CPSC 449 - Section 01: Web Back-End Engineering

Project 2 - Microservices and Read Replication

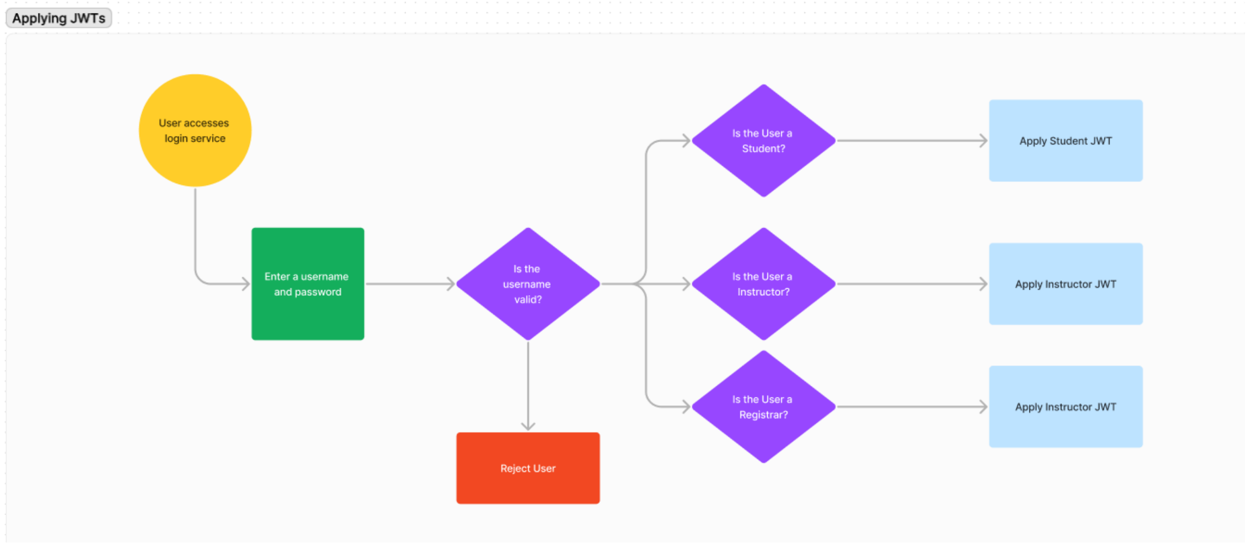
Fall 2023

By: Nguyen Nguyen, Joel Anil John, Logan Langdon, Sanjyot Satvi, Nathan Storm

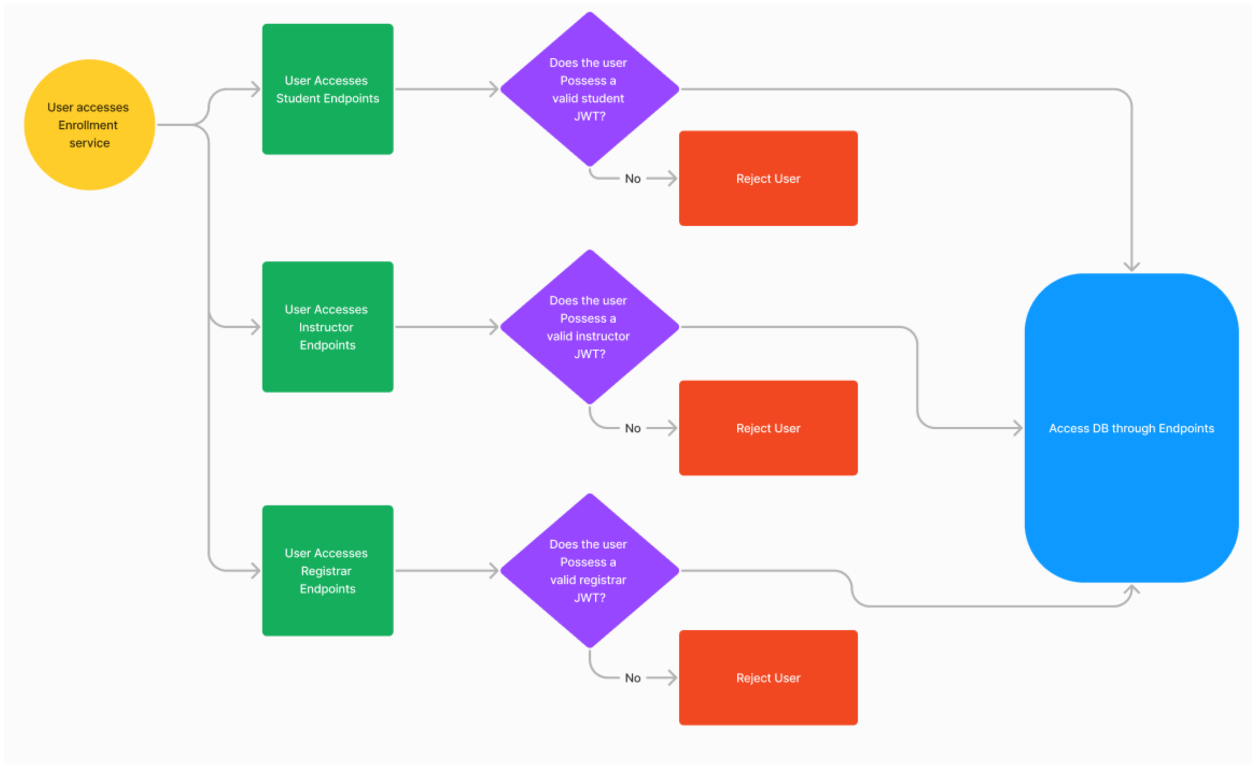
Implementing an Authentication Service

After completing the design for a Restful API for the previous project, our new task was to extend the previous project by including new features, including user authentication.

The main mechanism for authenticating users are JWTs or JSON Web Tokens. With JWTs the API can remain restful through not maintaining any user state and instead allowing the client to hold a token which proves their eligibility to hold a certain role in the system. Previously, any user could access any api regardless of the intended role for that endpoint. Now, each user would have to first go through the login service API in order to gain access to the enrollment service. Pictured below is a sequence diagram of the flow of user access to the API endpoints.



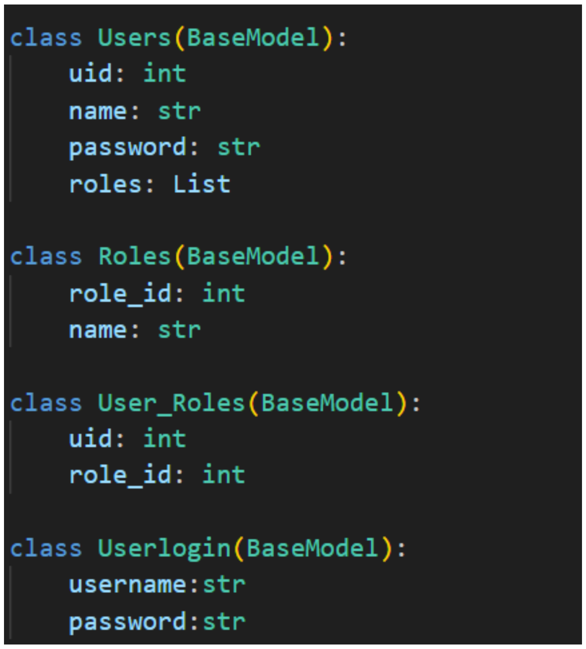
Next, when the user attempts to access the enrollment service API, the user's JWT determines whether they are able to access given endpoints. The logic is represented below.


