Project Report: Secure Database-as-a-Service System

**Team-20: Members**

Sai Meghna Ambati (811256385)

Lakshmi Sathwik Challa (811281131)

Raghunath Male (811257226)

**Instructor: Xu Lei**

Table of Contents

[1. Introduction 1](#_Toc152739985)

[Background 1](#_Toc152739986)

[Objectives 1](#_Toc152739987)

[2. Project Overview 1](#_Toc152739988)

[System Architecture 1](#_Toc152739989)

[Technologies Used 1](#_Toc152739990)

[Database Schema 2](#_Toc152739991)

[3. Project Requirements 2](#_Toc152739992)

[Basic Setup 2](#_Toc152739993)

[Security Features 2](#_Toc152739994)

[4. Implementation Details 2](#_Toc152739995)

[User Authentication 2](#_Toc152739996)

[Access Control Mechanism 3](#_Toc152739997)

[Query Integrity Protection 4](#_Toc152739998)

[Data Confidentiality Protection 4](#_Toc152739999)

[Faker for Data Generation 4](#_Toc152740000)

[Flask for Web Application 4](#_Toc152740001)

[5. Testing and Evaluation 4](#_Toc152740002)

[Testing Methodology 4](#_Toc152740003)

[Evaluation of Security Features 4](#_Toc152740004)

[6. Challenges Faced 5](#_Toc152740005)

[Challenge 1: Implementation of Fine-Grained Access Control 5](#_Toc152740006)

[Challenge 2: Ensuring Data Confidentiality without Information Leakage 5](#_Toc152740007)

[Challenge 3: Scalability for Large Datasets 5](#_Toc152740008)

[Challenge 4: Balancing Query Integrity Protection and Performance 6](#_Toc152740009)

[7. Conclusion 6](#_Toc152740010)

[Achievements 6](#_Toc152740011)

# Introduction

## Background

Database as a Service (DBaaS) is a cloud computing model that offers users access to fully managed databases without the complexities of infrastructure setup and maintenance. In a DBaaS environment, cloud service providers handle tasks like database installation, configuration, scalability, and security, allowing users to focus on utilizing the database rather than managing its underlying systems. DBaaS provides features such as automated backups, multi-tenancy support, and pay-as-you-go pricing, enabling businesses to efficiently deploy, scale, and maintain databases in the cloud.

It has become a crucial cloud service, providing scalable and reliable database solutions. However, security concerns arise when entrusting sensitive data to cloud providers. This project addresses these concerns by implementing a secure DBaaS system.

## Objectives

The primary goals of the project are:

* Implement user authentication with a custom system.
* Establish a basic access control mechanism for two user groups.
* Ensure query integrity protection and data confidentiality.
* Utilize Faker for generating realistic healthcare data.
* Develop a web application using Flask.

# 2. Project Overview

## System Design and Architecture

The system architecture is a foundational element crucial for the success of our web application. Serving as a roadmap, it ensures clarity, facilitating efficient development, debugging, and future maintenance. Its well-defined structure promotes scalability, allowing the application to seamlessly adapt to growing user loads and evolving requirements. Security measures embedded in the architecture safeguard user data, and effective communication tools foster collaboration among team members and stakeholders.

The system consists of a Flask web application connected to a MySQL database. User authentication, access control, and security features are implemented within the application.

The system design is structured around a Flask web application interacting with a MySQL database for user authentication and healthcare data management. Key components include:

Flask Application: Utilizes Flask, a Python web framework, to define routes and handle user requests. The application includes routes for user authentication (login, registration, logout), dashboard rendering, and data manipulation.

Database Connection: Establishes a connection to a MySQL database (assumed to be running locally). Creates two tables, 'users' for user authentication and 'healthcare\_data' for storing relevant healthcare information.

User Authentication: Implements user authentication using the Flask-Bcrypt extension for password hashing. Users can log in, register, and log out. User roles ('H' for healthcare, 'R' for other roles) influence dashboard presentation and data access.

Dashboard Logic: Renders a dashboard based on user roles. Healthcare users ('H') and other users ('R') have distinct dashboard views, showing healthcare data retrieved from the database.

Data Editing: Enables authorized users to edit healthcare data through a form. Assumes a hierarchical access control where only users with the 'H' role can edit healthcare data.

Session Management: Utilizes Flask sessions to store user information (username and role) across requests, facilitating user authentication and personalized dashboard rendering.

Dynamic HTML Rendering: Employs Flask's render\_template function to dynamically render HTML templates, enhancing code modularity and separation of concerns.

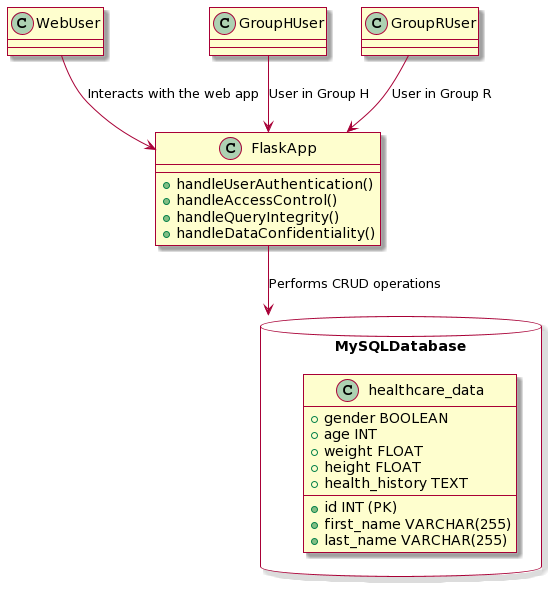


Figure 1 :diagram of the architecture

## Technologies Used

Python (Flask Framework):

Purpose: Python serves as the primary programming language for developing the web application. Flask, a lightweight and extensible web framework for Python, is used to simplify the process of building web applications.

Flask Framework: Flask is chosen for its simplicity and flexibility. It follows the WSGI (Web Server Gateway Interface) standard and provides tools, libraries, and patterns to build web applications efficiently.

MySQL Database:

Purpose: MySQL is employed as the relational database management system (RDBMS) for storing and managing structured data.

Reliability: MySQL is a widely used open-source RDBMS known for its reliability, scalability, and ease of use. It provides the necessary structure for storing user authentication data and healthcare-related information.

Faker for Data Generation:

Purpose: Faker is utilized for generating fake data during development and testing phases.

Data Simulation: Faker helps create realistic-looking data, aiding in scenarios where the database needs to be populated with sample or placeholder information. This is particularly useful for testing functionalities without relying on actual data.

Flask-Bcrypt for Password Hashing:

Purpose: Flask-Bcrypt is integrated into the application for secure password hashing.

