Mimir: A Password Manager Project

By Edward William Bennett

Supervisor: Atta Badii

Abstract

A password manager is a piece of software used to store and manage online account details. Using one “master” account users can access every other online account they own, without the hassle of needing to remember each password. More than just convenience, password managers offer a strong method for improving the security of one’s online presence by increasing the feasibility of using more secure passwords. While experts recommend the use of a unique password for each online account, this is often ignored by users in favour of using one password for all their online activities. In addition, users tend to select shorter, less diverse passwords to ensure they can remember them; however, this results in weaker passwords which are more vulnerable to brute force and dictionary attacks. A password manager – if used correctly – promises to reduce these problems, by providing a way of managing many unique, long, complex passwords. This report details the design and creation of such a password manager as a web application, from initial research of literature and existing products, through the approach and practical implementation. The resulting password manager was evaluated using user feedback before a brief discussion is included as to where this project could be taken in the future. The resulting manager was determined to be mostly a success, providing a method of generating and storing passwords users would struggle to create and remember on their own.

Acknowledgements

I would like to especially thank Andy and Gio for their advice throughout this project, absolute legends.

I would also like to thank my housemates, my family, my friends from home, and my cats – because why not.

Table of Contents

Introduction

A growing problem with online security is the tendency for users to choose passwords which are insecure due to length, lack of character variety, popularity, or reuse. This can open users up to having accounts hijacked, leading to stolen personal information, impersonation, or the simple inconvenience of having to make new accounts and change passwords.

At its most fundamental, a password manager is a database that allows users to store usernames and passwords for their online accounts. Users would log in to their "master" account using a username or email address and one strong "master" password. This reduces the cognitive burden of remembering multiple accounts to just one.

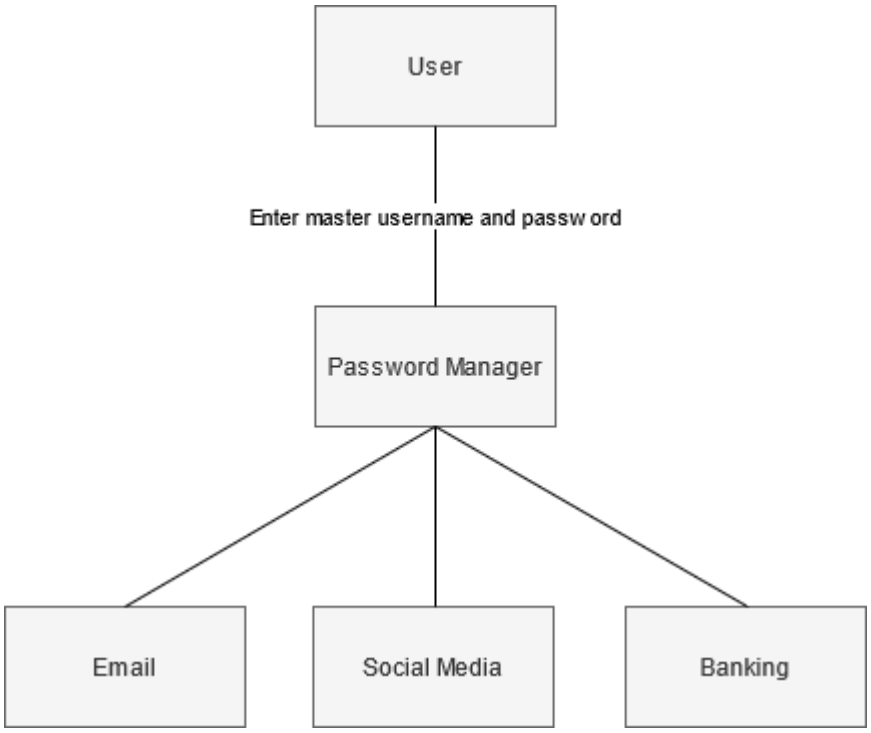


Figure 1 - A user only needs to remember their master username and password to access other credentials

As users can access all other account details from this master account using a unique password for each online profile becomes much more feasible. Each unique password can now be much stronger, with every password being dozens of characters long. A common tendency is for a user to reuse either the same password for every online account, or variations of the same password along a theme. A password manager often includes facilities for generating new passwords, used to ensure that users can instead create a

password that consists of random characters, rendering the password much more resistant to dictionary attacks or guesswork.

This project aims to create a simple but functional password manager available as a Web application, allowing a user to improve the security of their online presence using stronger and more unique passwords. In order for such a tool to be effective, it must be used. With this in mind, usability has been an additional focus, with the project also aiming to offer a service as simple and convenient as not using a password manager.

To achieve these aims the project must allow the user to create a master account and use it to store, retrieve, and manage their external account credentials. The project must also include a password generator tool to allow the user to create stronger passwords than they would be able to create independently. To fulfil the second aim of convenience, the project must be considered user friendly, with positive user feedback indicating ease of use.

This report is divided into several chapters. The first chapter is a problem articulation and literature review which will provide more context to the reason for password managers, and an examination of the literature around them. The second chapter, the methodology, will describe the process initially outlined to plan this project and the thought process behind planning the research and evaluation. The third chapter is a systemic review of existing password manager projects. The fourth chapter will describe the solution approach used for this project. The fifth chapter, the implementation, will guide the reader through the process of conceiving and constructing a password manager. The sixth chapter will show the results of a series of functionality tests, along with the responses to a user feedback questionnaire about the end product. In the discussion and analysis, the seventh chapter, this report will evaluate and speculate on the results of testing and user feedback. Finally, in the conclusion these results will be discussed, the project will be evaluated, and suggestions will be made as to where the project could move in the future.

Problem Articulation and Literature Review

Online security has grown increasingly important as more and more of our day-to-day lives takes place online. This importance is magnified further as more significant actions are performed online, such as digital contract signing and online banking.

Poor password habits such as failing to frequently change passwords and choosing low entropy, easy to remember passwords are prevalent among many users (Florencio & Herley, 2007). A 2005 study found that end users have a "rather dismal record of enacting the basic hygiene behaviors", failing to change passwords frequently or select strong passwords. Indeed, the study found that 62.5% of respondents declared they "never used numbers or punctuation marks" and 48.5% said they "had not changed their passwords in the last six months". While the study found that training, awareness, and knowledge of monitoring passwords on average resulted in better password hygiene, the study also found that these improvements "seemed to associate with a greater likelihood of writing down one's password" (Stanton et al. 2005).

An additional pervasive issue with password security is password reuse – the tendency for users to use the same password for multiple online accounts. In 2014 a study of leaked passwords across 11 web sites found an estimated 43-51% of users reused the same password across multiple web sites (Das et al. 2014). Furthermore, the study identified that many users use the same basic password across multiple web sites which was transformed using minor variations. While this could be considered an improvement, the researchers were able to create a password-guessing algorithm which was able to guess 30% of transformed passwords within 100 attempts – a vast increase in vulnerability compared to 14% found for a standard password-guessing algorithm without cross-site knowledge. This indicates that for many users a data breach of one service can lead to compromised accounts elsewhere.

Password managers can thus be considered a more secure alternative to the average user's password habits, allowing for longer passwords which are unique to each online service. Indeed, members of the United States Computer Emergency Readiness Team asserted that password managers are "one of the best ways to keep track of each unique password or passphrase", and that "Although moving to a password manager may take a little effort, in

the long run it is a safe and convenient method of keeping track of your passwords and guarding your online information" (Huth, Orlando and Pesante, 2012).

One concern with password managers is that if the master account is hacked, the user loses all their online accounts. Hashing is a one-way cryptographic function which is used to convert a password into a different format using a cryptographic salt. Rehashing the user's submitted password for comparison with the stored hash is still useful for authentication, but much less useful to a malicious actor as they cannot reverse the hashing to find the original password to use elsewhere. For an attacker to obtain the original password from a hash, they must hash password guesses until they find a match. Some password hash functions such as Bcrypt employ key-stretching methods to increase the amount of time taken for a guess to be calculated (Blocki, J. and Datta, A., 2016). One example of key-stretching is rehashing the password multiple times. One method of attacking a hash, lookup tables, can be thwarted using a second input called a salt (Provos, N. and Mazieres, D., 1999). The salt is random data which is added to the password before it is hashed. As an attacker does not know the value of the salt, this adds additional difficulty in trying to guess the password from a hash. Bcrypt also uses salting.

With regards to stored password records, these must be encrypted reversibly so they can be sent back to the user in a usable way. If the user were sent back a password hash the client would be unable to reverse this, and they would not be able to use their password to login. Passwords should none-the-less be stored in as secure a manner as possible, using a private key known only to the server to encrypt them. The Advanced Encryption Standard (AES) is one such encryption algorithm which was selected by the US National Institute of Standards and Technology as the Advanced Encryption Standard from 15 candidate algorithms (Atasu, Breveglieri and Macchetti, 2004).

Methodology

Research

To research features and design choices for this project the first step was to examine existing products and identify features of these products which were both successfully and poorly implemented. Five existing password managers were identified: Keeper, Bitwarden, Dashlane, LastPass, and SplashID Safe. For each of these products an account was created with the same five example password records in two different folders to demonstrate how the manager might look under normal use.

Each password manager has been evaluated based on the common components users will interact with most: the "vault" page where the user's password records are listed; the password records where an individual record is shown in greater detail; the password generator tool which allows the user to generate new passwords; the account settings page which can be used to update any account details and set personal preferences; and any miscellaneous features such as a security evaluator or two-factor authentication. Evaluating each of these components for existing products should give a strong foundation as to the features required of a password manager and could give an indication in how to construct them. The aesthetics and usability of each manager has also been reviewed throughout to obtain an idea as to how the interface of this project should be designed to maximise user friendliness and familiarity.

User Feedback

When evaluating usability, Whitney Quesenbery suggests in "Dimensions of Usability: Defining the Conversation, Driving the Process" the "5E" dimensions of usability: effective, efficient, engaging, error tolerant, and easy to learn. These dimensions are described below.

Effective describes the completeness and accuracy with which users achieve their goals. If a program is usually used successfully without errors, it is usually effective.

Efficient determines the speed with which work can be done. If a user feels that a task takes "too long" or "too many clicks" the user feels the software is not efficient.

Engaging describes how pleasant, satisfying, or interesting and interface is to use.

Error tolerant describes how well the product prevents errors, and helps the user recover from any errors that do occur.

Easy to learn describes how well the product supports initial orientation, and deeper learning. If a user feels an interface is immediately obvious to learn – or relearn - to use, the user feels the software is easy to learn.

It is further suggested that creating first-person statements that express a usability requirement for each dimension can aid in discussing usability (Quesenbery, 2003). With this in mind, a user feedback questionnaire based on statements which target each dimension using Likert scales has been designed. Likert scales are a tool "to measure 'attitude' in a scientifically accepted and validated manner" consisting of a set of statements, each of which participants are asked to rate their level of agreement with from strongly disagree to strongly agree (Joshi, Kale, Chandel and Pal, 2015). For this questionnaire, the statements used are the first-person statements that express a usability requirement to gather an attitude as to the usability of the project. The questionnaire is included below.

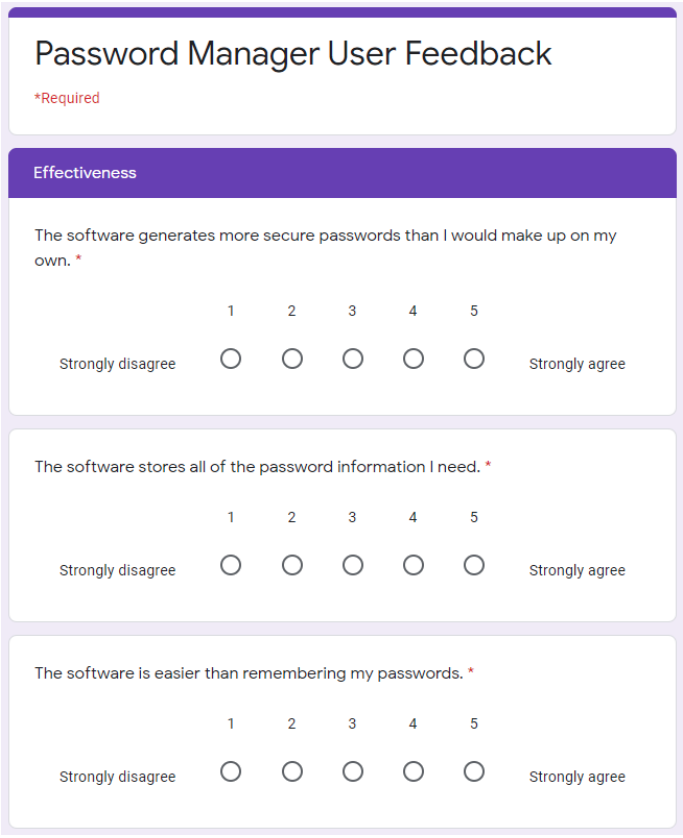


Figure 2.1 - User Feedback Form Effectiveness Section

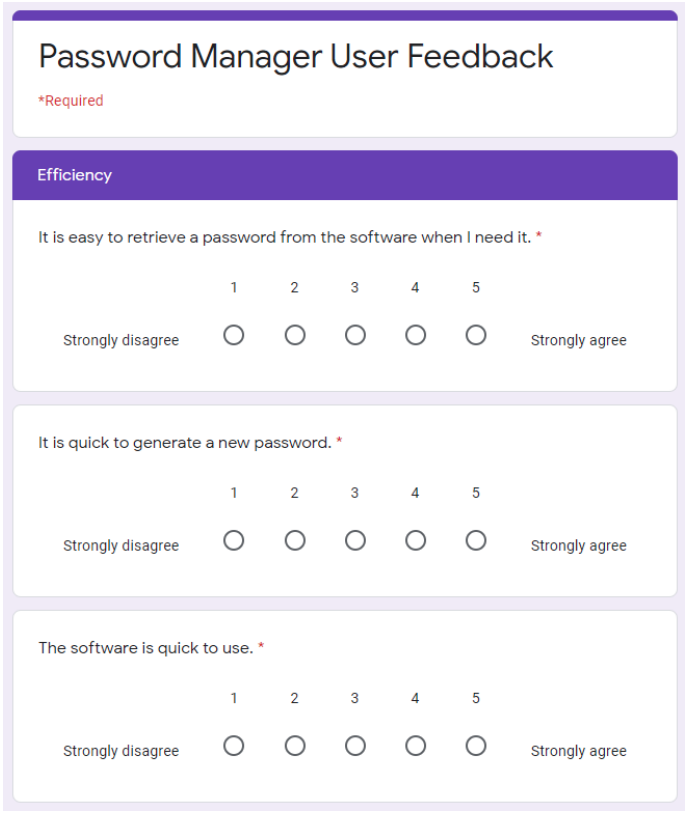


Figure 2.2 - User Feedback Form Efficiency Section

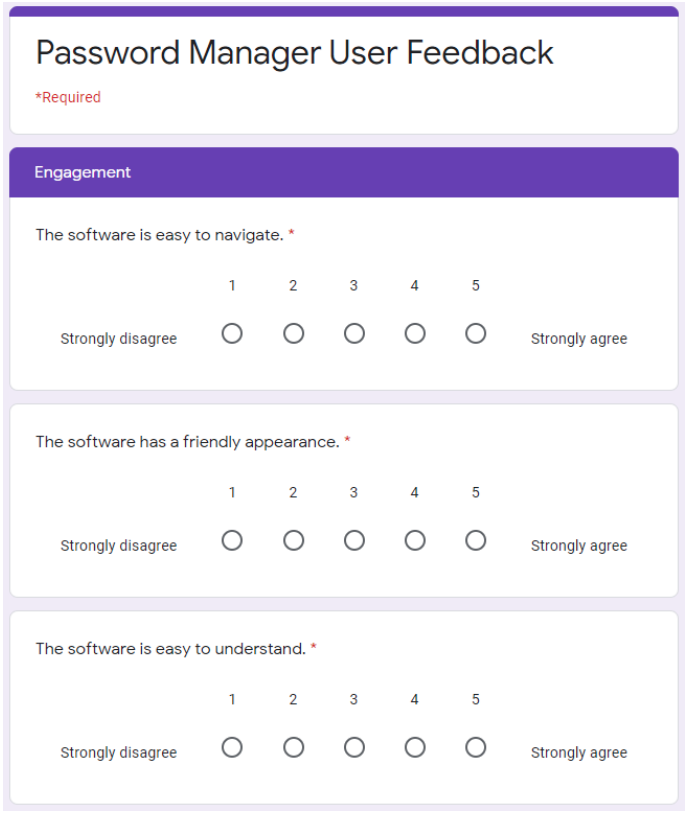


Figure 2.3 - User Feedback Form Engagement Section

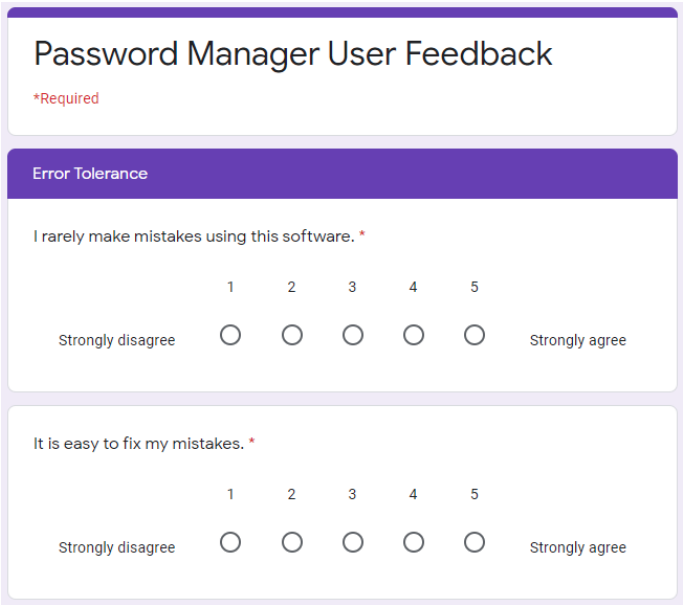


Figure 2.4 - User Feedback Form Error Tolerance Section

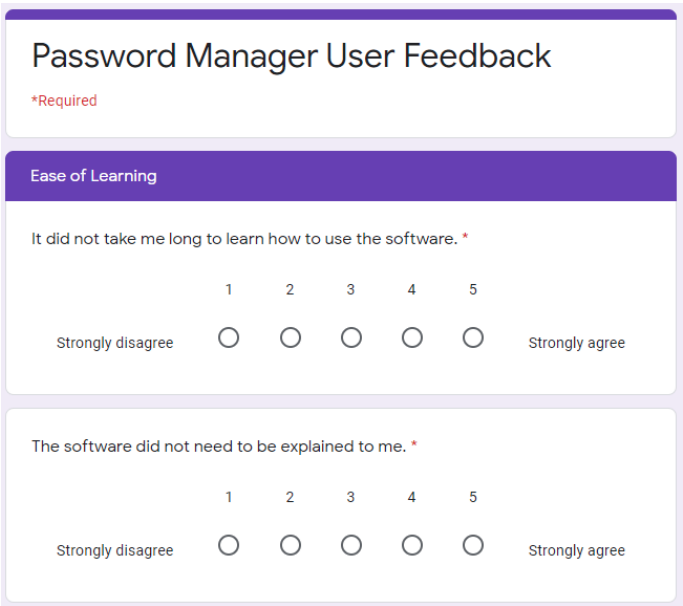
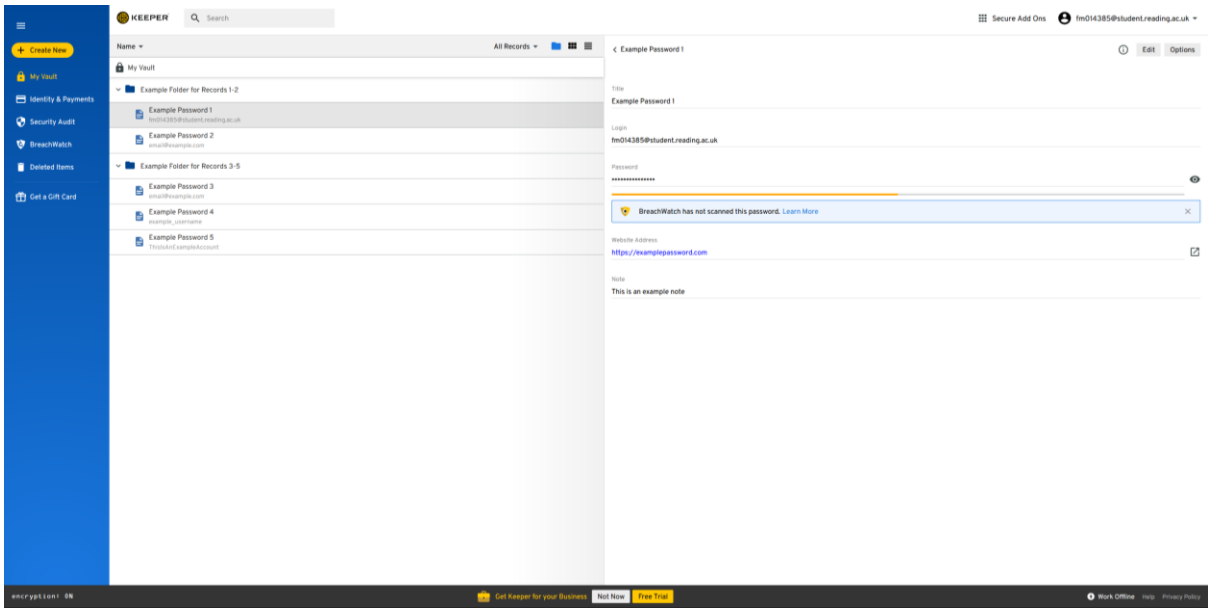


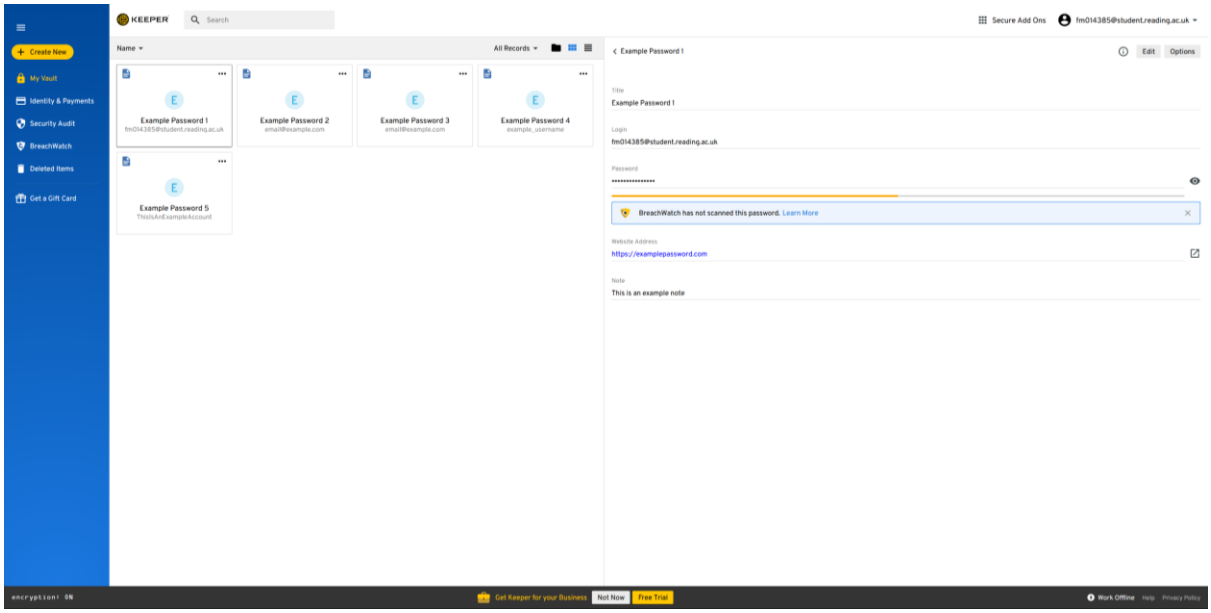
Figure 2.5 - User Feedback Form Ease of Learning Section

Systemic Review of Existing Password Managers

Keeper

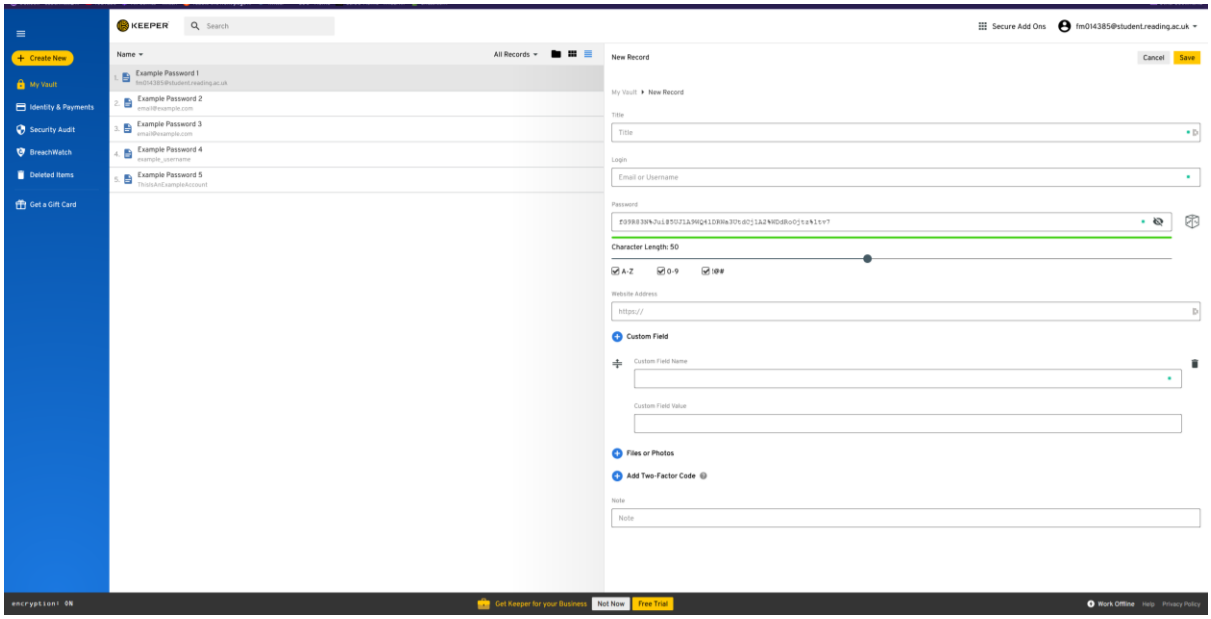
Keeper is a password manager with a clean and simple modern design with little wasted space. Keeper has a simple intuitive interface which clearly communicates to the user how to add a new password, view existing passwords, and organise their database. Most of the real estate of the page is dedicated to password records and clicking on a password opens a window to the right showing all the information about the record. The passwords in the main window are organised alphabetically, inside drop-down lists of folders if the passwords are assigned to a folder. Three different views for the records are also available, a list view of passwords and folders, a card view of passwords, and a list view of all passwords without folders. These three views are shown below with the first example password selected.





Records in Keeper are displayed simply, with each record possessing a title, a login value for a username or password, a field for the password itself, the website address, and a notes section. Each of these sections are optional, except for the title which must be filled in. In addition to these standard fields, Keeper allows custom fields, with the user able to give the field a name and a value, file or photo attachments, or a two factor authentication code. Once a record has been saved to the vault clicking on a field copies the field's value to the clipboard, allowing for more convenient copy and pasting of a username or password

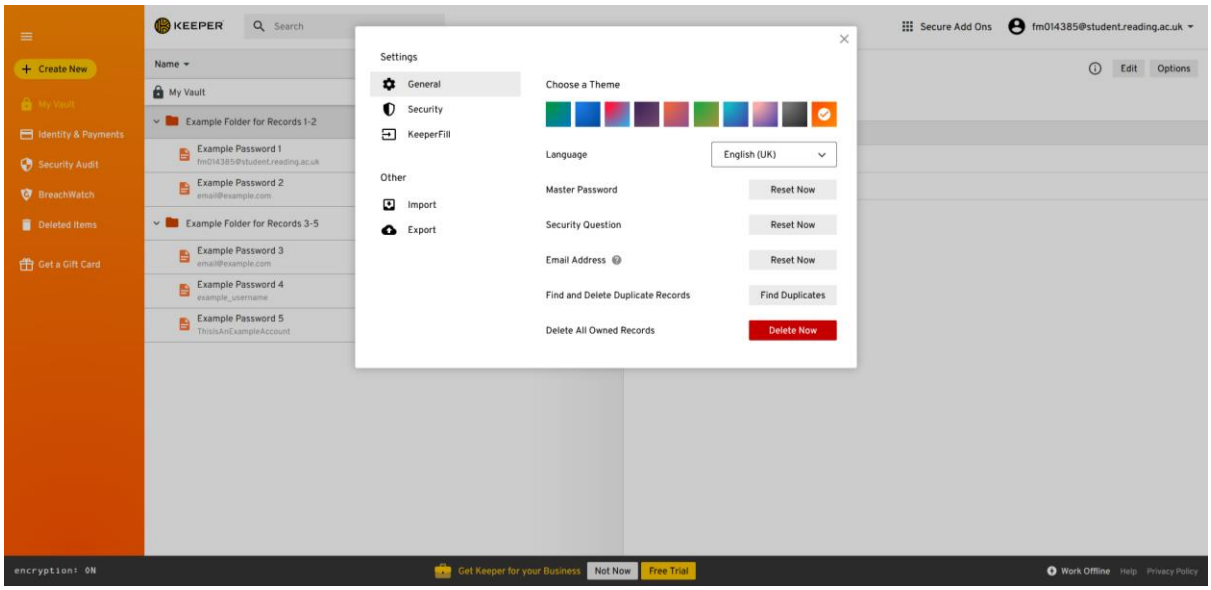
elsewhere, however this does mean altering a record can only be done after clicking a small edit button in the top right corner of the record. This could make editing a record frustrating for a user, and as the button is grey instead of highlighted a new user could struggle to find this feature. A second button labelled “Options” is next to the edit button, and this offers a shortcut to open the corresponding website for the record, a premium feature to roll-back the version of the record, sharing the record, adding the record to a list of favourites, assigning a colour to the record, duplicating the record, or deleting the record. Offering a colour coding system for records is a unique feature that could allow for quick identification of a password at a glance. The option to mark select passwords as favourites could also offer a convenient timesaving measure for a user, however placing the function to delete a record in such an obscure position could cause irritation for users as such a key feature is not immediately apparent. The window for creating a new record is shown below.



