Fall 2025 CSE 380
Project 1: User Management and Authentication

September 30th, 2025

---

1 A Word of Warning

Make sure you read through the submission instructions thoroughly before you submit your project. Failure to comply with the submission instructions will result in a zero for this project. Read this document thoroughly, it is very long (I know), you might hate reading (I know I do), but almost everything you could need to know is in here. If you have questions about something, and the specs are confusing, put that in your question. For example: "The specifications say this about this topic, but I was wondering if this means this other thing as well". If you ask me a question that is directly in the specs, I will just tell you to go read the specs.

2 Project Due Date

This project is due on October 14th, 2025, before 10:00pm. Follow all submission instructions at the end of this document. There is also a project checkpoint submission on October 9th before 10:00pm. Follow the same submission instructions for the checkpoint as for the final submission.

3 Project Overview

In this project you will build a flask application connected to a SQLite database that will handle user management and authentication. Your project will allow users to create accounts, login to their accounts, change their information, etc. Your project will also handle authorization by implementing JSON web tokens from scratch. We will not be implementing any front end, and instead the testing will be done using scripts that send HTTP requests to your flask server. An example request is on D2L, and more test cases will be released. The lecture will go over a few ways that you can test without sending requests, instead viewing the results on a web browser.

Flask is python-based, meaning that the code you write is mostly python, with just a few flask components (for our course). Here are installation instructions, but you can install flask through a few other means https://flask.palletsprojects.com/en/3.0.x/installation/. If you want a tutorial on flask, I recommend using either https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world or the official flask documentation at https://flask.palletsprojects.com/en/3.0.x/tutorial/. This project assumes you have knowledge of python, but little knowledge of flask. In general we will not require any in depth flask knowledge. You can design your solution in any way you prefer, as long as it follows the project specifications. We have provided a sample flask application that can be run immediately. It is very simple, and all contained in one file. You can build off of the sample flask server if you like, but it is not required.

4 Allowed Resources

You are not allowed to use any python libraries that are not built-in, or flask. For a list of built-in python modules, see this link.

If you use a python library/module that is explicitly not allowed, you will receive an ADR for purposely and dishonestly circumventing the project requirements.

5 Project 1 Specifications

You are going to implement a user authentication and authorization system, using a SQLite database and a flask server. Your implementation will consist of the database design, and the flask server. The requests to the server will be handled by the instructor's code for testing.

This project will follow the client-server model discussed in lecture, with your flask implementation being the "always-on" server. Your SQLite database will be run locally for simplicity's sake. In an actual deployment, your database would most likely be on its own dedicated database server.

Your server must accept requests to create a new user, log the user in, change the user's password, view the user's data etc. You must also build the database infrastructure to support these operations.

A sample flask project that includes a SQLite database has been released on D2L. You can build off of this code, or start from scratch. As long as your project can be run, and has endpoints as specified, the structure of your project is up to you.

All requests will come in as form data, exactly how it is shown in the example request file. You can assume this will always be the case (even though it's not always realistic).

5.1 The Database

For your database, you must use the sqlite3 module that is built in to python 3.X. You cannot use any other database implementation, including anything like SQLAlchemy. Failure to use SQLite for the database will result in an automatic zero for the project, and a possible ADR.

The released starter code sets up a sample database already, you can build off of this structure if you like. There is a released .sql file that has all of the DROP and CREATE statements to set up a table. You must submit your final .sql file for the project in addition to all of the other project requirements.

Whenever you start the server and the database, you should DROP all of your tables (as seen in the .sql file) in case a previous database file is still present. This will make sure that you never error out when trying to start your project. Making sure the database is up and running is handled rudimentarily in the example code, using a simple global variable to determine if the database needs to be set up. You can use this solution, or you can handle it in any other (more elegant) way that you want.

You should assume that every time the server is started up, that database does not exist. So the database only needs a lifetime for as long as your flask server is running. We will assume that the database is deleted once the server is shut down, although if you do not handle database file deletion it is not a problem. But you do have to make sure that when the server starts up the database is recreated from scratch and doesn't contain any of the data from a previous run.

