

Password Manager Application

JAYARAJ J PILLAI

ROJIN JOHN JOSE

SHARON ABRAHAM

SREEHARI P

Department of Computer Applications

SAINTGITS COLLEGE OF ENGINEERING (AUTONOMOUS)

Kottukulam Hills, Pathamuttom P.O., Kottayam 686 532

NOVEMBER, 2024

ABSTRACT

The project aims to develop a secure password manager application to address the increasing cyber threats and password-related breaches. Here we integrate 2 applications such as Password Manager Application, Password Generator Application and Password Strength Checker. This application allows users to add username and passwords along with their associated domain name/URL's, listing stored usernames and passwords, deleting them, and more. The password manager authenticates users into the "Vault" manager using a master password, which stores all passwords. The Password Generator allows users to generate random passwords based on the custom conditions which comply with various password standards accepted by various sites, which can be further stored in the vault. The password strength checker provides users with feedback on the strength and complexity of their passwords, indicating their security. With a user-friendly interface, the application offers secure storage, providing a convenient solution for users to manage their passwords securely, customization options.

CHAPTER 1 INTRODUCTION

1.1 INTRODUCTION TO THE PROJECT

In today's digital age, managing multiple online accounts and their corresponding passwords has become increasingly complex. As people are required to create and store numerous passwords for various services, it's essential to have a reliable and secure way to handle this sensitive information. The Password Manager

Application, developed using Python Flask, aims to address these challenges by providing users with a secure, efficient, and user-friendly solution for storing, retrieving, and managing their passwords. This application serves as a centralized repository for users' passwords, ensuring that their sensitive information is stored in an encrypted format. By leveraging modern cryptographic techniques, the application protects users' data from unauthorized access. Additionally, the application includes a Password Strength Checker powered by Machine Learning to help users create strong and secure passwords.

1.2 ORGANIZATION PROFILE

Founded by a group of pioneering educators, we at Saintgits College of Engineering (Autonomous) seek to expose young minds to the world of technology and encourage all- round development of the mind. Our team of dedicated and caring faculty works with the student to reach academic excellence through a scientifically devised methodology. The high quality of our graduates, many of who have been university rank holders, and our distinctive results and placements are proof of the great distance we have travelled in 20 years. Saintgits College of Engineering has been granted Autonomous status by the University Grants Commission (UGC), making the institution one among the first three Engineering Colleges in Kerala to achieve this coveted status. The status was conferred in recognition of the academic excellence, expert faculty, industry connect, placement records, extracurricular activities and state of the art infrastructure offered by the institution. Saintgits is the only unaided institution in Kerala with 9 NBA accredited programmes.

1.3 OBJECTIVES OF THE PROJECT

The objective of the Password Manager Application project is to develop a secure, user- friendly solution for managing passwords, addressing the growing need for effective online security in today's digital landscape. The application aims to provide robust password storage through strong encryption methods, ensuring that sensitive data remains protected from unauthorized access. A reliable user authentication system will guarantee that only registered users can access their passwords. To enhance usability, the application will feature an intuitive interface that allows users to easily input, retrieve, and organize their passwords. Additionally, a password generation tool will encourage secure practices by offering strong, random passwords. The project also seeks to ensure cross-platform accessibility, enabling users to manage their passwords anytime, anywhere, while implementing backup and recovery options to safeguard data integrity. Ultimately, the application will educate users on best security practices, promoting safer online habits and contributing to overall cybersecurity awareness.

1.4 PURPOSE, SCOPE, AND APPLICABILITY OF THE PROJECT

1. Purpose

The purpose of a password manager project is to enhance online security by providing users with a secure and convenient way to manage their passwords. It simplifies the login process through features like auto-fill, saving users time and effort while encouraging the use of strong, unique passwords for each account. By centralizing the storage of passwords and sensitive information, the project makes it easier for users to access their credentials across multiple devices, promoting flexibility and ease of use. Additionally, it serves as an educational tool, raising awareness about password security and best practices for managing online identities. Furthermore, the password manager enables secure sharing of credentials with trusted individuals, facilitating collaboration while maintaining robust security measures. Overall, this project aims to significantly improve the security hygiene of individuals and organizations in the digital landscape.

2. Scope

The Password Manager Application is designed to provide a secure and user-friendly solution for managing passwords. Key features include password generation that allows users to create strong, customizable passwords, and a password strength checker that evaluates password security in real-time, offering suggestions for improvement based on factors like length, complexity, and entropy. The application ensures secure password storage with encryption, cross-platform compatibility for syncing across devices, and auto-fill capabilities for easy login. Additional features include secure password sharing, compliance with data protection standards, educational resources on password security, and tools for password backup and recovery. The goal is to deliver a comprehensive tool that enhances online security, simplifies password management, and encourages strong password practices for users across devices.

3. Applicability

The applications of a password manager project are wide-ranging and beneficial for various user groups, including:

1. **Individual Users:** Helping individuals securely manage their personal passwords, enhancing their online security and simplifying the login process across numerous websites and applications.
2. **Businesses and Organizations:** Assisting employees in managing work-related passwords, ensuring that sensitive business information is protected and reducing the risk of data breaches due to weak or reused passwords.
3. **Freelancers and Contractors:** Enabling secure management and sharing of passwords for different clients or projects, ensuring confidentiality and easy access to necessary tools.
4. **Family Sharing:** Allowing families to manage and share passwords securely among members, ensuring that

everyone has access to essential accounts without compromising security.

5. **Educational Institutions:** Supporting students and staff in managing their educational resources and accounts securely, fostering a culture of strong password practices.
6. **E-commerce and Online Services:** Providing a secure way for customers to manage their accounts on various platforms, enhancing user trust and security in online transactions.

CHAPTER 2 REQUIREMENTS AND ANALYSIS

INTRODUCTION

The development of a password manager involves several key requirements to ensure its functionality, security, and user experience. First and foremost, strong user authentication methods, such as multi-factor authentication (MFA), are essential to restrict access to authorized users only. Data encryption is critical, with all passwords and sensitive information securely encrypted both at rest and in transit to protect against unauthorized access. The application should offer a robust password generation feature that creates strong, unique passwords, minimizing the risk of reuse. Auto-fill capability for web forms is also vital for quickly logging into accounts without manual entry. Secure sharing options should enable users to share passwords with trusted individuals, while a user-friendly interface is necessary for ease of navigation. Password health monitoring features can provide insights into password strength and flag any weak or compromised passwords. Furthermore, the inclusion of backup and recovery options is essential to ensure that users can restore their information in case of loss. Finally, the project must adhere to relevant data protection regulations and best practices to build user trust and ensure compliance. By addressing these requirements, the password manager can deliver a secure, efficient, and user-friendly solution for managing passwords effectively.

2.1 PROBLEM DEFINITION

With the growth of digital services, password management has become a crucial challenge for users. Many struggle to create and store strong, unique passwords for their online accounts, often resorting to weak or reused passwords due to forgetfulness or the inconvenience of remembering multiple complex credentials. This issue leads to heightened risks of unauthorized access, security breaches, and identity theft. This project addresses three key issues: secure password storage, effective password generation, and ensuring password strength. Users need a reliable solution that securely stores passwords, protecting them from unauthorized access and data breaches. The project will employ strong encryption to ensure that sensitive information remains confidential and secure while providing an intuitive interface for easy management. Additionally, many users lack the knowledge or resources to create strong passwords capable of resisting cyber threats like guessing or brute-force attacks. This project will include a feature for generating strong, random passwords, allowing users to create unique and secure credentials for each account

and promoting better password practices to reduce password-related vulnerabilities. Beyond generation, the project will address the common problem of users unknowingly creating weak passwords. A password strength meter will guide users in creating passwords resistant to attacks, assessing criteria such as length, complexity, and character variety. This feature will encourage stronger password habits by providing real-time feedback and suggesting improvements. By addressing these challenges, this password manager will enhance account security, foster better password practices, and contribute to the reduction of identity theft and data breaches in an interconnected digital world.

2.2 PROPOSED SYSTEM-REQUIREMENTS SPECIFICATION

1. Software requirements:

1.1 Visual Studio Code

1.2 Python 3.12

1.3 Flask 2.1.1

1.4 SQLAlchemy

2.3 PLANNING AND SCHEDULING

Planning Phases involved in the Project Design:

Phase 1: Define the Application's Scope and Features.

1. Defining the core features for the application.
2. UI/UX Design.
3. Installing Python 3.8.
4. Defining Additional features.

Phase 2: Choose the Technology Stack

1. Determining the backend modules and tools (Flask, Flask-WTF, SQLAlchemy).
2. Determining the frontend language (HTML, CSS, JavaScript).
3. Configuring the database and session.

4. Implementing security modules (Flask-Bcrypt, Flask-Login).

Phase 3: Plan the Database Schema.

1. Design the database to handle the data models for user's passwords and entry. (Users, Passwords, Entry)

Phase 4: Security Considerations

1. Encryption

2. User Authentication

3. Protection Against Attacks

Phase 5: Define Application Architecture

1. Organize the app with the Model-View-Controller architecture.

2. Create the App Structure

2.4 SYSTEM SPECIFICATION

System specifications include software specification for development and implementation.

2.4.1 SOFTWARE SPECIFICATION FOR DEVELOPMENT, IMPLEMENTATION

1. Visual Studio Code

Some of the specifications of Visual Studio Code are:

- Version: 1.95.1 (user setup)
- Commit: 65edc4939843c90c34d61f4ce11704f09d3e5cb6
- Date: 2024-10-31T05:14:54.222Z
- Electron: 32.2.1
- ElectronBuildId: 10427718
- Chromium: 128.0.6613.186
- Node.js: 20.18.0
- V8: 12.8.374.38-electron.0
- OS: Windows_NT x64 10.0.22631
- Basic support: Syntax highlighting, bracket matching, code folding, and configurable snippets.

2. Python 3.12

Some of the specifications of Python 3.12 are:

- Release date: October 2, 2023
- Stable release: Python 3.12.1 (October 31, 2023)
- New syntax features: “match” statements support inline “case” expressions, error messages are clearer and more detailed, and more precise error locations are provided in traceback messages.
- New built-in features: “Self” type annotation for methods, “Override” and “Final” decorators are now standardized for better type hinting and inheritance control.
- New features in the standard library:
- A new “tomllib” module for reading TOML files directly.
- Enhanced logging configuration options with “logging.config.dictConfig”.
- Updates to the “zoneinfo” module for time zone handling.
- Improvements to “asyncio”, including a new “timeout” context manager.

