UNIVERSITY OF LONDON

Designing a User-Centric and Security-Focused Password Management Application: Implementation of Best Practices and Encryption Techniques

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Author:
Brendon Curmi

September 2023

Abstract

Passwords are the keys to the kingdom in the digital age. Most websites and services rely on passwords for user authentication, either directly or indirectly through other authentication methods such as OAuth, fingerprints, or biometrics, which only supplement passwords but cannot fully replace them. Since text-based passwords are so important to user security, cybersecurity experts recommend following good security practices. Whilst most people are vaguely aware of these recommended practices, it is very difficult for a person to create, manage, and remember long, complex, and unique passwords for every website and service. Password manager applications can provide users with the tools for taking their password security into their hands by alleviating and simplifying the process of password creation, storage, and management. This project develops a password manager that allows users to store and retrieve passwords, pin numbers, and notes in a secure way, that ensures data security while providing a positive user experience. It provides users with the ability to generate strong passwords, to accurately view the strength of their passwords, and to have an overview of the health of their security. The project is developed to educate users on strong password creation practices where possible, and to have a simple, straightforward, and intuitive design to allow users to quickly adopt and employ the application.

Contents

1 Introduction 4

2 Background and Related Work 4

2.1 Basis for Password Managers 4

2.2 Comparison of Existing Password Managers 5

2.2.1 Dashlane 5

2.2.2 1Password 6

2.2.3 LastPass 7

3 Methodology 8

3.1 System Architecture 8

3.1.1 Frontend 9

3.1.2 Backend 9

3.1.3 Database 10

3.2 Project Structure 11

3.3 Password Security 11

3.4 Cryptography 14

3.4.1 Authentication 14

3.4.2 Vault 16

3.5 Security 17

3.6 Project Management 20

3.7 Testing 21

4 Limitations 23

5 Results and Analysis 24

6 Evaluation 24

6.1 Evaluation Framework and Outcomes 24

6.2 Critical Evaluation 27

7 Conclusion and Future Work 28

7.1 Conclusion 28

7.2 Future Work 29

Glossary 29

Appendix A: Software Used 30

Appendix B: Application Set Up 30

Appendix C: REST API Documentation 34

Bibliography 45

1 Introduction

Everything online requires passwords for security purposes, from personal emails, to social media accounts, to banking services, to work and company accounts. Passwords function as the keys to everything in the modern world, and hence password security is extremely important. Cybersecurity experts recommend following security practices, such as having long passwords and using unique passwords for every service or website. Despite most people being vaguely aware of these principles, a lot of people do not follow them, as it is inconvenient or even impossible for most people to remember unique, long, and complex passwords, which puts their security at risk.

Users need to be provided with an organised way to create, store, and manage their passwords and other sensitive information. Password manager application are a good way of accomplishing this [3] however users tend to find them intimidating, risky, and untrustworthy [4]. Use of a password manager does not replace good password hygiene, but rather password managers should encourage users to employ strong passwords and educate users on the best practices to create strong passwords.

The right password manager solution needs to win user trust and ensure data security, while meeting an appropriate balance with user experience to ensure user security without burdening users. The user interface needs to be simple and intuitive to allow users to quickly adopt and use the application. It needs to provide an accurate evaluation of password strength to keep users updated on the quality of their passwords. The easy generation of strong passwords is an important feature to ensure users can quickly and effortlessly generate unique passwords for every service.

2 Background and Related Work

2.1 Basis for Password Managers

According to a study by the Ponemon Institute, 59% of IT security respondents reported that colleagues in their organisation use human memory to manage passwords, and 42% said sticky notes were used [1]. Sticky notes, notes apps, text or word documents are commonly used to manage passwords, even though these expose passwords in plain text, making them easy to compromise or steal. Human memory alone is also not a reliable way of managing passwords, as it might result in passwords that are not very long or complex as these would be harder to remember, especially with most websites having passwords policies requiring arbitrary metrics such as numbers and special characters. It also increases the chances of users reusing passwords across services to avoid remember multiple passwords, and risks losing access if the user forgets their password. According to the 2022 Verizon Data Breach report, 81% of hacking-related breaches leveraged either stolen and/or weak passwords [2]. Password strength, evaluation, and management need to be improved in businesses and for individuals.

One of the safest solutions to store and manage passwords is to use a password manager [3]. Despite this, only 22.5% of Americans use a password manager, and 65% of Americans do not even trust password managers, according to a PasswordManager.com YouGov survey [4]. In the survey, 30.5% of respondents said they do not trust password manager companies with their private information, and 34% said they worry their password manager could be hacked. This distrust is not completely unwarranted. In December 2022, LastPass, one of the most popular password manager applications, announced they had suffered a security breach where an unauthorised party gained access to their cloud-based storage environment and obtained pieces of customer information including backups of encrypted customer vault data [5].

An online survey of 248 people shows both users and non-users of password managers misunderstand the purpose of using a password manager, and do not think of the security benefits as being a main factor in their use of password managers [6]. From the respondents, password manager users said convenience and usefulness were the main reasons they used password managers, while non-users said security issues is the main reason they do not. These findings show that in order to persuade people to use password managers, a solution must earn the users' trust and be able to explain the technology and its purpose clearly and simply to the user.

2.2 Comparison of Existing Password Managers

2.2.1 Dashlane

Dashlane (https://www.dashlane.com) is one of the most popular password managers, used by tens of millions of users and thousands of businesses.

According to their whitepaper [7], all encryption and decryption happen locally on the user device. Encrypted sensitive data is stored in the cloud, encrypted using AES-256 and further strengthening it with Argon2d or PBKDF2. In the web app version of Dashlane, the user can change this cryptography setting to use [8]:

- Argon2d with 3 iterations and 32 MB memory cost
- or PBKDF2 with 200,000 iterations using SHA-256 hashing

The master password is never used in user authentication [7]. When a user creates a new Dashlane account or adds a new device to their account, Dashlane generates a unique random 320-bit key, using the OpenSSL `RAND_byte` function. Each device will have a different unique key. This key is sent from the Dashlane servers to the user device, where it is encrypted using the master password, and securely stored locally. When a user attempts to log into their Dashlane account or authenticate with the Dashlane servers, they input their master password, which decrypts the device key. The decrypted device key is communicated with the Dashlane servers to verify device identity and authenticate the user. This type of authentication is beneficial because the master password is never communicated to Dashlane servers in any way, and hence has no way of being intercepted or stolen since it is never transmitted or stored.

2.2.2 1Password

1Password (https://1password.com) is a popular password manager commonly used in commercial enterprises and businesses.

The 1Password Security Design Whitepaper [9] explains that the user Vault is encrypted using AES-256 before being stored in the database, and further strengthened with PBKDF2, using HMAC and SHA-256, with 650,000 iterations.

Security is handled by generating a salt from a non-secret salt and the user email address and other non-secret information and hashed using HKDF (Hash-based Key Derivation Function) to create a 32-byte salt. This newly generated salt is added to the trimmed and normalised user account password and passed through 650,000 iterations of PBKDF2-HMAC-SHA-256 to produce 32 bytes of data, which is combined with a processed machine-generated locally-stored 128-bit Secret Key, to produce the Account Unlock Key used to encrypt the private key that is used to decrypt the vault keys that are used to encrypt the database.

Through this process, decrypting the database requires two secret pieces of information: the user account password and the Secret Key. By using a combination of two different secrets, the data is more secure and harder to decrypt. This differentiates 1Password, as most other password managers only use the user account master password as the key for the encryption key. This creates a single point of failure, where a malicious actor could potentially decrypt a credentials database if the user password was compromised or easy to guess. Using this dual-key approach, if 1Password suffered a breach and a malicious actor got a copy of a credentials database and the user account password, they would still not be able to decrypt the data, without the locally-stored 128-bit Secret Key.

2.2.3 LastPass

LastPass (https://www.lastpass.com) is an award-winning password manager used by millions of people.

The LastPass whitepaper [10] explains that when a user creates an account on LastPass, the user creates their account username and password which are used to authenticate user access. The user password serves as a "Master Password," being partly used in the data encryption process and hence being the main password providing users access to their other passwords and sensitive information.

Entries in the user vault database are encrypted using the AES algorithm with 256-bit keys in Cipher-Block-Chaining mode, referred to as AES-256-CBC. On top of this, thousands of iterations of PBKDF2 are used to prevent brute-force attacks, dictionary attacks, and rainbow table attacks. LastPass uses PBKDF2 to generate the encryption key and user login hash, running 100,100 iterations client-side using SHA-256 hashing and another 100,100 iterations server-side using Scrypt hashing, to ensure data is protected end-to-end. Data is only ever encrypted and decrypted locally on the user device, and never happens on the server. The encrypted data is transmitted as a base64 encoded blob to the LastPass servers.

	Dashlane	**1Password**	**LastPass**
Encryption End	Local	Local	Local
Encryption Algorithm	AES-256	AES-256	AES-256-CBC
Hashing Algorithm	Argon2d or PBKDF2-SHA-256	PBKDF2-HMAC-SHA-256	PBKDF2-SHA-256 (local) PBKDF2-Scrypt (server)
Iterations	200,000 (PBKDF2)	650,000	100,100 (local) 100,100 (server)
Encryption Keys	Master Password	Master Password and 128-bit Secret Key	Master Password
Storage	Passwords, Secure Notes, Personal Information, Payment Methods,	Passwords, Secure Notes, Personal Information, Payment Methods,	Passwords, Secure Notes, Personal Information,

	Identification Documents	Bank Accounts, Identification Documents	Payment Methods, Bank Accounts
Password Generator	Yes	Yes	Yes

Table 1: A tabular summary of encryption algorithms, important features, and security practices in different password managers.

3 Methodology

An application named Guard was developed to help users create strong customisable and optionally memorable passwords, to securely store and retrieve passwords and other sensitive user information, and to aid users in managing their password security through health overviews and password strength evaluations.