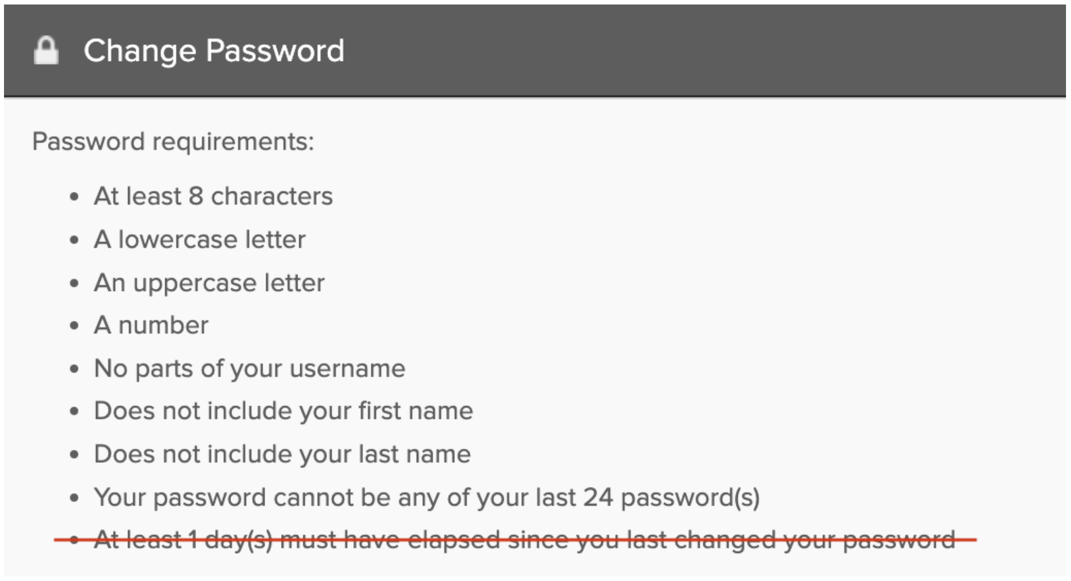
All of your SQL querying that happens in python with information provided by users (GET and POST requests) must be done using parameterized queries. Parameterized queries were discussed in lecture, and some example code has already been released. I will be intentionally trying to break you database with SQL injection attacks, and if they succeed then you will get a zero on the project.

You should read this entire document thoroughly, and then design a database schema based on what is being asked. It would be valuable to create an E/R diagram of your database, however that will not need to be submitted as part of the project. If you are unsure of your DB design, please come to office hours or make a private piazza post, and we will be happy to review your DB design.

5.2 User Creation

The first type of user query that we'll implement is user creation. As we discussed in lecture, MSU's password requiremenmts leave a lot to be desired. But we'll implement them anyways. Normally, you'd check for these requirements on the client side, however, since we are only implementing the server, we'll check them in our flask application. Figure 5.2 shows all of the requirements a password must meet. The requirement on not using a previous password is not relevant for user creation, but will be relevant for user editing/password changing. For our sanity, you only need to check if the ENTIRE username, first name, or last name is in the password, not only part of it, as MSU requires. Additionally, we will have a requirement that you cannot ever reuse a password, you do not have to keep track of the 24 most recent, just all previously used passwords.

Your endpoint must accept a POST request that contains the user's first name, last name, user-



Change Password

Password requirements:

- At least 8 characters
- A lowercase letter
- An uppercase letter
- A number
- No parts of your username
- Does not include your first name
- Does not include your last name
- Your password cannot be any of your last 24 password(s)
- ~~At least 1 day(s) must have elapsed since you last changed your password~~

Figure 1: MSU's Password Requirements

name, email address, and a password. Usernames and email addresses must be unique to each user, and must not be NULL. The password submitted must follow all of the requirements specified.

The endpoint for user creation must be '/create_user', meaning that to navigate to a web view of this end point you would enter http://127.0.0.1:5000/create_user.

