UNIVERSITY OF LONDON

ShareAware: A Custom Approach to Federated Identity

A final report based on template **7.1 Project Idea 1: Identity and profile management API**. Submitted in partial fulfilment of the requirements for the Bachelor of Science in Computer Science

of the

University of London

by

Ahmed Alsammarai

Submission Date:	September 2025
GitHub Repositories:	Frontend Backend Toy RP
Live Deployments:	Frontend Application API Documentation Toy RP Demo

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction (883/1000 words)

1.1 Motivation and Real-World Context

Identity is mutable, evolving as a person goes through life in response to shaping experiences and situational requirements. By the time one has reached adulthood, they have amassed many *facets* of their identity, these can include how they present in social, professional or religious contexts. The digital world is very much the same. Individuals interact with many online services and present different facets of their identity depending on the context. A user may, for example, operate under a formal name for work, a nickname for social media and something entirely different for religious or gaming communities. This nuance appears to be overlooked within identity management systems with most only supporting a single facet. This calls into question whether Single-Sign-On (SSO), wherein a user can sign up and log-in using their account with an identity provider, is appropriately named. For example, a user who wishes to use SSO for professional and social media contexts would have to merge those identity facets into one or create multiple accounts.

The existing federated identity protocols OAuth 2.0 and OpenID Connect (OIDC), are difficult to implement securely and do not support context-sensitive identity sharing well. They often expose more data than necessary and lack explicit user consent mechanisms. This project aims to address these limitations by designing a simplified, secure, identity-sharing system designed for multiple contexts.

1.2 Interpretation of Brief

The brief was not specific on how user identity information should be shared. I identified two possible interpretations:

- **Interpretation 1:** A consent-driven model that requires explicit user authorisation each time a third party requests to access identity information.
- **Interpretation 2:** A role-based access model in which some users (e.g., recruiters) have persistent access to the context-specific identities (e.g., professional) of all users who have defined the context.

This project adopted **Interpretation 1**, based on the following justifications:

- Explicit consent aligns closely with the brief's emphasis on handling identities "securely and sensitively" and ensuring that "confidential information should be concealed".
- It allows for the inclusion of private identity contexts (e.g., social) and for simpler additions of more contexts without needing the complexity associated with the role-based model. These contexts are not easily justifiable in the role-based model. For example, the social context would either have to be public (no role required), removing its obscurity from employers, or there would have to be a mechanism by which enquirers (friends) would have to be approved. The latter would lead to a large deviation in the scope towards a social media project.
- It enables stricter adherence to the General Data Protection Regulation (GDPR) through data minimisation and requiring "specific, unambiguous consent". [1]
- It allowed for a simpler and more focused system architecture, avoiding the added complexity of determining what roles there should be, verifying of roles, and persistent permissions.

1.3 Research Problem

While protocols like OAuth 2.0 and OIDC offer robust and theoretically secure mechanisms for authentication and authorisation, they tend to be overly complex, with insecure, privacy-invasive defaults. They provide minimal user control on what data is shared with relying parties (RPs) and when.

This project investigated how a custom identity provider (IdP) could be designed to allow users to set up multiple contextual identities and control which identities are shared with third parties on an explicit, per-request basis. The primary research question is:

> How can an identity system be designed and implemented to support secure, transparent, consent-driven, contextual identity sharing through a simplified SSO-like flow?

1.4 Aims and Objectives

The aim of this project was to design and develop a secure identity API that allowed users to create multiple identity profiles and enabled SSO via user-consented, one-time third-party access. The key objectives were:

- To implement a backend REST API for identity management that supports multiple identity profiles.
- To develop a frontend interface that interacts with the backend API, enabling users to conduct the same tasks through their web browser.
- To design a secure token-based access system that enables scoped sharing of identity data.

- To align the design with GDPR principles such as data minimisation and explicit consent.
- To derive a set of requirements from academic literature and domain knowledge, and design and evaluate the system based on these requirements.

1.5 Scope

The scope of the project, informed by the literature and domain knowledge, was defined as follows.

