An implementation of a password-less ecosystem

Raluca Angheluta (1931966)

Final project report submitted
in partial fulfillment for the degree of
BSc. in Computer Science with a Digital Technology Partnership - Degree Apprenticeship with Vodafone

Date: 11th April 2022
Word count: 9453

Project supervisor:
Dr Eike Ritter

Contents

1 Introduction

Digital authentication has been a concern in the world of computing ever since computer systems became commercially available in the 1950s. Originally, multiple users were sharing the resources and capabilities of a single machine, therefore protecting an individual's digital assets from other users became desirable. As a result, in 1961, passwords were introduced as a means to identify users, marking the beginning of digital authentication.

Since then, computers have evolved to become useful in many aspects of our everyday life, and the system that was initially used to protect local resources is now being used to secure access to digital services and assets across the internet.

Of course, what some people try to protect, others try to get hold of. Since their invention, password-based authentication systems have been the focus of many attackers' efforts. There have been many successful attempts to crack, bypass, intercept and steal passwords throughout history, making cybersecurity experts reconsider how these systems are being developed, deployed, and maintained. Over time, the implementation of password authentication has been subjected to enough scrutiny and modification that, in itself, can be considered sound.

However, despite the security of the implementation, there is a crucial factor that contributes to many types of phishing attacks still being viable, that factor being the human element.

2 Literature review

2.1 Problem

To understand why passwords fail to provide adequate protection, it is essential to understand where they fail. Although simple in appearance, much of the security of password-based authentication systems lies on users' shoulders - it is up to users to choose strong, high-entropy passwords; to set and memorise different passwords for each account, and to validate the legitimacy of the websites asking for their credentials. Studies point to the fact that these requirements are, most often than not, not enforced by users (Hanamsagar et al., 2016), therefore leaving a huge gap in the security of password authentication for brute-force, phishing, and credential-leakage attacks.

2.2 Attempted solutions

Several improvements, additions, and alternatives to password-based authentication systems have been researched and developed, such as graphical passwords, dynamic analysis of keystrokes (Elftmann, 2006); MFA and others.

By far, the most widely adopted is the Multi Factor Authentication scheme, which employs multiple verification factors to authenticate users. These factors are usually divided and characterised in the literature as "something you know" (password), "something you have" (security key, device, phone number, email address), and "something you are" (biometrics). Although a factor can be considered weak when taken individually, it is the combination of factors across different categories that is considered much harder for an attacker to get hold of. Therefore, by using MFA, the security of an account is no longer directly correlated to how weak or strong a password (or factor) is, but to the level of difficulty of obtaining the specific combination of credentials.

2.3 Attack Model

The use of MFA drastically reduces the chances of an account being hacked through attacks exploiting vulnerabilities in the password itself, such as brute-force, dictionary, and credential stuffing attacks. However, users of most MFA schemes used in commercial products continue to remain vulnerable to some types of spoofing attacks, specifically verifier impersonation attacks, as detailed in (Grimes, 2019) under the name of Network Session Hijacking. This is when a malicious website, pretending to be the authentic website, tricks a user into submitting their credentials (password + any other factors). The malicious verifier forwards all credentials received from the user, in real-time, to the authentic website and is thus granted access to the victim's account. This type of attack is possible when only one party is authenticating to the other instead of both - ie. only the user authenticates to the authenticating service - and thus the attack can be very effective on targets who do not verify the legitimacy of the system they are trying to authenticate to.

2.4 Solution

The best way to counter such attacks is by solving the underlying issue, which is the reliance on users for any security matters. Such a solution was proposed by the FIDO alliance under the name of FIDO2. FIDO2 is a term used to encompass a set of specifications (WebAuthn + CTAP) promising secure, password-less authentication through the use of authenticators (embedded - such as biometric scanners; or external - such as security keys or mobile devices).

FIDO2 authentication makes use of public-key cryptography to validate users' identity online. Authenticators registered with an account create a public-private key pair, that is locally saved against the associated account and domain. When used in an authentication flow, a cryptographic exchange takes place to validate the security key, however, the identity of the website requesting the exchange is also verified against the registered domain on the key.

This authentication mechanism does not rely on users to create fresh, strong passwords for each of their accounts - the authenticator creates a fresh public-private key pair to replace the password instead. The responsibility of validating the authenticity of the authenticating party is also taken away from the user and embedded into the authentication flow instead, thus answering all mentioned security concerns.

2.5 Disadvantages

In real world, however, no one-size-fits-all solution exists for a given problem, and this is also true with the FIDO2 authentication mechanism. Although the promise of strong security with no passwords is appealing, there are a number of drawbacks preventing FIDO2 from being the norm, as revealed in a number of studies (Lyastani et al., 2020) (Alqubaisi et al., 2020).

1. **Authenticator loss**
   The most significant issue identified in both studies is the difficulty in handling authenticator loss scenarios.
   Firstly, there no official solution to how such use cases should be handled, and the FIDO alliance's general advice is to enforce having two authenticators registered with an account at any time. Considering the price, availability and diversity of authenticators, this could be both unfeasible and impractical for users. Other account recovery methods have been evaluated in a recent paper (Kunke et al., 2021), however, the outcome suggests that more research needs to be done in this area.
   Secondly, as users register more and more accounts with their security key, losing it could require extensive efforts in trying to recover all lost accounts. The recovery process could differ from one application to another, and a large number of accounts registered on one authenticator could make this process unscalable for end users.

2. **Authenticator revocation**
   The second issue people showed concerns about was whether an attacker could gain access to their accounts if they managed to steal their authenticator. To combat the point developed by Lyastani in their paper (Lyastani et al., 2020), when set up accordingly, an authenticator can ask for proof of ownership upon a registration/authentication attempt - this usually takes the form of a key pin or, when

available, biometric identification. As such, even if an attacker gains physical access to an authenticator, they cannot use it for authentication without knowing the key's PIN or being the owner of the key.

3. **Corner cases, form and features**
   Another point was raised regarding the limited connectivity of authenticators. Improvements to this area can and have been made by companies offering FIDO2 enabled authenticators, such as Yubico, who now offer a USB-A & NFC security key model replacing the USB-A only model which was available at the time the studies were conducted.

4. **Mental model**
   Last, but not least, Lyastani concluded that people involuntarily associated the concept of authentication with passwords, which, coupled with the lack of transparency of the FIDO2 protocol, lead to mistrust (Lyastani et al., 2020). The author suggested making connections and associations to currently existing and relatable objects and concepts (such as physical keys) could lead to improvements in adoption rates.

2.6 Remedying disadvantages

From the few studies done, it is clear that FIDO2 authentication has not reached the maturity to obtain widespread adoption. While the standard is being continuously revised and altered, it is crucial that people get as much exposure to this new type of authentication as possible, to allow them to break the involuntary association between authentication and passwords and help them accept new solutions. As such, it becomes important to find ways in which FIDO2 authentication could be adapted to suit current users' needs and concerns, using contemporary technologies and solutions.

The loss of authenticators is perceived as the greatest concern amongst potential users, and requires careful consideration and a complex solution to address the problems that were raised.

When it comes to scalability of recovering accounts registered with a lost authenticator, inspiration could be drawn from in-use solutions for limiting password usage, namely Single Sign-On (SSO). Single Sign-On originated as an authentication scheme that allowed users to access services and resources from multiple sources by using just one set of credentials. Its aim was to reduce password usage and, with it, the opportunities for password reuse, loss and theft. Similarly, SSO could be adopted to reduce the number of accounts registered with an authenticator, thus limiting the scale of authenticator loss impact.

To address the security concerns surrounding account recovery, a new perspective can be explored, that of identifying trusted devices through web fingerprinting. Web fingerprinting is a term describing the collection and analysis of certain data with the aim to uniquely identify users based on the unique combination of hardware, software, and user-based customisation and configuration of their device and browser. A number of studies describe the effectiveness in differentiating users of a number of fingerprinting techniques, with some claiming to achieve close to 90% fingerprint uniqueness (Laperdrix et al., 2016). Although this is most often a method used for tracking users across different websites, the ability to passively and reliably identify users through their personal devices could be employed in an account recovery strategy.

Web fingerprinting is not a widely researched topic and, as such, claims in this area should be considered with caution. Moreover, networking and web technologies are constantly changing, leaving room to either improvements in web fingerprinting reliability or, the contrary, as concluded in a more recent study (Laperdrix et al., 2019). Although some studies (Alaca and van Oorschot, 2016) express concern in using web fingerprinting as a single factor authentication mechanism, its usability in account recovery strategies, with appropriate implementation of protection against spoofing attempts, has not yet been researched.

2.7 Project Proposal

For too long users have been trusted with the security of their accounts, leaving them vulnerable in front of phishing attempts. Alternatives such as FIDO2 exist, though users are reluctant in adopting them. This project aims to address this, by implementing a password-less authentication mechanism following the FIDO2 specifications, and adapting it to users' needs and concerns, by using existing technologies in a way that does not involve users in the authentication process.

3 Design

In order to ensure the success of the project, a thorough approach to planning and designing the solution was necessary. As such, the directions that were outlined in the Project Proposal section were broken down into high-level requirements, and a high-level design of the system was drawn to facilitate choosing the appropriate technologies and architectures.

3.1 Planning the design

In order to protect users and authentication systems from phishing attacks, a password-less authentication mechanism that follows the FIDO2 specification was chosen to be implemented. However, taking into consideration people's reluctance to adopt a strategy as dissimilar from password-based mechanisms as FIDO2, the project aims to bring a sense of familiarity, while also addressing the concerns related to authenticator loss.

To lessen the impact of authenticator loss, an ecosystem comprised of multiple applications was envisioned. The system would feature a number of applications called **Service Providers** (two, for the purposes of our application), offering access to resources and services to authenticated users of a central, **Identity Provider** application. The **Service Providers** would rely on the Identity Provider to securely authenticate its users, which can be achieved through a Single Sign-On authentication scheme. The **Identity Provider** would be responsible for providing users with all necessary means to register, authenticate, manage and recover their account, using authenticators as proof of account ownership. A high-level view of the design described was translated into a graphical representation, as seen in Figure 1.

The ecosystem was designed so that, in the case of authenticator loss, a user would only need to recover one account (that of the Identity Provider) in order to regain access to all supported services, as opposed to following an account recovery process for every service they lost access to when losing the security key. This solution also has the advantage of already being familiar to users, which could be a positive factor driving increase in adoption rates of FIDO2 authentication.

For the account recovery mechanism itself, it was desired to not rely on third-party systems (emails, OTPs) that may still be vulnerable to phishing attacks. In addition, to eliminate the impact of user error on the security of the overall system, a mechanism that does not involve users actively performing actions was desired. This resulted in the idea of being able to recover an account from a trusted device and, in order to identify trusted devices, web fingerprinting was proposed as a potential solution.

3.2 Functional Requirements

To better assess which technologies should be used and to prepare for the implementation stage, a list of feature-level requirements was drawn from the high-level design and then prioritised using the MoSCoW method.

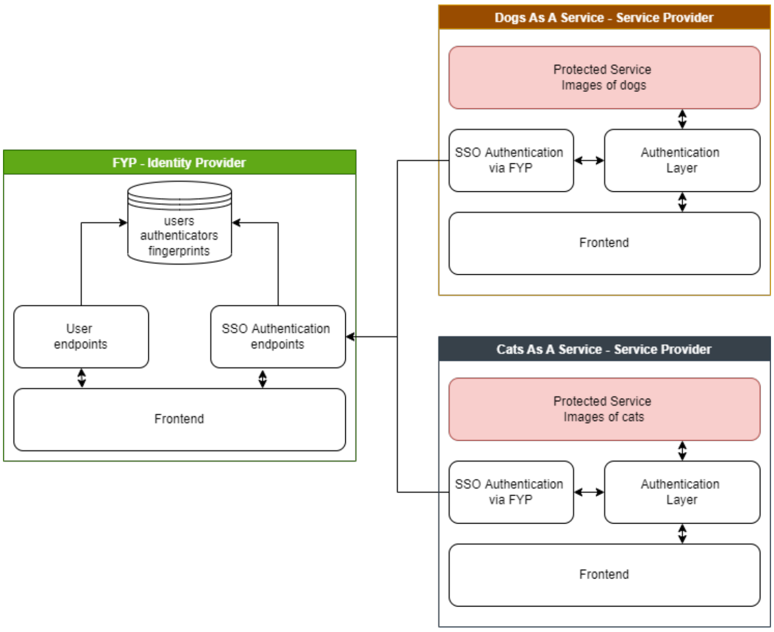

Figure 1: High Level Design

As a user of the **Identity Provider**, I want to...

FR1	**(must)**	... be able to create an account secured with an authenticator
FR2	**(must)**	... be able to log into my account using a registered authenticator
FR3	**(must)**	... be able to log out of my account
FR4	**(must)**	... be able to recover my account from a trusted device(and browser) should I loose or damage my authenticator
FR5	**(must)**	... be able to choose whether I trust a device(and browser) or not
FR6	**(should)**	... be able to manage my authenticators
FR7	**(should)**	... be able to manage my trusted devices(and browsers)
FR8	**(could)**	... be able to give user friendly names to registered authenticators
FR9	**(could)**	... be able to give user friendly names to devices(and browsers)

As a user of the **overall system**, I want to...

FR10	**(must)**	... be able to gain access to advertised services using my Identity Provider account
FR11	**(should)**	... be able to gain access to other external services using my Identity Provider account
FR12	**(could)**	... be able to control what personal data a service gets access to

3.3 Non Functional Requirements

Aside from the functional requirements, the system is expected to meet certain non-functional requirements as well. Although not attributed a priority class, these requirements are equally important, as they ensure a good user experience, which is essential in not deterring users away from novelty authentication systems.