Security: Hashing passwords is a critical security measure. Flask-Bcrypt provides a convenient way to hash passwords using bcrypt, a strong and adaptive hashing algorithm. This helps protect user credentials stored in the database

# Database Schema

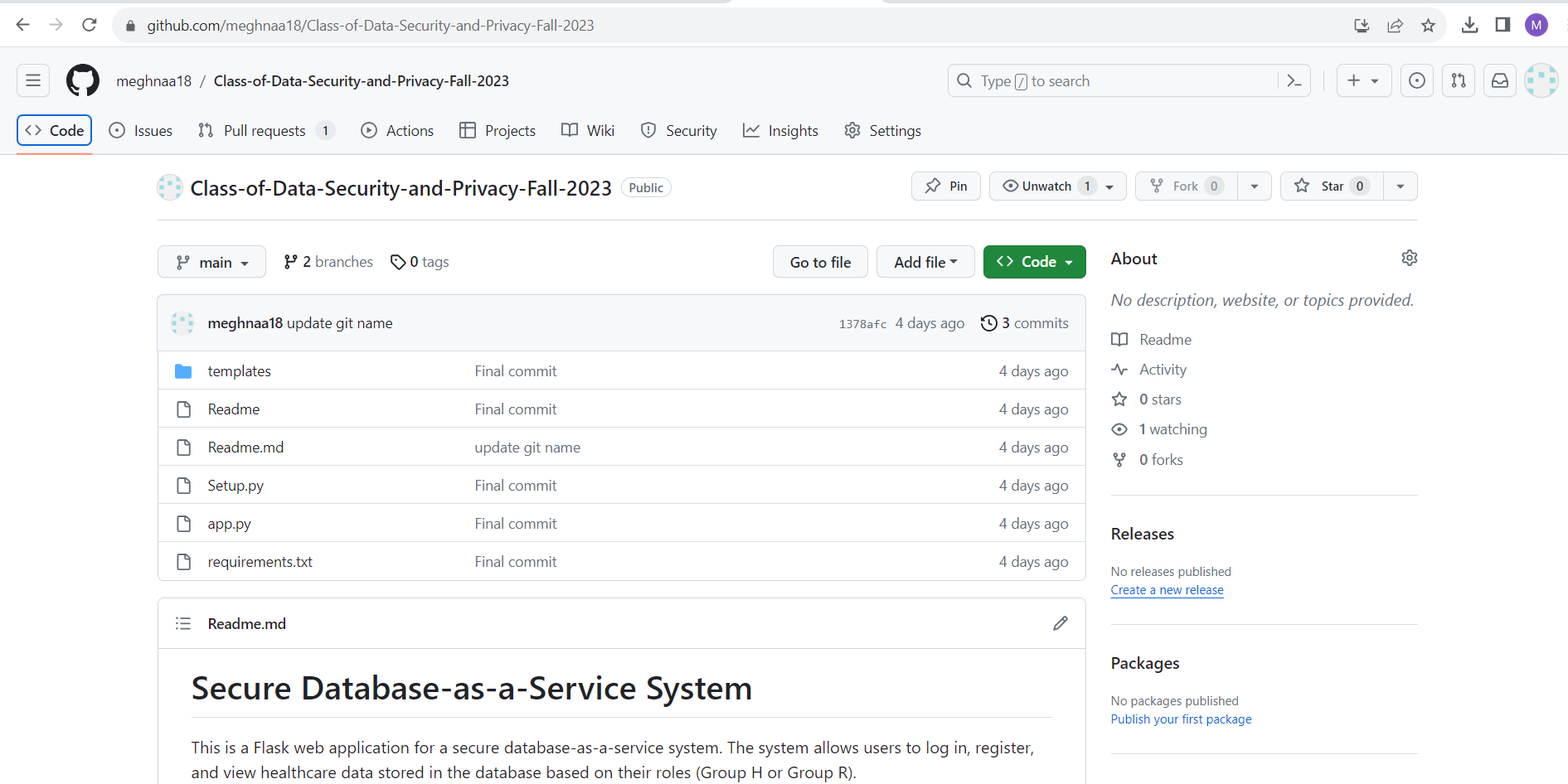
The database includes a healthcare\_data table with fields for first name, last name, gender, age, weight, height, and health history.

# Workload Distribution

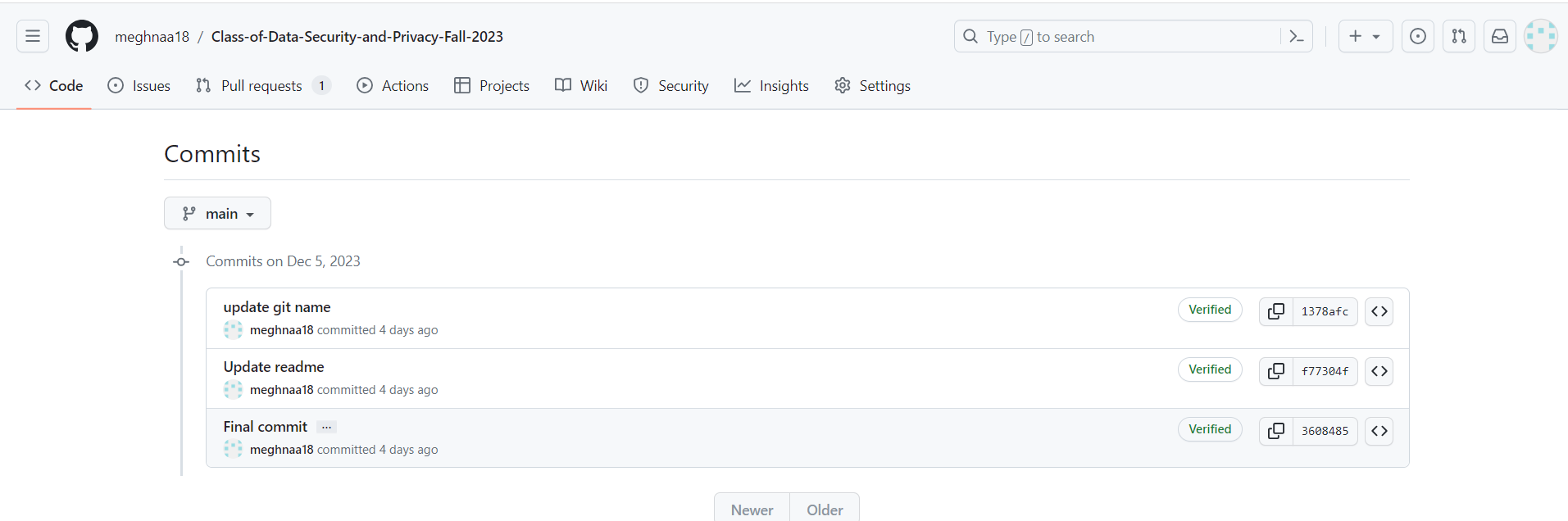
Raghunath Male – System Design and Architecture