3. Flask 2.1.1

Some of the specifications of Flask are:

- Release date: Initial release April 1, 2010.
- Current stable version: Flask 2.3.3 (as of September 2023).
- Framework type: Micro web framework (minimalistic and lightweight).
- Programming language: Python
- Core features:
- Routing: URL routing with dynamic URL support.
- Templating: Jinja2 templating engine integration.
- Session management: Secure cookies for client-side sessions.
- Extensions: Modular extensions for ORM, authentication, and more.
- Development server: Built-in development server with debugger and reloader.

2.4.2 SOFTWARE/TOOLS

Software /Tools required for execution of the project are given below:

1. Visual Studio Code

Visual Studio Code (VS Code) is a free, open-source code editor created by Microsoft. Lightweight yet powerful, it supports numerous programming languages and runs on Windows, macOS, and Linux. Key features include IntelliSense (smart code completion and syntax highlighting), a built-in debugger, Git integration, and a vast extensions marketplace. VS Code also supports remote development, allowing work on code hosted in containers, virtual machines, or even on remote servers. Highly customizable and widely used, it's a go-to tool for developers across various domains.

2. Python 3.12

Python 3.12, released on October 2, 2023, introduces several enhancements to syntax, error handling, and the standard library:

- Improved error messages: Clearer, more detailed error messages and precise traceback locations make debugging easier.
- New type hints: Self type and Override and Final decorators improve typing and inheritance support.
- Enhanced standard library: Includes a new tomllib module for reading TOML files, updated asyncio features, better time zone handling with zoneinfo, and new logging configurations.
- Syntax enhancements: Inline case expressions within match statements offer more flexibility in pattern matching.

Python 3.12 focuses on code readability, better diagnostics, and more robust type hinting, making it a strong choice for modern development.

3. Flask 2.1.1

Flask 2.1.1, released in March 2022, is a version of the popular lightweight Python web framework that introduces performance improvements, new configuration options, and enhanced support for asynchronous programming. Key features in Flask 2.1.1 include:

- Enhanced async support: Allows asynchronous route handlers, improving support for concurrent operations.
- Response caching: Improved caching capabilities for faster response times.
- Updated error handling: Better error messages and debugging information.
- Dependency updates: Compatibility with newer versions of Werkzeug and Jinja2, Flask's core dependencies.

Flask 2.1.1 continues to prioritize simplicity and flexibility, with added features to enhance both performance and developer experience.

4. SQLAlchemy

SQLAlchemy is a powerful SQL toolkit and Object Relational Mapping (ORM) library for Python, designed to simplify database interactions. It provides developers with high-level ORM capabilities, allowing them to work with databases using Python objects and classes

instead of raw SQL queries. SQLAlchemy also offers a robust Core system for those who prefer direct SQL expression language.

Key features:

- **ORM:** Maps Python classes to database tables, enabling object-oriented database interactions.
- **SQL Expression Language:** Provides flexibility to write raw SQL queries in a Pythonic way.
- **Database support:** Works with major databases, including PostgreSQL, MySQL, SQLite, and Oracle.
- **Session management:** Manages connections and transactions, making it easier to handle database sessions.
- **Migrations:** Integrates with Alembic for version control and schema migrations.

SQLAlchemy is known for balancing ease of use with advanced control, making it popular for both simple and complex database applications.

2.4.3 CONCEPTUAL MODELS

A conceptual model is the model of an application that the designers want users to understand. By using the software and perhaps reading its documentation, users build a model in their minds of how it works. It is best if the model that users build in their minds is like the one the designers intended. That is more likely if you design a clear conceptual model beforehand. Creating a conceptual model for a password manager application using Python Flask involves defining the core components and their relationships.

1. User

- **Definition:** Represents an individual using the password manager. Each user can store multiple passwords.
- **Attributes:**
 - **id:** Unique identifier for the user (Primary Key).
 - **username:** Unique username for the user.
 - **email:** Email address for the user, used for notifications and recovery.
 - **password_hash:** Securely hashed password for authentication.
 - **created_at:** Timestamp for when the user account was created.

2. PasswordEntry

- **Definition:** Represents a stored password entry belonging to a user. Each entry can contain details about a specific account.
- **Attributes:**
 - **id:** Unique identifier for the password entry (Primary Key).
 - **user_id:** Foreign Key linking to the User who owns this entry.
 - **site_name:** Name of the website or service for which the password is stored.
 - **username:** Username associated with the account for the site.
 - **password_hash:** Securely hashed password for the account.
 - **notes:** Additional notes related to the account (optional).
 - **created_at:** Timestamp for when the entry was created.
 - **updated_at:** Timestamp for when the entry was last modified.

3. EncryptionService

- **Definition:** A service responsible for encrypting and decrypting password data. This ensures that passwords are stored securely.
- **Methods:**
 - **encrypt(data):** Encrypts the given data (e.g., passwords).
 - **decrypt(encrypted_data):** Decrypts the given encrypted data.

4. AuthenticationService

- **Definition:** A service that manages user authentication and session handling.
- **Methods:**
 - **login(username, password):** Authenticates a user and starts a session.
 - **logout():** Ends the current user session.
 - **register(username, email, password):** Creates a new user account.
 - **reset_password(email):** Initiates a password reset process.

Relationships

- A User can have multiple “PasswordEntries”, but each “PasswordEntry” belongs to one specific User.
- The “EncryptionService” is utilized by both “PasswordEntry” and “AuthenticationService” to ensure that passwords are encrypted when stored and decrypted when accessed.

CHAPTER 3 SYSTEM DESIGN

3.1 BASIC MODULES

Creating a password manager application using Python and Flask involves several key modules and components to handle user authentication, password storage, encryption, and frontend interaction. Below are the primary modules required are listed below:

1. Flask:

A micro web framework for Python that allows developers to build web applications with minimal setup. It handles routing, HTTP requests, and responses.

2. Flask-WTF:

An extension that simplifies form handling in Flask applications. It provides form validation, CSRF protection, and integrates well with Flask templates for secure user input handling.

3. Flask-SQLAlchemy:

An extension that simplifies database integration with Flask using SQLAlchemy, an ORM (Object-Relational Mapper). It allows you to easily manage database records as Python objects, such as storing user credentials and encrypted passwords.

4. Flask-Login:

A module that manages user authentication in Flask applications. It handles user login, session management, and restricts access to certain views unless the user is authenticated.

5. Flask-Bcrypt:

An extension for securely hashing and checking passwords. It uses the bcrypt hashing algorithm to ensure that user passwords are stored safely, preventing plaintext storage.

6. Cryptography (Fernet):

A library used for encrypting and decrypting sensitive data (like passwords). It provides symmetric encryption using the Fernet algorithm, ensuring that passwords are stored securely in the database.

7. Flask-Mail:

An extension that enables sending emails from a Flask application. It is often used for sending password recovery or verification emails to users.

8. Flask-Session:

An extension that provides server-side session storage, allowing user session data (such as login state) to be stored on the server rather than in cookies, enhancing security and scalability.

9. Pandas library:

The Pandas library is a powerful, open-source data manipulation and analysis tool for Python. It is particularly useful in applications that involve handling structured data, as it provides data structures and functions to work with data tables (DataFrames) and time series.

10. Numpy library:

The NumPy library is a fundamental package for numerical computing in Python. It provides support for arrays, matrices, and high-level mathematical functions, making it essential for applications that require fast numerical computations.

11. Sklearn:

The scikit-learn (sklearn) library is a powerful tool for machine learning and data analysis in Python. It provides a wide range of algorithms for supervised and unsupervised learning, as well as tools for model evaluation and data preprocessing.

12. HTML/Jinja2 Templates:

Jinja2 is a templating engine integrated with Flask. It is used to render dynamic HTML content by embedding Python variables, loops, and conditional statements inside HTML templates for user-facing views.

3.2 PROCEDURAL DESIGN

Procedural Design refers to a method of software design that emphasizes the use of procedures (or functions) to structure a program. In procedural design, a program is broken down into a series of steps or instructions that are executed in a sequence to perform tasks or solve problems. Each procedure performs a specific task, and these tasks are executed in a specific order to achieve the desired outcome.



Figure 3.1: Procedural Design flow chart

3.3 USER INTERFACE DESIGN

The user interface design of the password manager application is focused on providing an intuitive, user-friendly experience that emphasizes security, ease of navigation, and simplicity. The design prioritizes a clean, minimalistic layout that allows users to quickly understand and access core functionalities, such as adding, viewing, editing, and deleting passwords.

Key elements of the UI design include:

1. **Navigation Bar:** A clearly organized navigation bar allows users to access essential features like “Add Entry,” “View Vault,” “Password Generation,” and “Settings” with ease.
2. **Password Vault Display:** The password vault presents saved entries in a structured, tabular format. Each entry is organized by account name, username, and a masked password field with an option to reveal or copy the password when needed.
3. **Add and Edit Password Forms:** Forms for adding or editing entries are designed with clearly labelled fields and secure password input fields. The layout ensures that users can input and save details without confusion.
4. **Password Generation Tool:** The password generation tool provides input fields for length and complexity options, along with an instant display of the generated password.
5. **Password Strength Indicator:** A visual password strength indicator is integrated within the add/edit password forms to guide users in creating strong passwords.
6. **Export Options:** The export feature, accessible from the settings or tools menu, provides users with options to save their data in CSV or JSON format, making it easy to back up or transfer information.

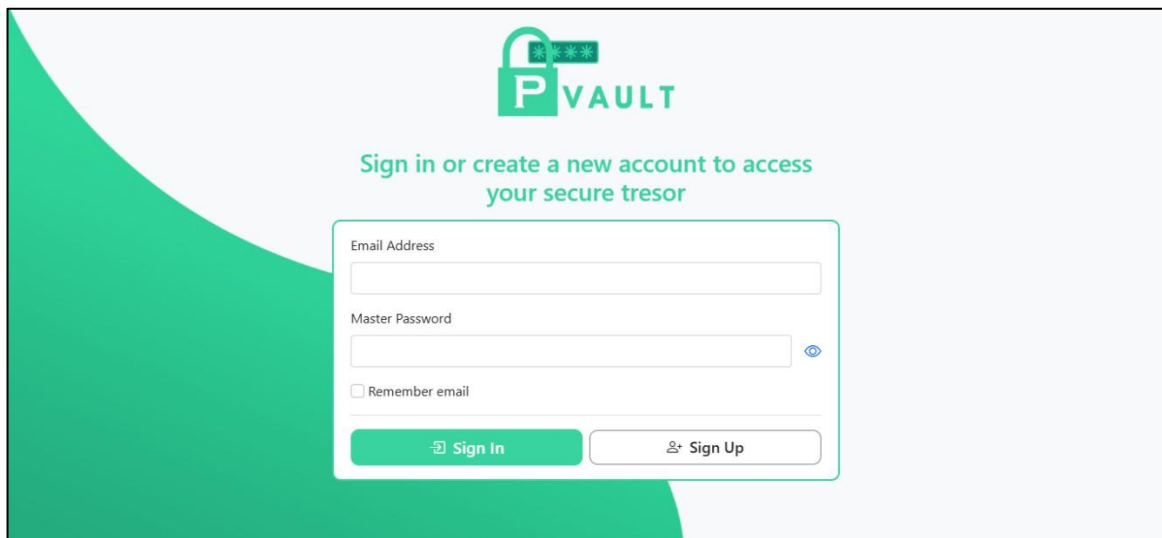


Figure 3.2: Sign-in page image.