In Scope:

- Multiple identity contexts.
- Per-identity visibility controls.
- User authentication and session management through JSON Web Tokens (JWTs).
- Consent-based access with scoped, expiring tokens.
- Decoupled frontend/backend architecture.
- Token audit log and revocation.
- Demonstration of the SSO functionality through integration with a toy-relying party.
- Fully responsive, mobile-first design.
- Accessibility through controls (dark mode, link underline toggle), visual design (contrast ratios), and semantics (ARIA tags).
- API rate throttling to mitigate brute-force account compromise and denial-of-service.

Out of Scope:

- Full OAuth 2.0 / OIDC compatibility.
- Role-based access control.
- Full GDPR compliance.
- Third-party registration.
- Multifactor Authentication (MFA) and password reset.
- Field-level visibility configuration of identity fields.

1.6 Report Structure

The remainder of this report is structured as follows.

- **Chapter 2: Literature Review:** A critical evaluation of identity protocols, token formats, and implementation issues in OAuth/OIDC and token-based flows.
- **Chapter 3: Design:** Outlines the system architecture, design decisions, data model, and detailed requirements specification.
- **Chapter 4: Implementation:** A detailed discussion of the implemented system and the developed SSO flow.
- **Chapter 5: Evaluation:** An evaluation of the system against the requirements and analysis of user feedback.
- **Chapter 6: Conclusion:** A summary of the overall project and suggestions for further work.

Chapter 2

Literature Review (1684/2500 words)

2.1 Introduction

Modern web services increasingly rely on delegated authentication and authorisation protocols to manage user identities across platforms. Solutions such as OAuth 2.0 and OpenID Connect (OIDC) have become dominant in enabling SSO and third-party access. These protocols provide formal security guarantees, but their complexity and reliance on correct implementation have led to widespread vulnerabilities privacy failures in real-world deployments.

The analysis is structured thematically, beginning with the challenges of online identity then the disconnect of theoretical security and deployment reality. This is followed with discussion on the erosion of user privacy through weak consent mechanisms and ending with a look at recurring vulnerabilities in token configuration and management. By synthesising these findings, this review aims to establish an urgent need for a simplified, privacy-preserving alternative, and directly informs the design of the ShareAware system.

2.2 The Challenge of Digital Identity and Context Collapse

Research shows that identity depends on the social context. In face-to-face life, people naturally adjust their self-presentation for different audiences. The way someone acts at work is different from how they behave with friends or family. However, online platforms collapse this into a single, undifferentiated audience. Marwick et al. [2] describe this as "context collapse", where users are forced to navigate the conflicting expectations of a networked audience that cannot easily be separated. This increases the risk of information being exposed to unintended groups. While their research provides valuable insight, it is drawn largely from the practices of highly-followed social media users, meaning its generalisability to the average user may be limited. Regardless, the principle that people present differently depending on the context is widely observed and is highly relevant to digital identity.

The principle of "contextual integrity" frames privacy as the maintenance of appropriate flows of information within specific contexts [3]. When contexts collapse, these flows are disrupted, and information intended for one audience may reach another. Federated identity systems often worsen this problem by enforcing a single account across roles, undermining privacy and user agency.

Together, these insights highlight a gap between how identity functions in practice and how identity systems are designed. They provide the user-centred justification for this project: to support multiple contextual identities and allow users to share them selectively, in line with contextual integrity.

2.3 Theoretical Security vs. Deployment Reality

Fett et al. [4] carried out the "first in-depth security analysis of OpenID Connect" using their FKS model of the Web. This included the 3 modes which the OIDC defines as "authorisation code mode, the implicit mode and the hybrid mode". They showed that OIDC, when strictly following best practices, can resist a wide range of attacks including IdP mix-up, cross-site request forgery (CSRF), and server-side request forgery (SSRF). They provided a set of security guidelines that are formally proven to mitigate attack classes, such as explicit user intention tracking via sessions and ensuring use of Transport Layer Security (TLS) via HTTPS. However, their model made several simplifying assumptions:

1. Tokens never expire.
2. No browser extensions were used.
3. Implementation follows best practices perfectly.

These choices mean that their results demonstrate theoretical security but are not necessarily representative of real-world deployments.

