CS 310
Scalable Software Architectures
Fall 2023

Extra Credit Project[1] (v1.0)

Assignment: Benford's Law Part 02 --- authentication
Submission: via Gradescope (unlimited submissions)
Policy: individual work only, late work is accepted
Complete By: Friday Dec 08th @ 11:59pm CST

Late submissions: late submissions are not accepted since 12/8 is the last day of the quarter.

What is meant by "extra credit"

This project is completely optional. Given its late release, it can be used for extra credit worth up to 30 points. You must complete the project in its entirety to receive extra credit --- i.e. you must earn a Gradescope score of 30/30. The extra credit will first be applied to your exam, for a max exam score of 100. If extra credit points remain after application to the exam, the points will be used as needed on projects 01 – 03 and the final project --- wherever the points do the most good. No project score may exceed 100.

Overview

In Project 03, you built a serverless and event-driven application that (1) analyzed a PDF when one appeared in S3, (2) wrote the analysis results back to S3, and (3) exposed an API to allow users to upload these PDFs, view the status of the analysis, and download results. However, the application lacked a critical component of cloud computing and web application development: security.

In Project 04, you'll build off your Benford's Law application from Project 03 and add authentication. This entails the following:

1. *Users will be required to sign in before uploading or downloading a PDF for analysis.*
2. *Users will only be able to download the results for their own PDFs.*
3. *The database will only hold hashes of users' passwords rather than passwords in plain text.*
4. *The API will respond with a temporary access token upon user login for the user to authenticate API requests (like uploading and downloading).*

We'll touch on concepts like password hashing and using access tokens as you implement them.

[1] This project was designed, implemented, tested, and written by NU undergraduate Dilan Nair.

Step 1: Setting up your AWS infrastructure

Step 1.1: Setting Up the Database

The database here in project 04 is identical to the MySQL database in Project 03. If you still have your MySQL BenfordApp database from project 03, you can skip to **Step 1.2** on the next page.

If you deleted your RDS server or don't remember how to execute SQL queries, review Project 01 - Part 01 (page 13 for creating an RDS server and page 21 for running SQL queries).

If you deleted your database from project 03 and need to recreate, open a connection to MySQL (using VS Code or MySQL Workbench) and run the provided SQL file benfordapp-database.sql.

For reference, here's a summary of the database schema. Do NOT copy-paste this code in order to create your database, follow the link above to the provided SQL file:

```
CREATE TABLE users
(
    userid    int not null AUTO_INCREMENT,
    username  varchar(64) not null,
    pwdhash   varchar(256) not null,
    PRIMARY KEY (userid),
    UNIQUE      (username)
);

ALTER TABLE users AUTO_INCREMENT = 80001;  -- starting value

CREATE TABLE jobs
(
    jobid             int not null AUTO_INCREMENT,
    userid            int not null,
    status            varchar(256) not null,  -- pending, completed, error msg
    originaldatafile  varchar(256) not null,  -- original name from user
    datafilekey       varchar(256) not null,  -- filename in the bucket
    resultsfilekey    varchar(256) not null,  -- results filename in bucket
    PRIMARY KEY (jobid),
    FOREIGN KEY (userid) REFERENCES users(userid)
);

ALTER TABLE jobs AUTO_INCREMENT = 1001;  -- starting value
```

<< continued on next page >>

Step 1.2: Creating the Lambda Layer

You built a Lambda layer in Project 03, but since the dependencies for Project 04 code are a bit different, you'll need to create an additional layer for the new modules we are using. We'll use **AWS CloudShell** to package the new dependencies need by the project 04 Lambda functions.

1. Open **CloudShell** by searching for it and selecting it on your AWS console (or click the > icon on the title bar). This opens up a command-line shell on Linux that has all of the tools we need.
2. Create a .zip file that contains the necessary modules:

```
rm -r python
mkdir python
cd python
pip3 install bcrypt PyJWT -t .
cd ..
zip -r bcrypt-pyjwt-layer.zip python
```

3. Run `ls` to verify that **`bcrypt-pyjwt-layer.zip`** is there!
4. Copy your .zip file to one of your S3 buckets, which will make the file accessible to AWS Lambda:

```
aws s3 cp bcrypt-pyjwt-layer.zip s3://your-bucket-name
```

   [Note: you can get a list of your buckets using this command: **`aws s3 ls`**]

5. Find your .zip file in your S3 bucket, select to view properties, then copy the **Object URL**.

Now, let's build the Lambda layer itself so your Lambda functions can use it.

