

TCS673 Software Engineering
Team X - Project Name
Software Design Document

<u>Team Member</u>	<u>Role(s)</u>	<u>Signature</u>	<u>Date</u>
Francis Pulikotil	Team Lead	<i>fxp</i>	<u>05/31/2021</u>
Kayla Bayusik	Security Lead	<i>Kayla B</i>	<u>05/31/2021</u>
Alexander Dewhirst	Design Leader	<i>abd</i>	<u>05/31/2021</u>
Zhaowei Gu	Quality Assurance	<i>Zhaowei Gu</i>	<u>5/31/2021</u>
Andrew Klimentyev	Configuration	<i>Aklm</i>	<u>5/31/2021</u>

Revision history

<u>Version</u>	<u>Author</u>	<u>Date</u>	<u>Change</u>
0	Group	05/31/2021	Initial version
1	Group	06/14/2021	Security Design and Key Algorithms

[Introduction](#)
[Software Architecture](#)
[Design Patterns](#)
[Key Algorithms](#)
[Classes and Methods](#)
[References](#)
[Glossary](#)

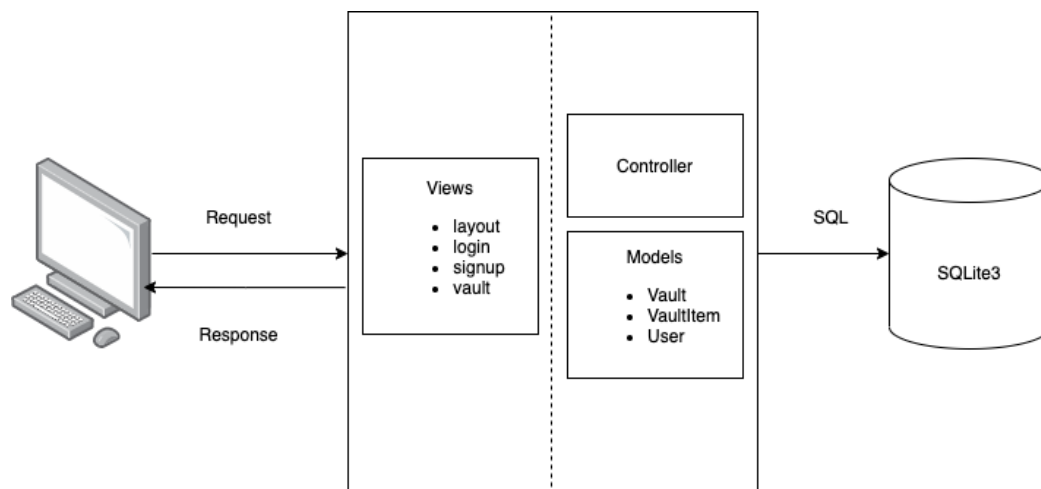
● Introduction

This document is to provide an overview and describe the design goals of the *PassMan* password manager web application. The *PassMan* application will be developed using Python/Flask and various helper libraries for the backend service. The *PassMan* frontend interface will be written using vanilla HTML5 and JavaScript, but templated using Jinja web templating engine. The backing database will be SQLite, which could then be migrated to PostgreSQL towards the end of the project. This document will describe the application architecture, database design, security design and choices, UI wireframes/mockups, as well as the major classes and methods which will be implemented.