In contrast, Westers et al. [5] examined OIDC in practice. They conducted a large-scale empirical investigation of OIDC based SSO implementations, focusing on the privacy risks. Using a dataset of the top one million websites, they detected:

1. 11,189 partial leaks, which allow IdPs to track their users across sites.
2. 2,947 full leaks in which RPs receive a user's identity data without consent.
3. 6 escalated leaks where third parties obtain the identity data.

The research clarifies these leaks as those occurring without the user using the SSO or intending to login. [5]. The leaks occur because the SSO IdP session is alive via cookies, meaning that even if a user hasn't visited an RP in months and has cleared their browsing history, the RP may still initiate a hidden request to the IdP. Importantly, the majority of the leaks were traced to official Software Development Kits (SDKs) provided by the IdPs, notably Facebook's and Google's, which were among the worst. The method was conservative as cookies were rejected. These cookies can sometimes enable third-party tracking functionalities so it is likely that they reported only a lower bound.

Philippaerts et al. [6] further highlight implementation weaknesses by evaluating eight popular and maintained open source IdP middleware that implement the OAuth 2.0 protocol, evaluating each against the OAuth best current practice (BCP). They found vulnerabilities in seven of them, including weak redirect URI validation, reuse of single-use tokens, and poor refresh token handling. These issues occurred despite the researchers carefully configuring them. This shows that even secure setups are flawed. This is likely an underestimate of the true risks as they did not test the default configurations.

These studies reveal a consistent pattern. The theoretical security of these protocols does not mean their deployments are secure. This appears to be due to complexity, flawed SDKs, and poor implementation. This project deliberately avoids using third-party middleware and instead opts for a simplified, custom approach.

2.4 Consent, Privacy, and User Control

A recurring weakness identified across the literature is the enforcement of user consent. Fett et al. [4] included a recommendation to track the user's intention to improve this, but this is their guideline rather than built into the OIDC protocol. As a result, adoption is probably limited, and many real-world deployments still treat consent as implicit or persistent.

Westers et al. [5] provided stronger evidence of the scale of this problem. They showed that identity data can be leaked automatically, without the user performing any deliberate action, even if a user has not actively logged into a relying party. This behaviour directly contradicts the GDPR requirement for "specific, unambiguous consent" [1]. They found that EU-hosted sites, which are legally bound by GDPR, were over-represented in the leaks. This suggests that regulation is not enough to prevent the problem. As discussed, the choice of Westers et al. [5] to not include cookies acceptance or browser extensions, which can be compromised, likely leads to any extrapolation of the scale of the problem to be conservative.

It's evident that neither the protocols nor regulations truly ensure genuine consent in practice. This project responds to that gap by making consent explicit, per-request, and time-limited, preventing identity data from being shared without deliberate user approval.

2.5 Vulnerabilities in Token Management and Cryptography

Identity data is often transmitted and verified through tokens, the security of which depends heavily on design choices and configuration. Nugraha et al. [7] explored the comparative performance and security of JWTs and Platform-Agnostic Security tokens (PASETO). Their study was motivated by the increasing frequency of attacks on REST APIs. Their methodology involved testing the performance metrics (token generation time, transfer time, and size). They also conducted security testing on the top three API Open Worldwide Application Security Project (OWASP) vulnerabilities (Broken Object Level authorisation, Broken User Authentication and

Excessive Data Exposure). PASETO showed consistently stronger resilience against the OWASP defined threats, especially in scenarios involving algorithm spoofing and payload tampering, where JWT appeared vulnerable. Conversely, JWTs appeared significantly more performant, as they were approximately five times faster on token generation and approximately twice as fast on transfer time. However, their study tested JWT only in its HMAC (a symmetric cryptography algorithm) configuration, which is considered weaker and less common in production environments. They did not include RSA-signed JWTs (RS256), which are widely deployed in OAuth and OIDC ecosystems. By omitting this configuration, the study's findings on performance and security trade-offs are of limited practical relevance.

Fett et al. [4] and Philippaerts et al. [6] also reported recurring problems in token management. These included reuse of single-use tokens and refresh token mismanagement. Both studies used careful and security focused configurations and therefore likely underestimate the extent of token-related risks in real-world systems.