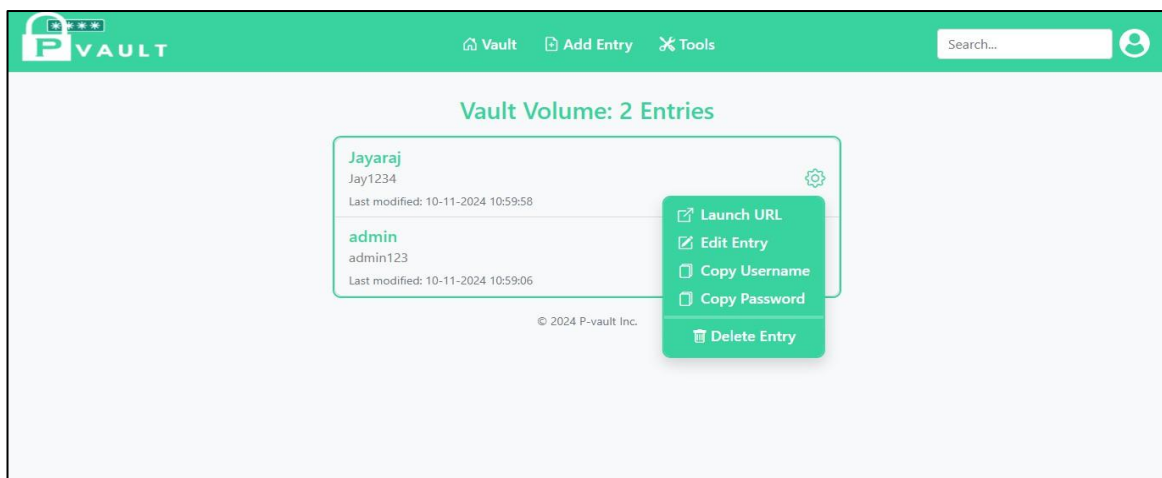


Figure 3.3: Dashboard image.

3.4 SECURITY ISSUES

Major security issues faced by Password Manager Application are:

1. Insecure Session Management

- Issue: Poor session handling, such as lack of session expiration or insecure session tokens, may lead to session hijacking, allowing unauthorized access to accounts.
- Solution: Implement secure session management with short session expiration times and renew tokens regularly. Use HTTP-only and secure cookies, and consider enabling two-factor authentication (2FA) to strengthen account security.

2. Lack of Protection Against Brute Force Attacks

- Issue: Attackers might try to guess user passwords by repeatedly attempting logins, especially if no rate limiting or account lockouts are implemented.
- Solution: Implement rate limiting and account lockout mechanisms after a certain number of failed attempts. Consider CAPTCHA or challenge-response tests for additional protection.

3. Session Management

- Issue: Poor session handling can lead to session hijacking, allowing attackers to gain unauthorized access.
- Solution with Flask: Set sessions to expire after a certain period and use secure and “httponly” flags on cookies. Implement Flask-Login or similar tools to manage user sessions securely and ensure that user authentication is robust.

4. HTTPS for Data Transmission

- Issue: Without encryption in transit, data can be intercepted, potentially revealing sensitive information.
- Solution: Use HTTPS with SSL/TLS to secure all data transmissions. Flask supports HTTPS, so ensure SSL certificates are configured correctly, especially in production.

CHAPTER 4 IMPLEMENTATION AND TESTING

4.1 IMPLEMENTATION APPROACHES

To implement a Password Manager App using Python Flask, we'll go through a step-by-step approach that includes backend logic for storing, retrieving, and managing passwords securely. Flask will be used as the web framework, and we'll use a combination of cryptography for encryption, SQLite for the database, and Flask's built-in features for user management and security.

Key Features:

- User Authentication (Master password login)
- Password Storage (Securely storing passwords in the database)
- Password Encryption (Encrypt passwords before storing them)
- Password Retrieval (Retrieve and view stored passwords)
- Password Generation (Generate strong random passwords)
- Password Strength Checker (Indicates the level of security for the password)
- Flask Web Interface (Front-end for user interaction)
- Data Exporting (Export data into “.csv” or “.json” format)

Key Libraries:

- Flask: Web framework for Python.
- Flask-SQLAlchemy: ORM for interacting with the database.
- Flask-WTF: For form handling.
- Flask-Login: For user authentication.
- Cryptography: For password encryption.
- Werkzeug: For password hashing.

4.2. INSTALLATION PROCEDURE

1. Installing Visual Studio Code

1. Run the Installer:

- Once the installer has downloaded, locate the file (usually in the Downloads folder) and double-click to run it.
- Double click and run it.

2. Follow the Installation Steps:

Step 1: Accept the License Agreement.

Read the license terms and click “I Accept” to continue.

Step 2: Choose Installation Folder.

You can either leave the default folder or select a different directory where you'd like to install VS Code.

Step 3: Select Additional Tasks:

Check the following options (these are optional but recommended):

- Add "Open with Code" to the context menu (allows opening folders directly from File Explorer with VS Code).
- Add "Open with Code" to the right-click menu (allows opening files and directories via the context menu).
- Add to PATH: This option is very useful, as it will allow you to open VS Code from the Command Prompt or PowerShell.
- Register VS Code as an editor for supported file types.

Step 4: Install:

- Install to begin
- Click Install to begin the installation. Wait for the process to complete.

3. Finish Installation:

After installation is complete, click the Finish button. You may also choose to launch Visual Studio Code directly after the installation.

4.3 TESTING APPROACH

Testing is a critical component in the development of any application, ensuring that the functionality works as expected and that the application is secure, reliable, and performs efficiently. For the Password Manager application built with Python Flask, a comprehensive testing approach was followed to verify both the functionality and security of the application. The testing approach was divided into multiple levels, including unit testing, integration testing, and security testing. It includes the following testing phases:

- Unit Testing
- Integration Testing
- Security Testing
- End-to-End Testing
- Performance Testing

4.3.1 UNIT TESTING

Unit testing focuses on testing individual components or functions in isolation to ensure that each part behaves as expected. In the case of the Password Manager, unit tests were written to test the core functionalities, such as:

- **User Registration:** Ensuring that a user can successfully register with a unique username and a securely hashed password.
- **Login Authentication:** Verifying that users can log in using correct credentials and are denied access when providing incorrect credentials.
- **Password Storage:** Ensuring that passwords are securely encrypted before storing them in the database and that they are properly decrypted when retrieved.

These tests ensure that:

- Users can register correctly with unique usernames.
- The system properly handles duplicate usernames.
- The password is correctly hashed and stored securely.

4.3.2 INTEGRATION TESTING

Integration testing checks if the components of the application work together as expected. For the Password Manager application, integration tests were written to verify how well the front-end and back-end interact, as well as how the application interacts with the database. Key integration tests included:

- **User Authentication:** Verifying that the user login flow is integrated with both the database and session management, ensuring that users can only access protected routes after successful authentication.
- **Password Entry Management:** Ensuring that users can securely add, view, and delete password entries, and that the data is correctly stored in and retrieved from the database.

4.3.3 SECURITY TESTING

Given that the Password Manager application deals with sensitive user data, security testing is of paramount importance. Key aspects of security testing included:

- **Password Hashing and Encryption:** Ensuring that passwords are hashed using a secure algorithm (e.g., bcrypt

or argon2) and that the application does not store plaintext passwords. This is tested by checking that stored passwords are never retrievable in their plaintext form.

- **SQL Injection:** Verifying that user inputs (such as usernames and passwords) are properly sanitized to prevent SQL injection attacks. This was tested by attempting to input malicious SQL code into user fields and verifying that it is safely handled.
- **Session Management:** Testing the session management mechanisms to ensure that users cannot access protected routes without authentication. This includes verifying that session cookies are properly secured (e.g., using HTTPOnly and Secure flags).

This test checks that the password is securely hashed and that the hash can be verified correctly during login.

4.3.4 END-TO-END TESTING

End-to-end (E2E) testing was performed to simulate real-world user interactions with the application. These tests ensure that the application functions as a whole and that all components (front-end, back-end, database) work together seamlessly. Selenium or Playwright can be used to simulate user actions in the web browser, such as:

- **Navigating the application:** Ensuring users can access the login page, register new accounts, and add/view passwords securely.
- **Data flow testing:** Verifying that password entries are added to the database and displayed correctly in the user interface.

For example, an E2E test might involve simulating the process of creating a new account, logging in, adding a password entry, and checking if the entry appears correctly in the UI.

4.3.5. PERFORMANCE TESTING

Finally, performance testing was conducted to ensure that the Password Manager can handle multiple simultaneous users, especially as it is likely to be used on a larger scale. This involved testing the application's response times under load and ensuring that database queries remain efficient even with a large number of users or password entries.

Tools like Locust or JMeter can be used to simulate traffic and measure how the application performs under stress, ensuring that it remains responsive and stable under various conditions.

4.4 RESEARCH METHODOLOGY

The research methodology for developing a password manager application using Python Flask involves a structured approach aimed at creating a secure, user-friendly solution for managing passwords. This methodology combines theoretical research, practical design, and testing processes to address both security and usability requirements.

Step1: Objectives and Scope

- Objective: Clearly define the goals of the password manager, such as securing password storage, enhancing user convenience, and implementing strong encryption.
- Scope: Outline the functionalities, including password generation, encryption, storage, retrieval, and any additional features like two-factor authentication or password health checks.

Step 2: Conduct a Literature Review

- Research Existing Solutions: Study existing password manager applications to understand current standards, usability practices, and security protocols.
- Explore Security Concepts: Review relevant security methodologies, including hashing algorithms (e.g., bcrypt, PBKDF2), encryption methods (e.g., AES), and secure storage techniques.
- Identify Gaps: Identify areas in existing solutions that could be improved, such as enhanced usability or stronger encryption practices.

