**Research Labs Inventory**

Lizzett Tapia

**database subsystem final report**

REVISION – Final

5 December 2024

Subsystem final report

for

Research Lab Inventory

Prepared by:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Author Date

Approved by:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Project Leader Date

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

John Lusher, P.E. Date

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

T/A Date

**Change Record**

| **Rev.** | **Date** | **Originator** | **Approvals** | **Description** |
| --- | --- | --- | --- | --- |
| **1** | 12/05/2024 | Lizzett Tapia |  | Final Release |

**Table of Contents**

[**1. Subsystem Introduction 6**](#_heading=h.b3lx0f9n51qk)

[**2. Database Creation and Design 6**](#_heading=h.xjsdosn6crsg)

[2.1. Relational & Non-Relational Databases 6](#_heading=h.s35hhws85p3o)

[2.2. Implementation of Relational Design 6](#_heading=h.pe0n40luss8y)

[**3. Implementation 13**](#_heading=h.7h37yim292jm)

[3.1. Transitioning to a Hosted Server 13](#_heading=h.46t29gddgtrt)

[3.2. Data Migration with SQLAlchemy 13](#_heading=h.fz050gvj014c)

[**4. Database Validation 15**](#_heading=h.vtrrh5puvwg3)

[4.1. Query Validation 15](#_heading=h.21o99dt785nm)

[4.2. Connection Validation 16](#_heading=h.yi0kvts1lze3)

[**5. Conclusion 16**](#_heading=h.ka4zp0bujtrv)

[**6. References 17**](#_heading=h.kyxpdv6xrk4g)

**List of Tables**

No table of figures entries found.

**List of Figures**

[Figure 1: Python code for Users Table 8](#_heading=h.vetgl9bidkto)

[Figure 2: Continued Python code for Users Table 8](#_heading=h.72ud3ogptgmg)

[Figure 3: Users Table 9](#_heading=h.rlnjmal5yln1)

[Figure 4: Category Table 10](#_heading=h.p5u3pdn82t79)

[Figure 5: Supplier Table 10](#_heading=h.v9cpjbjr65t5)

[Figure 6: Items Table 11](#_heading=h.nkp6q7t7d56d)

[Figure 7: Items Table Continued 12](#_heading=h.fo6yiq5dnvq6)

[Figure 8: Orders Table 13](#_heading=h.nd3petsxu21w)

[Figure 9: SQLAlchemy Migration Code 15](#_heading=h.elaf551sh93a)

[Figure 10: SQLAlchemy Migration Code Continued 15](#_heading=h.vl66atpyksba)

[Figure 11: Query Validation 16](#_heading=h.rvrqylgwz4d5)

[Figure 12: Query Validation 17](#_heading=h.vidlgzug6ui7)

[Figure 13: Establishing Proper Connection 17](#_heading=h.b9gcs8u61752)

# Subsystem Introduction

The database subsystem serves as the main repository where all the inventory data is stored. It helps manage important information such as user details, supplier details, inventory stock, and orders. The database can be seen as the foundation of the research lab inventory tracker, as it enables secure data storage and ensures that there is consistency between all interfaces. This subsystem handles all backend data operations, which includes any sort of data update or retrieval.