It is clear token design and management are frequent sources of vulnerability, and can undermine the entire security model. To address this, this project adopts RS256-signed JWTs, which benefit from asymmetric cryptography enabling detection of tampering and validation whilst being familiar to developers integrating the system. Additionally, tokens for identity sharing are not refreshable and expire after 5 minutes, avoiding persistent access. Finally, sharing tokens are single-use and are logged and marked as used to directly mitigate token reuse.

2.6 Synthesis of Literature

Several common themes emerge from the literature that justify this project. Firstly, there is a clear misalignment between how self-representation functions in the real world and how it is represented in current SSO systems. Research in online identity shows that people manage different facets of self across contexts, and that context collapsing is a privacy concern [2, 3]. Despite this, established SSO providers collapse the contexts into a single entity, forcing either the merging of identities or the creation of multiple accounts.

Second, there is a clear gap between the design of the protocols and the reality of their implementations. Fett et al. [4] present OIDC as theoretically secure but acknowledge that security guidelines are sometimes unfeasible in real-world deployments. This concern is validated by Westers et al. [5] who identified thousands of privacy leaks in OIDC deployments caused by implementation and SDKs flaws. Philippaerts et al. [6] similarly found that seven out of eight OAuth 2.0 middleware solutions had vulnerabilities despite being configured using BCP.

Third, the studies reveal a consistent lack of consent enforcement. Fett et al. [4]suggest implementing user intention tracking among their guidelines. Meanwhile, Westers et al. [5] show that many systems leak information automatically, without explicit consent or user interaction, and suggest a browser extension as a possible solution. In both cases, the fact that this is a gap in the protocol design itself seems to be overlooked. Both user intention tracking and the suggested browser extension become unnecessary within a protocol that places user consent at its core. Moreover,

the studies tend to underestimate the scale of the issue, which does not seem to be helped by GDPR either.

Finally, token management is a consistent point of weakness that highlights that security at the protocol level is not enough if the tokens themselves are poorly configured or managed.

These themes justify the approach taken in this project, which focuses on an alternative model, with consent at its core. This model is simplified, eliminating the need for complex third-party middleware. The project design directly responds to the recurring themes identified here.

Chapter 3

Design (1659/2000 words)

3.1 Introduction to Design Section

The project is called ShareAware. Its design has been based on specific requirements supported by research with a focus on security and privacy. The design delivers a secure, context-aware consent-driven identity system, implemented using a modern decoupled architecture.

3.2 Domain and Users

The project sits within the domain of identity and profile management. It enables users to present different facets of their identity depending on context, mirroring real-world behaviour [2]. Most SSO IdPs assume either a single identity reused across services or that users will create multiple accounts.

This system is designed for two main user groups:

- **Primary users (identity owners):** Individuals who want to manage different identity profiles in one place, rather than fragmented across services or accounts. They want the convenience of a single solution that clearly shows what data is shared, when, and with whom.
- **Third-party services (relying parties):** Platforms such as job sites, e-learning tools, or forums that want secure SSO functionality.

Example User A

Alice, a developer applying for jobs, wants to sign-up for multiple job boards quickly, sharing her professional name, bio, photo, and work email, without sharing her personal email or social media photo. This system lets her share only her professional identity and sync updates each time she logs in.

Example User B

Bruce is a CEO by day and a crime-fighting influencer (social identity) "Batman" by night. He wants one account that strictly separates his personas. This system

lets Bruce manage both personas in one place and choose which to share with each service. He needs guardrails against accidental sharing, which the system provides through consent and visibility flags.

Example User C

Charlie runs an online business. He wants simple, low-risk SSO as he's worried about GDPR liability. Charlie feels this system is easier to understand and would allow him to reduce his risk, without forcing him to give up analytics or tracking.

Context-specific, consent-driven sharing helps users maintain natural boundaries between different parts of their lives. It also helps RPs by adding a privacy-respecting SSO option that reduces their risks compared to traditional IdPs.

3.3 System Overview

The ShareAware system is designed to support contextual identity management and secure consent-driven identity sharing to third-party services. It uses a decoupled architecture consisting of a frontend Single-Page Application (SPA) built with React and a backend API implemented using the Django REST Framework (DRF).

Users can interact with the system through the SPA to manage identities and authorise third-party access when needed. The access is one-time and short-lived (five minutes). The users can revoke access at any point prior to expiry. Third-party services (Relying Parties, RPs) can integrate the system easily via a simplified SSO flow that lets them request identity access in a privacy-respecting manner.