NFR1. **The interface must be responsive.** The system is intended to work not just with external authenticators, but with platform authenticators as well. Since authenticators embedded into mobile devices amount to a significant proportion of platform authenticators, the web app should accommodate all screen sizes. Alternatively, the architecture of the system should be easily extensible to support native mobile applications.

NFR2. **The authentication mechanisms should be intuitive.** As the intent of this project is to drive users to make the switch from passwords to authenticators, simplicity and familiarity of use of the system is key in encouraging users to take this step.

NFR3. **The system should be easily extensible to accept new services.** In order to create a strong service base for users to encourage adoption, the effort of adding support for a new service should be minimal.

NFR4. **The system must handle faults and errors gracefully**, however, there is some nuance to this requirement. Due to the novelty of the FIDO2 authentication mechanism, the system could display transparent error messages to users, so that any underlying issues could be acknowledged and, if the case, dealt with accordingly. There is a trade-off between displaying transparent error messages and potentially giving away sensitive information to an attacker though. For example, when trying to recover an account using an untrusted device, a transparent error could state that *the device is not trusted for the given account*, allowing an attacker to learn that, firstly, the account exists and, secondly, the device used to recover the account is not trusted for that particular account. If the system is concerned more on the security of the solution rather than user experience, then the same error could produce a more generic message, such as *Unsuccessful account recovery attempt*, therefore obfuscating sensitive information.

In order to highlight the functionalities of the project, transparent error messages should be adopted, however, a production version of this should implement more abstract error messages.

3.4 Architecture and Technologies

The aim of the project carried a significant role in the choice of technologies and architectures being employed. Given that web applications are the main target of phishing attacks, it became apparent that such an authentication system would be most beneficial in the webspace and, as such, the focus of the project was directed at creating a web-based solution.

The particular choice of web technologies used for development was heavily influenced by the requirements. For all application composing the ecosystem, a **client-server architecture** was chosen, where the client is a statically served page and the server is a RESTful API. This architecture allows web applications to offer restricted access to resources and functionalities, since the implementation enforcing the restriction, and the restricted data itself would live on the server and not be visible to client-side users or attackers.

On **client-side**, I decided to develop all apps as Single-Page Applications (SPA) using ReactJS. The primary reason for adopting a SPA design instead of the traditional Multi-Page Application was to achieve a highly responsive feel for users (Wikipedia, 2022). In addition to the responsiveness of the apps, I also wanted to produce an intuitive and affordable user design. For that, I made use of Material-UI to create components and designs that adhere to Google's good design guidelines (Google, 2022).

On **server-side**, each web app was designed to communicate to its own RESTful API. Although the applications of the ecosystem communicate between themselves, they serve individual purposes, offer different services and, therefore, should be served by individual APIs. In a production scenario, the Identity Provider and all Service Provider application would most likely be developed by separate entities, and released to production as separate products. Therefore, the project was built trying to replicate the most probable real-world scenario, which is the one described.
A decision to implement all APIs in NodeJS, with the help of the Express framework was made. This was because, compared to other popular languages and frameworks used to develop RESTful APIs (eg. Java, .NET, etc.) the event-driven architecture, coupled with the asynchronous I/O capabilities of NodeJS shows better performance in more I/O intensive tasks, at the expense of reduced performance in CPU intensive tasks, according to a study (Chitra and Satapathy, 2017). Since most API calls would involve communication with the database and limited computations, NodeJS appeared to be the

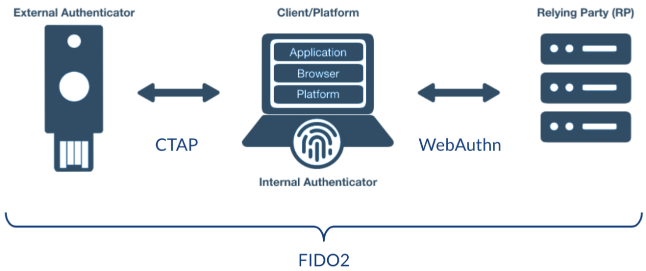

Figure 2: FIDO2 communication flow

more appropriate choice.

Additionally, the **Identity Provider** would also require a database solution to store details about users, their authenticators and their trusted devices, thus uplifting the architectural design of this app to a three-tiered model. Throughout the implementation process, I wanted to have the freedom to modify the structure of the data being stored as needed. With that in mind, the MongoDB NoSQL solution appeared to best facilitate this type of changes, since it is not restricted by a predefined, rigid schema which characterizes SQL solutions.

To aid the development of the project, I also considered making use of several libraries:

1. **SimpleWebAuthn** (Miller, 2020)

   As described in Literature review, FIDO2 encapsulates a set of specifications called WebAuthn (Web Authentication) and CTAP (Client to Authenticator Protocol). WebAuthn is a browser built-in API that creates a standard interface, which enables web applications to indirectly communicate with an authenticator device via the browser. The WebAuthn API forwards requests to authenticators in a format they understand. The API either communicates directly to platform authenticators or, if available, to external authenticators through CTAP. The FIDO2 communication flow was depicted, in a simplified manner, in an online blog (Sinitsyna, 2019), as shown in FIDO2 communication flow.

   Since the ecosystem will be web-based, the WebAuthn API will be the medium of communication between the authenticating application and authenticators. To facilitate making the relevant API calls and to ensure the responses from the API are validated appropriately, the **SimpleWebAuthn** library was chosen to be used.

2. **jsonwebtoken** (Auth0, 2015)

   This library provides an implementation of RFC7519 Jones and Sakimura (2015), which is a secure method of transmitting information in a JSON format.

3. **node-oidc-provider** (Skokan, 2015) and **node-openID-client** (Skokan, 2016)

   The implementation of a complete Single Sign-On solution - encompassing both the Identity Provider and the client - can be a challenging and time-consuming task even for large development teams. As such, to speed up the development and guarantee the security of the solution, I have used the above-mentioned libraries.

4. **FingerprintJS** (FingerprintJS, 2022)

   This library implements some web fingerprinting techniques. In order to create a relatively persistent but complex enough fingerprint to differentiate user agents and devices reliably, various techniques of data analysis and gathering are normally employed. Some of the information composing the fingerprint is easily accessible by interrogating the user agent (such as installed fonts, preferred languages, IP address, etc.). There is another more opaque layer of information forming a fingerprint, which is obtained by analysing various processes and their results and making inferences based on the analysis. The types of processes that are being analysed are ones where the end result is heavily influenced by the hardware of a particular system. An example of this is the way in which an image is rendered is heavily influenced by the CPU and the GPU of a computer. Another example is that certain constants (pi, e) also vary slightly between CPU classes.

   Because of the complexity of data gathering and analysis, I have opted to use the above library for web fingerprinting.

5. Public APIs: **Dog API** (Dog-API, 2022) and **The Cat API** (ThatAPICompany, 2021)

   The public APIs provide pictures of dogs and cats respectively. These images would represent the "service" being offered by the two Service Providers, which should be implemented as part of the project.

4 Implementation

Having established the requirements of the project and having sketched a high-level design, the project was ready to enter the implementation stage. This section will cover the fine details of the implementation - the finalised architecture of the individual applications, how the apps communicate, and how the code was structured to allow easier development and easier extensibility of existing functionalities.

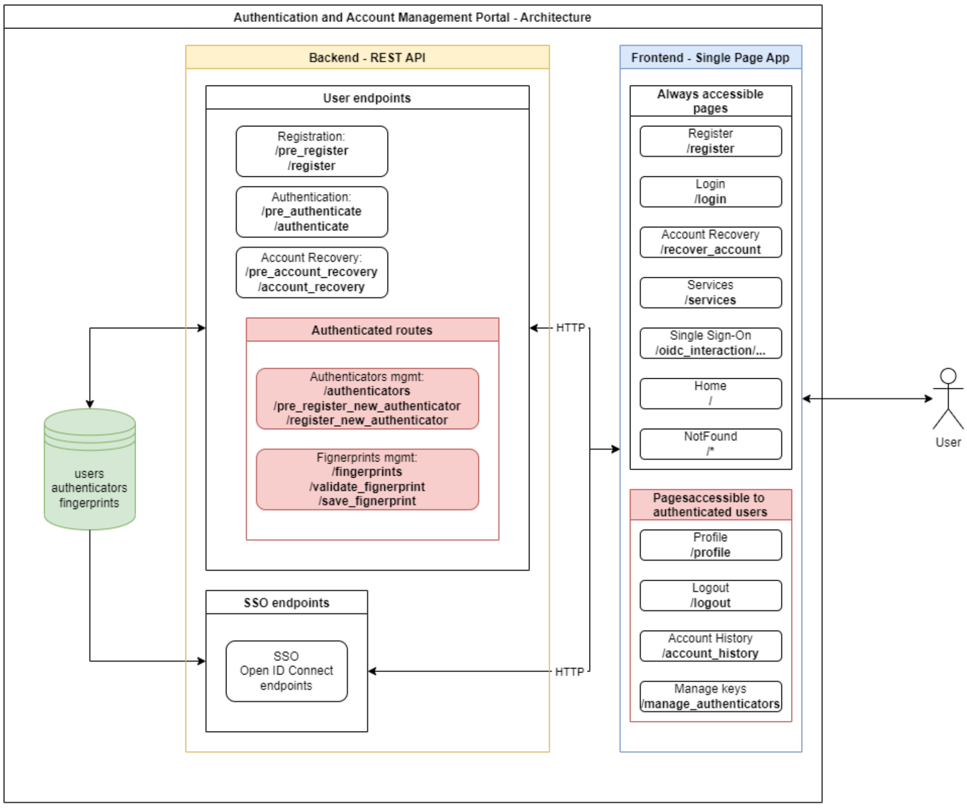

Figure 3: Identity Provider - Architecture

4.1 Identity Provider

The Identity Provider is the central application in the ecosystem and it was designed to serve a dual role. Firstly, this is the app that users come into contact with when they want to interact with an account - whether it is to create, log into, recover or manage - and, as previously mentioned, all interactions would feature the use of security keys as a replacement for passwords. Its second role is to authenticate users of a number of Service Providers in order to grant them access to the resources and services offered by those.

As planned in the Design stage, the **Identity Provider** adopts a three-tiered architecture, where the tiers are the client, the REST API, and the NoSQL Database.

Identity Provider Client

The client is the web application being rendered in the browser when a user accesses the Identity Provider. Its role is to display graphical components that can either present information, capture user input, or be interacted with to achieve various outcomes. It also serves a role in all procedures involving security keys, and in collecting fingerprinting data.

Although the client is developed as a Single Page Application, I wanted to maintain the look of a Multi Page Application, with each page offering distinct functionalities. To achieve this, I made use of browser routing and complex elements which imitate the look of pages.

As detailed in Identity Provider - Architecture, the client provides users access to multiple pages (captures in Appendix), named after the supported functionality:

1. **Register**

   The Registration page provides a responsive form, which users are expected to fill in with their personal information to create an account. When the user initiates the registration procedure, the client performs several calls to the Identity Provider API and also to the WebAuthn API in the background.

   The Account Registration is a three-step process:

   1. The client sends the user details to the server for validation. If successful, the server replies with **registration options**, necessary for the client to make a request to the WebAuthn API to register a new authenticator.

   2. The client forwards the **registration options** to the WebAuthn API, which prompts the user to locally complete some authenticator registration checks. On successful completion, the WebAuthn API responds with an **attestation response**, containing details identifying the key and proving the key was involved in the process.

   3. The client forwards the **attestation response** to the server via an HTTP request for validation.

   As highlighted above, the role of the client in the authenticator registration step is to forward requests between the server and the WebAuthn API.

   Once an account is registered, there are two more events taking place client-side. Firstly, the user receives a session token - useful in server-side session management - which is saved in the local storage of the browser. Secondly, a fingerprint of the user's device and browser is produced. If the user allows, the fingerprint is sent to the server to be saved as a trusted device.

2. **Authenticate**

   The Authentication page allows users to log into their accounts. It provides a responsive form, which the user is expected to fill in with their personal information.

   Client-side, authentication and registration are similar. The difference lies in the data being passed between the Identity Provider API and the WebAuthn API. Instead of registration options, the server sends **authentication options**, and instead of attestation response, WebAuthn sends an **assertion response**.

Similarly to registration, once the user is authenticated, the client receives a session token which it saves in the local storage. It also computes the user's fingerprint and sends it to the server for validation. If it does not match existing records, the client asks for user permission to save it.

3. **Account Recovery**

   The Account Recovery page presents a form to the user asking for the user details provided at the registration of the account.

   When the account recovery procedure is initiated, the client sends the user details, along with a freshly recorded user fingerprint to the server to be verified against existing records. If a match is found, then the user is allowed to register a new authenticator, similar to the registration flow.

4. **Services**

   The Services page is the place where all supported Service Providers are advertised.

5. **Single Sign-On**

   The Single Sign-On section includes two pages - Consent and Login. These pages are responsible for handling the login and consent user interaction within the Single Sign-On context.

6. **Home**

   The Home page is the page the user is redirected to once registration/login/account recovery is complete.

7. **Not Found**

   The Not Found page is displayed to inform users that the route they are trying to access is not available. This page is shown for nonexistent routes or protected routes being accessed by unauthenticated users.

All the above pages are public, however, there are certain pages that only authenticated users can access.

Client-side, authentication status is maintained in a state variable named **user**, containing user details. To make it accessible globally, the React Context component was used.