Step 3: Define Research Questions or Hypotheses

- Formulate specific questions or hypotheses that the project aims to address.

Examples:

- “How does AES encryption affect application performance in securing passwords?”
- “Can a simpler, more intuitive interface encourage safer password practices?”
- These questions will guide the application’s design and help evaluate its effectiveness.

Step 4: Design the Application Architecture

- Frontend Design: Design a user-friendly interface using HTML, CSS, and JavaScript for easy navigation and intuitive user experience.
- Backend Development: Plan the Flask-based backend architecture to securely handle data processing and

storage.

- Database Structure: Design a secure data storage model, using encrypted fields to store sensitive information like passwords.
- Security Layers: Integrate libraries like cryptography and bcrypt to handle encryption and hashing securely.

Step 5: Develop and Implement the Application

- Frontend Implementation: Build and style the user interface with a focus on usability.
- Backend Implementation: Develop the core functionalities in Flask, including password storage, retrieval, encryption, and user authentication.
- Security Implementation: Apply encryption techniques (e.g., AES for encryption, bcrypt for hashing) and implement secure storage measures.

Step 6: Testing and Evaluation

- Functional Testing: Ensure each component of the application functions as expected, from password generation to encryption and retrieval.
- Security Testing: Conduct vulnerability testing to assess the application's resilience against attacks like brute force, SQL injection, and data leaks.
- Usability Testing: Gather user feedback to evaluate ease of use, identifying areas for improvement.

Step 7: Analyse and Document Findings

- Analyse Data: Assess security, performance, and usability data to evaluate whether the application meets the initial objectives.
- Document Results: Create a comprehensive report covering the development process, testing outcomes, and any insights from user feedback.

Step 8: Finalize and Deploy

- Make Improvements: Based on testing and analysis, make any necessary adjustments to enhance security, performance, or usability.
- Deploy: Launch the application, ensuring all security measures are in place and the user experience is smooth.

Following these steps ensures a systematic approach to developing a secure and user-friendly password

manager using Python Flask.

CHAPTER 5 RESULTS

5.1 RESULTS

The password manager can securely store passwords up to 12 characters in length. For longer passwords, i.e., more complex passwords, the program's performance (encryption and retrieval) may gradually degrade. The password retrieval accuracy is relatively strong for passwords in the range of 8 - 12 characters. Variations in special characters or uppercase letters have a minimal impact on the accuracy of password storage and retrieval. The optimal password length for the program is between 8 to 12 characters. The maximum supported password length is 20 characters, while the minimum supported length is 4 characters. The results from testing show that running the password manager on low-end CPU devices can be slow. However, speed can be improved by reducing encryption complexity, though this may impact security. In some cases, quick access may be prioritized over complex encryption, while in other cases, security takes precedence over speed. When implementing a password manager on a low-end device, this balance between speed and security often must be adjusted, and the results of these tests can aid in selecting the appropriate encryption method and storage format. The password's complexity (such as character variety) has minimal effect on the program's storage accuracy.

5.2 CONSTRUCTION

5.2.1 PASSWORD MANAGER PROTOTYPE

Building a prototype for the password manager was essential to evaluate its functionality, identify shortcomings, and refine it for the final version. The purpose of the iterative design process was to minimize issues in the final password manager. Key aspects considered were security and accessibility, aiming to ensure a secure yet user-friendly experience in the final product. One of the primary challenges was developing the password encryption and retrieval system, which required a secure framework to facilitate early testing of encryption algorithms and data storage components. The design goal was a streamlined, lightweight interface with robust encryption protocols. The system includes secure modules for storage and retrieval, each handling separate functions to maintain security and efficiency. For initial testing, a placeholder password entry interface was created, omitting some complex features like multi-factor authentication, which required additional setup time. This approach allowed for rapid testing of core functionality, such as password storage and retrieval. Due to quick setup, the initial prototype had limitations in user experience and

security, but it enabled faster testing of the encryption modules. The prototype, seen in the image below, was not fully optimized, but it allowed for early evaluation and laid the groundwork for refining the final password manager

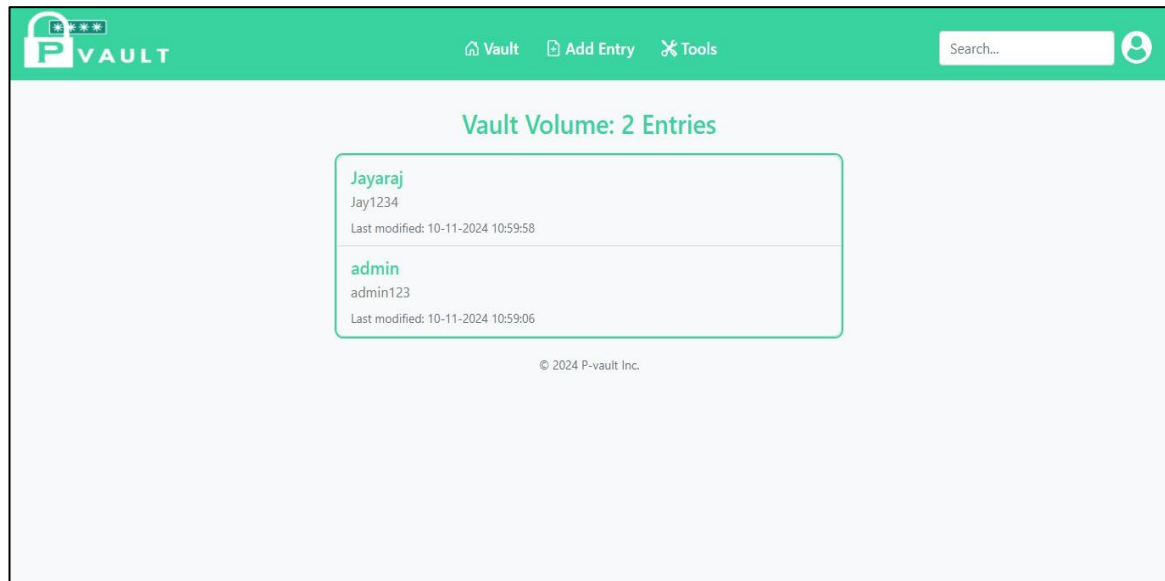


Figure 5.1: Dashboard image with few entries.

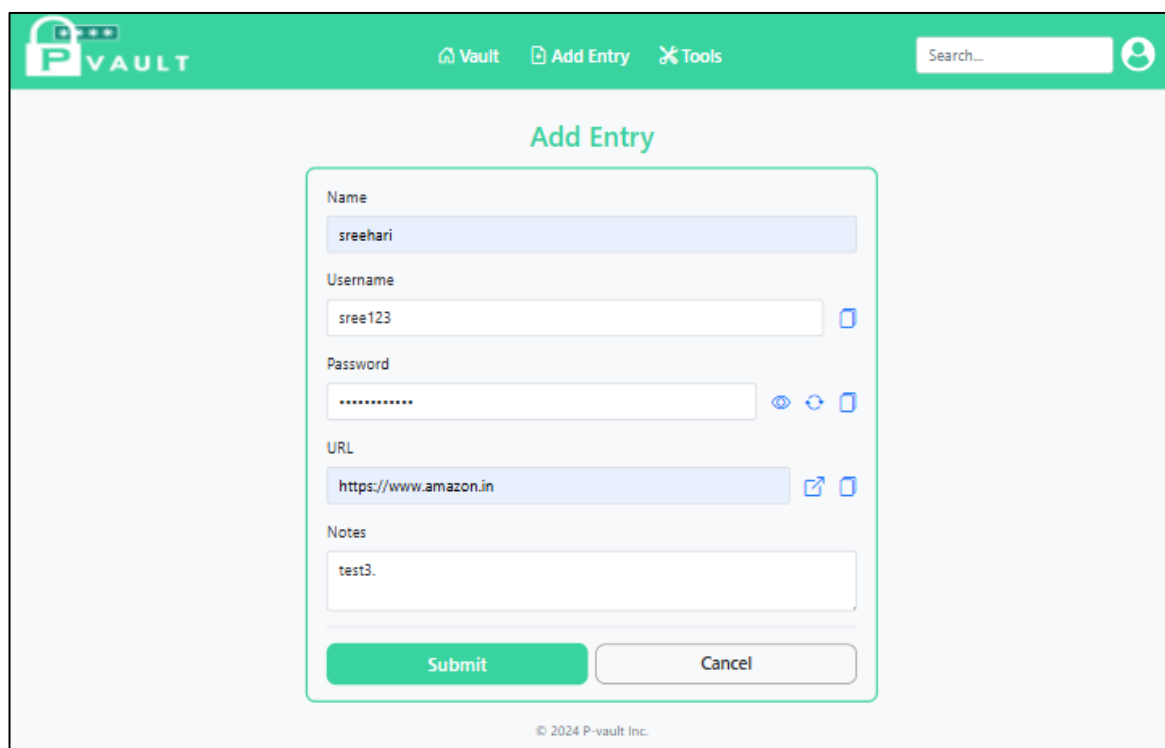
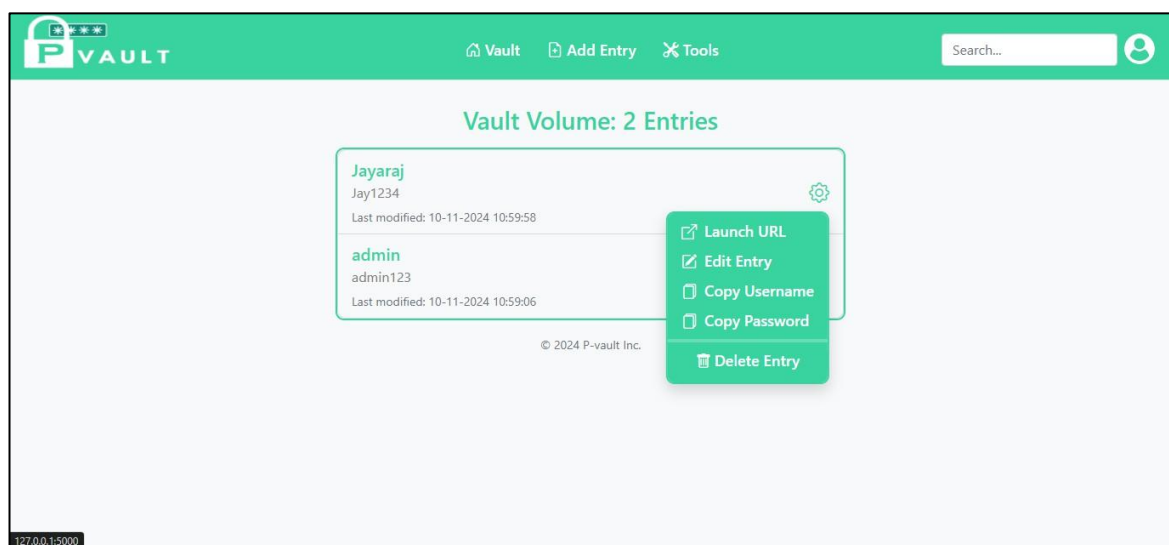


Figure 5.2: Add entry page image.

