

Table of Contents

- User Documentation
 - Intro
 - Installation steps
 - How to start app
 - API documentation
 - Example usage
- Technical Documentation
 - Application architecture / methods
 - Technical details / functionality
 - Library dependencies

User documentation

- Intro / Overview of app
 - Everybody wants a password manager - a streamlined way to organize highly secure passwords that are impossible to memorize due to volume and uniqueness. Many of us, however, may not trust a company like Apple or Google to store those passwords. There are three reasons to be wary of company-based password managers: first, storing passwords with a company may enable them to misuse our personalized information. Second, individuals may want to avoid storing any passwords on the web whatsoever due to the vulnerability inherent to cloud-based storage. Third, many professional password managers who try to mitigate these concerns operate on a subscription model.
 - We created a free-to-use password manager packaged as a downloadable application to avoid company interference, cloud-based security risks, and predatory pricing models. Users are able to download our app onto their local machines. After a successful download and set up, users will be directed to a user interface, allowing for easy access and creating a user friendly experience.
 - Each user will locally host their own container to store their database of passwords. The app follows a master-key workflow that imitates existing cybersecurity workflows to enable verification and validation alongside secure, encrypted storage of sensitive information. A master password will give users access to their entire password database.
- Installation steps
 - Clone the Repository
 - Option 1: Clone this GitHub repository and checkout the proper branch:
 - git clone <https://github.com/cburkhardt27/password-manager-cs-370.git>
 - Windows:
 - git checkout windows
 - Mac
 - git checkout mac_download

- Option 2: Download as a zip file
- Download this repo as a zip file and extract to desired location
-
- Setup your Python Environment
- Windows
 - `python -m venv win_venv`
 - `win_venv\Scripts\activate`
 - `pip install -r requirements.txt`
- Mac
 - `python3 -m venv mac_venv`
 - `source mac_venv/bin/activate`
 - `pip3 install -r requirements.txt`
- Install Front-End Dependencies
 - `npm install`
- Package the Application Pages and Main Process
 - `npm pack:r`
 - `npm pack:m`
- Start the Application
 - `npm start:e`
- Build the Application for Desktop
 - `npm run make`
- *Note: The output files will be in the /out directory. Navigate through the `make` folder to locate the .dmg file, then install it.*
- Advanced Instructions / Installation
- Build and Setup Scripts
- Scripts are managed through package.json, which uses Electron Forge for building the desktop application.
- To view other build/setup scripts:
- ``npm run``
-
- These include:
- ``build:dev`` & ``start:dev``: Development environment for React/Electron renderer. Not recommended for pages interacting with the backend.
- ``start:e``: Starts Electron.
- ``pack:m``: Webpack for main.js and preload.js. Outputs to the pack folder.
- ``pack:r``: Webpack for React/renderer pages. Outputs to the pack folder.
- ``package``: Creates an Electron executable.
- ``make``: Builds the installer.
-
- How to start the app

- Follow installation steps and run the executable
- Create a username and master password to gain entry to the app
- API documentation
 - validate_master_password(): User authentication via master password
 - When the user first downloads the application and it, they must create a username and master password to gain access. This username and master password is stored in a master password table in the database. Subsequently, whenever the user wants to login into the app, they have to input their login information. This verifies the identity of the user and allows the user to utilize the other app functions.
 - Example usage:
 - Input:
 - Username: username
 - Master Password: mp
 - Outcome: successful login
 - add_password_entry(): Add a new password entry to the database
 - This app feature takes input from the user—username, url, and passwords—and then inputs that information into the vault as a password entry. The app encrypts the password and then inserts the encrypted password along with the url and username into an already initialized table(solely meant for password storage) in the database/vault. Then the user receives a password stored successfully message.
 - Example usage:
 - Input:
 - Username: username
 - Password: password
 - Url: website.com
 - Output: Password stored successfully
 - get_password(): Retrieve stored passwords from the vault
 - This feature takes input from the user (a url), retrieves the password associated with that input, and displays that password and its corresponding username to the user. This function retrieves the encrypted password and username, associated with the url, decrypts the password, and returns the decrypted password and username. If the password entry does not exist then a password not found error message is shown to the user.
 - Example usage:

- Input:
 - Url: website.com
- Output:
 - Username: username
 - Password: password
- delete_password(): Delete password entry
 - This feature allows the user to delete a password entry. When prompted, the user inputs the url and username of the password they want to delete. The password entry is then deleted from the vault and the user receives a prompt that the password was successfully deleted. If the password entry was not originally in the vault, then the user will receive an error message.
- Example usage:
 - Input:
 - Url: website.com
 - Username: username
 - Output: Password successfully deleted
- display_all_passwords(): Retrieves all passwords, usernames, urls in database
 - This feature allows a user to see all their passwords they have stored. When called, the function retrieves the username and password associated with each url in the database.
- Example usage:
 - Inputs: None
 - Outputs:
 - Url1: website1.com
 - Username1: username1
 - Password1: password1
 - ... for all stored data
- get_repeated_passwords(): Retrieves passwords that have been used multiple times across different urls in the database
 - This feature is used to see which passwords are being used multiple times across different urls. This is a security feature that allows users to see when a password is being overused and thus at higher risk.
- Example usage:
 - Inputs: None
 - Outputs:
 - Url1: website1.com
 - Username1: username1

- Password1: password1
- ... for all repeated passwords
- generate_random_password(): creates a random password
 - This feature allows a user to create a random password. It generates a password of length 12 made up of a random assortment of letters, numbers, and punctuation (special characters).
- Example usage:
 - Inputs: None
 - Outputs:
 - Password: Somevalidpassword123