The protected pages are as follows:

1. **Profile**

The Profile page displays the full user details. The information is obtained from the **user** state and does not necessitate API calls to the server.

2. **Logout**

   The Logout component resets the **user** state variable and removes the session token from local storage, thus clearing any session information that was allowing the user to stay authenticated.

3. **Account History**

   The Account History page allows users to visualise their trusted fingerprints (corresponding to a device-browser pair). This is done by requesting all fingerprints associated with that particular account from the server. The device-browser pair currently in use is highlighted accordingly.

4. **Manage Keys**

   The Manage Keys page allows users to manage their authenticators. A list containing all registered keys is displayed, based on information retrieved from the API. Users can delete keys and create new ones, provided at least one key is registered at all times.

Registering and authenticating security keys requires the use of the WebAuthn API. Client-side, this was facilitated by the SimpleWebAuthn library from which two functions were imported: *startRegistration* and *startAuthentication*. These functions take as input the **registration/authentication options**, received from the server's API, and convert the information into the appropriate format used by the WebAuthn API (Uint8Array). The appropriate WebAuthn function (create/get) is then called with the converted data as input, which returns an **attestation/assertion response** to the client. The client is then responsible for forwarding the response to the server for verification.

The other commonly used feature is fingerprinting. As discussed in the Design section, obtaining a complex and reliable fingerprint requires extensive data gathering and analysis techniques that would necessitate more time to research and develop than the project timeline would have allowed. As such, I used the FingerprintJS library, which combines multiple fingerprinting techniques from three out of the four types defined in literature (Alaca and van Oorschot, 2016) - those categories being *Browser provided information*, *Inference based on device behaviour* and *Extensions and plugins*. To create a more reliable fingerprint, fingerprinting data from the fourth category is extracted, however, this is done server-side in an attempt to prevent spoofing.

Identity Provider Server

While the client is the medium the user interacts with, the server is what enables it to offer users the information and functionalities that it does. It achieves so by listening for requests from the client, processing them, and providing appropriate responses.

As motivated in the Design section, the server implementation conforms to a REST API architecture. When developing the API, it was important that its codebase had a modular, easily extensible structure. As such, the key pieces of implementation that make up the API were split into components that served a particular function. The most notable components are:

1. **Routes** - The **routes** component (residing in the Routes folder) sets the path names and associated HTTP methods where endpoints are mounted, essentially determining the structure of the API. Each endpoint is associated a **controller** that handles the incoming request to that particular path and method. Any changes to the API structure (route names, HTTP methods) are made in this component.

2. **Controllers** - A **controller** (located in the Controllers folder) contains the function being executed to process and respond to requests hitting the associated endpoint - these functions often involve data validation and CRUD operations on the database (accessed from **Models**). Changes to how a request is handled only require modifications in the associated controller.

3. **Models** - The **model** component (located in the Models folder) sets the structure of the data stored in the database and provides functionality to perform CRUD operations on said data. It exposes a number of queries that are imported by controllers as needed. As such, changes to the data being stored in the database or the addition of new queries are done in the Models component.

The API exposes several endpoints, grouped by the functionality they support:

1. User registration endpoints: POST/**pre_register** and POST/**register**

   The /**pre_register** endpoint is called in preparation for the authenticator registration. This endpoint expects to receive user details from the client (email, name, username), which it then looks up in the database to ensure no account with the same details exists. If so, then a fresh set of registration options is generated to be passed to the authenticator. The set includes, amongst others, important details such as:
   - name and ID or the application requesting the authenticator registration
   - the ID and username of the user for which the key is being registered
   - a fresh challenge that the key is expected to sign as part of the registration procedure.

The user details are then saved as a pre-registered user - meaning the user has not completed the registration yet. The freshly generated challenge is also saved against this entry, in order to be later validated when a security key is registered with the account. Storing the challenge in the database is a necessary step in a REST API, as the server does not maintain any session information locally.
Finally, the registration options are sent as a response to the client's request, which are then passed to the browser's WebAuthn API to initiate the authenticator registration procedure.

The /**register** endpoint is called by the client following the registration of a new public-private key pair on a security key. The expected request parameters are user details and the attestation response, containing:

- the key's credential ID
- the public key
- the challenge signed with the private key
- other attestation data

The signed challenge is validated against the shared public key and the challenge is saved in the database. If valid, the authenticator id and public key are stored in the authenticators collection in the database, together with the user's email. This marks the completion of the registration process.

Lastly, a JWT session token is generated, storing user details. This is passed in the response to the client to be stored in the browser for future requests.

2. User authentication endpoints: POST/**pre_authenticate** and POST/**authenticate**

   The /**pre_authenticate** endpoint is called in preparation for the security key authentication. This endpoint expects user details (email address) from the client. The user details are looked up in the database and, if an account is identified, the list of registered authenticators is returned. This list serves as input to generate the authentication options, containing an array of allowed credentials and a fresh challenge to be signed by the security key. Again, the challenge is saved in the database against the corresponding user for persistence. The options are sent to the client, who forwards them to the WebAuthn API to initiate the authenticator verification procedure.

   The /**authenticate** endpoint is called once the client receives the assertion response from the WebAuthn API, containing, amongst others, the credential ID of the key and the signed challenge. The server verifies the signed challenge against the public key (corresponding to the credential ID) and challenge retrieved from the database. If the two challenges match, then the authentication is declared successful.

Finally, the server replies with a session token to be used in future calls.

3. Account recovery endpoints: POST/**pre_account_recovery** and POST/**account_recovery**

   These behave similarly to the registration endpoints.

   The /**pre_account_recovery** endpoint expects full user details and a web fingerprint as parameters. These are validated against existing records in the database. If they match, then, just like in the pre_registration flow, options to register a new security key are created, the challenge is saved in the database and the options are sent back to the client. The client then proceeds to register a new security key by forwarding the response to the WebAuthn API.

   The /**_account_recovery** endpoint, expects user details and an attestation response, which are validated as detailed in /**register**. The key difference is that, once the attestation object is successfully validated, all previously registered authenticators are deleted from the database before saving the new authenticator details.

   Finally, a session token is created and sent to the client.

Just as some Frontend pages are protected from unauthenticated users, so are their corresponding endpoints. In contrast with the client-side implementation of protected pages, the server is a REST API and, therefore, does not keep track of state or session.

To protect endpoints, the client is required to send the session token (created and shared to the client upon successful registration/authentication/account recovery) in the header of the request. The session token is based on the JSON Web Token (JWT) standard (Jones and Sakimura, 2015), and contains user information (email address, username, name), so that the server can identify the user making the call. The data is also signed with a secret only known to the server so that a tampered token can be rejected along with the request.

To avoid code repetition, the validation of tokens and the data extraction functionality have been implemented in a middleware function sitting in front of all protected routes. Requests to these endpoints hit the middleware layer first. There, the token is retrieved from the request headers, is verified and the user information is extracted. If successful, the middleware saves the user information in the request object, which is then passed to the associated controller. If the signature verification of the token fails, the middleware immediately sends a 401 Unauthorized response, and the request does not reach the corresponding controller anymore.

The endpoints below have been protected as described, as they provide either personal information about a user's account, or functionalities that should only be available to authenticated users.

1. Manage authenticators endpoints: GET, DELETE/**authenticators**

   A GET request to /**authenticators** is expected to come with a session token in the request header. After the user details are extracted from the token, all user's authenticators are retrieved from the database and returned to the client.

   A DELETE request to **authenticators** is expected to come with a session token and the credential ID of the authenticator to be deleted. The user details are passed to the relevant controller, where the authenticator is deleted from the database. Checks were put in place so that the last registered authenticator cannot be deleted.

2. New authenticator registration: POST/**pre_register_new_authenticator** and POST/**register_new_authenticator**,

   The /**pre_register_new_authenticator** endpoint is similar in role with /**pre_register**, however, the main difference lies in how the request is made. Instead of user details being passed as request parameters, this endpoint expects a session token in the request header. This is then processed by the middleware to return the user details necessary for the registration of a new security key. Next, the registration options are generated, the freshly generated challenge is saved in the database against the user, and the endpoint responds with the registration options.

   The /**register_new_authenticator** endpoint is also similar with /**register**. The only distinction between the two is that /**register_new_authenticator** takes a session token in the request header to extract the user details, while the other takes the user details in the request parameters. The rest of the functionality is replicated.

3. Web fingerprint endpoints: GET/**fingerprints**, POST/**validate_fingerprint** and POST/**save_fingerprint**

   The first endpoint returns a list of recorded fingerprints, the second takes a fingerprint and determines whether it is new (has not been recorded yet) and the third saves a fingerprint in the database.

   GET/**fingerprints** expects a session token in the request header. A list of recorded fingerprints is retrieved for the user, which is parsed to aggregate the repeating fingerprints. A new list of unique fingerprints, featuring the fingerprint ID, the number of times it was observed, and when it was last observed is returned to the client.

   POST/**save_fingerprint** expects a session token and a fingerprint. To improve the discriminatory quality of the fingerprint, the server collects the IP address of the client to add to the fingerprint data. The fingerprint is then saved against the user in the session token.

POST/**validate_fingerprint** also expects a session token and a fingerprint. Once the middleware returns the user details, a query is made to the database to determine whether the given fingerprint and the client's IP address match any existing records, and the result is returned to the client.

There are some additional endpoints exposed by the API which are used in the Single Sign-On Implementation. These shall be covered in the Communication between Identity Provider and Service Provider section.

To ensure conformance to FIDO2 specs, the SimpleWebAuthn library was used to help generate registration and authentication options and to help validate attestation and assertion objects.

4.2 Service Provider

A Service Provider allows access to its services to users who successfully authenticate with the Identity Provider via Single Sign-On.

Within the ecosystem, there are two supported Service Providers - one providing pictures of dogs, and the other pictures of cats.

As planned in the Design stage, both Service Providers adopt a client-server architecture, where the client is a React Application and the server is a REST API.

Client:

The client offers a limited number of pages, as a result of the limited functionality the application needs to support:

1. **Home Page** - This is the landing page. If a user is authenticated, it displays a hello message that includes user details, otherwise, it advises the user to log in. The purpose of the page is to visualise the results of authentication via Single Sign-On.
2. **Login Page** - This page also implements limited functionality. It features a button that, once clicked, redirects the user to the Identity Provider to complete login. The purpose of this page is to emphasize that the user is about to be redirected to a different web application for authentication.

3. **Receive Token Page** - Following the completion of Single Sign-On authentication, the user is redirected with a token from the Identity Provider. The token is sent to the Service Provider API to be exchanged for a session token.
4. **Service Page** - This page is a protected page, similar to the protected pages on the Identity Provider. This is where images of dogs/cats are provided.
5. **Logout** - This deletes all session information.

Server:

The APIs of the two Service Providers are mostly alike, except for the resources being provided. The two APIs expose the following endpoints:

1. GET/**login** This endpoint is called when the user initiates the Login with the Identity Provider flow. It redirects the user to the appropriate Single Sign-On route of the Identity Provider.
2. GET/**dogs or /cats** These endpoints are protected endpoints. Similar to the protected endpoints in the Identity Provider API, these require a session token to be sent in the request header. If a session token is received, then an external API is called to obtain the resources to be sent to the client (images of dogs or cats).
3. GET/**logout** This endpoint deletes all Single Sign-On session cookies. As session cookies are set to be HTTP only, removing them can only be done server-side.
4. GET/**oidc/** and POST/**oidc/access_token** are both endpoints involved in the Single Sign-On process, the details of which are expanded in the Communication between Identity Provider and Service Provider section.

4.3 Communication between Identity Provider and Service Provider

The Service Provider relies on the Identity Provider to authenticate its users. In other to achieve this, the two applications communicate through a Single Sign-On authentication scheme.

Initially, the development plan featured SAML as the Single Sign-On strategy to be adopted. However, at the time of refining the approach to the implementation, its integration into the existing applications appeared to require a redesign of the architecture. The SAML protocol makes heavy use of assertions, which are usually large XML data structures, containing several statements. As part of the Single Sign-On flow, there are a significant number of redirect operations between all involved parties. Sending a SAML assertion as part of a redirect response is a challenging task since most browsers ignore the body of 302 status responses. Moreover, the assertion easily exceeds the maximum URL length for it to be passed as a URL parameter. Finding a secure way to send a SAML assertion would have possibly involved a few modifications, such as saving the assertion in a database to be later retrieved by the service requesting it. In addition, dealing with concurrent SAML authentication requests would have involved a way of tracking each authentication session against its corresponding request. At the time, this appeared to require some state awareness capabilities in the API, which would be in contradiction with the REST architecture that was adopted.

With the detailed complexities in mind and with the limited time available to finalise the implementation, I decided to go over the decision to use SAML and, instead, replace it with OpenID Connect. OpenID Connect is a more recently developed standard for Single Sign-On and it essentially adds an authentication layer to the OAuth 2.0 protocol. As detailed in the specs sheet (Sakimura et al., 2014), it allows clients (the Service Providers) to validate and request details about the identity of a user, based on the authentication performed by an Authorisation server (the Identity Provider), all in a REST-friendly manner.

