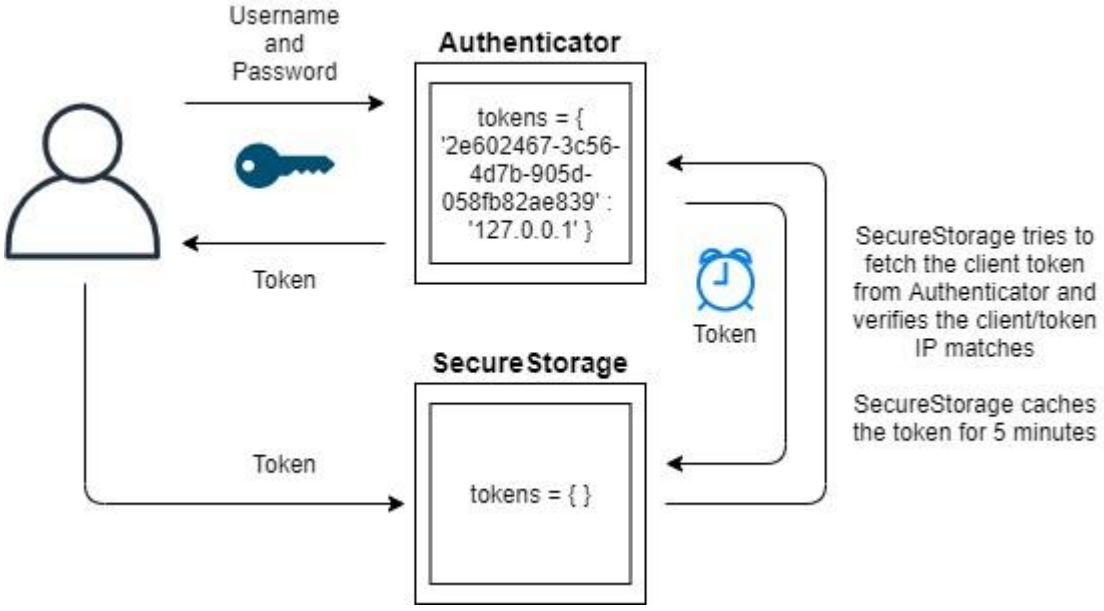
COSC 331
Lab 4: A Simple Token IP Authentication System

In this lab, we'll be building a simple authentication service that uses a token and the client's IP address to authenticate a user and permit them access to a resource. This lab will use two services: a **stateful** service that handles the user login and produces a token associated with the client's IP, and a **stateless** service that accepts a token from the user and, if valid, returns a resource to the authenticated user. This lab will introduce the idea of working with stateful and stateless services, as well as an introduction to **inter-service communication and caching.**

How it Works



This lab uses two distinct microservices: the **Authenticator**, and the **SecureStorage**. We'll discuss the Authenticator first.

The authenticator catches **a POST request** from the client on port 8080 which contains the user login details: a username and a password. If these match a correct username and password (stored in a dictionary in the Authenticator), the Authenticator will return a token to the client. This token is a random, 36-character UUID which is stored, along with the client's IP address, in the **tokens** dictionary in the Authenticator.

A **GET** request to the Authenticator performs one of two tasks: it can either **log the user out** with the **/logout** path, which will remove all tokens that match the client's IP address in the **tokens** dictionary. The other task a **GET** request can do is by fetching a token page, using a path like this: **http://127.0.0.1:8080/cdfd2219-015f-440c-b5bc-256562b0d8d9**. If the specified token (UUID) exists, the GET request will return **the associated client IP**.

The SecureStorage service interacts with both the client, and the authenticator. In order to access the secure storage, the user has to pass the **token** as a GET query string to the SecureStorage service. The SecureStorage service will then do the following:

- **Check it's local cache, stored in its own tokens dictionary, for a matching token**
- **If the token is in the cache, but was inserted into the cache more than 5 minutes ago, delete the cached token and re-fetch it from the authenticator (see next line)**
- **If the token isn't in the cache, make a GET request to the authenticator for the token and fetch the token and associated client IP**
- **Compare the client's IP from the client request to the IP associated with the token – if they match, allow the user to access the resource**

The goal with this particular set-up is to make a stateful (i.e. not easily scalable) service that can handle authentication, **but limit how many requests are necessary** to this less-scalable resource. The authenticator only needs to handle requests when a user logs in or out (generating and deleting tokens), and then only needs to be interacted with once every 5 minutes per client, thanks to the caching of the tokens and associated IP addresses in the SecureStorage service. This achieves isolation between the resource-fetching code in our storage service, and the login-handling code in our authenticator, in addition to reducing the load on the authenticator.