# Database Creation and Design

## 2.1. Relational & Non-Relational Databases

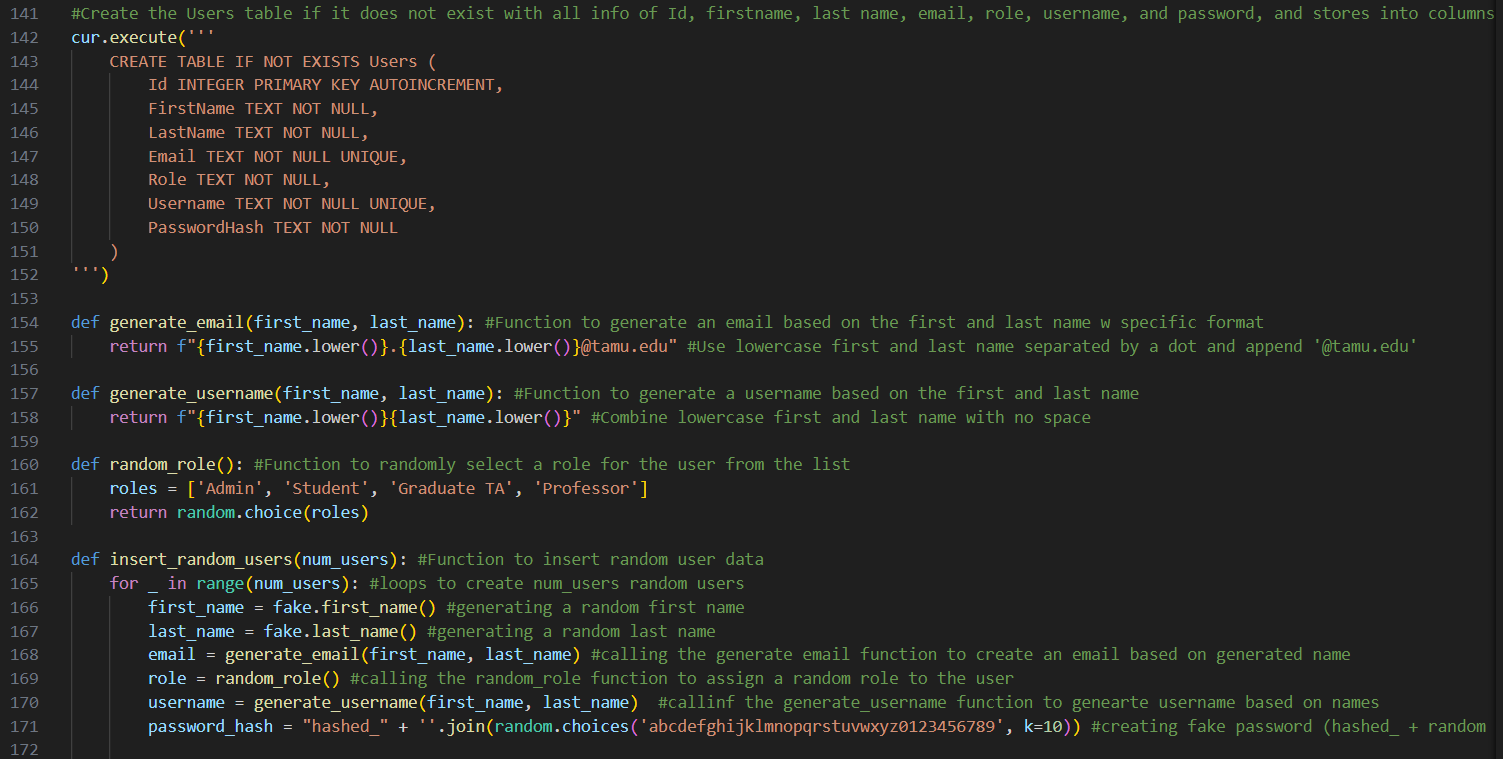
When it comes to creating and designing a database, an important decision that must be determined is whether the data falls under a relational (SQL) or non-relational (NoSQL) model. Relational databases, such as MySQL and SQLite, use structured schemas to help organize the data into tables. This type of model works best when the data needs to be kept strict and consistent, show clear relationships, and overall be able to handle all sorts of data queries. Non-relational databases are simply the opposite. These models are schema-less and work with unstructured data. For this research lab inventory tracker, the data structure falls under a relational model. All the data sets created and used are interconnected within each other. This can be seen when users place item orders, or items belonging to specific categories and suppliers. Overall, working with a relational database helps establish data consistency and maintain system reliability.