The POST request will contain all of the parameters mentioned above, with the following naming:

- first_name: <user first name>
- last_name: <user last name>
- username: <user username>
- email_address: <user email address
- password: <user password>
- salt: <user salt>

You'll notice that there is an additional parameter that will be sent in with user creation, and this is the salt that will be used for hashing the user's password. Normally, the salt would be generated on the server, but so that we can test reliably, we'll send it in with the user creation POST request.

Passwords should be hashed using the input salt, and using the SHA256 hashing algorithm. As discussed in lecture, passwords will never be stored in plaintext. You need to use the python hashlib library to do your password hashing. You should use the .hexdigest() method for the hashing (this will make more sense once you look at the library documentation).

Your user creation endpoint needs to return a JSON object that contains a status code, as well as the hashed password. Returning the hashed password is not standard for user creation, but is how we'll test your hashing. You should use the python JSON module to craft the return value. You can possibly use another method of generating JSON, as long as it comes from a built-in module of python. You cannot use any non-built-in modules of Python (except flask).

The JSON object you return will have two key-value pairs, one for the status code, and one for the hashed password.

```
{
    "status": <code>
    "pass_hash": <password hash>
}
```

The status code will be a simple integer based on the scenario. The status code should be 1 for a successful user creation, 2 for an invalid username (already created), 3 for an invalid email address (already created), and 4 for an invalid password (doesn't follow the password rules). When a user creation fails, the password hash returned should be a string with the value 'NULL'. Please note that this is not the same as an HTTP status. The HTTP status should always be 200 for a valid request, even if the user creation fails. You do not need to handle the HTTP response, only returning the correct JSON object.

Whitespace does not matter in your JSON return value (except within a JSON string), we will strip all whitespace when we do the comparison. You just need a valid JSON object with the correct key-value pairs.

5.3 User Login

After a user has been created, they should also be able to login to the system. Login is relatively straight forward. Your endpoint will accept a POST request with the username and password for the user. If the password and username are correct, then return a message that says the authentication was successful, if not, return a message that says it was not successful.

The endpoint for user creation must be '/login', meaning that to navigate to a web view of this end point you would enter http://127.0.0.1:5000/login.

The POST request will contain all of the parameters mentioned above, with the following naming:

- username: <user username>
- password: <user password>

If the username or password is incorrect, you should return a fail status code. It does not matter what the failure is, all failures should return the same status code.

Upon successful authentication, your project should generate a JSON web token that can be sent in with future requests to avoid needing to login again. To see more about JSON web tokens, view this link or the lecture slides. You can see more on the implementation of the JSON web tokens in section 6.

The JSON object you return will have two key-value pairs, one for the status code, and one for the JSON web token.

```
{
    "status": <code>
    "jwt": <JSON web token>
}
```

The status code returned should be the integer 1 if the authentication was successful, and the integer 2 if it was unsuccessful. If the authentication was unsuccessful, the jwt key should have an associated value of the string 'NULL'.

Whitespace does not matter in your JSON return value, we will strip all whitespace when we do the comparison. You just need a valid JSON object with the correct key-value pairs.

5.4 User Editing

Your system will allow the user to edit their information. The user will be able to edit either their username or password, however, for simplicity's sake, can only edit one piece of information at a time. That means that you can update your username or your password, but only edit one per request.

When editing a user's information, all of the same restrictions apply, such as password constraints and username uniqueness, etc.

The user will make a post request that contains the previous value that existed, as well as what it should be updated to. If the previous value does not match what is in the database, the request to update should be denied. This is for every possible field, including passwords and regular text fields.

For password changing, you will need to also follow the last restriction shown in Section 5.2. That restriction is that you cannot change your password to be equal to a previously used password. MSU says it can't be the same as the previous 24 passwords, but for our purposes it cannot be the same as any password that user has used before. You should use the same salt value for every password for a user. This means that you'll only receive a salt value on user creation, and that salt should be used to hash all the passwords that user ever creates.

The endpoint for user editing must be '/update', meaning that to navigate to a web view of this end point you would enter http://127.0.0.1:5000/update.