Keeper also includes a convenient password generator built into the new record window. The generation of a password is depicted by a dice icon, and once clicked a 20-character password is immediately generated and the generation settings expand if the user would like to tweak these settings. The settings for password generation include capital letters, numbers, and symbols, however lowercase letters are enabled by default and cannot be turned off. The length of the password can be customised between 8 and 100 characters in

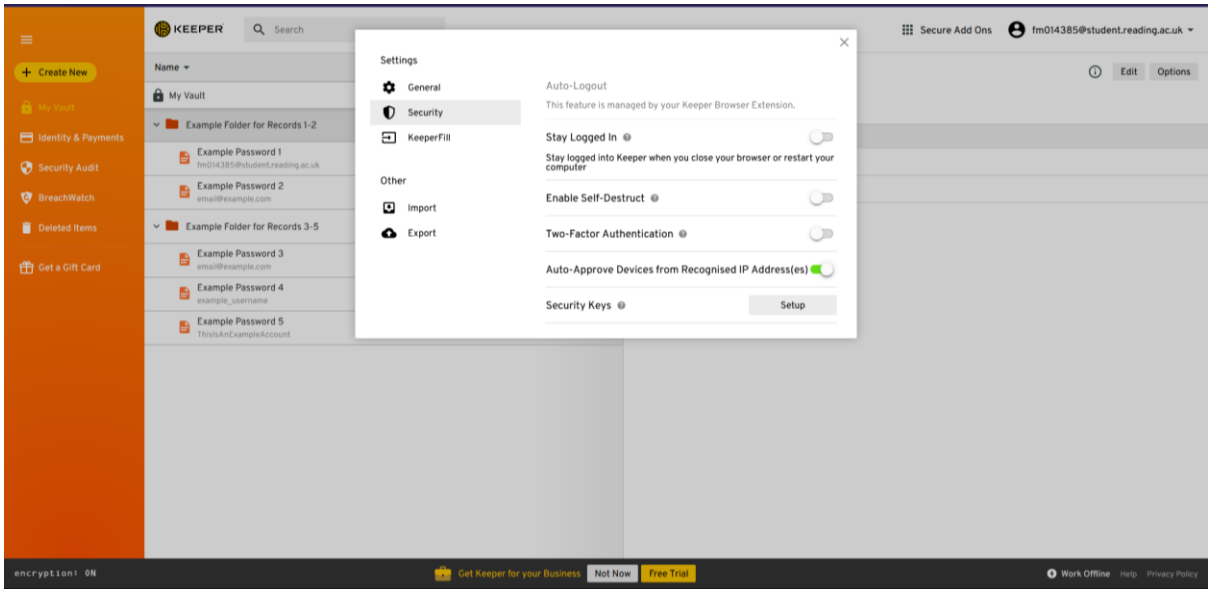
length. The above screenshot also shows the password generator with the settings expanded and a generated 50-character password with all characters enabled.

The settings and account page of Keeper is simple, with multiple tabs offering separation of concerns. The first tab shows general settings, and this allows a user to pick a theme. A feature for colour control could offer greater accessibility for users with conditions such as colour blindness or dyslexia, however the implementation here appears to mainly target aesthetics, as it only changes the colour of the sidebar and some icons. The general options page also includes options to reset the account's master password, security question, and email address. An organisational feature to automatically find and delete any duplicate records is included here and could be valuable for any users with large and poorly managed databases. Finally, a feature to delete all records is included at the bottom, highlighted in red to indicate that it is a dangerous option a user should not click out of curiosity. The general settings page is shown below.

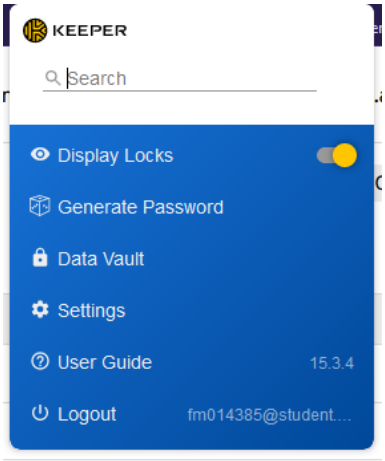


The second tab in the options menu is the security tab, and this section offers the user toggleable options to keep the user permanently logged in, a "self-destruct" option to enable automatic deletion of all records if five incorrect attempts to log in have been made in a row, an option to enable two-factor authentication for the account (both mobile app

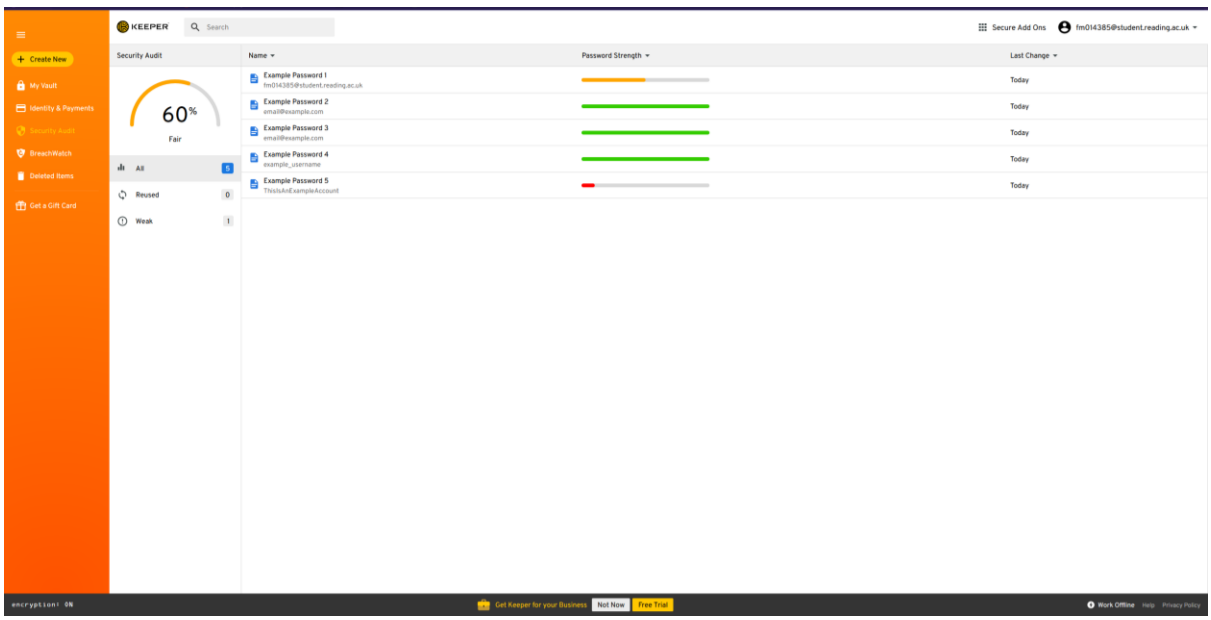
based and physical key based), and automatically approve devices from recognised IP addresses. This window of the settings menu is shown below.



The final sections of the settings menu offer links to download the Keeper browser extension and import or export records as a file. The Keeper browser extension allows for the quick searching of password records, generation of new passwords, and shortcuts into the main vault or the settings page. When logging into a website with the extension installed an icon appears next to the username and password fields with the Keeper logo. Upon clicking this icon, the extension offers to automatically fill the fields with stored records, or if no corresponding records are found the extension offers to create a new record and generate a password. This extension is shown below.



One other feature Keeper offers is a security rating page. This page rates the security of every password in the account with clear, simple breakdowns of each record, however the metric Keeper uses to determine the strength of a password, and specific reasons why a password is weak are not visible. The security audit page is shown below.



Bitwarden

Bitwarden is a free open-source password manager. Bitwarden's vault page has much more wasted space than Keeper's, with records being listed in a centre column. Various filters and folders are shown in a panel to the left, with no obvious indication on the password records themselves which folders they belong to. To open a folder the folder must be clicked on the left panel, which narrows down the list of records to only those records which are assigned

to the selected folder, making Bitwarden's implementation of a folder system more similar to a filter system. Keeper's folder system appears superior in this regard, as multiple folders can be open and viewed simultaneously. Bitwarden's main vault page is shown below.

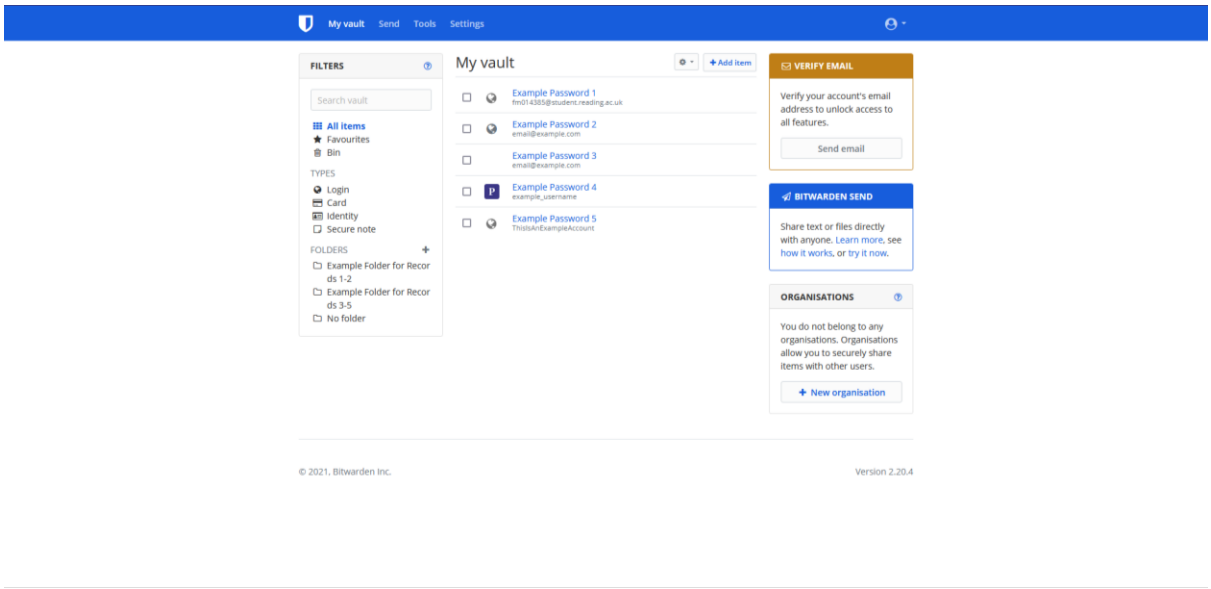


Figure 4.1 – Bitwarden

To view a record, one must click on the record in the centre column to expand information about the record. The record window in Bitwarden is much simpler to edit, with each field simply being a text box which can be typed in. Any changes are discarded unless a highlighted save button is clicked at the bottom. Bitwarden supports most of the standard fields, with fields for a record name, a username, password, and a notes section. Much like Keeper, Bitwarden supports custom fields, with the user able to choose from a text field, a hidden field, or a Boolean field. The records page is more upfront than Keeper's, with options being immediately visible as links and buttons rather than hidden behind drop-down menus, potentially making the records page easier to navigate for a less experienced user. Bitwarden also supports a field for authentication keys, much like Keeper. Instead of URLs, Bitwarden uses a uniform resource identifier (URI) system, supporting multiple identifiers such as URLs, IP addresses, or mobile app package IDs. This allows for a greater range of identifiers to be used but may be confusing for a less knowledgeable user. A password record in Bitwarden is shown below.

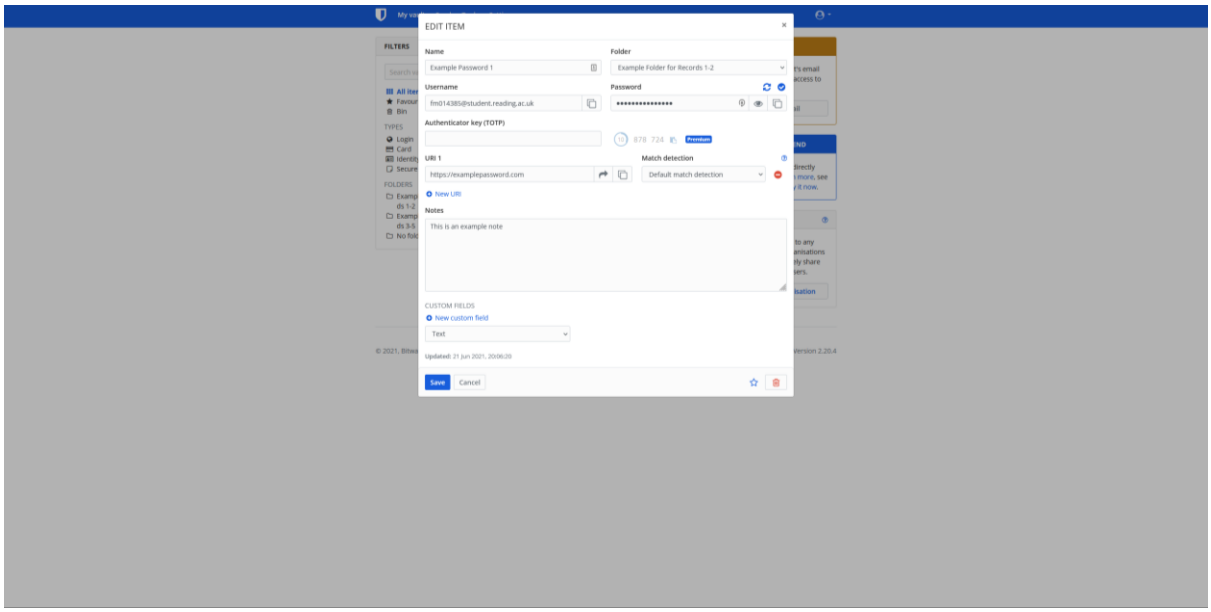


Figure 4.2 – Bitwarden

With regards to password generation Bitwarden's record page lacks behind Keeper in terms of customisation. Just above the field for a password a button offers to generate a new password, however there are no settings for this generation. A randomised password including lowercase letters, uppercase letters, numbers, and symbols is immediately generated and filled in the password field. In addition to the records page, however, Bitwarden has a tools page which contains a more detailed password generator, offering a checklist of character types to include along with a customisable length, a minimum number of numbers and symbols, and an option to avoid ambiguous characters. Bitwarden also includes the option to generate a passphrase instead, randomly selecting a set number of words and putting them together. Nielsen et al believe "One way to deal with the failings of password based authentication systems is to increase the length without necessarily increasing the complexity of recalling the password". Passphrases are then suggested as "Most people find it easier to remember passphrases, consisting of real sentences, than passwords created from random characters" (Improving usability of passphrase authentication, 2014). This suggests that Bitwarden's passphrase generator could be more effective for password security than a standard password generator. The more detailed password generator is shown below, followed by the passphrase generator.

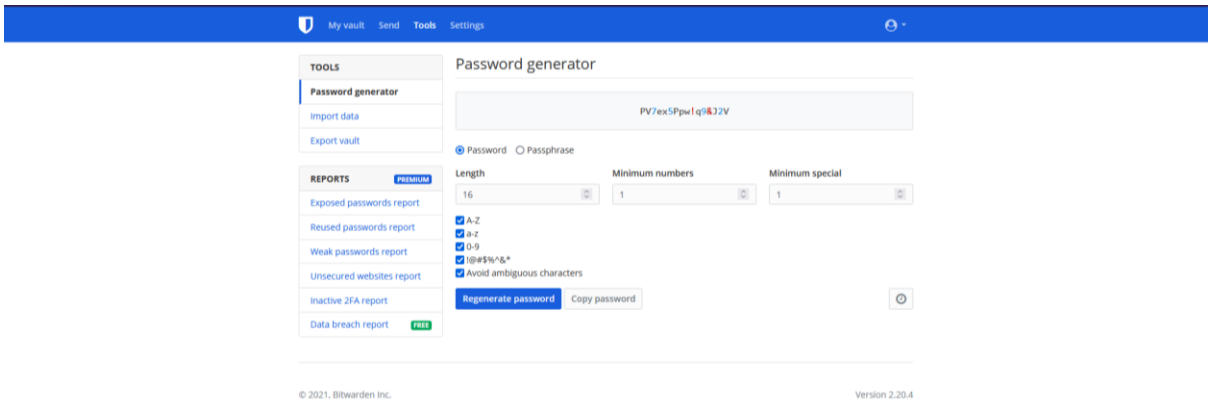


Figure 4.3 - Bitwarden

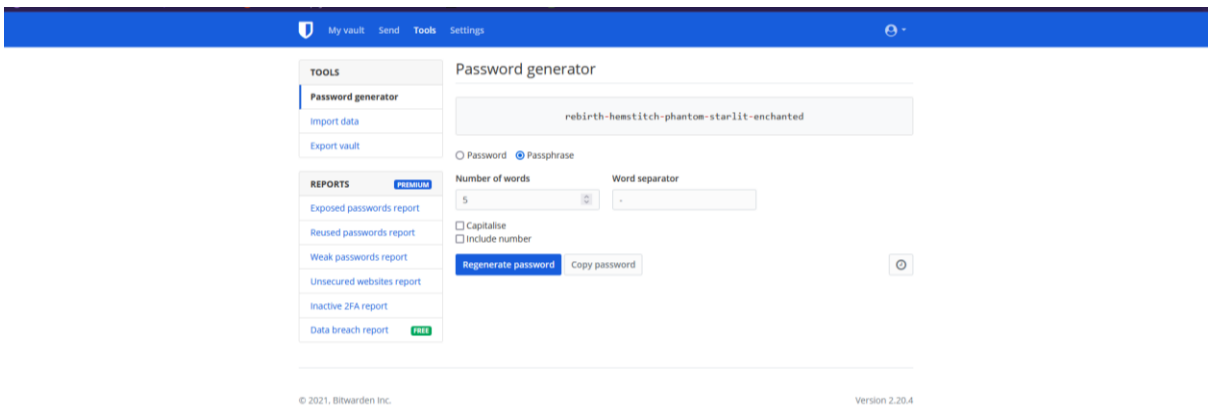
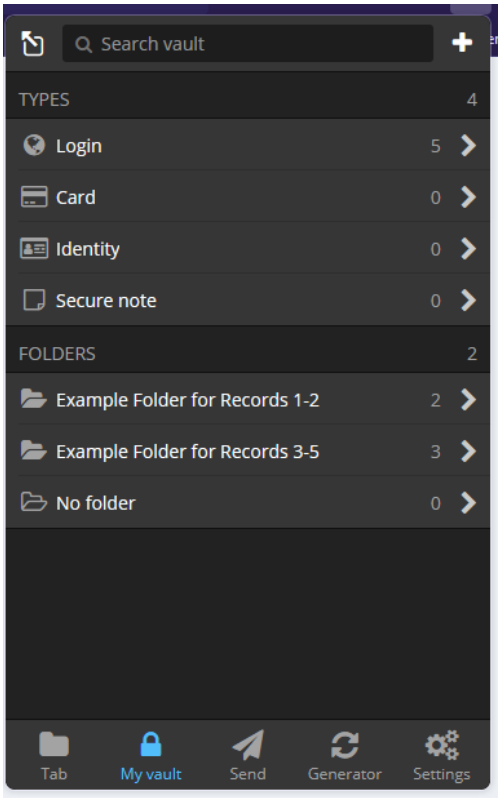


Figure 4.4 – Bitwarden

Under the remainder of the tools page Bitwarden offers features to import and export password data, and premium features to analyse whether stored passwords are weak or have been exposed. Under the settings page Bitwarden first offers account settings, with options to change the account name and password hint at any time, with the master

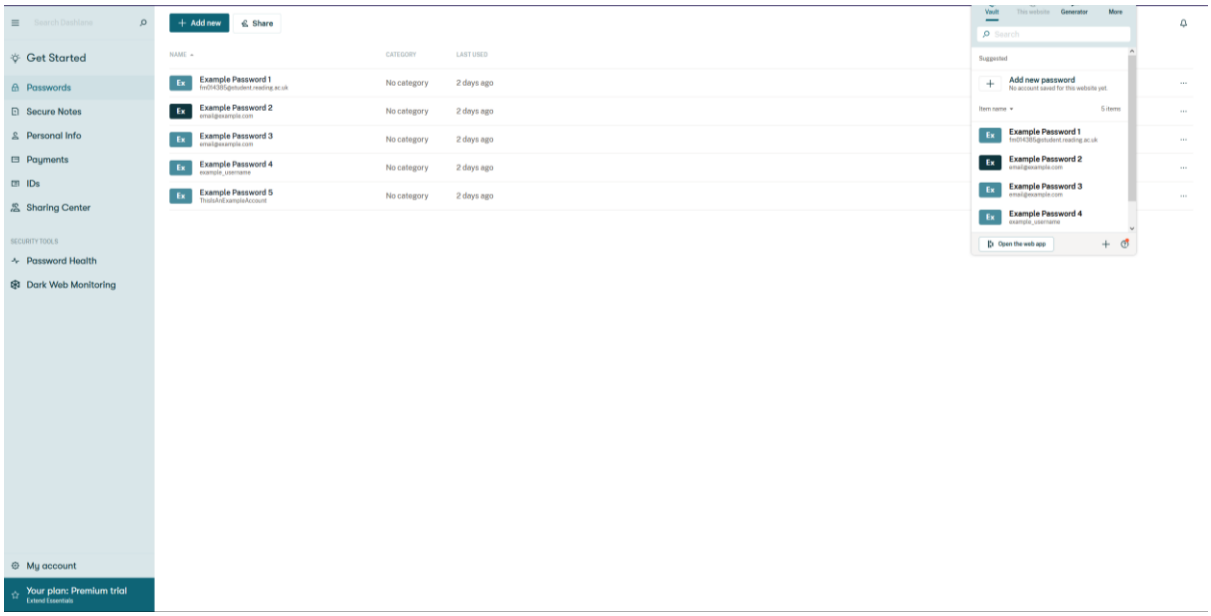
password and the account email only being changeable once the current master password has been entered to confirm. Settings are also offered to change the encryption of the account, such as changing the number of key derivation function iterations. The second section of the settings page refers to customisation options and allows the user to set an interval after which their account will be locked and will require the master password to be re-entered. This page includes a few other miscellaneous settings such as setting an avatar for the account. In a third “organisations” tab Bitwarden allows the user to add groups they can share passwords with. The final tabs of the settings page offer billing information, emergency contacts, and two-factor authentication. Bitwarden’s two-factor authentication supports various mobile apps such as Google Authenticator and physical security keys such as Yubikey.

Bitwarden also offers a browser extension, which shows any logins for the website open in the current tab, along with tabs to search the entire vault, generate a new password, and change settings for the extension. While the Keeper extension placed an icon next to any login forms, Bitwarden’s extension can autofill a login form by right clicking on a username or password field and selecting autofill. An option is also given to generate a password and copy it to the clipboard; however, the extension does not automatically fill the field. This implementation is less obvious than Keeper’s icon, and many users may not find this feature. Upon logging in to a new account Bitwarden does not have a record for, a banner appears at the top of the tab offering to automatically add the account to the user’s vault. Bitwarden’s extension is shown below.

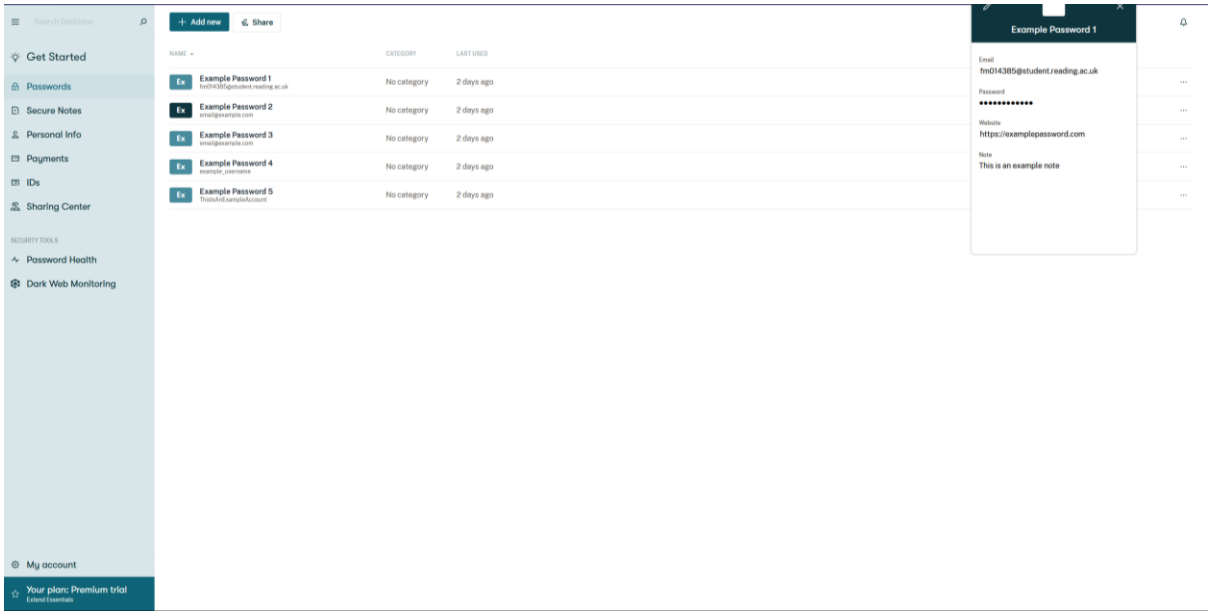


Dashlane

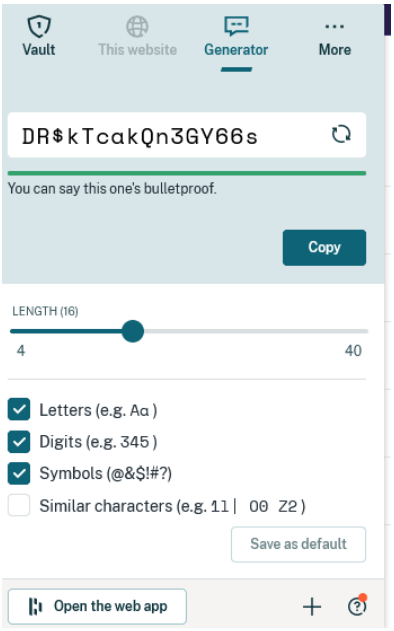
Dashlane is a password manager that functions through a browser extension. Rather than being a website which can be accessed from any device, Dashlane requires the user to have installed the extension to use it. Upon clicking the icon for the extension, it expands similarly to the Keeper extension.



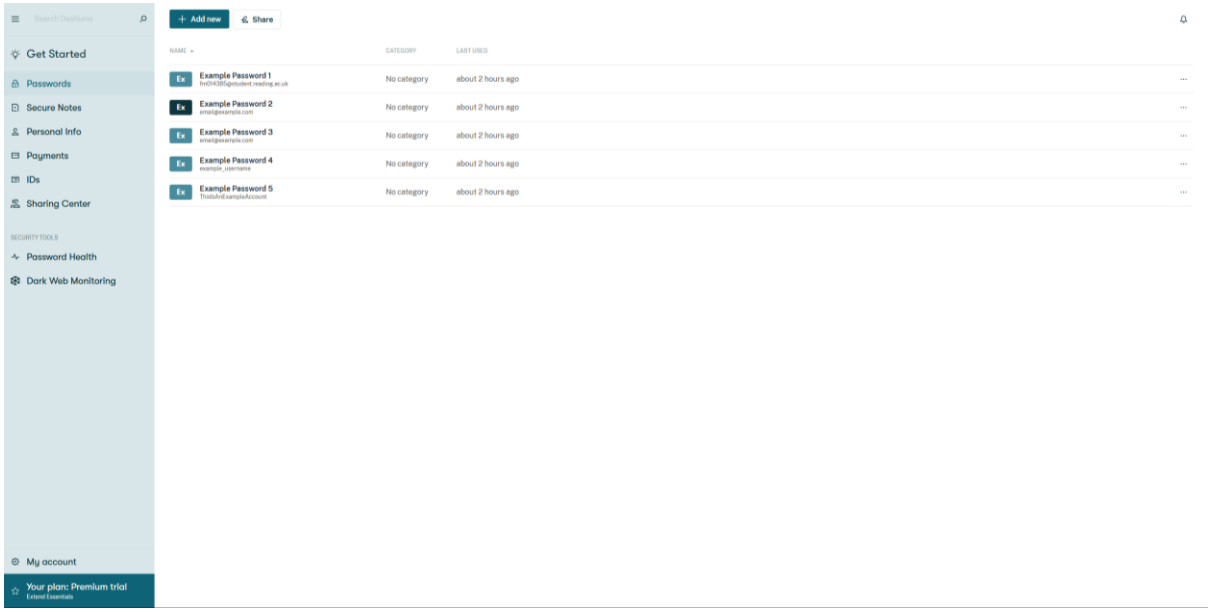
The user's vault can be accessed and searched from this extension and clicking on a password will open that password record, shown below.



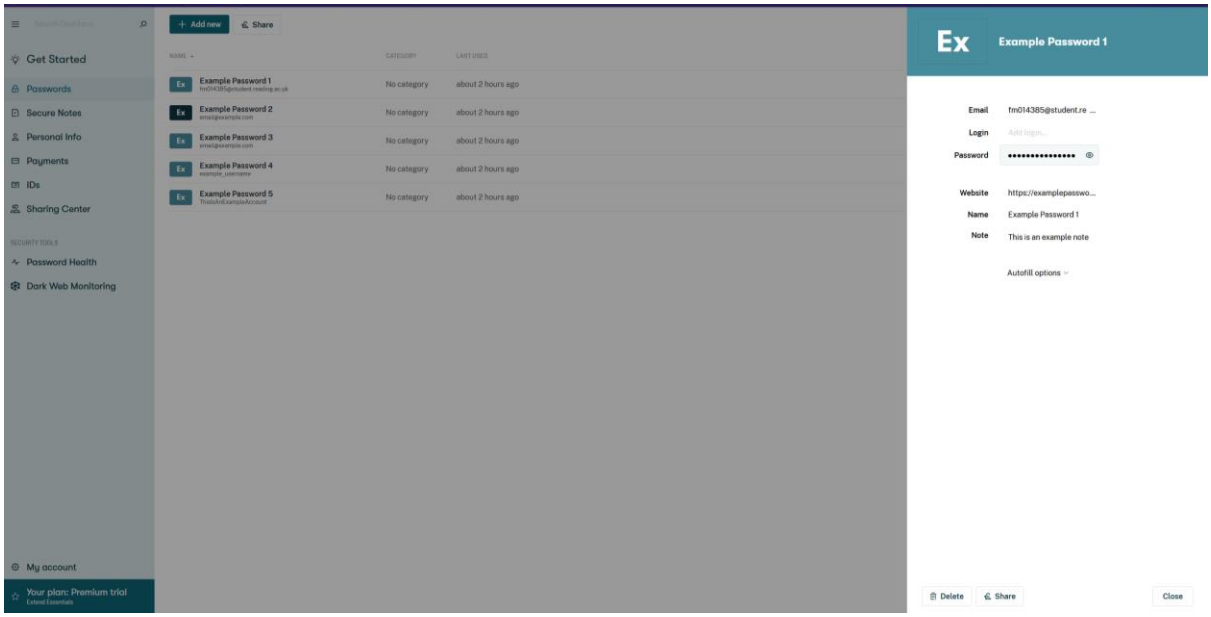
Dashlane's extension also includes a password generator, automatically displaying a generated 16-character password including letters, numbers, and symbols. No distinction is made between uppercase and lowercase letters. This generator is shown below.