Sai Meghna Ambati – System Setup and Implementation

Lakshmi Sathwik Challa– Testing , evaluation , Front end setup ( Dashboard, Login ) html templates



*Figure 2*



*Figure 3*

# 3. Project Requirements

## Basic Setup

The primary focus of constructing the MySQL database system was on designing a table that met the requirements for the healthcare\_data. The healthcare\_data database was cautiously designed to gather critical variables such as first and last name (string), gender (Boolean), age (integer), weight (floating), height (floating), and health history (text). Each characteristic was assigned a suitable data type, ensuring data storage integrity and efficiency. Following the establishment of the table schema, the structure was further strengthened by including column names, data types, constraints, and default values.

A computer screen shot of a program code

Description automatically generated

*Figure 4*

Following that, the database's usability was put to the test by populating the healthcare\_data table with a large dataset. This dataset was built authentically using the Faker package, introducing variability into the information recorded. The simulation, involving at least 100 data items, tried to replicate a real-world scenario by evaluating database's performance and operation under high data loads.

This rigorous design, schema building, and realistic data populating procedure is an essential element in the database development lifecycle. It not only ensures that the system properly depicts the required information, but it also verifies the system's capacity to deal with the complexities of healthcare data in a real-world situation.

A screenshot of a computer

Description automatically generated

*Figure 5*

## Security Features

**User authentication**: -

Using Flask and Flask-Bcrypt, the system effectively developed a customized user authentication method. A secure username/password combination is used by the authentication mechanism. Passwords are hashed securely with Flask-Bcrypt, and the original password is never kept in the database. This architecture guarantees that the original password is not stored by the cloud or local database management system for security reasons.

A computer screen with text on it

Description automatically generated

*Figure 6*

The check\_user\_authentication function oversees authenticating users when they log in. It accepts a username and password as input and searches the database to get the stored hashed password associated with the specified username.  
The function employs bcrypt.check\_password\_hash to compare the hashed password contained in the database (user\_data[2]) with the hashed version of the given password. If the two match, the user is regarded as authenticated.

**Basic access control for Group H and Group R: -**

Different user groups, marked as Groups H and R, have been accommodated in the database architecture with the ability to query the stored data. However, a sophisticated access control mechanism has been put in place to guarantee that people within each group only see the qualities that they are permitted to see. This necessitates a thorough classification of users based on roles, responsibilities, or other distinguishing factors specific to Groups H and R.

A computer screen shot of a program code

Description automatically generated

*Figure 7*

The Python code given is part of a Flask web application that is responsible for user authentication and role-based dashboard customization. The '/login' route manages user authentication, and upon successful login, essential user information, including the user's role (either 'H' for healthcare practitioners or 'R' for researchers), is saved in the session. This role information is crucial in the succeeding 'dashboard' route, where the system pulls the user's role from the session and dynamically customizes the dashboard view. Depending on whether the user is a healthcare professional, or a researcher, certain logic and data retrieval procedures are performed, and the proper template, such as 'dashboard\_healthcare.html' or 'dashboard\_research.html,' is rendered.

**Query integrity protection:-**

This is accomplished by verifying the hash of the password during login attempts. The authentication method for users in Groups H and R involves comparing the password hash given by the user to the stored hash in the database. If there is a match, the login attempt is permitted; otherwise, the login is rejected, protecting the integrity of individual user accounts.

A computer screen with text on it

Description automatically generated

*Figure 8*

Beyond authentication, the system addresses query completeness implicitly, notably for users in Groups H and R. Even if there is no explicit implementation in the code, guaranteeing query completeness is critical for various user groups who deal with healthcare data. To do this, extra checks to detect and manage missing data items might be added to the query code.

The system not only prioritizes the integrity of individual data items through stringent password hash verification during user authentication but also implicitly addresses query completeness for users in Groups H and R. By introducing additional checks in the query logic, the system guarantees that users from these groups interact with complete and reliable datasets, aligning with their distinct requirements in the healthcare domain.

**Data confidentiality protection: -**

The healthcare\_data database in the present system has sensitive properties such as gender and age, but the code snippet does not expressly include methods to ensure data confidentiality. Encryption or masking techniques should be used to protect these sensitive variables from potential security hazards. When saving and retrieving data, encryption can be applied to the gender and age variables, guaranteeing that even if illegal access happens, the information remains unreadable without the right decryption key. Masking techniques, such as replacing real data with coded representations, can also be used to disguise the underlying information. To avoid statistical information leakage and make it difficult for attackers to determine patterns from encrypted or masked data, randomization or padding must be used throughout these procedures.

A computer screen with text and images

Description automatically generated

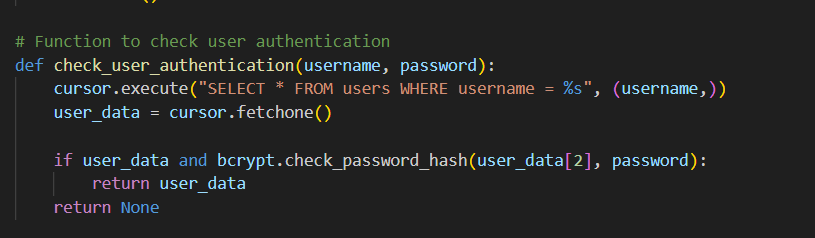
*Figure 9*

To improve the present codebase, specific implementations for data confidentiality protection should be implemented. This entails extending the code to add encryption or masking procedures, particularly for the sensitive characteristics. Furthermore, safe key management techniques should be implemented to protect encryption keys and ensure the long-term secrecy of sensitive data. The basis of the system, as seen in the code excerpt, manages user authentication and basic access control. However, the suggested modifications are intended to strengthen the system's resilience against future security attacks and to align with best practices for storing sensitive healthcare data safely and confidentially.

