236369

Winter 2021-2022

Managing Data on the World-Wide Web

Assignment 2 – Web Server

Assignment is 25% Takef.

Submission date: 7.12.2021 23:55 PM

Introduction

In this assignment, you will implement a generic web server that supports a simplified version of dynamic pages (templates) and basic HTTP authentication. The architecture is based on the async model that was taught in class.

Server Architecture

Your server should be an asynchronous server that runs on one single-threaded server. The server should use non-blocking I\O calls and manage connections asynchronously.

In addition, the logic itself should be asynchronous, too (e.g., reading a file is also a long I\O operation that should be non-blocking).

The server should attempt to recover from whatever failures it may encounter (e.g., bad requests, database failures) and return an informative error response.

HTTP Requests

Your web server should handle **concurrent** HTTP/1.1 requests.

```
GET relative-URL HTTP/1.1\r\n
header 1\r\n
…
header K\r\n
\r\n
*body*
```

The relative-URL corresponds to a path on the file system of the server. In this case, the path of the requested resource is considered relative to the base directory (where your main.py file is). For instance, if the base directory is ***e:/236369/HW/hw2/serverBase,*** then the request ***GET /a/b/c.html*** corresponds to the file ***e:/236369/HW/hw2/serverBase/a/b/c.html***

Upon receiving a request for a resource with a path, which may correspond to a file in the file system, the server should return the resource and provide the Content-Type header with the appropriate MIME correspondent to the file's extension. (use the provided file - mime.json)
The request should be handled as follows:

1. If the request is for an existing and readable *dynamic page* (.dp) you should handle it according to the *dynamic pages* section.
2. If the request is for the raw contents of an existing and readable file. In this case reply with 200 response that contains the contents of the file.

Make sure that the users database and the configuration file are never returned to the client.

API For Managing Users

You will implement an API for managing users that is inspired by REST.
Only the admin (a special user whose credentials are specified in the config file) is allowed to use this API. Users other than the admin should not be able to create/delete users. Identifying the users' credentials and gaining authorization is discussed below in the Authentication section.

Creating Users

To create a user, the admin will send a POST request to the **/users** path. The body

```
POST /users HTTP/1.1\r\n
header 1\r\n
…
Content-Type: application/x-www-form-urlencoded
Authorization: Basic …
…
header K\r\n

username=user1&password=1234
```

will contain a **username** and a **password** (one user per post request). The body will be in the application/x-www-form-urlencoded Format.

Deleting Users

To delete a user, the admin will send a DELETE request to the /users/<username> path, where *<username>* is the username of the user that the admin would like to delete (one per post request), and has the following form:

```
DELETE /users/<username> HTTP/1.1\r\n
header 1\r\n
Authorization: Basic …
…
header K\r\n
```

Users' Database

The users' credentials are stored in the database – **users.db** that is managed by sqlite3. You should access the database for acquiring and updating the users' credentials. Example:

```
conn = sqlite3.connect('users.db')
```

The schema of the database contains one table – Users(username,password).

Responses

Your server will respond with legal HTTP/1.1 responses. A HTTP/1.1 response has the following form:

```
HTTP/1.1 Status-Code Reason-Phrase\r\n
header 1\r\n
.
header K\r\n
```

The reason-phrases are specified at: https://www.w3.org/Protocols/

example:

```
GET /images/norway.gif HTTP/1.1
Host: ibm222.cs.technion.ac.il:8001
Accept: image/gif, image/x-xbitmap, image/jpeg, images/pjpeg*/* ,
Accept-Language: en-us
Accept-Encoding: gpzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Authorization: Basic YWRtaW46MTIzNDU3\r\n
Connection: close
```

```
HTTP/1.1 200 OK
Date: Sun, 08 Sep 2019 18:02:37 GMT
Content-Length: 3231
Connection: close
Content-Type: image/gif
… the image's bytes …
```

If the server is unable to satisfy a request, it issues an error response. For example, if the server doesn't find the requested file, then it may issue the following response:

```
HTTP/1.1 404 Not Found
Date: Sun, 08 Sep 2019 18:02:37 GMT
Content-Length: 177
Connection: close
Content-Type: text/html

<html>
<head>
  <title>404 Not Found</title>
</head>
<body>
  <h1>Not Found</h1>

  <p>The requested URL /images/norway.gif was not found on this server</p>
  <hr />
</body>
</html>
```

Configuration Files

You are given with two configuration files:

1. A python properties file, called *config.py* where following variables are defined:
    a. **port:** the port number that the server listens to.
    b. **timeout:** the timeout for a connection
    c. **admin:** A dictionary containing the username and password of the admin.

```
port = 8000
timeout = 10
admin = {'username': 'admin', 'password': '123456'}
```

2. A JSON file named ***mime.json***. The file includes a mapping between file extensions to mime types. For example, the first element says that the file ***a.abs*** has the mime type ***audio/x-mpeg.***

You can assume these files exist on the working directory and are valid. To parse the JSON, you can use any API\package you like.