As the OpenID Connect protocol is essentially a communication protocol between multiple parties, its implementation required development on both client and server, on all applications involved in the Single Sign-On flow, where part of the development has been facilitated by the use of two certified libraries. Specifically, within the Identity Provider, the functionality provided by the /oidc endpoints - session management, authorization code creation and validation, serving user data, etc. - is, for the most part, provided by the **node-oidc-provider** library (Skokan, 2015). Part of the library implementation can be altered (for example, configuring the list of accepted clients, changing session management parameters such as TimeToLive, type of session cookies, adding claims to be supported by the server, etc) - this is done in a configuration file, that can be easily modified, ensuring new Service Providers can easily be added to the supported Service Providers list. In the Service Provider, functionality such as the discovery of the Identity Provider configuration, creation of code and code verifier, deconstruction of the token, is provided by the **node-openID-client** library (Skokan, 2016).

The OpenID Connect protocol features multiple flows that cater to different contexts. For this project, the authorization code flow, as described in the OpenID Connect specifications(Sakimura et al., 2020), was chosen, as it was recommended to be used when the OpenID Connect client can maintain a shared secret between it and the OpenID Connect server. Given the Service Provider application implements a client-server architecture, the shared secret can live on the server and not be accessible to users or attackers in the frontend.

The flow of a Single Sign-On authentication, as observed in Single Sign-On strategy using OpenID Connect, is quite involved and, for simplicity, has been divided into smaller steps, as detailed below:

Service Provider:

(1) The user initiates the flow in the Service Provider client.

(2) A GET call is made to the /login endpoint of the Service Provider. When the call reaches the **/login** endpoint, a code and an associated code verifier are generated, which are later used in validating the token received from the Identity Provider.

Identity Provider:

(3) The user is then redirected to the Identity Provider. The request is made to the **/oidc/auth** endpoint, which takes the previously generated code and the scope (the information to be shared with the Service Provider) as parameters.

(4) **/oidc/auth** assesses the request. It creates a new session (and saves the session in a cookie), or re-establishes an existing session, and forwards the user to **/oidc_interaction** with additional session details.

(5) **/oidc_interaction**, based on the session cookie and information shared by **/oidc/auth**, assesses which interaction the user needs to complete next - login or consent - and redirects the user to the appropriate page. Since the user has not logged in yet, **/oidc_interaction** redirects the user to the Login Page.

(6) The user completes the login flow as normal, using their authenticator. Once successful, the frontend performs a POST request to **/oidc_interaction/login**, which is responsible for making updates to the session information to include the user's details and the state of the session (login complete, pending consent).

(7) **/oidc_interaction/login** gets the user details from the request, and passes these to **/oidc/auth**.

(8) **/oidc/auth** assesses the requests. If there are still user interactions to be handled (consent), it creates a new session for that particular interaction. It then forwards the session details to **/oidc_interaction**.

(9) **/oidc_interaction**, based on the session cookie and information shared by **/oidc/auth**, assesses which interaction the user needs to complete next. Since the user has logged in, but has not given consent to share their personal data, **/oidc_interaction** redirects the user to the Consent Page.

(10) Once the user gives consent for sharing their personal data with the Service Provider, a POST call is made to **/oidc_interaction/consent**, where the details the user consented to share are added to a list of grants.

(11) The grants are passed to **/oidc/auth** for processing. The request is in the final stage, where no more user input or action is required. As such, the endpoint returns an authorization code to the Service Provider.

Service Provider:

(12), (13), (14) The authorization code is received by the Service Provider. The token is validated against the code verifier created at step (2), and, if successful, is then exchanged for user info. Finally, the user info is converted into a JWT session token and is given to the Service Provider client in order to maintain a persistent session on the Service Provider.

The implementation of the Single Sign-On solution was the most challenging part of the project, even with the use of helper libraries.

The main difficulties came from handling networking restrictions imposed by web browsers. Many times throughout the implementation process, I have encountered CORS errors returned by the browser, where the error description was either vague and not suggestive of the underlying issue, or the cause of the error was clear, however, it could not be linked to a poor implementation within my code. For instance, a CORS request receiving a 302 Redirect response to a different server appears to have its Origin header set to null by the browser, as suggested in a Stackoverflow post (Stackoverflow, 2014) and in the specs of the Origin header (Barth, 2011) - I have continuously faced this issue despite my active efforts of trying to set the Origin header appropriately. Not only that, but 302 Redirect responses received in response to POST requests appear to trigger a generic CORS error in Chrome; although Redirect responses are not conventional, they are not prohibited by the specs, and so debugging such an issue was very difficult.

Apart from the obscure CORS errors, another element of difficulty was posed by the libraries themselves. Although rich in features, I found the documentation to be quite vague. Often times when trying to modify the configuration of the OpenID Connect server to suit the needs of my project (for example, modifying the session TTL, adding unsupported claims), I found myself having to read through the implementation of the library and work out how to modify any default set values by myself.

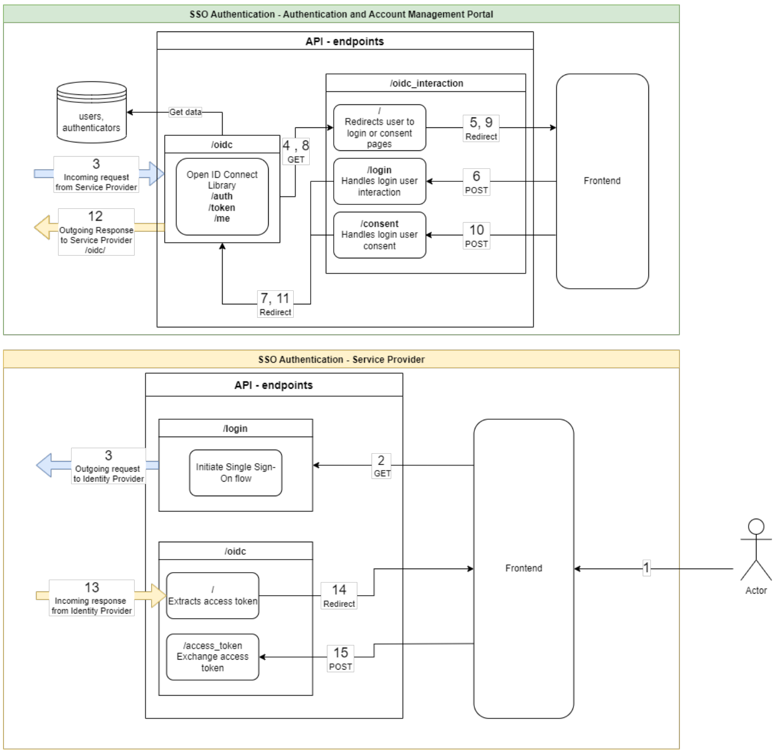

Figure 4: Single Sign-On strategy using OpenID Connect

5 Evaluation and results

To provide a comprehensive evaluation of the solution, the end solution needs to be analysed from both an implementation and a user experience perspective.

5.1 Limitations

Due to a number of impediments, not all functionalities of the project could be evaluated to the desired standard.

Firstly, the full solution could not be easily deployed to a live environment, due to the costs associated with purchasing domain names, SSL certificates, and hosting resources for all three applications. This made it difficult to conduct user evaluation with a large number of candidates.

Secondly, setting up a secure environment in the local network also proved challenging within the remaining time frame and was not achieved. As a result, the only environment where the registration and authentication of security keys on the Identity Provider can work is localhost, since all WebAuthn API calls are restricted to secure environments only. This made it impossible to test the functionalities of the applications on mobile devices since these do not have access to the localhost environment. However, the frontend has been tested to ensure its responsive design works on smaller screens as well.

5.2 Technical evaluation

To ensure the overall system provides all essential features captured in the requirements and works as expected, a technical evaluation was performed.

The adherence to the functional requirements was probed through Functional Unit Testing. A comprehensive testing plan was constructed, as shown in Functional Testing Plan. The plan details the functionalities and the encompassed functional requirements that are being tested, the steps to be executed and what the expected outcome is. As highlighted in the testing plan, all implemented requirements passed the Functional Unit Testing.

There are some requirements that were not met, due to the restrictive timeline. These were all lower priority and were not essential functionalities of the ecosystem. However, since they add an increase in the usability of the system, they should be considered in future work.

FR7 **(should)** ... be able to manage trusted devices(and browsers)
The feature, in itself, is not difficult to implement, and only requires a dedicated endpoint for fingerprint deletion. However, since the system does not support friendly names for devices (fingerprints), it would be difficult for a user to know which device they are deleting based on the information displayed currently.

FR8 **(could)** ... be able to give user-friendly names to registered authenticators
This feature would involve changes to all three tiers. Client-side, this requires careful consideration. It would not be desirable to introduce complexity in the account creation procedure by adding an extra step for naming the authenticator. Similarly, in the Manage Authenticators menu, the user would not be able to reliably distinguish between multiple keys based on information that is currently available (credential ID, last used, times used), and so they may not be able to name authenticators appropriately. Server-side, this would only require a dedicated endpoint for updating authenticators in the database. Data-wise, the schema of the authenticators collection would only need to be modified to feature an optional field for the key alias.

FR9 **(could)** ... be able to give user-friendly names to devices(and browsers) This is similar to the above, however, for fingerprints instead of authenticators.

FR11 **(should)** ... be able to gain access to other external services using the Identity Provider account To allow any application to register as a client of the Identity

Provider would require implementation of the OAuth Dynamic Client Registration Protocol in the Identity Provider.

FR12 **(could)** ... be able to control what personal data a service gets access to Giving users control of the data being shared to external apps requires modifications to both Identity and Service Providers. On Service Providers, attention should be paid to how missing data is handled. On Identity Provider, the client must allow the user to make the choice. Server-side, only the allowed information should be added to the user's grant.

Although the non-functional requirements were not captured within the Functional Unit Testing plan, they have either been indirectly evaluated as part of the testing, or they have been met through design and architectural choices, as detailed throughout the paper.

5.3 User evaluation

To assess how users feel about the implemented authentication system, a user evaluation was carried out. A number of people were presented with an overview of the ecosystem and were then asked to test it and answer a short questionnaire. The questions were inspired by Lyastani's work (Lyastani et al., 2020), so that the impact of the additional account recovery functionalities implemented (adopting Single Sign-On and device trust) could be assessed in relation to users' acceptance of password-less authentication.

Since the number of participants was limited, the results cannot be reliably generalised, however, they could give an indication of whether the system achieves to make password-less authentication more appealing.

Below is a comparative view between Lyastani's authentication system and the one presented in this paper. The techniques used in Lyastani's paper to measure **system usability** (the affordability of the presented system), **acceptance of technology interactions** (users' willingness to use technology), and **privacy concerns** were applied to our data to ensure a fair comparison:

Criteria	Lyastani's study	Our evaluation
System usability score	0.8	0.9
Acceptance of technology	0.6	0.056
Privacy concern	0.82	0.8
Computer Science background	60%	66%
Participants by education:		
< High school / A-levels	2 (4.2%)	1 (16.7%)
High school / A-levels	14 (29.8%)	2 (33.3%)
Bachelor	20 (42.6%)	2 (33.3%)
Master	11 (23.4%)	1 (16.7%)

Although the number of participants is much lower, the overall scores are similar, with System Usability being 10% higher.

Similar to the authors' findings, the participants found using a security key more facile than using regular password-based authentication systems:

- *"I found it easier to use a key to access all my accounts than using a password for every single account I have"*
- *"less of a need to remember loads of passwords"*
- *"no need to ... type passwords"*

However, in contrast to Lyastani's study, where participants expressed concerns regarding how an account can be recovered, the users testing the proposed authentication system mentioned they found the recovery process easy. They even expressed that the process was much simpler than other account recovery mechanisms currently in use:

- *"It was easier ... to recover the account compared to existing solutions"*
- *"If you lose the physical key, there is a way to recover your account and set up a new one without going through extensive setup (like answering security questions, clicking links from your email address)"*

Some participants even expressed their desire to have one centralised account, and overall, no privacy concerns were raised:

- *"I would use this method to aggregate all my accounts..."*
- *"I would use this authentication method to manage all my gaming accounts, since they are many and are hard to manage."; "I appreciate being told what personal information the external sites collect"*

There were still some negative points that the participants raised, most of which involved the key itself:

- *"Having to carry the authenticator with me at all times"*
- *"too expensive"*
- *"Connecting the key through USB port is a disadvantage for me because my laptop only has one port which is usually in use"*

6 Conclusion and further work

The goal of the project was to implement an authentication method that is resistant to all password attacks and, especially, phishing attacks. This was made possible by utilising state-of-the-art password-less authentication technology - FIDO2 - and addressing the shortcomings of this novel authentication mechanism by integrating it into a complex ecosystem and adopting a new account recovery strategy.

Although the overall system successfully answers users' concerns regarding authenticator loss, the security of the system, particularly the security and reliability of using web fingerprinting and 'device trust' in account recovery should be further assessed.

Moreover, the adoption of password-less authentication is still heavily influenced by the lack of diversity in authenticator devices. With most concerns being the price and connectivity of security keys and the fact they have to be actively carried, it is clear that improvements are yet to be made in this area. However, the fact that many mobile devices and laptops now feature biometric-enabled platform authenticators is encouraging, as it could help increase adoption rates of password-less authentication.