To initiate SSO, the RP redirects the user to the SPA. The system prompts the user to authenticate if necessary. The system then displays the user's shareable identities, showing the exact profile data that would be shared for each. The user is prompted to select the identity they wish to use for SSO and approve access to the displayed data. If access is consented to, a signed JWT containing a reference to the identity is generated and POSTed to the callback URI. The RP validates the token and then depending on whether the user already exists in their system does the following:

- If the user exists: log them in and optionally use the token to fetch updated profile data.
- If the user does not exist: fetch identity data using the token, pre-fill a sign-up form if fields are missing/unsuitable, and then create and log in the user.

A high-level overview of this interaction is shown in Figure 3.1. For a more in-depth look, see section 4.4. All access is logged and can be audited or revoked. This system supports privacy by design, explicit consent, and GDPR aligned data sharing in a simplified OAuth and OIDC like flow that avoids many of the risks present in those protocols.

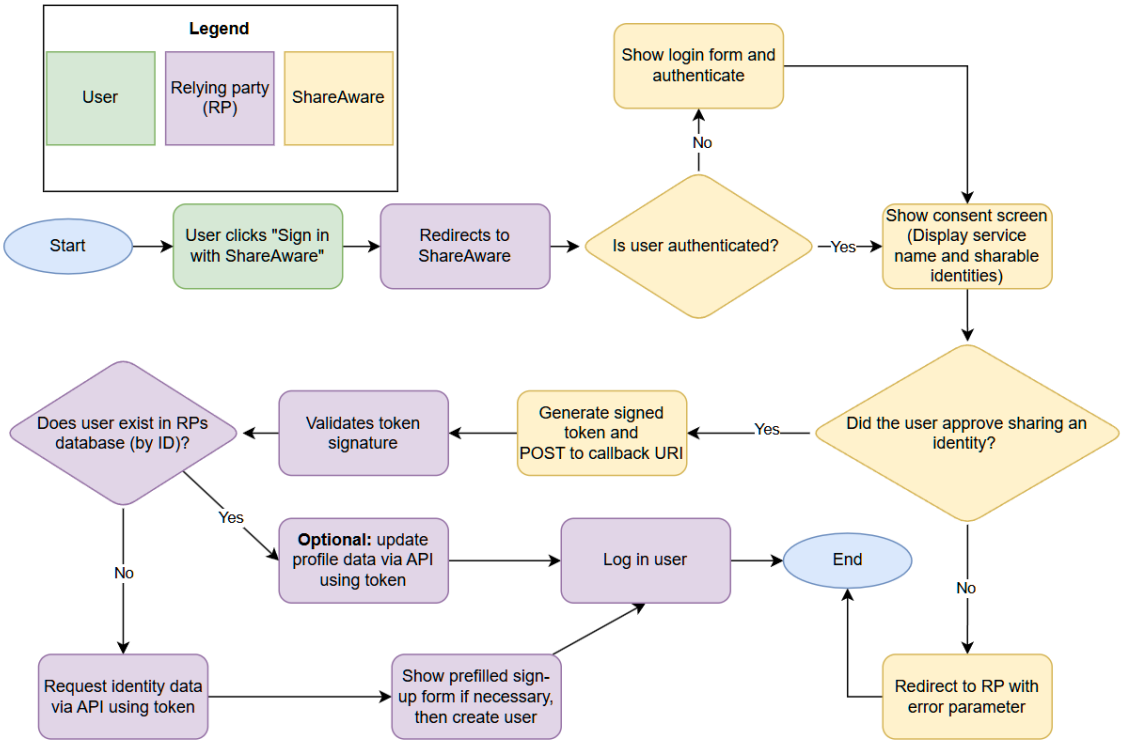


Figure 3.1: Consent-based contextual identity sharing flow between IdP and RP.

3.4 Functional Requirements and Justification

The functional requirements are grounded in the interpretation of the project brief, insights from the literature review, user feedback, and domain knowledge.

Core Functional Requirements

Table 3.1 outlines the core functionality required for secure identity and session management within the system.

Table 3.1: Core Functional Requirements.