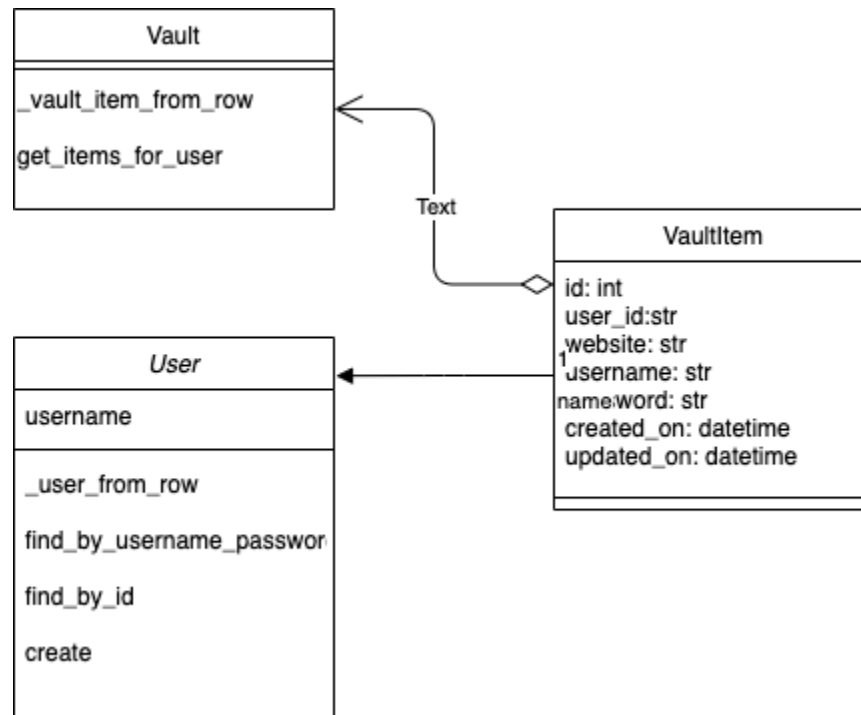
● Software Architecture

In this section, you will describe the decomposition of your software system, which includes each component (which may be in terms of package or folder) and the relationship between components. You shall have a diagram to show the whole architecture, and class diagrams for each component. The interface of each component and dependency between components should also be described. If any framework is used, it shall be defined here too. Database design should also be described if used.

Our application consists of a minimalistic Flask framework with a relational SQLite3 database. Within our Flask app, we have designed our server-side components with a similar flavor to a Model-View-Controller framework.



We currently have one Controller, which defines all RESTful routes within our app. This class responds to the request and performs interactions with our Model components, User and Vault. The Model components then contain the application's data structure and map to our database layer. After the Controller completes the necessary interactions, it then responds by fulfilling the request and rendering a View. Our Views represent the data returned from the controller.



We have integrated a few libraries to provide useful functionality and eliminate the need to duplicate standard practices.

Our password-based authentication design requires encryption and decryption, and we accomplish this with the *bcrypt* method from the *passlib* library. This method allows us to safely store and verify sensitive data with encryption and decryption, respectively. As security is a critical aspect of our application, leveraging well-maintained encryption libraries is essential.

Our Model classes *User* and *VaultItem* take advantage of a decorator method from the *dataclasses* library to easily implement strict-typing of the fields. This allows us to safely store data in our SQLite3 database without applying extensive type logic on user input.

Our SQLite3 database consists of two tables, *user* and *vault_item*, allowing us to store data in our application in a structure best suited for our application's purpose. We use common practices for field types and constraints in a relational database. We also have foreign key constraints for our *vault_items* to specify the *user* object each item belongs to.

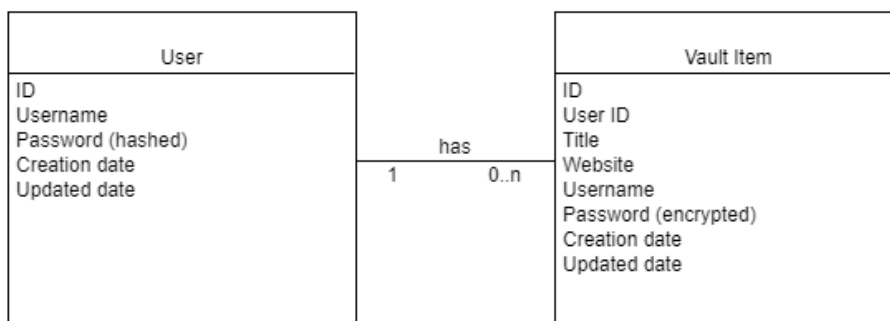
- Database Design

In this section, you shall describe any database if used in your software system.

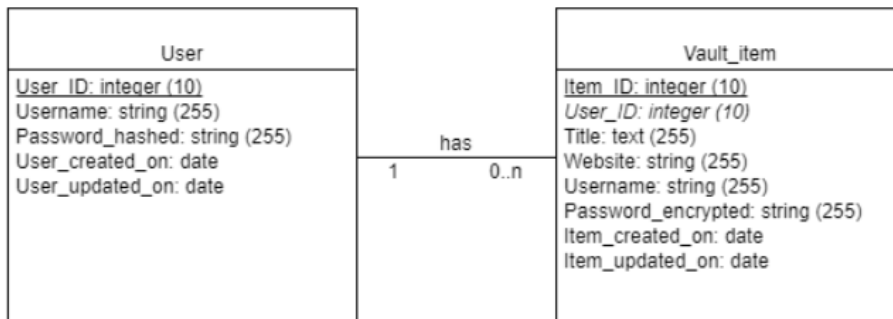
The design of our database is quite simple, since the only aspects of the application are the user themselves and their vault, which consists of all the stored items inside of it. Given this, our database design and models only needed two tables, one for the user data and one for the vault entry data. In the database design, the relationship from the user table to the vault item table is classified as optional-many, and the relationship from the vault item table to the user table is classified as mandatory-one. These classifications were determined as such because each user's vault can have many vault entries in it but a vault may also be empty, as is the case with a newly created account's vault, and because each vault item must be connected to a user who created and owns that item and can only be connected to one user. Each user and all their respective vault items track the creation date and last update date of the data in question, as a measure for the developers to check that all edits on the website are saved properly to the database.