A Appendix

A.1 Identity Provider - screenshots

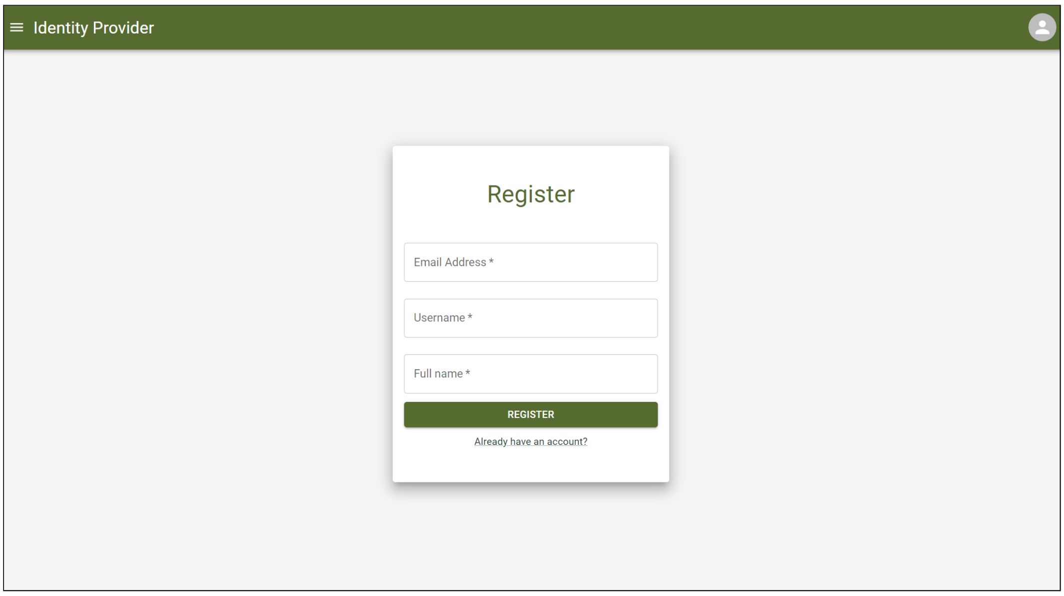

Figure 5: Registration Page

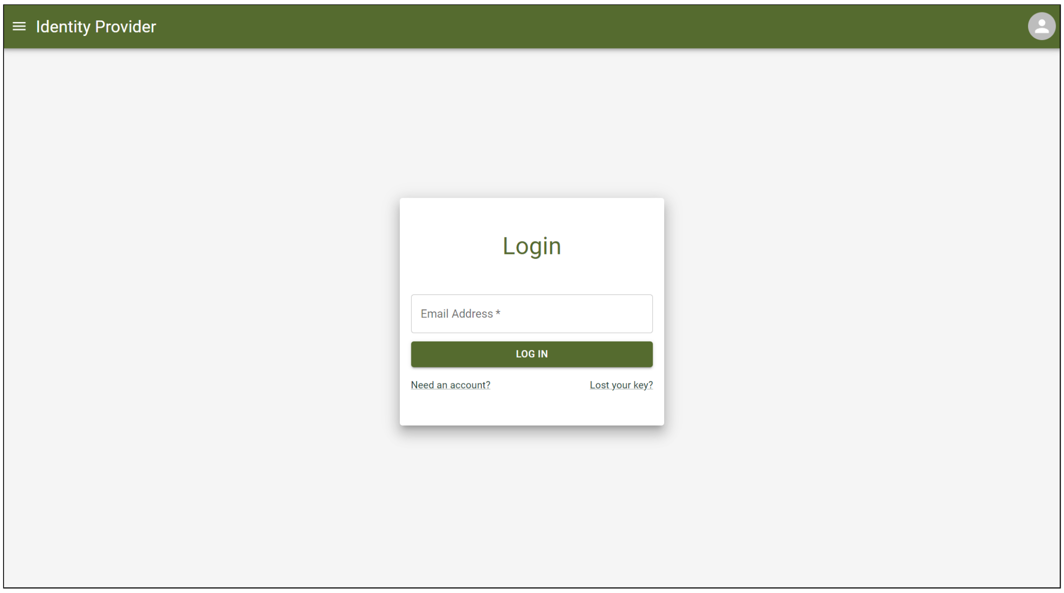

Figure 6: Login Page

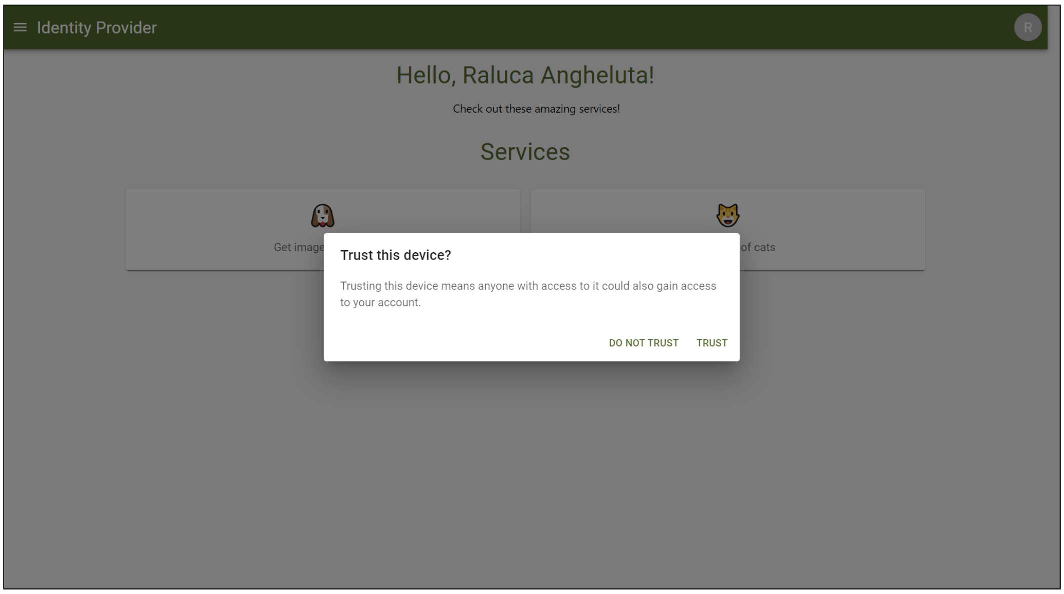

Figure 7: Trust Device Prompt

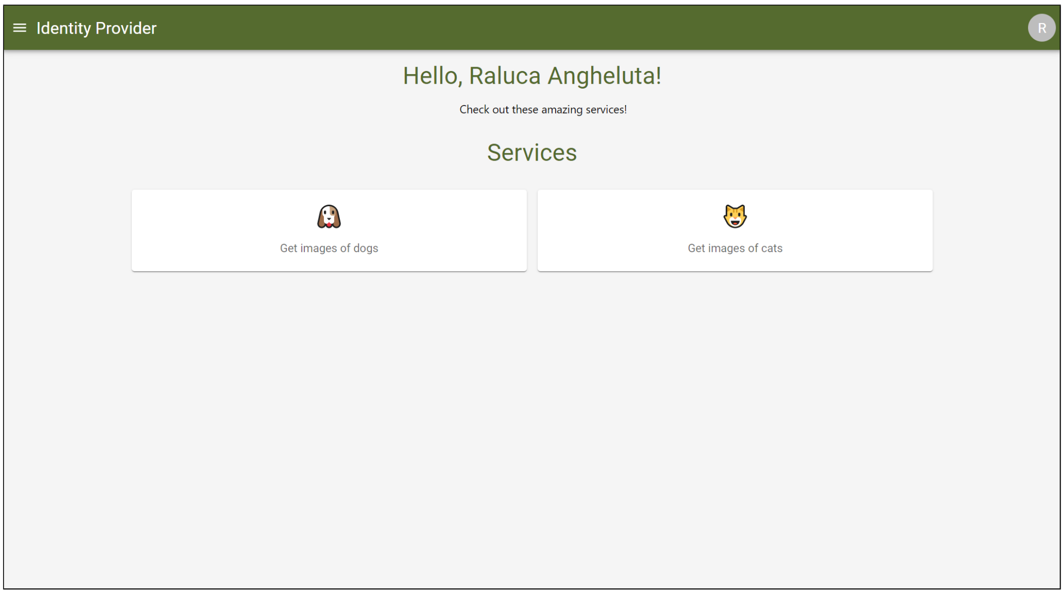

Figure 8: Home Page

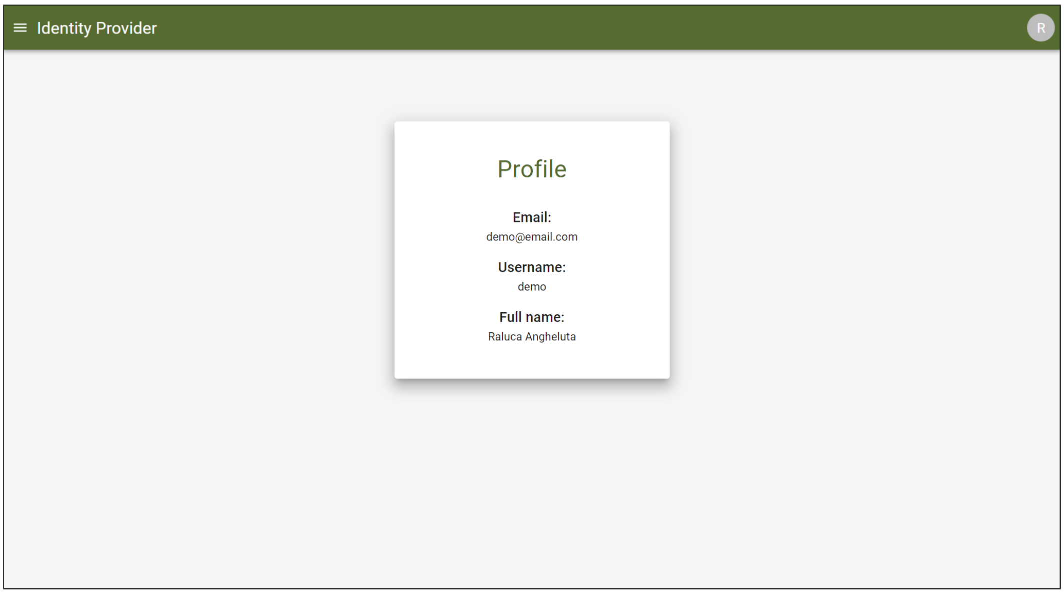

Figure 9: Profile Page

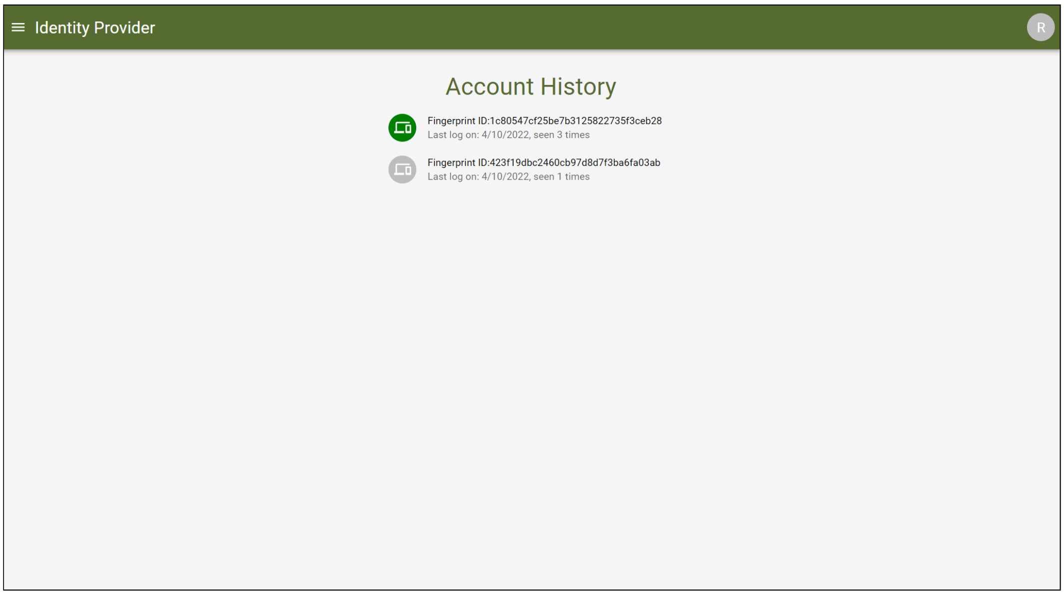

Figure 10: Account History Page

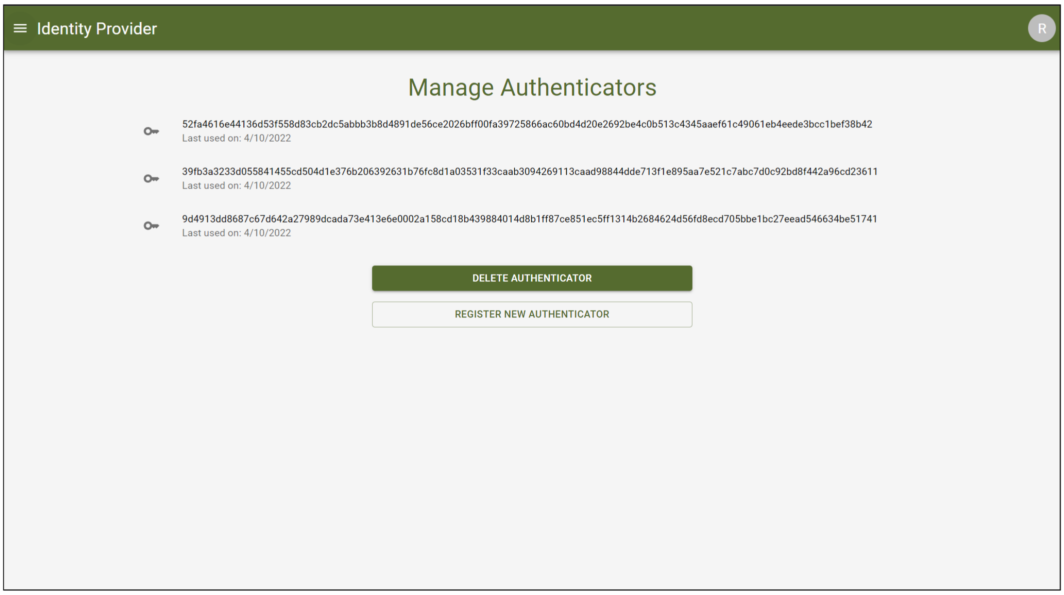

Figure 11: Manage authenticators Page

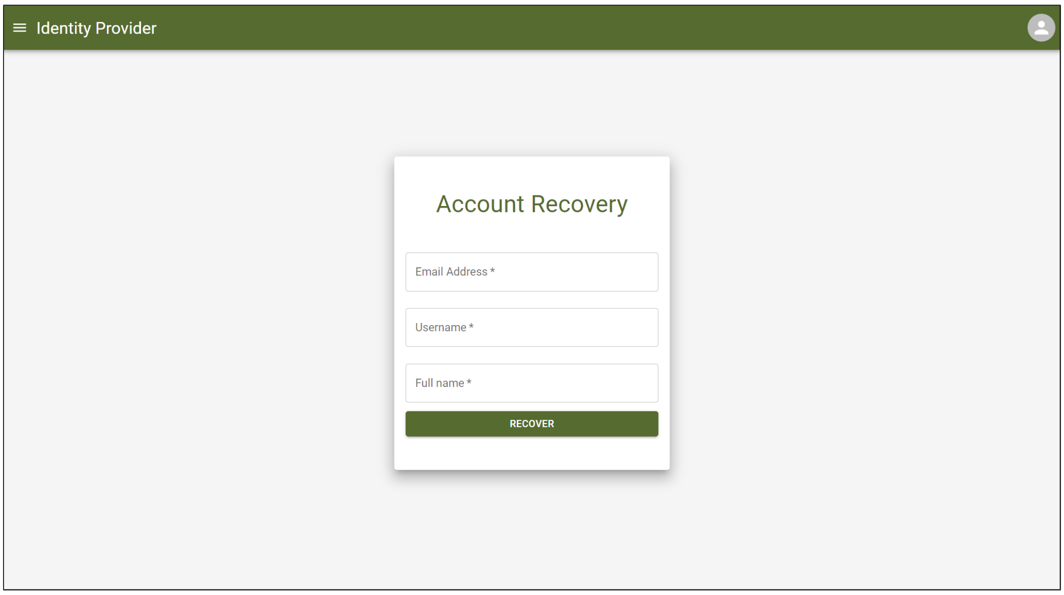

Figure 12: Account Recovery Page

A.2 Service Provider - Dogs As A Service - screenshots

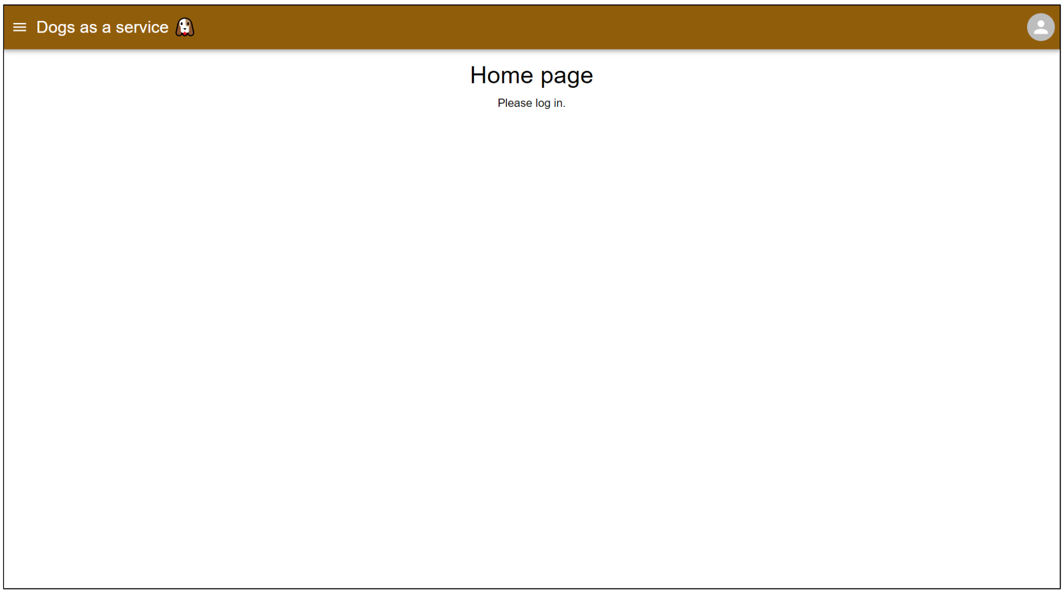

Figure 13: Dogs As A Service - Home Page

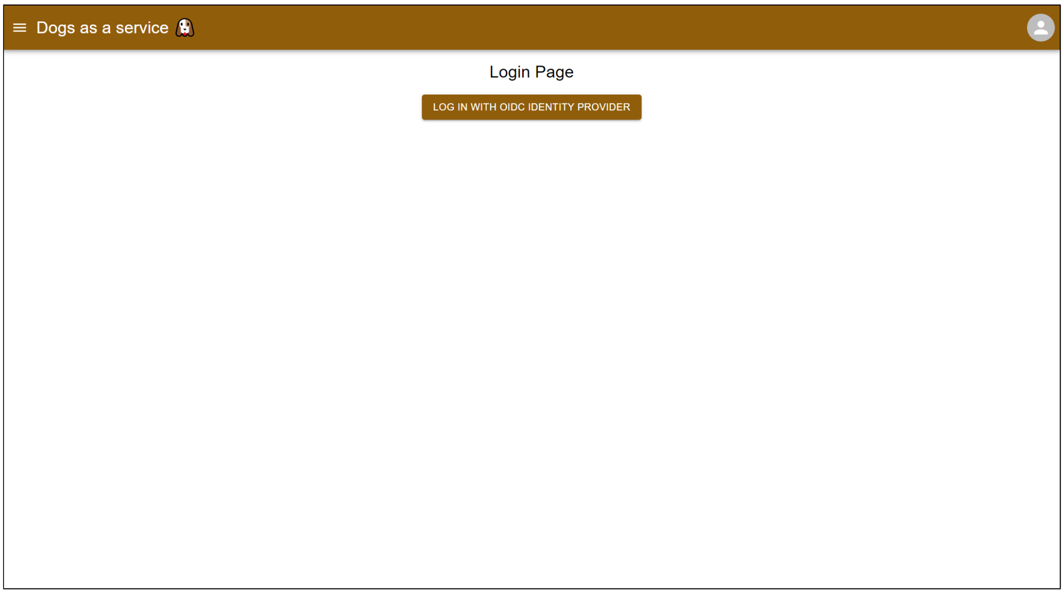

Figure 14: Dogs As A Service - Login Page