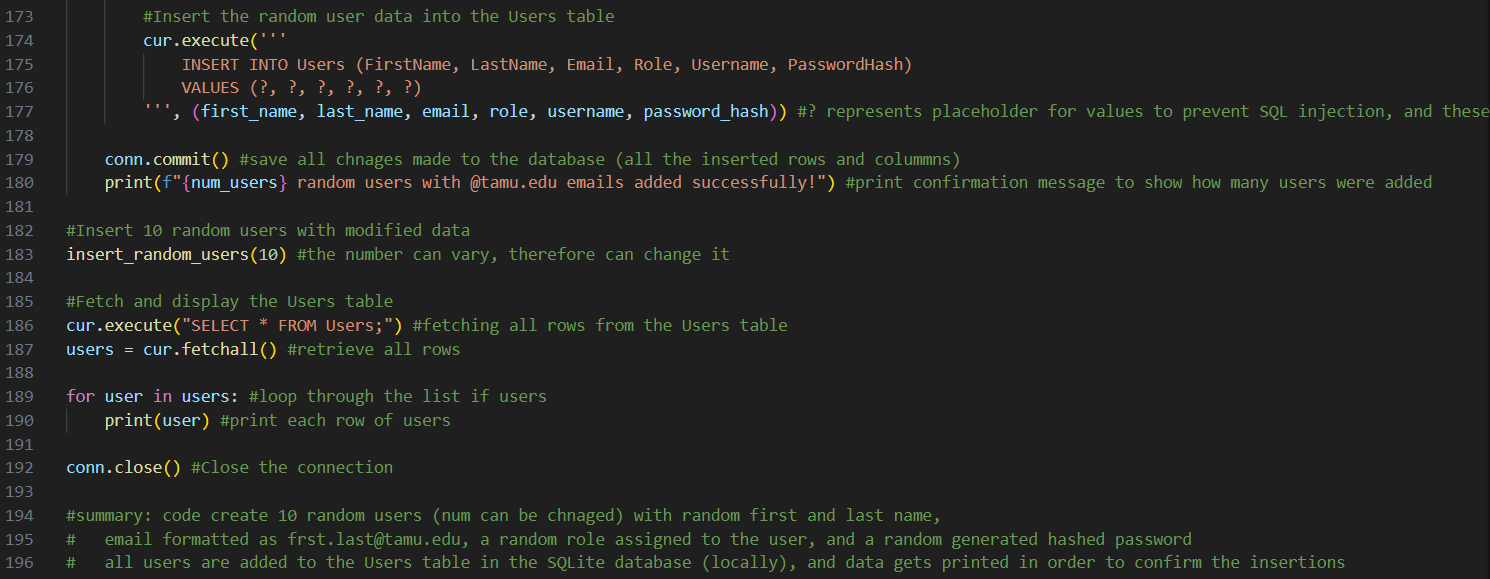
## **2.2. Implementation of Relational Design**

To begin implementing this database relational model, five data tables were created: *Users, Suppliers, Category, Items, and Orders.*

* *Users*: Data table to store user information, such as name, email, role, username, and hashed password.
* *Category*: Data table to help group items into categories like *Basic Electronic Component* or *Power Component*.
* *Items*: Data table to help track inventory details for each item, such as the quantity and location.
* *Suppliers*: Data table to keep track of vendor contact information in case a user wants to place a specific order.
* *Orders*: Data table to log transactions, which helps link users to the items that they ordered.