To be able to update the user, they must have already logged into the system, which means that a JSON web token will always be sent with this request to verify that this user has the permissions to update their information. For our purposes, the JSON web token will be sent as a parameter in a POST request. The JSON web token must be verified before any updating occurs. The JSON web token will be captured from the return of the login action, and then sent back by our testing code.

The POST request will contain any of the parameters mentioned about, with the following naming:

- username: <old username>
- new_username: <new username>
- password: <old password>
- new_password: <new password>
- jwt: <JSON web token>

The jwt parameter will always be present for a user editing request, but the username or password fields will be present based on what is going to be updated. So if the username is updated, the the POST request will have username and new_username parameters, but if attempting to update password it will have the parameters corresponding to updating a password.

We will have three possible statuses returned from this endpoint, each specified with an integer.

The JSON object you return will have one key-value pair for the status code

```
{
    "status": <code>
}
```

The status code returned should be the integer 1 if the update was successful, and the integer 2 if it was unsuccessful because the username/password were incorrect (either the new or previous). If the authorization was incorrect (the JSON web token didn't match) then the returned status should be 3.

Whitespace does not matter in your JSON return value, we will strip all whitespace when we do the comparison. You just need valid JSON object with the correct key-value pairs.

5.5 User Viewing

Your system will allow the user to view their information. If a view request comes in, then a JSON object with all of their information (username, email address, first name, last name) is returned.

The endpoint for user creation must be '/view', meaning that to navigate to a web view of this end point you would enter http://127.0.0.1:5000/view.

While not traditional for viewing, this endpoint will also take a POST request, that contains only the JSON web token for the current user trying to view their information. This is done so that we can test more easily and so that you don't have to deal with issues of how to easily pass the JSON web token using HTTP. HTTP GET requests and authorization headers will be handled in Project 2 and beyond.

The POST request will contain only one parameter, the JSON web token, in the following format:

- jwt: <JSON web token>

The return value will contain a status code, as well as the information on the user, if authentication was successful. The JSON object you return will have two key-value pairs, for the status code and the information

```
{
    "status": <code>
    "data": {
                "username": <username>,
                "email_address": <email address>,
                "first_name": <first name>,
                "last_name": <last name>
            }
}
```

The status code returned should be the integer 1 if the request was successful, and the integer 2 if it was unsuccessful because the authorization was incorrect (the JSON web token didn't match). If the request was successful, the data of the user should be returned in a JSON object as shown above, in the same order. If the authorization was unsuccessful, return the string 'NULL' for the user data.

Whitespace does not matter in your JSON return value, we will strip all whitespace when we do the comparison. You just need valid JSON object with the correct key-value pairs.

5.6 Clearing

Your project should be able to accept a request to the endpoint '/clear'. This will tell the server to clear everything from it's database and act as if the server was restarted.

We will use this so that we do not have restart your server after each test.

This will come in as a GET request with no parameters.

The clear endpoint will be called for every single test case at the beginning, so this is the first endpoint you should set up. If your clear does not work (or does not exist), then every single test case will fail.

It is recommended that you simply delete the .db file from the clear endpoint. You can do this with the https://docs.python.org/3/library/os.html. This will avoid any concurrency issues that we didn't cover in the class.

One major potential issue that you can have is that your DB might become locked. What this means is that your database has an open connection on it, and a second connection cannot be established. This is particularly relevant for clearing and deleting. When we run our test cases, we will run them back to back without restarting your server. If a test case errors out, and you had an open DB connection, this will cause all subsequent cases to fail as well.

To avoid this, the first thing to do is to make sure that whenever you have a connection, you commit and close the connection when you are done with it. The only time you need more than this is in the case of errors. Any time you have a DB connection open, you should put the code that runs when the connection is open in a try-except block. This is not just your queries themselves, but any code that runs with a connection being open. In the case of an error, in the except block, make sure you call .close on the connection so that you don't have any dangling connections left open. If you do not handle this kind of error, then it is possible all of the test cases will fail.