User accesses Enrollment service
User Accesses Student Endpoints
Does the user Possess a valid student JWT?
No
Reject User
User Accesses Instructor Endpoints
Does the user Possess a valid instructor JWT?
No
Reject User
User Accesses Registrar Endpoints
Does the user Possess a valid registrar JWT?
No
Reject User
Access DB through Endpoints

As with the previous project, the login service must be restful, which means it cannot store any state which determines its behavior and all such information should be stored in its own database. To do this, we created a new database file and used sqlite3 to instantiate it in a db file. The contents of this database includes users which each have their own username and password which are to be stored upon registration and verified in the case of a login. The registration of a user involves hashing their entered password and the hashing function is also used to read the password for verification on login. The login service database and the enrollment service database were created to be decoupled, only sharing the JWT from the initial login but otherwise not communicating whatsoever.

Login Service Schema

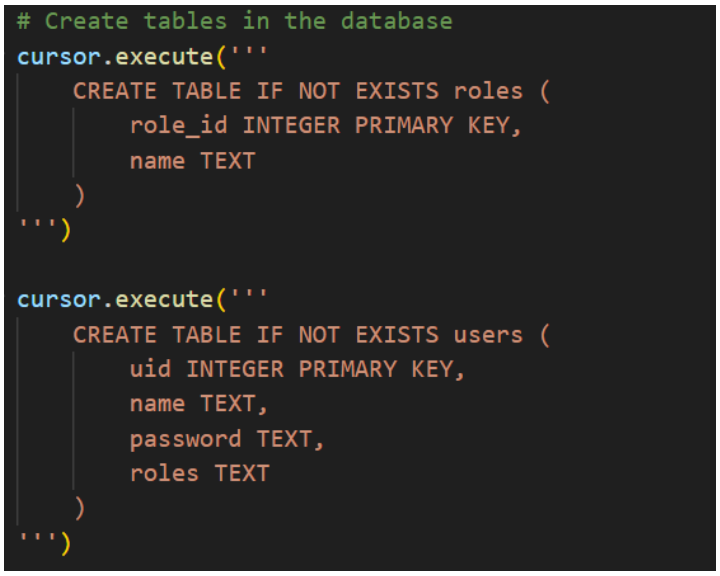
Only four models were necessary for creating the database schema, which are shown below:



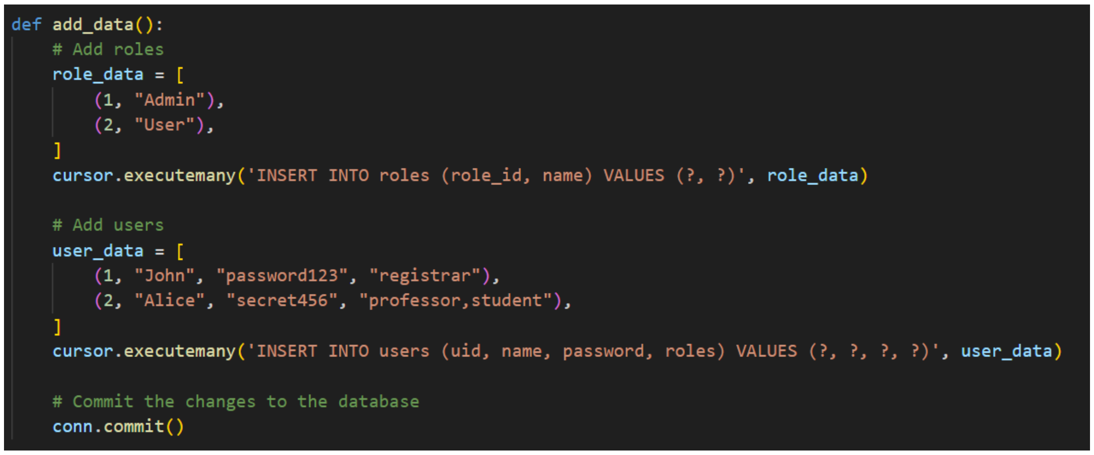
```python
class Users(BaseModel):
    uid: int
    name: str
    password: str
    roles: List

class Roles(BaseModel):
    role_id: int
    name: str

class User_Roles(BaseModel):
    uid: int
    role_id: int

class Userlogin(BaseModel):
    username:str
    password:str
```

There was some disagreement about the layout of the schema, as originally the role_id belonged to a given user entry in the table. However, we found that by separating the roles, user roles, and user login information into their own tables, we were able to have a more normalized database that was easier to query.

Rather than populating a sql file, we used python code to generate the tables and populate them. Each of these steps are documented in the populate.py file and are shown below:



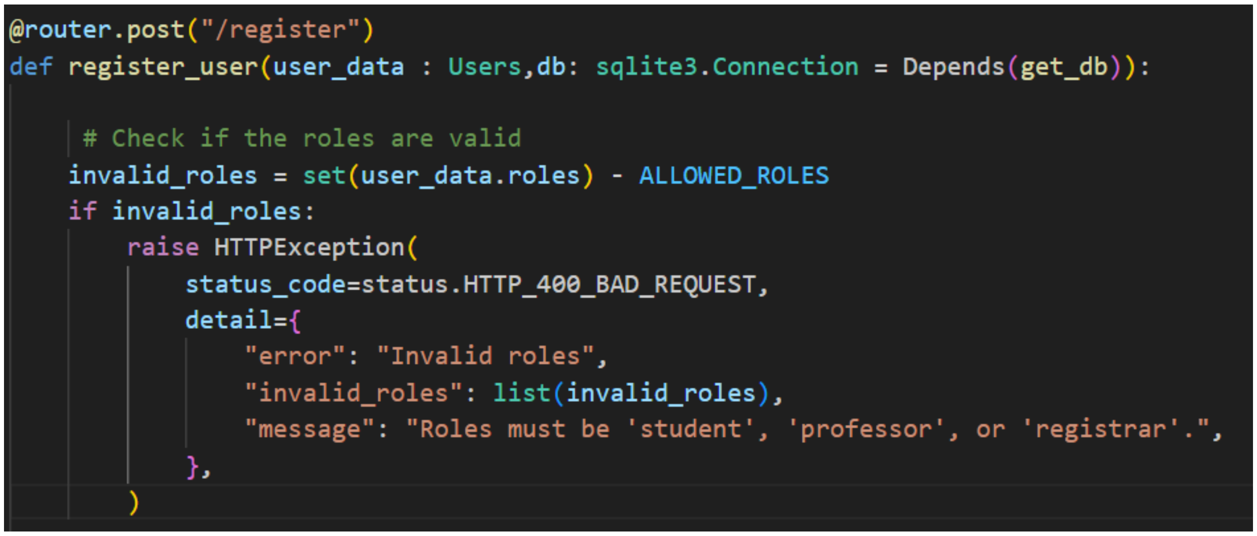
```python
# Create tables in the database
cursor.execute('''
    CREATE TABLE IF NOT EXISTS roles (
        role_id INTEGER PRIMARY KEY,
        name TEXT
    )
''')

cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        uid INTEGER PRIMARY KEY,
        name TEXT,
        password TEXT,
        roles TEXT
    )
''')
```



```
def add_data():
    # Add roles
    role_data = [
        (1, "Admin"),
        (2, "User"),
    ]
    cursor.executemany('INSERT INTO roles (role_id, name) VALUES (?, ?)', role_data)

    # Add users
    user_data = [
        (1, "John", "password123", "registrar"),
        (2, "Alice", "secret456", "professor,student"),
    ]
    cursor.executemany('INSERT INTO users (uid, name, password, roles) VALUES (?, ?, ?, ?)', user_data)

    # Commit the changes to the database
    conn.commit()
```