Writing the Authenticator Service

To begin with, we'll implement the POST request handling logic for our authenticator service. This function, **do_POST()**, will catch a user POST request containing a username and a password. It will verify these credentials are correct, and also that a **token does not already exist for the client's IP**. If the credentials are correct and a token doesn't already exist, we'll create a new token using the **uuid.uuid4()** function, save the **token and the client's IP in our tokens dictionary**, and then return a 200 response to the user along with the token. If anything goes wrong here (wrong credentials, token already exists), we'll return an HTTP error status and an error message.

First, we'll want to double check that user actually request the **/login** page with their POST request. If they did, we should get the request data. The **getRequestData()** function is included in this file, and it will fetch the POST request parameters for you. They are returned in a dictionary with two keys: **username** and **password**. We'll then first check to see if the **username exists within the logins dictionary:**

```
if self.getPage() == '/login':
    requestData = self.getRequestData()
    if requestData['username'] in self.logins.keys():
```

After we've confirmed the username actually exists in our user space, we'll check to see if the request password is correct. If it is, we should then fetch the **client's IP address using the self.client_address[0] object:**

```
if self.logins[requestData['username']] == requestData['password']:
    clientIP = self.client_address[0]
```

If we've confirmed both the username, and the password are correct, the last check we need to perform is that the client IP doesn't have a token associated with it yet. We can do this by checking the **values** (not the **keys**) of our **tokens** dictionary, and verifying the user's IP doesn't exist there:

```
if clientIP not in self.tokens.values():
    token = str(uuid.uuid4())
    self.tokens[token] = clientIP
    self.set_headers(200)
    self.wfile.write(bytes("Login successful, your token: " + token, "utf-8"))
```

Once we've confirmed that the clientIP doesn't have a token yet, we do a couple things. We create a new token using **uuid.uuid4()** – this returns a 36 digit, random UUID (Universally Unique Identifier). We then create a new entry in the **tokens** dictionary, with our UUID (or token) as the **key**, and the client's IP as our **value**. We then set the headers for a 200 (OK) response, and return a line of text to the user with the token included.

Each of these if statements in the **do_POST** function will require an **else** statement, which sets the headers for the correct error code (**409 (conflict), or 401 (unauthorized)**) and returns an error message. The username and password checks should return a 401, and the token-alreadyexists check should return a 409. I have provided one of the 401 error code else clauses below:

```
else:
    self.set_headers(401)
    self.wfile.write(bytes("Login failed: Username or password was incorrect.", "utf-8"))
```

I leave it up to you to add the remaining error-return clauses for the if statements above, using the correct HTTP status code. You should have three of them in total in this function, all with a format similar to the one shown above.

We can verify that our POST request handling is working using the **TestAuth.html** file included with this lab. Try entering both correct and incorrect credentials; the results of your POST request will be shown below the login box, as shown in the following image:

With our POST function complete, we can move onto our GET request handling, using the **do_GET()** function. In this function, we'll need to check if the page (the URI) requested is a valid token, or if it's the logout URI; if it's neither, we need to return a **404** (Not Found) error status.

First, we'll use the **getPage()** function to get the path the client has requested. The path will include a leading slash, i.e. **/55b04075-582f-433b-8d70-c3d236976917**. In order to do search our dictionary for an equivalent token, we'll trim the leading slash with string slicing, and then check if **in** our **tokens** dictionary as a **key**:

```
if self.getPage()[1:] in self.tokens.keys():
    self.set_headers(200)
    self.wfile.write(bytes(self.tokens[self.getPage()[1:]], "utf-8"))
```

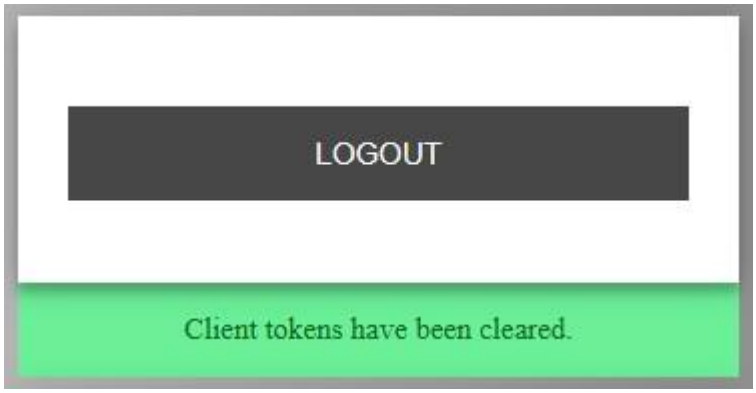
If the path requested is a valid token, we'll return a **200 OK** status, and then return the **IP address associated with that token**. We do this by once again using our string-sliced path as the key (the token UUID) for our **tokens** dict, which will return a **value** of the client's IP address. Don't forget to cast to bytes!