3.1 System Architecture

The application was developed using a distributed architecture, with separate frontend, backend, and databases services, instead of a single monolithic application. This headless architecture allows the frontend and backend to be separate applications which can be developed, tested, and run independently from each other. They can even be owned by completely different development teams and employ different technologies. The loosely-coupled frontend and backend services communicate through a JSON REST API using HTTP requests. For the purpose of this project, both the frontend and backend services will run on the same machine, however in a real production environment, the frontend and backend can run separately as virtualised machines or containerised instances, either in the cloud to be accessible over the open internet, or on a local server to restrict access within a private intranet at home or within a business. This architecture also allows for different types of user-facing frontend technologies to be employed. This project implements a web-based frontend website, but in a real production environment, this can be extended to also include other technologies, such as phone apps, desktop applications, and browser extensions. If the application was used in a large business with heavy internet traffic, this architecture makes it easy to horizontally scale the project, as the business would only need to create more virtual machines or container instances. Below is an overview of the system architecture of this project.

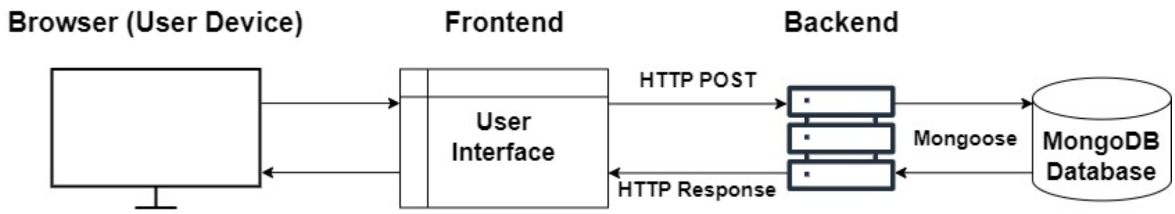

Figure 1: High-level overview of the architecture of the Guard project.

3.1.1 Frontend

The frontend is the locally-executed webpage that renders in the browser when the user visits the application. The frontend service was built using the React web framework and using JavaScript for code logic, and HTML5 and CSS3 for structuring and styling webpages. SASS was used as a CSS pre-processor, to prevent code duplication and improve styling code quality and efficiency. The SASS files are compiled into CSS files by the module bundler Webpack when the frontend service is launched. Node Package Manager (NPM) is used in the development environment to import and manage project dependencies, launch the application, and perform unit testing of code through the `Jest` testing framework. The frontend implements functional components, as opposed to class components, to define custom React components. The frontend mainly uses this functional programming paradigm because it is more concise to code and clearer to read and understand, since it does not require the added complexities of defining and managing objects. Another reason is because functional components provide the ability to use React hooks, which can be used to manage state and side effects clearly, which cannot be used in class components. A variation of the Factory Pattern is used for abstracting and centralising the information for the data types (accounts, pins, notes, etc). This makes the code more organised as the data for each type can be contained within their page files, and returned by the factory based on the passed parameters.

3.1.2 Backend

The backend is the cloud-executed service running on the servers of the application owner. The backend service was built using the `Node.js` runtime environment, the JavaScript language, the `Express.js` library to define the REST API service that enables communication with the frontend, and the `Mongoose` library to connect to and manipulate the database. NPM is used to manage project dependencies and launching the application. The backend mainly uses functional programming, but employs the object-oriented programming paradigm when defining data type controllers and data models. Employing both functional programming and object-oriented programming provides the benefits of both paradigms, with most of the codebase using functional programming to keep code concise and easy to understand, while also using object-oriented programming for modelling the data models of the database and to prevent code duplication in the data controllers. The files of the components are organised by component, instead of by file type. This is because organising by component makes the files easier to organise and traverse, and keeps related files within the same folder. The below diagram compares the file structure when organised by data type with the structure when organised by component. The structure appears clearer and easier to traverse and maintain when organised by component.

Organised by file type:	Organised by component:

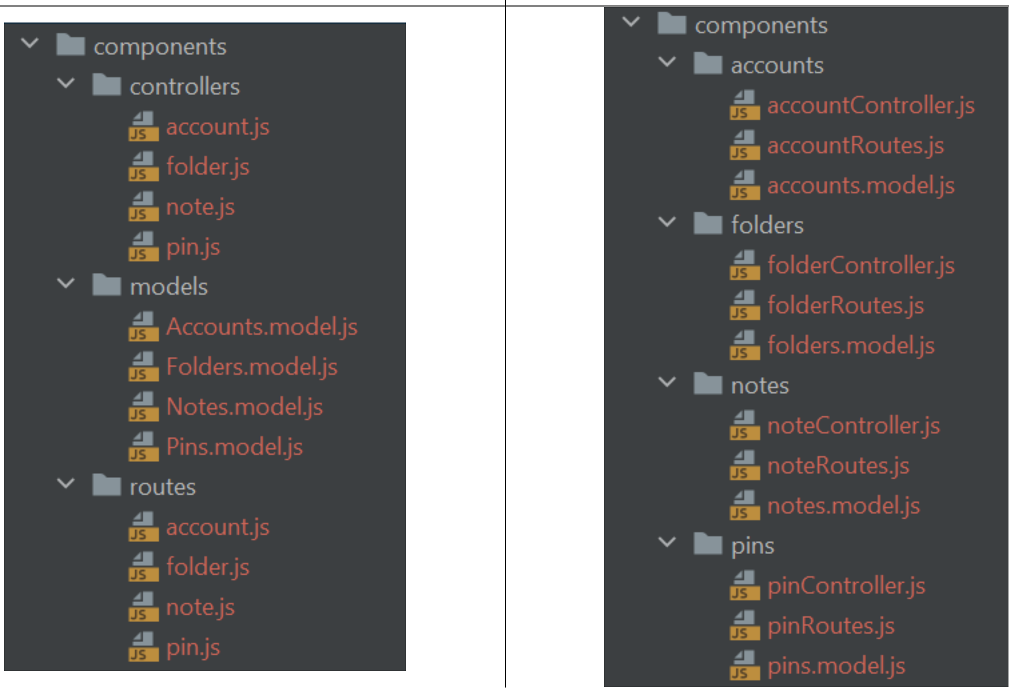

Figure 2: File structure and organisation comparison.

3.1.3 Database

The database is a MongoDB database called `guard_db` by default. MongoDB was used because it provides the flexibility of dynamic data schema and models, as a NoSQL solution, and easy integration with JavaScript since the documents can be retrieved as JSON objects. Each data type of confidential user data the application supports (accounts, notes, trash, etc) is implemented as a collection in the database, with the data modelled in the backend to define the schema of the collection. Since the application supports multiple users, the documents are associated with the appropriate user through manual referencing, where each document in the data type collections is associated with the user id from the user management collection. This is done to create a relationship between the documents, similar to the concept of foreign keys in relational databases. The relationship is managed manually through the controllers on the backend. An alternate way of structuring this in MongoDB is to implement the data type collections as embedded subdocuments in the user document. The current approach was chosen because it promotes data normalisation which reduces data redundancy and hence saves storage space, it is easier to maintain data integrity as changes need only be made to the referenced data and will be reflected across all uses of the data, and for scalability since breaking the data down into smaller related documents results in simpler queries and quicker read and write operations.

3.2 Project Structure

The project is named Guard, and contains a folder named `backend` containing the server-side backend code, and a folder named `frontend` containing the frontend code. These folders are treated as different projects and are different Node Package Manager (NPM) packages, each containing their own NPM dependencies and configurations. There is also a folder named `tests` which contains a JSON file containing a Postman collection for the project, which includes a folder for testing the backend endpoints and another folder for creating test data in the MongoDB database by making HTTP calls to the backend. By default, the frontend port is `8080`, the backend server port is `4000`, the MongoDB port is `27017`, and the MongoDB name is `guard_db`.

3.3 Password Security

Password Manager applications such as this, can make it more convenient to create and manage passwords, but fundamentally no Password Manager can protect a person if their passwords are weak. An individual is only as secure as their passwords. Hence, password security, strength, good practices, and education are important elements of this project.

Research into password strength by Shenoy [11] took 300,000 unique user-created passwords obtained from a dataset of 28,836,775 users and their passwords, and evaluated them from “Very Weak” to “Very Strong” based on attributes of password length, numbers, and special characters. These attributes are aligned with the digital identity guidelines issued by NIST [12], and are widely employed by many websites, even forming the basis of their password requirements or password strength evaluation metrics [13].

The research agrees with the suspicion that these password creation policies are not effective in creating strong passwords, as users find ways around them, such as writing down their passwords in plain text or reusing passwords across services. Since these password policies are not effective, it is important to understand what does make a password strong. Mathematically, the longer the password, the harder it will be to brute-force. The human problem is of course, a longer password is harder to remember, so it is important to stop looking at password as pass-words, and start looking at them as pass-phrases.

Passphrases are the strongest types of passwords [14], because they use spaces to make up sentences, instead of a single word made of a set of characters. This makes passwords longer and hence more secure, while also making them more human-readable and hence making them easier to remember. For example, the password “py7Wc!|n” fulfils the password requirements when creating a Microsoft account in September 2023, which are 8 characters, lowercase and uppercase characters, numbers, and special characters. PasswordMonster (https://www.passwordmonster.com/), an online password strength evaluator, categorises the password as “Strong” and predicts it would take 1 year to crack. However, this is not very human-readable or easy to remember. On the other hand, take a passphrase like “pouring h0ney WAS a m1stake” which is much longer, more readable, and easier to remember, even though it fulfils all the previous password requirements. PasswordMonster calls this “Very Strong” and predicts it would take 122 million years to crack. This shows it is more user friendly and secure to promote long memorable passphrases, instead of short passwords with arbitrary requirements, like including numbers and special characters. Some websites actively hinder user security by preventing the use of passphrases due to preventing the use of whitespaces, while promoting the standard view that numbers, uppercase, and special characters are what make a password secure.

This application was built to encourage passphrases, especially for the master password. To achieve this, the application does not set limits to password length or the use of whitespaces. The application password generator is implemented into the registration process, so users can automatically generate and modify strong complex passphrases to ensure their master password is readable, memorable, long, and secure.