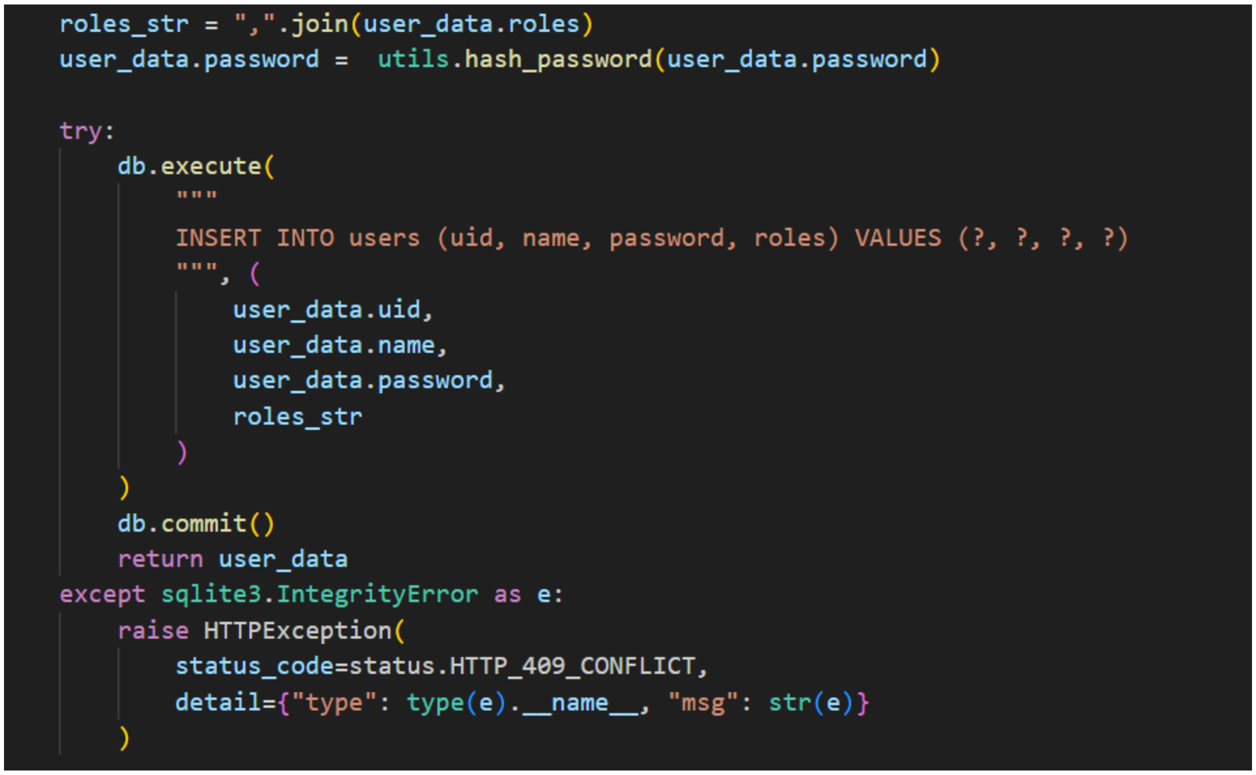
Login Service Endpoints

The first endpoint serves the functionality of registering a new user with a specified role:



```
@router.post("/register")
def register_user(user_data : Users,db: sqlite3.Connection = Depends(get_db)):

    # Check if the roles are valid
    invalid_roles = set(user_data.roles) - ALLOWED_ROLES
    if invalid_roles:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail={
                "error": "Invalid roles",
                "invalid_roles": list(invalid_roles),
                "message": "Roles must be 'student', 'professor', or 'registrar'.",
            },
        )
```