5.2.2 FINAL APPLICATION

A secure and stable platform was developed for the final password manager, incorporating additional features and improvements over the initial prototype. In the final model, a strong encryption mechanism was implemented, taking approximately 5.30 hours of configuration and testing within a secure environment. The encryption utilized AES-256, ensuring high security without requiring additional adjustments, allowing for efficient integration into the password manager. Additional components for password storage, retrieval, and encryption, password generation and password strength checking capability were also integrated, including a secure database and hash-based authentication. A rechargeable power supply was implemented to support offline storage access, providing flexibility for use on low- power devices. All hardware and software components were carefully assembled to achieve optimal security and usability. The development environment used for encryption and testing was Python-based, with coding done in PyCharm IDE. The password manager's user authentication was strengthened by training a machine learning model with data on common and complex password patterns, enhancing its ability to detect potentially insecure passwords. This model was trained using 288 unique password patterns gathered from a secure, open-source dataset. The password manager's secure storage was built on a robust structure to withstand potential threats, ensuring that all data was well-protected. The encryption and storage mechanisms were connected and tested thoroughly to prevent any data leakage or accessibility issues. Additionally, the authentication module was bolted into the application's framework, stabilizing access management and making password retrieval smooth and secure.

Figure 5.3: Dashboard image with few entries and edit options.



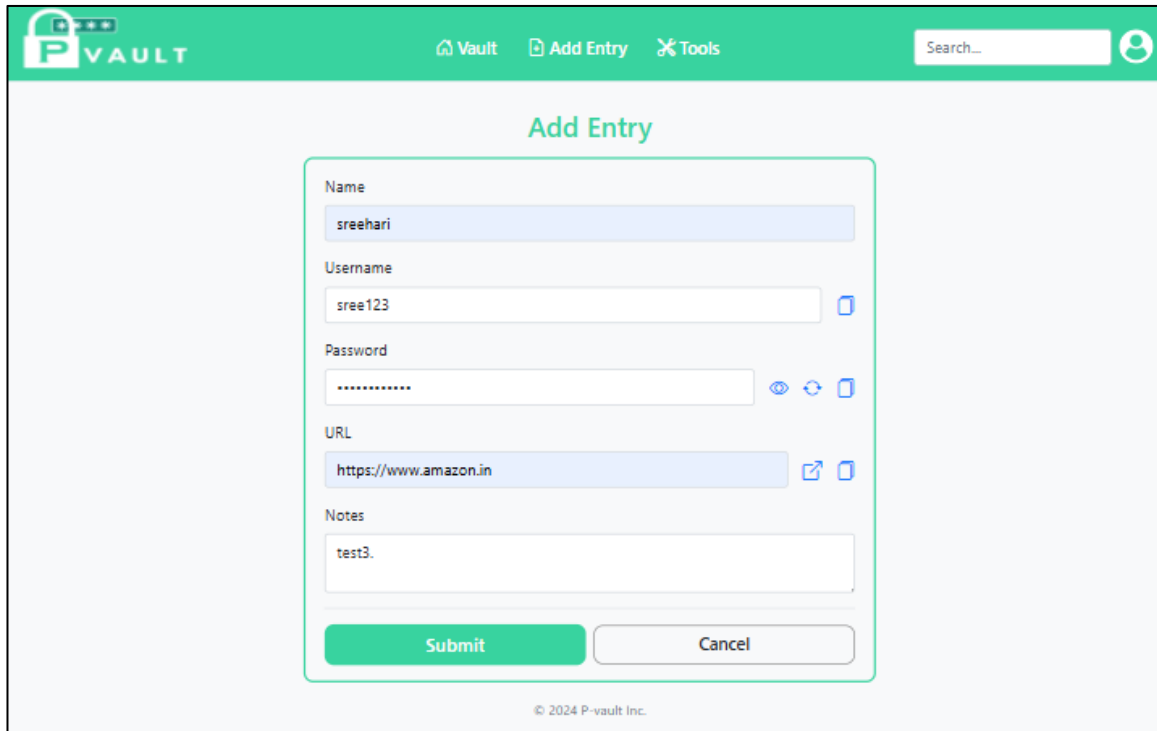


Figure 5.4 Add entry page.

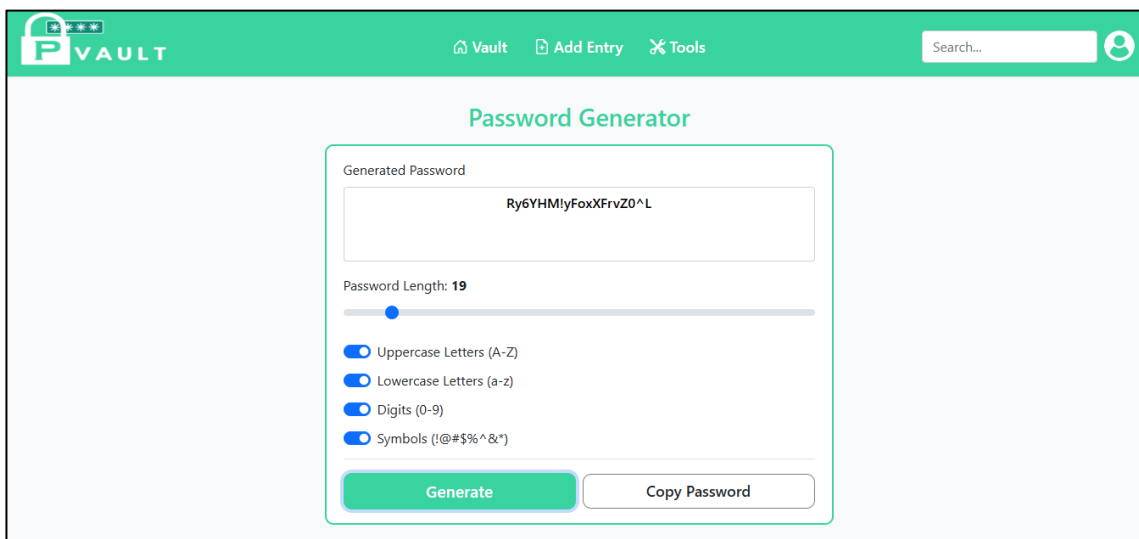


Figure 5.5: Password Generator page image

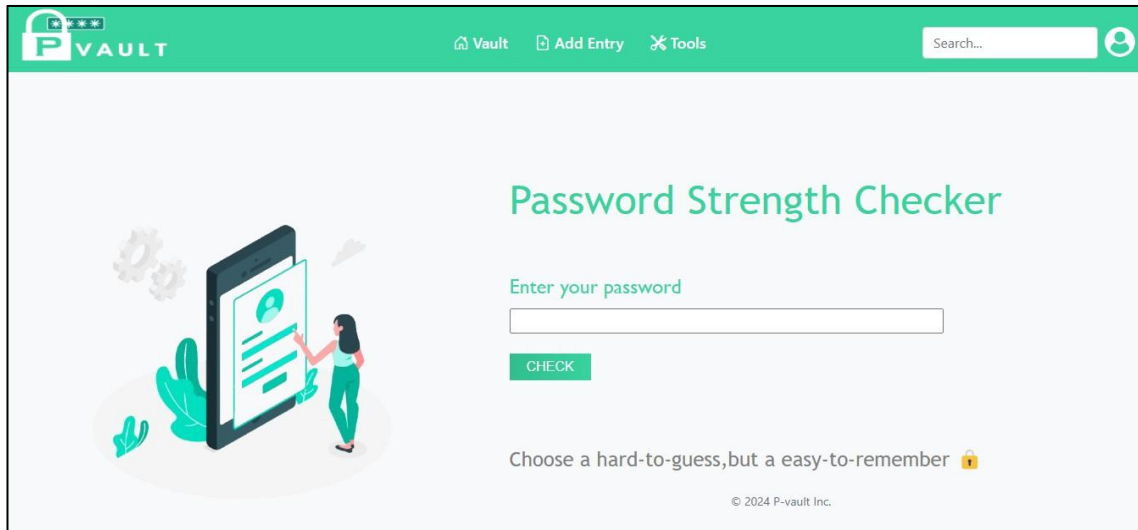


Figure 5.6: Password Strength Checker page image.

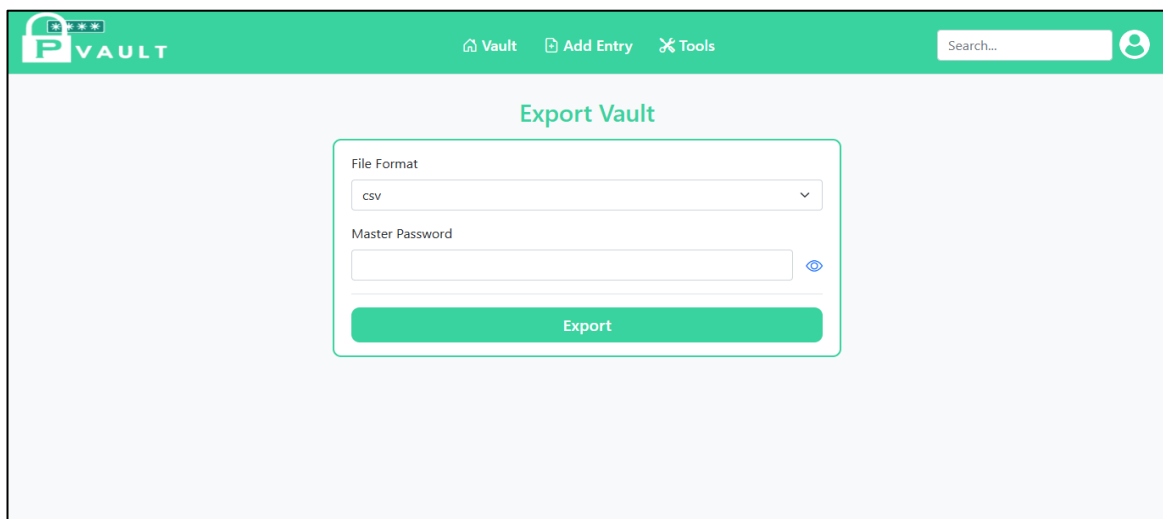


Figure 5.7: Export Vault image.

