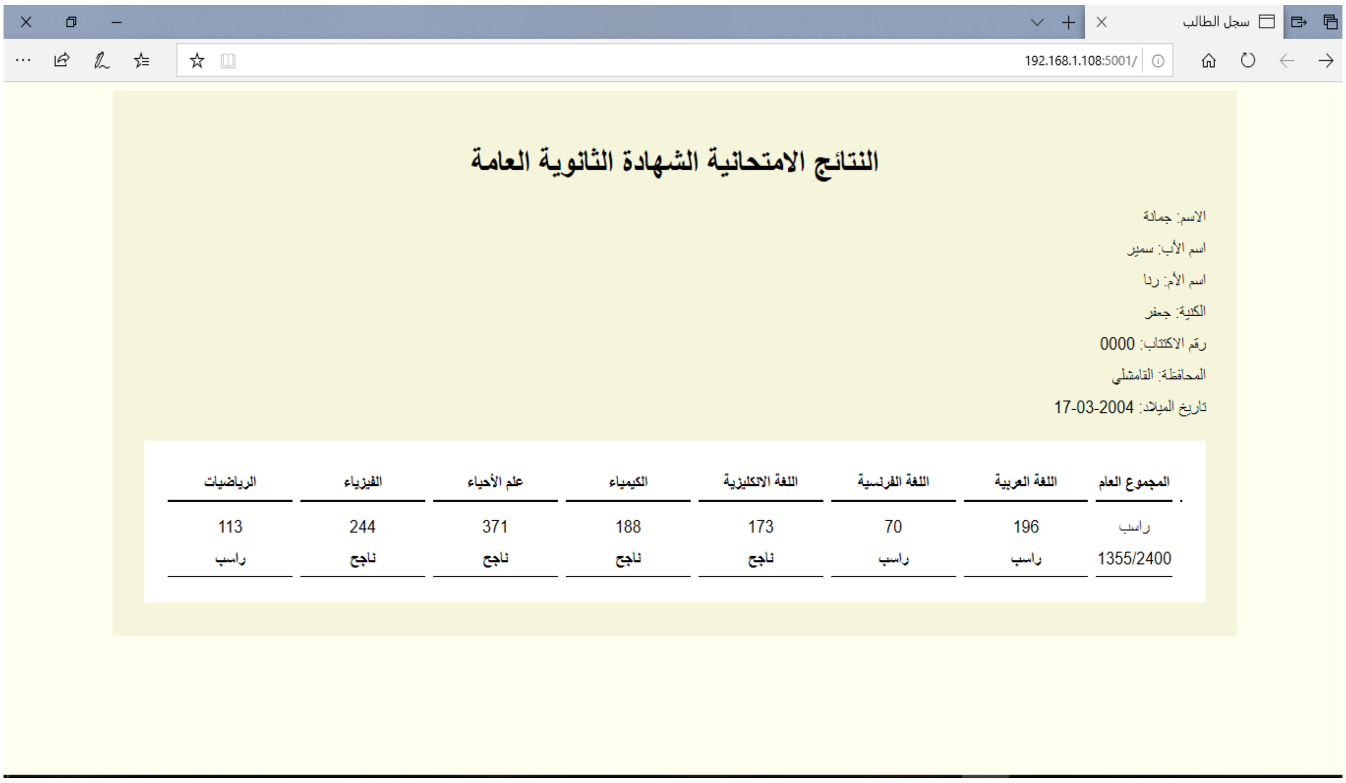
و أيضا زر للبحث عن الرقم

و من ثم عنصر جدول يحتوي على المتحول {{message}} الذي سيتم إسناد قيمته من الكود الأساسي بالاعتماد على الرقم المدخل حيث في الكود الأساسي سيتم إسناد الجملة المناسبة لها