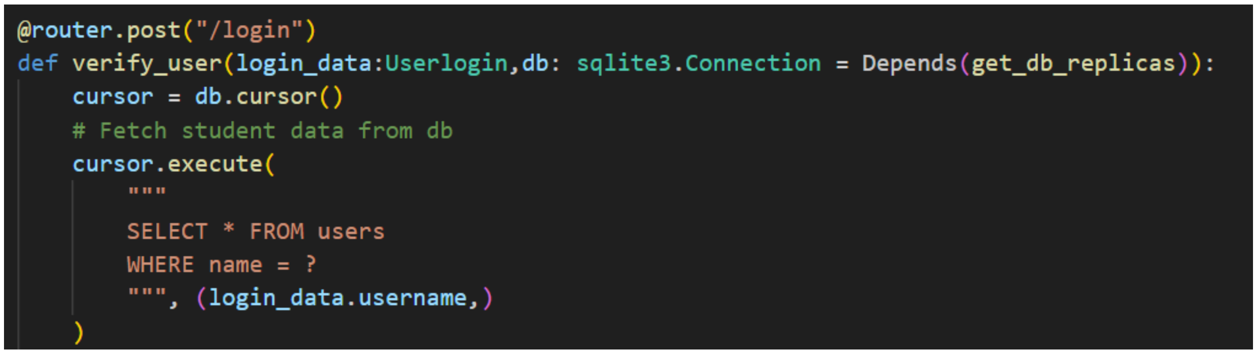
First, we check the specified roles against the set of allowed roles to determine if the chosen set is not allowed. If this is the case, a bad request is raised as the user has not input a correct role out of the set of roles.



```python
    roles_str = ",".join(user_data.roles)
    user_data.password =  utils.hash_password(user_data.password)

    try:
        db.execute(
            """
            INSERT INTO users (uid, name, password, roles) VALUES (?, ?, ?, ?)
            """, (
                user_data.uid,
                user_data.name,
                user_data.password,
                roles_str
            )
        )
        db.commit()
        return user_data
    except sqlite3.IntegrityError as e:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail={"type": type(e).__name__, "msg": str(e)}
        )
```

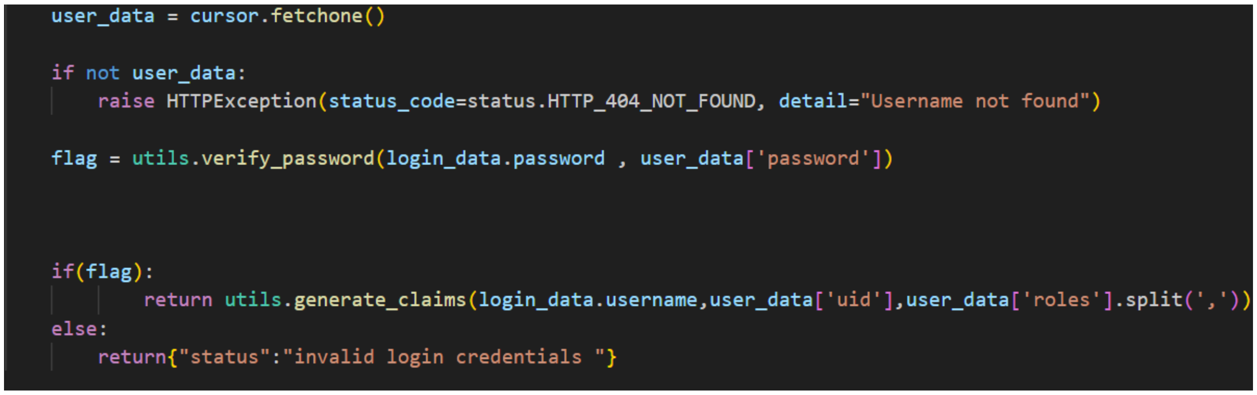
Next, we concatenate all chosen roles into a string and create a hashed version of the password the user input. Finally, we attempt to add the new user to the database, raising a 409 conflict upon a conflicting id, username or password being input.