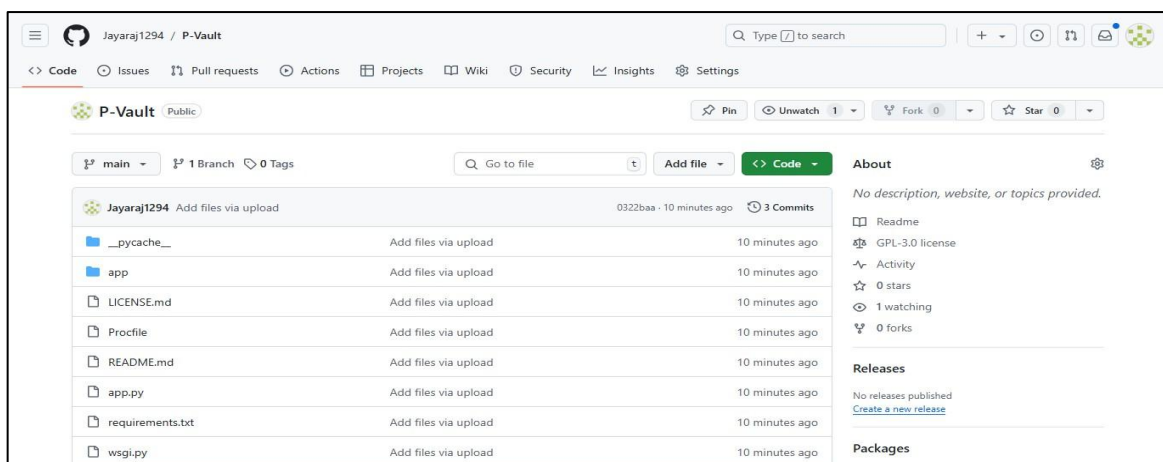


Figure 5.8: GitHub repository image.

5.3 EVALUATING THE PASSWORD MANAGER

The password manager was thoroughly tested to ensure it functions as intended, securely storing and retrieving passwords. During testing, the encryption algorithm took additional time to secure longer or more complex passwords, requiring multiple processing cycles to complete. This increase in processing time was accounted for in the results. Various tests were conducted under different conditions, including varying password lengths and complexities, to evaluate encryption strength and retrieval speed. The password manager achieved a mean processing speed of approximately 16.3 passwords per second for both storage and retrieval. The manager's performance varied slightly based on password complexity and the number of unique characters. In cases where passwords contained special characters or were particularly lengthy, processing times increased slightly. Additionally, testing under high encryption settings showed a slight reduction in speed, but with a noticeable improvement in security, ensuring that data was securely stored without the risk of leakage. New features were integrated into the password manager, including a password generation tool based on user input. This feature allows users to create passwords by specifying parameters such as length and complexity. The manager also includes a password strength checker, which provides real-time feedback on password strength, alerting users when their password is weak and suggesting improvements. Furthermore, the password manager now supports exporting saved data in CSV or JSON formats. This export feature allows users to back up their data or transfer it to other systems as needed. With these additional features, the password manager provides a robust, secure, and user-friendly solution for managing and safeguarding passwords.

5.4 TEST REPORTS

Test Report is a document which contains a summary of all test activities and final test results of a testing project. A test report summary contains all the details of the testing process, what was tested, when was it tested, how it was tested, and the environments where it was tested. Test report is an assessment of how well the Testing is performed. Based on the test report, stakeholders can evaluate the quality of the tested product and make a decision on the software release. The test report for the password manager application evaluates its security, functionality, usability, and performance to ensure whether it meets the defined objectives.

1. Introduction

- Objective: Verify that the password manager is secure, reliable, and user-friendly.
- Scope: Testing includes functional, security, usability, and performance aspects.

2. Test Types and Key Results

- Functional Testing: Ensures features like user registration, login, password storage/retrieval, and management

work correctly. Outcome: Most features pass; a few minor bugs identified and fixed.

- Security Testing: Assesses encryption, password hashing, protection against SQL injection, and session management. Outcome: Strong encryption verified; minor improvements recommended for session management.
- Usability Testing: Evaluates navigation ease, UI consistency, feedback messages, and responsiveness. Outcome: Generally user-friendly; slight adjustments in error messages and design consistency suggested.
- Performance Testing: Measures response time, load handling, and database efficiency. Outcome: Good performance under normal load; optimization suggested for high data volumes.

3. Summary of Results and Recommendations

1. Pass/Fail Summary: Majority of tests passed; any failed cases were addressed.
2. Issues and Fixes: Documented issues were resolved; improvements recommended for session security and UI feedback.

4. Conclusion

- Overall Assessment: Application meets core requirements; minor enhancements are advised.
- Next Steps: Implement suggested improvements and proceed to final testing and deployment.

CHAPTER 6 CONCLUSION

6.1 CONCLUSION

In conclusion, the Password Manager Application successfully fulfills its goal of providing a secure and efficient solution for managing passwords. By incorporating strong encryption, a user-friendly interface, and multi-platform support, it ensures that users can safely store and access their passwords across different devices. Features like password generation and two-factor authentication further enhance security, reducing the risk of unauthorized access. The project addressed key challenges related to encryption, usability, and cross-platform compatibility, ultimately delivering a tool that promotes better password hygiene and digital security. As digital threats continue to evolve, this application stands as an essential resource for safeguarding personal and sensitive information. Moving forward, additional features like biometric authentication or enhanced password sharing could be integrated to make the app even more versatile and secure. Overall, the Password Manager Application offers a reliable and comprehensive solution to the growing need for secure password management in today's digital world.

The development of the Password Manager Application has been an insightful and rewarding project, offering a comprehensive solution to the growing need for secure and convenient password management in today's

digital age. As individuals increasingly rely on online platforms for both personal and professional activities, the challenge of remembering and securing multiple complex passwords has become a significant concern. This application aims to solve this problem by providing a secure, user-friendly, and efficient tool that stores, organizes, and protects users' sensitive login credentials.

One of the core features of the Password Manager Application is its strong encryption capabilities. By utilizing advanced encryption algorithms, such as AES (Advanced Encryption Standard), the application ensures that all stored passwords are securely encrypted, making it virtually impossible for hackers or unauthorized individuals to access them. This focus on security is critical in an era where data breaches are becoming more frequent and sophisticated. By safeguarding passwords through encryption, the application protects users' sensitive information from potential threats, thereby offering peace of mind that their data is stored securely.

6.2 LIMITATIONS OF THE SYSTEM

While the Password Manager Application provides a highly secure and efficient solution for managing passwords, like any software, it has certain limitations that users should be aware of. These limitations stem from the challenges in balancing security, usability, and the varying needs of different users. The following are some key limitations of the system:

1. Single Point of Failure (Master Password Dependency):

The primary limitation of the system is that it relies on a master password to access the entire password vault. If a user forgets their master password, they may lose access to all their stored credentials. Although password recovery options (such as email or security questions) can help, there is no universal "reset" option due to the encrypted nature of the data, which helps maintain security but can be inconvenient if the master password is lost. While the master password is a crucial security feature, it also becomes a potential weak point. If the master password is weak or compromised, an attacker could gain access to all the stored passwords. Therefore, users must be diligent about selecting a strong, unique master password.

2. Vulnerability to Phishing Attacks:

Although the application employs encryption and two-factor authentication (2FA), it is still vulnerable to phishing attacks. Users may be tricked into entering their master password or 2FA codes on fake websites or applications that mimic the legitimate password manager. The app cannot fully protect users from social engineering attacks, which remain a common method of breaching security.

3. Performance Overhead:

The encryption and decryption processes required to secure passwords can sometimes introduce a performance overhead, especially on lower-end devices or systems with limited processing power. While this is usually minimal, users with older or less powerful hardware may notice slower performance when accessing or updating large password vaults.

4. Lack of Biometric Support (if not implemented):

Although some password managers offer biometric authentication (fingerprint or facial recognition) for quicker and more secure access, this feature may not be fully implemented in some versions of the application. If biometric support is lacking or not available across all platforms, users may still have to rely solely on their master password, which can be less convenient and less secure than biometric alternatives.

5. No Protection Against Local Device Theft:

While the application encrypts user data, it cannot fully protect against local device theft. If a user's device (e.g., smartphone, laptop, or tablet) is lost or stolen and the password manager application is not properly secured with a lock screen or other device-level security (e.g., PIN, password, or biometric), an attacker may potentially access the vault. While the master password and 2FA provide protection, the device itself must be secure to prevent unauthorized access.

6. Legal and Regulatory Compliance:

Depending on the user's location and the type of data being stored, the Password Manager Application may not be fully compliant with legal or regulatory standards. For example, businesses in certain industries may require compliance with GDPR, HIPAA, or other privacy regulations. If the application does not adhere to these standards, it could expose users to potential legal risks, particularly when storing sensitive data.

6.3 FUTURE SCOPE OF THE PROJECT

The Password Manager Application has the potential for continued evolution, offering numerous opportunities to enhance its functionality, security, and usability. As the digital landscape continues to evolve, new challenges and threats emerge, creating an ongoing need for stronger, more versatile password management solutions. The following outlines the future scope of the project, highlighting areas where the application can be expanded and improved to meet the evolving needs of users:

1. Integration of Biometric Authentication:

One of the most promising future enhancements for the Password Manager Application is the integration of biometric authentication. While many devices already support fingerprint scanning or facial recognition, the app could take advantage of this technology to streamline the user experience. With biometric authentication, users would be able to access their vaults more quickly and securely, without relying solely on passwords. This added layer of security would make unauthorized access even more difficult and could reduce the reliance on traditional master passwords, making the process faster and more user-friendly.

2. Cloud-Based Password Sharing for Teams and Families:

The ability to securely share passwords is becoming an increasingly important feature for families, teams, and organizations. The application could expand its password-sharing capabilities to allow users to share passwords securely with family members, colleagues, or teammates. Features could include:

- Granular permission controls: Allowing users to define what others can do with the shared passwords (e.g., view-only, edit, or delete).
- Secure password sharing links: Providing a secure, encrypted link for sharing passwords that self-destruct after being viewed.
- Team management tools: Allowing organizations to manage multiple users, assign access to different categories, and track who has access to what credentials.

Such features would make the app more suitable for both personal and business use, improving collaboration while maintaining security.