The first password generation algorithm developed during this project built a string of possible characters, based on user-defined values to optionally add capital letters, digits, and special characters:

```
function getStringChars(params) {
    let chars = "abcdefghijklmnopqrstuvwxyz";
    if (params.useCapitals)
        chars += "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    if (params.useDigits)
        chars += "0123456789";
    if (params.useSymbols)
        chars += "@!$%&*";
    return chars;
}
```

A random character is chosen from this set each iteration for a user defined length, `L`. This generates a password of length `L` composed of random characters, including uppercase, digits, and special characters, as configured by the user in the `values` parameter. While this can generate strong, complex, and long passwords, they are not easy to pronounce or remember. To solve this, a second algorithm was developed to generate phonetic passwords in the form of `Consonant + (Vowel or Digit or Symbol)` for length `L`:

```
function generate(params) {
    const vowels = "aeiou";
    const consonants = "bcdfghjklmnpqrstvwxyz";
    const numbers = "0123456789";
    const symbols = "@!$%&* ";

    let password = "";

    for (let i = 0; i < params.length; i++) {
        let chars = "";
        if (i % 2 === 0)
            chars = consonants;
        else if (params.useDigits && Math.random() < 0.10)
            chars = numbers;
        else if (params.useSymbols && Math.random() < 0.10)
            chars = symbols;
        else
            chars = vowels;

        if (params.useCapitals && Math.random() < 0.10)
            chars = chars.toUpperCase();

        password += getChar(chars);
    }

    return password;
}
```

This second algorithm produces passwords formed from syllables and so are much easier to pronounce, with optional capitals, digits, and special characters including whitespaces. This is the final algorithm used for password generation in the application.

Since traditional password requirements do not necessarily translate to password strength [11], this also brings into question password strength evaluators. Password evaluators would not accurately reflect the strength of the password if the evaluation is based on the attributes of lowercase, uppercase, numbers, and special characters. For example, the LastPass online password strength tester (https://lastpass.com/howsecure.php) calls the password "Gl!sten1ng" to be "Moderately Strong" because it includes 10 characters, uppercase, lowercase, numbers, and special symbols. PasswordMonster considers this same password to be "Weak" and can be cracked in 1 hour. Due to this, a simple password evaluator based on these attributes was not developed since such evaluators are insufficient. Instead, the application employs the `zxcvbn` library for password strength evaluation. `zxcvbn` is a password strength estimator developed by Dropbox [15]. In order to train the evaluator to accurately measure password strength, it is trained on the same password leaks and dictionaries that attackers would use to compromise individuals. The password strength evaluator is implemented into the registration process, as research has shown users tend to create longer and more secure passwords when presented with a combined visual view and textual description of their password strength [22].

3.4 Cryptography

3.4.1 Authentication

During user registration, the user submits their username and user password, which acts as their Master Password. The Master Password is hashed using SHA-256, a strong hashing algorithm, using the known username as the salt. It is also passed through 1000 iterations of Password-Based Key Derivation Function 2 (PBKDF2), to further increase the computational cost to crack the password through brute-force attacks, dictionary attacks, and rainbow table attacks. This produces the Encryption Key, which is stored in-memory locally on the frontend and is used to encrypt and decrypt user Vault data. The Encryption Key is passed through another 10 iterations of PBKDF2 with SHA-256 hashing to produce the Authentication Key. The Authentication Key is encoded into a base64 string and transmitted to the backend through HTTP calls. The hashing process that derives the Encryption Key and Authentication Key is done on the frontend on the local device of the user to prevent the pre-hashed data from being intercepted by malicious actors. This project uses HTTP, but in a live environment, the application will use HTTPS which encrypts data communication using TLS encryption.

The Authentication Key is sent to the backend, where it is further hashed with Scrypt, which is a key derivation function which adds further security by making it more costly to crack a hash by requiring a large amount of memory. The generated hash is stored in the MongoDB database in a newly created object representing the new user, whose id is added to the payload of a JSON Web Token. A JSON Web Token (JWT) is a self-contained means of securely transmitting information between parties as a JSON object. The user id is included in the newly created JWT payload, and signed using a secret key, which is found in the backend `.env` file. For the purposes of this project, the `.env` file is public and committed to the GitHub repository, but the file would be kept private in a real live application. An HTTP 201 Created response is sent back to the frontend, along with the newly created JWT, which is sent as an HTTP-only cookie to prevent being read or collected by JavaScript. The JWT and cookie expire after 25 minutes to prevent a potential attacker from indefinitely impersonating a user if their user session was hijacked by stealing an active JWT or cookie. This process can be seen in the diagram below.

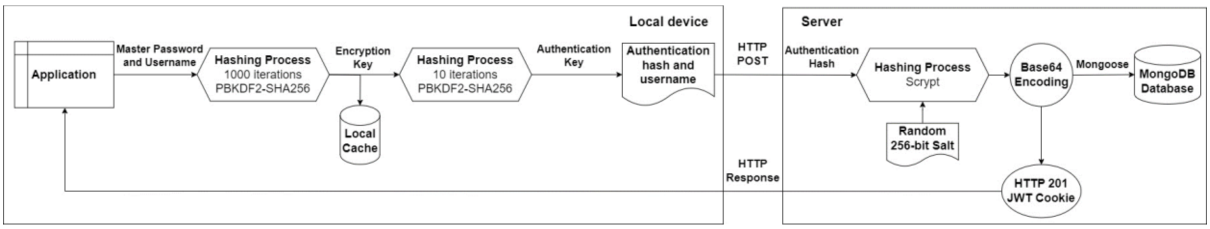

Figure 3: Overview of the user registration flow.

When a user attempts to login, they must submit their password and username, where the password is hashed using the same process described earlier. The resulting login Authentication Key is sent along with the username to the backend. The username is used to lookup the user in the database. If the user does not exist, an HTTP 400 Bad Request error is returned. HTTP 404 Not Found is not used, to avoid confusion for the endpoint itself not being found. If the username does exist, the stored user hash is loaded and extracted into the user salt and user Authentication Hash. The salt is used to hash the login Authentication Hash using Scrypt, and this new hash is compared to the user Authentication Hash. If they do not match, then the passwords do not match and the user is not verified, so an HTTP 401 Unauthorised response is sent. If they do match, then the passwords match and the user is verified. An HTTP 200 OK is sent back to the frontend, along with setting a JSON Web Token to verify user access. This process can be visualised with the below diagram.

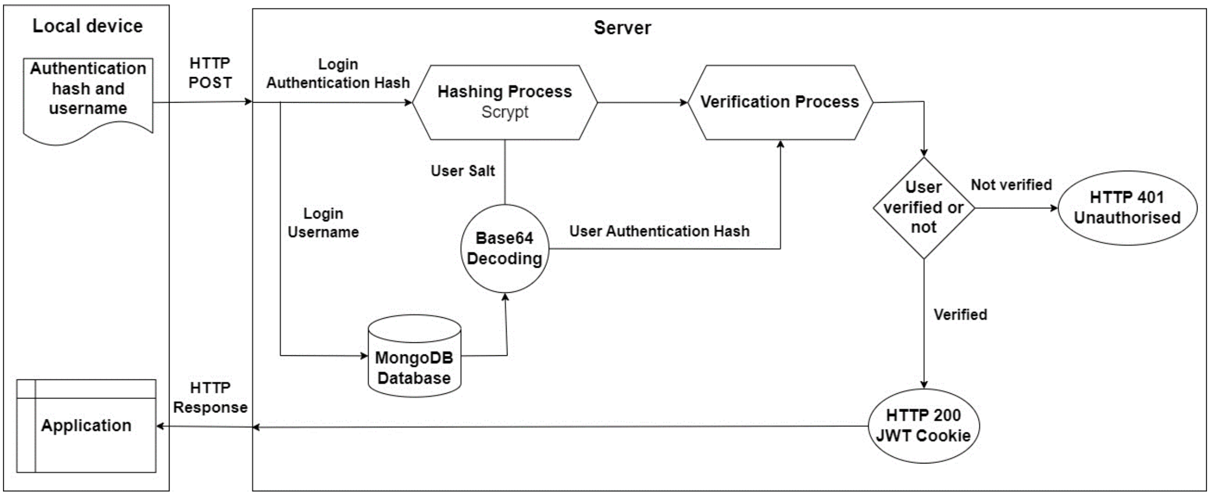

Figure 4: Overview of the user login flow.

3.4.2 Vault

Whilst data needed to be one-way hashed in the Authentication process, the Vault needed to be two-way encrypted to allow users to view their data on the frontend.

The previously generated Encryption Key and a randomly generated 128-bit salt are passed through 1000 iterations of PBKDF2 with SHA-256 hashing to generate a 256-bit key and 128-bit Initialisation Vector (IV), which increases security by making it more difficult for attackers to find patterns in the encrypted text or deducing the Encryption Key by introducing greater cryptographic variance.

The user data is encrypted using AES, which is a military-grade encryption used by the US government. The user data is passed into AES with the 256-bit key, Pkcs7 padding, and using Cipher Block Chaining mode, which all increases security through algorithmic complexity. The encrypted cipher text and original 128-bit salt are concatenated into a string in the form `salt$ciphertext`, and encoded in Base64.

Base64 is used as an encoding schema instead of another like Hex, which is Base16, because it is easily compatible with text formats like JSON which is used to store and transmit encrypted user data in the application, and because it is more space efficient since Base64 uses 4 characters to represent 3 bytes of data, and Hex uses 2 characters to represent 1 byte of data, making Base64 ~33.33% more space efficient than Hex.

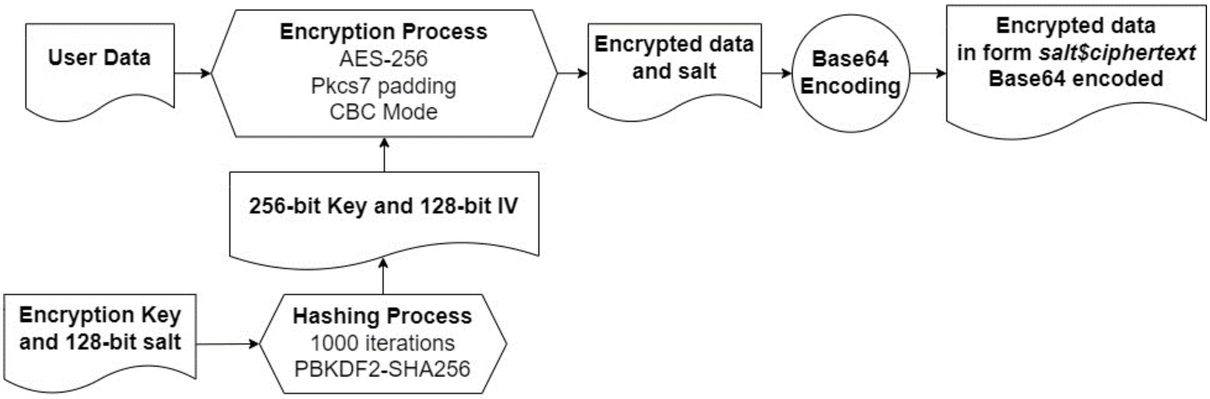

Figure 5: Overview of the encryption process.