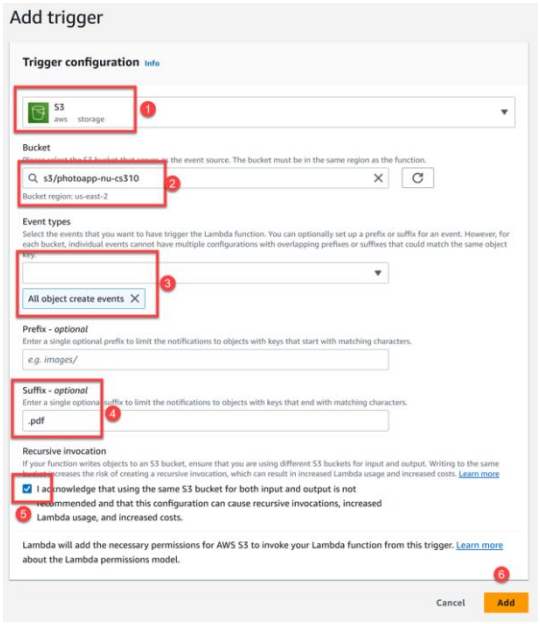
1. Open **Lambda** by searching for it and selecting it on your AWS console.
2. Select Layers in the sidebar (under "Additional resources"), then press Create layer.

3. Configure and create the layer.
   a. Set the name to **`bcrypt-pyjwt-layer`**.
   b. Select **Upload a file from Amazon S3**, then paste in the URL you copied earlier.
   c. Select **x86_64** under compatible architectures.
   d. Select **Python 3.11** as the compatible runtime. That's what our functions will use.
   e. Press **Create**!

Step 1.3: Creating the Lambda Functions

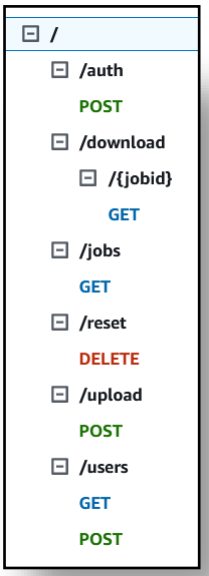
The application consists of 7 Lambda functions. Most of the code in each will be provided, but some small chunks involving authentication will be incomplete for you to implement.

1. Download the Project04 lambda functions from dropbox, each is a separate .zip file.
2. You will need your "config.ini" file from project 03; a template is available here.
3. **For each of the 7 Lambda functions**:
    a. Add your "config.ini" file to the .zip file; you should be able to drag-drop.
    b. On your **Lambda** dashboard on your AWS console, press **Create function** to supply the necessary information and create the function.
        i. Set the name of your function to the name of the folder. For example, **proj04_auth**. This just helps with organizing things, but you can technically name it whatever you want.
        ii. Set the runtime to **Python 3.11**.
        iii. Select **x86_64** as the architecture.
        iv. Press **Create function**!
    c. You should be redirected to the function's configuration page. Scroll down and in the **Layers** section, press **Add a layer** to add each of the following two layers:
        i. Select **Custom layers** as the layer source. Select the layer you created in Project 03, which should be "pymysql-pypdf-layer". Select the latest version, and click **Add**.
        ii. Repeat… Add a layer, custom layers, select the layer you created here in Project 04, which is "bcrypt-pyjwt-layer". Select the latest version, and click **Add**.
    d. On the function's configuration page, switch to the **Configuration** tab. Under **General configuration**, press **Edit**, change the timeout to **5 min 0 sec**, then press **Save**.
    e. On the function's configuration page, switch to the **Code** tab. Press **Upload from**, select **.zip file**, then upload the zipped function file for the function. This should populate the online editor and deploy the function for you.
4. Remove the event trigger from your **proj03_compute** lambda function: select the function, Configuration tab, Triggers, select the S3 trigger, and delete.
5. **For proj04_compute**, visit the function's configuration page. Under **Function overview** at the top, press **Add trigger** to start configuring the function to run for each PDF uploaded to S3.
    a. Select **S3** as the source.
    b. Select the bucket to which PDFs will be uploaded.
    c. Ensure the event types include **All object create events**. This should be the default.
    d. Set the suffix to be **.pdf**. This is IMPORTANT! Omitting this can lead to recursive invocation when your function uploads the results .txt file.
    e. Select the box to acknowledge recursive invocation (which shouldn't happen if you set the suffix).
6. Done!