3. Integration with Password less Authentication Methods:

As the trend of password less authentication (e.g., via biometrics, OTPs, or authentication apps like Google Authenticator or Authy) continues to grow, the Password Manager Application could integrate with these emerging technologies. By offering support for password less login systems, such as WebAuthn, the app could support more advanced login methods that bypass traditional passwords altogether, improving security and convenience. This integration could position the app as a forward-thinking solution as online services increasingly move away from passwords.

4. Enhanced AI-Based Password Security Recommendations:

To help users maintain better security hygiene, the application could incorporate artificial intelligence (AI) to provide smart, dynamic password suggestions and security alerts. The AI could:

- Analyse password strength: Automatically flag weak or reused passwords and recommend stronger alternatives.
- Monitor for breaches: Integrate with online databases to alert users if their passwords have been involved in a recent data breach.
- Generate context-aware recommendations: Offer more personalized password suggestions based on the type of service being used (e.g., suggesting a longer password for banking services or a simpler one for less critical accounts).

This could significantly enhance the security posture of users by proactively monitoring and improving their password habits.

5. Integration with Password Security Audits and Reports:

The app could include a security audit feature that evaluates the strength of all stored passwords and provides actionable recommendations. These audits could check for:

- Password reuse: Alerting users if they are using the same password across multiple accounts.
- Password strength: Highlighting weak passwords and suggesting stronger alternatives.
- Outdated passwords: Reminding users to update passwords for accounts that have not been changed in a long time.

This feature would be especially beneficial for users managing a large number of passwords, as it would allow them to easily assess the overall security of their vault and take steps to improve it.

6. Integration with Digital Identity Management Systems:

With the growing emphasis on digital identity management, integrating the Password Manager Application with other identity solutions (such as OAuth, SAML, or OpenID Connect) could allow users to manage not only their passwords but also their entire online identity. This could include storing digital credentials, security tokens, and private keys in a secure, encrypted vault, making it easier for users to manage their authentication across various online services.

CHAPTER 7 APPENDIX

7.1 SAMPLE SOURCE CODE/PSEUDO CODE

1. Main codes:

```
from app import create_app

if __name__ == '__main__':
    app = create_app()
    app.run()
```

Figure 7.1: Screenshot of Main codes.

2. Configuration codes:

```
from datetime import timedelta

class Config:
    SECRET_KEY = "2b518e810ee6f1bdcc8edacffcd09277"
    FERNET_KEY = "nQjLLEDZwf_dmoOEEIIIs4IVkBD7xXUbkbzqkrejchUxA="
    SQLALCHEMY_DATABASE_URI = "sqlite:///site.db"
    SESSION_PROTECTION = "strong"
    PERMANENT_SESSION_LIFETIME = timedelta(minutes=15)
    REMEMBER_COOKIE_DURATION = timedelta(days=7)
```

Figure 7.2: Screenshot of the configuration codes.

3. Connection codes:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt
from flask_login import LoginManager
from cryptography.fernet import Fernet
from app.config import Config

db = SQLAlchemy()
bcrypt = Bcrypt()
login_manager = LoginManager()
login_manager.login_view = "users.signin"
login_manager.login_message_category = "danger"
fernet = Fernet(Config.FERNET_KEY.encode("utf-8"))

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(Config)

    db.init_app(app)
    bcrypt.init_app(app)
    login_manager.init_app(app)

    from app.main.routes import main
    from app.users.routes import users
    from app.entries.routes import entries
    from app.errors.handlers import errors

    app.register_blueprint(main)
    app.register_blueprint(users)
    app.register_blueprint(entries)
    app.register_blueprint(errors)

    return app
```

Figure 7.3: Screenshot of the connection codes.

4. Database model codes:

```
from datetime import datetime
from flask_login import UserMixin
from sqlalchemy import UniqueConstraint
from app import db, login_manager
from app.entries.utils import decrypt

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(120), nullable=False)
    entries = db.relationship("Entry", backref="owner", cascade="all", lazy=True)

    def __repr__(self):
        return f"{self.name} <{self.email}>"

class Entry(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    last_modified = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    username = db.Column(db.String(120), nullable=False)
    password = db.Column(db.String(120), nullable=False)
    url = db.Column(db.String(120), nullable=False)
    notes = db.Column(db.String(1000), nullable=True)
    user_id = db.Column(db.Integer, db.ForeignKey("user.id"), nullable=False)

    __table_args__ = (UniqueConstraint("user_id", "name", name="user_entries"),)

    def __repr__(self):
        return f"{self.name} <{self.username}>"

def prettify(self):
    return {
        "ID": self.id,
        "Name": self.name,
        "Last Modified": self.last_modified.strftime('%d-%m-%Y %H:%M:%S'),
        "Username": self.username,
        "Password": decrypt(self.password).decode("utf-8"),
        "URL": self.url,
        "Notes": self.notes,
    }
```

Figure 7.4: Screenshot of database model codes.

5. Entry codes

- forms.py

```
from flask_wtf import FlaskForm
from wtforms import (IntegerField, StringField, PasswordField,
                     BooleanField, TextAreaField, SelectField)
from wtforms.validators import DataRequired, Length, URL

class EntryForm(FlaskForm):
    name = StringField("Name", validators=[DataRequired()])
    username = StringField("Username", validators=[DataRequired()])
    password = StringField("Password", validators=[DataRequired(), Length(min=8, max=128)])
    url = StringField("URL", validators=[DataRequired(), URL()])
    notes = TextAreaField("Notes")

class GeneratorForm(FlaskForm):
    password = TextAreaField("Generated Password")
    length = IntegerField("Password Length")
    uppercase = BooleanField("Uppercase Letters (A-Z)")
    lowercase = BooleanField("Lowercase Letters (a-z)")
    digits = BooleanField("Digits (0-9)")
    symbols = BooleanField("Symbols (!@#$%^&*)")

class ExportForm(FlaskForm):
    file_format = SelectField("File Format", choices=[("csv", "csv"), ("json", "json")])
    password = PasswordField("Master Password", validators=[DataRequired()])
```

Figure 7.5: Screenshot of entries forms.py code.

- Routes.py

```
import secrets
import string
from datetime import datetime
from flask import (Blueprint, render_template, url_for, flash, redirect,
                  request, abort, session)
from flask_login import current_user, login_required
from sqlalchemy.exc import IntegrityError
from app import db, bcrypt
from app.models import Entry
from app.entries.utils import (encrypt, decrypt, check_generator,
                              check_password, to_csv, to_json)
from app.entries.forms import EntryForm, GeneratorForm, ExportForm

entries = Blueprint("entries", __name__)

@entries.route("/add", methods=["GET", "POST"])
@login_required
def add():
    form = EntryForm()
    if form.validate_on_submit():
        entry = Entry(
            name=form.name.data,
            username=form.username.data,
            password=encrypt((form.password.data).encode("utf-8")),
            url=form.url.data,
            notes=form.notes.data,
            owner=current_user,
        )
        db.session.add(entry)
        try:
            db.session.commit()
        except IntegrityError:
            db.session.rollback()
            flash("Entry already exists.", "danger")
            return redirect(url_for("entries.add"))
        flash("Entry has been successfully created.", "success")
        return redirect(url_for("main.index"))
    return render_template("add.html", title="Add Entry", form=form, text="Add Entry")

@entries.route("/edit/<int:entry_id>", methods=["GET", "POST"])
@login_required
def edit(entry_id):
    entry = Entry.query.get_or_404(entry_id)
    if entry.owner != current_user:
        abort(403)
    form = EntryForm()
```

```
if request.method == "GET":
    form.name.data = entry.name
    form.username.data = entry.username
    form.password.data = decrypt(entry.password).decode("utf-8")
    form.url.data = entry.url
    form.notes.data = entry.notes
elif form.validate_on_submit():
    entry.name = form.name.data
    entry.last_modified = datetime.utcnow()
    entry.username = form.username.data
    entry.password = encrypt((form.password.data).encode("utf-8"))
    entry.url = form.url.data
    entry.notes = form.notes.data
    try:
        db.session.commit()
    except IntegrityError:
        db.session.rollback()
        flash("Entry already exists.", "danger")
        return redirect(url_for("entries.edit", entry_id=entry.id))
    flash("Entry has been successfully updated.", "success")
    return redirect(url_for("main.index"))
return render_template("add.html", title="Edit Entry", form=form, text="Edit Entry")

@entries.route("/delete/<int:entry_id>", methods=["GET", "POST"])
@login_required
def delete(entry_id):
    entry = Entry.query.get_or_404(entry_id)
    if entry.owner != current_user:
        abort(403)
    db.session.delete(entry)
    db.session.commit()
    flash("Entry has been successfully deleted.", "success")
    return redirect(url_for("main.index"))

@entries.route("/tools/generator", methods=["GET", "POST"])
@login_required
def generator():
    form = GeneratorForm()
    if request.method == "GET":
        form.length.data = session.get("length", "12")
        form.uppercase.data = session.get("uppercase", "y")
        form.lowercase.data = session.get("lowercase", "y")
        form.digits.data = session.get("digits", "y")
```



```

        form.symbols.data = session.get("symbols", None)
    elif request.method == "POST":
        length = int(check_generator(request.form, "length", "12"))
        uppercase = (string.ascii_uppercase
                     if check_generator(request.form, "uppercase", "y") else "")
        lowercase = (string.ascii_lowercase
                    if check_generator(request.form, "lowercase", "y") else "")
        digits = string.digits if check_generator(request.form, "digits", "y") else ""
        symbols = "!@#$%^&*" if check_generator(request.form, "symbols", None) else ""
        alphabet = uppercase + lowercase + digits + symbols
    try:
        while True:
            password = "".join([secrets.choice(alphabet) for _ in range(length)])
            if (check_password(password, uppercase, lowercase, digits, symbols)):
                break
        return password
    except IndexError:
        return ""
    return render_template("generator.html", title="Password Generator", form=form)

@entries.route("/tools/export", methods=["GET", "POST"])
@login_required
def export():
    form = ExportForm()
    if form.validate_on_submit():
        if bcrypt.check_password_hash(current_user.password, form.password.data):
            session["download"] = "csv" if form.file_format.data == "csv" else "json"
            session.pop("csrf_token", None)
            return render_template("export.html", form=form, show_modal=True)
        else:
            flash("Master Password is incorrect.", "danger")
    return render_template("export.html", title="Export Tresor", form=form)