The following models lay out the database schema. The logical model is the more simplistic model, which depicts the tables in the database, the relationships between them, and the variables stored within them. The physical model, according to database model standard, depicts everything from the logical model, in addition to underlining the table's primary key, italicizing the table's foreign key, showing each variable's data type, and the proposed length for all data types for which a length is relevant. Also in accordance with physical model standards, all data names taken from the logical model were changed to be one word (no spaces) and, for this model, the names were also made unique for clarity.

Database Logical Model:



Database Physical Model:



● Security Design

In this section, you shall describe any security design in your software system.

Since PassMan is a password manager application, security was paramount for our implementation. We covered the security of the user's master password, the user's active session, the user's stored vault passwords, and randomly generated passwords. We have implemented password requirements when a new user creates a master account password in the registration page. By requiring the new password to have a capital letter, a lowercase letter, and a number, we are making the password complexity higher and as a result making it harder to crack. In the next iteration, we will also implement a character length requirement to the newly entered passwords before they can be accepted as a user credential for further security. As for password security after the master password has been chosen, in order to keep the user's chosen master password secure, we have implemented password hashing using Passlib's bcrypt function. This function both salts and hashes the entered password before storing it in the database, since storing a password as plaintext would be incredibly insecure. In order for a user to log in to their account after creating it, the password they enter into the input field on the login page will be salted and hashed using the same library functions, then the hashed version will be compared to the stored hashed master password.

Once the user has logged in to their account, the PassMan application creates a session for them, so that all of their information will be safely stored within that session, including the displayed vault data and any generated cookies. Upon logging out of their account, the user's information will be destroyed along with the session instance. In the next iteration, the same will also happen when the user closes out of the browser tab since closing the tab should in effect be the same as specifically logging out.

Storing a user's actual vault entry passwords proves to be more complicated than simply hashing and comparing the hashes, since the item passwords need to be made visible to the user upon request. Hashing, by definition, is not two-way so once it has been done it cannot be undone. Thus, instead of hashing the passwords stored in the vault, we must encrypt them so that they can be decrypted on command. Encryption will be done using the National Institute of Standards and Technology's established algorithm of AES-256 and the key used within the AES encryption and decryption algorithms will be generated with password-based PBKDF2. The AES functionality is imported from the Pycrypto library using the Crypto Cipher AES class, and the PBKDF2

is imported from the Pycrypto library using the Crypto Protocol PBKDF2 class. The in-progress implementation of the encryption and decryption is shown in the Key Algorithms section below.

Finally, on the create or edit vault entry page, is the option to randomly generate a password for a certain website rather than manually enter one into the vault. If the user selects this option, the randomly generated passwords will follow the same length and complexity requirements as our users are given for their PassMan master password. Randomly generated passwords will still be stored via AES-256 encryption in the vault database, despite being 'random.'

According to the current scope of the application, PassMan will not check manually entered vault item passwords for meeting those same complexity standards, since presumably some of the passwords entered into PassMan are already in existence and use, but an optional feature, should the opportunity arise, may be checking manually entered vault passwords against these complexity requirements and perhaps gently suggesting that the user consider creating a new, more secure password for that website.

- **Design Patterns**

Within our Flask application, we took advantage of a Flask library called Blueprints. Blueprints allow us to componentize our resources for a more RESTful approach, similar to controllers in an MVC-framework. While not purely a controller, these decorated objects are composed together using the method ``create_blueprint``, to construct the entire ``url_map`` or routing of our application. This composition pattern allows us to map our models to Blueprints and delegate methods to those Blueprints that will not conflict with the global namespace. This approach allows us to separate our concerns and provide additional flexibility and scalability.

- **Key Algorithms**

In this section, you shall describe any key algorithms used in your software system, either in terms of pseudocode or flowchart.

For securely storing the passwords for user accounts, we use the bcrypt hashing algorithm via the passlib library. Our program makes a call to the library when creating a user password, returning a hashed password which is then stored. During login, we make another call to the library to verify the user inputted password against the stored hash.