٢. Portal.html :

```
<!DOCTYPE html>
<html lang="en">

<head>
    {% block head %}
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="apple-mobile-web-app-status-bar-style" content="default">
    <meta http-equiv="Content-Security-Policy" content="default-src * 'self'
'unsafe-inline' 'unsafe-eval' data: gap:">
    <title>{% block title %}{% endblock %} سجل الطالب </title>
    <link rel='stylesheet' href="{{ url_for('static',
filename='css/style.css')}}">
    {% endblock %}
</head>
<main>
    <h1>النتائج الامتحانية الشهادة الثانوية العامة</h1>
    <p> الاسم: {{student_firstname}}
        <br>
        اسم الأب: {{father}}
        <br>
        اسم الأم: {{mother}}
        <br>
        الكنية: {{student_lastname}}
        <br>
        رقم الاكتتاب: {{std_id}}
        <br>
        المحافظة: {{city}}
        <br/>
        تاريخ الميلاد: {{date_of_birth}}
        <br/>
    </p>
    <figure>
        <div>
            <table>
                <colgroup>
                    <col>
                    <col>
                    <col>
                    <col>
                    <col>
                    <col>
                    <col>
                </colgroup>
                <thead>
                    <tr>
                        <th scope="col">الرياضيات</th>
                        <th scope="col">الفيزياء</th>
                        <th scope="col">الأحياء علم</th>
                        <th scope="col">الكيمياء</th>
                        <th scope="col">الانكليزية اللغة</th>
                        <th scope="col">الفرنسية اللغة</th>
                        <th scope="col">العربية اللغة</th>
                        <th scope="col">العام المجموع</th>
                        <th scope="col"></th>
                    </tr>
                </thead>
                <tr>

<td>{{actual_marks['math']}}<br><strong>{{pass_fail['math']}}</strong><br></td>

<td>{{actual_marks['physics']}}<br><strong>{{pass_fail['physics']}}</strong><br>
</td>

<td>{{actual_marks['biology']}}<br><strong>{{pass_fail['biology']}}</strong><br>
</td>

<td>{{actual_marks['chemistry']}}<br><strong>{{pass_fail['chemistry']}}</strong>
<br></td>

<td>{{actual_marks['english']}}<br><strong>{{pass_fail['english']}}</strong><br>
</td>

<td>{{actual_marks['french']}}<br><strong>{{pass_fail['french']}}</strong><br></
td>

<td>{{actual_marks['arabic']}}<br><strong>{{pass_fail['arabic']}}</strong><br></
td>

<td>{{pass_fail['total']}}<br><strong></strong>{{actual_marks['total']}}/2400<br
></td>
                </tr>
            </table>
        </div>
    </figure>
</main>

</html>
```

الصفحة السابقة ستظهر بالشكل :



النتائج الامتحانية الشهادة الثانوية العامة

الاسم: جمانة

اسم الأب: سمير

اسم الأم: رنا

الكنية: جعفر

رقم الاكتتاب: 0000

المحافظة: القامشلي

تاريخ الميلاد: 17-03-2004

المجموع العام	اللغة العربية	اللغة الفرنسية	اللغة الانكليزية	الكيمياء	علم الأحياء	الفيزياء	الرياضيات
راسب	196	70	173	188	371	244	113
1355/2400	راسب	راسب	ناجح	ناجح	ناجح	ناجح	راسب

حيث في بداية صفحة ال html و في عنصر ال head لدينا نفس الكتلة في الصفحة الأولى

ولدينا فقرة تحتوي على المتحولات:

{{date_of_birth}} ، {{city}} ,{{std_id}}،{{student_lastname}}،{{mother}}،{{father}}،{{student_firstname}}

التي سيتم إسناد قيمها في الكود الأساسي

ثم لدينا جدول يحتوي في السطر الأول على خلايا لأسماء المواد والمجموع العام ثم في السطر الثاني لدينا العلامة الحقيقة للطالب حيث نحصل عليها من الكود الأساسي و نتيجته في حال كان راسب أم ناجخ خيث التابع passfail معرف في الكود الأساسي و سنعرف عمله لاحقا

و فيما يلي سأوضح صفحة ال css المستخدمة في الصفحتين السابقتين :

```
@import url(https://fonts.googleapis.com/earlyaccess/droidarabickufi.css);
.droid-arabic-kufi{font-family: 'Droid Arabic Kufi', serif;}

body {
    font-family: sans-serif;
    font-size: 130%;
    background-color: ivory;
    text-align: center;
```

```
  vertical-align: middle;
  line-height: 30px;
}

h1 {
  font-size: 2rem;
  font-weight: bold;
}

p {
  text-align: right;
  text-orientation: right;
  direction: rtl;
}

main {
  margin: 0 auto;
  width: 80%;
  background-color: beige;
  padding: 2rem;
  font-size: 1rem;
}

figure {
  margin: 0;
}
figure figcaption {
  color: gray;
  font-size: 1.275rem;
  margin-top: 1rem;
}
figure div {
  padding: 0;
  overflow-x: auto;
  background-color: white;
}

table {
  /*border-collapse: seperate;*/
  border-spacing: 0.5rem;
  margin: 1rem;
}

th {
  border-bottom: 2px solid black;
  text-align: center;
}

td {
  border-bottom: 1px solid black;
  min-width: 4%;
  width: calc(100% / 7);
  white-space: no-wrap;
  vertical-align: bottom;
}

.elective1,
.elective2 {
  background: navajowhite;
}
```

```
  @media only screen and (max-width: 900px) {
    body {
      background: yellow;
    }

    .elective1,
  .elective2,
  th:nth-of-type(5),
  td:nth-of-type(5),
  th:nth-of-type(7),
  td:nth-of-type(7) {
      display: none;
    }
  }
```