When the user successfully logins in, the Vault data is loaded after the authentication process. The encrypted Vault data is stored in-memory, to prevent being intercepted from a semi-permanent storage location such as `localStorage`, and is cleared when the browser is reloaded.

The encrypted Vault data is decrypted on-the-fly when it is needed by the user. The data is decoded from Base64 to a string, which is split into the salt and encrypted text. The Encryption Key is loaded from memory and used with the salt to generate the 256-bit key and 128-bit IV using the process explained earlier. AES is used to decrypt the encrypted text using the key and IV, where the decrypted text is returned as a UTF-8 encoded string. The human-readable decrypted text is then shown to the user.

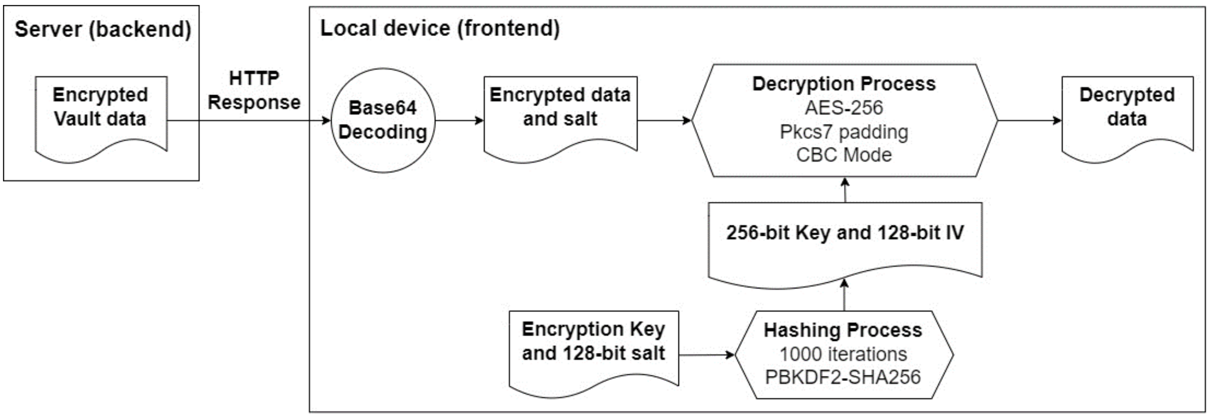

Figure 6: Overview of the decryption process.

3.5 Security

Security is very important for any application, especially for a password manager like this project, to ensure user credentials are not leaked or accessed by attackers. Hence, this project implements many layers of security to be as secure as possible.

All encryption and decryption of confidential user data is performed locally on the frontend on the user device, to prevent raw data being intercepted over the internet by an attacker. Raw confidential user data is never transmitted over the internet or stored in the database. Confidential user data is sent over the internet as an encrypted base64 encoded string. For the purposes of this project, the application is executed locally and over HTTP. In a real production environment, the application would be executed on servers with an SSL server certificate to communicate using HTTPS, which would encrypt the traffic data using TLS.

Both the encryption/decryption and hashing processes use 1,000 iterations of PBKDF2, which is a key derivation function that hashes user passwords to prevent brute-force attacks by increasing the computational cost to crack a password.

The user Master Password is hashed both on the frontend using PBKDF2-SHA-256 and again on the backend using Scrypt. This is to prevent compromising the user password in case the frontend-backend communication is intercepted. SHA-256 is a cryptographic hash function, while Scrypt is a password-based key derivation function, which means they are functionally and algorithmically independent of each other and hence employing both increases security through increasing cryptographic variance and complexity.

Once a user is authenticated and logged in, Vault data and the Encryption Key are not persistently stored anywhere on the frontend, such as cookies or in `localStorage`. They are stored in-memory as a cache using simple JavaScript variables, and hence are cleared every time the page is reloaded or closed, making it harder for an attacker to steal these details since they are not persisted, which means the user needs to insert their credentials every time they wish to access the application.

A JSON Web Token is sent from the backend to the frontend during successful login for user authentication. The Token is sent and stored on the frontend as an HttpOnly cookie, which prevents client-side JavaScript from accessing it or reading it, to prevent attackers from extracting the cookie and impersonating the user, or extracting the encrypted payload to acquire information. This Token is included in every subsequent HTTP request made from the frontend to the backend to securely verify the request is being made by a trusted authenticated user and which user it is specifically. The backend will refuse any attempt to communicate with endpoints other than the authentication endpoints, if the request does not contain a valid Token cookie to prevent unauthorised users from accessing these endpoints. The JWT and cookie each have an expiration time of 25 minutes, after which the user will need to log in again to receive a new JWT cookie. This is to prevent an attacker from indefinitely impersonating an authenticated user, if an attacker was able to obtain or steal the JWT cookie. The expiration time also prevents abandoned or forgotten open tabs from being taken advantage of by other individuals, in case users forget to log out of the application.

The frontend defines a Content Security Policy [18] to specify which resources are allowed to be loaded from which sources. This adds an extra layer of security that helps to prevent Cross-Site Scripting and data injection attacks by restricting resource loading. Content sources are restricted to the backend API, the frontend itself, and specific external fonts or services used in the application.

The backend defines a configuration for Cross-Origin Resource Sharing (CORS), which restricts resources loaded from a different domain. The backend CORS policy is defined to only accept requests from the frontend, which has domain `http://localhost:8080/` by default as defined in the `backend/.env` file. The backend also accepts requests from an undefined origin, to allow requests from REST API tools like Postman.

Cross-Site Scripting (XSS) [20] is a type of attack where an attacker inserts executable code into the existing code of an application, usually through user input fields or parameters. The backend uses the `xss-clean` middleware to escape and sanitise all request bodies, queries, and parameters to prevent attackers from injecting code into the REST API or database. To prevent XSS on the frontend, the React framework automatically escapes values before rendering them to the page [21]. Escaping encodes potentially dangerous characters into a string that will not be interpreted as code, while code sanitisation completely changes or removes the potentially dangerous characters. The backend should always receive hashed or encrypted data, and hence has the strictest user input cleaning by employing both escaping and sanitation, while only sanitation is employed on the frontend to allow users to insert special characters in input fields, such as passwords and secure notes. This was tested by attempting to pass JavaScript code as a user input, and the framework successfully prevented code injection by escaping the user input before rendering it to the page as a simple text string, as shown in the below image.

```
▼<div>
   <div><script>alert("XSS Attack");</script></div>
 </div>
```

Figure 7: Failed attempt to inject code in frontend.

The Axios library is used for performing HTTP calls. An external library is used, to avoid using the Fetch API to prevent attackers from overriding the `fetch` function to listen to each request. Below is an example of overriding and listening to the fetch function.

```
> window.fetch = (data) => console.log("Listening to fetch: " + data);
< (data) => console.log("Listening to fetch: " + data)
> fetch("confidential user info");
  Listening to fetch: confidential user info
```

Figure 8: JavaScript fetch command overridden to capture confidential user data.

The backend uses the `Mongoose` and `joi` libraries to define schemas for data validation to verify that all user data received in the backend as HTTP requests and that are attempted to be saved in the database follow the defined schema.

3.6 Project Management

Kanban was the chosen methodology employed during the development of the project, due to its active and flexible practices, as an Agile methodology. Kanban provides flexibility, focused development, and clear visibility into what is being worked on, what stage each task is in, and what is planned to be worked on soon. Kanban provides the ability to focus on tasks, while allowing the flexibility to reassess the workload based on changing priorities. The project did not have direct stakeholders that needed to be updated with progress. To keep track of project development, a Kanban board was set up for the project on Trello. The development of the project was broken down from high-level items into low-level actionable tasks that were created as cards on Trello. The different development stages were set up as columns to track the state of each development task.

Kanban Board on Trello: https://trello.com/b/oMeNIkgJ/guard-project-kanban-board
Project on personal GitHub: https://github.com/BrendonCurmi/Guard
Project on University GitHub: https://github.com/University-of-London/project-module-BrendonCurmi

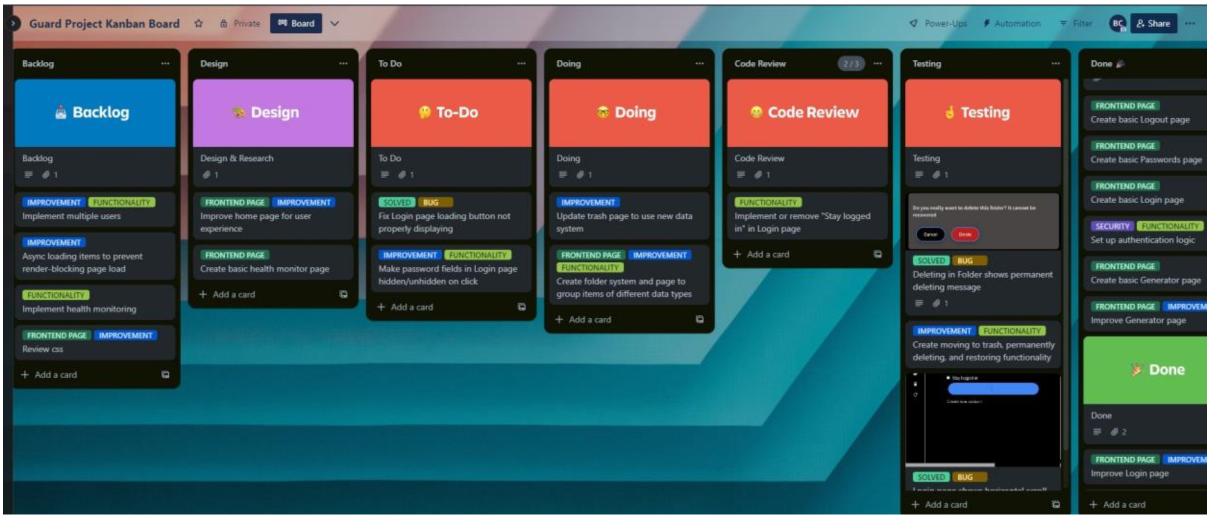

Figure 9: View of the project Kanban board on Trello with tasks.