In addition, the following algorithms have not yet been officially implemented in the Github repository but are in progress. Together these algorithms would allow for encryption and decryption of the passwords stored in users' password vault items. The first algorithm creates a key for use during AES-256 encryption and decryption by generating a salt based on the hashed password and passing both the password and the salt through the PBKDF2 class. The second algorithm generates a key from the first, then pads it accordingly and encrypts via Pycrypto Cipher AES and encodes the result. The third algorithm takes the second and does its steps in reverse to decrypt the stored password.

```
def generate_key(password):
    salt = bytes(bcrypt.hash(password))
    kdf = PBKDF2(password, salt, 64, 1000)
    key = kdf[:32]
    return key

def encrypt_256(raw, password):
    private_key = generate_key(password)
    raw = pad(raw)
    iv = Random.new().read(AES.block_size)
    cipher = AES.new(private_key, AES.MODE_CBC, iv)
    return base64.b64encode([iv + cipher.encrypt(raw)])

def decrypt_256(enc, password):
    private_key = generate_key(password)
    enc = base64.b64decode(enc)
    iv = enc[:16]
    cipher = AES.new(private_key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(enc[16:]))
```

Algorithm to encrypt the vault

Based on input from the Professor, we have formulated the following algorithm to encrypt the vault:

Encrypting the vault itself

- Generate a cryptographically strong random number (X) (e.g. using [secrets](#))
- Encrypt the user's vault with X
- Store the encrypted vault in the database

Encrypting X (the vault key)

- User provides password (P)
- Create a unique salt (S) from P (e.g. hash using bcrypt)
- Store S in the database for that specific user.
- Hash P with S to generate an encryption key (K) (e.g. using bcrypt or PBKDF2)
- Use K to encrypt the vault key X
- Store the encrypted vault key in the database

Decrypting the vault

- User provides password (P)
- Hash P with the user's salt (S) to regenerate the encryption key (K)
- Use K to decrypt the vault key X
- Use X to decrypt the user's vault

User changes password

- User provides old password (P) and new password (P2)
- Hash P with the user's salt (S) to regenerate the encryption key (K)

- Use K to decrypt the vault key X
- Hash P2 with S to generate a new encryption key (K2)
- Use K2 to encrypt the vault key X
- Store the new encrypted vault key in the database
- Now the next time the user wishes to access their vault, they will need to use P2

Other Notes

- We encrypt the user's vault using X and not directly using K so that we don't need to re-encrypt the entire vault whenever the user changes their password. We only need to re-encrypt X.

● UI Design

In this section, you can describe your UI design

The UI design for PassMan was created to be simple and elegant and to match the PassMan logo decided upon in iteration 0. The design utilizes the purple and pale blue colors of the logo as a theme throughout the application and the significant messages in the wireframes are shown in a (similar) cursive script as the text in the logo. The landing page was designed as a login page to eliminate the need for creating an additional page for users to choose either logging in or registering an account. The login page includes text input boxes for the user's account user name / email address and password. Below are buttons for logging in with the entered information, a forgotten password, and switching over to the registration page.

The registration page is akin to the login page, except with an additional text input box for confirming a created password. The buttons below allow the user to register with the entered information or switch over to the login page.

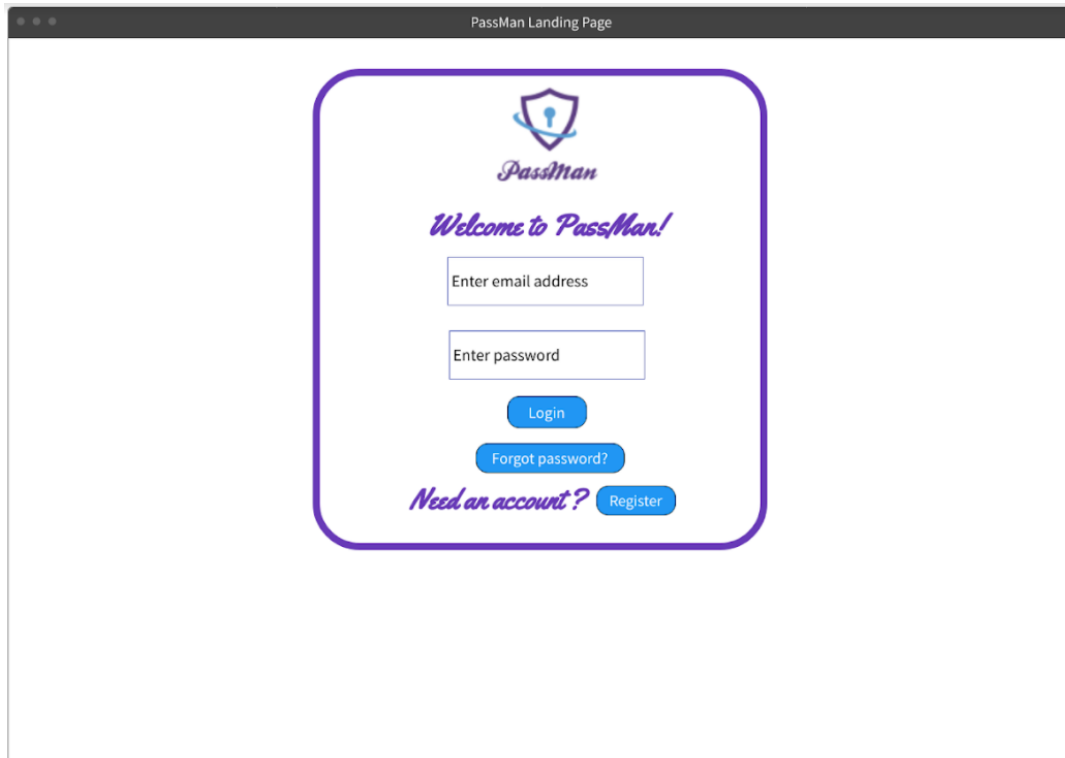
The vault home page design is likely to change as the project develops, but this version shows a table in the PassMan colors that contains the logged in user's stored vault items. Each item in the table has buttons for viewing the stored account details, editing them, and deleting the entry altogether. Each item in the table also shows its ID number in the vault, but this detail from the designs is likely not to make it into the final application as it is mostly for implementation and pre-production purposes. The vault also has buttons in the top bar for creating a new item in the vault and for logging out.