If the requested path isn't a valid token, there is still one GET request path that it could be: it could be the **/logout** URI. We'll use an **elif** to check if the path is equal to /logout, and if it is, we'll want to fetch the current client's **IP address**. We'll then search our **tokens** dictionary for any matching IP addresses; if we find any, we should go ahead and **delete them** from the dictionary. This removes the tokens from service and makes them no longer valid (although it may take up to 5 minutes for this to be registered, due to caching):

```
elif self.getPage() == '/logout':
    clientIP = self.client_address[0]
    for key, value in dict(self.tokens).items():
        if value == clientIP:
            del self.tokens[key]
```

If we've deleted tokens (and thus successfully logged the user out), we should return a **200 OK** status, and a message to the user informing them they have logged out. We also need to deal with the final **else** case, which fires if the requested path is neither a valid token, or the logout page. In that case, we should just return a **404 status**. **I leave this last bit of the do_GET()** function up to you.

You can test the functionality of the **GET** request handling using your web browser. Try logging in (using the test page above), and then try going to **http://127.0.0.1:8080/<your UUID here>.** If successful, you should see your IP address (probably 127.0.0.1, or localhost, since this is all local) in the page output. You can test the logout functionality using the **TestAuth.html** page again.



You should verify that if you log in, you can visit your UUID URI and get your IP. You should also verify that if you log out, and then try to visit your UUID URI, that the page will then return a 404 error.

Writing the Secure Storage Service

Now, we can begin on our **SecureStorage service**. We'll go ahead and work on our token caching and handling function first.

The **getToken()** function performs a number of tasks. It checks to see if the token passed from a client request exists within the current cache. If it doesn't, it will try to fetch the token from the authenticator. If it succeeds, it stores the **token** as a **key** in the **SecureStorage tokens** dictionary, and uses a list of the **client IP**, and the **current time** (in seconds since UNIX epoch) as the **value**. If the token fetching fails (token doesn't exist, or is malformed), it returns **None**. If the fetching and caching is successful, it returns the **IP** address associated with the token (retrieved from the authenticator). We also need to make sure someone doesn't try to trick us or break our service by submitting a token named **logout**, since that's also a legitimate GET request that the server can make.

```
if token == 'logout':
    return None
if token not in self.tokens:
    try:
        fetchedIP = urllib.request.urlopen('http://127.0.0.1:8080/' + token).read()
        fetchedIP = fetchedIP.decode("utf-8")
        self.tokens[token] = [fetchedIP, time.time()]
        return fetchedIP
    except:
        return None
```

If the token already exists in the cache, we check the timestamp in the **tokens** dictionary to see when it was added, and compared it against the current time. If the difference is more than **300 seconds** (5 minutes), we delete the cached version, and we go ahead and re-call (a single recursion) the **getToken()** function to trigger a re-fetch from the authenticator. If the time difference is less than 300, we just return the **IP** associated with the token.

```
else:
    tokenVals = self.tokens[token]
    if (time.time() - tokenVals[1]) > 300:
        del self.tokens[token]
        return self.getToken(token)
    else:
        return tokenVals[0]
```

Finally, we'll need to complete the **do_GET()** function. We'll begin by verifying that the client has requested the root ('/') page. If they have, we'll go ahead and fetch the GET request query string parameters using the **getParams()** function. This returns a dictionary of the query string parameters; we are looking for one named **token**. We'll go ahead and check to see if **token** exists:

```
if self.getPage() == '/':
    parameters = self.getParams()
    if parameters.get('token'):
```

If the token does exist in our parameters, we can move on to checking the cache and fetching the token if needed using our **getToken()** function. We'll also want to fetch the current client IP address, as well:

```
clientIP = self.client_address[0]
fetchedIP = self.getToken(parameters.get('token'))
```

With this data, there are three possible states we need to handle:

1. If the client IP and the fetched IP are the same, return a 200 OK status, along with our super secret resource content (your choice)
2. Otherwise, if the IP addresses do not match, return a 401 (Not Authorized), and return an error message stating the client and token do not match.
3. Finally, if the token parameter wasn't included at all, we should return a 401 not authorized error and an error message stating that not token was provided.

I leave the final handling of these three conditions to you. The super secret content you return may be anything of your choosing – HTML, a message, etc.

You can test to see if your page is working correctly using the **TestAuth.html** file. When you successfully log in, it will generate a link for you to your SecureStorage service (on port 80), with the token already included for you:

Submission

Submit both the **SecureStorage.py** and **Authenticator.py** files on Moodle. Marking schema is provided below:

Marks	Item
4 Marks	Authenticator service works, creates tokens, and allows IPs to be fetched using GET requests
3 Marks	SecureStorage service only allows authenticated users to access resource

2 Marks	SecureStorage properly caches tokens and refreshes cache regularly
1 Bonus Mark	Custom secret content – be creative!