Upon attempting to create a new password, or upon clicking the button labelled “open in web app” the larger web app opens, showing a vault much more similar to the other password managers. Aesthetically, Dashlane’s web app is closer to Keeper than Bitwarden, with records shown in a full-window width list next to a large sidebar which can be minimised to just icons. Each section of the web app is clearly labelled on the sidebar, which should make the app easy to navigate. A search bar is clearly presented at the top of the sidebar with placeholder text and an icon, clearly identifying it as a search box. Dashlane uses a relatively muted colour scheme and mainly uses shades of green which could aid accessibility as there are no additional colours for colour blind users to confuse.



Clicking on a password record in the vault expands the record in a new panel to the right. This panel shows all the fields of the password along with buttons at the bottom to share or delete the record. Autofill options for use with the browser extension are also available. Interestingly, Dashlane includes separate fields for an email and a login, whereas both Bitwarden and Keeper elected to treat the email of a record as a type of username. Each field is a text box and can be edited simply by selecting the field and starting to type. Dashlane includes no folder system, and passwords must be found simply by either scrolling through the list of all records or by searching for them. Dashlane also does not include custom records. The Dashlane vault with an expanded record is shown below.

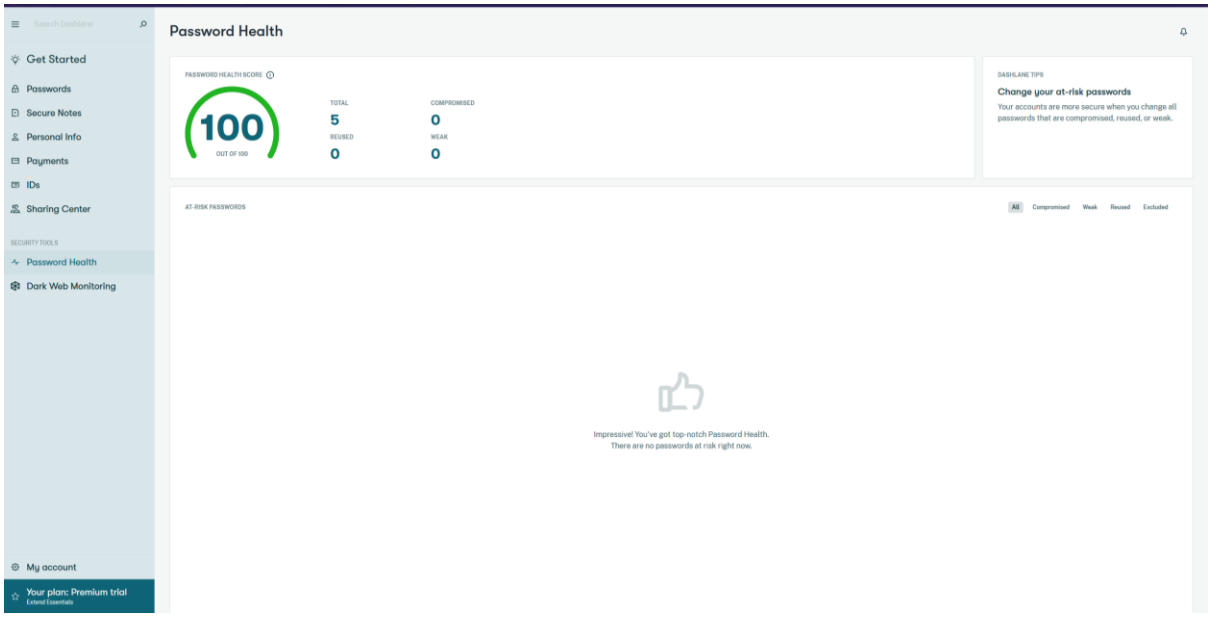


Creating a new password in Dashlane is started by clicking the green button at the top of the vault. This expands a blank record panel to the right, with empty fields. While Keeper and Bitwarden required the record to be given a name, Dashlane only requires that one field has a value, which could potentially make finding the record again later difficult. Password generation in Dashlane takes place solely in the extension, with no generation offered in the web app.

Dashlane's settings page is accessed from the "My account" button at the bottom of the sidebar which expands another panel to the right. From here users can view and update their account information, manage any devices connected to the account, import and export password data, and set up two-factor authentication for the account. Dashlane's two-factor authentication only supports biometrics and does not appear to support mobile or physical security key authenticators.

In terms of other features Dashlane supports the storage of personal information, such as emails, credit card and banking information, and IDs. Dashlane also offers a "Dark Web Monitoring" feature, promising a tool which "scans the dark web for leaked or stolen personal information". If a tracked email address is found the user will be notified so they can act. Finally, Dashlane has a "Password Health" page, much like Keeper's security audit page. This page shows a password health score out of 100, with any at-risk passwords displayed below. Interestingly, Dashlane gave differing results from Keeper's security audit -

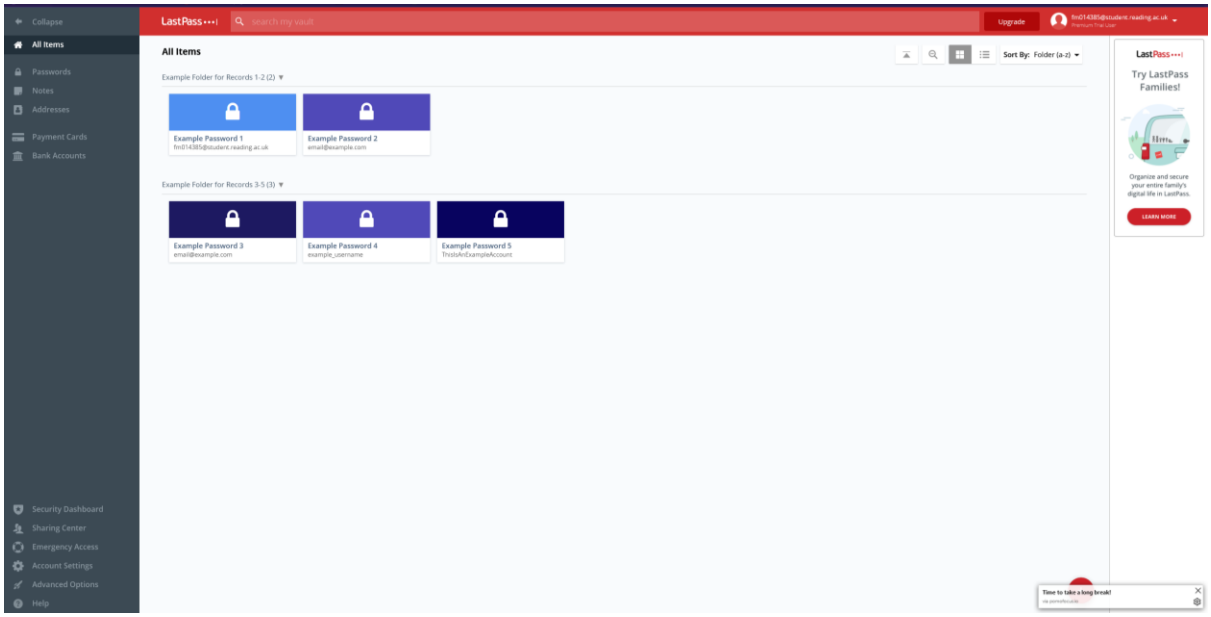
though as neither service explains the metrics used to measure password security it is difficult to see why. Dashlane's password health page is shown below.



LastPass

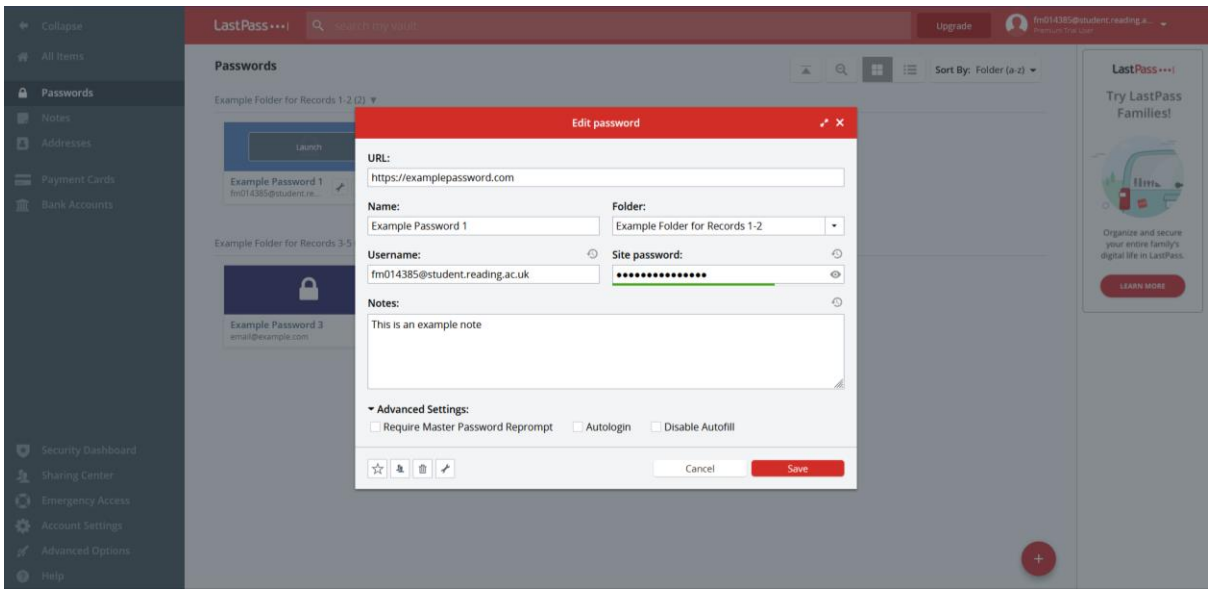
LastPass is another popular password manager which operates primarily through a web app rather than an extension. LastPass' vault page displays records primarily as cards, with various buttons to change the layout offered above the list of records. These views include a more compact view, and a list view. Records are listed in folders which can be collapsed, much like Keeper's vault design. LastPass has little wasted space, with the vault taking up the full size of the window, two sidebars, and a top navigation bar with a search bar. The top search bar narrows down the list of passwords displayed in the vault, rather than opening a drop-down menu with the search results, and next to the search bar is a button showing the user's avatar (if one has been set) and email address. When clicked a drop-down menu is expanded which offers account settings, a forum-based support centre, and a log-out button. The support centre could be incredibly valuable for a new user initially getting to grips with the software, offering a dedicated place to learn from others and ask any questions about LastPass. The right sidebar contains little content valuable to the user, only containing an advert for LastPass' other services, while the left sidebar is used for

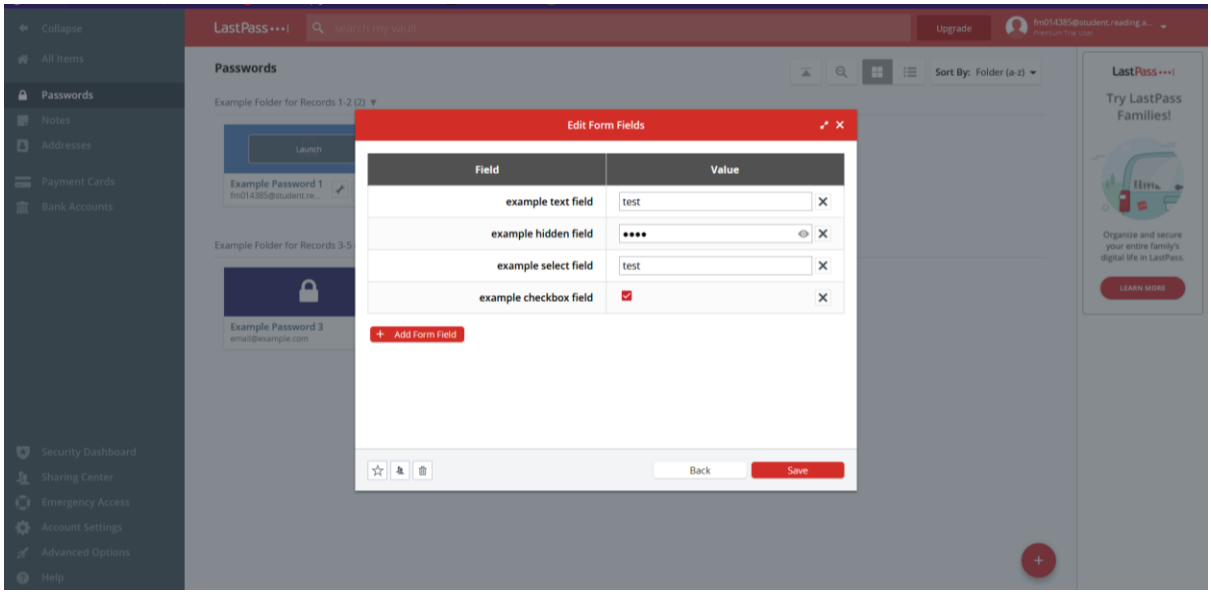
navigation, containing a list to different categories in the vault such as passwords, addresses, and payment cards. These categories are clearly labelled along with icons, which should make navigating the website as painless as possible for users. The home vault page for LastPass is shown below.



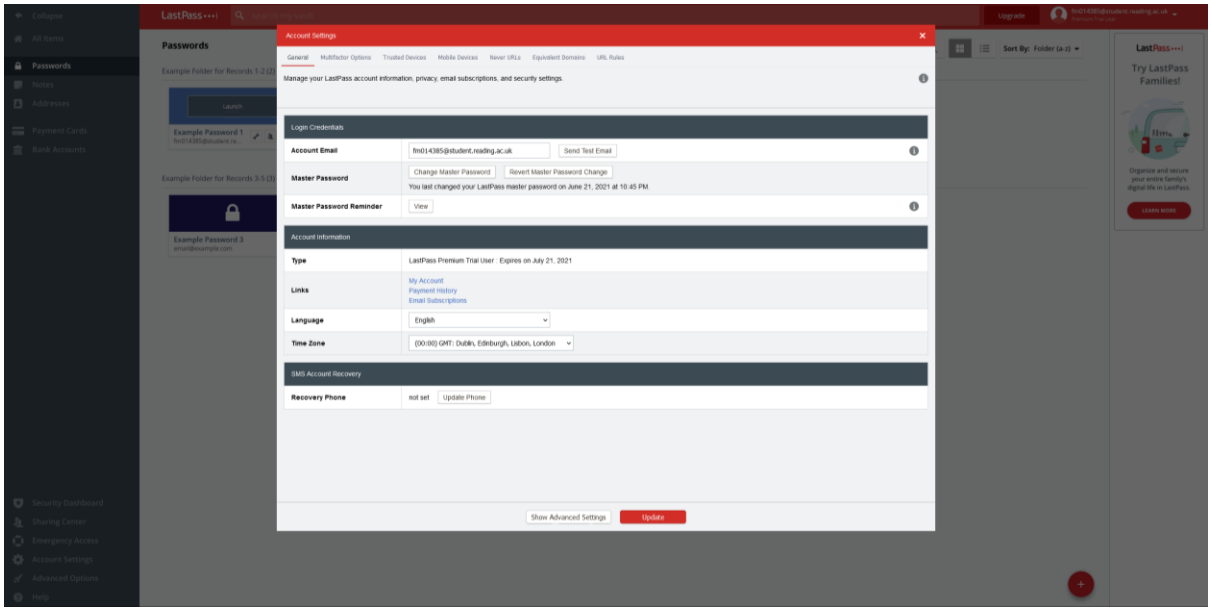
When a record in the vault is hovered over, buttons are displayed over the card. Three smaller buttons can be clicked to edit, share, or delete the record, while most of the card is dedicated to a launch button which acts as a shortcut to immediately open the record's corresponding website. This could potentially become a source of frustration for a user, as if they click the centre of the card for a record expecting to see an expanded record they would instead be redirected to a different website. Clicking on the record anywhere other than these four buttons selects the record and would allow the user to select multiple records for deleting or moving to a different folder. Clicking the edit button opens an expanded record view like the previously examined password managers. LastPass has all the standard fields, each being a textbox, with values that can be changed by simply selecting the field and typing, unlike Keeper. LastPass has a drop-down menu towards the bottom of the record window for advanced settings where the user can toggle whether their master password will be required to view the record again, and whether this webpage will be automatically logged in to. LastPass includes buttons for viewing the version history of some fields, allowing the user to view previously viewed usernames, passwords, and notes for the

record. This could be extremely useful for users who mistakenly change values in the wrong record, and now find themselves unable to reverse the changes they have made. Buttons to cancel any changes, save any changes, edit custom fields, delete the record, share the record, and mark the record as a favourite are placed at the bottom of the record window, with most of them identified by iconography. Custom fields are strangely placed behind one of these buttons, instead of being appended to the main record as they are in Keeper and Bitwarden. The placement of this feature could make it difficult for users to find or quickly refer to later. The image below shows a LastPass record followed by example custom fields assigned to the record.

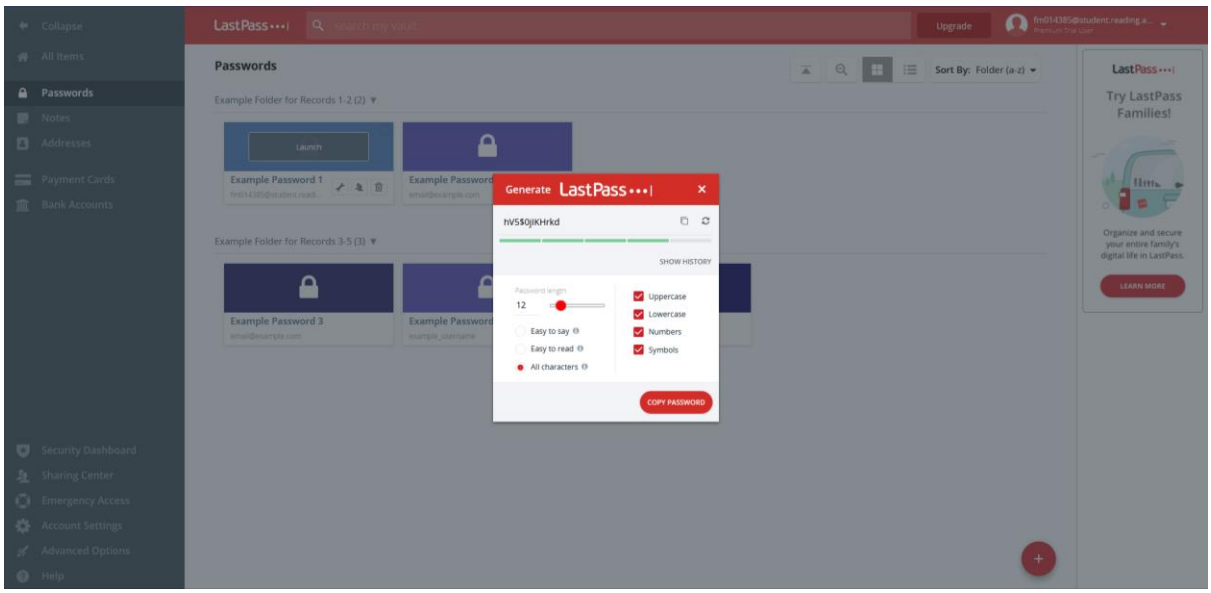




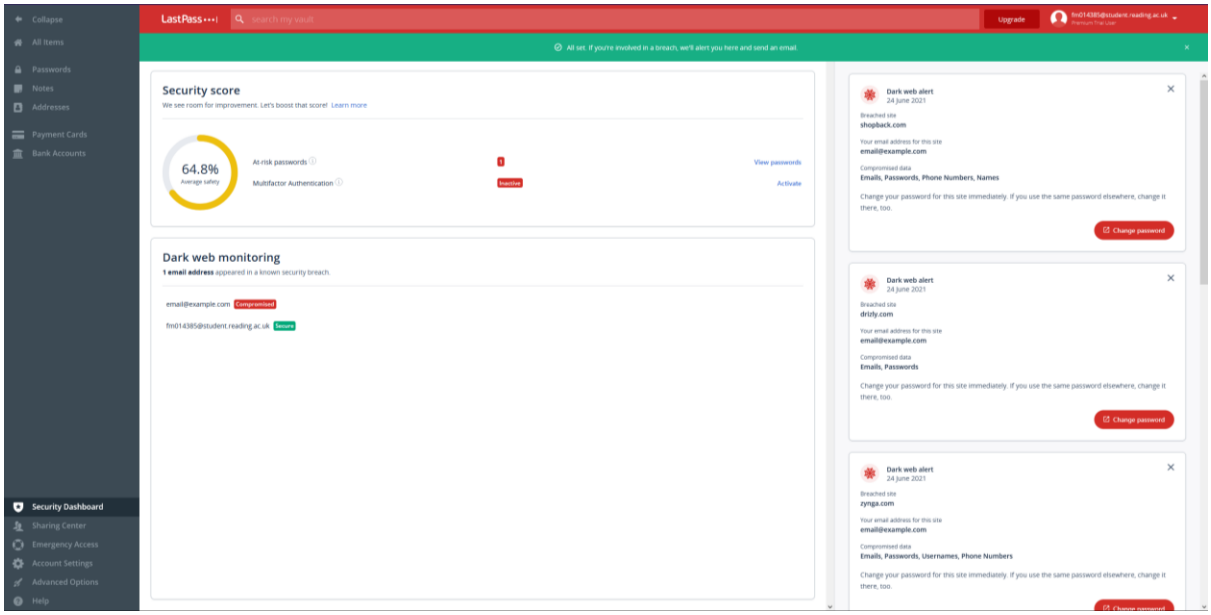
The settings page of LastPass can be accessed either from the drop-down menu on the navbar under the username or through a button on the sidebar and appears much more detailed than the other password managers'. The general page allows the user to change account information such as email, master password, and master password reminder, along with settings other account information such as a preferred language and time zone. The second tab of the settings window is for multi-factor authentication, and it lets the user set up authentication with a range of apps or physical keys. The third and fourth tabs allow the user to manage devices which have access to the account, and revoke this access, while the remaining tabs govern autofill settings, such as blocking the LastPass browser extension from filling certain websites. LastPass has a second, advanced options settings window which allows the user to perform other actions such as importing or exporting password data, upgrading their subscription, view favourite records, or generate a new password. The main settings page is shown below.



When it comes to password generation, LastPass has a panel which allows the user to select various settings. The default password length is 12 characters, with checkboxes offered to include uppercase, lowercase, numbers, and symbols. The user can customise the length of the password with either a slider or a text field, and radio buttons are offered to make the password easier to say or read by avoiding numbers, symbols, or ambiguous characters. Buttons are then displayed to allow the user to regenerate the password or copy the password to the clipboard for use. The password generator is shown below.

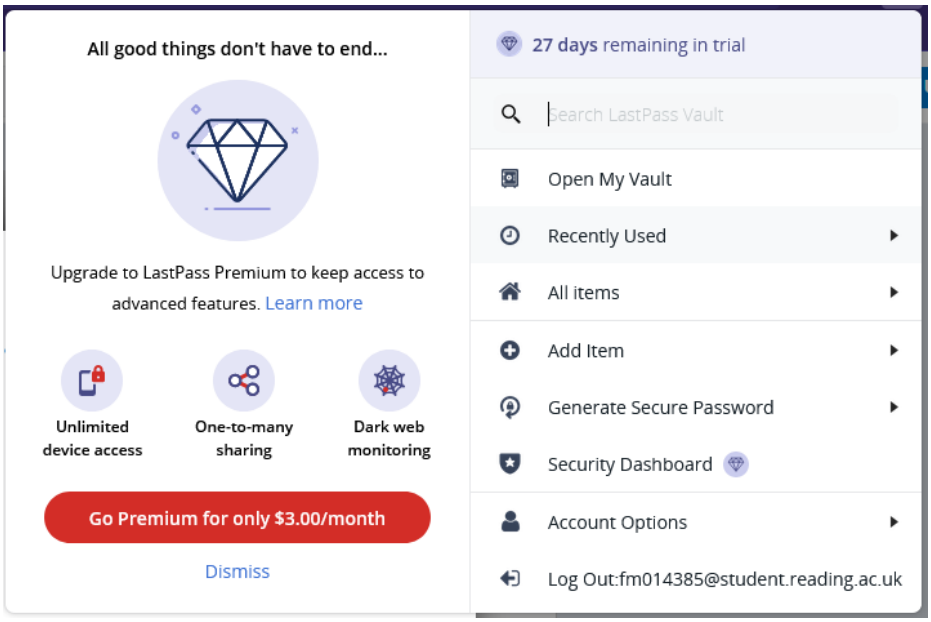


In terms of other features LastPass includes storage of private items other than passwords, including debit and credit cards, bank account information, and addresses. LastPass also includes a security dashboard, which fills a similar role to the security audit page in Keeper or Dashlane's password health page. The security dashboard shows an overall security score which rates the average safety of all passwords, along with showing any passwords determined to be at risk. For this account LastPass also suggested turning on multi-factor authentication to improve the account's security. Much akin to Dashlane, LastPass offers a dark web monitoring feature, which promises to proactively alert users "if sites from your vault are breached. Monitor these addresses. All day, every day". Upon opting into the service, the master account email address and the email address of any other records are automatically searched and identified if they appear in any known security breaches. Alerts are then shown on the right-hand side for any compromised email address, describing the security breach and the specific data that was leaked so that the user can change compromised information to protect themselves. The security dashboard is shown below.



Like every password manager seen thus far, LastPass offers a browser extension. This extension can be used to autofill login forms for websites with existing records. In a manner very similar to Keeper and Dashlane an icon is placed at the end of username or password fields. If LastPass finds any corresponding records it will display them in a list for the user to select for auto-filling. If LastPass detects no such records, it will offer to generate a new

password for a new record. Upon manually logging into a website for the first time the extension will pop up with a message offering to automatically add a record for the website with the username and password used. Clicking on the icon for the extension in the top right of the browser opens a menu with various options, including a search bar to search for records within the smaller extension window, a shortcut to open the user's vault, and password generation within the extension. The main menu of the extension is shown below.



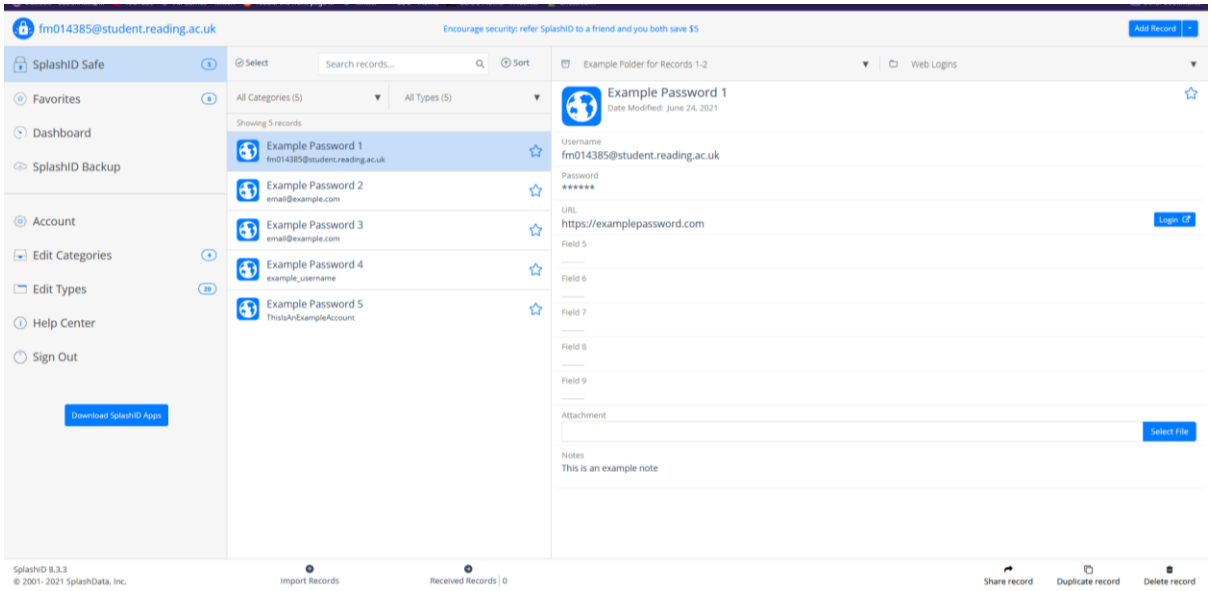
SplashID Safe

SplashID is the final password manager this report will examine. SplashID's vault page is divided into three segments, with a top and a bottom navigation bar. The top navigation bar consists of text displaying the master account email address along with a button to add a new record or category, while the bottom navigation bar contains labelled buttons to import, share, duplicate, or delete records. The first of the three main panels in SplashID's vault design is a sidebar used for navigation, with links leading to the user's main vault, a list of favourite records, and premium dashboard and backup features. The second panel shows a list of password records along with a search bar and various buttons to change how records are filtered and sorted. Each record displayed in the list is marked with either an empty or solid star to indicate whether the record has been marked as a favourite record, and the star can be clicked to toggle this state. The final, right-hand panel displays a password record in greater detail, showing the record's fields along with information about

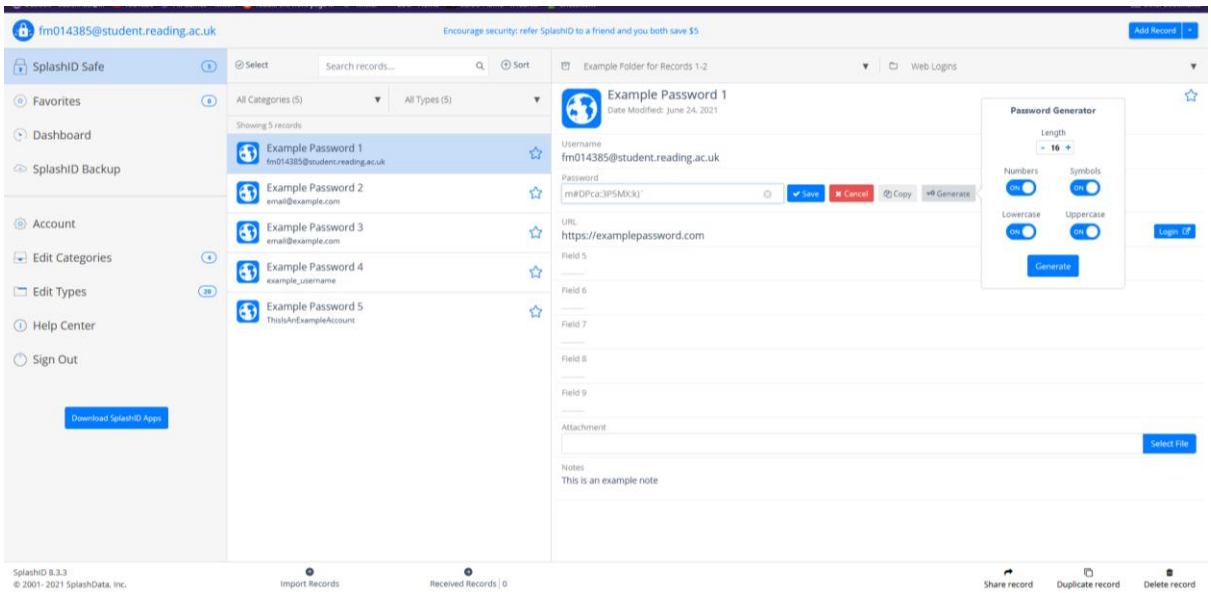
the record such as the date it was last modified and its type and category. Clicking on the type or category expands a drop-down list which allows the user to set a different type or category for the record. These types and categories act as SplashID's substitute for a folder system, with both acting as customisable assignable tags and folders used to identify records. Default suggestions for categories include business and personal, while built-in options for types include email addresses, bank accounts, and credit cards.