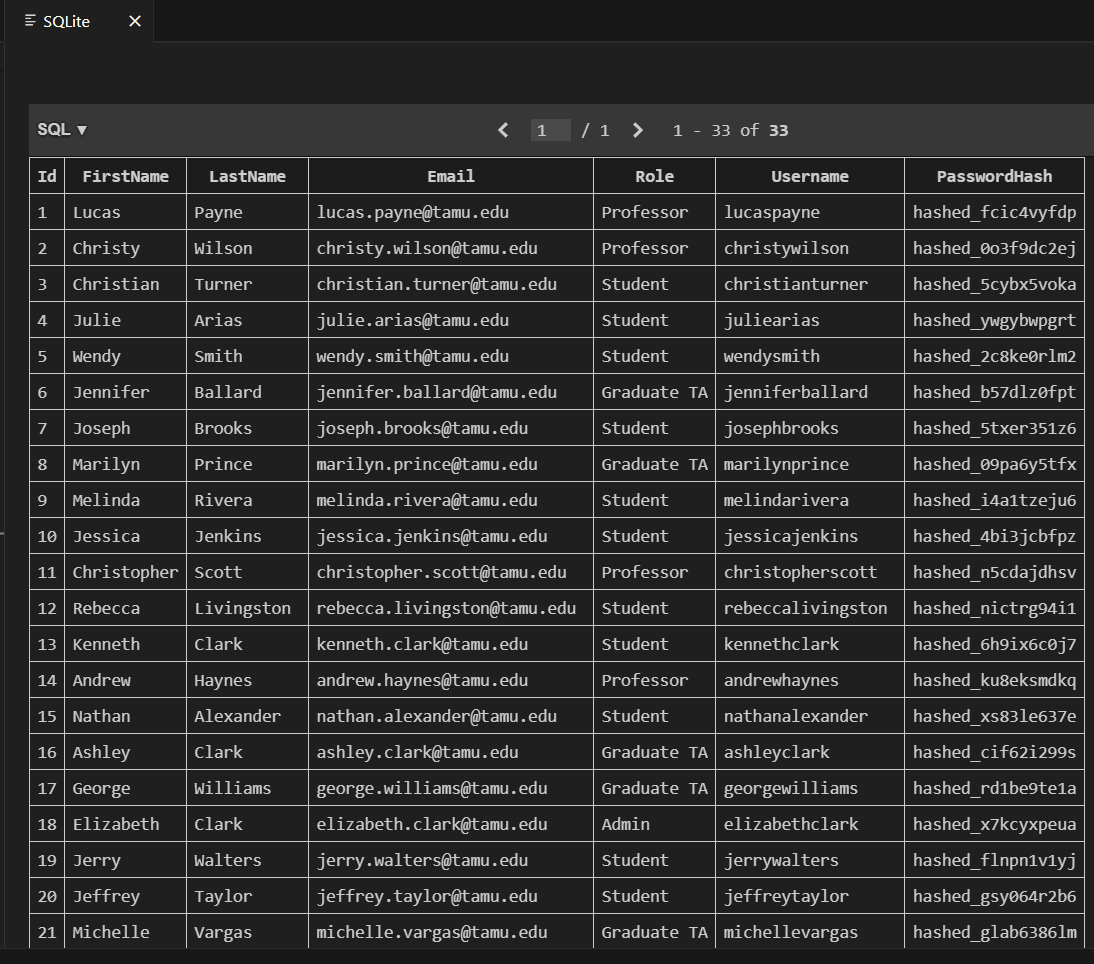
Instead of manually executing SQL commands line by line and one by one, I used Python and SQLite to help with the creation of the database. Using Python is more efficient, as these tables are able to get populated each time the code is run. For example, with the *Users* table, it would have been a hassle to insert user data one by one. It would basically be never ending. Using the *CREATE TABLE* and *INSERT* commands in Python automated the operations quicker and overall helped avoid any potential human errors that could have occurred. The *Faker* library and the *Random* module were used to help generate random realistic data. A similar concept and approach was applied to the other data tables. The screenshot below shows a portion of the Python code used to define and create the *Users* table.

**Figure 1:** Python code for *Users Table*



### Figure 2: Continued Python code for *Users Table*

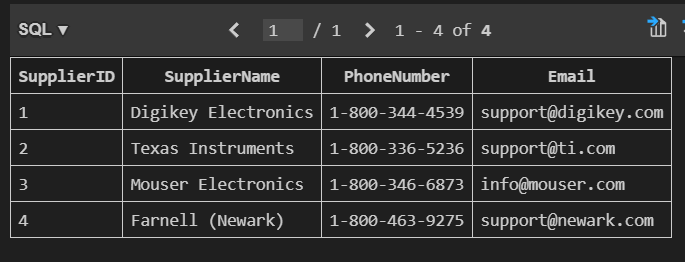
Once the code gets run, data gets populated into the data table. The number of data added can be changed, as seen on line 183. The screenshot below shows the local *Users* table populated with user information.



