Topic 2.2: MCQ Server Project – Part 2

The goal of this project is to set up a basic Python web server that serves multiple-choice questions to the user, scores those questions, and keeps the user's score.

This part of the project will add user sessions using browser cookies.

Browser cookies are sent in the HTTP Header when a client makes an HTTP request.

Goals for Part 2

Initially, we want to get the server code running that serves:

- Add code that will read the session cookie sent from the client, and generate a session identifier if the request does not have a session cookie.
- Add code that attaches the session cookie to the response sent to the client – either the same session identifier as the client already has, or a new session identifier if it's a new session.

HTTP Request Message

An HTTP request message is a text message that a client (often a web browser) sends to a server asking it to perform an action such as fetching a page or submitting form data.

Below is an example of an HTTP POST request from a HTML form that contains `username` and `password` fields in the request body, intended for login to the server.

```
 1 POST /login HTTP/1.1
 2 Accept: */*
 3 Accept-Encoding: gzip, deflate, br, zstd
 4 Accept-Language: en-US,en;q=0.9
 5 Connection: keep-alive
 6 Content-Length: 29
 7 Content-Type: application/x-www-form-urlencoded
 8 Cookie: sessionId=459e7226-5ea1-47be-b817-c3fb8fc87611
 9 Origin: https://localhost
10 Priority: u=3, i
11 Referer: https://localhost/
12 Sec-Fetch-Dest: empty
13 Sec-Fetch-Mode: cors
14 Sec-Fetch-Site: same-origin
15 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
16 AppleWebKit/605.1.15 (KHTML, like Gecko) Version/26.5 Safari/605.1.15
17
18 username=alice&password=apple
```

Code Block: HTTP Request Message

Each HTTP request message consists of:

- a **request line** (line 1, above): the HTTP method (GET, POST, etc.), the path (/login, /info, …), and the HTTP version (HTTP/1.1), all on one line
- **headers** (line 2-16, above): zero or more lines in `Name: value` format
- a **blank line** (line 17, above)

- the **body** (optional; line 18, above): the actual data being sent, used mainly with POST requests; its presence and size are usually described by the `Content-Length` header.

Lines 2 through 16 contain HTTP headers. Each line contains a `key:value` pair. HTTP headers are defined by the HTTP specification to consist only of visible, non-delimiter ASCII characters. It isn't necessary to memorize every header, but it's helpful to understand their general format and the purpose of a few common ones.

Line 6 contains the `Content-Length` HTTP header. If there is no body, this header may be omitted, but if present, the value is the number of bytes contained in the body of the message. The value of `Content-Length` for this HTTP request is 29. Verify for yourself that the body of the message has 29 ASCII characters.

Line 7 contains the `Content-Type` HTTP header, which may be values such as `text/html`, `image/jpg`, or `application/javascript`. The header tells the server how to interpret the body of the message. In this example, the content type is `application/x-www-form-urlencoded`, which tells the server that the body is in the format of submitted form data. This format encodes form data as `key=value` pairs joined by `&`, with special characters replaced by `%` followed by their two-digit hexadecimal ASCII code. In our message, the login data can be seen in the body, encoded as:

```
username=alice&password=apple
```

Sometimes, if the form data is processed using client-side JavaScript, the data will be sent with `Content-Type` set to `application/json`, the `Content-Length` set to a value 39, and the body containing login data in JSON format:

```
{"username":"alice","password":"apple"}
```

Line 8 contains an HTTP Cookie header. The example header contains a single cookie. The header format is a list of `key=value` pairs, each of which is separated by a semi-colon (`;`). If there are multiple cookies to send, the format looks like this:

```
Cookie: name1=value1; name2=value2; name3=value3
```

Maintaining a Session: the Life Cycle of a Session Cookie

A session cookie is the simplest form of cookie: the server generates a random session identifier (usually a number), sends it to the client it in a `Set-Cookie` HTTP header, and keeps the real data (username, game score, etc.) in memory on the server, keyed by the session identifier. The browser automatically sends the cookie back in the `Cookie` HTTP header on every request, allowing the server to look up the session and "remember" who the client is. This is the original, most straightforward use of cookies: a plain identifier that ties a user to server-side state.

When the client logs out and/or the server wishes to end the user session, the server removes the session identifier and data associated with it from its list of sessions, and tells the client that the cookie is no longer valid by setting the `Max-Age` to zero (`0`), as shown in this example HTTP response header:

```
Set-Cookie: sessionId=459e7226-5ea1-47be-b817-c3fb8fc87611; Max-Age=0
```

The `Max-Age` is the number of seconds the cookie is valid, and the browser should automatically delete the cookie after it has expired.

When the client next makes a request to the server, there will be no session cookie, so the server treats the client as a new client and will send a new session cookie to start a new session.