As an FYI, we did not cover things like DB transactions and concurrency in CSE 380. These are very important parts of databases and are covered in depth in CSE 480.

6 JSON Web Tokens

Perhaps the least intuitive part of the project is the implementation of the JSON web tokens. JSON web tokens are used for authorization of a user to perform some tasks, in our case view their data or update their data. For a full discussion of JSON web tokens, see the lecture slides or https://jwt.io/.

Our JSON web tokens will follow the standard and will contain a header, payload, and a signature. The header and payload should be Base64 URL safe encoded, and we will use the HS256 algorithm for the signature. You should use the base64 URL safe method for the header and payload, and must use the hmac module for the signature. Same as for password hashing, you should use the .hexdigest() method for the signature. The key for the signature generation will exist in a file named: key.txt. key.txt will contain a single line that is the key that should be used. You should assume that the key.txt file will be in the outermost directory of your project.

Our JSON web token will have the following format.

The header will be as follows:

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

The payload will contain two fields, as follows:

```
{
    "username": <username>,
    "access": <boolean string>
}
```

Where the boolean string is either the string "True" if the user has permission, or "False" if the user does not have permission.

The signature encrypts the base64 url encoded header, concatenated with a '.' and then concatenated with the base64 url encoded payload using the key provided in the key.txt file. This is done using the HS256 algorithm and the HMAC module in python.

The final JSON web token is in the following format:

<base64 url encoded header>.<base64 url encoded payload>.<signature>

Note that the signature is not base64 encoded. Some implementations encode the signature, but we won't for this project.

When a JSON web token is sent in as a parameter for user editing or viewing, you must decode the JSON web token and confirm that the username is correct, the permission is set to True, and the signature matches what is expected. You should use the same key to encrypt and decrypt the JSON web token.

To use the HMAC library, you'll have to convert your python strings to a bytes object, you can do this with python's str.encode() method. You should use the 'utf-8' encoding.

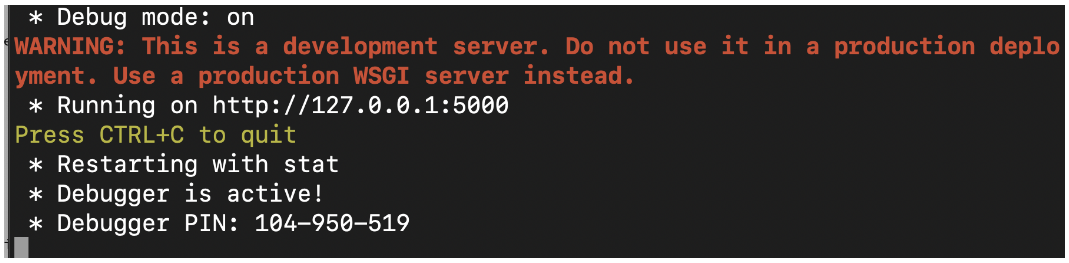
7 Running the Flask Server

Running the flask server is straight forward, from the command line you can run the command:

```
flask run –debug
```

This should start up the server and return something similar to Figure 2. Once you see this, your server is up and running on your localhost on port 5000. If you keep the server running, you can make requests to port 5000 with the appropriate endpoint, and start interacting with your server.

The only requirements that we have for this project is that we must be able to run your server with the 'flask run' command in the outermost directory of your submission. We will not use the '–debug' flag, but you should use it while testing, as it is very helpful. And we must be able to hit the endpoints as we specified above. Whenever we specified a specific module must be used



```
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deplo
yment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 104-950-519
```

Figure 2: Flask Server running

is also a requirement, but everything else in the implementation is up to you.

We expect that every will run their flask server as mentioned above. If you try to run your server through an IDE, then we will not be able to help you debug issues with running your server. When we test, we will run it as mentioned, and suggest you do that whenever you run your server, even for testing.

8 Running the Client

Once the server is up and running, we will run some python scripts that will send HTTP requests to the server and collect the results. The testing scripts are simply python files that will be run. You can run them as you would any other python file. The client requires the python 'requests' module. This is not a built-in module, so you should not use it in your server, but you will need it to run the client.