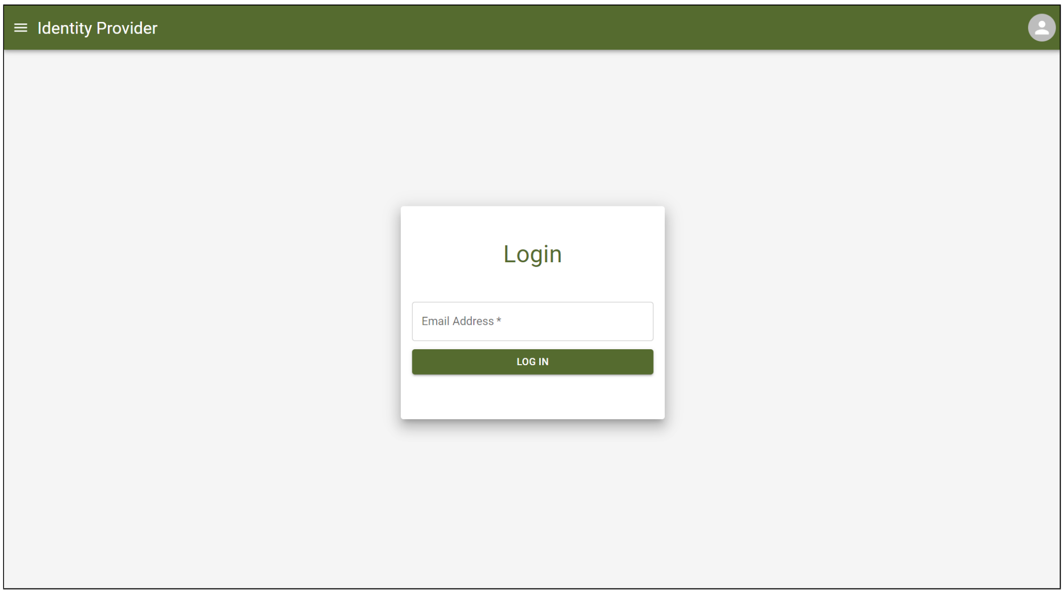

Figure 15: Dogs As A Service - Login Identity Provider Page

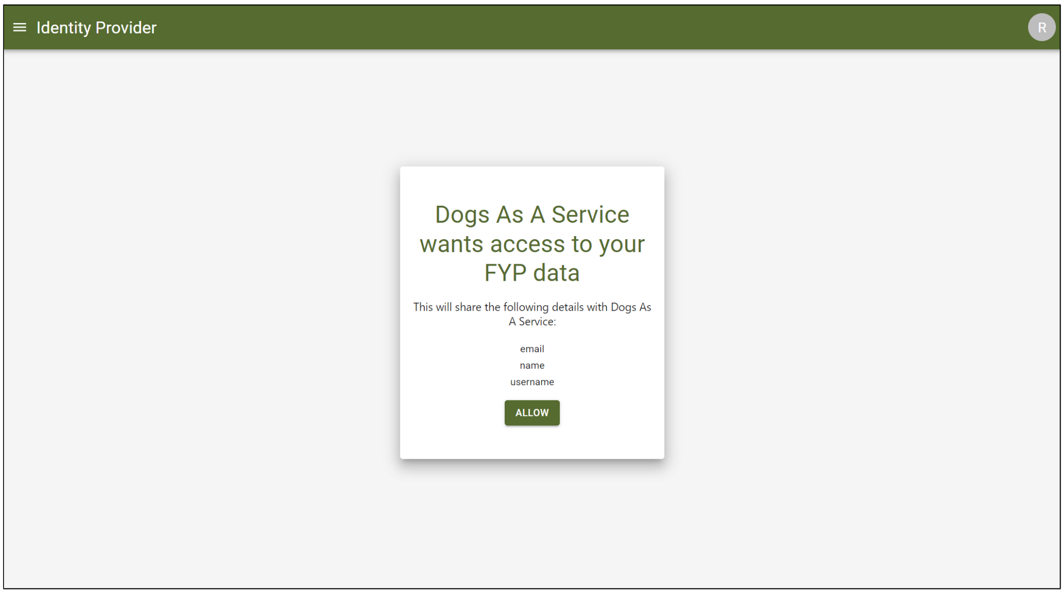

Figure 16: Dogs As A Service - Consent Identity Provider Page

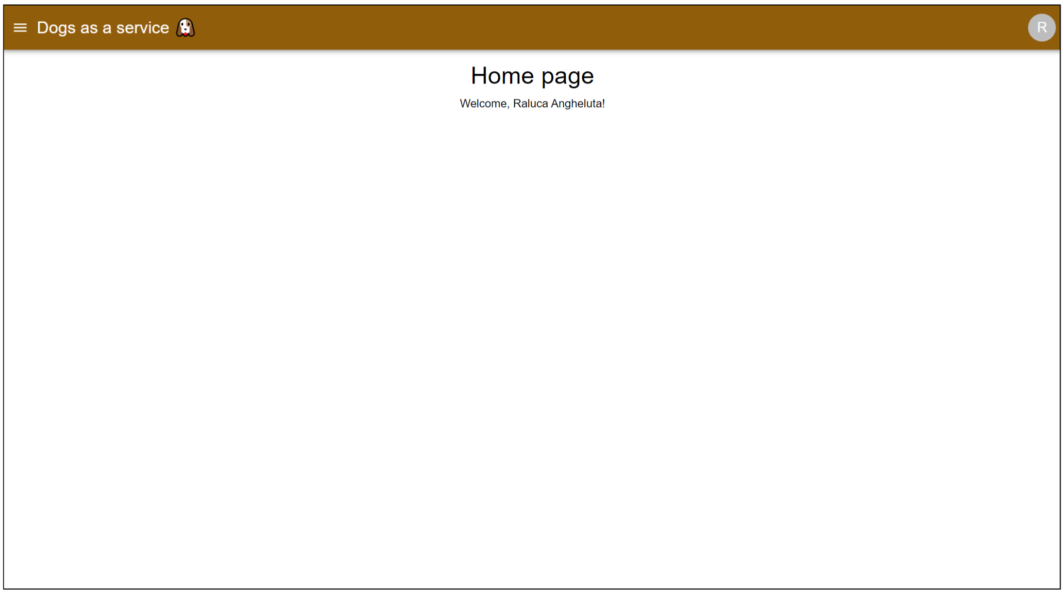

Figure 17: Dogs As A Service - Home Page

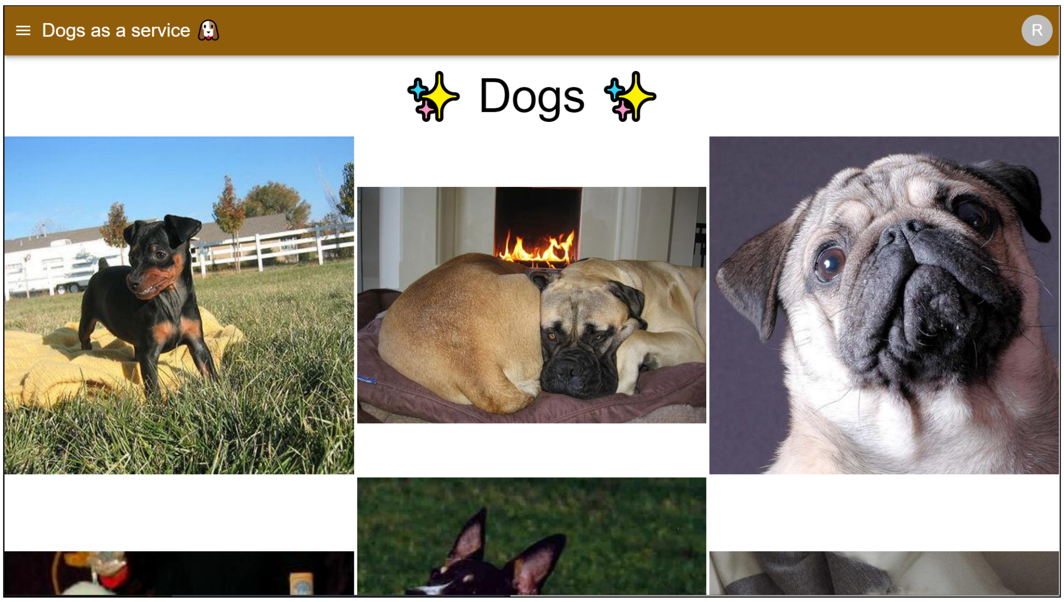

Figure 18: Dogs As A Service - Dogs Page

A.3 Service Provider - Cats As A Service

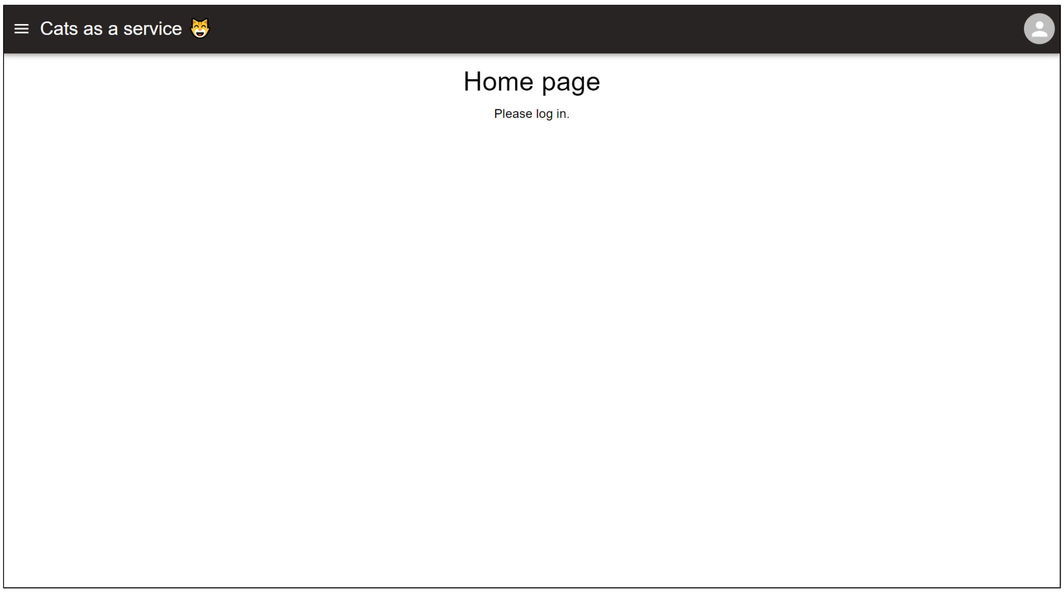

Figure 19: Cats As A Service - Home Page

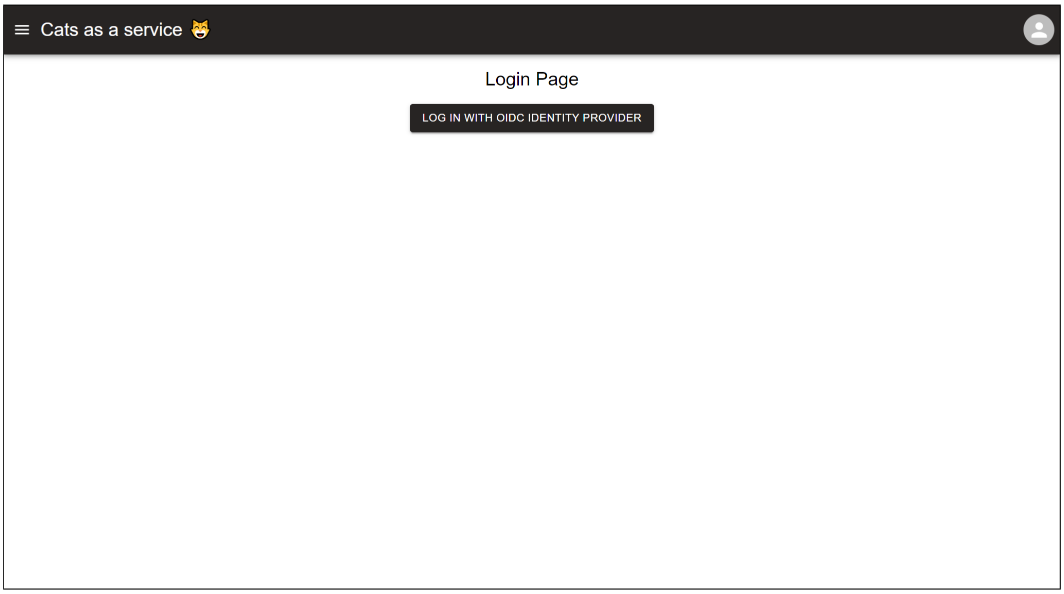

Figure 20: Cats As A Service - Login Page

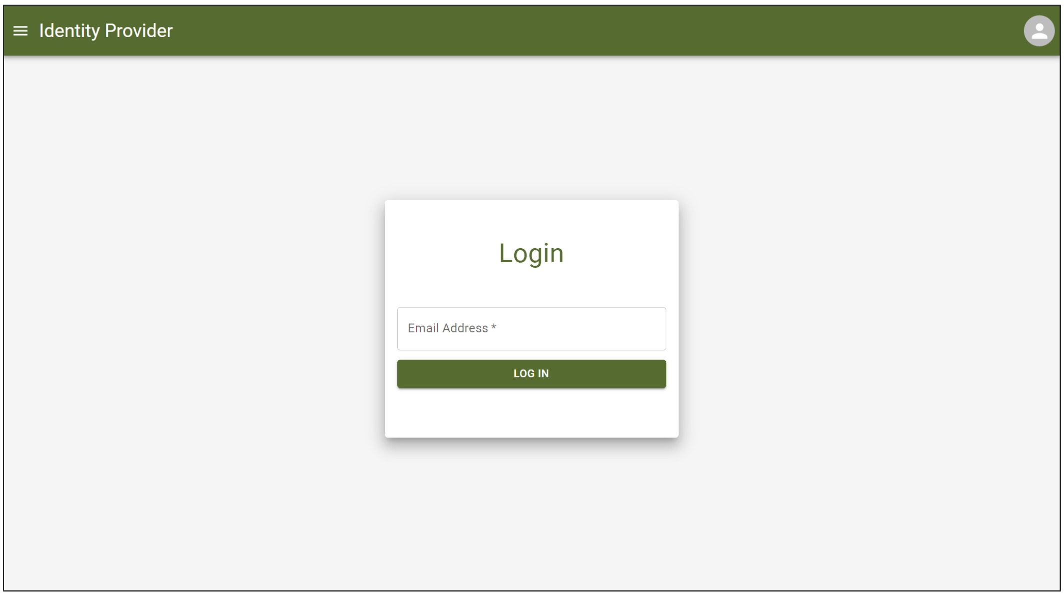

Figure 21: Cats As A Service - Login Identity Provider Page

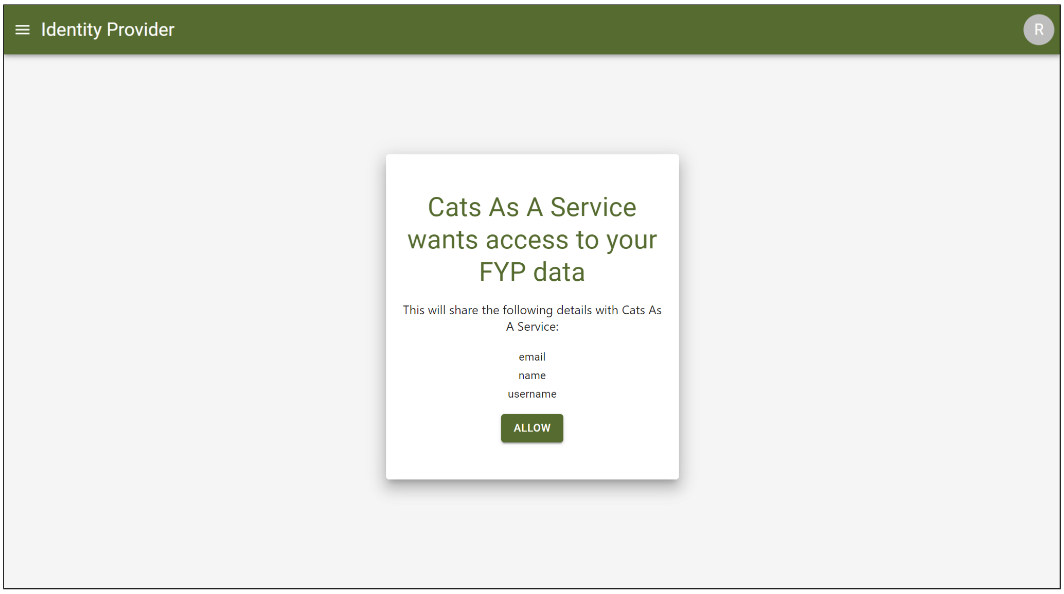

Figure 22: Cats As A Service - Consent Identity Provider Page

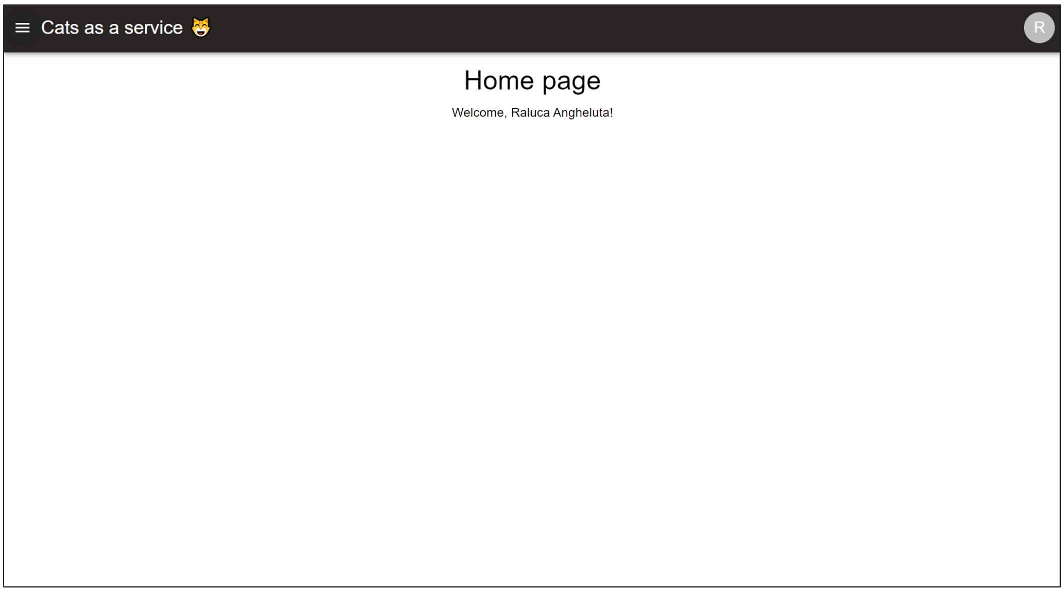