@entries.route("/download", methods=["GET", "POST"])
@login_required
def download():
    records = Entry.query.filter_by(owner=current_user)
    filename = f"passtresor-export-{datetime.utcnow().strftime('%Y%m%d%H%M%S')}"
    try:
        return to_csv(records, filename) if (session.pop("download") == "csv") else to_json(records, filename)
    except KeyError:
        return redirect(url_for("entries.export"))

```

Figure 7.6: Screenshot of entries routes codes.

- Utils.py

```
import re
import io
import csv
import json
from app import fernet
from flask import make_response, session

def encrypt(message):
    return fernet.encrypt(message)

def decrypt(token):
    return fernet.decrypt(token)

def check_generator(form, label, default):
    if form:
        form_value = form.get(label) if form.get(label) else None
    else:
        form_value = session[label] if label in session else default
    session[label] = form_value
    return form_value

def check_password(password, uppercase, lowercase, digits, symbols):
    uppercase_error = re.search(r"[A-Z]", password) is None if uppercase else False
    lowercase_error = re.search(r"[a-z]", password) is None if lowercase else False
    digit_error = re.search(r"\d", password) is None if digits else False
    symbol_error = re.search(r"[!@#%&*]", password) is None if symbols else False
    return not (uppercase_error or lowercase_error or digit_error or symbol_error)

def to_csv(records, filename):
    si = io.StringIO()
    cw = csv.writer(si)
    columns = ["ID", "Name", "Last Modified", "Username", "Password", "URL", "Notes"]
    data = [record.prettyify().values() for record in records]
    cw.writerow(columns)
    cw.writerows(data)
    output = make_response(si.getvalue())
    output.headers["Content-Disposition"] = f"attachment; filename={filename}.csv"
    output.headers["Content-type"] = "text/csv"
    return output

def to_json(records, filename):
    data = [record.prettyify() for record in records]
    output = make_response(json.dumps(data, indent=4))
    output.headers["Content-Disposition"] = f"attachment; filename={filename}.json"
    output.headers["Content-type"] = "application/json"
    return output
```

Figure 7.7: Screenshot of entries utils codes.

6. Dashboard setup code:

```
from flask import Blueprint, request, render_template, redirect, url_for
from flask_login import current_user
from app.models import Entry
from app.entries.utils import decrypt

main = Blueprint("main", __name__)

@main.route("/")
@main.route("/index")
def index():
    if current_user.is_authenticated:
        search = request.args.get("q", "", type=str)
        page = request.args.get("p", 1, type=int)
        entries = (
            Entry.query.filter_by(owner=current_user)
            .filter(Entry.name.contains(search))
            .order_by(Entry.name.asc())
            .paginate(page=page, per_page=10)
        )
        context = "Entry" if entries.total == 1 else "Entries"
        return render_template("index.html", entries=entries, context=context,
                               search=search, decrypt=decrypt)
    return redirect(url_for("users.signin"))
```

Figure 7.8: Screenshot of Database setup code.

7. Users' codes:

• forms.py

```
from flask_wtf import FlaskForm
from flask_login import current_user
from wtforms import StringField, BooleanField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Length, Email, EqualTo, ValidationError
from app.models import User

class SignupForm(FlaskForm):
    email = StringField("Email Address", validators=[DataRequired(), Email()])
    name = StringField("Your Name", validators=[DataRequired()])
    password = PasswordField("Master Password", validators=[DataRequired(), Length(min=8)])
    password_confirm = PasswordField("Re-type Master Password", validators=[DataRequired(), EqualTo("password")])

    def validate_email(self, email):
        email = User.query.filter_by(email=email.data).first()
        if email:
            raise ValidationError("Email address is already taken.")

class SigninForm(FlaskForm):
    email = StringField("Email Address", validators=[DataRequired(), Email()])
    password = PasswordField("Master Password", validators=[DataRequired()])
    remember = BooleanField("Remember email")

class ChangeNameForm(FlaskForm):
    name = StringField("Name", validators=[DataRequired(), Length(min=2, max=120)])
    name_submit = SubmitField("Change Name")
```

```
class ChangeEmailForm(FlaskForm):
    email = StringField("New Email Address", validators=[DataRequired(), Email()])
    password = PasswordField("Master Password", validators=[DataRequired()])
    email_submit = SubmitField("Change Email Address")

    def validate_email(self, email):
        if email.data != current_user.email:
            email = User.query.filter_by(email=email.data).first()
            if email:
                raise ValidationError("Email Address is already taken.")

class ChangePasswordForm(FlaskForm):
    old_password = PasswordField("Current Master Password", validators=[DataRequired()])
    new_password = PasswordField("New Master Password", validators=[DataRequired(), Length(min=8)])
    new_password_confirm = PasswordField("Re-type New Master Password", validators=[DataRequired(), EqualTo("new_password")])
    password_submit = SubmitField("Change Master Password")

class DeleteAccountForm(FlaskForm):
    password = PasswordField("Master Password", validators=[DataRequired()])
    delete_submit = SubmitField("Delete")
```

Figure 7.9: Screenshot of user forms codes.

• Routes.py

```
from flask import Blueprint, render_template, url_for, flash, redirect, session
from flask_login import login_user, current_user, logout_user, login_required
from app import db, bcrypt
from app.models import User
from app.users.forms import (
    ChangeEmailForm,
    DeleteAccountForm,
    SignupForm,
    SigninForm,
    ChangeNameForm,
    ChangePasswordForm,
)

users = Blueprint("users", __name__)

@users.route("/signup", methods=["GET", "POST"])
def signup():
    if current_user.is_authenticated:
        return redirect(url_for("main.index"))
    form = SignupForm()
    if form.validate_on_submit():
        hashed_password = bcrypt.generate_password_hash(form.password.data).decode("utf-8")
        user = User(email=form.email.data, name=form.name.data, password=hashed_password)
        db.session.add(user)
        db.session.commit()
        flash("Account has been successfully created.", "success")
        return redirect(url_for("users.signin"))
    return render_template("signup.html", title="Sign Up", form=form)

@users.route("/signin", methods=["GET", "POST"])
def signin():
    session.permanent = True
    if current_user.is_authenticated:
        return redirect(url_for("main.index"))
    form = SigninForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user and bcrypt.check_password_hash(user.password, form.password.data):
            login_user(user, remember=form.remember.data)
            return redirect(url_for("main.index"))
        else:
            flash("Email Address or Master Password is incorrect.", "danger")
    return render_template("signin.html", title="Sign In", form=form)
```

```
@users.route("/signout")
def signout():
    logout_user()
    return redirect(url_for("users.signin"))

@users.route("/account", methods=["GET", "POST"])
@login_required
def account():
    name_form = ChangeNameForm()
    if name_form.name_submit.data and name_form.validate():
        current_user.name = name_form.name.data
        db.session.commit()
        flash("Name has been successfully changed.", "success")
        return redirect(url_for("users.account"))
    email_form = ChangeEmailForm()
    if email_form.email_submit.data and email_form.validate():
        if bcrypt.check_password_hash(current_user.password, email_form.password.data):
            current_user.email = email_form.email.data
            db.session.commit()
            flash("Email has been successfully changed.", "success")
            return redirect(url_for("users.account"))
        else:
            flash("Master Password is incorrect.", "danger")
    password_form = ChangePasswordForm()
    if password_form.password_submit.data and password_form.validate():
        if bcrypt.check_password_hash(
            current_user.password, password_form.old_password.data
        ):
            current_user.password = bcrypt.generate_password_hash(
                password_form.new_password.data
            ).decode("utf-8")
            db.session.commit()
            flash("Master Password has been successfully changed.", "success")
            return redirect(url_for("users.account"))
        else:
            flash("Master Password is incorrect.", "danger")

    delete_form = DeleteAccountForm()
    if delete_form.delete_submit.data and delete_form.validate():
        if bcrypt.check_password_hash(current_user.password, delete_form.password.data):
            db.session.delete(current_user)
            db.session.commit()
            flash("Account has been successfully deleted.", "success")
            return redirect(url_for("users.signin"))
        else:
            flash("Master Password is incorrect.", "danger")
    return render_template(
        "account.html",
        title="My Account",
        name_form=name_form,
        email_form=email_form,
        password_form=password_form,
        delete_form=delete_form,
    )
```

Figure 7.10: Screenshot of user's routes codes.

8. Password Strength Checker code:

- routes.py

```
from flask import Flask, Blueprint, render_template, request
from app.strength.pass_check import *

strength = Blueprint("strength", __name__)

@strength.route("/")
def display():
    return render_template('strength.html')

@strength.route("/tools/check_strength", methods=['GET', 'POST'])
def check_strength(sum=""):
    if request.method == 'POST':
        password = request.form['pwd']
        x.append(password)
        ip = vectorizer.fit_transform(x)
        switcher = {
            0: "Weak password!!!",
            1: "Moderate password!!!",
            2: "Strong password!!!",
        }
        ans = m.predict(ip)
        return render_template('strength.html', sum=switcher.get(ans[-1], " "))
    else:
        return render_template('strength.html', sum=" ")
```

Figure 7.11: Screenshot of password strength checker routes codes.

- pass_check.py

```

import pandas as pd
import numpy as np
import warnings
import random

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer

warnings.filterwarnings('ignore')

def read_file():
    data = pd.read_csv('app/strength/data.csv', on_bad_lines='skip')
    return data

def preprocess(data):
    data.dropna(inplace=True)
    return data

def input_provider(data):
    data_array = np.array(data)
    random.shuffle(data_array)

    x = [labels[0] for labels in data_array]
    y = [labels[1] for labels in data_array]

    return x, y

def word_divide(text):
    char = []
    for i in text:
        char.append(i)
    return char

def vectorize(word_divide, x):
    vectorizer = TfidfVectorizer(tokenizer=word_divide)
    X = vectorizer.fit_transform(x)

    return X

def model(X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
    clf = LogisticRegression(random_state=101, multi_class='multinomial')
    clf.fit(X_train, y_train)

    return clf

def main():
    data = read_file()
    data = preprocess(data)
    x, y = input_provider(data)
    X = vectorize(word_divide, x)
    m = model(X, y)
    vectorizer = TfidfVectorizer(tokenizer=word_divide)
    return vectorizer, x, m

vectorizer, x, m = main()

```

Figure 7.12: Screenshot of password strength checker model training codes.

7.2 REFERENCES

1. Hanif Mohammed (2022). *E-vault*. <https://github.com/1Hanif1/Password-Manager>.
2. Evgeny Zorin (2022). *PassTresor*. <https://github.com/evgenyzorin/PassTresor>.
3. Akhil (2021). *Password-Strength-Checker*. <https://github.com/akhil-here/Password-Strength-Checker>.
4. Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media.
<https://flask.palletsprojects.com/>.