Step 1.4: Creating and Routing Your API

Now, similarly to Project 03, you'll need to create and route your API using AWS API Gateway. This allows your Lambda functions to be invoked by HTTP requests.



1. Open **API Gateway** by searching for it and selecting it on your AWS console.
2. Press **Create API**.
3. Under **REST API**, press **Build**.
4. Name the API **Project04 Benford API** and press **Create API**.
5. Configure each resource.
    a. /auth
        i. Select the root resource (labeled **/**), then press **Create resource** above.
        ii. Name the resource **auth**, press **Create Resource**.
        iii. Select the **/auth** resource, then press **Create method**.
        iv. Select **POST** as the method type.
        v. Select **Lambda function** as the integration type.
        vi. Enable **Lambda proxy integration**.
        vii. Select your **proj04_auth** Lambda function.
        viii. Press **Create method**.
    b. /jobs
        i. Repeat the steps in 5a for **jobs** with **proj04_jobs**, but use **GET** as the method.
    c. /reset
        i. Repeat 5a for **reset** with **proj04_reset**, but use **DELETE** as the method type instead.
    d. /upload (there is no `userid` path parameter here in project 04)
        i. Repeat 5a for **upload** with **proj04_upload**, with **POST** as the method type.
    e. /users
        i. Select the root resource (labeled **/**), then press **Create resource** above it.
        ii. Name the resource **users**, press **Create Resource**.
        iii. Select the **/users** resource, then press **Create method**.
        iv. Select **GET** as the method type.
        v. Select **Lambda function** as the integration type.
        vi. Enable **Lambda proxy integration**.
        vii. Select your **proj04_users** Lambda function.
        viii. Press **Create method**.
        ix. **Repeat steps iii-viii for a POST method type for /users as well.** Use the same Lambda function.
    f. /download
        i. Select the root resource (labeled **/**), then press **Create resource** above it.
        ii. Name the resource **download**, press **Create Resource**.
        iii. Select the **/download** resource, then press **Create resource** again.
        iv. Name the sub-resource **{jobid}** (this is a path parameter), press **Create Resource**.

v. Select the **/{jobid}** sub-resource below the **/download** resource, then press **Create method**.
vi. Select **GET** as the method type.
vii. Select **Lambda function** as the integration type.
viii. Enable **Lambda proxy integration**.
ix. Select your **proj04_download** Lambda function.
x. Press **Create method**.

Wow, that was a lot, but it should have been straightforward to do.

The last step is to deploy your API. Press **Deploy API**, select ***New stage***, name the stage **prod** for production, then press **Deploy**. Once AWS provisions the stage, copy the **Invoke URL** and save it somewhere. You'll need it in the next step below (step 1.5).

Step 1.5: Setting Up the Test Client

While the client will fail on operations that you have not yet implemented, it's good to get it set up now so you can test parts of your application as you develop it. Find **Project 04 (client)** on the Replit team. Open the project and paste your **Invoke URL** from step 1.4 into the **`benfordapp-client-config.ini`** as the value for the **webservice** property. Now, you can test as we go through the rest of the handout…

Step 2: User Signup

The application should allow a user to sign up by providing a username and a password. The **proj04_users** Lambda function does two things:

- On GET request, it lists all users in the database.
- On POST request, it adds a user to the database, storing a username and a hash of a password, and then returns autogenerated user ID.

