Fall 2024 CSE 380
Project 3: Dockerized Microservice

November 15, 2024

---

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.

2 Project Due Date

This project is due on December 5th, 2024, before 10:00pm. Follow all submission instructions at the end of this document. There is a mandatory checkpoint submission due on Tuesday December 2nd before 10:00pm. This is Tuesday instead of Sunday because of the Thanksgiving holiday. This also means that there will be no checkpoint second chance. If you do not pass the checkpoint test cases by the deadline, you will get -20 points on your project score. The dockerization of this project is the most complicated and error prone component of this project. Make sure to start it early. If you wait until the checkpoint deadline day to start, and you run into issues, you will likely not be able to finish it. Please come to office hours early if you think you need help.

3 Project Overview

In this project you will build multiple flask applications, each connected to their own SQLite database (if needed), and each working as a microservice in a larger application. We are building the back end of a document sharing/editing service. This project will use the user management system you built in Project 1, with additions to the database for new functionalities. That means that your project 1 must be mostly complete before you can start with Project 3. This will need the exact same things from Project 1 as Project 2 needed. If your project 1 was working well enough for Project 2, it will be working for this project as well. We will be making a few small changes to the user system from Project 1, but they will be minimal. We will not directly use anything from Project 2, but some of what you have written maybe useful for Project 3. You can

reuse any of your project 2 code, but make sure that you cite in a comment that it was taken from your Project 2, in case it is flagged for plagiarism.

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 Project 3 Specifications

You will implement four microservices all working towards the overall application goal of a document management system. You will have a microservice for user management and authentication/authorization, a microservice for document creation/editing, a microservice for logging, and finally a microservice for document searching. Each microservice will be publicly addressable for requests from a client, but will also contain endpoints used for inter-microservice communication. Each microservice will be containerized using docker, and must communicate through a docker network. Each microservice will have its own database (if it needs one), not one centralized database.

This project will follow the microservices model discussed in lecture, with your flask implementations each being a single microservice that is part of the overall application. Each of your SQLite databases will be run locally for simplicity's sake. In an actual deployment, your database would most likely be on its own dedicated database server.

The user management microservice must accept client requests to create and login users, generate JWTs, and handle permissions. It must also provide authorization information to the other microservices in the project. The logging microservice will log the activity of all of the microservices in the system, and must accept client requests for the log information. The document management microservice should manage creating and editing documents. The searching microservice must be able to return relevant document information to any client that requests them.

As long as your project can be run, and has endpoints as specified, the structure of your project is up to you. The newly released sample code only has a few modifications to what was previously released, relating to a Flask server communicating with another flask server.

I will not provide you with more Docker instruction than has already been discussed in lecture. It will be up to you to read the Docker documentation to set up your application that follows our specification requirements, discussed below in Section 4.10.

4.1 Assumptions on the Project

Anything not directly discussed in the specifications, you can assume is not part of the project (within reason). If there's any confusion, please let me know. For example, I do not discuss user deletion in these specs, that means we will not be handling user deletion in this project.

You only have to handle the errors that I discuss in the specifications. For example, if I do not say that you have to handle the case where a boolean input is actually sent in as an integer, then you do not have to worry about that error case.

4.2 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 implemention, including anything like SQLAlchemy. Failure to use SQLite for the database will result in an automatic zero for the project, and a possible ADR.

You should have a separate database for each microservice that needs it, so for this project you will more than one database, and will have a maximum of four databases, each with their own schema.

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. **It is required for you to have a .sql file that initializes each of your databases. Each database you create must also be named as described in each section below.** I've bolded this portion, as it is different from Project 1, make sure that you follow this.

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 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.

For the final submission, you will have to provide a pdf that has your full SQL schema, as well as a description of your design and the reasoning for the design. You should read this en-

tire 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. Your pdf must be generated using a word processing application (Microsoft Word, Google Docs, etc). If you submit a photo or anything handwritten, you will get zero credit for the pdf.

Your database design will be worth points towards your final submission, and a poor database design will lose you points.