Perhaps confusingly, SplashID displays records with five empty fields with placeholder names. While the titles of these fields can be changed, they cannot be removed, and new custom fields cannot be added. The values of these fields can only be text values, meaning SplashID does not support the same diversity of custom field types as Keeper, LastPass, and Bitwarden. SplashID does support attachments, allowing seemingly any file type to be added to a record. SplashID also sports a login button next to the URL field of a record which can be clicked to open the corresponding URL and automatically log the user in.

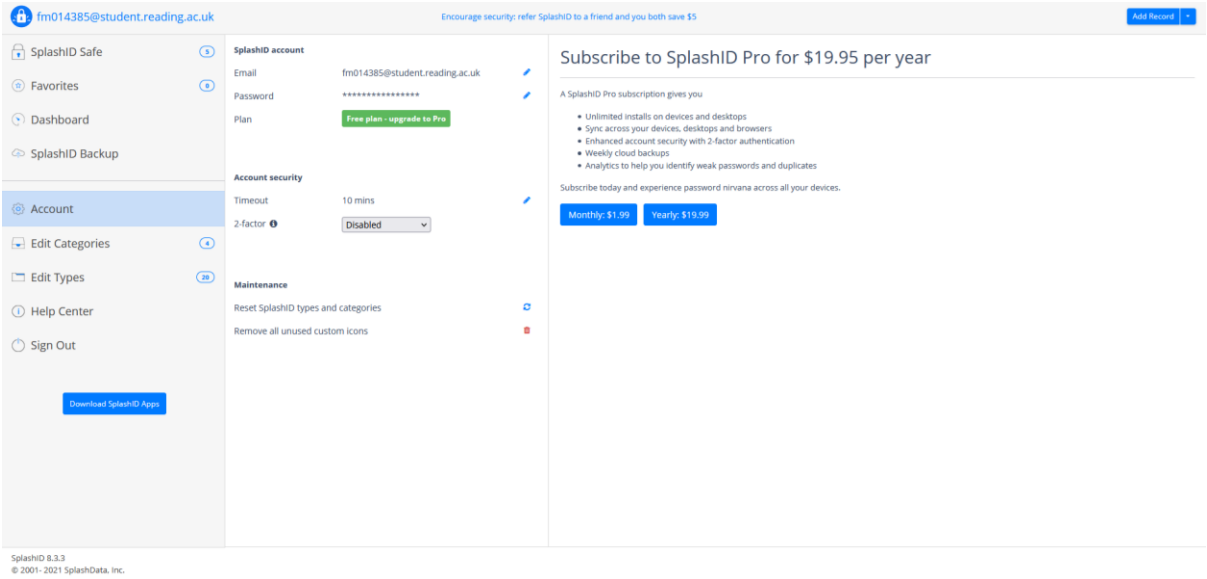
SplashID uses very little in the way of colour, keeping much of the page in shades of white and grey, while highlighting important icons and actions in blue. While appearing plain, this keeps contrast high, making the page clear and easy to read for visually impaired users, and the use of blue as the sole accent colour will help prevent confusion. The layout of the page, however, is clustered and busy which could leave users feeling claustrophobic as little space or padding is placed between items. This could be alleviated somewhat if the user were able to resize the various panels, however SplashID has not included this as an option. The main vault page of SplashID is shown below.



Password generation in SplashID is performed by clicking on a field within a password record. The field becomes an editable textbox and buttons appear to save changes, discard changes, copy the contents to the clipboard, generate a new password, and mask a field. The length of a generated password must be four characters or above with a default length of 16. Toggles are available to include lowercase, uppercase, numbers, and symbols. Clicking a highlighted “Generate” button immediately fills the selected field with the output. The SplashID password generator is shown below.

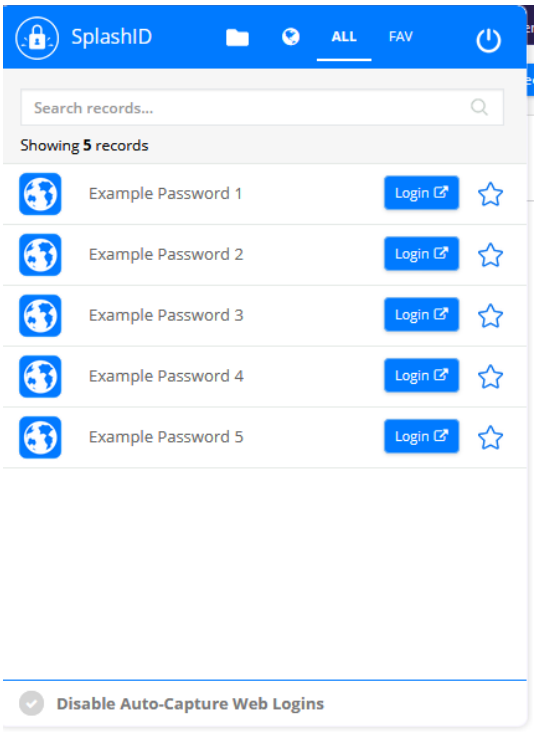


The account and settings page in SplashID is relatively bare, containing options for the user to change their master account email and password, upgrade their plan, change the length of time before the user is automatically logged out, enable two-factor authentication, and reset all types and categories. The two-factor authentication supported by SplashID is only offered through SMS text messaging or email authentication. Mobile apps and physical security keys are not supported. The settings page is shown below.



Similarly, to LastPass, SplashID also includes a link to a “Help Center”, offering the user a frequently asked questions page, and the option to submit a support form to receive help or advice from the SplashID team, although there is no forum section. This could be extremely valuable for any users who are easily confused or less experienced with password managers and being able to contact a support team member directly could especially help a user obtain an accurate and relevant answer rather than a forum system which relies more heavily on community members.

Like all previously examined managers, SplashID also offers a browser extension. This extension displays a searchable list of all records along with a tab to only show favourite records or records the extension believes are relevant to the currently open tab. The extension does not support password generation or the creation of a new record, these functions must be performed from the main web app which can be accessed through the SplashID logo, which acts as a shortcut. The browser extension is shown below.


SplashID
ALL
FAV
Search records...
Showing 5 records
Example Password 1
Login
Example Password 2
Login
Example Password 3
Login
Example Password 4
Login
Example Password 5
Login
Disable Auto-Capture Web Logins

Solution Approach

The chosen solution for this project was to create a Javascript web app using the MERN software stack consisting of MongoDB for the database, Express for routing and HTTP requests, React as the client-side frontend framework, and Node as the server runtime. NestJS, a Javascript framework was selected to be used in the construction of the backend to provide advantages such as “a level of abstraction above these common Node.js frameworks” along with “[exposing] their APIs directly to the developer” (Mysliwiec, n.d.). To integrate MongoDB with NestJS, a MongoDB object modelling tool for Node called Mongoose was chosen. Mongoose focuses on schema-based solutions to model data, each of which maps to a MongoDB collection (Mongoose v5.12.15, n.d.). To handle user authentication, a library called Passport was chosen as NestJS supports Password with a dedicated module. Passport can be used to authenticate users by defining strategies which determine when authentication succeeds or fails (Hanson, n.d.), and for this project users will be authenticated via JSON Web Tokens (JWTs), an open industry standard used to securely transmit data using digitally signed tokens (auth0.com, n.d.) and to authenticate users to access resources (Ethelbert, O et al. 2017). Bcrypt has been selected as the library for hashing, due to its key-stretching feature as identified in the literature review. The Advanced Encryption Standard has been chosen for encryption, as it was also identified in the literature review as a powerful encryption specification.

Much of the backend will be designed as a RESTful API, a type of API which meets 6 criteria: that client and server concerns are separated; that the API is stateless, meaning each request from the client to the server must contain all necessary information without using stored context on the server; that data is labelled as cacheable or non-cacheable so that data can be reused for equivalent requests; that there is a uniform interface between components; that the system is hierarchically layered; and that code is available on demand, extending client functionality by allowing the downloading and executing of code as scripts (Fielding, 2000). A significant portion of the project will be programmed using TypeScript, a language which compiles to Javascript.

The web-client will then consist of sign-up and login pages which direct the user to a main dashboard. This dashboard will act as a central recognisable page for the user, off which many components and most of the features will be usable.

Requirements Engineering

To create a list of requirements the MoSCoW prioritisation method was used. This method is used to identify the features that a project can include, from the most important features the project must have, to the least important features the project could have. The MoSCoW technique prioritises requirements based on the following criteria, “must have”, “should have”, “could have”, and “won’t have this time”. Must have describes requirements that must be included in the final product. Should have defines requirements which are high priority and should be included if there is time. Could have requirements are “desirable or nice to have”, but are the first to be dropped if the project will not be finished in the timeframe. Finally, “Won’t have this time” describes features a stakeholder would desire, but a consensus is reached that the requirement will not be implemented at this time (Ahmad et al., 2017).

In the list below various features have been identified as features the project must have, should have, could have, or features the project will not have. These features have been identified based on the review of existing password managers, along with the literature review. Optional feature 1 especially was inspired by the literature review, particularly describing key stretching methods used to slow down attempts at brute forcing or guessing password hashes (Blocki, J. and Datta, A., 2016). Optional feature 4 was inspired by the review of existing managers, particularly Keeper’s option to select alternate colour themes for the dashboard. Two factor authentication was determined to be out of scope as the budget for this project did not support fingerprint sensors or physical security keys. Custom fields were also determined to be out of scope as many of the existing password managers either did not support the feature, or supported the feature in a manner not significantly distinct from a notes field.

Mandatory features (Must have):

1. Users are able to store usernames and corresponding passwords.
2. Users are able to retrieve usernames and corresponding passwords.
3. Users are able to delete entries.
4. Users are able to update entries.

5. Users are able to search for specific entries.
6. Users are able to create an account.
7. Users are able to log into an existing account with their username and password.
8. Users are able to update their account details.
9. A password generator tool to allow the user to generate strong, random passwords.

Desirable features (Should have):

1. A folder system can be used to organise records.
2. A master password hint can be offered to remind the user of a forgotten master password.
3. The user's password is stored as a hash.
4. The passwords on the database are encrypted.
5. The application should use HTTPS over HTTP.

Optional features (Could have):

1. If a user makes a certain number of unsuccessful attempts to log in in a row, they are locked out of their account for one hour.
2. The user receives a verification email upon sign-up to verify their account.
3. The user can store secret questions to be used in the event their password has been forgotten and their account needed to be recovered.
4. The user can select an alternative colour palette.

Excluded features (Will not have):

1. The user can store custom fields in a record.
2. Security evaluator.
3. Two-factor authentication.
4. Biometric authentication.
5. Physical security key authentication (such as Yubikey).

Implementation

The construction of this project began with the backend, particularly the MongoDB database. Once the fundamentals and essential files had been set up using NestJS various schemas were designed with Mongoose to encapsulate the functionality that would be required of the database as identified during requirements engineering. Schemas are used with Mongoose to define a model, an instance of which is a document – a term MongoDB uses for records which consist of field-and-value pairs (Mongodb.com, n.d.). Schemas were designed for users and for password records. This was done as both objects require different properties to be stored. The user model must store details for a user's master account, while the password model was designed to store the details for a user's external service.

The user schema contains properties for the user's email address, master username, the hash of their master password, their password hint, their role which is used for authorisation, a Boolean to remember whether they have verified their email address, secret questions, a value to store the number of consecutive failed login attempts, and the date of their last login attempt. The last four variables were intended to be used to implement desirable feature 2, and optional features 1 and 2. The secret questions property would have been used to offer an alternative to the password hint, allowing the user to set a question with an answer only they would know. Upon answering the question correctly an email would be sent to the user to allow them to reset their password. The failed attempts and last attempt date properties would have been used to implement optional feature 1, as the number of attempts a user makes to login could be stored. Once this number reached a certain threshold the user would not be allowed to login until it had been at least one hour since their last login attempt. The intention behind this feature was to slow down any attempts at guessing or brute forcing a password.

The first property of the password schema is the corresponding user's ID, used to identify which user's record it is and fetch password records for a user. The second property is the actual password of the record. The iv variable stands for initialisation vector and is used to encrypt the password in the record. The name, URL, username, and notes properties are all other details of a password record and have been included to store these details for a user. The decision behind what details to include was informed by the review of existing

managers, particularly the various ways password details were stored and displayed in them.

```typescript
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';
import { Role } from 'src/roles';
import { enumValues } from 'src/utils';

export type UserDocument = User & Document;

@Schema()
export class User {
  @Prop({ required: true })
  email: string;

  @Prop({ required: true })
  username: string;

  @Prop({ required: true, enum: enumValues(Role), default: Role.User })
  role: Role;

  @Prop({ required: true })
  passwordHash: string;

  @Prop()
  passwordHint: string;

  @Prop({ required: true, default: false })
  hasVerifiedEmail: boolean;

  @Prop()
  secretQuestions: { question: string; answer: string }[];

  @Prop({ required: true, default: 0 })
  failedAttempts: number;

  @Prop()
  lastAttemptDate: Date;
}

export const UserSchema = SchemaFactory.createForClass(User);

```

```typescript
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';

export type PasswordDocument = Password & Document;

@Schema()
export class Password {
  @Prop({ type: 'ObjectId' })
  userID: string;

  @Prop()
  password: string;

  @Prop()
  iv: string;

  @Prop()
  name: string;

  @Prop()
  URL: string;

  @Prop()
  username: string;

  @Prop()
  notes: string;
}

export const PasswordSchema = SchemaFactory.createForClass(Password);
```

With schemas for the database designed, the next step was to begin work on the endpoints. When a client communicates with the backend, the requests made by the client will reach a certain endpoint, and the server must perform certain actions depending on the nature of the request and the endpoint it was sent to. In NestJS, endpoints are built using controllers, providers, and modules. Controllers are responsible for handling incoming requests and returning responses to the client (Mysliwiec, n.d.). In essence, the controller receives the requests, determines the correct service needed, and makes the correct service calls in the providers before returning any responses or HTTP status codes to the client.

The service, on the other hand, acts as a provider for the complex requests, carrying out the request and performing the necessary operations. In MongoDB, create, read, update, and

delete (CRUD) operations are used to interact with the database. (Truica, Boicea and Trifan, 2013). Create operations are used to create a collection and add any new documents. Read operations are used to retrieve data from the database. Update operations are used to edit a record. Finally, delete operations are used to delete a document (Truica, Boicea and Trifan, 2013) (Mongodb.com, n.d.).

Controllers and services were made for user and password endpoints, with the user endpoint being used to deal with CRUD operations for users and the password endpoint being used to deal with CRUD operations for an individual user's passwords. The API was separated into user and password sections to provide a clear, intuitive structure and obvious breadcrumbs within the endpoints.

Separating the controllers and services provides a clear hierarchical structure to the API, one of the essential criteria of a RESTful API.

As an example, the first two decorators from the user controller are shown below.

```
17  @Controller('api/users')
18  @ApiTags('User')
19  export class UserController {
20    constructor(private readonly userService: UserService) {}
21
22    @Get('')
23    @Roles(Role.Admin)
24    @UseGuards(AuthGuard(JWT_STRATEGY), RolesGuard)
25    async getAllUsers(@Res() response: Response): Promise<void> {
26      const result = await this.userService.getAllUsers();
27      response.status(HttpStatus.OK).send(result);
28    }
29
30    @Post('')
31    @Roles(Role.Admin)
32    @UseGuards(AuthGuard(JWT_STRATEGY), RolesGuard)
33    async createUser(
34      @Body() body: UserCrudDto,
35      @Res() response: Response,
36    ): Promise<void> {
37      response
38        .status(HttpStatus.CREATED)
39        .send(await this.userService.createUser(body));
40    }
```

Both of these decorators handle requests made to the /api/users endpoint, however the first decorator deals with HTTP GET requests and the second deals with HTTP POST requests. Both consist of an asynchronous function which calls a certain function in the user service. While the GET decorator does not need to pass any additional information to the service function, the POST decorator passes the body of the response. The two code snippets below show the corresponding functions in the user service. The getAllUsers function simply finds every user and returns them, while the createUser function performs a little more business logic. The createUser function uses the body of the request to create a new userModel, which is used to map the request to the format of a user which can be saved to the MongoDB database.

```
17    /**
18     * Get all users in the database, only executable by an admin
19     */
20    async getAllUsers(): Promise<UserDto[]> {
21      const result = await this.userModel.find().exec();
22      return result.map(fromUser);
23    }
```

```
43    /**
44     * Create a new user and save it to the database
45     * @param {UserCrudDto} createUserDto A DTO of the new user to be saved
46     */
47    async createUser(createUserDto: UserCrudDto): Promise<string> {
48      const createdUser = new this.userModel(createUserDto);
49      const result = await createdUser.save();
50      return result._id;
51    }
```

It should be noted that in the createUser function the body of the POST request is used as a UserCrudDto. Data transfer objects (DTOs) are objects which define how data will be sent over a network (Mysliwiec, n.d.) and were created to describe the properties of an object transmitted over the network. Three DTOs were outlined for users and passwords, each following the same pattern. The UserBaseDto contains only properties used by both of the other DTOs, which for the user is a username and an email. The UserCrudDto contains properties needed to perform CRUD operations and extends UserBaseDto but also includes password and passwordHint properties. Finally, the UserDto extends UserBaseDto, however, it additionally includes required id and role properties which will contain a unique

ID and a role - both of which will be assigned. The UserDto also contains a function which can be used to map from a UserDocument - defined by the user schema - to a UserDto.

The password DTOs follow suit, with the PasswordBaseDto containing reused properties, the PasswordCrudDto containing additional properties which can be used to perform CRUD operations, and the PasswordDto containing an id property and a function to map from a PasswordDocument to a PasswordDto.

The DTOs are separated to prevent unnecessary transmission of data, as RESTful APIs are designed to be scalable and maintain network efficiency (Fielding, 2000). Part of the purpose behind a RESTful API being cacheable is to limit the amount of wasted data transmitted to maintain scalability and network efficiency and transmitting unnecessary data would clash with this concept. The user DTOs can be seen in the three code snippets below.

```
1  export interface UserBaseDto {
2    username?: string;
3    email?: string;
4  }
5
```

```
1  import { UserBaseDto } from './user-base.dto';
2
3  export interface UserCrudDto extends UserBaseDto {
4    password?: string;
5    passwordHint?: string;
6  }
7
```

```
import { Role } from 'src/roles';
import { UserDocument } from 'src/schemas/user.schema';
import { UserBaseDto } from './user-base.dto';

export interface UserDto extends UserBaseDto {
  id: string;
  role: Role;
}

export function fromUser(user: UserDocument): UserDto {
  return {
    id: user._id,
    username: user.username,
    email: user.email,
    role: user.role,
  };
}

```

With the endpoints constructed, the next step was to handle authentication and authorisation. Authentication involves ensuring that a user is who they claim to be, and authorisation involves ensuring that a user can only perform actions they should be allowed to perform.

Authentication was performed through the use of JSON Web Tokens as identity verification. When a user logs in an HTTP POST request containing their username an password is submitted to the api/auth endpoint handled by an auth controller. Much like the user and password controllers, this controller receives the request and calls a function in an auth service which performs the logic to authenticate the user.

```
1  import { Controller, Post, Request, UseGuards } from '@nestjs/common';
2  import { AuthGuard } from '@nestjs/passport';
3  import { Role, Roles } from 'src/roles';
4  import { AuthService } from './auth.service';
5
6  @Controller('api/auth')
7  export class AuthController {
8    constructor(private authService: AuthService) {}
9
10   @UseGuards(AuthGuard('local'))
11   @Post('login')
12   @Roles(Role.Admin, Role.User)
13   async login(@Request() req) {
14     return this.authService.login(req.user);
15   }
16 }
17
```

This logic is performed using the Passport library supported by NestJS. An AuthGuard method is placed around the login endpoint inside the controller which uses a “local” strategy to authenticate a user.

```typescript
import { Strategy } from 'passport-local';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UserDto } from 'src/dto/user.dto';

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy, 'local') {
  constructor(private readonly authService: AuthService) {
    super();
  }

  /**
   * Validates user and throws exception if the user could not be validated
   * @param {string} username The username
   * @param {string} password The password
   * @returns A promise of the user DTO
   */
  async validate(username: string, password: string): Promise<UserDto> {

    const user = await this.authService.validateUser(username, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}

```

This local strategy calls a method in the auth service used to validate a user. The validateUser method takes the user's username and password from the POST request, and finds the user on the database by username, throwing an exception if no such user exists. The submitted password is then hashed using Bcrypt and compared to the value for the user's password hash already stored on the database. If the hashes match a UserDto of the user who has logged in is returned. If the hashes do not match, the method returns null. With the user validated, the login method inside the auth service can be called. The login method takes the user and creates a payload of the user's username, ID, and role. A JWT token is created using this payload to authenticate them. The JWT token will then be cached by the client, in keeping with the RESTful API criteria of caching data that can be reused for later requests.

```typescript
import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { fromUser, UserDto } from 'src/dto/user.dto';
import { UserDocument } from 'src/schemas/user.schema';
import { UserService } from '../user/user.service';
import * as bcrypt from 'bcrypt';
import { AuthTokenPayload } from './jwt.strategy';

@Injectable()
export class AuthService {
  constructor(
    private readonly userservice: UserService,
    private readonly jwtService: JwtService,
  ) {}

  /**
   * Validates the user against their password hash
   * @param {string} username The username
   * @param {string} password The password hash
   * @returns The user if validated, otherwise returns null
   */
  async validateUser(username: string, password: string): Promise<UserDto> {
    //Get a user by username
    const user = await this.userservice.getByUsername(username);
    if (!user) {
      throw 'User does not exist';
    }
    //Hash the passed through password and compare it to the user's stored hashed password
    if (await bcrypt.compare(password, user.passwordHash)) {
      const result = fromUser(user);
      return result;
    }
    return null;
  }

  /**
   * Generate a JWT token based on a user object
   * @param {UserDocument} user The user document
   * @returns An object containing the access token
   */
  async login(user: UserDocument) {
    //Create a payload for the token of the user's username, id, and role
    const payload: AuthTokenPayload = {
      username: user.username,
      sub: user.id,
      role: user.role,
    };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}

```

Authorisation was implemented using a roles decorator and a RolesGuard class. Endpoints in the API were decorated with the roles decorator, allowing any route handler to be

marked as only usable by clients who have one of the listed roles. When a decorated endpoint is used the RolesGuard class is called. This class determines whether the user can execute the decorated role based on their assigned role.

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { HttpArgumentsHost } from '@nestjs/common/interfaces';
import { Reflector } from '@nestjs/core';
import { JwtService } from '@nestjs/jwt';
import { ExtractJwt } from 'passport-jwt';
import { Role, ROLES_KEY } from 'src/roles';
import { Request } from 'express';
import { AuthTokenPayload } from './jwt.strategy';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(
    private reflector: Reflector,
    private readonly jwtService: JwtService,
  ) {}

  canActivate(context: ExecutionContext): boolean {
    const httpctx: HttpArgumentsHost = context.switchToHttp();
    const request: Request = httpctx.getRequest();
    const token: string = ExtractJwt.fromAuthHeaderAsBearerToken()(request);
    const payload: AuthTokenPayload = this.jwtService.verify(token, {
      ignoreExpiration: true,
    });
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }

    return requiredRoles.some((role) => payload.role === role);
  }
}

```

```
71   @Patch('me')
72   @Roles(Role.Admin, Role.User)
73   @UseGuards(AuthGuard(JWT_STRATEGY), RolesGuard)
74   async updateSelf(
75     @Body() body: UserBaseDto,
76     @Request() request,
77     @Res() response: Response,
78   ) {
79     await this.userService.editUser(request.user.userId, body);
80     response.status(HttpStatus.OK).send();
81   }
82
83   @Patch(':userID')
84   @Roles(Role.Admin)
85   @UseGuards(AuthGuard(JWT_STRATEGY), RolesGuard)
86   async updateUser(
87     @Body() body: UserBaseDto,
88     @Param('userID') userID: string,
89     @Res() response: Response,
90   ) {
91     await this.userService.editUser(userID, body);
92     response.status(HttpStatus.OK).send();
93   }
```

The final work on the backend involved encrypting the user's passwords. The password service was updated with three functions which four functions which handle encryption and decryption, and the function used to create or retrieve a password for a user was updated to use these functions.

The first function to encrypt a password takes a plaintext password and encrypts it using AES. Firstly, 16 random bytes are generated and saved as the initialisation vector before a key is created using the .env variable's secret key and a random salt. The initialisation vector and the key are used together to create a cipher using AES which is then applied to the plaintext password. This encrypted password and the initialisation value are then stored as a base64 string in the password record. The initialisation value acts as another random layer of security and is never sent to the frontend, only being used on the server to encrypt or decrypt the password. If there is an error in this process the plaintext password is returned with an initialisation vector of null and an error message is displayed on the console.