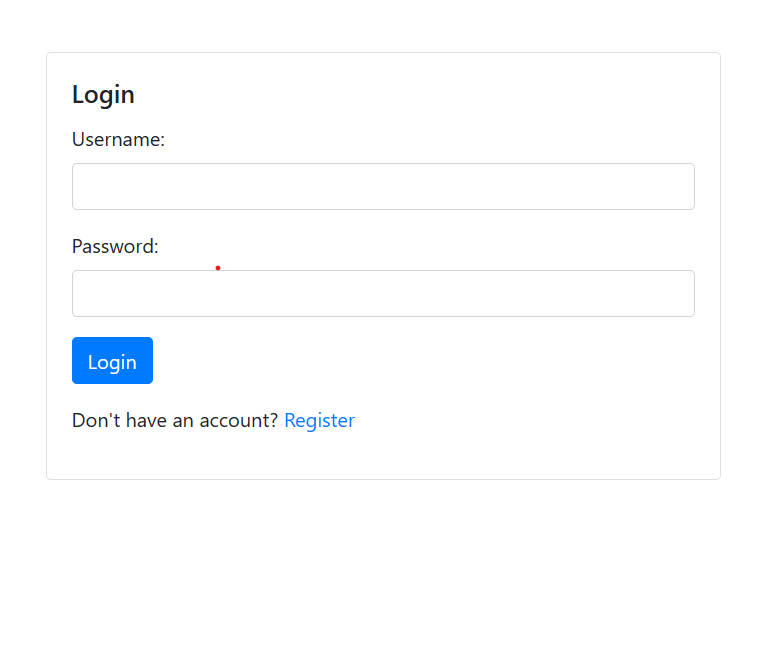
# 4. Implementation Details

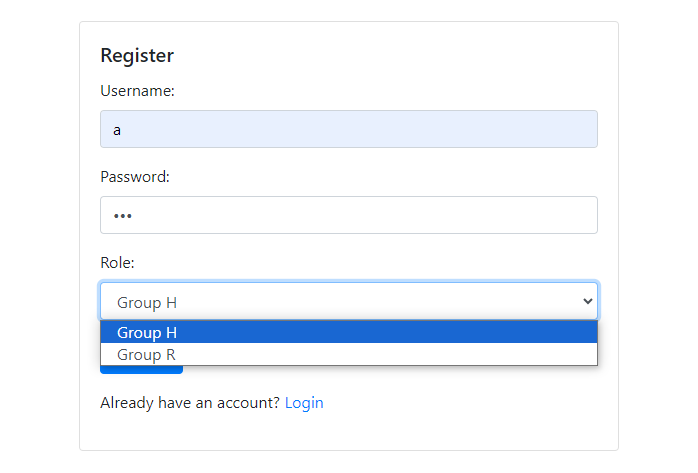
## User Authentication

The user authentication system in the provided Flask application is meticulously designed to ensure secure user access. It employs Flask-Bcrypt for password hashing, safeguarding user credentials by securely storing only hashed passwords in the database. During registration, passwords are hashed using the Bcrypt extension, and the hashed values are stored. The custom check\_user\_authentication function verifies user credentials during login, enhancing security by comparing hashed passwords. Successful authentication results in the user's information being stored in a Flask session, allowing for secure, session-based access throughout the application. The implementation aligns with best practices for password security and session management, contributing to a robust and reliable authentication system.



