eecs441§3&4

Chatter with Sign-in Back End

Cover Page

DUE Wed, 11/10, 2 pm

You've designed and implemented a front end that signs the user in and collects an `ID Token` from Google to send to the back end. We'll now extend our `Chatter` back end to (1) receive the Google ID Token from the front end for authentication and to (2) verify users when they post `chatts`.

Install updates

Every time you ssh to your server, you will see something like:

```
N updates can be applied immediately.
```

if `N` is not 0, run the following:

```
server$ sudo apt update
server$ sudo apt upgrade
```

Failure to update your packages could lead to the lab back end not performing correctly and also make you vulnerable to security hacks.

If you see `*** System restart required ***` when you ssh to your server, please run:

```
server$ sync
server$ sudo reboot
```

Your ssh session will be ended at the server. Wait a few minutes for the system to reboot before you ssh to your server again.

Modified `Chatter` API

We'll add an `adduser` API and modify the `chatt` posting API for the new sign-in related functionalities.

`adduser` API

When a user signs in and submits their ID Token from Google, `adduser` will receive the token, make sure it hasn't expired, generate a new `chatterID`, store it in the database along with the user's username (obtained from the ID Token) and the `chatterID`'s expiration time. The `adduser` API will then return this `chatterID`, along with its lifetime, to the user.

API:

```
/adduser/
<- clientID, idToken
-> chatterID, lifetime 200 OK
```

The data format `adduser` expects is:

```
{
    "clientID": "YOUR_APP'S_CLIENT_ID",
    "idToken": "YOUR_GOOGLE_ID_TOKEN"
}
```

Notice how we don't do anything with user data upon sign out. In a real-world app, we would need to remove the user from the back end and add a button to revoke Google SignIn on the app.

Posting a `chatt` API

To post a `chatt` in this lab requires that `chatterID` be sent along with each `chatt`. The back end first verifies that the `chatterID` exists in the database. If the `chatterID` is found, the new `chatt`, along with the user's username (retrieved from the database) will be added to the `chatts` database. If it isn't, an error will be returned to the front end.

API for `postauth`:

```
/postauth/
<- chatterID, message
-> {} 200 OK
```

The data format `postauth` expects is:

```
{
    "chatterID": "YOUR_CHATTER_ID",
    "message": "Chitt chatts"
}
```

Adding a `chatters` table to `chatterdb`

We will be using two tables in this lab: the original `chatts` table from lab0 "as is" and a new `chatters` table to keep track of authorized users. Both tables will be part of our `chatterdb` database.

To keep track of users, we will add a new `chatters` table to our `chatterdb` database. Assuming you're running `psql` and already connected to `chatterdb`, create a `chatters` table and grant PostgreSQL user `chatter` access to it:

```
CREATE TABLE chatters (chatterid char(256), username varchar(255), expiration bigint);
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO chatter;
```

We will use `SHA256` to compute `chatterID`, so it should be of fixed size, 256 bytes.

Routing for new urls

First add the following new routes to the `urlpatterns` array in `~/441/chatter/routing/urls.py`:

```
    path('postauth/', views.postauth, name='postauth'),
    path('adduser/', views.adduser, name='adduser'),
```

Editing `views.py`

Adding `adduser()`

To verify Google's ID Token, we'll need to install the Google API python client:

```
server$ cd ~/441/chatter
server$ source env/bin/activate
(env):chatter$ pip install -U pip
(env):chatter$ pip install -U google-api-python-client
```

Next edit your `~/441/chatter/app/views.py` file and `import` the following libraries, including two from Google:

```
from google.oauth2 import id_token
from google.auth.transport import requests

import hashlib
```

Finally, add the following `adduser()` function to your `views.py`:

```
@csrf_exempt
def adduser(request):
    if request.method != 'POST':
        return HttpResponse(status=404)

    json_data = json.loads(request.body)
    clientID = json_data['clientID']   # the front end app's OAuth 2.0 Client ID
    idToken = json_data['idToken']     # user's OpenID ID Token, a JSon Web Token (JWT)

    now = time.time()                  # secs since epoch (1/1/70, 00:00:00 UTC)
```

```
    try:
        # Collect user info from the Google idToken, verify_oauth2_token checks
        # the integrity of idToken and throws a "ValueError" if idToken or
        # clientID is corrupted or if user has been disconnected from Google
        # OAuth (requiring user to log back in to Google).
        # idToken has a lifetime of about 1 hour
        idinfo = id_token.verify_oauth2_token(idToken, requests.Request(), clientID)
    except ValueError:
        # Invalid or expired token
        return HttpResponse(status=511)  # 511 Network Authentication Required

    # get username
    try:
        username = idinfo['name']
    except:
        username = "Profile NA"

    # Compute chatterID and add to database
    backendSecret = "giveamouseacookie"   # or server's private key
    nonce = str(now)
    hashable = idToken + backendSecret + nonce
    chatterID = hashlib.sha256(hashable.strip().encode('utf-8')).hexdigest()

    # Lifetime of chatterID is min of time to idToken expiration
    # (int()+1 is just ceil()) and target lifetime, which should
    # be less than idToken lifetime (~1 hour).
    lifetime = min(int(idinfo['exp']-now)+1, 60) # secs, up to idToken's lifetime

    cursor = connection.cursor()
    # clean up db table of expired chatterIDs
    cursor.execute('DELETE FROM chatters WHERE %s > expiration;', (now, ))

    # insert new chatterID
    # Ok for chatterID to expire about 1 sec beyond idToken expiration
    cursor.execute('INSERT INTO chatters (chatterid, username, expiration) VALUES '
                   '(%s, %s, %s);', (chatterID, username, now+lifetime))

    # Return chatterID and its lifetime
    return JsonResponse({'chatterID': chatterID, 'lifetime': lifetime})
```

> For Python-PostgreSQL interaction, see Passing parameters to SQL queries.