3.7 Testing

Functions on the frontend were unit tested to ensure their correctness in producing the expected results. The JavaScript testing framework `Jest` was used for unit testing. JavaScript files containing tests were named in the form of `Name.test.js` and contain unit tests for a JavaScript file `Name.js`. Tests are executed in the frontend directory using the command: `$ npm test`

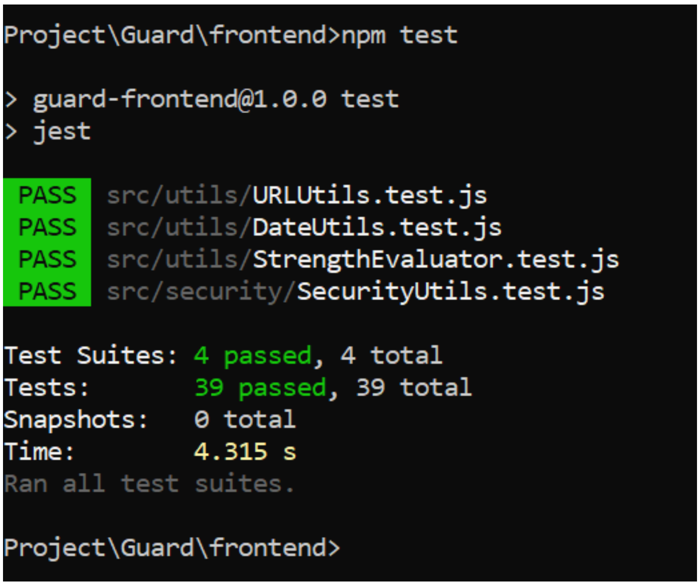

Figure 10: Successful execution of frontend tests.

The REST tool Postman (https://www.postman.com/) was used for API testing the backend. A Postman collection was created and exported to `Guard.postman_collection.json` in the `/tests` directory, which contains a Postman folder named `Test endpoints`. This

Postman folder contains other folders with requests for testing the REST API endpoints the backend exposes. Running the `Test endpoints` folder will test all endpoints. Below is a snippet of a run with all tests successfully fulfilled.

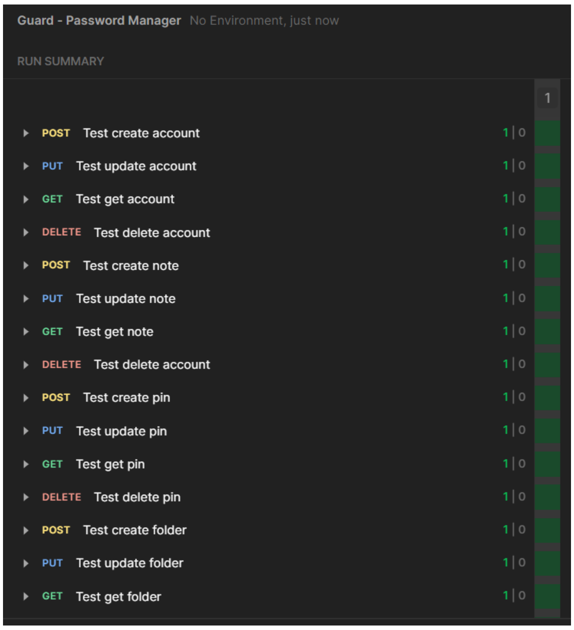

Figure 11: Successful API testing of backend endpoints.

Following the execution of these tests, the database was checked using MongoDB Compass (https://www.mongodb.com/products/tools/compass), which is a tool for analysing and querying MongoDB data. This tool was used to manually check the database to test the integration of the backend and MongoDB database by validating the database has been created and the data has been created successfully, as can be seen in the below figures.

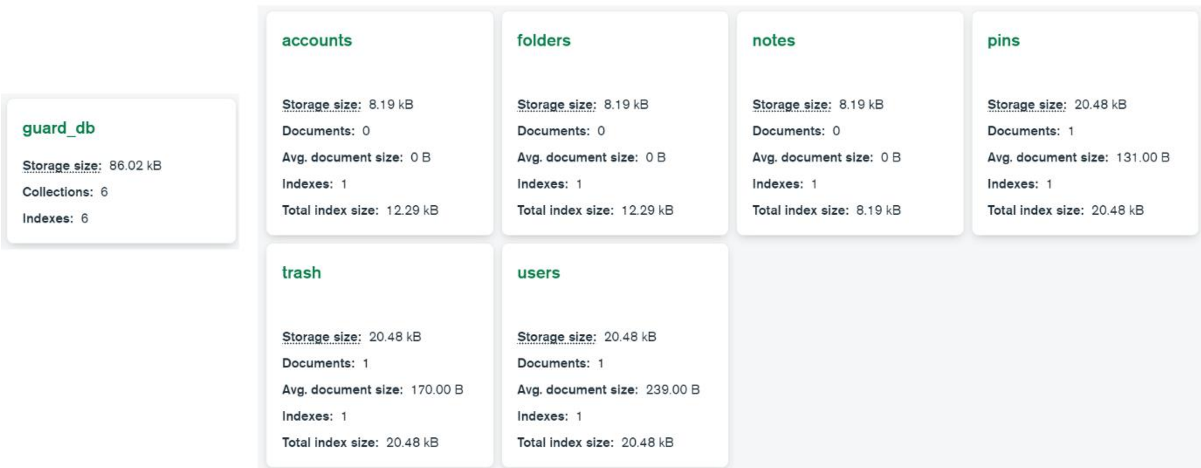

Figure 12: View of application database in MongoDB with data created successfully.

Integration testing of frontend-backend communication was manually performed to ensure the frontend can successfully communicate with the REST API and receive data payloads.

The entire application was end-to-end tested at the end of development, to ensure the application's features and components function as expected for end users and that data flow and communication worked as expected. The `Create mock data` folder in the Postman collection creates test data intended to be viewed on the frontend for user testing of the application.

4 Limitations

To prevent persistently storing the Encryption Key in the browser, the user needs to log into the application using their username and password every time the webpage is visited or reloaded. While this increases security, it negatively affects user experience. This might be able to be mitigated using a slightly different approach employing refresh tokens, which will be explained in the Future Work section.

The frontend uses the `CryptoJS` library to perform data encryption and decryption, which are performed synchronously and hence block DOM load. This can slow the webpage if there are many Vault items that need to be decrypted, or if the iterations of the PBKDF2-SHA-256 algorithm are too high. This could be resolved either by using a different library with asynchronous cryptographic operations, or possibly by using Web Workers to perform the operations in a separate thread to prevent blocking the main thread, so the user experience is not negatively impacted.

The application uses the HTTP protocol to load the frontend page and communicate with the backend, which reveals the data during communication since it is not encrypted, and hence is not secure for use in real production environments. A real production environment would need a Certificate Authority to generate an SSL certificate to employ HTTPS instead of HTTP, to digitally sign requests and encrypt transmitted data using TLS.

Since the frontend is headless, the application does not fully work if the backend is not running. Once a user is logged in, they can use the application even if the backend is down, but changes will not be saved since updates would not be received by the backend.

5 Results and Analysis

The result of this project has been a password manager application named Guard, composed of a frontend service and a backend service, employing a MongoDB database. The backend is a REST API, and the frontend is a website allowing users to store and retrieve passwords and secure user information. The application implements a password strength evaluator using the `zxcvbn` library to consider the latest research into password strength evaluators, as opposed to traditional evaluators which research has shown are insufficient [11, 13, 15]. The application has password generation functionality, which generates phonetic passwords to increase their pronounceability and memorability, and educates users on how to create strong passwords based on the latest research [14]. The password manager application provides a user-centric and security-focused solution for managing user passwords and sensitive information, which is secure and follows the latest security recommendations, and educates users on password security where possible. The project contributes to the field by offering users a solution that protects passwords while making their management as simple as possible, and password generation tools while educating users on password security.

6 Evaluation

6.1 Evaluation Framework and Outcomes

The application was intended to meet certain objectives determined prior to development. This section will discuss each objective and how it was achieved.

It should be easy to use

The application should be user-centric and easy for users to adopt and use. The user interface and user experience (UI/UX) should be intuitive, simple, and straightforward. Unintuitive design or poor UI/UX can result in users not using the application or not using every security feature for their benefit.

This was achieved through straightforward page design, with features being clear to view and use. Tooltips were used to display information of what each button does. Different data types, like accounts, pins, and notes, can be categorised and organised using folders. The user journey is simplified as the user only needs to know their username and master password.

It should support storing passwords and secure personal information

The application should support multiple types of secure personal information, including passwords, pin numbers, and secure notes.

Secure notes can store other types of plain text information, such as personal information, Wi-Fi codes, credit card details, and bank account information. These passwords and secure personal information should be securely transmitted and stored.

It should have a strong configurable password generator

The ability to generate passwords is one of the most important features of a password management application. Use of a password manager is irrelevant if the user employs weak passwords. A strong password generator should be implemented, allowing the user to configure the length and characters of the generated passwords.

This was achieved through the implementation of a password generator algorithm which generates passwords with configurable length, digits, special characters, and capital letters. To increase the pronunciation and possible memorability of generated passwords, the algorithm produces passwords with a phonetic system, rather than randomly-generated characters.

It should use the best encryption algorithms and techniques

The application should use the best encryption algorithms and security techniques as recommended by security experts and corroborated through active use in the industry. Market research must be done into the encryption algorithms and techniques used in the industry to identify the best standards and practices, while also considering the latest scientific research and suggestions.

The application implements the AES-256-CBC algorithm for encryption and decryption, with many iterations of PBKDF2 for added cryptographic complexity, and the SHA-256 algorithm for hashing. These are very secure algorithms that are widely used in real world applications, with AES even being a U.S. federal government standard. These algorithms are commonly used in the industry as found in the market research into other password managers.