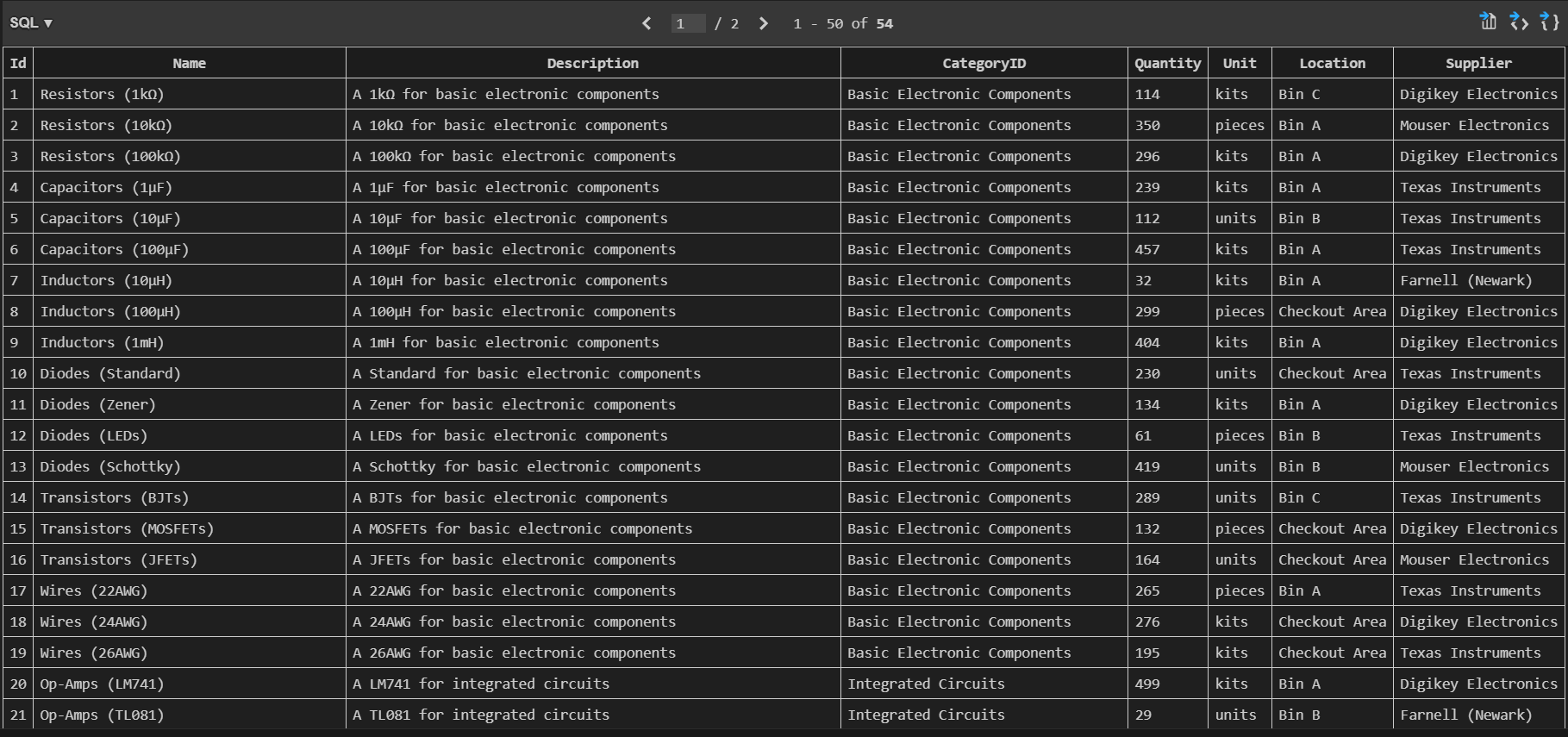
### Figure 3:*Users Table*

Like previously mentioned, a similar approach was done for the creation of the other data tables. The screenshots below show the data tables that got created with the Python code within SQLite. It is important to keep in mind that we can keep on populating the data tables with as much information as we’d like, as long as the code gets run. Same thing applies with deleting data or pulling certain information, one simply has to run SQL query commands.