و الأن لدينا الكود الرئيسي بلغة البايثون الذي يحتوي على التوابع المطلوبة و على مخدم ال flask :

```
from flask import Flask, render_template, request
from random import seed, randrange, choice
import datetime

app = Flask(__name__)


def check_passed(actual_marks):
    thresholds = {'math': 300, 'physics': 200, 'chemistry': 100,
                  'biology': 200, 'arabic': 200, 'english': 100, 'french': 100,
'total': 1500}
    pass_fail = {k: 'ناجح' if k in thresholds and actual_marks[k] >=
thresholds[k] else 'راسب'
                 for k in actual_marks}
    return pass_fail


def gen_dummy_info(random_set_indicator):
    seed(random_set_indicator)
    start_date = datetime.date(2001, 1, 1)
    end_date = datetime.date(2005, 2, 1)
    time_between_dates = end_date - start_date
    days_between_dates = time_between_dates.days

    female_names = ['جمانة', 'ريما', 'رنا', 'حلا', 'فيروز', 'سعاد', 'وداد',
'عفاف', 'مايا', 'ندى', 'سارة']
    male_names = ['علي', 'أحمد', 'يونس', 'حسام', 'جعفر', 'عبد الكريم', 'ابراهيم',
'سامي', 'سمير', 'مهند']
    city_names = ['اللاذقية', 'طرطوس', 'دمشق', 'حلب', 'حماه', 'حمص',
                  'ادلب', 'دير الزور', 'الرقة', 'القامشلي', 'ريف دمشق',
'السويداء', 'درعا']
    info = [
        choice(female_names + male_names),
        choice(male_names),
        choice(female_names),
        choice(male_names),
        choice(city_names),
        start_date + datetime.timedelta(days=randrange(days_between_dates))]
    return info
```

```
@app.route('/', methods=["GET", "POST"])
def index():
    if request.form:
        student_number = request.form['student_id']
        if not student_number or student_number is None:
            return render_template('./index.html', message='!! هذا الاكتتاب رقم
صحيح غير')
        student_info = gen_dummy_info(student_number)
        seed(student_number)
        actual_marks = {'math': randrange(100, 600),
                        'physics': randrange(100, 600),
                        'chemistry': randrange(100, 400),
                        'biology': randrange(100, 400),
                        'arabic': randrange(50, 200),
                        'english': randrange(50, 200),
                        'french': randrange(50, 200)}

        actual_marks['total'] = sum(actual_marks.values())

        return render_template('./portal.html',
                               std_id=student_number,
                               student_firstname=student_info[0],
                               student_lastname=student_info[1],
                               mother=student_info[2],
                               father=student_info[3],
                               city=student_info[4],
                               date_of_birth=student_info[5],
                               actual_marks=actual_marks,
                               pass_fail=check_passed(actual_marks)
                               )
    return render_template('./index.html', message='أدخل رقم الاكتتاب من فضلك')


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5001, debug=True)
```

في البداية و قبل تنفيذ الكود سنقوم بتثبيت المكتبات اللازمة للتطبيق باستخدام تعليمة :

Pip install –r requirements.txt

في ال terminal حيث الملف reguirements.txt موجود في ملفات المشروع

في بداية الكود قمنا باستيراد المكتبات اللازمة والأدوات اللازمة

حيث ال Flask هو سيرفر http يقوم بأخذ request من المستخدم و يعطيه respone بالمعلومات التي يحتاجها

حيث seed تابع لتوليد أرقام عشوائية بالاعتماد على الرقم الذي يسند إليه و يكون عبارة عن بذرة لهذا التابع

وتابع ال randrange لاختيار قيمة عشوائية من قيم الرقم المدخل إليه

وتابع choiceلاختيار قيمة معينة من القائمة المسندة إليه

في البداية عرفنا تطبيق من السيرفر flask و أسميناه app

و من ثم عرفنا تابع :check_passed(actual_marks) يأخذ قيمة العلامات التي تولد عشوائيا عند تشغيل السيرفر باستخدام الأداة randrange حيث تكون على شكل قاموس actual_marks يحوي اسم المادة وعلامة الطالب و يحتوي هذا التابع على عتبة الرسوب كقاموس thresholds يحوي اسم المادة وعلامة العتبة يقارن بين القاموسين و يعطينا النتيجو كناجح أم راسب في القاموس pass_fail

