Homework 5

CS 3640: Introduction to Networks and Their Applications
Due: 11:59 pm, Dec 8th (Fri), 2017.

Submission instructions: This homework can be done in groups of up to 2 people. Each student is responsible for contacting other students and form a group. Individual submissions are acceptable but not encouraged.

Submit all necessary files on ICON. **Only one group member should submit the work.** However, both the group members are required to submit separate group evaluations. Your group evaluation must clearly state the contribution and percentage of work done by each group member. Your individual score may be affected by your and your partner's evaluations.

You are not permitted to share or copy any written materials or code from anyone other than your group partner. Please mention the source and write sufficient comments to show your understanding if you use any code from the Internet.

We recommend Python 3 for this homework. However, C/C++ or Java are also acceptable.

Part 0

Find instructions on how to develop a simple web server in Python here. Note that the code given in this webpage is written for Python 2. A modified version of this code for Python 3 is posted on ICON. Download the code from ICON and run the server. While the server is running, open a browser and visit the URL http://127.0.0.1:8080/move?a=20. Study the instructions on the webpage carefully to understand the code.

Part 1

You are to develop a web server in this assignment. Your server must be started with two arguments specifying the IP address of the interface on which the server will bind to and the port number on which the server will run. The server should handle HTTP requests such as (assuming the server is running on port 8000):

- http://127.0.0.1:8000/index.html

- http://127.0.0.1:8000/cars/ford.html

The server will read html static contents from a local directory called **static**. Accordingly, the above URLs will be resolved to the following files on the server:

URL	file location
http://127.0.0.1:8000/index.html	static/index.html
http://127.0.0.1:8000/cars/ford.html	static/cars/ford.html

Create two simple webpages named index.html and ford.html and place them in the directories specified above.

The server should implement the following logic:

- **Connection Setup:** Create a socket to wait for incoming connections. The server should be able to listen to 5 pending connections. Upon receiving a connection, you should accept it and process the incoming http request as discussed in the next step.
- **Multiple Connections:** You should support multiple connections using *select*.
- **HTTP Request:** Your server should only handle HTTP GET requests. A complete description of how an HTTP request is formatted may be found in RFC 2616. For the purposes of this assignment it is safe to assume the following restricted format:

  GET SP Request-URI SP HTTP/1.1 CRLF
  (HTTPKey:HTTPValue CRLF)*
  CRLF

  In the above, the following notation is used: the text in blue will appear as it is in the header packet. SP stands for space and CRLF for carriage return line feed. The HTTPKey:HTTPValue are header fields that may appear in the request. The ()* indicates that there could be zero or more header fields. Check the output of the server in part 0 to see a sample request. Your server should read the header fields and make sure the header is formatted correctly but it can safely ignore them (i.e., you do not need to take any actions on them). The server must make sure that the incoming request is consistent with the above format.

- **Responding to an HTTP request:** There are three possible outcomes when handling an HTTP request:

  1. Bad Request: The HTTP request is invalid (i.e. it does not copy with the above format). In this case your server should respond with a Bad Request reply.

2. Not Found: If the HTTP request is valid, your program should map the RequestURI to the file location indicated by the URL. After performing this mapping, the server must check if the request file exists. If it does not, the server should respond with a Not Found error.
3. No errors: If the file exists, the server should read the contents of the file and generate an HTTP reply.

- **Generating an HTTP reply:** If the request is valid and the file exists, read the corresponding file and send the content to the client. You need to generate an appropriate HTTP header as a part of the response. For our case the minimal HTTP header is:

```
HTTP/1.1 SP 200 SP OK CRLF
Content-Type: text/html
CRLF
CRLF
```

After sending the header, you can send the content of the file.

Part 3

The server described in part 2 can serve static HTML files. In this part, you will extend the capabilities of your static web server to provide basic capabilities similar to those that web frameworks such as django or node.js provide. To be more specific, you need to implement the followings:

- handle content supplied by the user via forms whose action is set to be HTTP POST
- generate dynamic HTML content based on the values provided by the user

The dynamic server should work as follows:

1. A user may access an HTML form that you have previously saved in the **static** folder. Similar to part 2, the **static** folder includes all the content that is static. An example of a simple form is shown below:

```
<html>
<form action="/r1.html" method="post">
    First name: <input type="text" name="fname"><br>
    Last name: <input type="text" name="lname"><br>
    <input type="radio" name="gender" value="male" checked> Male<br>
    <input type="radio" name="gender" value="female"> Female<br>
```

```
    <input type="radio" name="gender" value="other"> Other<br>
    <input type="submit" value="Submit">
</form>
</html>
```

A detailed tutorial for writing forms is at `https://www.w3schools.com/html/html_forms.asp`. The form has several aspects worth highlighting:

- The form will use HTTP POST to send the information provided by the user to your server. The form specifies to use HTTP POST by setting the value of the attribute **method** to be *post*.
- The action attribute indicates where to send the form data when the form is submitted.
- The form has a number of fields for which you can set what is the name and the value. Depending on the input type other attributes may also be used.
- One of the possible ways to encode the frame is to use `application/x-www-form-urlencoded`. In this case, the names and values will be saved as a long string in the body of the HTTP post method. The pairs of values are separated by **&** and the names and values by **=**.

As an example, if a user fills out the form such that the first name is John, the last name is Doe, and the gender is male; then the browser will generate the following request:

```
POST /r1.html HTTP/1.1
Host: localhost:8001
Connection: keep-alive
Content-Length: 32
Cache-Control: max-age=0
Origin: http://localhost:8001
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_4)
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Referer: http://localhost:8001/f1.html
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8
Cookie: Webstorm-ea36d52b=7cf702c4-f920-40c7-acfd-a25b1d77b36c

fname=John&lname=Doe&gender=male
```

Note that, in order to handle HTTP GET in part 1, we just had to read the header. In this case, we have to also read the body of the HTTP POST request. The length of the HTTP POST is provided in the **Content-Length** of the HTTP POST. Also, as part of the validation process, you must make sure that the **Content-Type** is set to **application/x-www-form-urlencoded**.

2. Your dynamic web server should process the incoming HTTP POST requests. This requires that you validate the request and parse the name-value pairs in the body of the HTTP request

into a dictionary. You must handle concurrent clients similar to Part 1. Additionally, you must handle a mix of HTTP GET and POST requests.

3. Once you constructed the dictionary you need to respond to the request. Use a simple templating language to generate dynamic HTML content that you will use in the reply. The language allows a name that is included in curly brackets to be replaced by its value. This is best seen as an example. Consider the template shown below.

   ```
   <html>
   First name <b>{{fname}}</b><br/>
   Last name <b>{{lname}}</b><br/>
   Gender <b>{{gender}}</b><br/>
   </html>
   ```

   If the template is invoked using the dictionary {fname : John, lname : Doe, gender : male } will be rendered as:

   ```
   <html>
   First name <b>John</b><br/>
   Last name <b>Doe</b><br/>
   Gender <b>male</b><br/>
   </html>
   ```

   If a name is missing from the dictionary, you will render its value to be the string **??UNKNOWN??**

4. Now that you generated the HTML page, include the page in your HTTP reply.

The following resources may be useful in solving the homework:

- HTTP RFC (`https://tools.ietf.org/html/rfc2616`).
- Developer tools of any browser (I use Chrome)
- Postman extension for Chrome `https://www.getpostman.com/`

Submit **one** source file (covering all the functionalities of both the Parts 2 and 3) containing the code of your web server and all the HTML files on ICON.