ID	**Detail**
R-Core-1	The system shall allow users to register and login securely. Upon login, a refreshable, short-lived (5 minutes) simple JWT will be issued for session management. Justification: Enables authentication.
R-Core-2	The system shall allow users to create, edit, and delete identity profiles with a context label chosen from the set {professional, job-seeking, social, personal, gaming, other}. The users should be able to create multiple identities per context and use a label field to differentiate them. Justification: Enables context-specific identity representation. The predefined set simplifies RP flow.
R-Core-3	Each context will freely allow the user to provide, or not provide the following details in these formats: • name (max 255 characters) • gender (max 100 characters) • sexuality (max 100 characters) • relationship status (max 255 characters) • biography (text) • profile picture (image) • email address: (email) Justification: Supports diverse personal expression.
R-Core-4	Each identity shall have a visibility flag {`Shareable`, `Unshareable`} that defines access permissions. Justification: Allows the user to set their privacy preferences at a per-context level. Serves as the first layer of defence as the flag is checked before identity is shared even with a valid token.

Consent and Token-based Access

Table 3.2 describes the requirements for obtaining consent and managing third-party access.

Table 3.2: Consent and Token-Based Access Requirements.

ID	Detail
R-Share-1	The system shall implement a consent-based authorisation flow for third-party access to identity data. Justification: Directly addresses the un-consented identity leaks observed by Westers et al. [5]. Inspired by their browser extension that detects SSO and prompts the user for consent, but built directly into the protocol, avoiding requiring user awareness.
R-Share-2	The system shall provide a UI allowing the user to select an identity to share and see exactly what data (keys, values) will be shared with the RP as parts of the consent prompt. Justification: Enables explicit consent. Fulfils GDPR requirement for specific informed consent and transparency by ensuring users understand exactly what data is being shared.
R-Share-3	A single-use **non-refreshable** JWT shall be issued to the RP upon consent, scoped to a single identity, and cryptographically signed using RS256 (RSA + SHA256). This JWT will contain the following fields: • `user_id`: Internal primary key • `your_service_user_id`: Serves as the users identifier in the RP's database • `identity_id`: The internal identity identifier • `exp`: Expiry time (5 mins from issuance) • `callback_uri`: The provided endpoint for token delivery • `redirect_uri`: The provided RP UI page to redirect the user to after consent. • `jti`: Internal primary key used for tracking whether a token has been used Justification: The RS256 signature assures integrity and authenticity. jti avoids replay attacks [6]. The `user_id, jti, exp` and a used boolean are stored as a record in a token log. This can then be filtered and displayed to the user with the option to revoke access. The token log acts as documentary evidence of consent as per GDPR consent rules [1]
R-Share-4	Third parties shall receive the access tokens via a POST request over **HTTPS** to their callback URI. Justification: Avoids token leakage via referrer headers [5]
R-Share-5	Third parties shall retrieve identity data via a dedicated endpoint with a valid consent-issued JWT. Justification: Using a stateless **read-only** endpoint ensures that there is no permission escalation and the data is only accessible to clients with an authorised token. This aligns with the principle of least privilege and auditability.

Audit and Revocation

Table 3.3 describes features that allow users to track and revoke third-party access.

Table 3.3: Audit and Revocation Requirements.

ID	Detail
R-Aud-1	The system shall expose a user-facing log of previous access grants accessible through the front end and via API. Justification: Provides the users a summary of who has access to their data. Enables revocation.
R-Aud-2	The system shall support manual revocation of tokens using the API and UI button. Justification: Allowing users to audit and revoke access aligns with GDPR and project data privacy goals.
R-Aud-3	The system shall create an immutable snapshot of the exact identity data the agreed to share at the time of consent. Justification: Creates a definitive audit trail of exactly what was shared and when.

Account Management

Table 3.4 describes the requirement for managing user accounts data.

Table 3.4: Account Management Requirement.

ID	Detail
R-Acc-1	The system shall provide a UI to manage account-level data, including changing their email or password and deleting the account. Justification: GDPR alignment.

3.5 Security and Privacy Design

This system is designed with privacy, security and user consent at its core. It follows applicable best practices from OIDC and the laws of the GDPR whilst avoiding many of the risks associated with OIDC, Oauth 2.0 and middleware.

