Users.sql:

```
CREATE TABLE users(
user_id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT UNIQUE,
first_name TEXT,
last_name TEXT,
 email_address TEXT UNIQUE,
group_name TEXT,
password TEXT, salt TEXT);

CREATE TABLE password_history(
user_id INTEGER,
password TEXT,
FOREIGN KEY (user_id) REFERENCES users(user_id));
```

These are the same tables from project 1, since the user management system is almost the same. The users table stores a user's information, and the password history tables stores their password history. User_id serves as the primary key that is autoincremented with each user, so each users has a distinct id. Username must also be distinct for security and identification. First_name and last_name store text information. Email_address is also UNIQUE so one email can't be used to make multiple accounts. Password stores the hashed text value, not the plaintext, for security. Salt stores a value used to hash the password, as an extra layer of security

The password history table has a user_id that references the user_id of users as a foreign key, which associates each stored password with the user that created it. The password column again stores the hashed value of the password for security. There is some redundancy in storing a user's current password in both tables, but I chose to add it to both for accurate documentation, and it would be added when the password changes anyway.

Documents.sql:

```
CREATE TABLE files(
id INTEGER PRIMARY KEY AUTOINCREMENT,
creator TEXT,
filename TEXT);

CREATE TABLE file_groups(
id INTEGER,
groupname TEXT,
FOREIGN KEY (id) REFERENCES files(id) ON DELETE CASCADE);
```

The files table stores information about each file. Id is a unique identifier, which is autoincremented with each file to ensure that each file has its own unique identifier. Creator stores the username of the account that created the file, and filename stores the name of the file. File_groups stores the groups that have access to the file, stored in a separate table to avoid redundancy, and allow multiple groups to be associated with one file. Id is a foreign key that references the id of the file in

files, and groupname stores the name of the group. This table also deletes on cascade, when the a file is re-created that already exists, to delete any groups associated with that file that might be updated.

Log.sql

```
CREATE TABLE events(
id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT,
event TEXT,
timestamp DATETIME);

CREATE TABLE file_events(
id INTEGER,
filename TEXT,
FOREIGN KEY (id) REFERENCES events(id));
```

The logs.db stores the log information of the system. The events table stores each event, with an additional table storing information unique to the logging of a file. Events has an id PRIMARY KEY that is autoincremented so each log entry has a unique identifier. The username of the account that performed the action is stored in username, and what type of event is stored in event. There is also a timestamp to track the time when the action was performed. The file_events table stores the name of the file in filename, if an action involving a file was taken, along with an id foreign key referencing events.id to maintain that association. This avoids some redundancy by not including a filename column in events that would be NULL for any action that doesn't involve a file. There can be repeated filenames, but each instance is associated with a unique id in events to point to the file, which makes for easy counting.

I did not use a database for the search microservice, instead requesting information from the other microservices.