For the second and final endpoint, it is responsible for login attempts with existing users:



```python
@router.post("/login")
def verify_user(login_data:Userlogin,db: sqlite3.Connection = Depends(get_db_replicas)):
    cursor = db.cursor()
    # Fetch student data from db
    cursor.execute(
        """
        SELECT * FROM users
        WHERE name = ?
        """, (login_data.username,)
    )
```

Initially, a select statement is run to find whether the given name exists in the database. It does not need to be exception handled as the value returned from the select contains all the necessary information to handle the situation.

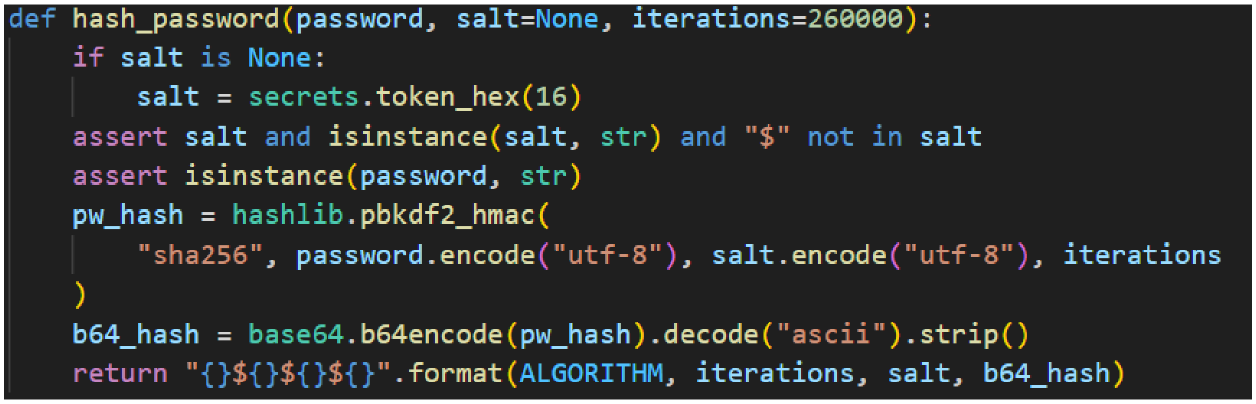


```
    user_data = cursor.fetchone()

    if not user_data:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Username not found")

    flag = utils.verify_password(login_data.password , user_data['password'])



    if(flag):
            return utils.generate_claims(login_data.username,user_data['uid'],user_data['roles'].split(','))
    else:
        return{"status":"invalid login credentials "}
```

Next, if the user data was not able to be found we raise an exception detailing the issue. Otherwise, the password is verified and the user receives the claims generated from their valid password. If their password was incorrect, however, they are rejected with a message.

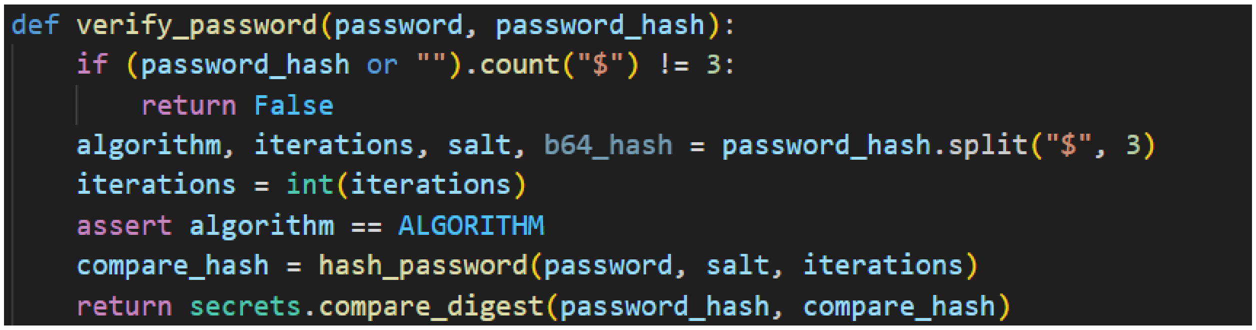
Hashing Functions

In the above functions, these three utility functions were used to secure and validate passwords and create JWTs for the end user.