Figure 23: Cats As A Service - Home Page

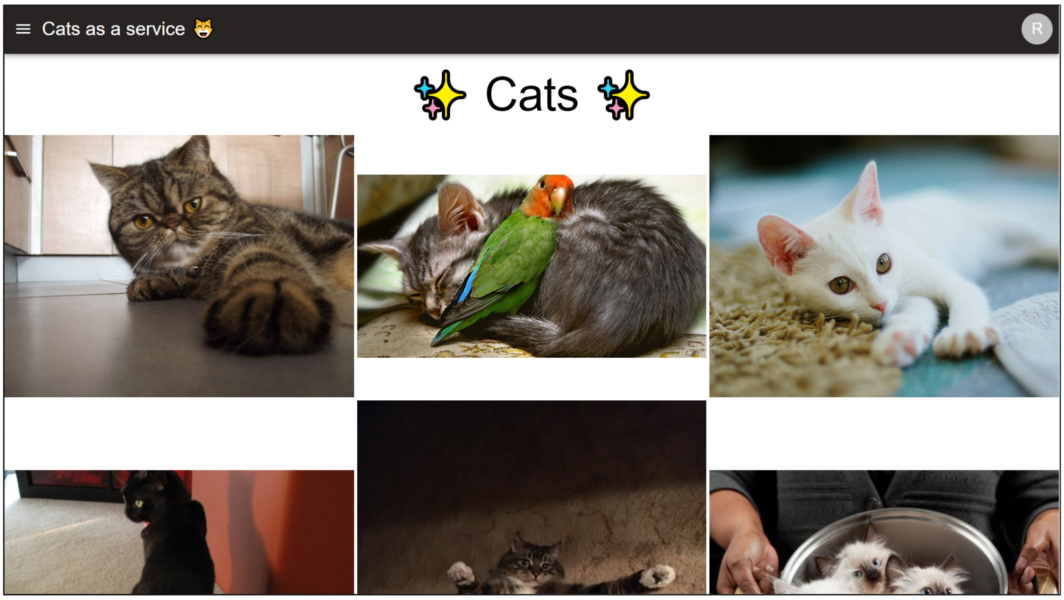

Figure 24: Cats As A Service - Cats Page

A.4 Functional Testing Plan

FYP - Manual Testing plan

<table>
<tr><th>Test index</th><th>FR</th><th>Test name</th><th>Description</th><th>Index</th><th>Test case</th><th>Description</th><th>Steps</th><th>Result</th><th>Overall result</th></tr>
<tr><td rowspan="3">1</td><td rowspan="3">FR1, FR3, FR5</td><td rowspan="3">Register user</td><td rowspan="3">Check the user registration functionality</td><td>1.1</td><td>Successful registration</td><td>Check if user registration is successful when providing a valid email</td><td>1. Navigate to http://localhost:8080/register and verify the following:
- Avatar icon does not feature any initials.
- When clicking on the avatar icon, the menu being present4ed features Register and Login