Universally Unique Identifier (UUID)

A UUID (Universally Unique Identifier) is a 128-bit identifier designed to be globally unique across space and time. It is commonly represented in a "8-4-4-4-12" format, which consists of five groups of hexadecimal digits separated by hyphens. This format is both human-readable and structured for specific purposes.

For UUID version 4, the contents of the standard UUID string:

```
xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx
```

Can be broken down into:

- **Version** – the `4` (the 13th hex digit, bits 48–51) are always `0100` in binary, for version 4 UUID.
- **Variant** – the `y` (the 17th hex digit, bits 64–65) is binary 10xx, so in this case variant `2`.
- Everything else is random bits.

UUID version 4 contains the largest number of random bits, 122, so is the most suitable for a random, unguessable number. Other versions of UUID contain additional information such as the a date and time, along with random bits, so are more suitable for other purposes.

New Functions for Handling Sessions: `session.py`

Add the new file `session.py` to the project folder. Examine the code for the file. This code has three functions: `get_session`, `set_session_cookie`, and `clear_session_cookie`.

Function `get_session`

The function `get_session` is intended to be called as soon as the client request is received. The first few lines of the code are given here:

```
1     # Look for a session cookie in the request
2     raw_cookie = handler.headers.get('Cookie', '')
3     cookie = SimpleCookie(raw_cookie)
4     sid = cookie.get(config.SESSION_COOKIE_NAME)
```

Code Block: Find Session Cookie

Line `2` finds the `Cookie` field the HTTP request header.

The function `SimpleCookie` in line `3` returns an object that has a `get` method to retrieve the value associated with key of the cookie key-value pairs. Thus, after line `4`, the variable `sid` will contain the session identifier retrieved from the `Cookie` field of the HTTP request header.

These next lines execute when a session identifier, `sid`, was found in the HTTP request headers. The server will continue to use the same session identifier with this client.

```
1     if sid:
2         sid = sid.value
3         # If the session ID is known, use the existing session
4         if sid in SESSIONS:
5             handler._sid = sid
6             handler._new_sid = False
7             handler.session = SESSIONS[sid]
8             handler._cookie_sent = True   # the cookie sent earlier
9             return handler.session
```

Code Block: Use Existing Session Identifier

If no session identifier was found in the HTTP request headers, the server generates a universally unique identifier (UUID) in line 2 of the code below, calling the method `uuid.uuid4()`. The server must remember it has an active session and store data associated with that session. This server does this in line 3 below, by adding a dictionary using the session identifier, `sid`, as the key to the list of sessions, `SESSIONS`.

When a client makes a request that includes a session identifier, the server verifies it is a true session by checking if the session identifier, `sid`, exists in the list of sessions. (The code is above, line 4 of *Code Block: Use Existing Session Identifier*).

```
1     # No valid session found - create a brand new one
2     sid = str(uuid.uuid4())               # random, unique ID
3     SESSIONS[sid] = {}                    # start with an empty dict
4     handler._sid = sid
5     handler._new_sid = True               # we need to send a cookie
6     handler.session = SESSIONS[sid]
7     handler._cookie_sent = False
8     return handler.session
```

Code Block: Generate a New Session Identifier

Upgrading the server

`session.py`

Ensure the file `session.py` is copied into the project root directory.

`mcq-server.py`

The `do_GET` method needs to be updated to call the `get_session` function at the beginning of processing the HTTP request, before we make further decisions about what to send back to the client.

```
1  def do_GET(self):
2      # Log every request so we can see what's happening
3      print(f"--> Received GET {self.path}")
4      # Attach session data to the handler.
5      # If a cookie exists it will use the given session identifier;
6      # otherwise a new session identifier is created.
7      session.get_session(self)
8      # Strip the query string for route matching
9      parsed = urlparse(self.path)
10     route_path = parsed.path
```

Code Block: Updates to `mcq-server.py` method `do_GET`

In order to call the `get_session` function, the code must be imported into `mcq-server.py`, so add this line near the top of the file, along with the other `import` commands:

```
import session
```

`static_handler.py`

Add line 9, below, so the server will send the new session identifier cookie to the response sent back to the client in the `Set-Cookie` HTTP response header.

```
 1  def serve(handler):
 2      # some code not shown...
 3      # HTTP response code 200 means success
 4      handler.send_response(200)
 5      # Tell the client the type and length of the file
 6      handler.send_header('Content-Type', mime_type)
 7      handler.send_header('Content-Length', len(content))
 8      # Add session cookie to the outgoing HTTP response header
 9      session.set_session_cookie(handler)
10      # Send the HTTP headers and file to the client
11      handler.end_headers()
12      handler.wfile.write(content)
```

Code Block: Updates to `mcq-server.py` method `do_GET`

In order to call the `set_session_cookie` function, the code must be imported into `static_handler.py`, so add this line near the top of the file, along with the other `import` commands:

```
import session
```