Security Principles

The following are the core security principles of the system:

- **Explicit Consent:** No persistent access and no sharing without explicit user approval each time.
- **Token Scope and Time-To-Live (TTL):** Tokens are scoped to a single identity instance and expire in five minutes, reducing the time available for misuse.

- **Single-Use Tokens:** Tokens are marked as used upon access mitigating replay attacks.
- **Stateless Authentication:** No cookies or server-side sessions. Purely stateless authentication.
- **Layered defences:** Each identity includes a visibility flag, preventing accidental sharing.
- **Safe Transmission:** Tokens are sent via POST to the callback URI, preventing leakage via URL fragments or referrer headers.

Threat Mitigation Mapping

Table 3.5 outlines key threats identified in the literature and how the design addresses them.

Threat	**Mitigation**
Partial Leaks [5]	Requests require consent via a redirection to the SPA. Partial Leaks are not possible.
Full Leaks [5]	The consent model again avoids this as the RP cannot receive claims without user action.
Escalated Leaks [5]	Tokens are short-lived, context-scoped, and non-refreshable, limiting exposure if compromised. This is anyway avoided as token transmission is over POST rather than by URI fragment and redirection.
Token Replay [6]	Tokens are single-use and have a short TTL, mitigating, but not wholly avoiding Replay attacks. Token transmission is conducted securely to reduce risk.
Algorithm Spoofing [7]	Tokens are signed using asymmetric cryptography, modifications are easily detected and rejected.
Other Middleware Flaws [6]	No external OAuth middleware is used. The authentication and authorisation is handled in a custom manner internally, removing the dependency on the security of third-party implementations.

Table 3.5: Security threat mitigation through design.

GDPR Alignment

Full GDPR compliance is out of scope, however the system was specifically designed to align with some GDPR principles. These and how the design achieves them are Table 3.6.

Table 3.6: GDPR Principles and System Alignment.

GDPR Principle	System Implementation
Consent (Art. 7 [8])	Users must explicitly grant consent for each data sharing instance. The consent prompt shows exactly what fields will be shared.
Data Minimisation (Art. 5(1)(c))[8]	Only the shared identity data is disclosed, and all fields are optional. Users can omit fields entirely if they choose.
Transparency (Art. 13[8])	Before granting access, users are informed which data fields (keys and values) will be shared, with whom, and for what purpose.
Rights to Access and Withdraw (Art. 15, 17[8])	Users can view past consents in an audit log and revoke access at any time.
Rights to rectification and Erasure (Art. 5(2)[8])	Users can update and delete contextual identity data freely. They may also update or delete their account information.

Design Limitations and Known Trade-offs

The system presents a secure and privacy-respecting identity sharing flow, but limitations and vulnerabilities remain. These are acknowledged below.

No RP Verification and Registration

The system does not require RPs to register or be validated, meaning that anyone with a callback URI can make claims. This simplifies the implementation, but allows for attack vectors such as phishing.

Callback URI Validation

Pre-registration of RP URIs was omitted for simplicity. This could be mitigated by comparing the URI base domains, but this was avoided as many services have different frontend/backend domains.

Optional Identity Fields and Over-disclosure

The optionality of the identity fields was a deliberate design decision. Including more fields improves compatibility with registration forms. Conversely, this can lead to users supplying more data needed (e.g., if the registration form lacks gender, but they provide it when they grant access). This somewhat violates the GDPR principle of minimisation. I decided that optionality lets users choose between privacy and convenience.

Custom Integration

Due to not using the full OIDC OAuth2.0 workflow, RPs integration requires more custom code rather than existing middleware or SDKs. This might be a pain point. The frontend includes integration documentation explaining how to do this.

