Final Detailed Report

# I. Introduction

This project aims to build a full-fledged weather application integrated with a MySQL database backend. The system allows users to sign up, log in, retrieve weather data for specified locations, and store their search history. The backend is implemented in Python, using popular libraries like mysql-connector-python, bcrypt, and requests. The database schema is designed to maintain user data and search history, ensuring both security and performance.

# II. System Architecture

The architecture of the weather application is composed of the following components:  
1. Database Layer: A MySQL database was selected for storing user credentials and search history. The database schema was designed with two tables—`users` and `search\_history`—to support efficient storage and retrieval of weather data and user information.  
2. Backend Logic: Python scripts handle the core logic of the application, including user registration, login, weather data retrieval, and history management. Various modules were written to handle the different parts of the application:  
 - db.py: Handles all database-related operations, including connections, queries, and transaction management.  
 - auth.py: Manages user authentication, including password hashing with bcrypt and session handling.  
 - weather.py: Handles the retrieval of weather data from an external API and processes the data for storage and display.  
3. Frontend Interaction: Although the primary focus is on backend development, the system includes a simple command-line interface for user interaction.

# III. Implementation Details

1. Database Design and Queries: The database schema follows a relational design with two tables—`users` and `search\_history`. The `users` table stores user information, and the `search\_history` table records each weather search performed by users. Password hashing is implemented using bcrypt to secure user credentials.  
2. User Authentication: User registration and login are managed using bcrypt for password hashing. The MySQL password validation policy was adjusted to allow more flexibility.  
3. Weather Data Retrieval: The application interacts with an external weather API to retrieve weather information using the `requests` library.  
4. Search History Management: Large amounts of search history data are managed efficiently using indexing and foreign key constraints.

# IV. Challenges Faced and Solutions Implemented

1. Password Management: A challenge with MySQL’s default password validation policies was addressed by setting the policy to `LOW`, and bcrypt was used for secure password storage.  
2. Database Connection Handling: Connection pooling was implemented to improve performance during concurrent access.  
3. API Data Management: Error handling was implemented for failed API requests, improving reliability.  
4. Search History: Indexing improved the efficiency of search queries for large datasets.

# V. Key Technologies Used

1. Python: Used for backend logic, including user authentication, API calls, and database interactions.  
2. MySQL: Chosen for storing user and search history data with a relational schema.  
3. bcrypt: Used for password hashing to ensure secure authentication.  
4. requests: Used for interacting with the external weather API.  
5. mysql-connector-python: The MySQL driver used for database connections.

# VI. Security Considerations

1. Passwords are hashed using bcrypt, ensuring they are never stored in plaintext.  
2. Database access is restricted to authorized users only.  
3. Sensitive information such as API keys is stored in environment variables using `python-dotenv`.

# VII. Future Enhancements

1. A more interactive and user-friendly interface could be implemented using a web framework like React.  
2. Advanced search functionality, such as filtering search history by date or location, could be added.  
3. Caching frequently requested weather data could improve performance.  
4. Mobile app support could be added using frameworks like React Native or Flutter.

# VIII. Conclusion

This project successfully implements the core functionality of a weather application, including user authentication, weather data retrieval, and search history management. Despite challenges such as password management and API integration, the project was completed with a focus on performance and security.

# File Structure and Code Explanation

The project structure is explained below with code snippets included for clarity.

## File Structure:

.  
├── README.md  
├── app  
│ ├── auth.py  
│ ├── history.py  
│ └── weather.py  
├── config  
│ └── db.py  
├── database.txt  
├── main.py  
└── requirements.txt

### 1. README.md

This file usually contains the project description, setup instructions, and usage guidelines. In this project, it likely details how to run the weather application, its dependencies, and provides insights into the functionality of the application.

### 2. app/auth.py

This file is responsible for handling user authentication logic. The code below demonstrates how bcrypt is used to hash and verify passwords:

import bcrypt  
  
def hash\_password(password):  
 return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())  
  
def verify\_password(stored\_password, provided\_password):  
 return bcrypt.checkpw(provided\_password.encode('utf-8'), stored\_password)

### 3. app/history.py

This module is responsible for managing the search history of the users. It interacts with the database to store and retrieve user search history.  
  
Example function:

def add\_history(user\_id, location, weather\_data, timestamp):  
 query = "INSERT INTO search\_history (user\_id, location, weather\_data, timestamp) VALUES (%s, %s, %s, %s)"  
 values = (user\_id, location, weather\_data, timestamp)  
 db.execute(query, values)

### 4. app/weather.py

This file handles retrieving weather data using an external API such as OpenWeatherMap or similar. Here’s a sample code to demonstrate how the weather API is called using the requests library.

import requests  
  
def get\_weather(location):  
 api\_key = "your\_api\_key"  
 url = f"http://api.openweathermap.org/data/2.5/weather?q={location}&appid={api\_key}"  
 response = requests.get(url)  
 return response.json()

### 5. config/db.py

This file manages the database connections. It initializes the connection to the MySQL database and provides a helper function to execute queries.

import mysql.connector  
  
def get\_db\_connection():  
 return mysql.connector.connect(  
 host="localhost",  
 user="sss\_assignment\_sep24",  
 password="doitnow",  
 database="weather\_app"  
 )

### 6. database.txt

This file contains SQL commands for setting up the MySQL database. The schema includes two tables, users and search\_history, as defined below:

CREATE TABLE users (  
 id INT AUTO\_INCREMENT PRIMARY KEY,  
 username VARCHAR(50) UNIQUE,  
 password VARCHAR(255)  
);  
  
CREATE TABLE search\_history (  
 id INT AUTO\_INCREMENT PRIMARY KEY,  
 user\_id INT,  
 location VARCHAR(100),  
 weather\_data TEXT,  
 timestamp DATETIME,  
 FOREIGN KEY (user\_id) REFERENCES users(id)  
);

### 7. main.py

This file serves as the entry point for the application. It typically combines all the functionalities and starts the application. A basic outline of the code could be as follows:

from app.auth import hash\_password, verify\_password  
from app.weather import get\_weather  
from app.history import add\_history  
from config.db import get\_db\_connection  
  
def main():  
 # User interaction and logic for weather search  
 pass  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

### 8. requirements.txt

This file lists the dependencies needed for the project. In your project, the dependencies include:

mysql-connector-python==8.0.31  
bcrypt==3.2.0  
requests==2.28.1  
python-dotenv==0.21.0