However, the POST request part is incomplete. You'll need to add the following missing functionality to your **proj04_users** lambda function:

1. If a user with the specified username already exists, respond with error **409 Conflict** and a message that says "user already exists".
2. If the user does not exist, hash their password and insert both the username and password hash to the database.

TIP: There are library files to help with a lot of the logic. Check these out:

- **`datatier.retrieve_one_row()`** and **`datatier.perform_action()`** from `datatier.py`
- **`auth.hash_password()`** from `auth.py` (you can use the default 12 salt rounds)

- **`api_utils.error()`** from `api_utils.py`

TEST: Use the provided client to test your functionality:

1. List all users.
2. Add a new user.
3. List all users again to make sure the new user is there.
4. Add a new user again and use the same username. It should fail with status code 409.

CONCEPT: Why hash a password?

Imagine a big company that requires a username and password to log in to their website. To log you in and load your data, this username and password combination must be stored somewhere, like in a database. If their database was breached and they stored your password in plain text, attackers would be able to log in as you. If you used the same password on multiple websites, they may be able to access other websites, too. Password hashing makes use of hashing algorithms, which are mathematical, uniform, consistent, and one way. Salting the password adds extra characters to make the password hash even more secure. A password can't be discovered from a hash. Instead, when a user tries to log in, the password can be hashed again and compared with the stored hash to determine whether the password is correct.

However, because password hashing is consistent, it's still important to have something unique. Something like "hello" will always output the same hash in a hashing algorithm. Hackers can guess simple passwords like these through brute force.

Step 3: User Login

The application should allow a user to log in with a username and a password. Once validated, the application should respond with an access token that a client can use to perform operations on behalf of the user. Update your **proj04_auth** lambda function so it does the following:

1. Given a username, try to find the user in the database.
2. If not found, respond with error **404 Not Found** and a message that says "no such user".
3. If the user does exist, verify the provided password with the stored hash.
4. If the password is wrong, respond with error **401 Unauthorized** and a message that says "password incorrect".
5. If the password is correct, generate an access token with their user ID. Encrypting the token requires a secret. Make it whatever you want. When verifying the token in other handlers, you'll need to decrypt it with the same secret.
6. Respond with the generated access token. This part is already completed for you.

TIP: There are library files to help with a lot of the logic. Check these out:

- **`datatier.retrieve_one_row()`** from `datatier.py`
- **`auth.check_password()`** and **`auth.generate_token()`** from `auth.py`

- **`api_utils.error()`** from `api_utils.py`

TEST: Use the provided client to test your functionality:

1. Log in with an invalid username and any password. It should fail with status code 404.
2. Log in with a valid username and an incorrect password. It should fail with status code 401.
3. Log in with a valid username and a correct password. It should be successful. You should see a new file in your client project called **`sessions.json`** and it should have the user's access token in it.

CONCEPT: What's the point of an access token?

Access tokens are encrypted strings that store some information about a user. Rather than requiring the user to log in with their username and password for every operation they want to perform, the access token is essentially a temporary cache of their credentials. They expire after a certain time (access token expiration is what can cause websites to require you to log in again after a while). They can also contain other data like scope, which determines what operations a client can take with the access token. These are stored on the client side.

One could ask: why not just store the username and password on the client side and use those to authenticate each call? Well, imagine if that data was seen or stolen by someone. Now, they have your password and will have permanent access to everything unless you change it. If they were to steal an access token, although they may be able to have some access, they'd be limited by the scope of the access token and the time the access token has remaining.

Step 4: Handling Uploads

In Project 03, the **/upload** endpoint would accept a user ID as a path parameter and upload the PDF data on behalf of that user. Here in Project 04, no user ID is accepted. Instead, this is an authenticated endpoint, so the client must send the access token to the API application, which must verify it and extract the user ID from it. All of the actual upload functionality is done for you, but you'll need to handle the authentication part. Update your **proj04_upload** lambda function so it does the following:

1. Extract the access token from the request headers.
2. If no token is found, respond with error **401 Unauthorized** and a message that says "no bearer token in headers".
3. If a token is found, extract the user ID from it. You'll need to use the same secret you used earlier to decrypt the token.
4. If the token is invalid (malformed, expired, etc.), respond with error **401 Unauthorized** and a message that says "invalid access token".
5. If the token is valid, store the user ID in a variable named **`userid`**. This is used for the remaining functionality of this handler, which is completed for you.