### Figure 4:*Category Table*

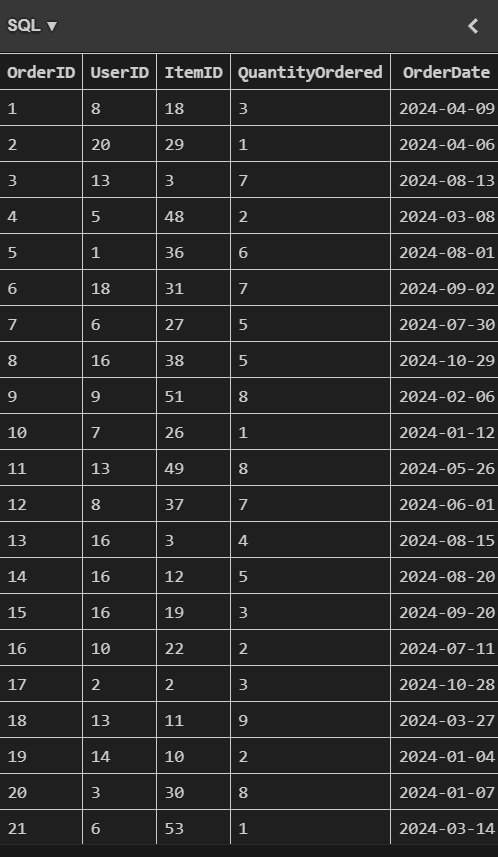
**

### Figure 5: *Supplier Table*

**

### Figure 6:*Items Table*

### Figure 7:*Items Table Continued*

**

### Figure 8:*Orders Table*

Overall, this approach helped make sure that the structure for the tables was properly defined, with very important constraints such as unique fields, ID’s, and primary keys. Using Python helped provide a seamless workflow for both debugging and testing. It helped keep the goal in check: accuracy and consistency.

# Implementation

The database subsystem was initially created and implemented locally using SQLite. However, a hosted server was eventually needed in order for this database to work and be integrated within the other subsystems for ECEN 404. Transitioning to a hosted server environment meant I would be working with MySQL, which is a bit different from SQLite.

## **3.1. Transitioning to a Hosted Server**

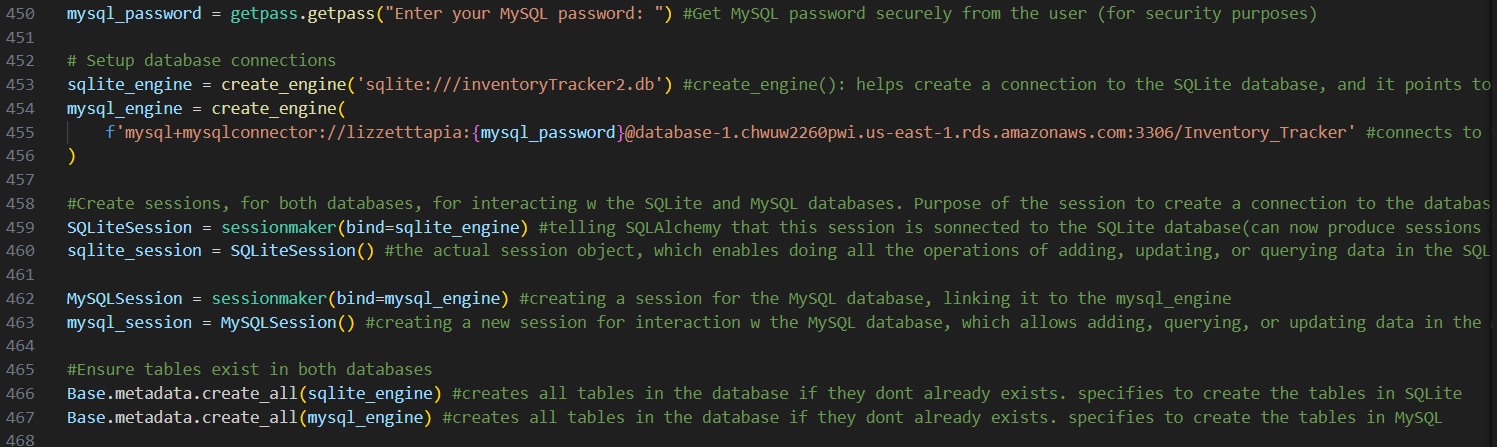
Hosting this database on a cloud server would better support the integration within the mobile application and website. MySQL was chosen due to its compatibility with relational schemas. In order to host the MySQL database, an Amazon Web Services (AWS) Relational Database Service (RDS) was configured. This AWS instance was the best option out of all other instances out there due to the fact that it is free for the first twelve months. It was also highly recommended to me by an ECEN 403 TA. This process involved setting up the instance, making sure that the AWS RDS version and MySQL Workbench were both the exact same version, installing appropriate security and database parameters, and establishing a secure working connection between the local SQLite environment and the MySQL cloud server.