4.3 Your APIs

Below I have described a few required endpoints that you need for this project. You will need to create additional endpoints to facilitate the communication between your microservices. In the same PDF as your database description, you will need to also describe the endpoints you have added to each microservice, including the type of HTTP request it is expecting, the parameters being sent in, and the format of the data being returned.

Additionally, please note that for this project you can use the requests module of python, in addtion to any built-in module of python and flask. This can be used to make the HTTP requests between your various microservices.

4.4 User Management Microservice

This microservice will handle user creation and user login/authorization. The database for this microservice (if you choose to have one) must be named user.db. For the final dockerized version of the user management microservice, it needs to be reached at port 9000 on the localhost.

4.4.1 User Creation

User creation is almost exactly the same as in Project 1. The only thing added is a new POST parameter that says what group the user belongs to. This is a text attribute.

The endpoint for user creation must be '/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
- group: <user group>
- password: <user password>
- salt: <user salt>

Your user creation endpoint needs to return a JSON object that contains a status code, as well as the hashed password. Everything else related to user management and password hashing should be handled the same as in Project 1.

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 codes will be the same as they were for Project 1.

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.

4.4.2 User Login

User login will be almost identical to login from Project 1. The only difference is a slight modification in the JWT returned.

The endpoint for user creation must be '/login'.

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. See Section 4.9 for how the JWT will be different for this project.

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 return will be the same as it was for Project 1.

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.

4.5 Document Management

This microservice will handle document creation and management. The database for this microservice (if you choose to have one) must be named documents.db. For the final dockerized version of the document management microservice, it needs to be reached at port 9001 on the

localhost.

4.5.1 Creating Documents

After a user has logged in, they are able to create and edit documents. This means that the user has to be created, and has also received a JWT after login. The JWT will be sent in with the HTTP header, in exactly the same was as it was for project 2.

The endpoint for document creation must be '/create_document'.

The POST request will contain all the following parameters:

- filename: <document name>
- body: <document body>
- groups: {group1: <group name>, group2: <group name>, etc.}

The filename and body of the document are in text format, and the filename will be unique. Each document will also have at least one group associated with the document. This group will describe which users can edit and view the document. You can assume that the group that the user creating the document belongs to will always be part of the groups sent in with this request. The groups will come in as a JSON string, not as a dictionary. You will need to parse the JSON string to a JSON object in your solution to then access the information.

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 post was successful, and the integer 2 if it was unsuccessful for any reason. In this situation, the only unsuccessful case will be for a JWT not being able to be verified properly.

When a document create request comes in, you should start a fresh document with the given filename. So you should delete the previous document if it exists, or simply overwrite the previous document.

You can write the file locally, even in the final docker container solution. Do not worry about being able to view the file.

Whenever you open a file to create or edit it, please specify in the python 'open' method that the newline character will be '\n'. You can do this by adding 'newline='\n' to the open parameters, after the filename and the file mode.

4.5.2 Editing Documents

Editing a document in our system consists of simply appending some text to the end of the document. You should not auto-append a new line, unless it is sent in with the edit. A user can only edit a document if they are part of a group that has access to the document. You should not be storing any user information in this microservice, so to determine if a user has access to the requested document, you should make a request to the user management microservice to find what group they are a part of. You can set up the endpoints for this purpose however you like.

The endpoint for post viewing must be '/edit_document'.

This endpoint will take a POST request containing the following parameters.

- filename: <filename>
- body: <text to append to document>

You will need to create your own endpoint in the user management microservice to accept a request from the document microservice to provide information about groups to authorize the user to edit the document. The JWT will be sent in with the header, which will only give you the username. You must verify the JWT, then check authorization with the user management microservice before proceeding.

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

The status should be 1 for a successful request, and 2 if the request was unsuccessful for a bad JWT, and 3 if the user's JWT was correct, but they did not have permission for the document.

Whenever you open a file to create or edit it, please specify in the python 'open' method that the newline character will be '\n'. You can do this by adding 'newline='\n' to the open parameters, after the filename and the file mode.