2. Enter email: email@mail.com, username: email, name: Name Surname
3. Click register
4. Click OK through the security key pop ups and enter the key's PIN
5. In trust device pop up, select yes
5. Verify the following:
- Redirected to http://localhost:8080/
- Name Surname appears in the welcoming message
- Initial N is now present in the avatar image in the corner
- When clicking on the Avatar icon, menu being displayed contains Profile, Account History, Manage Authenticators and Logout

6. In the Avatar menu, select Logout
7. The Login page is now available and the user is successfully logged out</td><td>Passed</td><td rowspan="3">Passed</td></tr>
<tr><td>1.2</td><td>Unsuccessful registration</td><td>Check if user registration is unsuccessful when providing an already registered email</td><td>1. Navigate to http://localhost:8080/register and verify the following:
- Avatar icon does not feature any initials.
- When clicking on the avatar icon, the menu being presented features Register and Login
2. Enter email: email@mail.com, username: email, name: Name Surname
3. Click register
4. Verify the following:
- Error is being displayed stating the user with email email@mail.com is already registered</td><td>Passed</td></tr>
<tr><td>1.3</td><td>Unsuccessful registration</td><td>Check if user registration is unsuccessful when authenticator registration is unsuccessful</td><td>1. Navigate to http://localhost:8080/register and verify the following:
- Avatar icon does not feature any initials.
- When clicking on the avatar icon, the menu being presented features Register and Login

2. Enter email: email@mail.com, username: email, name: Name Surname
3. Click register
4. At any point throught the key's secuirty pop ups, click cancel
5. Verify that an error message is displayed stating the registration has been aborted</td><td>Passed</td></tr>
<tr><td rowspan="3">2</td><td rowspan="3">FR2. FR3, FR5</td><td rowspan="3">Log in user</td><td rowspan="3">Check the user log in functionality</td><td>2.1</td><td>Successful log in</td><td>Check if user log in is successful when providing a correct email/key pair</td><td>1. Navigate to http://localhost:8080/login and verify the following:
- Avatar icon does not feature any initials.
- When clicking on the avatar icon, the menu being presented features Register and Login

2. Enter email: email@mail.com, username: email, name: Name Surname
3. Click register
4. Click OK through the security key pop ups and enter the key's PIN
5. In trust device pop up, select yes
5. Verify the following:
- Redirected to http://localhost:8080/
- Name Surname appears in the welcoming message
- Initial N is now present in the avatar image in the corner
- When clicking on the Avatar icon, menu being displayed contains Profile, Account History, Manage Authenticators and Logout

6. In the Avatar menu, select Logout</td><td>Passed</td><td rowspan="3">Passed</td></tr>
<tr><td>2.2</td><td>Successful login with new fingerprint</td><td>Check if user sees the trust device prompt when logging in from a new browser/device</td><td>From a different browser:

1. Navigate to http://localhost:8080/login and verify the following:
- Avatar icon does not feature any initials.
- When clicking on the avatar icon, the menu being present4ed features Register and Login

2. Enter email: email@mail.com, username: email, name: Name Surname
3. Click register
4. Click OK through the security key pop ups and enter the key's PIN
5. In trust device pop up, select no
5. Verify the following:
- Redirected to http://localhost:8080/
- Name Surname appears in the welcoming message
- Initial N is now present in the avatar image in the corner
- When clicking on the Avatar icon, menu being displayed contains Profile, Account History, Manage Authenticators and Logout

6. In the Avatar menu, select Logout

Repeat the steps above and the trust pop up in section 5 should again be visible.</td><td>Passed</td></tr>
<tr><td>2.3</td><td>Unsuccessful login</td><td>Check if user login is unsuccessful when authenticator verification is unsuccessful</td><td>1. Navigate to http://localhost:8080/login and verify the following:
- Avatar icon does not feature any initials.
- When clicking on the avatar icon, the menu being presented features Register and Login

2. Enter email: email@mail.com, username: email, name: Name Surname
3. Click register
4. At any point through the security key pop ups press cancel.
5. Verify the following:

- Error is being displayed stating the user login has been aborted</td><td>Passed</td></tr>
</table>

3	FR4	Account recovery	Check the account recovery mechanism	3.1	Successful account recovery	Check if account recovery is successful using valid credentials and a trusted browser	On the initial browser (the one that was trusted): 1. Navigate to http://localhost:8080/login and verify the following: - Avatar icon does not feature any initials. - When clicking on the avatar icon, the menu being presented features Register and Login 2. Select 'Lost your key' option 3. Enter email: email@mail.com, username: email, name: Name Surname 4. Click recover account 5. Click OK through the security key pop ups and enter the key's PIN 5. Verify the following: - Redirected to http://localhost:8080/ - Name Surname appears in the welcoming message - Initial N is now present in the avatar image in the corner - When clicking on the Avatar icon, menu being displayed contains Profile, Account History, Manage Authenticators and Logout 6. In the Avatar menu, select Logout	Passed	Passed
				3.2	Unsuccessful account recovery	Check fi account recovery is unsuccessful when providing wrong details	On the initial browser (the one that was trusted): 1. Navigate to http://localhost:8080/login and verify the following: - Avatar icon does not feature any initials. - When clicking on the avatar icon, the menu being presented features Register and Login 2. Select 'Lost your key' option 3. Enter email: email@mail.com, username: email, name: Name Middlename Surname 4. Click recover account 5. Verify that error message is displayed	Passed	
				3.3	Unsuccessful account recovery	Check fi account recovery is unsuccessful when attempting to recover an account from an untrusted device/browser	On the second browser (the one that was not trusted): 1. Navigate to http://localhost:8080/login and verify the following: - Avatar icon does not feature any initials. - When clicking on the avatar icon, the menu being presented features Register and Login 2. Select 'Lost your key' option 3. Enter email: email@mail.com, username: email, name: Name Surname 4. Click recover account 5. Verify that error message is displayed	Passed	
				3.4	Unsuccessful account recovery	Check if account recovery is unsuccessful when authenticator registration fails	On the initial browser (the one that was trusted): 1. Navigate to http://localhost:8080/login and verify the following: - Avatar icon does not feature any initials. - When clicking on the avatar icon, the menu being presented features Register and Login 2. Select 'Lost your key' option 3. Enter email: email@mail.com, username: email, name: Name Surname 4. Click recover account 5. At any point through the security key pop ups click cancel 5. Verify that error message is being displayed	Passed	
4	FR6	New authenticator registration	Check the new authenticator registration functionality	4.1	Successful registration of a new authenticator	Check if a logged in user can register a new authenticator	Log in as detailed in test 1.1. 1. In the user menu, select Manage authenticators 2. Click on Register new authenticator 3. Click OK through the security key pop ups 4. Click on the Logout button in the user menu 5. Log in with the new authenticator as detailed in test 1.1	Passed	Passed
				4.2	Unsuccessful registration of a new authenticator	Check if registration of a new authenticator is unsuccessful when authenticator registration fails	Log in as detailed in test 1.1. 1. In the user menu, select Manage authenticators 2. Click on Register new authenticator 3. Click Cancel at any point through the security key pop ups 4. Verify that error message is being displayed	Passed	
5	FR6	Delete authenticator	Check the authenticator deletion functionality	5.1	Successful deletion of an authenticator	Check if deletion of a chosen authenticator is successful	Log in as detailed in test 1.1. 1. In the user menu, select Manage authenticators 2. Select the authenticator with the last used date the furthest back 3. Click Delete authenticator. 4. Verify that the list of authenticators is updated accordingly. 5. Log out of the account 6. Attempt to log in using the removed authenticator. It should not work	Passed	Passed
				5.2	Unsuccessful deletion of an authenticator	Check if deletion of a chosen authenticator is unsuccessful when the authenticator to be deleted is the only authenticator registered with the account	Log in as detailed in test 1.1. 1. In the user menu, select Manage authenticators 2. Verify that there is only 1 authenticator in the list and select it 3. Click Delete authenticator. 4. Verify that error message is being displayed 5. Log out of the account 6. Attempt to log with the authenticator that was not deleted. It should work	Passed	

6	FR10	Access external service	Check the functionality of external services	6.1	Successful accessing of external service	Check if user can access content of external service when Single Sign-On authentication is successful	1. Navigate to http://localhost:4001 2. In the left hand side menu, select Dogs 3. Validate that page asks for login 4. In the right hand side menu, Click Log in 5. Verify that user is redirected to http:/localhost:8080/... 6. Authenticate with account detailed in test 1.1 7. Click OK through the security key pop ups and enter the key's PIN 8. Give consent when prompted 9. Verify that user is redirected back to http://localhost:4001 10. Accessing the Dogs page now gives access to photos of dogs	Passed	Passed

References

Alaca, F. and P. C. van Oorschot (2016). Device fingerprinting for augmenting web authentication: Classification and analysis of methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, New York, NY, USA, pp. 289–301. Association for Computing Machinery.

Alqubaisi, F., A. S. Wazan, L. Ahmad, and D. W. Chadwick (2020). Should we rush to implement password-less single factor fido2 based authentication? In *2020 12th Annual Undergraduate Research Conference on Applied Computing (URC)*, pp. 1–6.

Auth0 (2015). jsonwebtoken. [Online; accessed 1-April-2022].

Barth, A. (2011). The web origin concept. [Online; accessed 4-April-2022].

Chitra, L. P. and R. Satapathy (2017). Performance comparison and evaluation of node.js and traditional web server (iis). In *2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)*, pp. 1–4.

Dog-API (2022). Dog-api. [Online; accessed 1-April-2022].

Elftmann, P. (2006). Secure alternatives to password-based authentication mechanisms. *Lab. for Dependable Distributed Systems, RWTH Aachen Univ*.

FingerprintJS (2022). Fingerprintjs. [Online; accessed 1-April-2022].

Google (2022). Guidelines. [Online; accessed 29-March-2022].

Grimes, R. A. (2019). 12 ways to hack mfa. [Online; accessed 7-April-2022].

Hanamsagar, A., S. Woo, C. Kanich, and J. Mirkovic (2016). How users choose and reuse passwords. *Information Sciences Institute*.

Jones, M., B. J. and N. Sakimura (2015). Json web token (jwt). [Online; accessed 1-April-2022].

Kunke, J., S. Wiefling, M. Ullmann, and L. L. Iacono (2021). Evaluation of account recovery strategies with fido2-based passwordless authentication. *CoRR abs/2105.12477*.

Laperdrix, P., N. Bielova, B. Baudry, and G. Avoine (2019, 05). Browser fingerprinting: A survey.

Laperdrix, P., W. Rudametkin, and B. Baudry (2016). Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 878–894. IEEE.

Lyastani, S. G., M. Schilling, M. Neumayr, M. Backes, and S. Bugiel (2020). Is fido2 the kingslayer of user authentication? a comparative usability study of fido2 passwordless authentication. In *IEEE Symposium on Security and Privacy*, pp. 268–285.

Miller, M. (2020). Simplewebauthn. [Online; accessed 31-March-2022].

Sakimura, N., J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore (2014). Openid connect core 1.0. *The OpenID Foundation*, S3. [Online; accessed 4-April-2022].

Sakimura, N., J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore (2020). Openid connect basic client implementer's guide. *The OpenID Foundation*, S3. [Online; accessed 4-April-2022].

Sinitsyna, A. (2019). Beyond passwords: Fido2 and webauthn in practice. [Online; accessed 7-April-2022].

Skokan, F. (2015). node-oidc-provider. [Online; accessed 1-April-2022].

Skokan, F. (2016). node-openid-client. [Online; accessed 1-April-2022].

Stackoverflow (2014). Are there any browsers that set the origin header to "null" for privacy-sensitive contexts? [Online; accessed 4-April-2022].

ThatAPICompany (2021). The cat api. [Online; accessed 1-April-2022].

Wikipedia (2022). Single-page application. [Online; accessed 29-March-2022].