## **3.2. Data Migration with SQLAlchemy**

Considering the fact that I had been working with SQLite, and in order to migrate all the data into the MySQL database server, it meant that I would have to switch up the Python code due to slight differences in syntax. Rather than having to go back and retype the code, I worked with SQLAlchemy. SQLALchemy is an Object Relational Mapper (ORM) that allows us to translate python classes and python objects to database tables and database entries. SQLAlchemy made sure that data tables and relationships were accurately recreated in MySQL. This migration process involved using the *create\_engine* function in order to help define any necessary connections between SQLite and MySQL databases. I wrote another Python script, *main.py*, that helped transfer this data. Important information from my Amazon Web Services RDS account was needed, such as the endpoint, port, and password. This connection helped enable real-time interaction with the local database and the hosted database. Any changes done locally would be seen in the cloud. The screenshot below shows portions of the *main.py* code.



### Figure 9:*SQLAlchemy Migration Code*



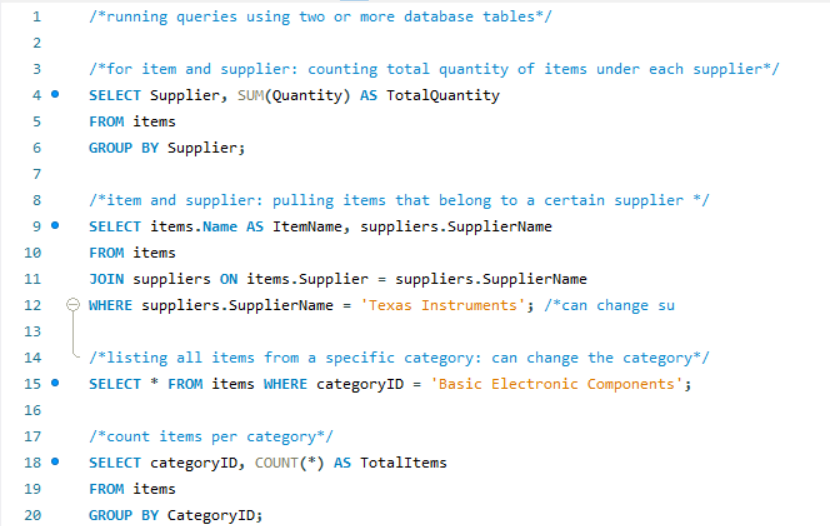
### Figure 10:*SQLAlchemy Migration Code Continued*