و من ثم عرفنا تابع :gen_dummy_info(random_set_indicator) يأخذ الرقم student_id الذي يدخله المستخدم و يقوم بتوليد قيم عشوائية بالاعتماد على هذا الرقم و باستخدام seed حيث (كل رقم يدخله المستخدم بشكل عشوائي سيعطينا نفس البيانات في كل مرة نعيد إدخاله

حيث بداخله سنحدد مجال للمواليد لاختيار عشوائيا منها ومن ثم نعرف ثلاثة قوائم تحتوي على أسماء ذكزر و أسما إناث و المدن

ليتم تعريف قائمة تتقوم بالاختيار عشوائيا من القوائم السابقة لاسم الطالب كذكر و الأم كانثى و الكنية كاسم مذكر وباستخدام التابع choice

ثم قمنا بتشغيل السيرفر الذي سيعمل على المخدم المحلي :

وعند تشغيل السيرفر سنقوم باستخدام الأداتين post و get حيث post تستخدم لارسال معلومات صفحة ال index.html إلى السيرفر (أي رقم الاكتتاب الذي يدخله المستخدم) و get هي أداة يستخدمها render_template لاحضار البيانات و تعبئتها في صفحة ال portal.html و إظهارها

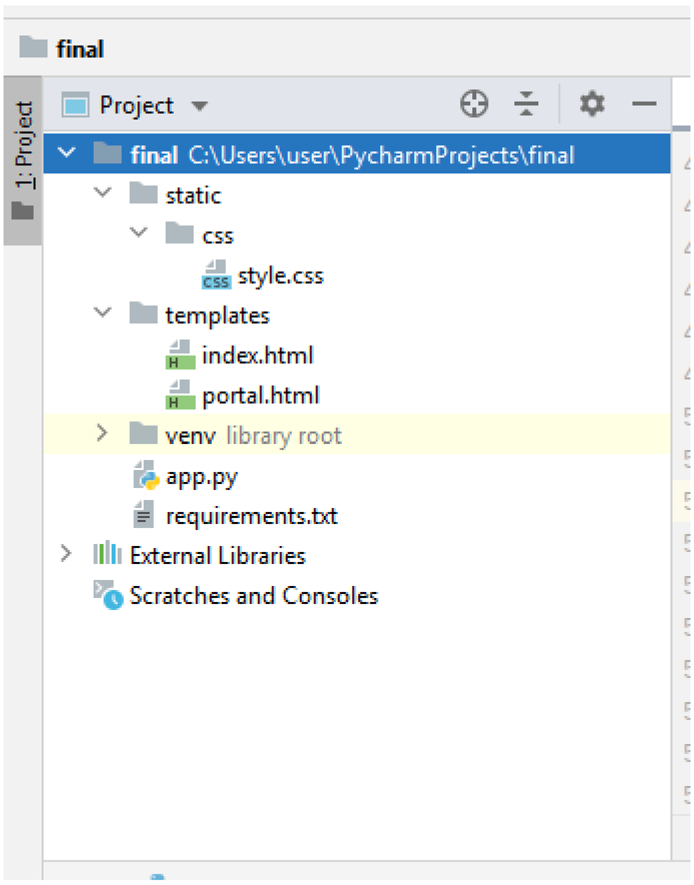
حيث الشرط :if request.form أي إن المستخدم يقوم بإدخال رقم حاليا و المخدم سوف يعمل حيث باستخدام تعليمة ['request.form['student_id سيحضر المخدم الرقم الذي أدخله المستخدم و يسنده إلى متحول student_number و لدينا شرط في حال ضغط المستخدم زر البحث بدون أن يدخل أي رقم ستظهر عبارة رقم الاكتتاب هذا غير صحيح

ومن ثم نعرف القاموس actual_marks الذي سيأخذ علامات عشوائية لكل مادة و سنستخدم هذا القاموس عند استعاء التابع :check_passed(actual_marks) كما ذكرنا سابقا

ومن في الجزء المتبقي سنقوم باستخدام ال render_template لتعبئة صفحة portal.html بالقيم المطلوبة و إظهارها للمستخدم في ال Localhost=0.0.0.0

ملاحظة هامة:

تم وضع أكواد ال html في مجلد templates و كود ال css في مجلد ststic والمتطلبات و الكود في المجلد الأساسي كما هو موضح في الصورة التالية :



```
final
Project
final  C:\Users\user\PycharmProjects\final
    static
        css
            style.css
    templates
        index.html
        portal.html
    venv  library root
    app.py
    requirements.txt
External Libraries
Scratches and Consoles
```

Reference:

1. https://www.guru99.com/restful-web-services.html
2. https://www.mulesoft.com/resources/api/what-is-rest-api-design
3. https://pythonise.com/series/learning-flask/flask-http-methods
4. https://flask.palletsprojects.com/en/1.1.x/quickstart/