```
121   /**
122    * Encrypt a password
123    * @param unencryptedPassword Plaintext password to be encrypted
124    * @returns Encrypted password and IV
125    */
126   private async encrypt(unencryptedPassword: string) {
127     try {
128       const ivBytes = randomBytes(16);
129       const key = await this.createKey();
130       const cipher = createCipheriv('aes-256-ctr', key, ivBytes);
131       const encryptedText = Buffer.concat([
132         cipher.update(unencryptedPassword),
133         cipher.final(),
134       ]);
135       const password = encryptedText.toString('base64');
136       const iv = ivBytes.toString('base64');
137       return { password, iv };
138     } catch {
139       console.error("Can't encrypt password");
140       return { password: unencryptedPassword, iv: null };
141     }
142   }
```

The second function decrypts passwords for GET requests, and works in much the same manner but reversed. Firstly, if the stored initialisation vector for the record is null, an exception is thrown as the password is already unencrypted and the unencrypted password is sent back. Otherwise, the initialisation vector is obtained from the record and converted to a byte array. A key is created using a random salt and the secret key from the .env file. A decipher is then created using the key and the initialisation vector and applied to the encrypted password to decrypt it.

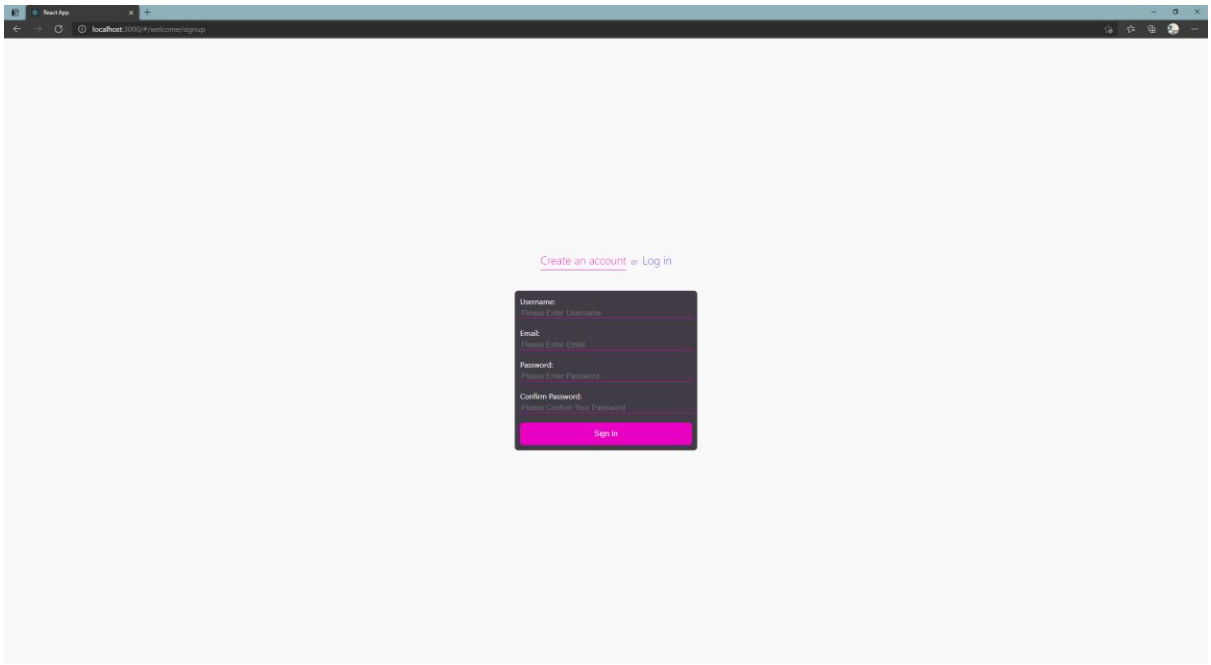
```
144   /**
145    * Decrypt a password
146    * @param encryptedPassword Encrypted password to be decrypted
147    * @param ivString Initialisation variable
148    * @returns Decrypted password
149    */
150   private async decrypt(encryptedPassword: string, ivString: string) {
151     try {
152       if (!ivString) throw 'Not encrypted';
153       const iv = Buffer.from(ivString, 'base64');
154       const key = await this.createKey();
155       const decipher = createDecipheriv('aes-256-ctr', key, iv);
156       const passwordBuffer = Buffer.from(encryptedPassword, 'base64');
157       const decryptedText = Buffer.concat([
158         decipher.update(passwordBuffer),
159         decipher.final(),
160       ]);
161       return decryptedText.toString('utf8');
162     } catch (e) {
163       console.error("Can't decrypt password " + e);
164       return encryptedPassword;
165     }
166   }
```

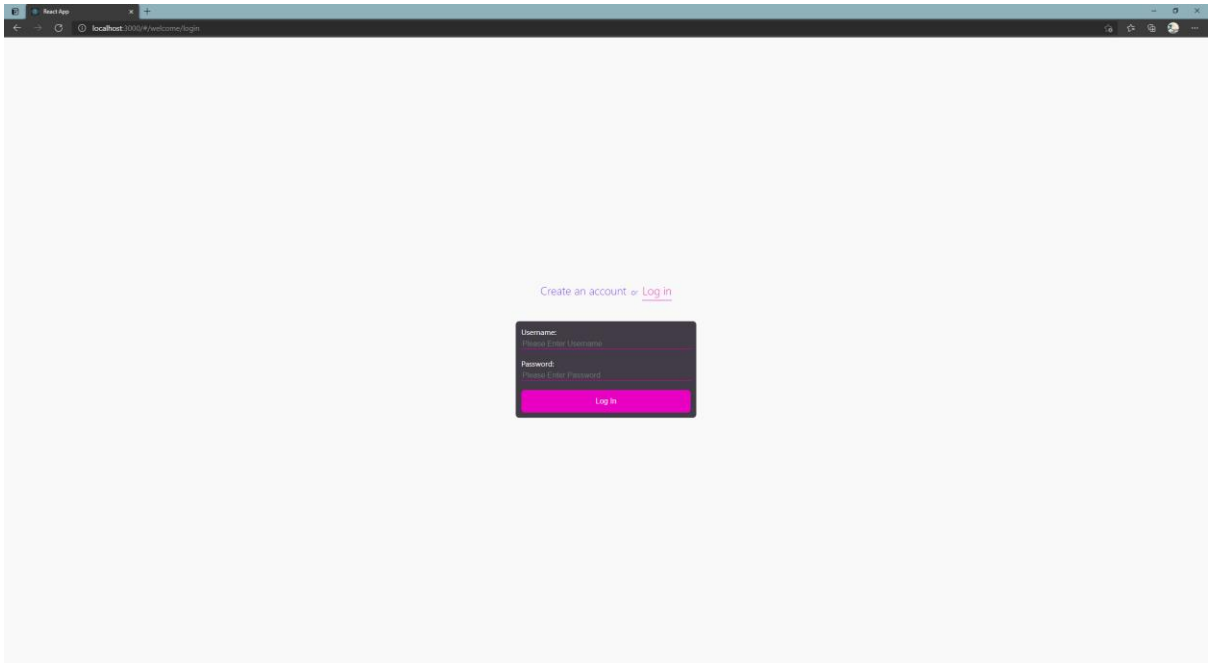
The third function is used to loop through many passwords and decrypt them. It loops through the passed in records and gathers the corresponding encrypted password and initialisation vector before calling the decrypt function. If any errors occur during the decryption of a record that password is returned without being decrypted. Finally, the function returns a promise of all decrypted passwords. This function is shown below, followed by the function which creates a key from the .env value and the random salt.

```
88    /**
89     * Loop through all passwords and decrypt, used to get all records
90     * @param passwords List of encrypted passwords
91     * @returns
92     */
93    private async convertList(
94      passwords: PasswordDocument[],
95    ): Promise<PasswordDto[]> {
96      return Promise.all(
97        passwords.map(
98          (password): Promise<PasswordDto> => {
99            const { iv, password: encryptedPassword } = password;
100           return new Promise((acc, rej) => {
101             try {
102               this.decrypt(encryptedPassword, iv)
103                 .then((decryptedPassword) => {
104                   acc(fromPassword(password, decryptedPassword));
105                 })
106                 .catch((e) => {
107                   return rej(e);
108                 });
109             } catch (e) {
110               // password is not encrypted
111               acc(encryptedPassword);
112             }
113           }).catch((e) => {
114             console.error(e);
115           });
116         },
117       ),
118     );
119   }
```

```
168     private async createKey(): Promise<Buffer> {
169       // The key length is dependent on the algorithm.
170       // In this case for aes256, it is 32 bytes.
171       return (await promisify(scrypt)(
172         process.env.ENCRYPT_PASSWORD,
173         'salt',
174         32,
175       )) as Buffer;
176     }
177   }
178
```

With the backend mostly complete, work shifted to focus mainly on the frontend of the app. Due to the use of the MERN software stack, the frontend was mainly produced through React, JSX, HTML, and CSS. To begin construction and to gain familiarity with frontend development, sign-up and login forms were first created. These forms were created using HTML and CSS, and represent initial experimentation with regards to colours, fonts, and other styles. The forms were initially created on a dark card background on a larger white background. Above the forms was a title which also acted as a React router NavLink component, allowing the user to swap between the sign-up and login forms, with the currently selected form being highlighted in a different colour. The forms themselves consisted of wide labelled textboxes above an equally wide highlighted submit button. The text fields were labelled and contained placeholder text to avoid any potential confusion. The first iteration of these forms can be seen below.





The second iteration of these forms was designed using Material-UI, a component based React library which is an implementation of Google's Material Design. Material Design is a design system created by Google to allow for easier implementation of cohesive, accessible, project-wide theming. While the developer can set some global themes such as fonts and colours, Material-UI aids accessibility and ease of use by calculating whether text or icons on a surface should be black or white to maximise contrast and legibility.

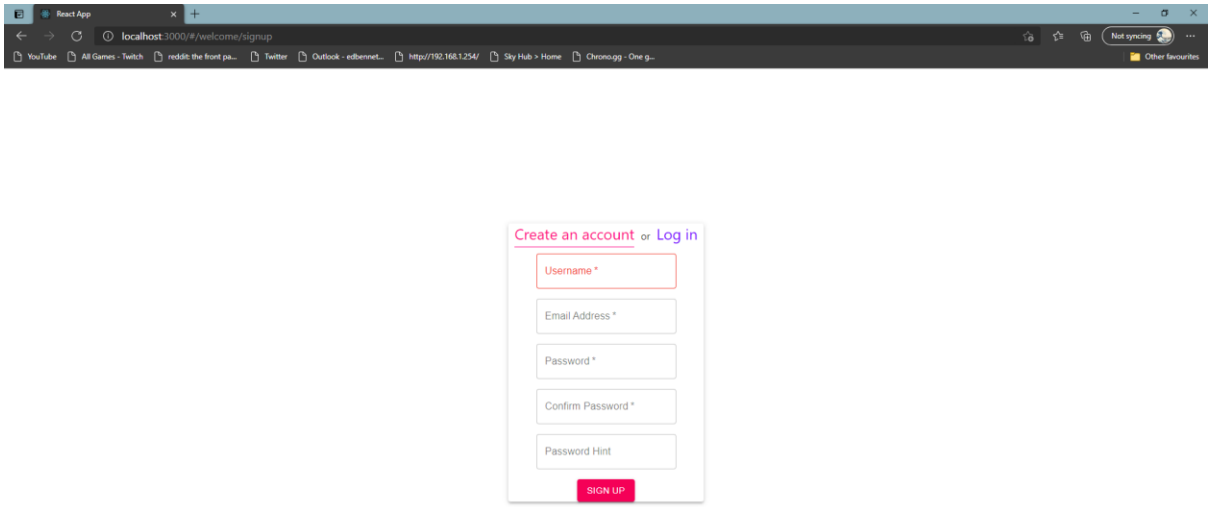
This second iteration of the forms consisted of the same NavLink title with the forms inserted below as components. The sign-up and login components themselves were built using Material-UI's TextField and Button components. Each field was clearly labelled and contained initial placeholder text to make its purpose clear to the user, and required fields are marked with an asterisk. Input validation was then completed, for which a React library called Formik was used, "the world's most popular open source form library for React" (Palmer, n.d.). Formik was used in conjunction with Yup, a Javascript schema builder which Formik documentation recommends. Once a validation schema was constructed with Yup, Formik could then be used to perform input validation and handle submission to the backend. A code snippet from the sign-up component is included below to exemplify this use of Yup and Formik. In the snippet below, every field except the password hint is required; the email field is validated to check the input's format meets that of an email

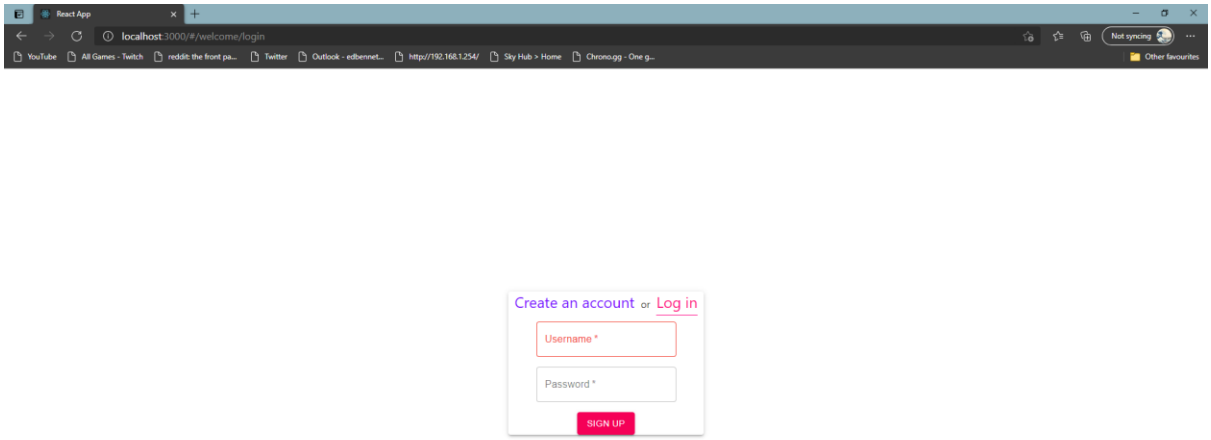
address; the password field must contain at least 8 characters; and the second password field must match the first.

```
const signUp = withFormik({
  mapPropsToValues: ({ username, email, password, confirmPassword, passwordHint }) => {
    return {
      username: username || "",
      email: email || "",
      password: password || "",
      confirmPassword: confirmPassword || "",
      passwordHint: passwordHint || "",
    };
  },

  validationSchema: Yup.object().shape({
    username: Yup.string().required(" "),
    email: Yup.string().email("Invalid email address").required(" "),
    password: Yup.string().min(8, "Password must contain at least 8 characters").required(" "),
    confirmPassword: Yup.string()
      .required(" ")
      .oneOf([Yup.ref("password")], "Passwords do not match"),
  }),

  handleSubmit: (values, { setSubmitting }) => {
    setTimeout(() => {
      // submit to the server
      setSubmitting(false);
    }, 1000);
  },
})(form);

export default signUp;

```

Input validation was performed for two reasons. Firstly, it helps ensure that no requests will be made to the server which may cause errors, such as attempting to sign up for an account without a password. Secondly, input validation helps users who sign up for an account ensure that they know their password, both because forcing the user to slow down and repeat their password helps cement the password in their short term memory, and because it reduces the chance the user submitted a password with a mistake.

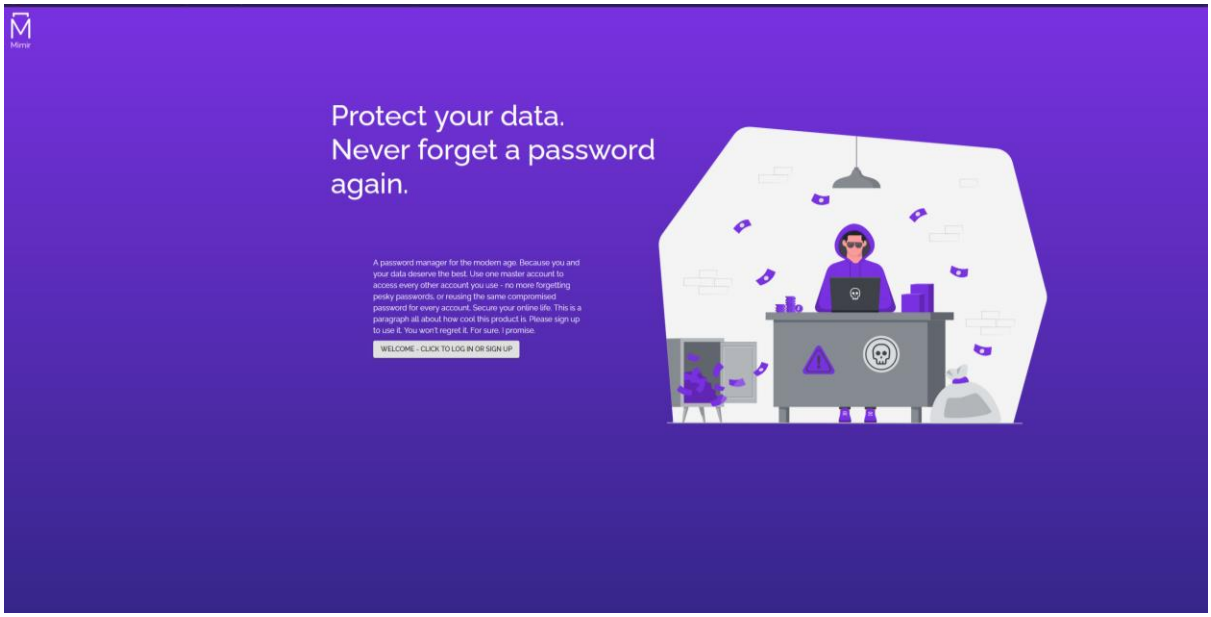
Material-UI's TextFields come with properties to mark whether an input causes an error, and this was used in conjunction with Formik so when a field has been touched without leaving a valid input the field changes colour to red to indicate to the user that there is a problem with their form values. Where applicable, helper text also appears under the field to elaborate on the error for the user. This was used under the email and password forms to indicate that the user's email address was invalid, or their password was too short. The second iteration of the sign-up and login forms are shown in the two screenshots below.



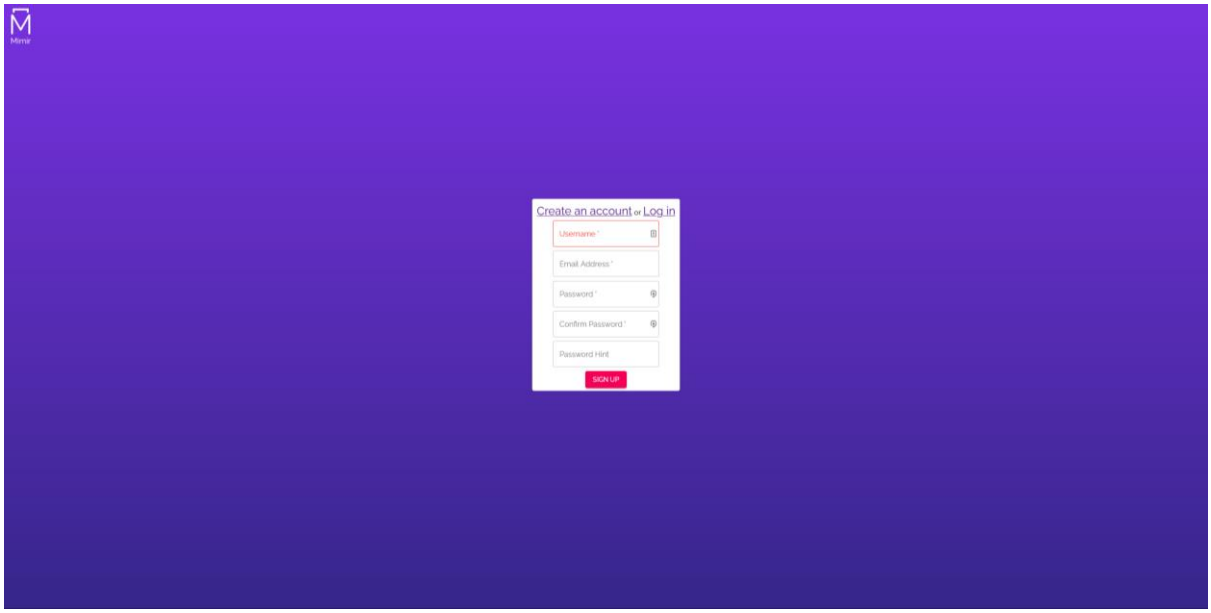


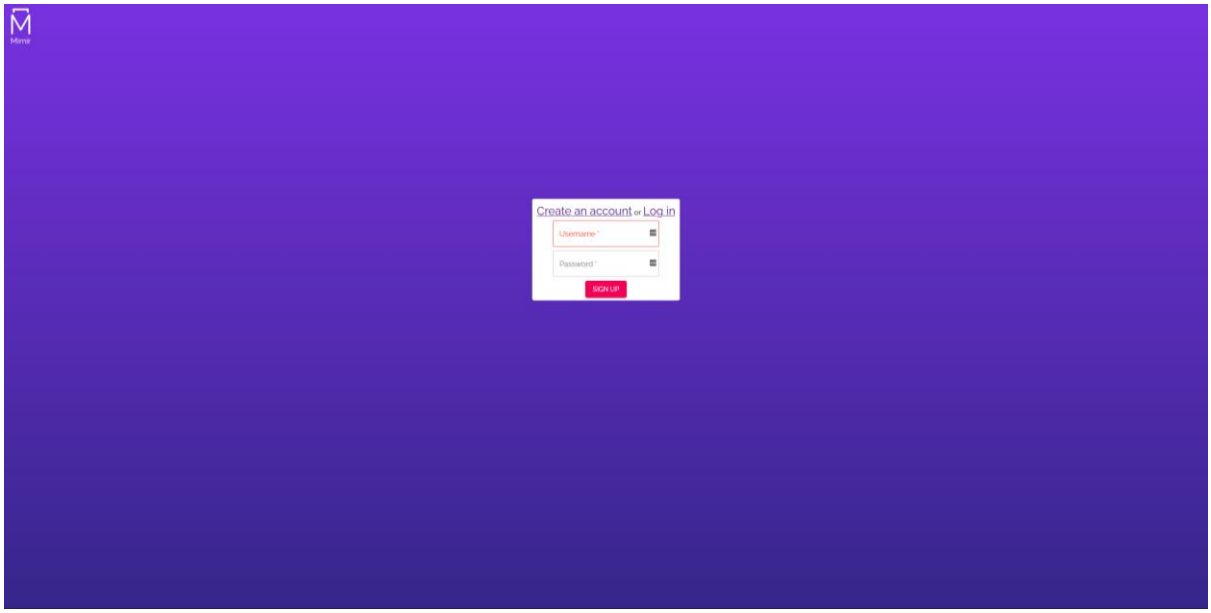
With the above forms brought to a relatively functional state, work began on the front page of the website. The front page of the website is used to provide to the user some context about the password manager, a brief explanation of a password manager's purpose, and offer them a way to sign up. The front page was designed as a simple hero section, with large white title text which stands out against a strong purple gradient background. This text consists of two short sentences designed to capture the user's interest. Below this title text an image and a paragraph are placed either side of the centre of the window. The image is a stock image used to represent cyber-crime and hackers who may try to steal user's data,

while the paragraph is intended to give additional context and persuade the user to sign up using the button placed below. A simple logo was designed using the Raleway font – the same font which has been used across the front page – and placed in the top right corner. The front page is shown below.



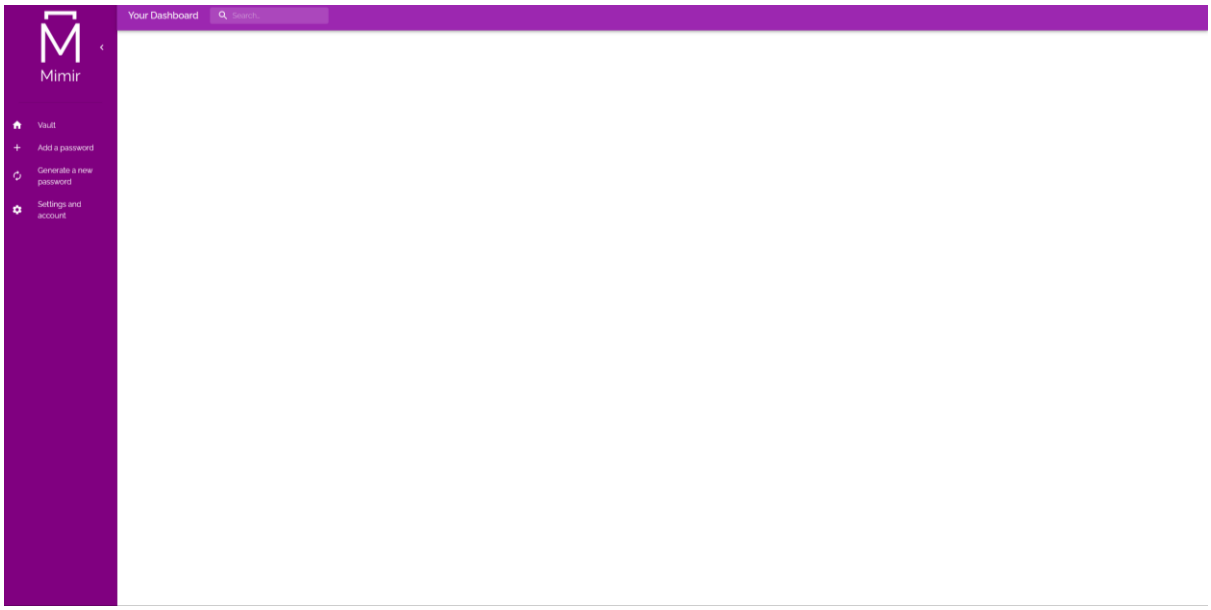
With the front page created, the next step was to update the sign-up and login forms to homogenise the two pages. The same gradient background was placed behind the form's paper background, and the project logo was placed in the same location in the top left corner, where it acts as a link back to the front page. This next iteration of the sign-up and login forms can be seen below.

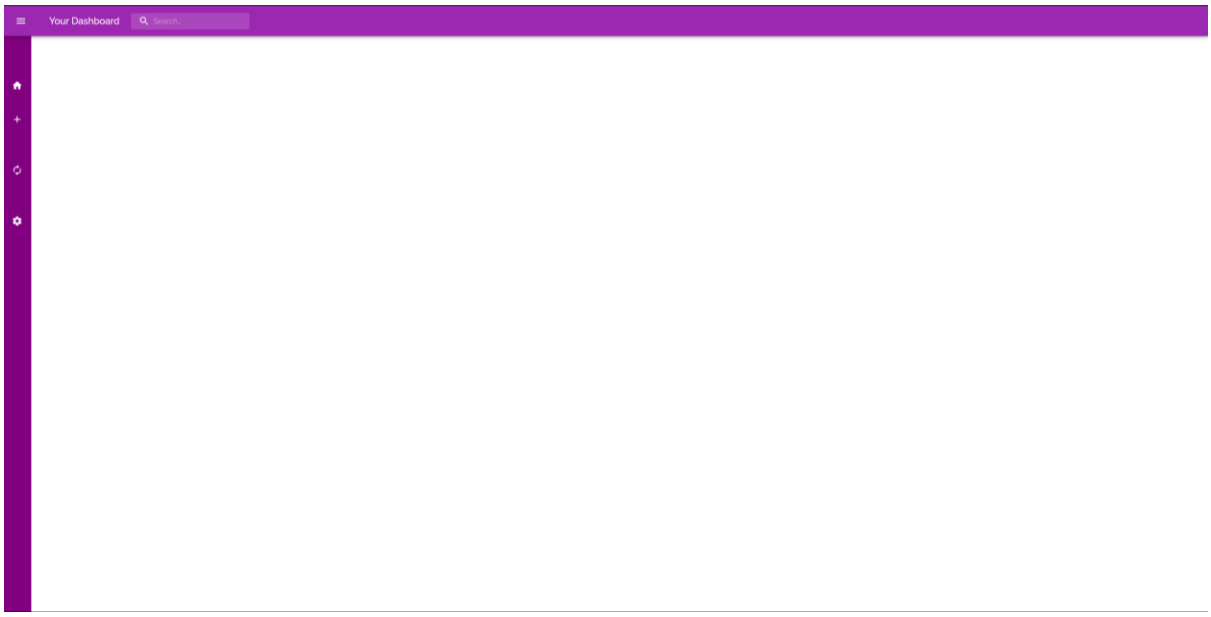




The final page of the web app was the dashboard. This would be the main user experience and would be the location the user interacts with most. Following the Material-UI documentation, the basic interface for a dashboard was designed with a large blank white space for various components to be placed. A purple AppBar was placed at the top of the window, with decorative typography and a search bar intended for searching the user's vault. A minimizable sidebar was placed on the left-hand side, containing the project logo, a button to minimise the sidebar, and a list of links to different components of the dashboard. This dashboard layout was inspired by the existing products researched, primarily Keeper,

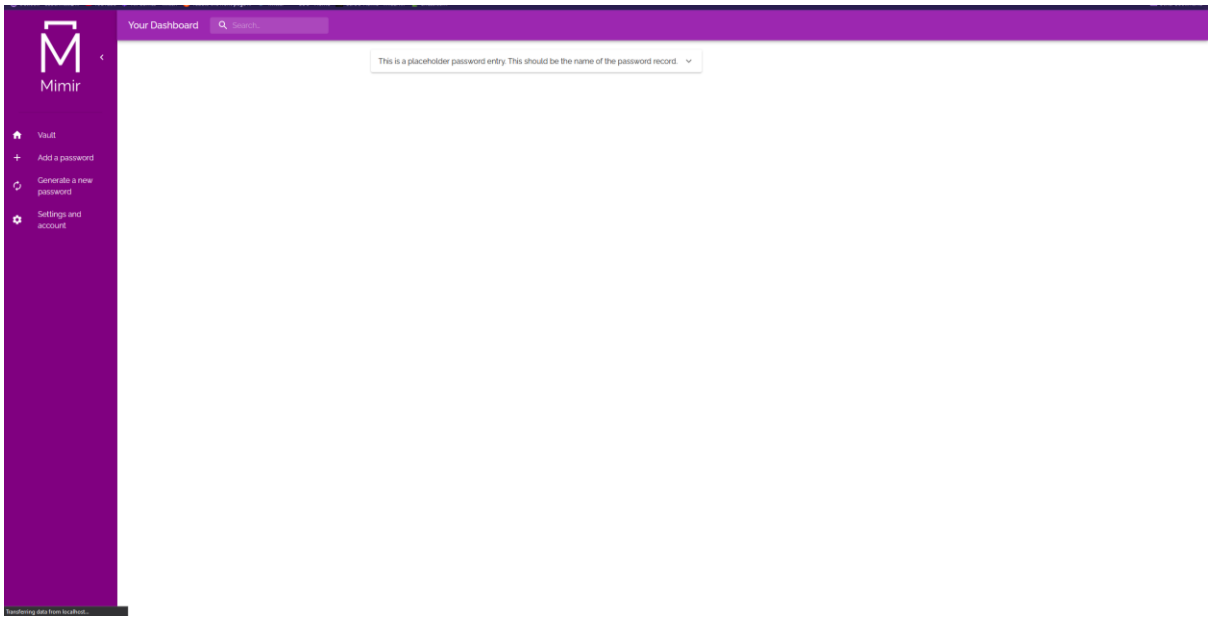
Dashlane, and LastPass. The design of these password managers was clear, presenting important information and navigation options to the user without overloading them with information. The sidebars provide a simple and obvious mechanism for navigating the dashboard, while the top search bar always remains visible. The basic interface of the dashboard is shown below, with the sidebar at full size and minimised.

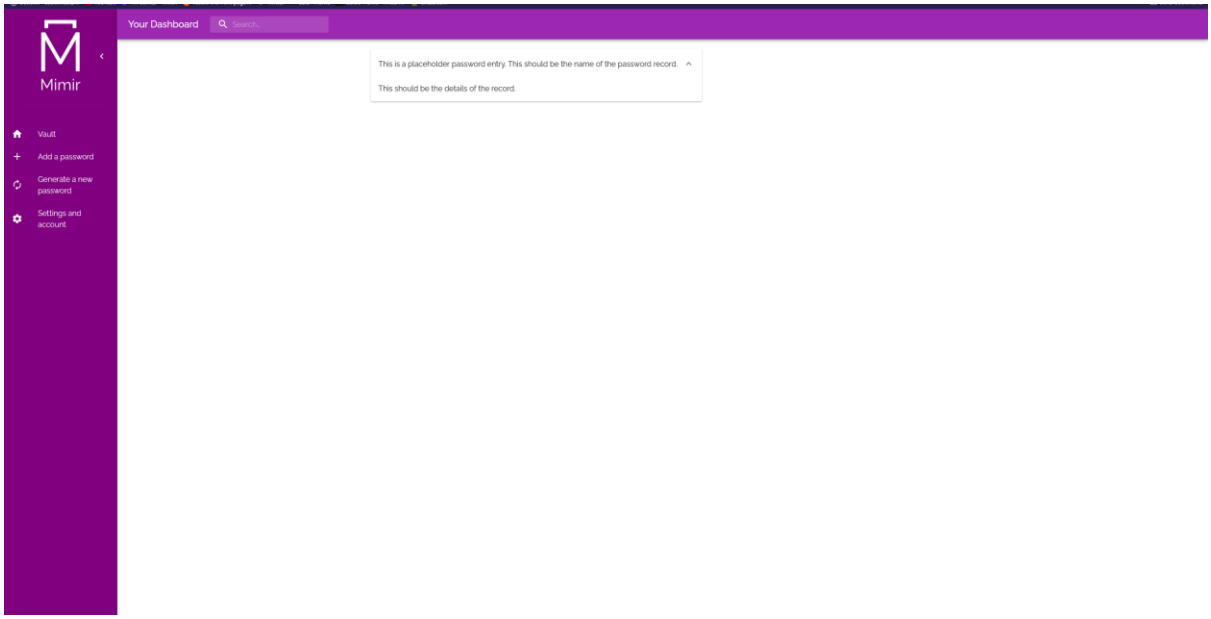




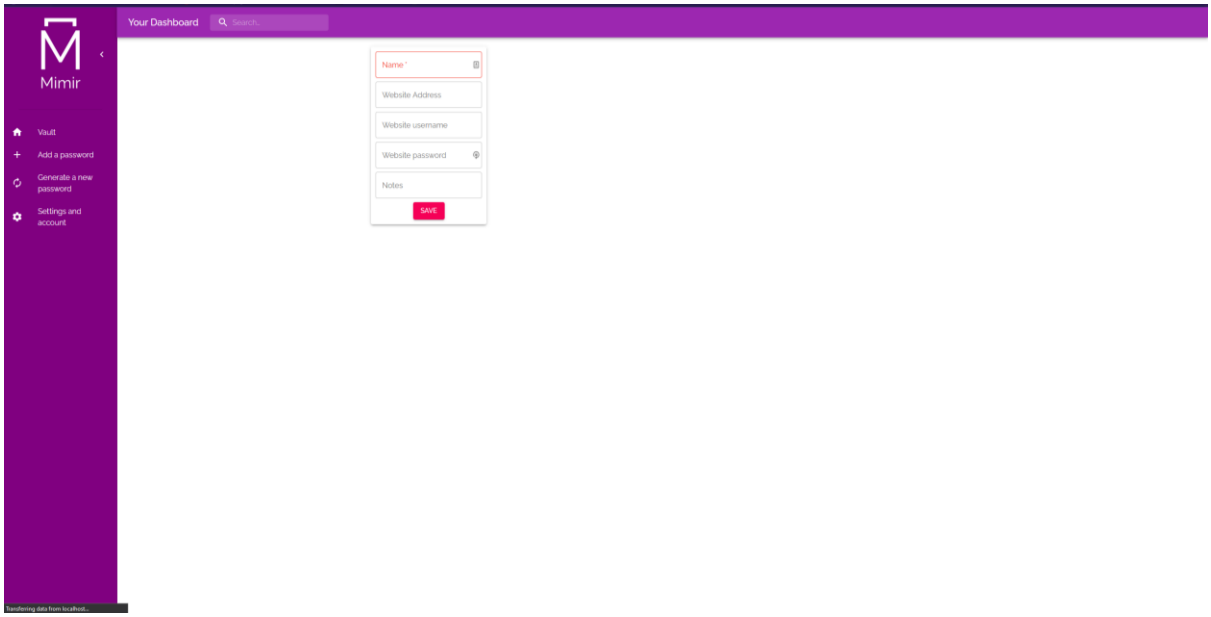
With the layout of the dashboard designed, the next step was to design the dashboard components. Following the structure of the researched existing products, I identified four components: the user's vault showing their password records, a component to create and save a new record, a component to generate passwords, and an account settings page for the user to update their account details.

The first component was the vault, and it was designed using Material-UI's Accordion component. Records would be displayed as entries in the accordion, with the record's name being used as the entry's summary. If the user were to click on an entry it would expand showing the rest of the record's fields such as the URL and password in a password form. An initial implementation of such an accordion is shown below, though this early implementation is not functional and uses hardcoded placeholder text.



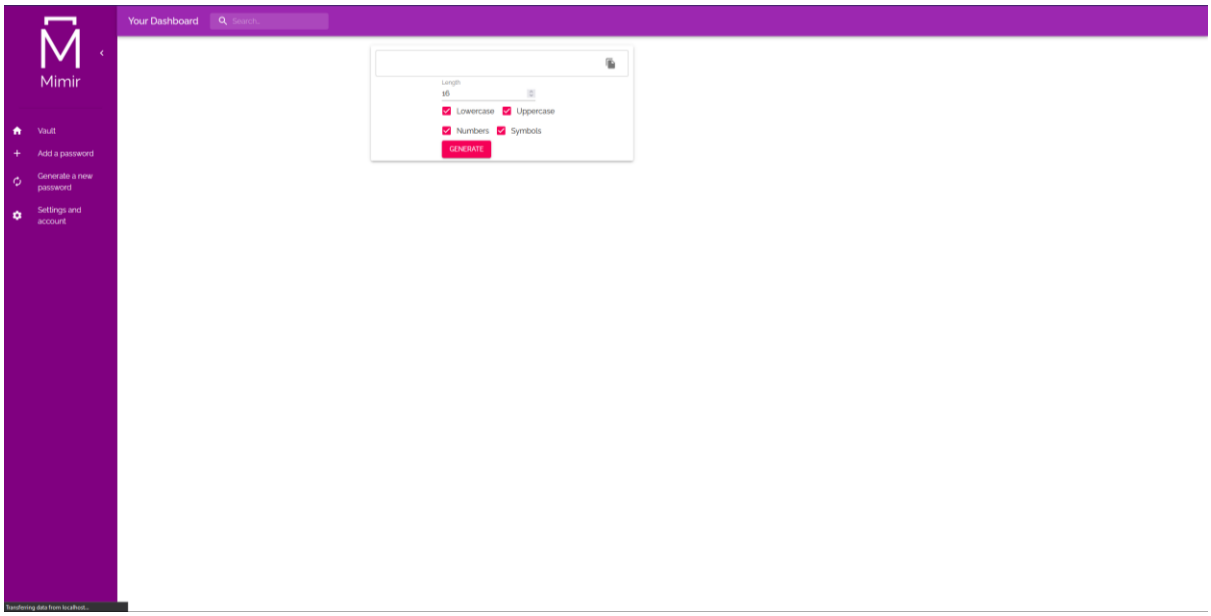


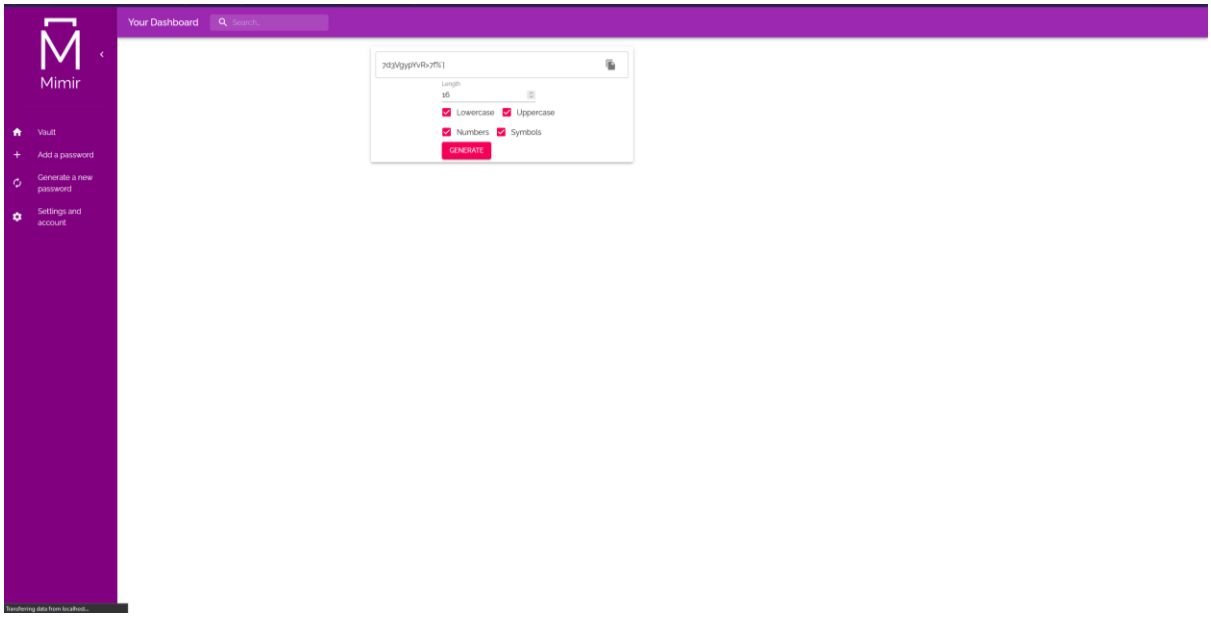
The second dashboard component, the record creation component, was designed as a new form. In keeping with the design of the sign-up and login forms, the new record form was designed as a column of TextField components followed by a highlighted save button all placed on a Material-UI Paper surface designed to give the appearance of elevation from the background and draw the user's focus. Formik was again used to validate the user's input, ensuring that the user entered a name for the record.

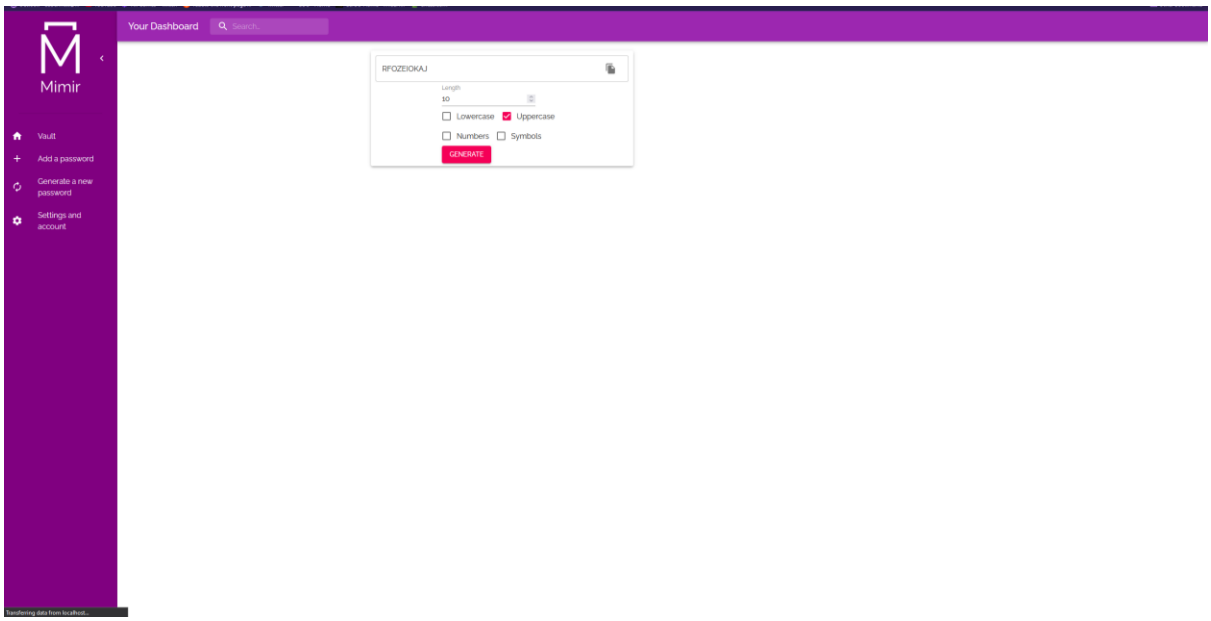


The next dashboard component was the password generator. The password generator was created as two TextFields, four checkboxes, and a button. The first TextField stays empty

until a password has been generated, after which the generated string is placed in the field. The second TextField was set as a numerical input field with a default value of 16. This is used to obtain the length of the password to be generated. The four checkboxes are used to allow the user to select whether uppercase letters, lowercase letters, numbers, and symbols are included in the generated password. Once the user clicks the generate button a Javascript snippet runs which constructs a password using the length and character settings. An icon was placed in the right edge of the password TextField, which copies the generated password to the user's clipboard when clicked. The aim behind this was to make using this generated password slightly more convenient, rather than forcing the user to manually click and highlight the contents of the field. This feature was inspired by the existing password managers examined in the research phase of this project. Example screenshots of the component are shown below, along with the code snippet which handles password generation.

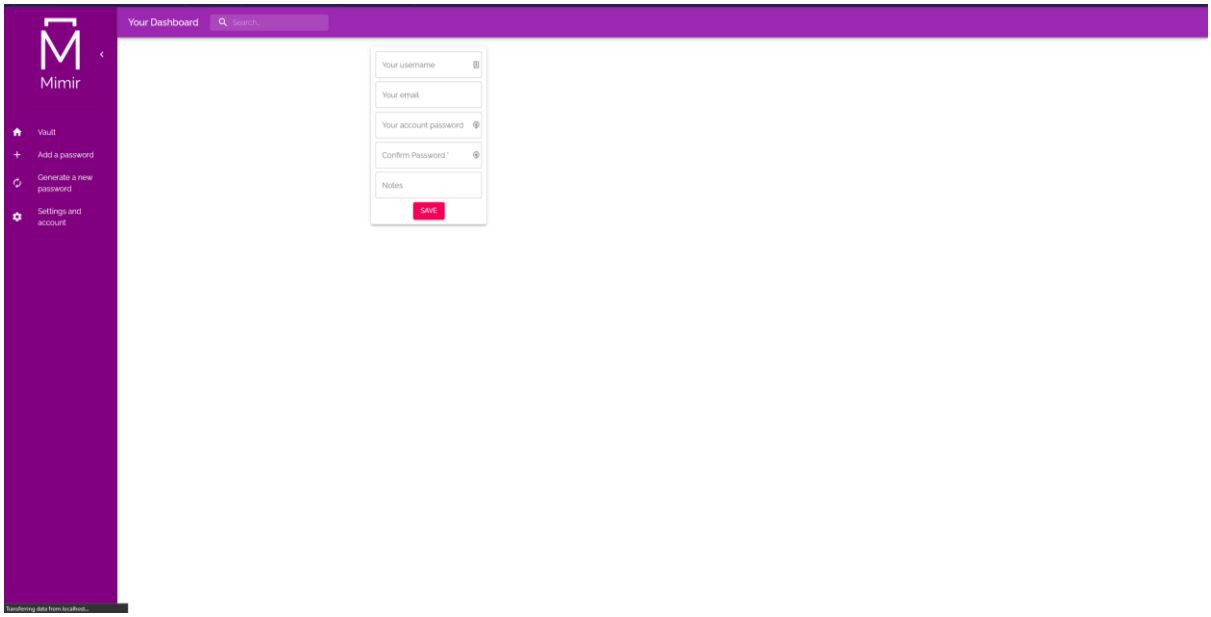



Your Dashboard
Search...
Mimir
Vault
Add a password
Generate a new password
Settings and account
Length
16
Lowercase
Uppercase
Numbers
Symbols
GENERATE


Your Dashboard
Search...
Mimir
Vault
Add a password
Generate a new password
Settings and account
RFOZEIOKAJ
Length
10
Lowercase
Uppercase
Numbers
Symbols
GENERATE