Finally, is the wireframe for creating a new entry in the password vault. This design drew from the login and registration layouts, utilizing the same clean, rounded form border and the same text input boxes and buttons. This design shows where the user can edit the name of their vault entry, edit the stored username, and edit the stored password. Alternatively, there is a button that asks the application to generate a random password for the user to use at that website and store that in the database rather than something the user came up with and manually entered. Just like the vault home page, this page also has a log out button, but it also includes a button beside that to return to the vault. Notably, in order to maximize work done, the new vault entry page will also double as or be repurposed as the edit vault entry page as well, since the fields available

on both pages will be the same and creating two pages for the same functions would not be a valuable use of time.


The wireframes are shown in the photos below, in the order they were described here.

Landing Page UI Wireframe Design:

A wireframe design for the PassMan Landing Page. The page is titled "PassMan Landing Page" in the browser's title bar. The main content is enclosed in a purple rounded rectangle. At the top is the PassMan logo, followed by the text "Welcome to PassMan!". Below this are two input fields: "Enter email address" and "Enter password". Under the password field is a "Login" button. Below the login button is a "Forgot password?" button. At the bottom of the rectangle is the text "Need an account?" followed by a "Register" button.

Registration Page UI Wireframe Design:


PassMan Register Page


PassMan
Sign up for PassMan?

Have an account ?

Vault Page UI Wireframe Design:


PassMan Vault Page


PassMan

Hello User!

Entry ID	Website			
1	Google	<input type="button" value="View"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
2	Amazon	<input type="button" value="View"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
3	Spotify	<input type="button" value="View"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
4	Netflix	<input type="button" value="View"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
5	Reddit	<input type="button" value="View"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>

Vault Entry Page UI Wireframe Design:



PassMan

Hello User!

Back to vault

Log out

New Entry

Entry title

Username

Password

OR

Generate secure password

Create Entry

- Classes and Methods

class User	
staticmethod	<ol style="list-style-type: none"> 1. <code>def _from_row(row)</code> 2. <code>def find_by_username_password(username, password)</code> 3. <code>def find_by_id(user_id)</code> 4. <code>def create(username, password)</code>

class Vault	
staticmethod	<ol style="list-style-type: none"> 1. <code>def _from_row(row, encryption_key: bytes)</code> 2. <code>def find_by_user(user_id, encryption_key: bytes)</code>

	<pre> 3. def find_by_id(user_id, vault_id, encryption_key: bytes) 4. def create(user_id, title, description, encryption_key: bytes) </pre>
--	--

class VaultItem	
staticmethod	<pre> 1. def make_empty() 2. def _from_row(row, encryption_key: bytes) 3. def find_by_vault(user_id, vault_id, encryption_key: bytes) 4. def find_by_id(user_id, vault_id, item_id, encryption_key: bytes) 5. def create(user_id, vault_id, title, website, username, password, encryption_key: bytes) 6. def update(user_id, vault_id, item_id, title, website, username, password, encryption_key: bytes) 7. def delete(user_id, vault_id, item_id) </pre>

- References
- Glossary