You should run your server in one terminal, and then run the client/requestor in another terminal while the server is running.

9 Testing and Test Cases

The released test cases with the word 'checkpoint' at the front, are what is expected to pass by the checkpoint submission. These will be required to be passed at the checkpoint submission.

When we run our testing suite, we will be using the same general test cases, but we will have a different key value, different salt, different usernames, etc. So you will not be able to hard code the results based on what is released.

Additionally, we are only releasing part of the testing suite, the rest will not be released. It is your job to read the specifications thoroughly so you understand all of the use scenarios, and write your own test cases to test these other scenarios. We are only expecting you to come up with situational test cases, meaning that we will not have test cases that change trivial things like the types being sent in. Do not waste your time trying to write cases for every small (and overall inconsequential) situation like type mismatches. Focus on situational use cases.

10 Assumptions and Requirements

You can assume that all of the requests will be valid, meaning that we won't send a POST request where a GET request is expected. You can also assume that the types of data will match what we say. For example, do not waste your time worrying about simple data type errors such as "what if an int is sent in but it's supposed to be a string". But, of course, as stated above, there might be other issues with authentication and other things that you will have to handle.

We assume that we can run your server using the: 'flask run' command in the outermost directory of your project.

You are not allowed to use any python libraries that are not built-in, or flask. For a list of built-in python modules, see this link.

We assume that you are running flask 3.X, and python 3.7 or greater. Our machines are running python 3.11.X.

Any software that does not compile or run, will receive an automatic 0 for the project.

If you use a python library/module that is explicitly not allowed, you will receive an ADR for purposely and dishonestly circumventing the project requirements.

11 Hints

I have given some required python modules for a few sections, I highly recommend that you get used to these modules well before the due date, as they can be confusing if you have not dealt with them before.

You can hard code parameters from the post request into your flask server to test, this will make initial development easier so you can quickly see the results. Once you are confident, you can remove the hardcoded values and take the post requests instead.

I will not be teaching most of these python modules, I assume that you will be able to read the documentation and figure out how they work, so starting early will be key. If you come to office hours, we will will help you in how to use the libraries, assuming you have already given it an honest attempt yourself.

12 Released Resources

We have released some sample starter code. You do not have to use it, but you can build off of it if you want. We have also released a key.txt file, and another python file that can create an HTTP query that is sent to the server.

13 Deliverables

Your final deliverable will be a single .zip folder, named with your netid (the part of your email address before the '@msu.edu'). You need to name your outtermost folder with your netid, and then zip it up. It will not work if you only rename the zip file. To check if it is correct, you should unzip your deliverable, and it should create a folder with your netid as the name.

Within this folder you can have your code arranged and structured in any way that you want, as long as it can be run with the command: 'flask run'.

Also in the outer most directory of your submission you need to submit your .sql file that creates your database. More information on this can be seen in Section 5.1.

As a final clarification, you are submitting a single zip folder, and when you unzip the file it creates a directory with the title that is your netid. Inside this directory is only the code needed to run your project.

Do not submit anything extra in your directory, for example, do you not include your python virtual environment. This is thousands of files, and takes a very long time to unzip and might timeout your project running. Be very careful to remove these extra files. You may need to show hidden files in your OS to be able to see if any extra files are present.

14 Checkpoint Submission

There is a mandatory checkpoint submission on October 9th before 10:00pm. In the checkpoint, I will run your project against a few test cases (totaling about 25-30% of the final project). You will need to pass all of these checkpoint tests, or get a 20 point deduction from your final project. We will also run all of our hidden tests against your project and let you know if it passes the checkpoint tests and what tests from the final submission you are failing.

For the checkpoint, you will be expected to have the database setup, the clear endpoint, and user creation done. This is about 25-30% of the entire project.

15 Submission Instructions

For project 1 you will be uploading your zip file directly to handin. See Section 13 for a full description of the deliverables. Your project must be submitted to handin.cse.msu.edu before 10:00pm on October 14th, 2025. No late submissions will be accepted.