Server Implementation

You are encouraged to use the packages and implementation ideas we saw in class as the base for your server (asyncio, async\await). Note that what we saw in class is a very basic example, and the assignment is much more complex, so think where you and how you implement each part.

Dynamic Pages

The dynamic pages that your server need to support are a restricted (made-up) version of templates. These files have the "**.dp**" extension (dynamic pages) .

Files with the extension **"dp"** are HTML-like files embedded with Python code, specifically Python expressions. The Python expressions are surrounded by {% %}. These expressions can be any Python expression that returns a value (you will probably want to return strings). Upon a request for such files, your parser will evaluate the Python expressions and build a valid HTML file.
An example for such file will be:

```
<!DOCTYPE html>
<html lang="en">
<body>
<p> {% 'Hi ' + user['username'] if user['authenticated'] else "Please authenticate so we'll know your name" %} </p>
{% "<p>We know that your name is {username}</p>".format(username=user['username']) if user['authenticated'] else '' %}
</body>
</html>
```

The templates will have access to following context variables:

1. **user**: a dictionary containing the following entries:
   a. **'authenticated'**: A Boolean stating whether the user is authenticated (see the section about Authentication).
   b. **'username'**: For authenticated users this will contain the username. For non-authenticated requests this will be None.

   Example: {'authenticated': True, 'username': 'user1'}

2. **params**: a dictionary containing the values of the query parameters (https://en.wikipedia.org/wiki/Query_string) sent with the GET request.
   Example: {'color': 'blue', 'number': '42'}

In general, we recommend the compilation method that was discussed in class: a class that "compiles" the template into an HTML file by evaluating the Python expressions while providing them the context. Remember that for invoking the rendering function you need the *context variables*.

Authentication

The server will provide a protection using the basic HTTP authentication scheme. To implement this part, read the Basic Authentication Scheme section of the HTTP/1.1 RFC. Given a GET request for a dynamic page, the server should confirm that the client is a registered user and to pass that knowledge to the rendering class of the dynamic pages.

Given a POST or a DELETE request, the server should confirm that the client is an admin with a username and a password equal to the admin's credentials stored in the config file. Otherwise, the server should respond with 401 with the suitable *WWW-Authenticate* header.

General Requirements and Assumptions

HTTP Communication

The following should hold for all the responses your server generates, **including the error responses.** First, the response must conform to the HTTP/1.1 standard. Second, the response should include, as HTTP headers, the following information:

- The *time* of the response generation (i.e. when the server responds). This *time* should be in one of the standard HTTP date formats.
- The *content-length* of the response.
- The *mime-type* of the response content (only if the configuration file includes the suitable *mime-type)*.
- A statement that the server closes the connection upon termination.

The headers should include other relevant information if needed.

Error Management

When your server responds with an error, **you are expected to return the most appropriate status code from the list below**. The response body should be a valid HTML that provides informative description of the problem that caused the error.

Status Codes: 200 (OK), 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden), 404 (Not Found), 409 (Conflict), 500 (Internal Server Error), 501 (Not Implemented).

Tips and Technical Details

Getting Started

You are given the following files:

1. **config.py** – as written in the configuration section.
2. **mime.json** – as written in the configuration section.
3. **environment.yml** – a Conda environment file, with all the required packages.
4. **users.db** – The users' database.
5. **example.dp** – an example of a dynamic page.

To get started, first install the environment using Conda:

```
conda env create -f environment.yml
conda activate [env_name]
```

Tips

- For the base of you server, you can use the *asyncio server* we saw in class.
- The aiohttp-server and aiohttp-client documentation could be very helpful.
- Remember – you control where "context-switch" happens, use the async\await mechanism wisely to get better performance than a single threaded blocking server.
- The HTTP protocol is well defined – if you encounter some unmentioned erroneous scenario or uncertainty look up the HTTP/1.1 RFC.
- Don't hesitate to ask questions – but first read the relevant section, already asked question on the piazza and F.A.Q. on the webcourse.

Submission

You should submit a **zip** named *"hw2_id1_id2.zip"*.

Your submission should include the following files:

1. **hw2.py** – We will run the server with the command:

   'python hw2.py'

2. Any other files you used for implementing your server.

Good Luck!