The function `adduser()` first receives a POST request containing an ID Token and Client ID from the front end. It calls Google's library to verify the user's ID Token, passing along the Client ID as required by Google. The verification process checks that the ID Token hasn't expired and is valid. If the token is invalid or has expired, a `511`, "Network Authentication Required" HTTP error code is returned to the front end.

Next, a new `chatterID` is computed as a SHA256 one-way hash of the ID Token, a server's secret, and the current time stamp. A `lifetime` is assigned to the `chatterID`. The `lifetime` should normally be set to be less than the total expected lifetime of the ID Token, and in any case not more than the remaining lifetime of the ID Token. The idea is that during the lifetime of `chatterID`, the user does not need to check the freshness of their ID Token with Google.

The `chatterID`, the user's name obtained from the ID Token, and the `chatterID`'s lifetime are then entered into the `chatters` table. At the same time, we do some house keeping and remove all expired `chatterIDs` from the database.

Finally, the `chatterID` and its lifetime are returned to the user as a JSON object.

Feel free to test the implementation with Postman. Adding users with a valid Google ID Token will result in a `200` response status while attempting to add users with an invalid token results in a `511` error. Before you test, remember to restart Gunicorn after you've updated `urls.py` and `views.py`.

If your front end is not yet completed, you can debug your back end by putting a debugging breakpoint at the start of your front end's `ChattStore.addUser()` function and inspect the first `idToken` parameter passed in to `ChattStore.addUser()`. You can use it, together with your front end's `clientID` to test your back end.

`postauth()`

We now add `postauth()`, which is a modified `postchatt()`, to your `~/441/chatter/app/views.py`:

```
@csrf_exempt
def postauth(request):
    if request.method != 'POST':
        return HttpResponse(status=404)
    json_data = json.loads(request.body)

    chatterID = json_data['chatterID']
    message = json_data['message']

    cursor = connection.cursor()
    cursor.execute('SELECT username, expiration FROM chatters WHERE chatterID = %s;', (chatterID,))

    row = cursor.fetchone()
    now = time.time()
    if row is None or now > row[1]:
        # return an error if there is no chatter with that ID
        return HttpResponse(status=401) # 401 Unauthorized

    # Else, insert into the chatts table
    cursor.execute('INSERT INTO chatts (username, message) VALUES (%s, %s);', (row[0], message))
    return JsonResponse({})
```

To post a `chatt`, the front end sends a POST request containing the user's `chatterID` and `message`. The function `postauth()` retrieves the record matching `chatterID` from the `chatters` table. If `chatterID` is not found in the `chatters` table, or if the `chatterID` has expired, it returns a `401`, "Unauthorized" HTTP error code. Otherwise, it retrieves the corresponding `username` from the table and inserts the `chatt` into the `chatts` table with that `username`. Note: `chatterID`s are unique in the `chatters` table.

To test using Postman, first remember to restart Gunicorn after you've updated `views.py`. Then create a new Postman HTTP POST request to `adduser` with a valid Google Client ID and a Google ID Token:

```
{
    "clientID": "YOUR_APP'S_CLIENT_ID",
    "idToken": "YOUR_GOOGLE_ID_TOKEN"
}
```

You should receive a `chatterID` and its lifetime as a response. Now use this `chatterID` in place of `YOUR_CHATTER_ID` below to POST a message to `postchatt`:

```
{
    "chatterID": "YOUR_CHATTER_ID",
    "message": "Chitt chatts"
}
```

You should get a `200` server response in return.

We will be using the original `getchatts()` from lab0 with no changes.

That's it! You're now all set up on the back end for `Chatter`! Be sure to confirm that your back end works with your front end.

Submission guideline

- Commit new changes to the local repo with:

  ```
  server$ cd ~/441/chatter
  server$ git commit -m "lab4 back end"
  ```

  and push new changes to the remote GitHub repo with:

  ```
  server$ git push
  ```

- If `git push` fails due to new changes made to the remote repo, you will need to run `git pull` first. Then you may have to resolve any conflicts before you can `git push` again.

If you're not done with the front end, you can now return to complete the front end.

References

- Authenticate with a backend server
  > Once the mobile client obtained an ID Token from an SSO, it presents the ID Token to a backend server. The backend server must authenticate that ID Token.
- Django Tables
- "django.db.utils.ProgrammingError: permission denied for relation django_migrations"
- SHA in Python

- SQL

Prepared for EECS 441 by Benjamin Brengman, Ollie Elmgren, Wendan Jiang, Alexander Wu, and Sugih Jamin	Last updated: June 11th, 2021