It should ensure data security

Since the application handles such sensitive user information, data security is extremely important. Data must be stored and transmitted securely to prevent it from being stolen. The application must have strict security to prevent attackers from stealing user data.

The application only ever encrypts and decrypts user data on the frontend, and implements many layers of security, as discussed in the Security section earlier, to ensure user data is not compromised, intercepted, stolen, or decrypted by attackers.

It should encourage strong passwords

The application should visually and verbally display to the user the strength or weakness of their passwords, to psychologically encourage users to employ stronger passwords. Use of a password strength evaluator is particularly important for this case.

The application uses the `zxcvbn` library developed by Dropbox to evaluate password strength. The application processes password data using `zxcvbn` and categorises password strength as either Weak, Moderate, or Strong. The password strength is visually represented under password input fields as horizontal bars whose progress and colour indicates the strength of the password. The strength classification is also written down in text when the input field is focused.

It should educate users on what makes passwords strong

The password strength evaluator in the application should not only display the strength of the password, but also provide information of how to improve their current passwords.

When users insert passwords into input fields, the application uses the `zxcvbn` library to display to the user information related to their password, including a prediction of how long it would take for that password to be cracked and educates the user on possible suggestions for how to improve and strengthen their password. Additionally, the Health Monitor page displays information of what makes passwords strong and suggestions of using the password generator or long sentences and passphrases to create passwords.

It should provide an overview of user password health

Users should have a high-level overview of their password health and be alerted when passwords are weak or reused across accounts.

The application has a Health Monitor page which displays to the user the overall combined strength of their passwords, while also showing the user which of their passwords are weak or have been reused.

It should be loosely-coupled

The application should be architecturally loosely-coupled to allow for greater organisation of the different aspects of the overall application and to allow easy vertical scaling to meet high workloads in commercial or business use cases. The loosely-coupled architecture allows the possibility of developing different headless frontends as part of future work to expand the application. The separated backend also allows the application to be used across different devices since the backend and database would be running on servers on the network and not on the user device.

The application is composed of a React frontend, a Node backend, and a MongoDB database. The frontend communicates with the backend through a REST API over HTTP calls, and the backend communicates with the database using the `Mongoose` library.

6.2 Critical Evaluation

The development of the application progressed and concluded as planned, following the Kanban methodology, and correctly utilising the project Trello board, keeping tasks regularly updated under the correct column for clear visibility of the workload. Timelines were correctly managed and followed, and all major features were correctly implemented. A few minor tasks were not implemented because they were discarded during development, either due to technical limitations such as in the case of asynchronous decryption, or due to being rejected during testing such as in the case where persistent login was discarded for security purposes, or due to being preferentially replaced with higher priority tasks.

The application correctly stores passwords and other types of secure user information. The Vault data is encrypted with military-grade encryption algorithms and employs expert-recommended security practices. The intuitive design and labelled tooltips provide the user with a positive experience, and this coupled with a simple user registration and login process allow a new user to quickly make use of the application and adapt to the modern, clear, and simple interface. The action buttons in the list in the Passwords page allowing user passwords to be quickly copied to the clipboard while simply hovering over the item in the list is a helpful feature for users to quickly copy and paste credentials into services.

Due to the sensitive user information the application manages, user experience and application security are very important aspects that contrast heavily. The application achieves a positive and manageable trade-off between user experience and application security, with the major user impact being that users need to log in upon every page visit and reload.

While the user interface is simple and the user experience is positive, it would have been beneficial to perform user testing with several potential users to receive external feedback on how the UI/UX could be improved and to detect any unnoticed issues with the current design.

It is not possible for a user to modify their login details or Master Password. Adding user account and profile settings to the application would be beneficial to users, as it would provide them with the ability to change emails or update their Master Password. Since the Master Password is used in the encryption and decryption process of Vault data, due care must be paid during development to ensure when users update their Master Password, all Vault data is encrypted from scratch using the newly updated Master Password.

The interface was designed by default as a dark theme, but lacks any user customisation options for those who prefer light themes. Adding layout and theme customisation options can provide users with more personalisation choices over the layout and hence improve UI/UX.

The application prioritises security at all levels and takes many measures as described in earlier sections to ensure security is always maintained and sensitive user data is never compromised, leaked, or stolen. The field of application and data security is always adapting and developing, and this application would need to keep up with the latest research and findings to identify and mitigate emerging security threats to ensure long-term data and application security is maintained.

7 Conclusion and Future Work

7.1 Conclusion

This report has demonstrated the development of a password manager application named Guard, which has fulfilled the objectives and aims as described earlier. The developed application contributes to the field by providing users with a secure way of generating, storing, evaluating, and managing their passwords and other sensitive information, while educating users on password security where possible.

With respect to the latest research, traditional password evaluation metrics were not utilised to not promote their use. Instead, password evaluation metrics were based on the `zxcvbn` library, which is trained on real-world password leaks to ensure evaluation accuracy. The user registration process does not employ these traditional password metrics as password creation policies, and instead allows whitespaces to advocate for the transition to passphrases.

The latest research shows the strongest types of passwords are passphrases, and as such, the application encourages users to make use of passphrases instead of simple passwords, as these are longer and more secure while also being more pronounceable and memorable. The password generation algorithm generates passwords phonetically using syllables and whitespaces to generate long sounds to be more pronounceable.

More effort is needed to educate users on password security, because at the end of the day, a person is only as secure as their passwords, and password managers can provide users with the tools to take control of their online security.

7.2 Future Work

With regards to the frontend application, there are a number of helpful features that can be implemented in the future. For example, allowing importing passwords from commonly used browsers like Google Chrome. This would make it very easy for users to export their passwords from their browsers, in commonly used formats like JSON, and import them into the application. Further improvements can be done to the user experience, such as by developing a new frontend application as a Chrome extension to allow webpage integration for auto-filling passwords in input fields on webpages.

There are potential improvements that can be made to the user experience. Instead of asking users to insert their credentials on every visit or reload of the webpage, a potential alternate is to return two cookies during the login and registration phases: an Access Token and a Refresh Token. The Access Token can be stored in-memory, which will be cleared when the page is reloaded or closed. The Refresh Token can be stored as an HttpOnly cookie, so that if the page is visited while the Refresh Token exists in the browser, the Refresh Token can be sent to the backend to automatically respond with an Access Token, as if the user had logged in manually. While this improves user experience, it could improve security risks, such as Cross-Site Request Forgery.

With regards to research, further research into password security and the balance between password security and user experience would be beneficial. The fields of cryptography, password cracking, and data security are always developing and advancing, so the application should be kept updated on the latest security research to mitigate emerging security threats.

Glossary

AES – Advanced Encryption Standard is a military-grade encryption specification using a symmetric key algorithm that can be used in both encrypting and decrypting data [16].

PBKDF2 – Password-Based Key Derivation Function 2 is a cryptographic key derivation function that hashes a password to prevent brute-force attacks, dictionary attacks, and rainbow table attacks by making it very computationally and financially expensive to crack.

Argon2d – A key derivation function that prioritises protecting against GPU cracking attacks.

Scrypt – A key derivation function that uses a lot of memory to prevent several types of attacks [17].

Content Security Policy – A security standard which adds an extra layer of security to prevent cross-site scripting, data injection, and other types of attacks [18].

Cross-Origin Resource Sharing – A mechanism to allow a browser to load resources from defined origins other than that of the server [19].

Cross-Site Scripting – A type of attack where an attacker injects malicious executable scripts into the code of the website or database [20].

Appendix A: Software Used

Software used to develop this project:

- Node.js (https://nodejs.org/) – dependency manager and application launcher
- MongoDB (https://www.mongodb.com/) – database software
- Postman (https://www.postman.com/) – REST API testing and mock data creation
- IntelliJ IDEA (https://www.jetbrains.com/idea/) – development environment

Appendix B: Application Set Up

To set up the application and populate the database with example data:

1. After cloning the project to a local device, use a command prompt to enter the backend, install dependencies, and start the backend:

```
$ cd Guard/backend
$ npm install
$ npm start
```

   This should start the backend service on port `4000`, and connect to the MongoDB database:

```
> guard-backend@1.0.0 start
> nodemon server.js

[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server is up
Connected to database
```

   Figure 13: Successful start of backend service.

2. In another command prompt, enter the frontend, install dependencies, and start the frontend service:

```
$ cd Guard/frontend
$ npm install
$ npm start
```

This should start the frontend service on port 8080:

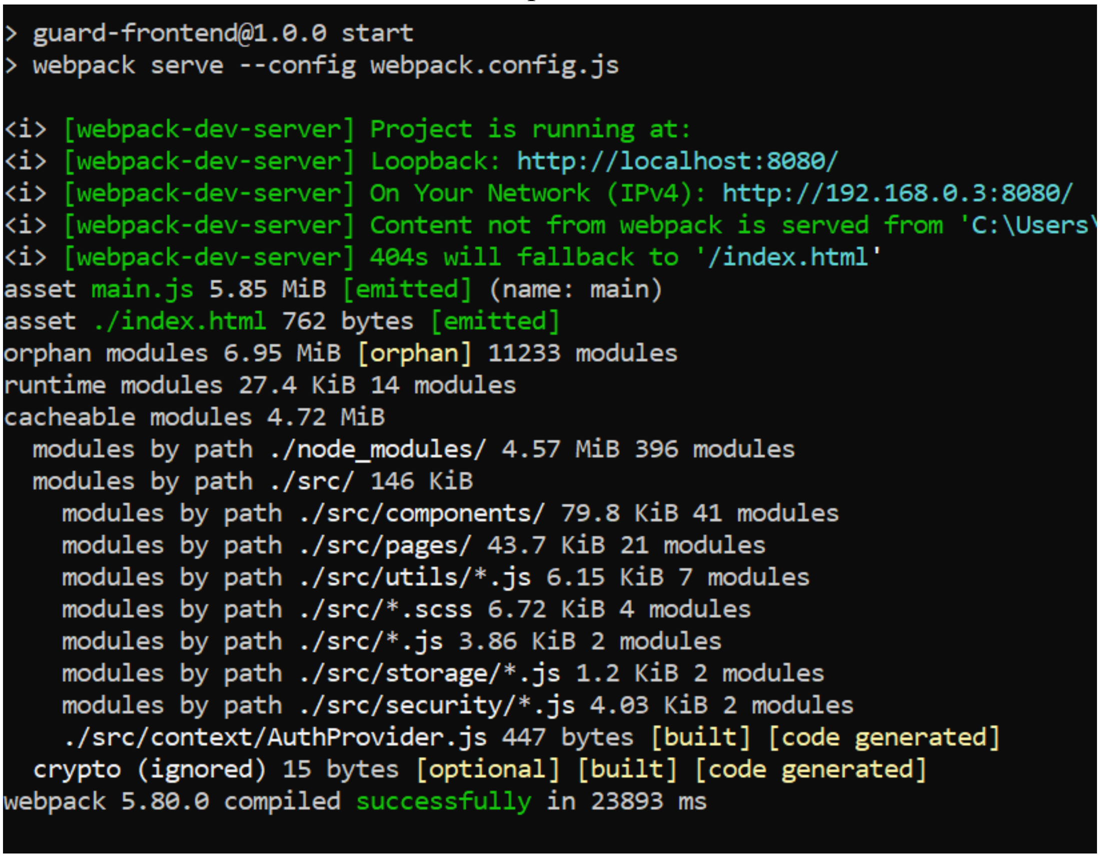

Figure 14: Successful start of frontend service.