TIP: There are library files to help with a lot of the logic. Check these out:

- **`auth.get_token_from_header()`** and **`auth.get_user_from_token()`** from `auth.py`
  Note that `auth.get_token_from_header()` returns `None` if the token is not found, but

`auth.get_user_from_token()` raises an exception if the token is invalid.

- **`api_utils.error()`** from `api_utils.py`

TEST: Use the provided client to test your functionality:

1. Ensure you're logged in as a user (list sessions to make sure there's an active session, otherwise log in as a user).
2. Upload a PDF.
3. View jobs and verify that the job's user ID matches the user ID you're logged in as (you may have to list all users to see which username belongs to which user ID).

CONCEPT: Where in the request headers is the access token stored?

Access tokens are stored in the Authorization request header under the Bearer auth scheme.

Step 5: Handling Downloads

In Project 03, the **/download** endpoint would accept a job ID as a path parameter and download the PDF analysis results text file. Any job ID would be accepted, regardless of who the job belonged to. Here in Project 04, a job ID is still required as a path parameter, but an error will be sent if the job ID does not belong to the user in this authenticated endpoint. Like the upload handler, all of the download functionality is done for you, but you'll need to handle the authentication part. Update your **proj04_download** lambda function so it does the following:

1. Extract the access token from the request headers.
2. If no token is found, respond with error **401 Unauthorized** and a message that says "no bearer token in headers".
3. If a token is found, extract the user ID from it. You'll need to use the same secret you used earlier to decrypt the token.
4. If the token is invalid (malformed, expired, etc.), respond with error **401 Unauthorized** and a message that says "invalid access token".
5. If the token is valid, store the user ID in a variable named userid. This is used for the remaining functionality of the handler, all of which is completed for you other than step 6.
6. After the job is retrieved from the database, check if the job owner user ID is the same as the user ID. If the job does not belong to the authenticated user, respond with error **403 Forbidden** and a message that says "job does not belong to user". Make sure that back in the auth handler you stored the user ID in the token and not the username, otherwise you may have some problems here.

TIP: There are library files to help with a lot of the logic. Check these out:

- **`auth.get_token_from_header()`** and **`auth.get_user_from_token()`** from `auth.py`
  Note that `auth.get_token_from_header()` returns None if the token is not found, but `auth.get_user_from_token()` raises an exception if the token is invalid.

- **`api_utils.error()`** from `api_utils.py`

TEST: Use the provided client to test your functionality:

1. Ensure you're logged in as a user (list sessions to make sure there's an active session, otherwise log in as a user).
2. Upload a PDF.
3. Log in as a different user and upload another PDF.
4. While logged in as the second user, download results using a job ID that belongs to the first user. It should fail with status code 403.
5. Download results using a job ID that belongs to the second user. It should succeed (or fail due to a pending status rather than a lack of access).
6. Switch to the first user, then try downloading results that belong to the first user and the second user. Make sure the successes and failures are what you'd expect.

Client-side Testing and Electronic Submission

Thoroughly test your code using the provided client. When you feel confident, submit your project for grading.

Your application will be collected and auto-graded using Gradescope. A few days before the due date, an assignment will open named **Project 04**. You'll submit the following six files for evaluation:

1. **benfordapp-client-config.ini** from your client-side testing
2. **config.ini** from your Lambda functions
3. lambda_function.py from *proj04_auth* renamed to **auth_lambda_function.py**
4. lambda_function.py from *proj04_download* renamed to **download_lambda_function.py**
5. lambda_function.py from *proj04_upload* renamed to **upload_lambda_function.py**
6. lambda_function.py from *proj04_users* renamed to **users_lambda_function.py**

You have unlimited submissions with a max score of 30/30 for 30 extra credit points.

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your "Submission History". This must be done before the due date.

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found here. In summary, here are NU's eight cardinal rules of academic integrity:

1. *Know your rights*

2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU's academic integrity website. With regards to CS 310, unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.