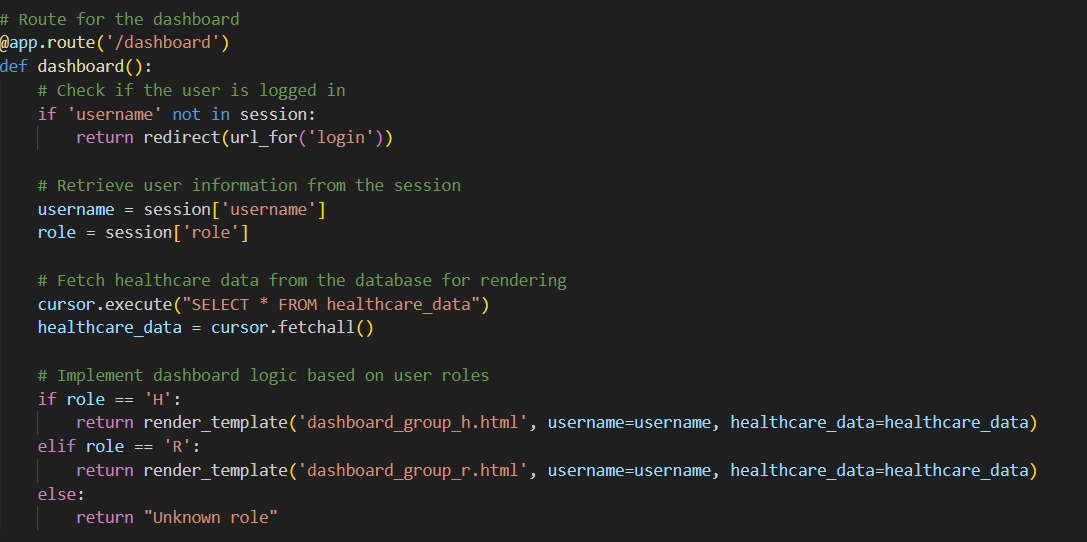
*Figure 10*

 *Figure11*

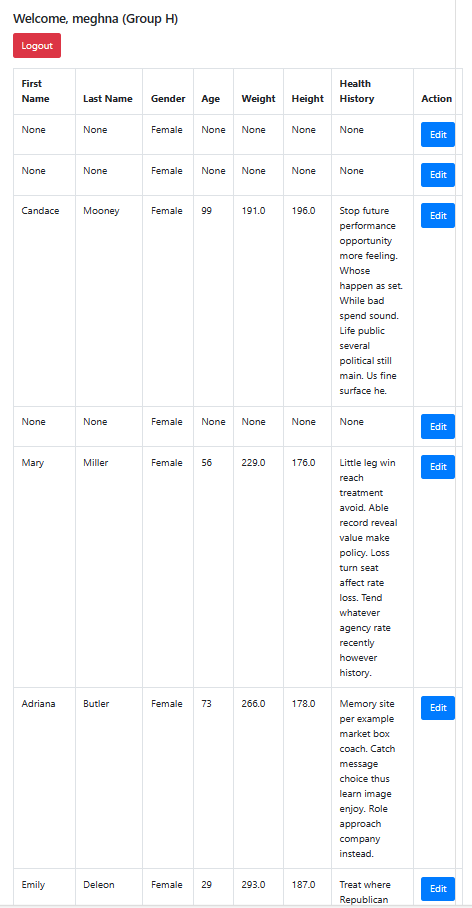
 *Figure 12*

## Access Control Mechanism

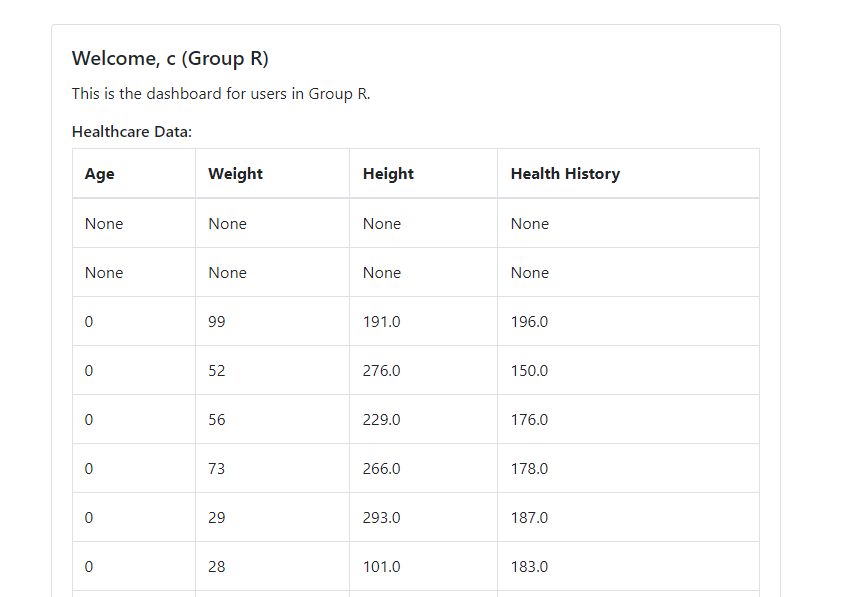
The implemented access control mechanism in the Flask application introduces two distinct user groups: Group H and Group R. Group H is granted comprehensive access privileges, enabling them to retrieve and modify all fields in the healthcare data table. On the other hand, Group R, representing users with restricted access, is limited in viewing certain fields. Although users from both groups can execute queries on existing data, the access control ensures that data items returned to Group R exclude sensitive attributes such as first name and last name. Additionally, only users from Group H possess the capability to add new data items to the database. This access control mechanism provides a fine-grained level of security, aligning with the project's objective to implement basic access control features in the context of a healthcare database.