3. Import test data into MongoDB:
   a. Import the `Guard.postman_collection.json` file from the `/tests` directory into Postman.

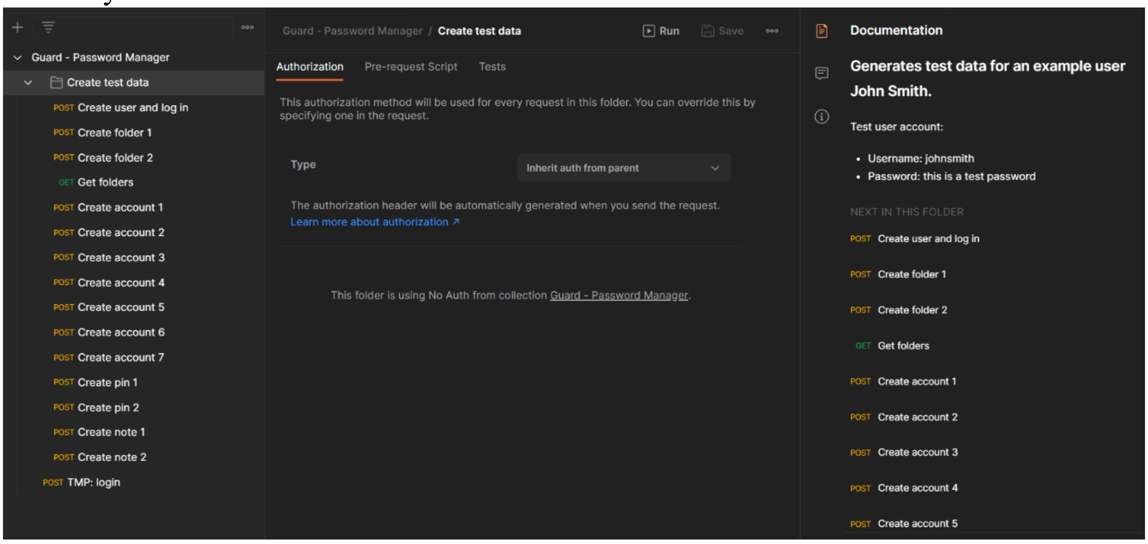

Figure 15: Project collection imported into Postman.

   b. Run the `Create mock data` folder:

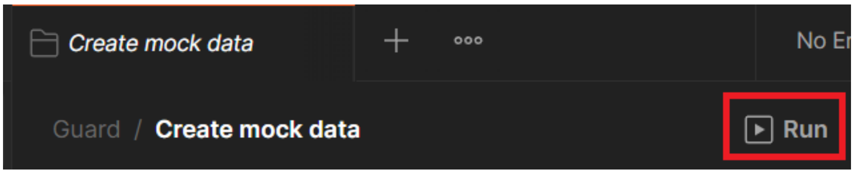

Figure 16: Run folder to create mock data.

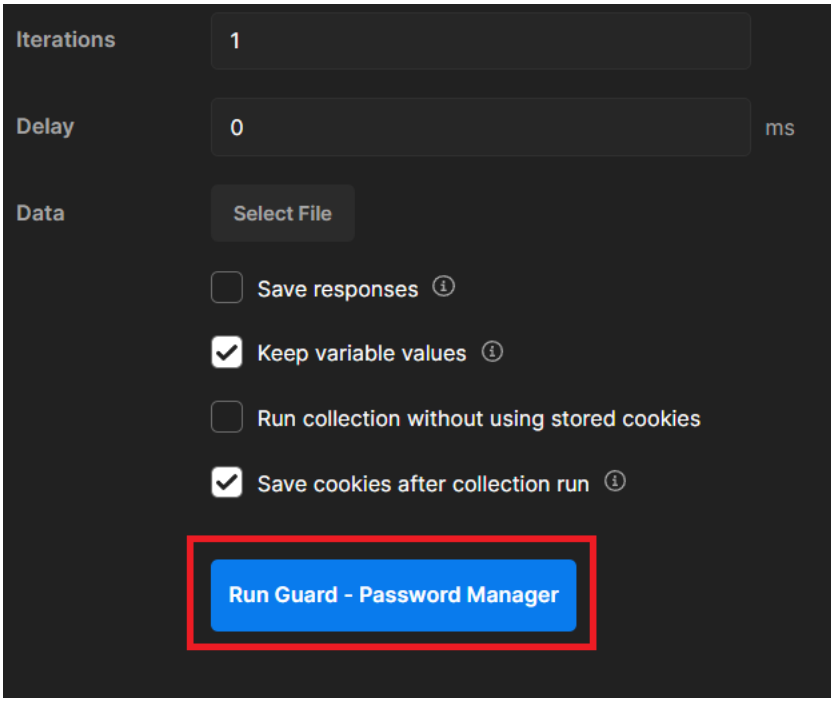

Figure 17: Run folder to create mock data.

This should successfully execute all requests, with all tests passing:

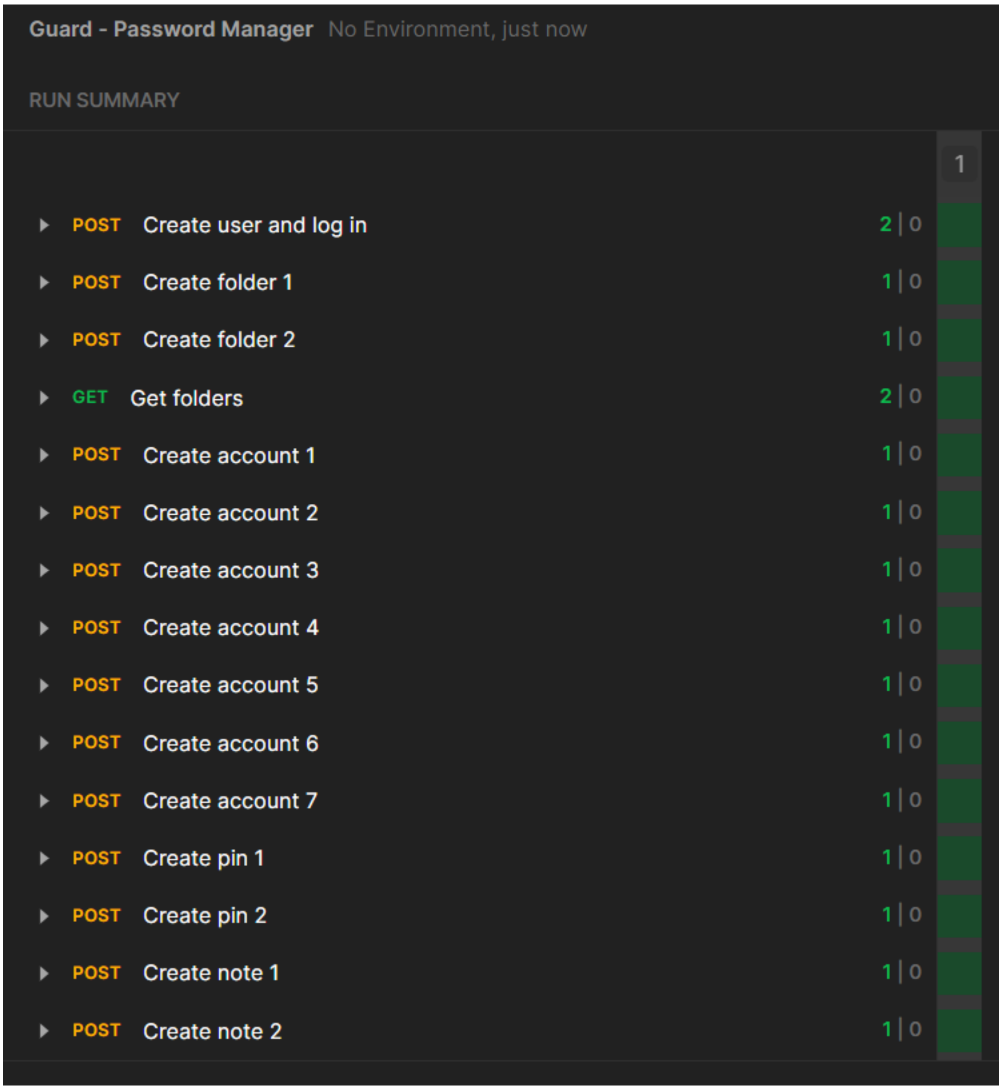

Figure 18: Data is imported successfully with all tests passed.

This should create the collections and data in MongoDB, which can be viewed using MongoDB Compass:

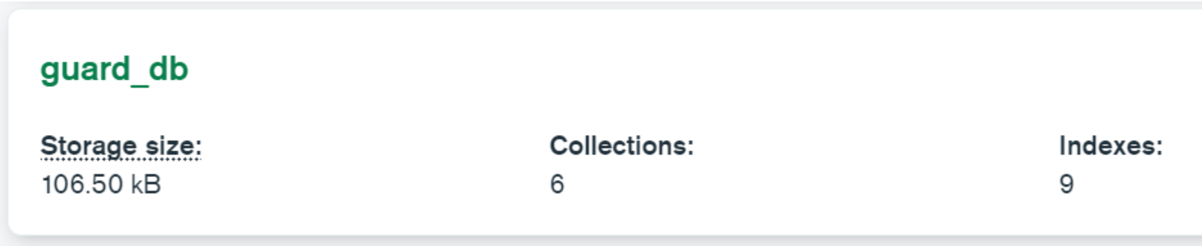

Figure 19: Successful creation of database in MongoDB.