Technical documentation

- Application architecture / methods
 - SQLite Database functions
 - Connect_db: This function takes the input DB_NAME and uses the sqlite3 function, connect, to connect to a sqlite database with the name DB_NAME. If this database does not exist, then one is created with the name DB_NAME. The established connection generates a connection object. That connect object can be used to create a cursor object. Both objects are returned The connection and cursor object will be necessary when executing database queries.
 - Create_password_table: This function takes the connection object as an input. First a query is used to create a table to hold password entries. The table has columns for an id, usernames, urls, passwords, and timestamps for when the entry is created and updated. A try and except block is used. The try block checks if the connection is established; if the connection is established, the query is executed using the cursor. In the except block, any errors are caught and printed. Lastly the cursor is closed.
 - Create_master_password_table: This function takes the connection object as an input. First a query is used to create a table to hold the master password. The table has a column for the username and the hashed master password. A try and except block is used. The try block checks if the connection is established; if the connection is established, the query is executed using the cursor. In the except block, any errors are caught and printed. Lastly the cursor is closed.
 - Add_master_password: This function initializes a hashed master password and username variables using the setup_user_master_pass() function from the encryption functions file. An insert query to insert into the master password table is made. First a connection is made using connect_db. Then an if statement is used to check if the connection is properly established; if the connection isn't properly established then the function

returns. A try and except block is used. Within the try block, the query is executed with the hashed master password and username inputs using the cursor object. The except block catches any errors. Lastly the connection and the cursor is closed.

- `Get_master_password`: This function first creates a retrieve query. Then a connection is made to the database. A try and except block is used. Within the try block, the cursor object is used to execute the retrieve query. `Cur.fetchone` is used to assign the retrieved information to a result variable. If the variable has no value then the master password table is empty and false is returned. The except query is used to catch any errors. Then the cursor and connection is closed and the username and hashed master password is returned.
- `Add_password_entry`: This function takes username, url, and password strings as inputs. First this function gets the master password and username and uses the username to encrypt the password input. A check query is made to check if the password entry input is already in the vault. An insert query is made to insert the input data into the password table. Then a connection is made to the database and connection and cursor objects are created. This function uses a try and except block. Within the try block, the check query is executed. If the input password entry already exists in the database then function is returned. Next, the insert query is executed using the cursor object. The except block is used to catch any errors and then the cursor and connection object is closed.
- `Get_password`: This function takes a url as an input. A select query is made to retrieve the password and username associated with the given url. A connection is established to the database; if the connection was not established, the function is returned. A try and except block is used. Within the try block, the query is executed using the cursor object. `Cur.fetchone` is used to assign the retrieved data to a result variable. If the result is not empty, the password, which is encrypted, is decrypted using `decode_vault_password` function from the encryption functions file. The `get master password` function is required to use the `decode vault password` function. Then the decrypted password and username are decrypted. If the result variable was empty, an error statement is printed and the function is returned. The except block catches any other errors. Lastly, the connection and cursor objects are closed.
- `Delete_password`: This function takes a username and url as inputs. Then a query to delete the password entry at the input username and url is created. A connection is established to the database; if the connection was not established, the function is returned. A try and except block is used.

Within the try block, the query is executed using the cursor object. The except block catches any errors and then the cursor and connection objects are closed.

- **Display_all_passwords:** This function first makes a query to retrieve all the passwords and their associated usernames from the password tables. A connection is established to the database; if the connection was not established, the function is returned. A try and except block is used. Within the try block, the query is executed using the cursor object. `Cur.fetchone` is used to assign the retrieved data, which is all the passwords in a list, to a result variable. For each password in the list, the `decode_vault_password` function from the encryption functions file is used to decrypt the password. The `get master password` function is required to use the `decode vault password` function. The decrypted password and its associated username are added to another list. Then this list is returned. If the result variable was empty, an error message is printed and the function is returned. The except block catches any errors and then the cursor and connection objects are closed.
- **Get_repeated_passwords:** this function returns every password, and its associated username, that is repeated. A try and except block is used. Within the try block, `display all password` function is called which returns a list with every decrypted password and its username. The first for loop uses a count list variable to keep count of the presence of each password. The second for loop uses that count list and for each count that is greater than one, the corresponding password entry is appended to a new list variable. If this list variable is empty, a no repeated passwords message is printed and the function is returned. Else, the list is returned. The except block catches any errors.
- **Flask endpoints**
 - Flask endpoints are defined for each of the SQLite database functions defined above with a corresponding method specification to GET, POST, or DELETE. These endpoints are called from the Electron application with Axios, an HTTP client.
- **Technical Details**
 - **Frontend:** user interface built with React and Electron to run as a desktop application
 - React: Provides responsive and interactive user interface
 - Electron: Wraps the frontend in a desktop application environment
 - Webpack: bundles the React code for production
 - **Backend:** Python based backend that manages logic and interacts with database

- Python: Handles backend logic including encryption, password management, and API endpoints
 - Flask: Provides API layer for frontend-backend communication
- Database: PostgreSQL database that stores user data
 - PostgreSQL: stores credentials and encrypted passwords securely
- Library dependencies
 - Encryption:
 - Bcrypt, string, secrets, Crypto.Cipher, base64, hashlib, pathlib, os.
 - SQLite Database:
 - Sqlite3, flask.
 - Frontend:
 - React, MUI(material, icons-material, system), PasswordItem, react-router-dom, electron.