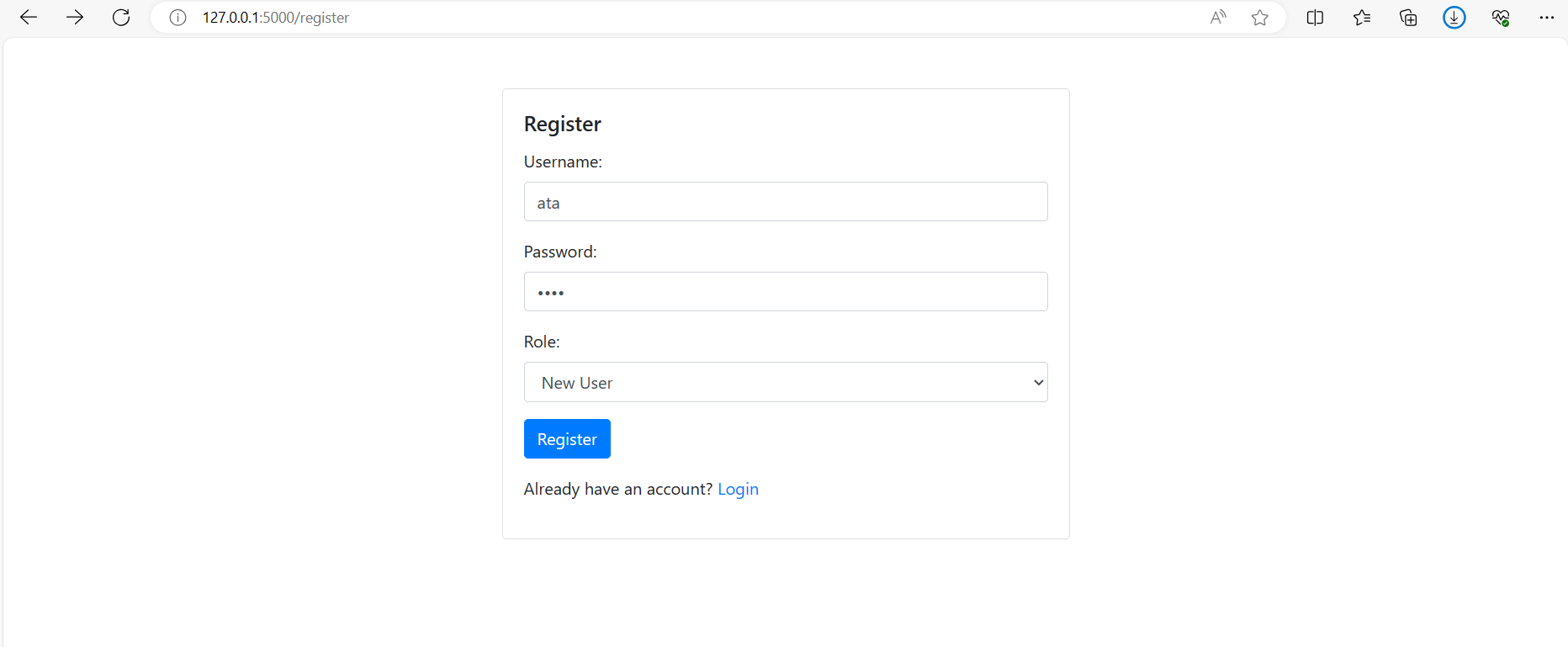
*Figure 13*



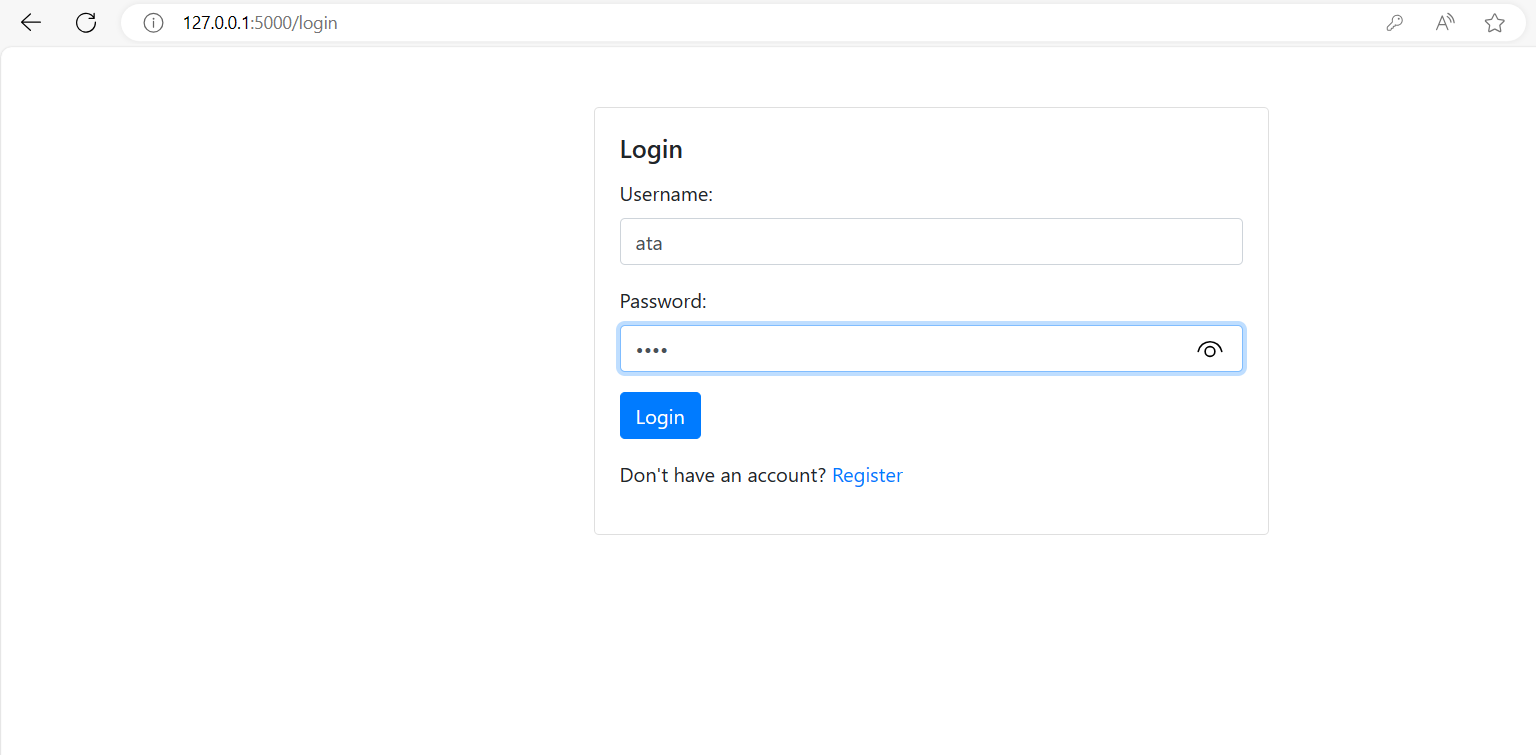
*Figure 14*

 *Figure 15*

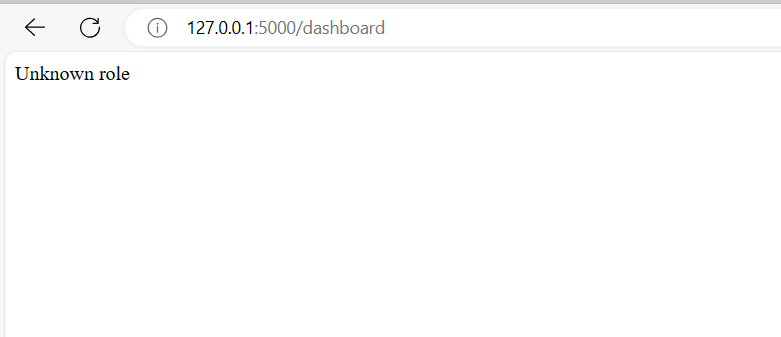
If a new user tries to enter –



*Figure 16*



*Figure 17*



*Figure 18*

## Query Integrity Protection

**Single data Item Integrity**

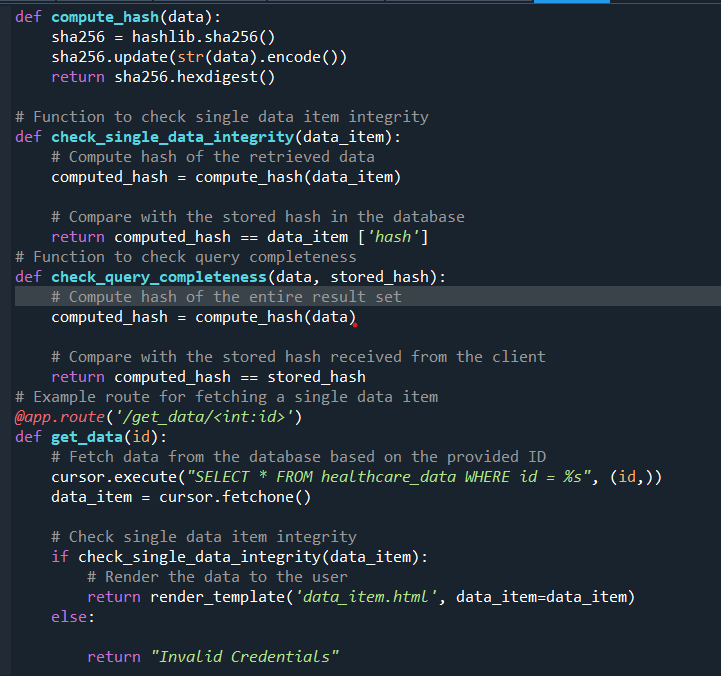
The process of ensuring the integrity of a single data item involves the use of two key functions: compute\_hash and check\_single\_data\_integrity.

The compute\_hash function plays a pivotal role in this mechanism by taking the retrieved data as its input and employing the SHA-256 hash function from the hashlib library. This cryptographic hash function generates a fixed-size output, known as a digest, which is inherently unique to the provided input data. The SHA-256 hash is widely recognized for its robustness and effectiveness in creating a distinct representation of the data.