# Database Validation

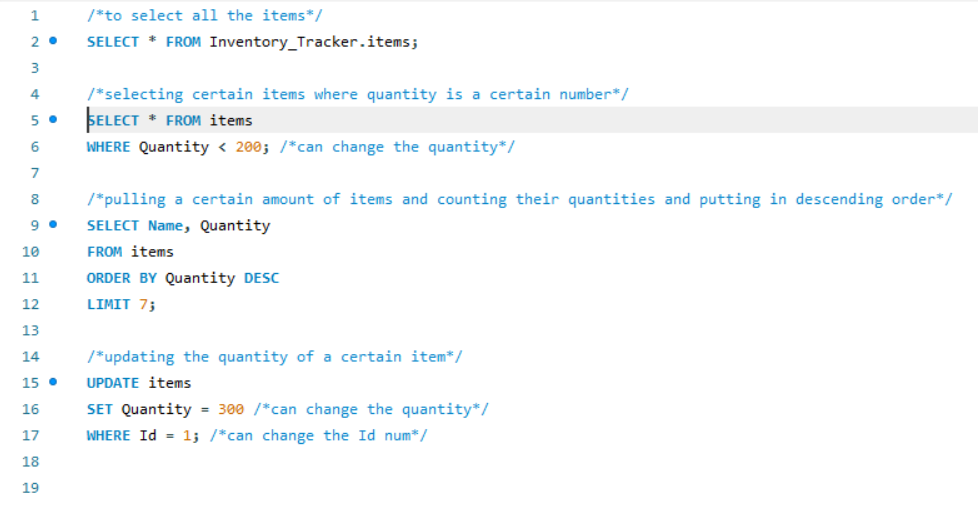
Validation was an important step in making sure that the database subsystem met the requirements outlined. The validation process involved testing the functionality and security of the database both locally and on the cloud. This involved running different queries in order to test different data operations, verifying connections between data tables, and ensuring password protection.

## 4.1. Query Validation

Query validation was done by retrieving specific data from the tables, such as counting the total orders placed by each user, finding items ordered along with the quantity and date, and pulling items that have less than a certain quantity. The screenshot below shows a few of the queries done on the cloud, validating functional testing. Many more were done and they successfully worked, such as inserting data and deleting data.



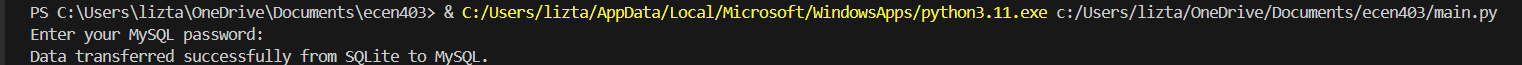
### Figure 11:*Query Validation*



### Figure 12:*Query Validation Continued*

## 4.2. Connection Validation

Establishing a proper connection between SQLite and MySQL was also very crucial. In order for any changes made locally to also be seen on the cloud, *main.py* needs to properly run. In order for it to properly run, the password to the cloud server must be typed in the terminal. If properly inputted, all changes will be made and be visible. If not, then the code simply will not run. This overall helps establish a proper secure connection.



### Figure 13:*Establishing Proper Connection*

# Conclusion

The database subsystem helps serve as the backbone for this research lab inventory tracker. It is able to provide a proper foundation for all the necessary data. Through the design and implementation of a relational database model, this database maintains data consistency and integrity. Transitioning from a local SQLite environment to a proper AWS cloud environment helps enable proper integration with the other subsystems that will eventually be done. Various validation techniques confirmed the functionality, reliability, and security of the database.

# References

[1]“Basic Use — SQLAlchemy 1.3 Documentation,” *docs.sqlalchemy.org*. <https://docs.sqlalchemy.org/en/13/orm/extensions/declarative/basic_use.html>

[2]Prosenjeet Shil, “SQLAlchemy Basics (2/2): The Declarative Mapping Way,” *Medium*, May 30, 2024. https://medium.com/@prosenjeetshil/sqlalchemy-basics-2-2-the-declarative-mapping-way-c9729c102f91 (accessed Dec. 06, 2024).

[3]Tutorialspoint, “SQLite - Python - Tutorialspoint,” *www.tutorialspoint.com*. https://www.tutorialspoint.com/sqlite/sqlite\_python.htm

‌[4]D. to, “Migrate SQLite Database to MySQL with Python,” *YouTube*, Sep. 17, 2023. https://youtu.be/Qv02PxiK938?si=UMBbcPYoTVCGOo-I (accessed Dec. 06, 2024).

‌

‌

‌