```
const [length, setLength] = useState("16");
const lower = "abcdefghijklmnopqrstuvwxyz";
const upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
const num = "1234567890";
const sym = "!@#$%^&*()_+~`|}{[]:;?><,./-=";

function generate() {
  var charset = "";
  var result = "";
  if (lowercase) {
    charset += lower;
  }
  if (uppercase) {
    charset += upper;
  }
  if (numbers) {
    charset += num;
  }
  if (symbols) {
    charset += sym;
  }
  for (var i = 0, n = charset.length; i < length; i++) {
    result += charset.charAt(Math.floor(Math.random() * n));
  }
  setPassword(result);
}
```

The final dashboard component was the account settings form, which should be used to update the user's own account details. This component was constructed using a modified version of the sign-up form, allowing the user to change any of the account settings they submitted when they created their account. Input validation was again performed using Formik and Yup and required the user to enter their account twice before submitting any changes to the settings fields. This was done to ensure the user was sure about their decision, and to ensure that if they changed their password there was unlikely to be a mistake in the spelling.



Once each component had been designed the frontend was connected to the backend using a library called Axios – “a simple promise based HTTP client for the browser and node.js” (Zabriskie, n.d.). Axios was used to make HTTP requests to the endpoints on the backend so data can be retrieved from the frontend and stored in the MongoDB database, and so data can be retrieved from the database to be displayed in the web client. The first endpoint connected was for the login form, using a POST request to send data to the endpoint created at http://localhost:3100/api/auth/login. The backend then responds with the user’s access token, which is saved into local storage before the user is redirected to their dashboard.

```
14  async function logIn() {
15    const { username, password } = values;
16    const { data, status } = await axios.post("http://localhost:3100/api/auth/login", {
17      username,
18      password,
19    });
20    if (status === 201 && data?.access_token) {
21      localStorage.setItem("access_token", data.access_token);
22      window.location.assign("/#/dashboard/");
23    }
24  }
```

This method was then repeated with the sign-up, create a record, and settings forms. The second step to connecting the client to the backend involved using HTTP GET requests with Axios to retrieve data from the database so that it could be displayed to the user. A user’s own details were obtained in the ChangeSettings component using the /api/users/me

endpoint. The user's bearer token from the cache was included in the request as a header, ensuring that the server could authorise the request.

```
async getUser() {
  const token = localStorage.getItem("access_token");
  try {
    const { status, data: user } = await axios.get("https://localhost:3100/api/users/me", {
      headers: {
        Authorization: `Bearer ${token}`,
      },
    });
    if (status === 200 && user) {
      this.setState({ user });
    }
  } catch (e) {
    console.error(`RETRIEVAL FAILED: ${e}`);
  }
}
```

The second GET request is performed from the vault component, and it retrieves the user's password records and stores them in the component's state.

```
async getPasswords() {
  const token = localStorage.getItem("access_token");
  try {
    const { status, data: passwords } = await axios.get(
      "https://localhost:3100/api/passwords/me/",
      {
        headers: {
          Authorization: `Bearer ${token}`,
        },
      }
    );
    if (status === 200 && passwords) {
      this.setState({ passwords });
    }
  } catch (e) {
    console.error(`RETRIEVAL FAILED: ${e}`);
  }
}
```

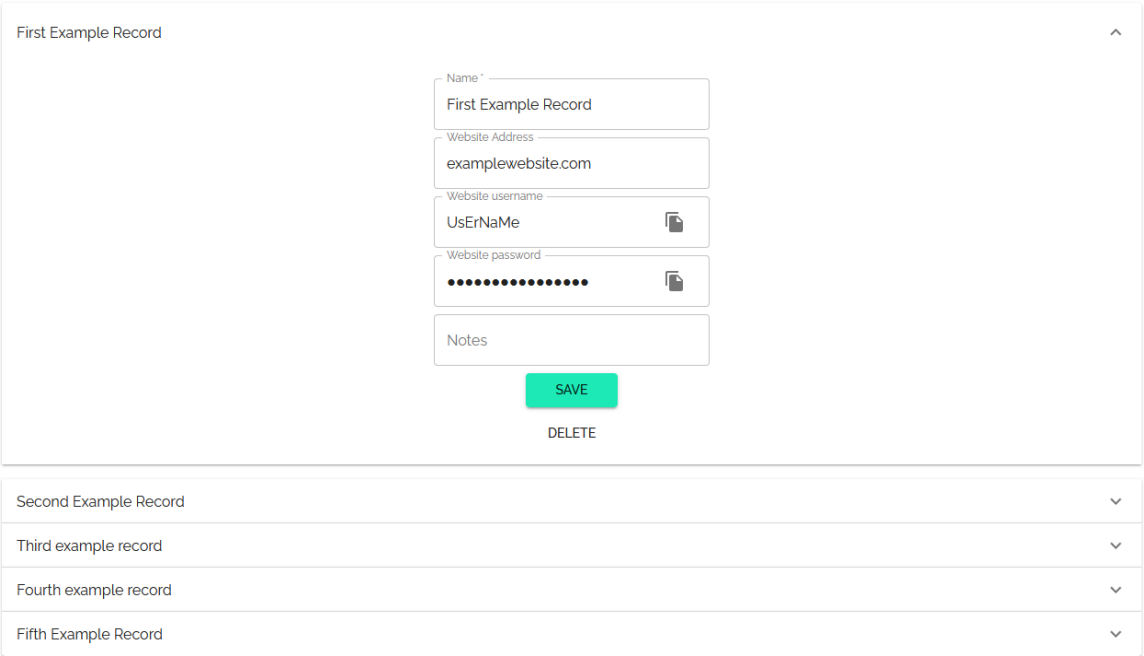
The vault component was then made functional using the PasswordForm from the component used to add a new password. A PasswordForm was used as the body of the Material-UI accordion component, showing the fields of a record. The vault component loops through the user's passwords, creating the accordion and an object called a passwordRecord which contains the current password record's details. This password record was then passed through into the PasswordForm in the accordion's details section, allowing

the fields to be populated with the record's values. Within the PasswordForm component the values are mapped to the form fields or set to empty strings if no data was passed through. A delete button was added to the accordion below the PasswordForm and connected to an endpoint with the password ID used as part of the endpoint so the backend can determine which password record is to be deleted. Icon buttons to copy the values of the username and password fields were then also added to the form using the same method as the password generator.

```
.map((password) => {
  const passwordRecord = {
    id: password?.id || "",
    passwordName: password?.name || "",
    URL: password?.URL || "",
    password: password?.password || "",
    username: password?.username || "",
    notes: password?.notes || "",
  };
  return (
    <Accordion style={{ width: "100%" }} key={password?.id}>
      <AccordionSummary expandIcon={<ExpandMore />}>
        <Typography>{password?.name}</Typography>
      </AccordionSummary>
      <AccordionDetails>
        <Grid container direction="column" alignItems="center">
          <Grid item>
            <PasswordForm passwordRecord={passwordRecord} />
          </Grid>
          <Grid item>
            <Button
              color="error"
              type="button"
              onClick={async () => {
                await this.deletePassword(passwordRecord.id);
              }}
            >
              Delete
            </Button>
          </Grid>
        </Grid>
      </AccordionDetails>
    </Accordion>
  );
})}
```

```
176   const PasswordForm = withFormik({
177     mapPropsToValues: ({ passwordRecord: pr }) => {
178       return {
179         id: pr?.id || "",
180         passwordName: pr?.passwordName || "",
181         URL: pr?.URL || "",
182         password: pr?.password || "",
183         username: pr?.username || "",
184         notes: pr?.notes || "",
185       };
186     },
```

```
43  async deletePassword(id) {
44    const token = localStorage.getItem("access_token");
45    try {
46      const { status } = await axios.delete(`https://localhost:3100/api/passwords/me/${id}`, {
47        headers: {
48          Authorization: `Bearer ${token}`,
49        },
50      });
51    } catch (e) {
52      console.error(`DELETION FAILED: ${e}`);
53    }
54    this.getPasswords();
55    this.render();
56  }
```



With the records successfully being displayed, the next step was to make the search bar functional. This was done by passing a property of the contents of the search bar from the dashboard component to the vault component. A filter function was applied to the

passwords within the vault to only show passwords which contained the string the user had entered into the search bar.

```
61          {this.state.passwords
62            .filter((password) => {
63              return (password?.name || "")
64                .toLowerCase()
65                .includes((this.props?.searchQuery || "").toLowerCase());
66            })
```