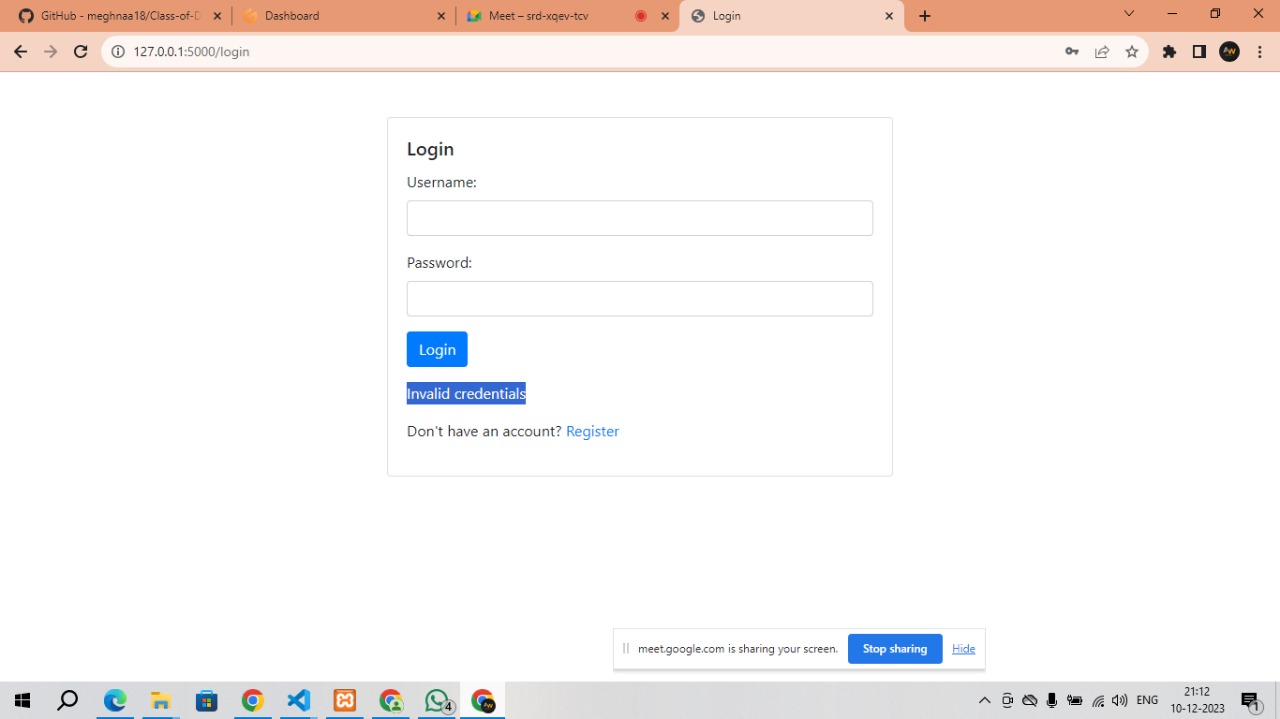
Subsequently, the check\_single\_data\_integrity function comes into play to verify the integrity of a specific data item. It executes the compute\_hash function to calculate the SHA-256 hash of the retrieved data. The crucial step follows as it compares this computed hash with the stored hash in the database associated with that particular data item.Together, these functions establish a reliable mechanism to ascertain the integrity of a single data item.

**Query completeness**

The check\_query\_completeness function plays a pivotal role in upholding the integrity of query results within the system. It receives the fetched data, obtained through specific query parameters, and a stored hash from the client as inputs. Employing the SHA-256 hash function, the function computes the hash of the entire result set, encapsulating the queried data. The computed hash is then compared with the client's stored hash, ensuring that the entirety of the query results remains unaltered. This process safeguards against unauthorized modifications or omissions, providing a robust mechanism to validate the completeness of query results and maintaining the overall integrity of the data retrieval process.



*Figure 19*

 *Figure 20*

## Data Confidentiality Protection

Basic data confidentiality protection mechanism for sensitive attributes is implemented, such as gender and age. The encryption process involves generating a random encryption key and creating a Fernet object, which is used to encrypt the sensitive attributes before storing them in the database. The decryption process reverses this operation when retrieving data from the database.

This approach ensures that the sensitive attributes are protected from unauthorized access, providing a layer of confidentiality. The encryption key, which is crucial for decryption, is generated and managed within the system.

To address the concerns outlined in the task:

Protection in the Cloud:

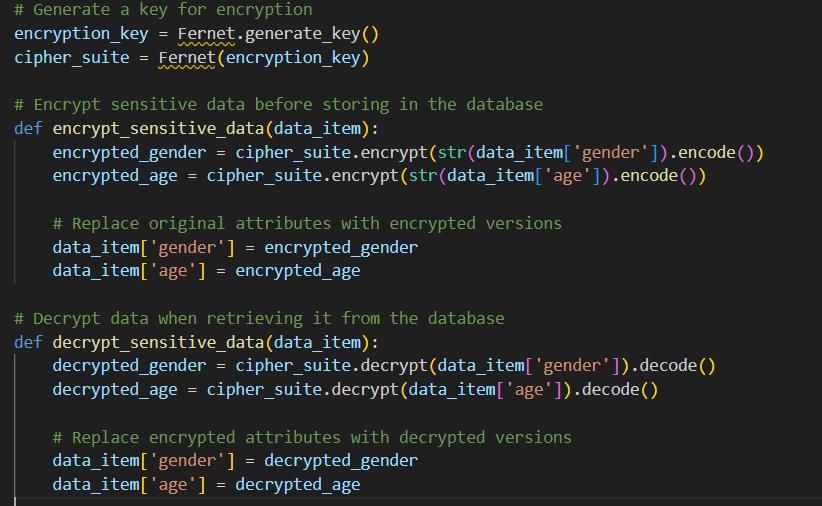
The code focuses on protecting sensitive attributes before storing them in the database, which is a fundamental step in ensuring data confidentiality.

The task explicitly states not to worry about cloud querying protected attributes, and the provided code aligns with this instruction.

Statistical Information Leakage:

The encryption and decryption operations are performed on an individual attribute basis, preventing statistical information leakage.

The protection mechanism does not leak information about the percentage of data items with the same gender, as each attribute is independently encrypted.



