SERVLIB/PYHP DOCUMENTATION

Contents

henriquestiaux@gmail.com

OPEN SOURCE SOFTWARE

Servlib

Getting started

Installing servlib on your computer

In order to have servlib installed on your computer, you will need to do a few thing:

- Create a folder named “Servlib”
- Download server.py and content-types.json from my github and place them in the “Servlib” folder
- Create a new folder named “sessions” in the “Servlib” folder
- Done, Servlib is now installed

Creating your first multi-pages project

They are few easy project that I’ve made to help you understanding how my framework work, download the on my github and look at them as your follow this tutorial. Keep in mind that you can almost do every thing with python so the limit of what you can do with your first project is only your imagination

In order to make your first web app, you will need to do a few things:

- In your “Servlib” folder create another folder named with your project’s name. This folder is the root of your web app (Ex: project1)
- In your project folder create a new python file named with your first page’s name (Ex: page1.py)
- Edit your first page using your favourite IDE
- In your python file, write “import server” in order to make your page able to use the framework I’ve created
- In the same file, create a function named main and taking one parameter (name it data for better understanding)
- This function will have to return a byte type object which is an HTTP response that had been encoded
- In order to make it easy, I’ve made a class named Response that will help you making your own HTTP responses. This function have two attributes, content and headers. Content is the content of you response, basically your HTML page or your json and headers are the headers of your response. It’s quite hard to understand for beginners so I will keep it simple : headers are the metadata of your response, they are helping it to understand your response.
- Now you’ve defined what will be the content and the headers of your response, make your response a byte like object using the Response.encode(“utf-8”) method
- Send your response by returning the byte like object
- You’ve created your first project’s first page.

Starting the server

You've created your first project, now it's time to see if it works.

In order to apply changing made on your .py web page, you will need to restart the server so I've made this step really fast.

Command line start

The easiest way to start the server is using your command prompt.

- Open your command prompt
- Match the path of your command prompt with the path of the "server.py" file using the "cd" command
- Type "py server.py -h" to get help
- Type py server.py -start "" "2555" "project1" "log.txt" -listen "3" to start serving the website that have for root the "project1" folder on port 2555

Python file start

If you are like me and you learn by making mistakes, you will restart the server a lot of times. The fastest and most configurable way to start a server is by creating your own server starting python file. You can add some things like custom code that will modify the content of the data transmitted to your .py web pages. It can be useful if you use a proxy like Cloudflare and you want to automatically set data["IP"] to the value of the "X-Forwarded-For" header.

- Create a python file and name it as you wish (EX: Project1_launcher.py)
- Open this file using your favourite IDE
- Import server like when you created your 1st python web page
- The server library have a class named Server. His __init__ function takes some arguments: the address, the port, the root of the website that you want to serve, and the log file. So you will write something like that : s = server.Server("",80,"project1","log.txt").
- Your server can also take others attributes:
  - homepage which define the homepage of your website. For example : if your homepage is "/index.py", and you type "localhost" in your browser's address bar, you will get the same result as if you typed "localhost/index.py"
  - subdomain_roots which makes you able to create subdomains ([WARINING]:Don't really work great with localhost and IP acces, this feature had only been done for websites that are served using a domain name). For example, if my main root is "project1" and my subdomain roots are {"hey_there":"project2"} if I type "example.com/page1.py", I'll get what's inside "project1/page1.py" and if I type "hey_there.example.com/page1.py", I'll get what's inside "project2/page1.py"
  - custom_thread which allows you to change what's inside the data processed by your .py web pages. For example:

```
from json import load
from urllib.request import urlopen
def location(data):
    data.update({"geolocation":load(urlopen("http://ipinfo.io/{}/json".format(data["IP"])
))})
return data
s.custom_thread = location
```

Will automatically geolocate the IP connecting to the website

- To start your server, use the listen function of your recently created and customised server. It takes one argument which is the number of request that can be in a state of waiting before getting processed. To keep it simple, use a number like 3 or 5.
- Start your program
- Your server is now serving your website

Using the config form

Depreciated and dangerous, don't use it unless you know what you are doing

The server is now running. Use the address bar of your browser and type localhost:2555/page1.py if you set the port to 2555 and named your first web page "page1.py"

Making web pages dynamic using the data gathered by the server

Your main functions are taking one argument but we've not use it already and it's time to change it. What this argument contains is all the headers of the request that had been send to the server which contains a lot of information like the device that requested the server, which browser where used, the cookies and even more. This argument also contains other information including the IP from where came the request, some path that will help you with local storage file (as sqlite3 DBs for example). This argument is a dictionary object, you can access any of this information using this syntax : data["IP"] to get the IP of the device that sent the request to the server. See "Dynamic_project_example" for more information

Using sessions and cookies to communicate through different web pages

You've already made at least one static and one dynamic project. You understand what's the difference between a root and a page and how to access and interact with them. Now it's time to store data either on your server (sessions) or on the client's computer (cookies). Keep in mind that your client can remove, modify and create cookies has he wants. Cookies are therefore only safe when they modifications does not interfere with your website's security. On the other side, sessions are stored on the server and therefore can only be created and modified by your website's web pages. Pay attention that the link between the client and his session are maintained by a cookie that can therefore be removed, modified and add but don't worry, the session will be automatically removed if there's any suspicion of spoofing (stealing someone else's session) and I'll show you how to make a program that handle the session removing quite easily. See "cookie_session_example" to see how it works.

Hashing sensible data

In case of a website that require account creation, you will need to store passwords. When you are storing such a sensible data, you have to secure them using encryption. Hashing something is like putting this thing into a chest that is locked forever. But on this chest, there is a unique number. If you hash two same things the unique number on their respective chest will be the same. Two different things can't give the same unique number. Finally, if you hash to thins that are just slightly different, you will get completely different unique numbers. So, basically, how can you integrate the hashing of password in the process of encoding them ?

What your actual program does:

Registering:

Password -> data base

Logging in:

Password -> program <- get password for this username <- data base

|

Are they the same ?

Incorrect <- No Yes -> Correct

You have only one simple step to do on registering and logging in:

Registering:

Password -> server.salthash(password, server.random_salt) -> data base

Logging in:

Password -> program <- get password for this username <- data base

|

server.salthash_verify(password_from_data_base, password)

Incorrect <- return False return True -> correct

Refer to the "login_example" for further explaination