```
289             <Route
290               exact
291               path="/dashboard/vault"
292               component={() => <Vault searchQuery={searchQuery} />}
293             />
294             <Route exact path="/dashboard/addpassword" component={AddPassword} />
295             <Route exact path="/dashboard/generate" component={PasswordGenerator} />
296             <Route exact path="/dashboard/settings" component={ChangeSettings} />
```

To prevent a user from accessing the dashboard when they should not, the dashboard component was programmed to check the client's cached JWT token. If the user has a token, the token is compared to the current date and time to determine whether the token is still valid. If the token is no longer valid it is deleted from local storage. After this, if the value for the token was null the user is redirected to the login page as they either never logged in, or their token has expired, and they need to log in again.

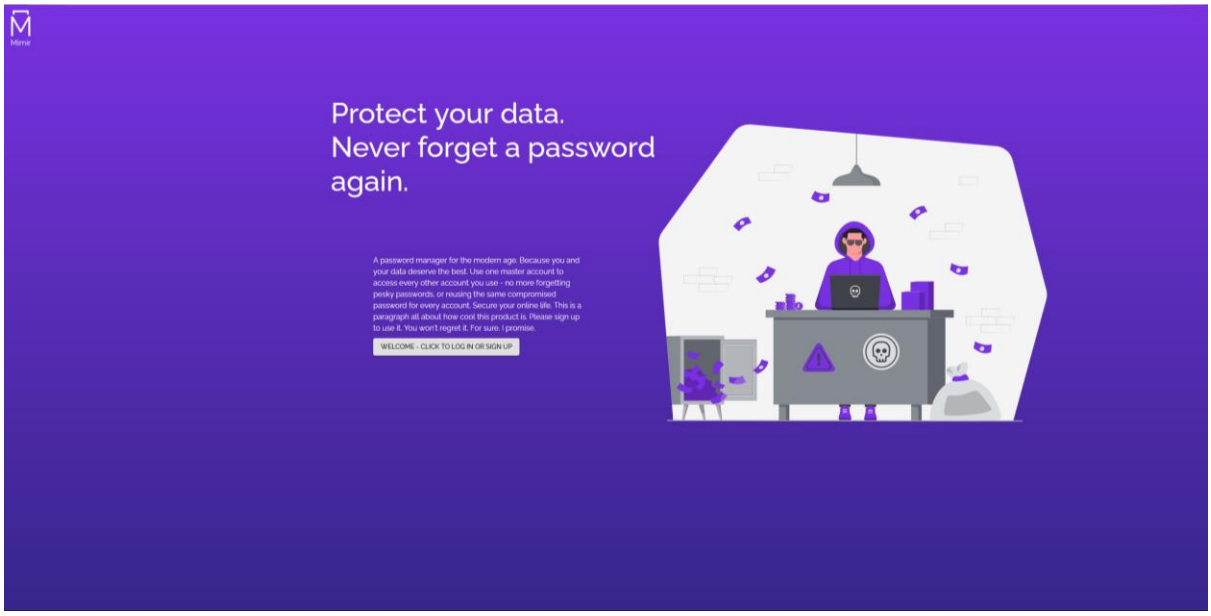
```
156   export default function Dashboard() {
157     let token = localStorage.getItem("access_token");
158     if (token) {
159       const { exp } = JSON.parse(atob(token.split(".")[1]));
160       if (exp * 1000 < Date.now()) {
161         localStorage.removeItem("access_token");
162         token = null;
163       }
164     }
165     if (!token) {
166       window.location.assign("/#/welcome/login");
167     }
```

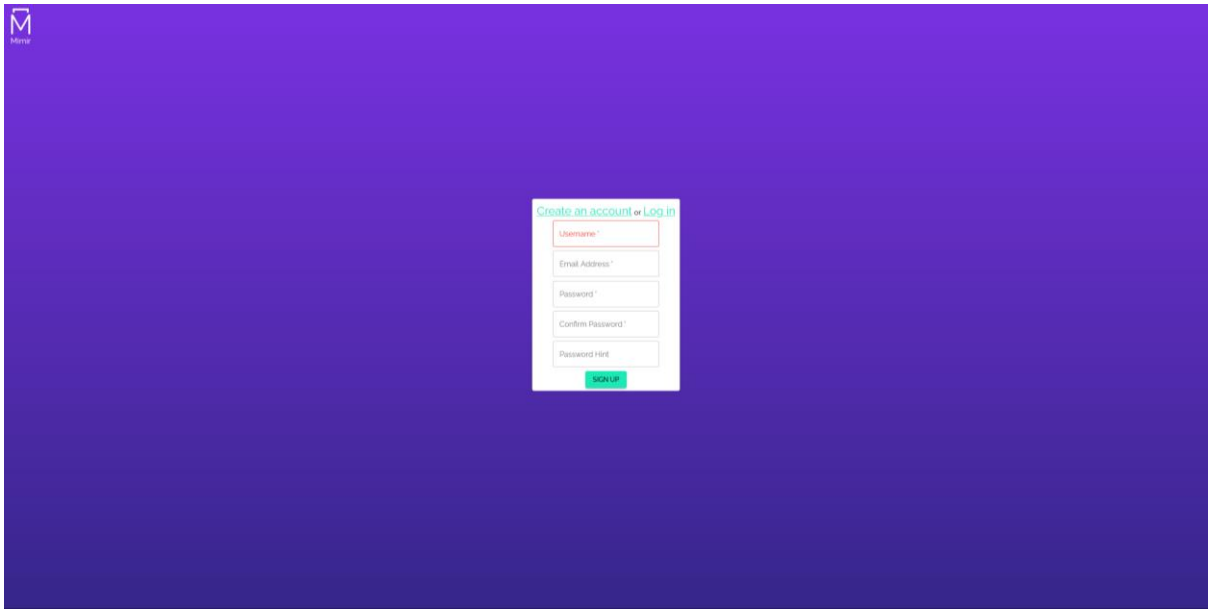
Beyond this point, any changes made to the interface were cosmetic in nature, involving spacing, styling, and colours. Components were set to the upper middle of the open dashboard space, allowing for the best use of negative space and the avoidance of any feelings the user might have that the interface is cramped or claustrophobic. The colour

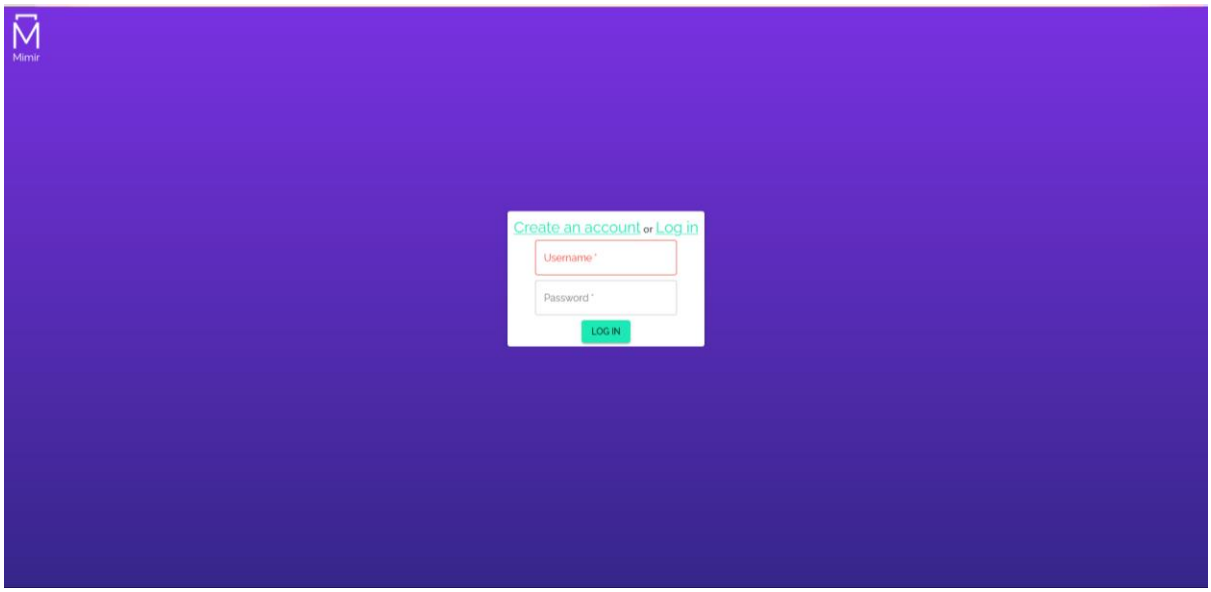
palette of Material-UI was adjusted to better match the front page, and a dark theme was added. If the user's operating system is set to prefer dark mode, the dark mode is automatically enabled. The section of App.js which creates the theme for the Material-UI components and determines whether dark mode should be enabled is shown below.

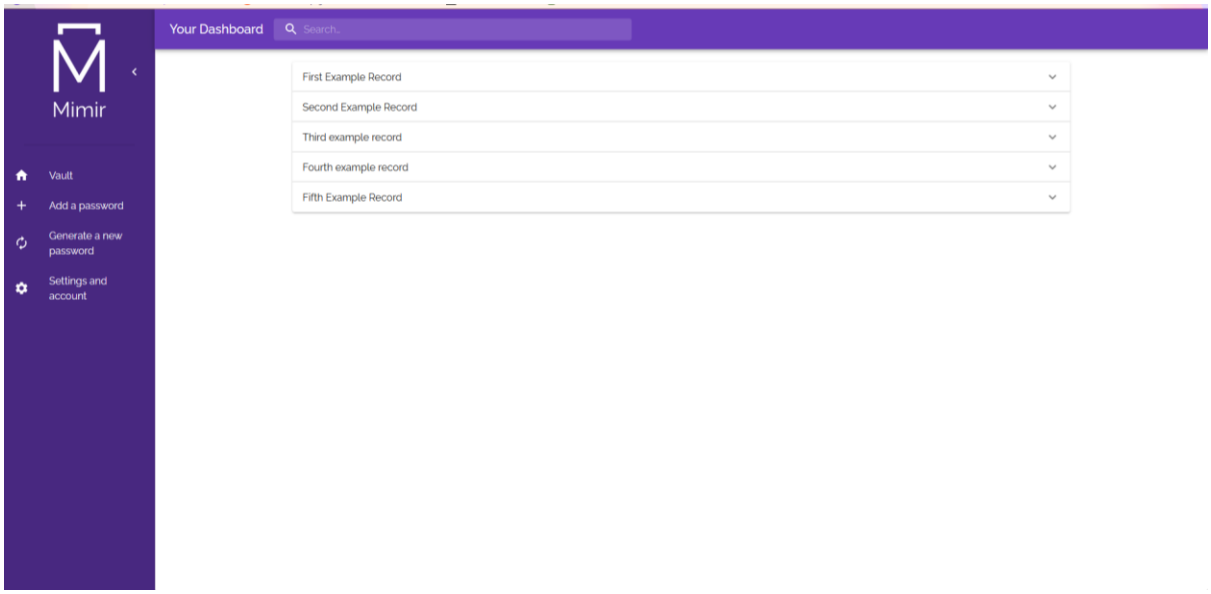
```
function App() {
  const prefersDarkMode = useMediaQuery("(prefers-color-scheme: dark)");
  const theme = React.useMemo(() =>
    createMuiTheme({
      palette: {
        primary: deepPurple,
        secondary: teal,
        type: prefersDarkMode ? "dark" : "light",
      },
      typography: {
        fontFamily: "Raleway",
      },
    })
  );
```

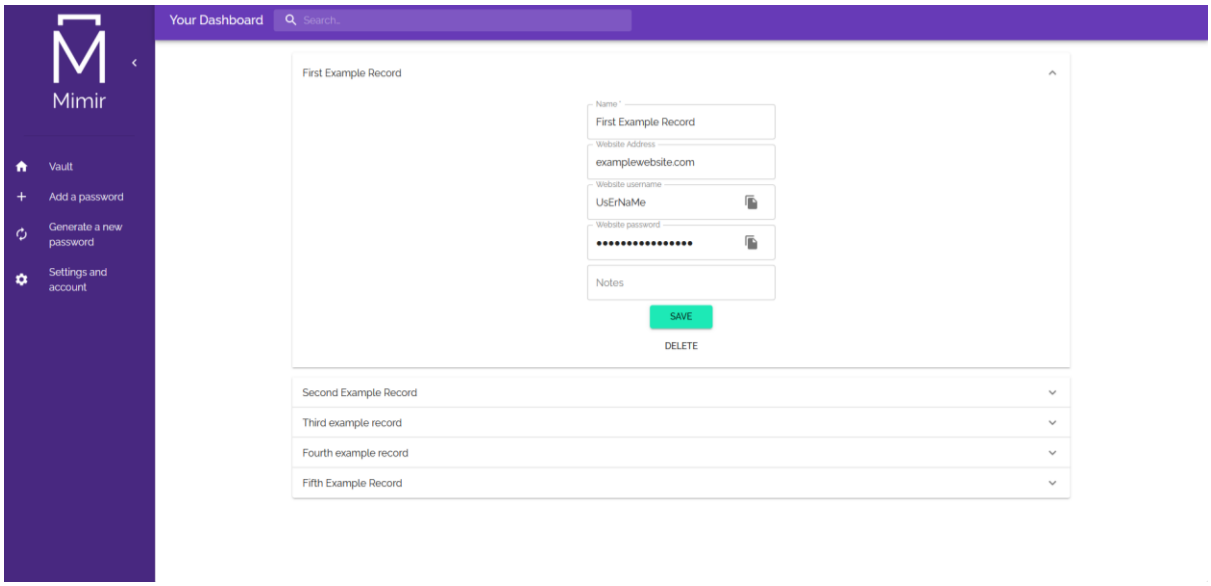
The final iterations of the front end are shown below.

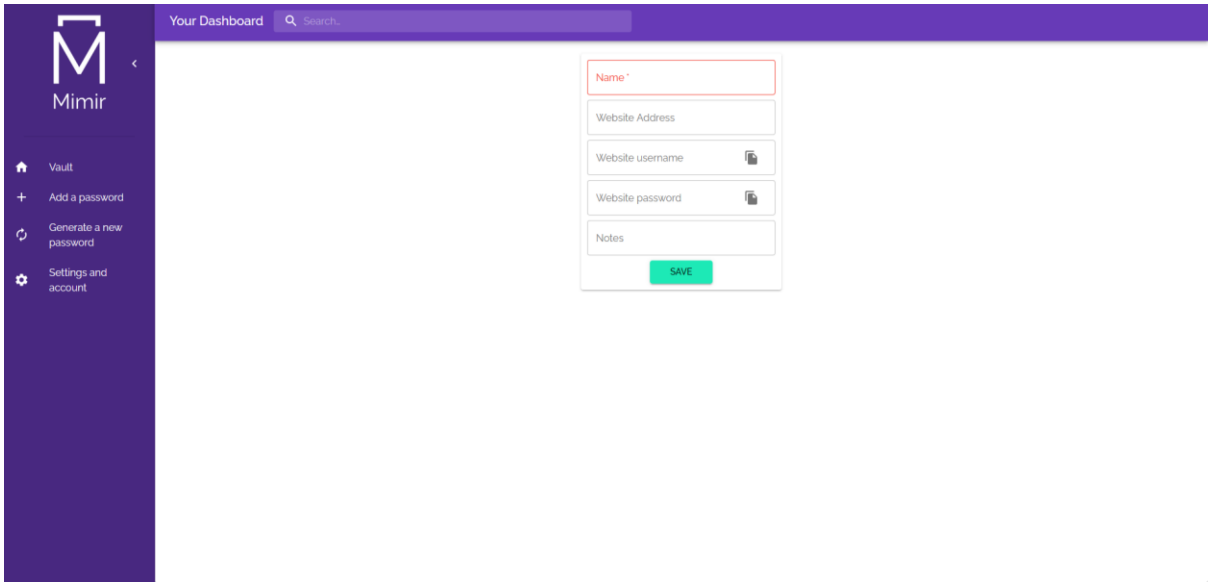


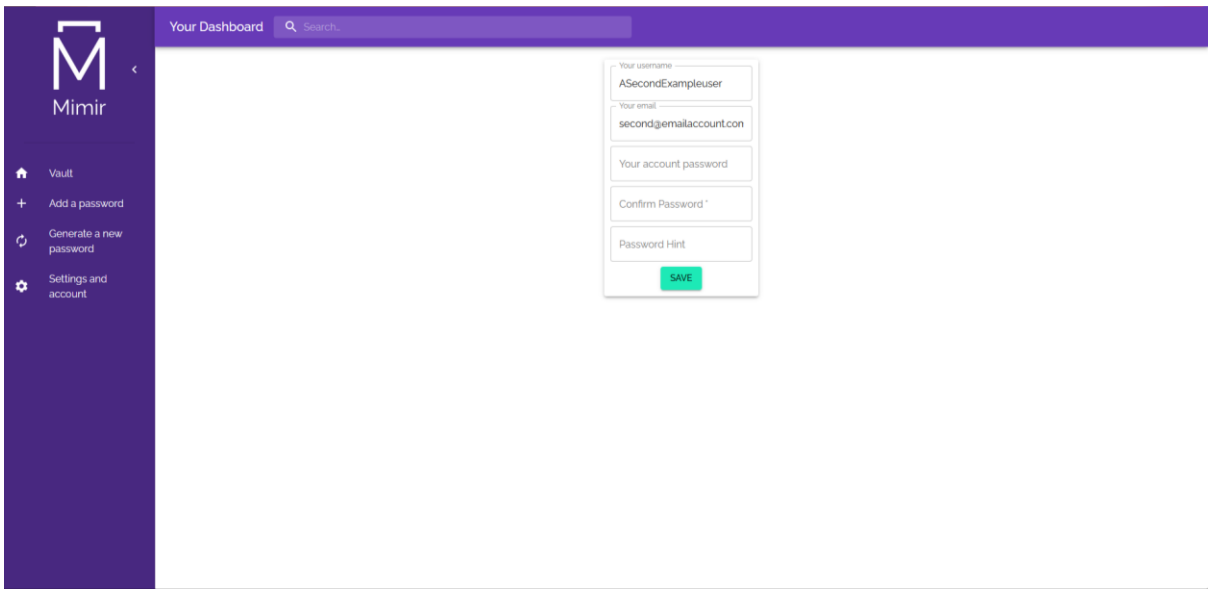

Mimir
Create an account or Log in
Username *
Email Address *
Password *
Confirm Password *
Password Hint
SIGN UP

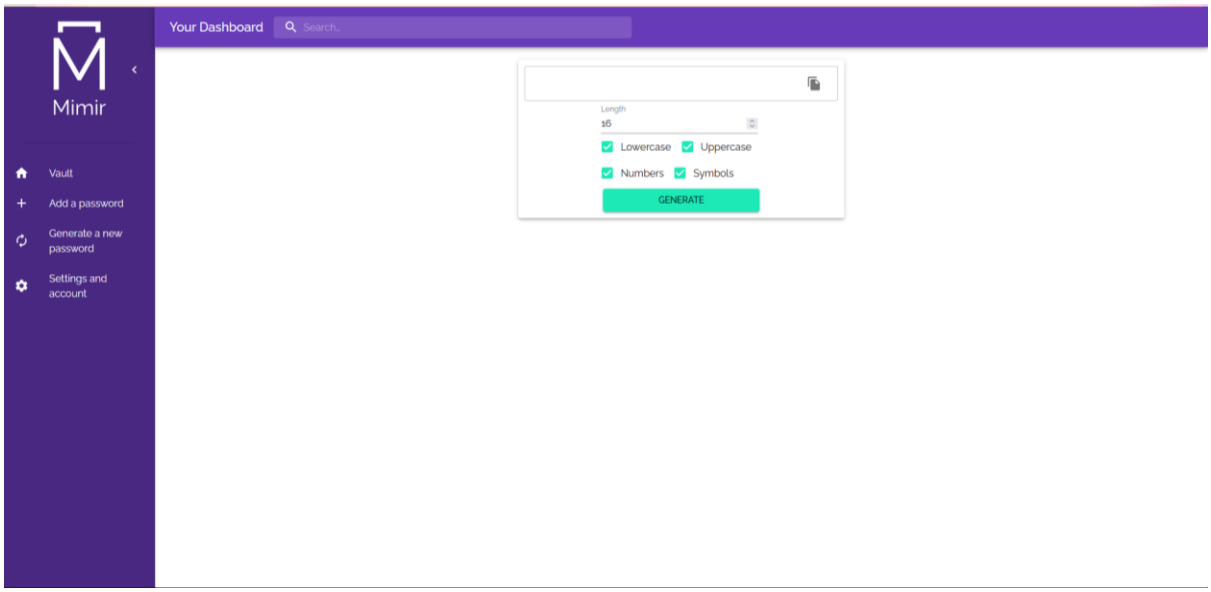

Mimir
Create an account or Log in
Username *
Password *
LOG IN

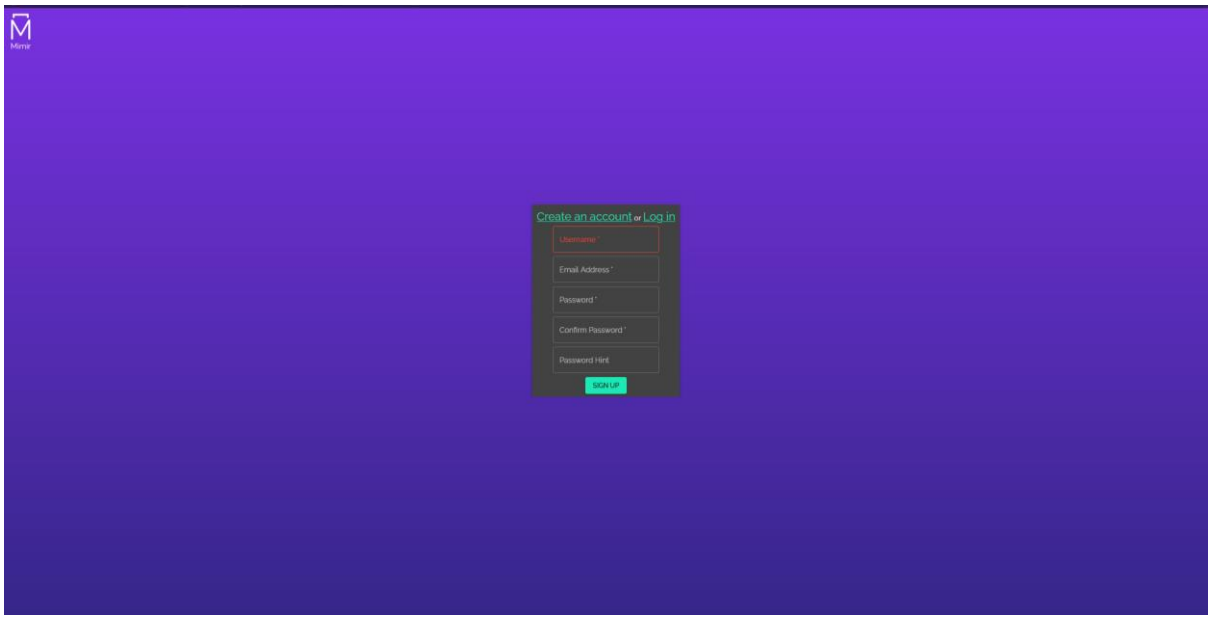

Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
First Example Record
Second Example Record
Third example record
Fourth example record
Fifth Example Record

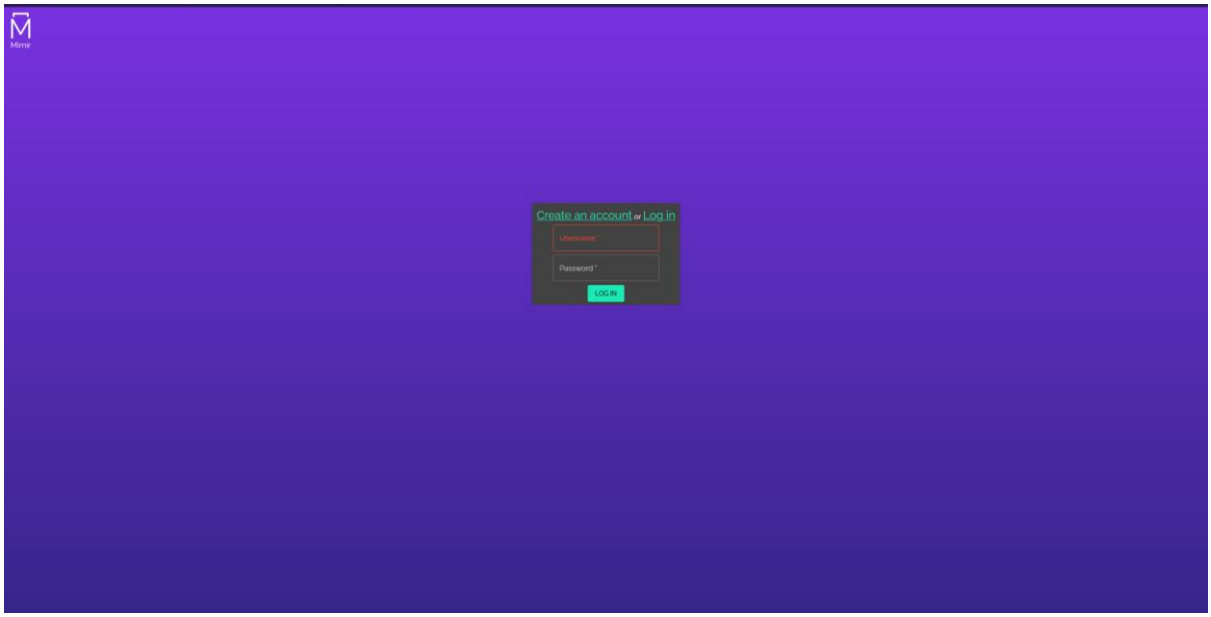

Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
First Example Record
Name *
First Example Record
Website Address
examplewebsite.com
Website username
UsErNaMe
Website password
Notes
SAVE
DELETE
Second Example Record
Third example record
Fourth example record
Fifth Example Record

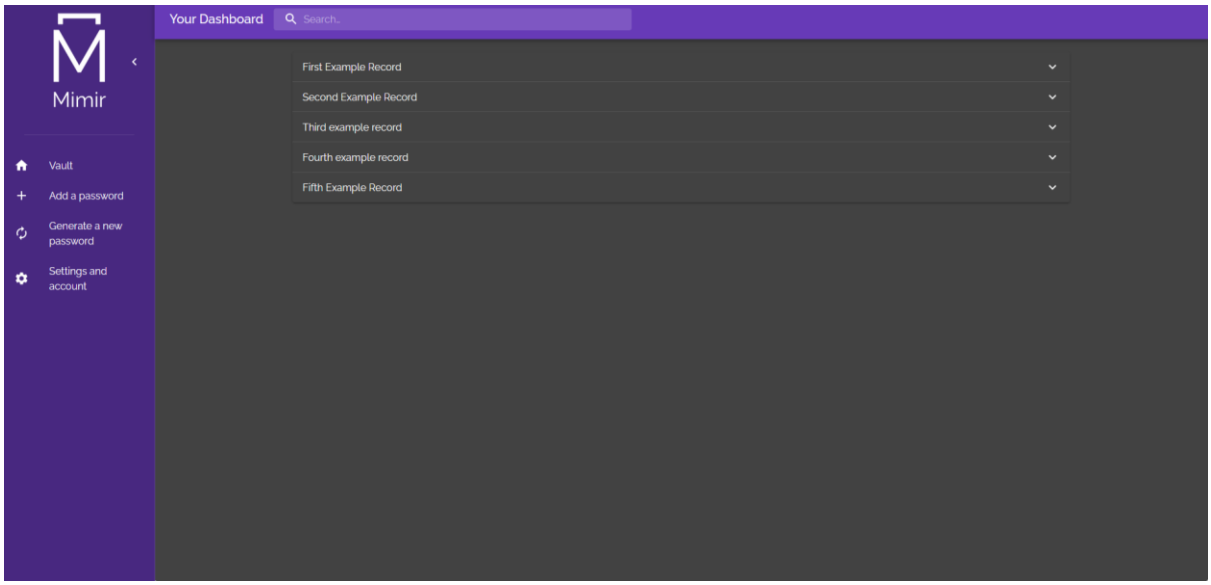

Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
Name *
Website Address
Website username
Website password
Notes
SAVE

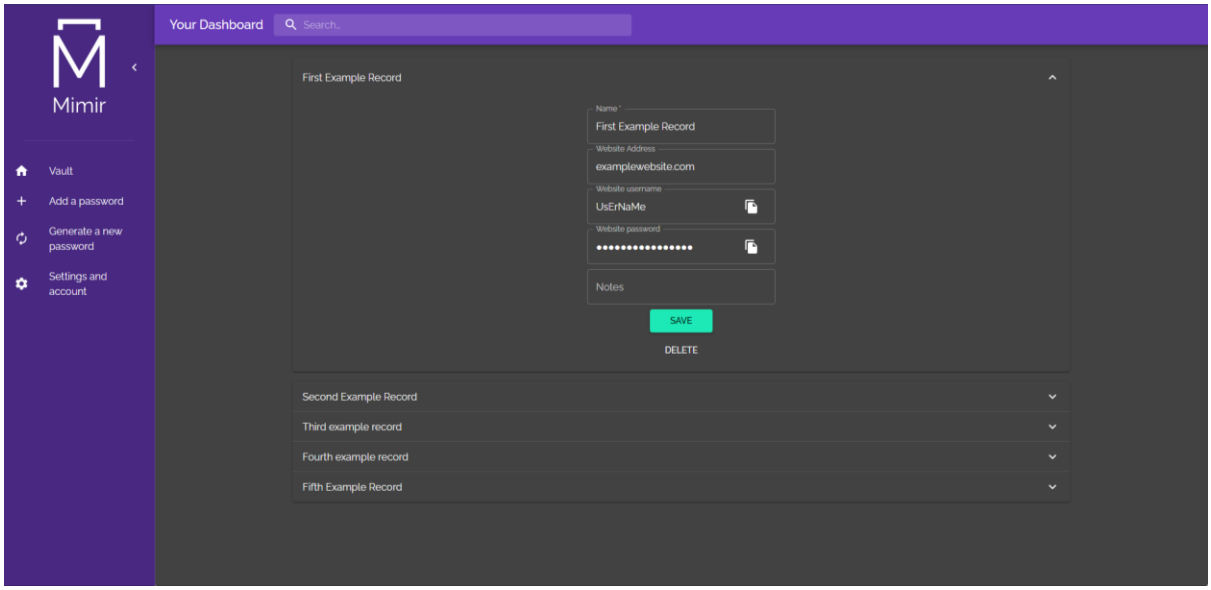

Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
Your username
ASecondExampleuser
Your email
second@emailaccount.con
Your account password
Confirm Password *
Password Hint
SAVE

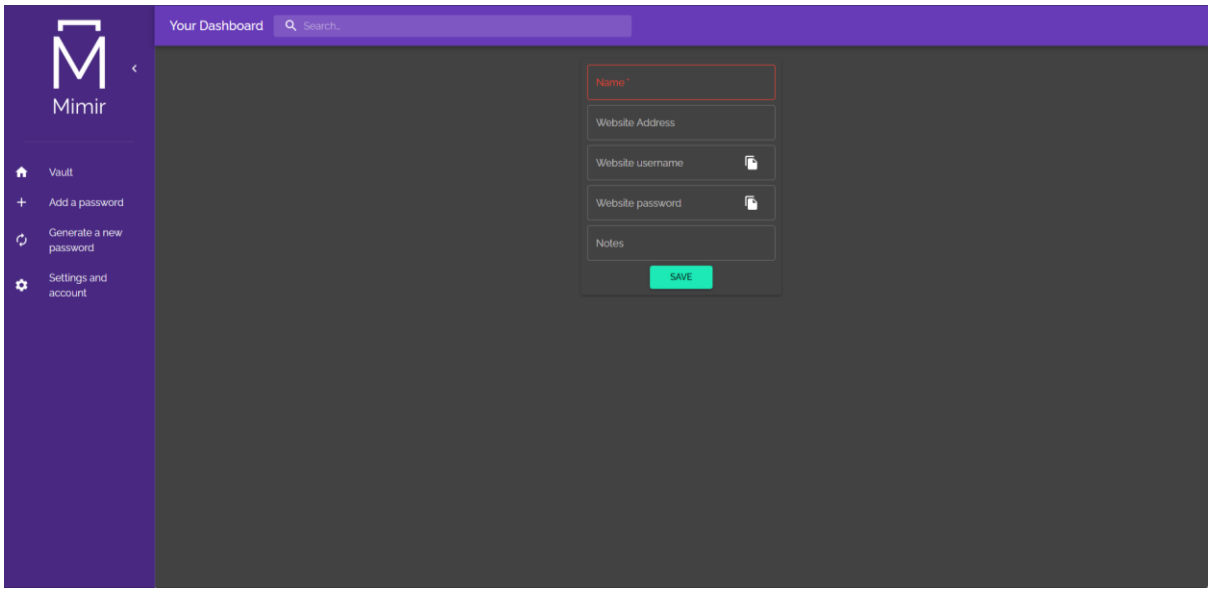

Mimir
Your Dashboard
Search..
Vault
Add a password
Generate a new password
Settings and account
Length
16
Lowercase
Uppercase
Numbers
Symbols
GENERATE

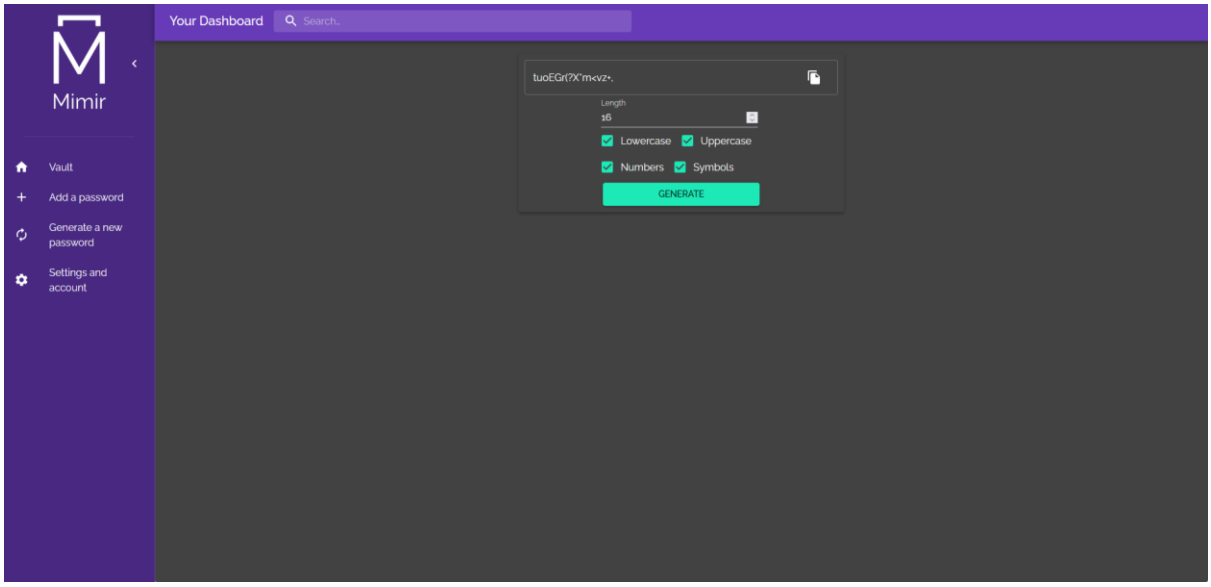

Mimir
Create an account or Log in
Username *
Email Address *
Password *
Confirm Password *
Password Hint
SIGN UP

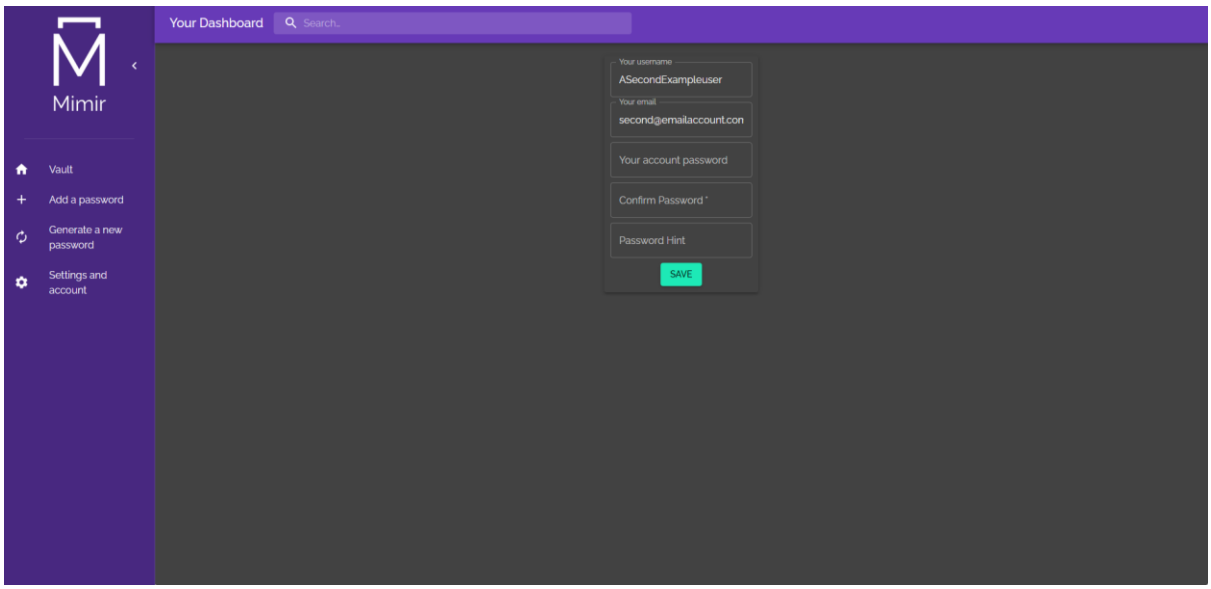

Mimir
Create an account or Log in
Username *
Password *
LOG IN


Mimir
Vault
Add a password
Generate a new password
Settings and account
Your Dashboard
Search..
First Example Record
Second Example Record
Third example record
Fourth example record
Fifth Example Record


Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
First Example Record
Name *
First Example Record
Website Address
examplewebsite.com
Website username
UsErNaMe
Website password
Notes
SAVE
DELETE
Second Example Record
Third example record
Fourth example record
Fifth Example Record


Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
Name *
Website Address
Website username
Website password
Notes
SAVE


Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
tuoEGr(?X"m<vz+,
Length
16
Lowercase
Uppercase
Numbers
Symbols
GENERATE


Your Dashboard
Search..
Mimir
Vault
Add a password
Generate a new password
Settings and account
Your username
ASecondExampleuser
Your email
second@emailaccount.con
Your account password
Confirm Password *
Password Hint
SAVE

Results

Functionality Testing

A series of tests have been constructed to validate whether the project functions as intended. For each test, a brief description is given, along with an expected outcome. The test is performed manually, and the actual outcome has been recorded. The actual outcome has been used to determine whether the tests were successful.

Test description	Expected outcome	Actual outcome	Successful?
Creating a new account	A new account is added to the database	A new account was added to the database	Yes
Attempting to create a new account with password fields that do not match	The user should not be able to create a new account and the request should not go through	The request did not go through	Yes
Attempting to create a new account with a password shorter than 8 characters	The user should not be able to create a new account and the request should not go through	The request did not go through	Yes
Logging in to an existing account	An existing account is successfully logged into, and the dashboard is opened	The account was logged into and the dashboard was opened	Yes
Attempting to login with incorrect details	The user is not allowed to login	The login attempt failed	Yes
Attempting to access a dashboard without logging in	The user is redirected to the login page	The attempt was redirected to the login page	Yes
Passwords are retrieved and displayed on the user's vault page	All password records for the logged in user are displayed on their vault page	All password records for the user were displayed on their vault page	Yes
Adding a new record	A new record is created using the component to add a record, and the record is displayed on the user's vault page	A new record was created using the corresponding component and the record was displayed in the vault page	Yes

Modifying a record	A record can be updated, and the changes are saved	A record was updated, and the changes were saved	Yes
Search bar	When a value is entered into the search bar, only relevant records in the vault will be shown	A value was entered into the search bar, and only records whose names contained that value were displayed	Yes
Changing user settings	The user's settings are successfully updated	The user's settings were successfully updated	Yes
Password encryption	The passwords in the database should be encrypted	The passwords in the database were encrypted	Yes
Password decryption	The passwords should be decrypted when retrieved and should be displayed in a record as plaintext	The passwords were decrypted upon retrieval and were displayed in the vault page as plaintext	Yes
Password generator	The password generator should generate a password of the specified length using specified characters	The password generator successfully generated a password 16 characters long using any combination of character sets	Yes
Copy to clipboard	A button on the textfield should allow for the contents of the textfield to be copied to the clipboard	The contents of the textfield were copied to the clipboard	Yes
HTTPS support	The application should run using HTTPS	The application ran using HTTPS	Yes
Dark mode theme	If the operating system theme preference is dark mode, the website should render in dark mode	The application rendered using dark mode	Yes

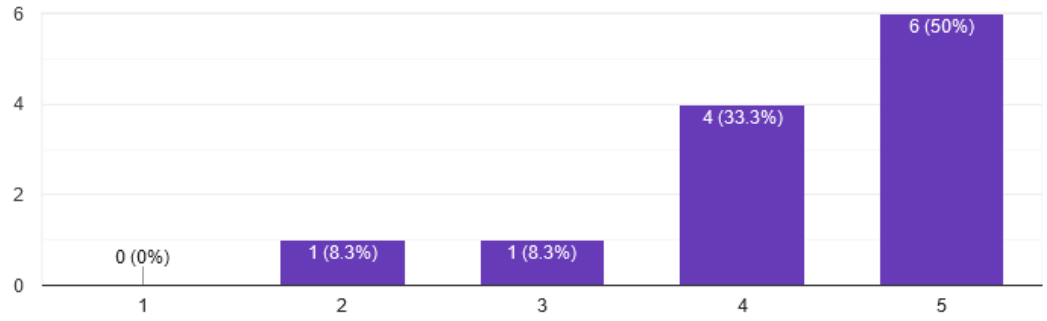
User Feedback

The questionnaire received 12 responses. Each respondent was allowed to use the web app until they felt they understood it completely and could give fair feedback. The respondents were allowed to ask questions, and some were given a brief demonstration. Some respondents were unfamiliar with the concept of a password manager, and the concept was explained before they used the application. The responses to the questionnaire are included below, divided into sections which contained questions focussed on one dimension of usability.

Effectiveness

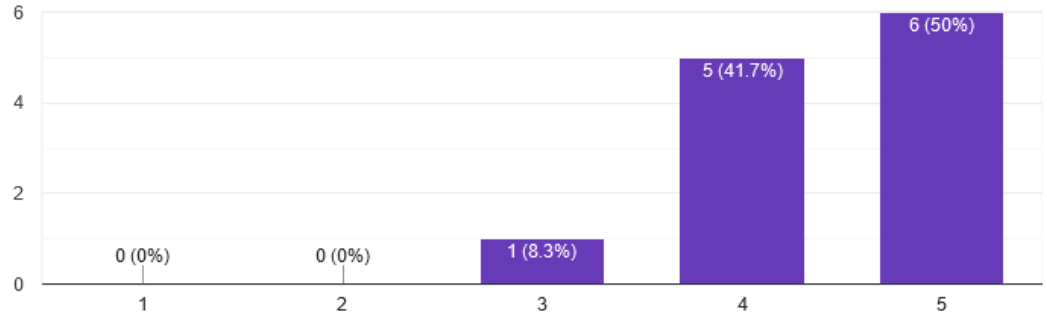
The software generates more secure passwords than I would make up on my own.

12 responses



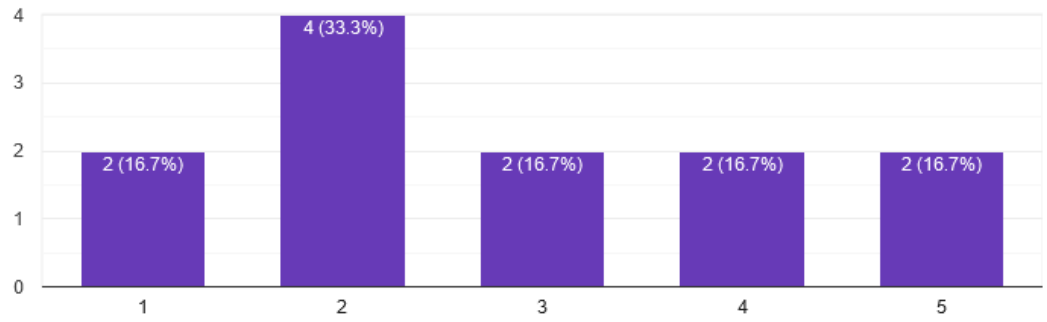
The software stores all of the password information I need.

12 responses



The software is easier than remembering my passwords.

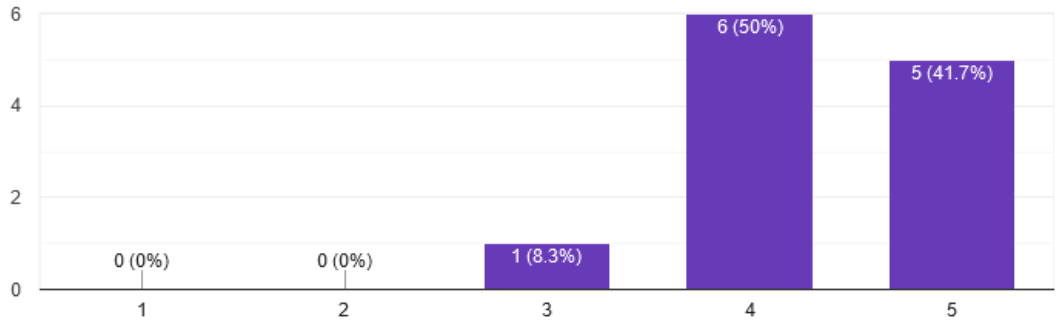
12 responses



Efficiency

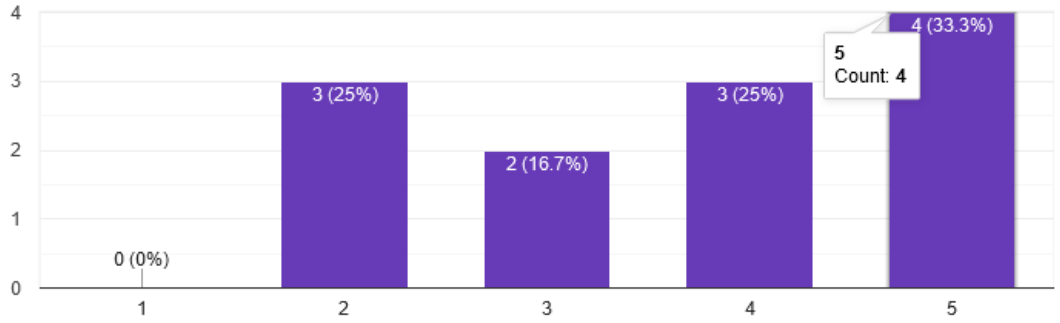
It is easy to retrieve a password from the software when I need it.

12 responses



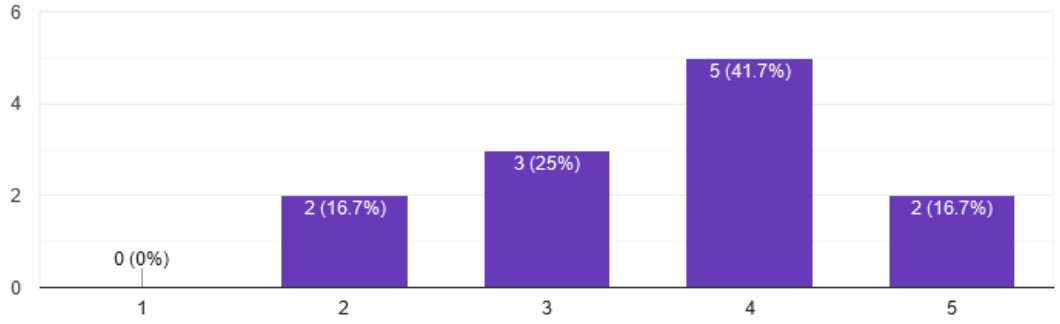
It is quick to generate a new password.

12 responses



The software is quick to use.

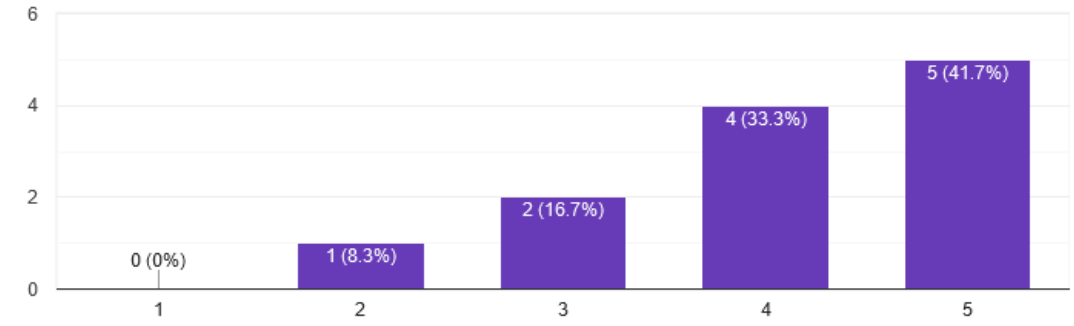
12 responses



Engagement

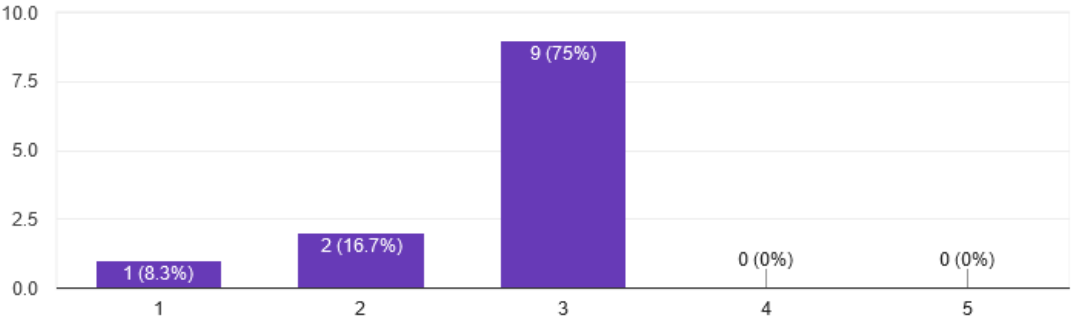
The software is easy to navigate.

12 responses



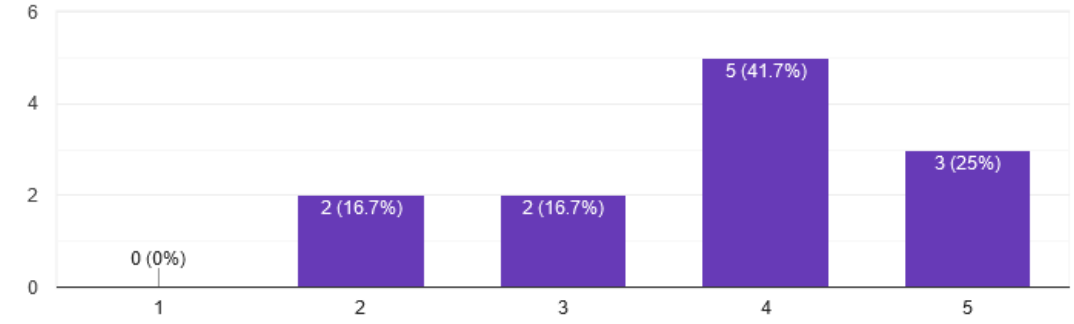
The software has a friendly appearance.

12 responses



The software is easy to understand.

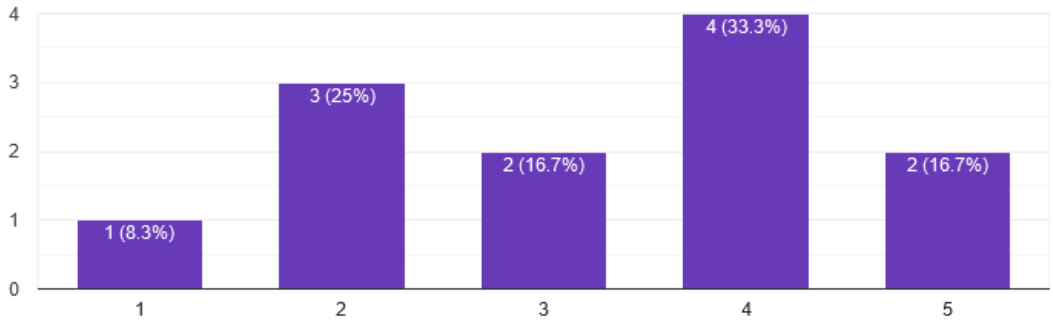
12 responses



Error Tolerance

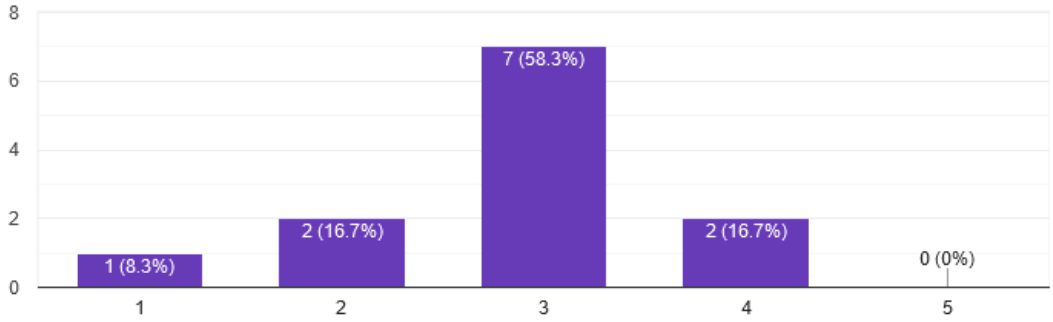
I rarely make mistakes using this software.

12 responses



It is easy to fix my mistakes.

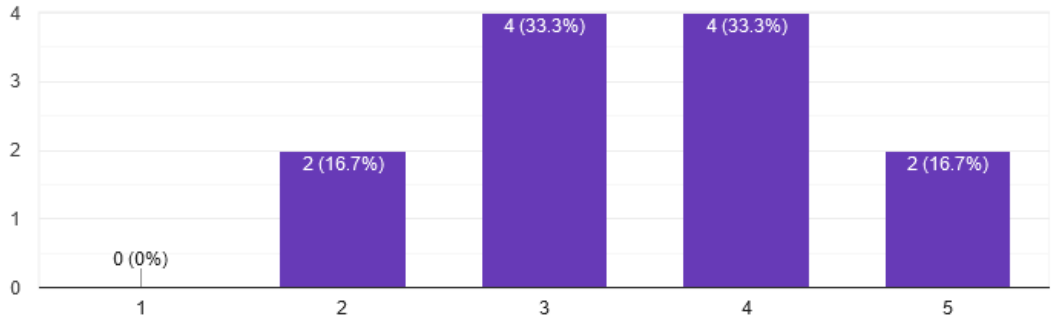
12 responses



Ease of Learning

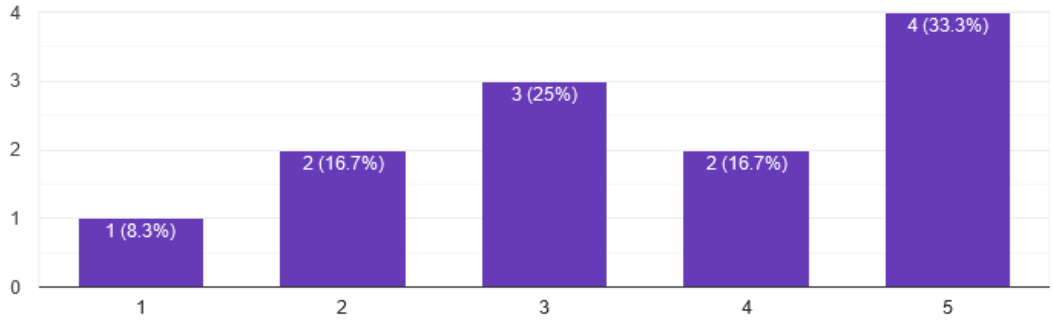
It did not take me long to learn how to use the software.

12 responses



The software did not need to be explained to me.

12 responses



Discussion and Analysis

Functionality Testing

The results of the validation tests show that the project passed every test. This indicates the software is robust and shows that each included feature has been successfully implemented without unintended side effects, achieving the desired outcome.

User Feedback

The first section of the questionnaire contained questions which targeted the dimension of effectiveness, aiming to gather opinions on how well the user felt they could achieve their goals. The mode response to the first statement “The software generates more secure passwords than I would make up on my own” was strongly agree, with agree being the second most common response. Only one response disagreed, strongly suggesting that users felt that the passwords generated by the project are stronger than passwords they otherwise use. The second statement “The software stores all of the password information I need” again had a mode response of strongly agree, and a second most popular response of agree. Only one result was neutral, and no responses disagreed. This overwhelmingly suggests that the users felt the password records held all the information the respondents would want to store in them, suggesting strong effectiveness. The third statement in this section “The software is easier than remembering my passwords” received much less agreement than the previous two statements, and a much more even distribution. Each response was chosen the same number of times, except for disagree, which was the mode with 4 responses. The wide variety of responses makes interpretation of these results difficult, however it is possible that many respondents chose a lower level of agreement than the previous statements because they practice poor password hygiene. While password managers provide increased security, they provide an additional layer of work which must be done to log into an online account, especially if the alternative is a short, reused password.

The second section focussed on statements targeting the dimension of efficiency, aiming to identify the respondent’s opinions on whether the password manager was quick and convenient to use. The first statement “It is easy to retrieve a password from the software