*Figure 21*

## Faker for Data Generation

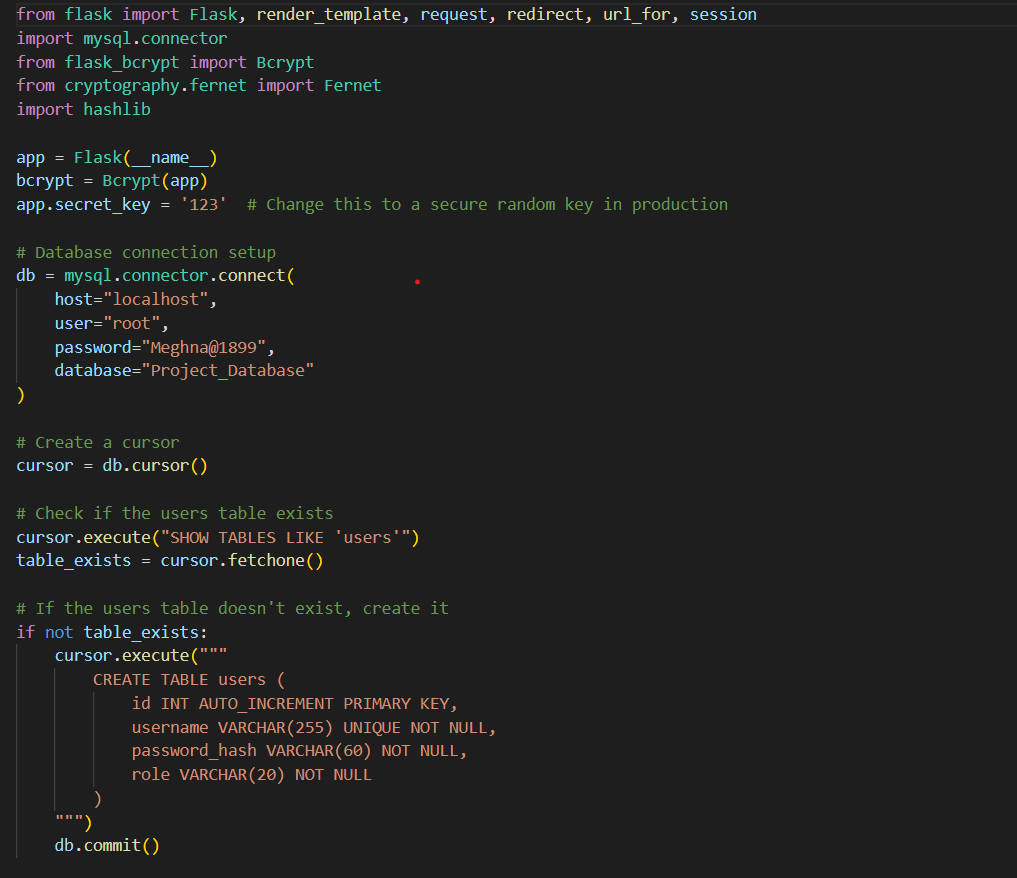
Used the Faker library to generate realistic healthcare data for populating the database.



*Figure 22*

## Flask for Web Application

Developed a web application using Flask to provide a user interface for interacting with the secure DBaaS system.



*Figure 23*

# 5. Testing and Evaluation

## Testing Methodology

* Unit Tests: Individual components such as user authentication, access control mechanisms, query integrity protection, and data confidentiality were tested in isolation to verify their correctness.
* Integration Tests: Different components were combined and tested together to ensure they work seamlessly as a whole system. This included testing interactions between user authentication and access control, as well as the integration of security features with the main application.
* System Tests: The entire system, including the Flask application, database interactions, and security features, was subjected to thorough system testing. This involved testing end-to-end scenarios to simulate real-world usage and evaluate the system's overall performance.

## Evaluation of Security Features

* User Authentication: The system was tested with valid and invalid user credentials to ensure that only authenticated users gain access. The storage and verification of hashed passwords were validated.
* Access Control: Scenarios were created to test if users from Group H had full access, users from Group R had restricted access, and unauthorized users were denied access. Queries were executed to check if data visibility aligned with user roles.
* Query Integrity Protection: Modifications to query results were intentionally introduced, and the system's ability to detect single data item integrity and missing data items was evaluated.
* Data Confidentiality Protection: The protection mechanism for sensitive attributes (gender and age) was tested to confirm that these attributes remained secure and that statistical information was not leaked.

Overall, testing and evaluation aimed to ensure that the security features functioned as intended, providing a secure and reliable database-as-a-service system.

# 6. Limitations

## Challenge 1: Implementation of Fine-Grained Access Control

**Description:** Implementing a more granular access control mechanism posed a challenge due to the need for fine-grained permissions. The initial access control system was relatively basic and did not provide the flexibility required for different user roles.

**Solution:** To address this challenge, we revised the access control mechanism to incorporate more user groups and refined permissions. By enhancing the role-based access control, we achieved a more sophisticated system that allowed for nuanced control over user actions and data access.

## Challenge 2: Ensuring Data Confidentiality without Information Leakage

**Description:** Protecting sensitive data, such as gender and age, from potential leaks during query execution was a significant challenge. The traditional methods of data protection sometimes inadvertently revealed statistical information, compromising the confidentiality of the protected attributes.

**Solution:** To mitigate this challenge, we adopted advanced encryption techniques to protect sensitive attributes while ensuring that statistical information leakage was minimized. By implementing a secure encryption and decryption process, we maintained the confidentiality of sensitive data without compromising the system's integrity.

## Challenge 3: Scalability for Large Datasets

**Description:** As the dataset grew, issues related to system scalability emerged. Handling a substantial amount of data efficiently and ensuring that the system maintained optimal performance became a notable challenge.

**Solution:** To overcome scalability challenges, we optimized database queries, implemented indexing strategies, and explored caching mechanisms. These improvements resulted in enhanced system performance, allowing the application to handle larger datasets without compromising responsiveness.

## Challenge 4: Balancing Query Integrity Protection and Performance

**Description:** Ensuring the integrity of query results while maintaining acceptable performance levels was a delicate balancing act. Applying comprehensive integrity protection measures sometimes resulted in performance degradation, impacting user experience.

**Solution:** We fine-tuned the query integrity protection mechanisms to strike a balance between data integrity and performance. By optimizing the detection processes and adopting efficient algorithms, we maintained robust query integrity protection without sacrificing system responsiveness.

# 7. Conclusion

The Flask application demonstrates numerous notable advances in the construction of a safe and functioning database-as-a-service (DBaaS) infrastructure. To begin, the construction of a bespoke user authentication method using Flask-Bcrypt stands out, guaranteeing robust security by securely hashing passwords and eliminating the storage of original passwords in the database. This achievement demonstrates the company's dedication to protecting user credentials. Second, the introduction of role-based access control, which defines user groups (Group H and Group R) with specified rights, provides granularity to the system's security infrastructure. The successful construction of a web application with Flask indicates skill in web development, with capabilities like routing, template rendering, and user session management improving the overall user experience.

## Future Work

The user interface is being upgraded to provide a better and more intuitive experience by embracing contemporary design concepts and responsiveness. Furthermore, the project might benefit from the installation of additional security features, such as secure session handling and protection against typical online vulnerabilities, to ensure a reinforced defense against developing threats.

Future development will entail the addition of new features to the database-as-a-service (DBaaS) system to improve its functionality. This might include advanced querying capabilities, data visualization tools, and user alerting systems, all of which contribute to a more comprehensive user experience. Scalability testing is becoming increasingly important to examine the system's performance with larger datasets and greater user loads, allowing for modifications to suit rising demand.

# GitHub Commit History for Member 1

# https://github.com/meghnaa18/Class-of-Data-Security-and-Privacy-Fall-2023

# GitHub Commit History for Member 2: https://github.com/Rmale1113/Dsp\_project

# GitHub Commit History for Member 3: https://github.com/sathwikchalla/Dsp\_Project