4.6 Document Searching

This microservice will handle file searching. A user can search for the metadata of a document for which they have authorization. The meta data includes the file owner, the filename, who last modified the document, total number of modifications, and a hash of the file. For the final dockerized version of the searching microservice, it needs to be reached at port 9002 on the localhost. If you will use a database for this service it must be named search.db.

The endpoint for user following must be '/search'.

This endpoint will take a GET request containing the following parameter.

- filename: <filename>

You must use the JWT sent in with the HTTP header to determine who is making the request. You must verify with the user management and authorization microservice to determine if this user has authorization to view the document.

The return value will contain a status code, as well as the information about the document. The status should be 1 if the search was successful, 2 if the JWT was not valid, and 3 if the user is not authorized to view that information. If the request is not valid, the value with the 'data' key should be the string 'NULL'.

```
{
    "status": <code>
    "data": {
                "filename": <filename>,
                "owner": <username>,
                "last_mod": <username>,
                "total_mod": <integer of modification>,
                "hash": <hash of the file>
            }
}
```

You will have the filename already sent in, and you will likely need to get the owner from the document management microservice, the hash from the document management microservice, and the last modified and total number of modifications from the logging microservice described below in Section 4.7. This microservice does not need a database, however, you may find a use for one, and that is allowed. As long as you are not storing any redundant data. You should use the hashlib file_digest() method using SHA256 for creating the hash of the file.

You will need to set up the endpoints in the other microservices to facilitate these requests.

4.7 Logging

The logging microservice will keep a log of every successful action taken by the other microservices. Users can request to view the logs for a given file, assuming they have the authorization. You can also view the history of your own user actions. For the final dockerized version of the logging microservice, it needs to be reached at port 9003 on the localhost. The database for the logging service must be named logs.db.

The order of the actions must be maintained, but you can handle that however you see fit. In the logging database you should store when a user is succesfully created, when a user successfully logs in, when a document is created, when a document is edited, and when a document is searched. In all situations, if a request fails for any reason, it should not be logged.

For user creation, you must store the username as well as the fact that this was a user creation event.

For logging in, you must store the username, as well as the fact that this was a login event.

For document creation you must store the filename that was created, as well as the username of the user that created the document. This counts as the first modification for purposes of document searching.

For document editing you must store the filename that was edited, as well as the username of the user that edited the document. Each edit counts as an additional modification for the purposes of document searching.

For document searching, you must store the filename that was searched, as well as the username of the user that requested the document. This does not count as a modification for the purpose of document searching.

The endpoint for viewing a log must be '/view_log'.

This endpoint will take a GET request containing one of the following parameters.

- username: <username>
- filename: <filename>

You must use the JWT sent in with the HTTP header to determine who is making the request. You then must determine if that user has the authorization to view the logs they are requested. A user can view the log for their own username or for a file that they have authorization to edit.

The return value will contain a status code as well as the log information. The status should be 1 if the request was successful, 2 if the JWT was invalid, and 3 if the user does not have authorization to view the logs. If the status is not 1, the data should only contain the string: 'NULL'.

```
{
    "status": <code>
    "data": {
                "<log counter>": <log info in json (format below)>
            }
}
```

The <log counter> key in the data is an integer that starts at 1, and increases sequentially for every new log entry. The data will contain many key value pairs of an integer to the log information. This log count key will always start at 1, regardless of if other actions occurred before this action on a different file or for a different user.

For each log value returned, you will need the following key-value pairs. If there is no filename (user creationg and login events) the value for that pair should be the string 'NULL'.

```
{
        "event": <event type>,
        "user": <username>,
        "filename": <filename>,
}
```

Event type will be 'user_creation' for user creation, 'login' for user login, 'document_creation' for document creation, 'document_edit' for a document edit, and 'document_search' for a document search. The username is the username of the individual who made the successful request, and the filename is the name of the file that was created, edited, or searched, or the string 'NULL' if it is not relevant.