when I need it" received a modal answer of agree, and a second most popular answer of strongly agree. This suggests that the users found the software efficient, and they were able to find particular passwords quickly. The second statement "It is quick to generate a new password" received mixed responses, with the modal response being strongly agree. Despite this, disagree and agree both were chosen just one less time. This indicates users have varied opinions on the password generator component, possibly suggesting they found it fiddly or difficult to use or to copy over to make a new record. The last statement in this section "The software is quick to use" received a modal response of agree, suggesting the users were not frustrated by the speed of the software or how long it takes to perform an action.

The third section revolved around the dimension of engagement and aimed to gage user opinions as to whether the software was pleasant to use. The modal response for the first statement "The software is easy to navigate" was strongly agree, followed by agree. Only one respondent disagreed, with two giving neutral answers. This overwhelmingly suggests that users found the software engaging. The second statement "The software has a friendly appearance" received a modal response of neutral with 9 respondents choosing this response. This could suggest that the users had no strong opinions towards the appearance, finding it functional but not overly enjoyable. 3 respondents chose disagreeing responses, while no respondents chose agreeing responses, suggesting that no users found the design friendly. The third statement "The software is easy to understand" received modal responses of agree, with strongly agree being the second most common answer. This suggests many users did not struggle to understand what they were doing or using. The positive responses to the first and last statements in this section combined with the mostly neutral response to the second statement could suggest that while the users found the design approachable and simple, they did not find the design friendly or fun to use.

The fourth section targeted respondent's opinions on error tolerance, and whether they made many mistakes while using the software. The first statement "I rarely make mistakes using this software" received very evenly distributed responses. The modal response was agree with just 4 votes, while the second most popular result was disagree with 3 votes. The second statement "It is easy to fix my mistakes" received a modal response of neutral with 7 votes. The next most popular responses had just 2 votes. The responses to this section could

suggest that the users may not have had any strong feelings towards error tolerance in this project, finding it was neither notably easy nor difficult to avoid making mistakes. Adding additional warnings and checks which pop up before a user performs a significant action such as deleting a record could go a long way towards raising user responses for this section.

The final section, ease of learning, focussed on whether users found the software easy or difficult to learn how to use. The first statement in this category “It did not take me long to learn how to use the software” received two equally popular responses, neutral, and agree with 4 votes each. Two users chose both disagree and strongly agree, resulting in a distributed set of responses. No users felt that they strongly disagreed, indicating no users found they were completely unable to use the software. The second statement for this section “The software did not need to be explained to me” had a modal response of strongly agree, and a second most popular response of neutral. This shows many users believed they did not need help to use the software, however half of the responses were not in agreement with this statement. This could be due to the fact that a number of respondents were unfamiliar with password managers, and briefly had the concept explained to them beforehand rather than because they found the software itself difficult to figure out.

Overall, it appears users found the project to be mostly effective, strongly efficient, mostly engaging, averagely error tolerant, and mostly easy to learn. This strongly suggests that the users found the project to be overall very usable.

Limitations

The functionality tests performed on this project were tested manually instead of automated testing. This could suggest a potential source of failure to correctly identify any weak points or malfunctions in the project. For this reason, the addition of automated unit testing may have been a better approach, allowing for the tests to be run many times very quickly over a longer span of time. This could identify any long-term or rare errors, which may not appear during manual testing, but may appear a tiny proportion of the time, after the server has been running for a long time, or after the server has handled a certain

number of requests. Automated testing is also less prone to bias, though as these tests were objective and not subjective it is difficult to see how any bias may have affected the results.

The user feedback questionnaire was hampered by the fact that there were a relatively small number of respondents. Ideally, more respondents would have been found, however as the project was locally hosted it was difficult to show to many people, especially under current social distancing guidelines. The questionnaire also suffered from a few poorly chosen statements – namely “The software is easier than remembering my passwords” and “The software did not need to be explained to me”. The intent behind the first of these statements was to measure whether users found it easier to practice good password habits with the password manager than without, however it is unclear whether users responded to this feeling that practising good password hygiene with the password manager was more hassle than practising poor password hygiene. The second of these statements was intended to identify whether users felt they were unable to use the software itself without assistance or explanation, however it is possible some users felt unfamiliar with the idea of password management in general and needed some explanation.

Conclusion and Future Work

The investigated problem was to produce a password manager which was available as a web application and could be used to improve one's password habits and online security. An additional aim was to offer a service no more difficult or less convenient than not using a password manager, however with reflection it is exceedingly difficult to make any effective and secure password solution as easy and convenient as practising poor password habits such as reusing a short, simple password for every online account. User feedback indicates the project has, however, mostly succeeded in the creation of a password manager which is at least no more difficult or inconvenient than practising good password habits without a password manager.

Of the requirements identified in the solution approach, all the "must have" requirements were successfully implemented. Three of the five "should have" requirements were successfully implemented, with password hints being stored but never used to help the user login, and a folder system being completely unimplemented. Of the "could have" requirements one was implemented, with the project supporting light and dark themes. Properties were included at the beginning of the project for secret questions, email verification, and user lockout upon too many failed attempts; however, these features were left unimplemented due to time constraints.

With all the most important requirements, and the majority of the second most important requirements included in the project, the project could be called a success.

There is a clear path to future work on this project, with several features unimplemented. The most obvious are the remaining "should have" features – a folder system and a method of displaying the user's password hint if they struggle to login. Both features should be relatively easy to support from the foundation this project provides and could be implemented with more time. Many of the "could have" requirements could also be implemented, with the secret questions, password hint, and lockout features even having slight progress towards them in the project.

Bibliography

Florencio, D., & Herley, C. (2007). A large-scale study of web password habits. In Proceedings of the 16th international conference on World Wide Web (pp. 657-666).

Stanton, J. M., Stam, K. R., Mastrangelo, P., & Jolton, J. (2005). Analysis of end user security behaviors. Computers & security, 24(2), 124-133.

Das, A., Bonneau, J., Caesar, M., Borisov, N., & Wang, X. (2014,). The tangled web of password reuse. In NDSS (Vol. 14, No. 2014, pp. 23-26).

Huth, A., Orlando, M. and Pesante, L., 2012. Password security, protection, and management. *United States Computer Emergency Readiness Team*.

Blocki, J. and Datta, A., 2016, June. CASH: A cost asymmetric secure hash algorithm for optimal password protection. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)* (pp. 371-386). IEEE.

Provos, N. and Mazieres, D., 1999, June. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track* (pp. 81-91).

Atasu, K., Breveglieri, L. and Macchetti, M. (2004). Efficient AES implementations for ARM based platforms. *Proceedings of the 2004 ACM symposium on Applied computing - SAC '04*. [online] Available at: https://dl.acm.org/doi/pdf/10.1145/967900.968073.

Quesenbery, W. (2003). *Dimensions of Usability: Defining the Conversation, Driving the Process*. [online] *https://www.wqusability.com/*. Whitney Interactive Design. Available at: https://www.wqusability.com/articles/5es-upa2003.pdf.

Fielding, R.T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [online] Uci.edu. Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

IEEE, 2014. Improving usability of passphrase authentication. [online] Available at: <https://ieeexplore.ieee.org/abstract/document/6890939>.

Truica, C.-O., Boicea, A. and Trifan, I. (2013). *CRUD Operations in MongoDB*. [online] www.atlantis-press.com. Available at: https://www.atlantis-press.com/proceedings/icacsei-13/7568.

Ethelbert, O., Moghaddam, F.F., Wieder, P. and Yahyapour, R., 2017, August. A JSON token-based authentication and access management schema for Cloud SaaS applications. In *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)* (pp. 47-53). IEEE.

Joshi, A., Kale, S., Chandel, S. and Pal, D.K., 2015. Likert scale: Explored and explained. *British Journal of Applied Science & Technology*, *7*(4), p.396.

Aggarwal, S., 2018. Comparative analysis of MEAN stack and MERN stack. *International Journal of Recent Research Aspects*, 5(1), pp.127-132.

IBM Cloud Education, 2021. *mean-stack-explained*. [online] Ibm.com. Available at: <https://www.ibm.com/cloud/learn/mean-stack-explained>.

Ahmad, K.S., Ahmad, N., Tahir, H. and Khan, S., 2017, July. Fuzzy_MoSCoW: A fuzzy based MoSCoW method for the prioritization of software requirements. In *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)* (pp. 433-437). IEEE.

BAW Agency. 2021. *The Hero Section Checklist: 5 Essential Elements of a Hero Section that will Wow Visitors*. [online] Available at: <https://baw.agency/hero-section/>.

Elizabeth, L., 2016. *A Simple Web Developer's Color Guide — Smashing Magazine*. [online] Smashing Magazine. Available at: <https://www.smashingmagazine.com/2016/04/web-developer-guide-color/>.

Fireship (2021). *RESTful APIs in 100 Seconds // Build an API from Scratch with Node.js Express*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=-MTSQjw5DrM.

Mysliwiec, K., n.d. *Documentation | NestJS - A progressive Node.js framework*. [online] Documentation | NestJS - A progressive Node.js framework. Available at: <https://docs.nestjs.com>.

Mongodb.com. (n.d.). *The MongoDB 4.2 Manual — MongoDB Manual*. [online] Available at: https://docs.mongodb.com/manual/.

Mongoosejs.com. n.d. *Mongoose v5.12.15*. [online] Available at: <https://mongoosejs.com/docs/>.

Hanson, J. (n.d.). *Passport.js*. [online] Passport.js. Available at: https://www.passportjs.org/.

auth0.com (n.d.). *JWT.IO*. [online] Jwt.io. Available at: https://jwt.io/.

Spec.openapis.org. 2020. *OpenAPI Specification v3.0.3 | Introduction, Definitions, & More*. [online] Available at: <https://spec.openapis.org/oas/v3.0.3>.

Reactjs.org. n.d. *Getting Started – React*. [online] Available at: <https://reactjs.org/docs>.

Material Design. n.d. *Material Design*. [online] Available at: <https://material.io/>.

Material-ui.com. n.d. *Material-UI: A popular React UI framework*. [online] Available at: <https://material-ui.com/>.

Palmer, J., n.d. *Formik*. [online] Formik.org. Available at: <https://formik.org/>.

Quense, J., n.d. *Yup*. [online] GitHub. Available at: <https://github.com/jquense/yup>.

Zabriskie, M., n.d. *Axios*. [online] Axios-http.com. Available at: <https://axios-http.com/>.