4. Visit the frontend on http://localhost:8080/login and log in using the test credentials:
   Username: johnsmith
   Password: this is a test password

Appendix C: REST API Documentation

Auth Management

POST /api/auth/user

Creates a new user

Body

object

email string The user's email of length 4-30
username string The user's alphanumeric username of length 6-30
authHash string The user's authentication hash

Responses

Status Code	Description
201	User successfully created. Access Token sent as cookie. object accessToken string JSON Web Token string
400	Incorrect data supplied. object err string Error message

POST /api/auth/login

Logs into existing user's account

Body

object

email string The user's email of length 4-30
authHash string The user's authentication hash

Responses

Status Code	Description
200	User credentials match and user has successfully logged in. Access Token sent as cookie. object accessToken string JSON Web Token string
401	Authentication Failed. object err string Error message
400	Incorrect data supplied. object err string Error message

POST /api/auth/logout

Logs out of user’s account

Responses

Status Code	Description
200	User logged out. Access Token cookie cleared.

Account Management

POST /api/account

Creates a password account

Cookies

x-auth-token string A valid access token

Body

object
- site string The URL of the website related to the account
- title string The user title of the account
- identity string optional The identity of the account, usually username or email
- pw string The account password
- folders array optional The user folders the account belongs to

Responses

Status Code	Description
201	Account successfully created. object site string title string identity string pw string folders array user string date string lastAccess string
401	Not authorised.
400	A user error has occurred. object err string Error message

GET	/api/account/:accountId

Retrieves a password account

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Account returned. object site string title string identity string pw string folders array user string date string lastAccess string
401	Not authorised.
400	A user error has occurred. object err string Error message

PUT	/api/account/:accountId

Updates a password account

Cookies

x-auth-token string A valid access token

Body

object
- site string optional The URL of the website related to the account
- title string optional The user title of the account
- identity string optional The identity of the account, usually username or email
- pw string optional The account password
- folders array optional The user folders the account belongs to

Responses

Status Code	Description
204	Account updated.
401	Not authorised.
400	A user error has occurred. object err string Error message

DELETE /api/account/:accountId

Deletes a password account

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Account deleted.
401	Not authorised.
400	A user error has occurred. object err string Error message

Pin Management

POST /api/pin

Creates a user pin

Cookies

x-auth-token string A valid access token

Body

object

title string The title of the pin

pin string The user pin

folders array optional The user folders the pin belongs to

Responses

Status Code	Description
201	Pin successfully created. object title string pin string folders array user string lastAccess string
401	Not authorised.
400	A user error has occurred. object err string Error message

GET	/api/pin/:pinId

Retrieves a user pin

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Account returned. object title string pin string folders array user string lastAccess string
401	Not authorised.
400	A user error has occurred. object err string Error message

PUT	/api/pin/:pinId

Updates a user pin

Cookies

x-auth-token string A valid access token

Body

object

- title string The title of the pin
- pin string The user pin
- folders array optional The user folders the pin belongs to

Responses

Status Code	Description
204	Pin updated.
401	Not authorised.
400	A user error has occurred. object err string Error message

DELETE /api/pin/:pinId

Deletes a user pin

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Pin deleted.
401	Not authorised.
400	A user error has occurred. Object err string Error message

Note Management

POST /api/note

Creates a user note

Cookies

x-auth-token string A valid access token

Body

object

title string The title of the note

note string The user note

folders array optional The user folders the note belongs to

Responses

Status Code	Description
201	Note successfully created. Object title string note string folders array user string lastAccess string
401	Not authorised.
400	A user error has occurred. Object err string Error message

GET	/api/note/:noteId

Retrieves a user note

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Note returned. object title string note string folders array user string lastAccess string
401	Not authorised.
400	A user error has occurred. object err string Error message

PUT	/api/note/:noteId

Updates a user note

Cookies

x-auth-token string A valid access token

Body

object

- title string The title of the note
- note string The user note
- folders array optional The user folders the note belongs to

Responses

Status Code	Description
204	Note updated.
401	Not authorised.
400	A user error has occurred. object err string Error message

DELETE /api/note/:noteId

Deletes a user note

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Note deleted.
401	Not authorised.
400	A user error has occurred. object err string Error message

Folder Management

POST /api/folder

Creates a user folder

Cookies

x-auth-token string A valid access token

Body

object

name string The name of the folder

Responses

Status Code	Description
201	Folder successfully created. object name string user string
401	Not authorised.
400	A user error has occurred. object err string Error message

GET	/api/folder/:folderId

Retrieves a user folder

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Folder returned. object name string user string
401	Not authorised.
400	A user error has occurred. object err string Error message

PUT	/api/folder/:folderId

Updates a user folder

Cookies

x-auth-token string A valid access token

Body

object

name string The name of the folder

Responses

Status Code	Description
204	Folder updated.
401	Not authorised.
400	A user error has occurred. object err string Error message

DELETE /api/folder/:folderId
Deletes a user folder

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Folder deleted.
401	Not authorised.
400	A user error has occurred. object err string Error message

Trash Management

GET /api/trash/:id
Retrieves an item from trash

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Trashed item returned. object
401	Not authorised.
400	A user error has occurred. object err string Error message

PUT /api/trash/:id
Updates an item in trash

Cookies

x-auth-token string A valid access token

Body

object

Responses

Status Code	Description
204	Trashed item updated.
401	Not authorised.
400	A user error has occurred. object

	err string Error message

DELETE /api/trash/:id

Permanently deletes an item in trash

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Trashed item permanently deleted.
401	Not authorised.
400	A user error has occurred. object err string Error message

GET /api/trash/:id/restore

Restores an item from trash

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	Trashed item restored.
401	Not authorised.
400	A user error has occurred. object err string Error message

Vault Management

GET /api/vault

Retrieves a user's secure Vault

Cookies

x-auth-token string A valid access token

Responses

Status Code	Description
200	User Vault returned. object accounts array pins array notes array folders array trash array

🟡 401	Not authorised.

Bibliography

References:

1. Ponemon Institute, "The 2020 State of Password and Authentication Security Behaviors Report," https://www.nass.org/sites/default/files/2020-04/Yubico%20Report%20Ponemon%202020%20State%20of%20Password%20and%20Authentication%20Security%20Behaviors.pdf [Accessed May 2023]
2. Verizon, "2022 Data Breach Investigations Report," https://www.verizon.com/business/en-gb/resources/2022-data-breach-investigations-report-dbir.pdf [Accessed May 2023]
3. National Cyber Security Centre, "Password managers: using browsers and apps to safely store your passwords," https://www.ncsc.gov.uk/collection/top-tips-for-staying-secure-online/password-managers [Accessed May 2023]
4. PasswordManager, "65% of people don't trust password managers despite 60% experiencing a data breach," https://www.passwordmanager.com/password-manager-trust-survey/ [Accessed May 2023]
5. LastPass, "Notice of Recent Security Incident," https://blog.lastpass.com/2022/12/notice-of-recent-security-incident/ [Accessed May 2023]
6. Fagan, M., Albayram, Y., Khan, M.M.H. et al, "An investigation into users' considerations towards using password managers," https://hcis-journal.springeropen.com/articles/10.1186/s13673-017-0093-6 [Accessed Jul 2023]
7. Dashlane, "Dashlane's Security Principles & Architecture," https://www.dashlane.com/download/whitepaper-en.pdf [Accessed Jul 2023]
8. Dashlane Support, "FAQ about security at Dashlane," https://support.dashlane.com/hc/en-us/articles/360012686840-FAQ-about-security-at-Dashlane [Accessed Jul 2023]
9. 1Password, "1Password Security Design," https://1passwordstatic.com/files/security/1password-white-paper.pdf [Accessed Jul 2023]
10. LastPass, "LastPass Technical Whitepaper," https://support.lastpass.com/s/document-item?language=en_US&bundleId=lastpass&topicId=LastPass/lastpass_technical_whitepaper.html&_LANG=enus [Accessed Jul 2023]
11. Shenoy, S., "Empirical Analysis on the Usability and Security of Passwords," https://scholarworks.calstate.edu/downloads/m326m2061 [Accessed Sep 2023]
12. NIST, "NIST Special Publication 800-63B: Digital Identity Guidelines," https://pages.nist.gov/800-63-3/sp800-63b.html [Accessed Sep 2023]
13. Carnavalet, X.D.C.D., and Mannan, M., "From Very Weak to Very Strong: Analyzing Password-Strength Meters," https://www.ndss-symposium.org/wp-content/uploads/2017/09/06_3_1.pdf [Accessed Sep 2023]
14. Stony Brook University, "Cyber Security: Passphrases and Passwords," https://it.stonybrook.edu/help/kb/cyber-security-passphrases [Accessed Sep 2023]

15. Wheeler, D.L., Dropbox Inc, “zxcvbn: Low-Budget Password Strength Estimation,” https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_wheeler.pdf [Accessed Sep 2023]
16. Bernstein, C., and Cobb, M., “Advanced Encryption Standard (AES),” https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard [Accessed Jul 2023]
17. Nakov, S., “Scrypt,” https://cryptobook.nakov.com/mac-and-key-derivation/scrypt [Accessed Jul 2023]
18. Mozilla, “Content Security Policy (CSP),” https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP [Accessed Sep 2023]
19. Amazon Web Services, “What is CORS?,” https://aws.amazon.com/what-is/cross-origin-resource-sharing [Accessed Sep 2023]
20. KirstenS et al, “Cross Site Scripting (XSS),” https://owasp.org/www-community/attacks/xss/ [Accessed Sep 2023]
21. React, “Introducing JSX: JSX Prevents Injection Attacks,” https://legacy.reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks [Accessed Sep 2023]
22. Ur, B., Kelley, P.G. et al, “How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation,” https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final209.pdf [Accessed Sep 2023]