As an example of what the log return will look like, we can imagine a situation as follows:

user1 creates a document named a.txt
user1 creates a document named b.txt
user2 edits a.txt
user2 edits b.txt
user1 edits a.txt
user1 searches for information on a.txt

If we were to then make a request to view the logs for the file 'a.txt' the return in the data would look like:

```
{
    "status": 1
    "data": {
                1: { "event": "document_creation", "user": "user1", "filename": "a.txt"},
                2: { "event": "document_edit", "user": "user2", "filename": "a.txt"},
                3: { "event": "document_edit", "user": "user1", "filename": "a.txt"},
                4: { "event": "document_search", "user": "user1", "filename": "a.txt"}
            }
}
```

The log counter is always an integer that starts at the value 1, and specified that the following log entry is the first for what is returned in this request. And 2 is the second log related to what is being requested, and so on.

4.8 Clearing

Each microservice should be able to accept a request to the endpoint '/clear'. This will tell that microservice to clear everything from it's database and act as if the server was restarted. You can assume that if we call clear on any microservice, then we will call clear on all of the microservices.

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

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

4.9 JSON Web Tokens

The JSON web token will be handled exactly as it was in Project 1. You must use all of the same libraries and methods. The only difference is that the JWT payload will only contain the username, nothing else.

The payload will contain the following:

```
{
    "username": <username>
}
```

4.10 Docker

Your microservice will be dockerized, and we will test your solution as it runs in four docker containers. You should test and make sure your project is working without dockerizing, and then dockerize the project when the functionality is complete.

The ports that each docker container needs to be listening at on the localhost are given in the preceeding sections for each microservice.

We will create a docker network named with your netid (all lowercase). When we deploy your containers, they should all be part of the same docker network so that the containers can communicate with each other.

You will need to create a dockerfile for each of your microservices, and you must build off the latest python base image (as shown in previous lectures). Within this base image, you can install flask and requests, but not other python modules. See previous lecture slides for how to set up your docker file. If you install any non-allowed modules, you will get a zero on the project and a possible ADR.

The dockerfile for the user management service should be named 'Dockerfile.users', for logging it should be named 'Dockerfile.logs', for document management it should be 'Dockerfile.docs', and for document searching it should be 'Dockerfile.search'.

Please note that you are not submitting any docker images, but your source code. In the outermost directory of your submissions should be a file named 'commands.txt' that contains the commands necessary to build and run your docker images. DO NOT have a command to create the docker network, our testing suite will do that already. You should have a command to build each docker image based on the placement of your source code in your submission, as well as the command to run each docker image and attach it to the correct port and the correct docker network. Our testing suite will handle killing the docker images, so you do not have to have this

in your commands.txt file. Each new command should be on a new line in the commands.txt file.

5 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. But, of course, as stated above, there might be other issues with authorization and other things that you will have to handle.

You are not allowed to use any python libraries that are not built-in, or flask. For this project you can also use the requests module, which has been used for the client requests for projects 1 and 2. For a list of built-in python modules, see this link.

We assume that you are running flask 3.X, and the latest version of python from their docker base image.

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.

6 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.

7 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. You should reuse the key.txt file from project 1.

8 Deliverables

Your final deliverable will be a single .zip folder, named with your netid. 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, and with the 'commands.txt' file in the outermost directory. Will we run each command from this directory, so the commands must be specific to your directory setup.

Also in the outer most directory of your submission you need a pdf that gives your overall SQL schema and API information for each of your databases, as well as an explanation of why your database is structured in this way. More information on this can be seen in Sections 4.2 and 4.3.

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, and a pdf explaining your SQL schema.

Do NOT submit a python virtual environment. It will make your submission very large, and could interfere with our testing software.

Do not submit a Docker image, it will be very large, instead, follow the instructions in 4.10 and we will create the image and deal with cleanup in our testing suite.

9 Submission Instructions

For project 3 you will be uploading your zip file directly to D2L. See Section 8 for a full description of the deliverables. Your project must be submitted to D2L before 10:00pm on December 5th, 2024. No late submissions will be accepted.

10 Checkpoint Submission

The checkpoint is required for project 3. 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 tests, or get a 20 point deduction from your final project. We will run all of our tests against your project and let you know if it passes the checkpoint tests and what tests from the final submission you are failing. You should submit your checkpoint with the exact same instructions as the final submission.