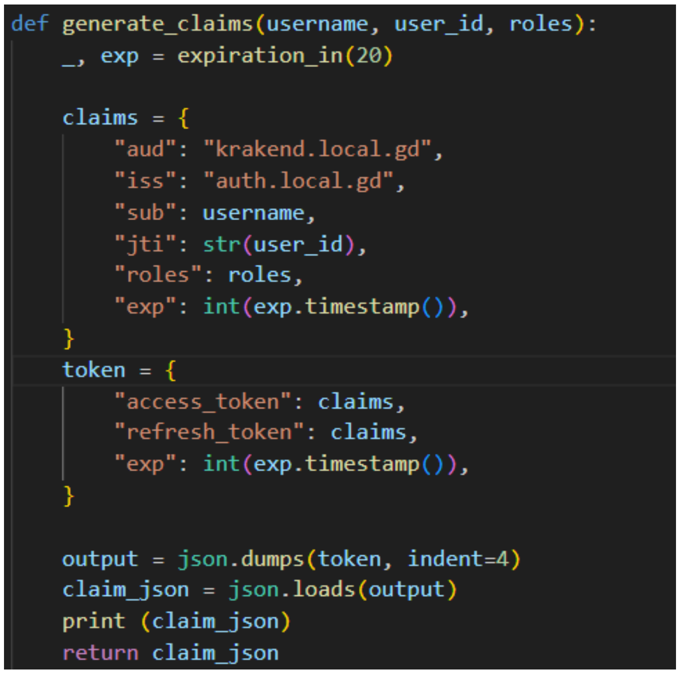
```
def hash_password(password, salt=None, iterations=260000):
    if salt is None:
        salt = secrets.token_hex(16)
    assert salt and isinstance(salt, str) and "$" not in salt
    assert isinstance(password, str)
    pw_hash = hashlib.pbkdf2_hmac(
        "sha256", password.encode("utf-8"), salt.encode("utf-8"), iterations
    )
    b64_hash = base64.b64encode(pw_hash).decode("ascii").strip()
    return "{}${}${}${}".format(ALGORITHM, iterations, salt, b64_hash)
```

In the above, a salt is created to be used in the hashing process first. Once the hashing function is complete, it is formatted into a base64 bitstring and returned with the number of iterations and the salt number.



```
def verify_password(password, password_hash):
    if (password_hash or "").count("$") != 3:
        return False
    algorithm, iterations, salt, b64_hash = password_hash.split("$", 3)
    iterations = int(iterations)
    assert algorithm == ALGORITHM
    compare_hash = hash_password(password, salt, iterations)
    return secrets.compare_digest(password_hash, compare_hash)
```

On the opposite end, to verify a password, the password is hashed and compared to the existing password hash to determine if it is the correct password.

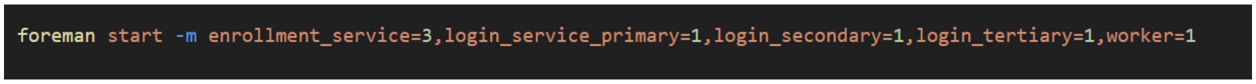


```python
def generate_claims(username, user_id, roles):
    _, exp = expiration_in(20)

    claims = {
        "aud": "krakend.local.gd",
        "iss": "auth.local.gd",
        "sub": username,
        "jti": str(user_id),
        "roles": roles,
        "exp": int(exp.timestamp()),
    }
    token = {
        "access_token": claims,
        "refresh_token": claims,
        "exp": int(exp.timestamp()),
    }

    output = json.dumps(token, indent=4)
    claim_json = json.loads(output)
    print (claim_json)
    return claim_json
```

Finally, to generate JWT claims, a set of claims and a set for the token are generated and formatted into JSON to be returned.

Enrollment Service Load Balancing

To ensure that the server made use of Krakend's built-in load balancing features, we created a shell script to store the command configuration to run foreman start with the correct input information:

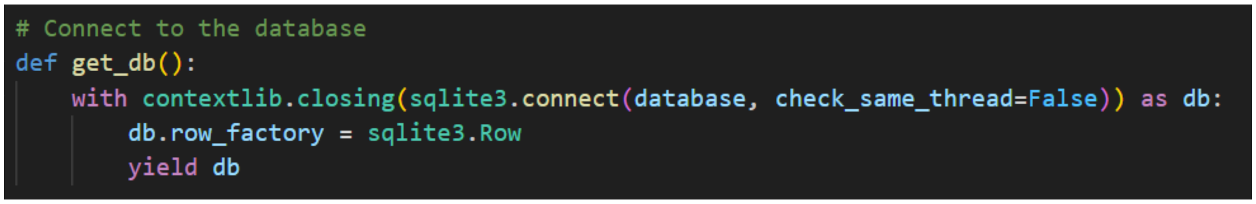


```
foreman start -m enrollment_service=3,login_service_primary=1,login_secondary=1,login_tertiary=1,worker=1
```

This includes the quantity for each of the databases as well as the number of available workers as input to foreman.

