Contents

[1. ERDs (Entity-Relationship Diagrams) 1](#_Toc180661231)

[1.1 Logical Model 1](#_Toc180661232)

[1.2 Relational Models 2](#_Toc180661233)

[In the relational model for this project: 2](#_Toc180661234)

[Users Table: 2](#_Toc180661235)

[Passwords Table: 2](#_Toc180661236)

[Explanation: 3](#_Toc180661237)

[Importance of Salts and Hashing: 3](#_Toc180661238)

[3. Security Considerations 3](#_Toc180661239)

[Benefits of This Approach: 3](#_Toc180661240)

[4. Python Code for Sign-Up and Login 4](#_Toc180661241)

[4.1 Sign-Up 4](#_Toc180661242)

[4.2 Login 4](#_Toc180661243)

[5. Conclusion and Future Considerations 4](#_Toc180661244)

[6. Additional Features 5](#_Toc180661245)

# 1. ERDs (Entity-Relationship Diagrams)

# 1.1 Logical Model

The Logical Model focuses on the high-level structure of the database, outlining the various entities, attributes, and relationships without considering the specific physical implementation in a database system.

In a logical model, the focus is on what data needs to be stored, how the different data entities are related, and what the constraints on the data are. For instance, a logical model might define the relationships between users and passwords tables (like one-to-one relationships) but without specifying the exact column types or indexes.

Typical components of a logical model include:

Entities (such as users, passwords)

Attributes (such as username, hashed password, salt)

Relationships (such as foreign keys, e.g., a user can have one password)

Logical models help us understand the abstract structure of the database and serve as a blueprint before moving on to more technical aspects like the Relational Model.

A diagram of a computer code

Description automatically generated with medium confidence

# 1.2 Relational Models

A Relational Model translates the logical design into a concrete structure, defining how data will be stored in a relational database like PostgreSQL. This model defines tables, columns, data types, relationships, constraints (e.g., primary and foreign keys), and normalization techniques.

# In the relational model for this project:

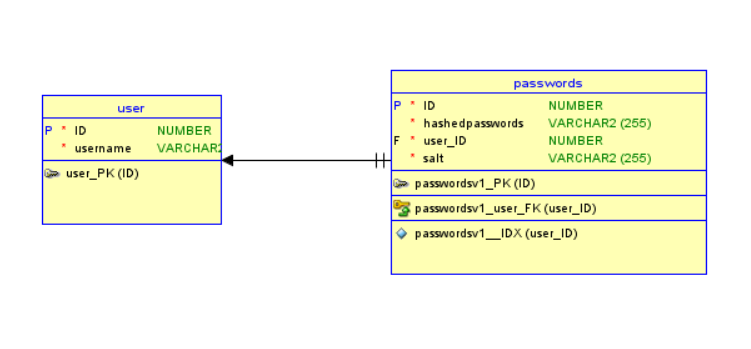
There are two main tables: users and passwords.

The primary keys are defined to ensure each entry in the table is unique.

Foreign keys are defined to link the tables together (i.e., user\_id in the passwords table references id in the users table).

2. PostgreSQL Code

This section includes the SQL statements to create the users and passwords tables in PostgreSQL.



# Users Table:

sql

Copy code

CREATE TABLE users (

id SERIAL PRIMARY KEY,

username VARCHAR(100) NOT NULL UNIQUE

);

The users table contains a unique ID (SERIAL PRIMARY KEY) and a username field.

The username is required to be unique, ensuring that no two users can have the same username.

# Passwords Table:

sql

Copy code

CREATE TABLE passwords (

id SERIAL PRIMARY KEY,

hashed\_password VARCHAR(255) NOT NULL,

salt VARCHAR(255) NOT NULL, -- New column to store the salt

user\_id INT NOT NULL UNIQUE,

CONSTRAINT fk\_user FOREIGN KEY(user\_id) REFERENCES users(id) ON DELETE CASCADE

);

The passwords table stores:

A hashed password.

A salt, which is a random value used to protect the hashed password.

A user\_id that is linked to the users table.

The foreign key constraint ensures that the user\_id corresponds to a valid user in the users table. If a user is deleted, the corresponding entry in the passwords table is also deleted

# Explanation:

Auto-incrementing IDs: The SERIAL data type in PostgreSQL automatically increments the id for each new user or password record, ensuring a unique identifier.

Password Security: The use of both salts and hashes ensures the security of user passwords, even in the event of a database breach.

# Importance of Salts and Hashing:

Salts make sure that even if two users have the same password, their stored hashed passwords will differ. This is a crucial defense against rainbow table attacks.

Hashes transform the plain-text password into a fixed-length string that can't easily be reversed, ensuring password security.

# 3. Security Considerations

I chose the combination of salts and hashes to secure passwords because this method is simple yet effective. This approach ensures that even if an attacker gains access to the database, they won’t be able to retrieve user passwords due to the irreversibility of the hash function and the uniqueness of the salt.

# Benefits of This Approach:

Hashing makes it computationally infeasible to reverse the hashed password back to its plain text form.

Salts add an additional layer of security, ensuring that each user's password is hashed uniquely, even if they use the same password.

SHA-256 is a widely recognized cryptographic hash function that provides a secure way to store password hashes.

# 4. Python Code for Sign-Up and Login

# 4.1 Sign-Up

The sign-up functionality in Python will:

Generate a salt when a new user is created.

Hash the user's password combined with the generated salt using the SHA-256 algorithm.

Store the salt and hashed password in the passwords table, along with a reference to the user's id in the users table.

# 4.2 Login

The login functionality will:

Retrieve the stored salt and hashed password for the user.

Hash the entered password combined with the stored salt.

Compare the computed hash with the stored hashed password to verify if the credentials are correct.

# 5. Conclusion and Future Considerations

The design of this system, leveraging relational models and PostgreSQL's built-in security features (e.g., constraints, cascading deletes), provides a solid foundation for user authentication. Going forward, we could enhance this design by:

Using more advanced password hashing algorithms such as bcrypt or Argon2, which are more resistant to brute-force attacks.

Adding support for two-factor authentication (2FA) to further secure user accounts.

Implementing rate-limiting on login attempts to mitigate brute-force attack attempts.

# 6. Additional Features

Potential future features that could be added include:

Password reset functionality: Allow users to reset their passwords in a secure manner, possibly sending a reset token to their email.

Account management: Enabling users to update their usernames and passwords directly from the user interface, while maintaining the security of their credentials.