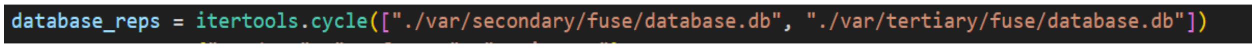
Database Replication

Now that the database has been replicated into a primary and two replicas, the get_db function we used to access the database in our original API will have to be changed to be able to access these tertiary databases:

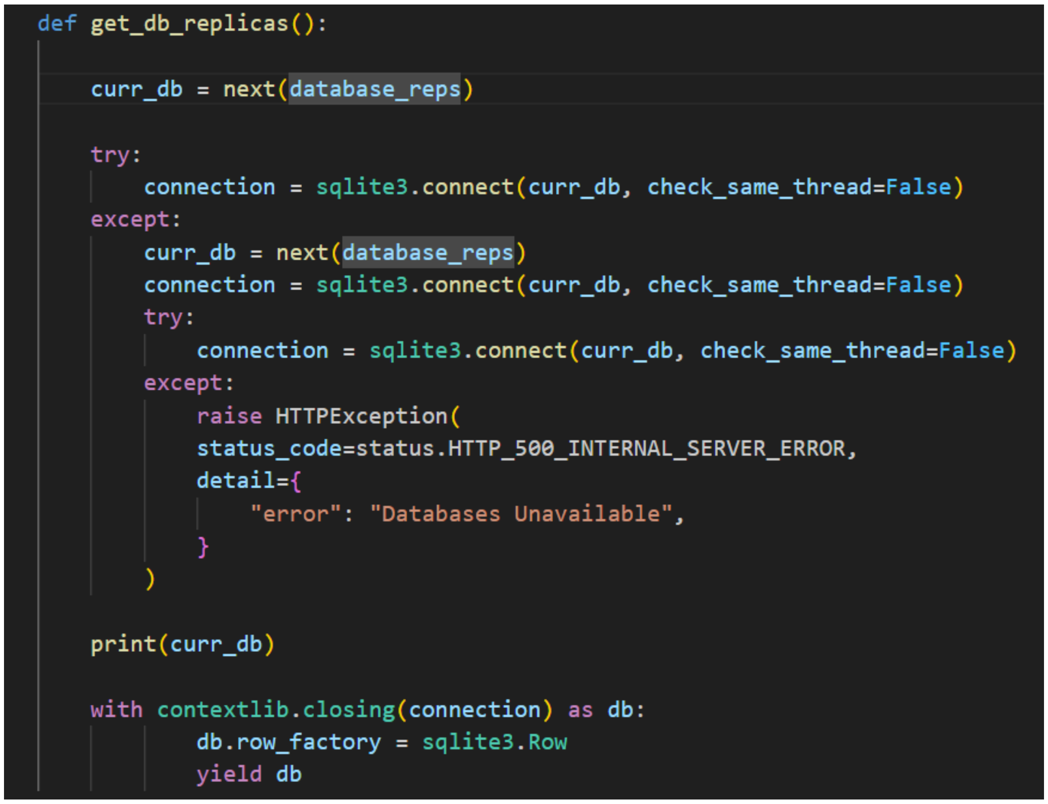


```python
# Connect to the database
def get_db():
    with contextlib.closing(sqlite3.connect(database, check_same_thread=False)) as db:
        db.row_factory = sqlite3.Row
        yield db
```

The original get_db was only concerned with the primary db, and so it only attempted to open the primary database. For endpoints that may accept data from the secondary or tertiary, the definition changed to:



```python
database_reps = itertools.cycle(["./var/secondary/fuse/database.db", "./var/tertiary/fuse/database.db"])
```



```python
def get_db_replicas():

    curr_db = next(database_reps)

    try:
        connection = sqlite3.connect(curr_db, check_same_thread=False)
    except:
        curr_db = next(database_reps)
        connection = sqlite3.connect(curr_db, check_same_thread=False)
        try:
            connection = sqlite3.connect(curr_db, check_same_thread=False)
        except:
            raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail={
                "error": "Databases Unavailable",
            }
        )

    print(curr_db)

    with contextlib.closing(connection) as db:
            db.row_factory = sqlite3.Row
            yield db
```

Now, rather than immediately creating a connection, we see if the connection is possible first by accessing the next database that wasn't accessed last time the endpoint was reached. If it is, the connection is used below and yielded. Otherwise, the next database is attempted, and if that fails an HTTPException is raised to notify the user that the databases are both unavailable.

Retrospective

Having completed this project, we were able to gain a better perspective of the techniques used to benefit scalability, namely the use of load balancing, as well as the necessary strategies to authenticate users for a Restful API. Additionally, we were able to expand our understanding of scaling the number of database replicas by using a third replica and allowing our existing endpoints to cycle through connections between each of them. Finally, making use of existing hashing functions to hash and verify passwords was an interesting foray into the practical applications of cybersecurity principles.