3.6 System Architecture

The project is built using a fully decoupled architecture:

- Frontend: React SPA.
  - React Router (routing).
  - Chakra UI (accessible, responsive styling).
  - A custom fetch API wrapper to automatically handle the attachment and refreshing of auth tokens.
- Backend: Django + Django REST Framework (DRF).
  - DRF generic class-based views (CRUD) and serializers for validation/escaping.
  - DRF-Spectacular: OpenAPI schema and interactive Swagger docs.
  - Request throttling: per-endpoint classes for registration/login and global user/anon limits.
- Communication: JSON requests over HTTPS between the SPA and the REST API.
- Authentication and Authorisation:
  - SPA sessions: DRF-SimpleJWT for access + refresh tokens.
  - Third-party sharing: Scoped, single-use, non-refreshable, time-limited (5 minutes) RS256-signed JWTs (RSA with SHA-256).
- Database:
  - PostgreSQL (production).
  - Core models: `User`, `Identity`, `TokenLog`, `IdentitySnapshot`.
- Security:
  - Stateless auth (no cookies therefore no CSRF) and serializers enforce input validation.
  - HTTPS enforced end-to-end.
  - Audit logging: token issuance/consumption recorded in `TokenLog` with revocation support.
  - Key management: RSA private key via environment variables and public key served via endpoint.

3.7 Database Design

The database consists of four core models. Each is detailed below along with its entity-relationship diagram (ERD). A complete ERD is provided in Figure 3.6.

User Model

This model is shown in Figure 3.2 and extends Django's built-in `AbstractUser` class, setting email as required and enforced to be unique. All other fields are the inherited defaults. Only name, email, username, and password are required.

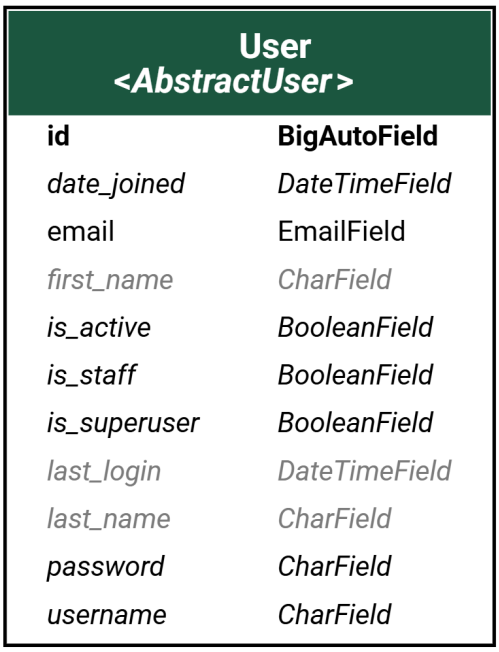


Figure 3.2: User model schema.

Identity Model

The `Identity` model shown in Figure 3.3 represents contextual identity. Users can create as many identities per context as they would like. This model holds optional profile data along with visibility flags.

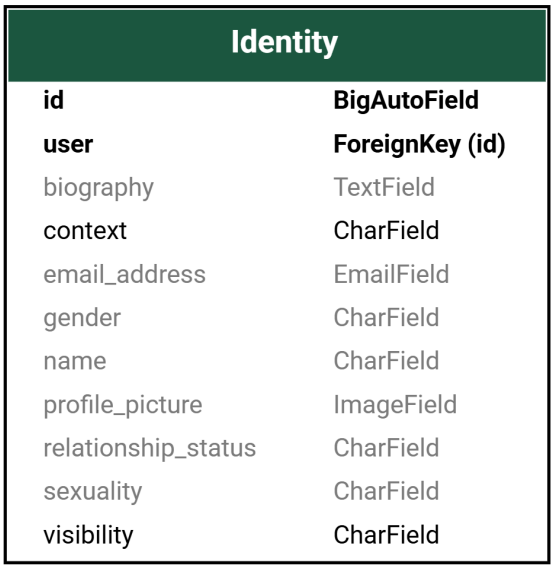


Figure 3.3: Identity model schema.

Key Fields:

- `user`: Foreign key linking the identity to its owner.

- `context`: Choice field (select from). Can be one of the contexts defined in R-Core-2.
- `label`: User-defined label to differentiate multiple identities with the same context.
- `name, biography, email_address, profile_picture`: Profile data which is optional to support data minimisation.
- `visibility`: A choice field (`shareable` or `unshareable`) that controls whether an identity is shareable, preventing accidental disclosure.

TokenLog Model

The model records data for each consented token issuance. Including context, URIs, and flags, all captured at the moment of consent, meaning that if the user later deletes their identity, the log retains the original record. This denormalises, as context could have been referenced, but this was intentional to improve auditability.

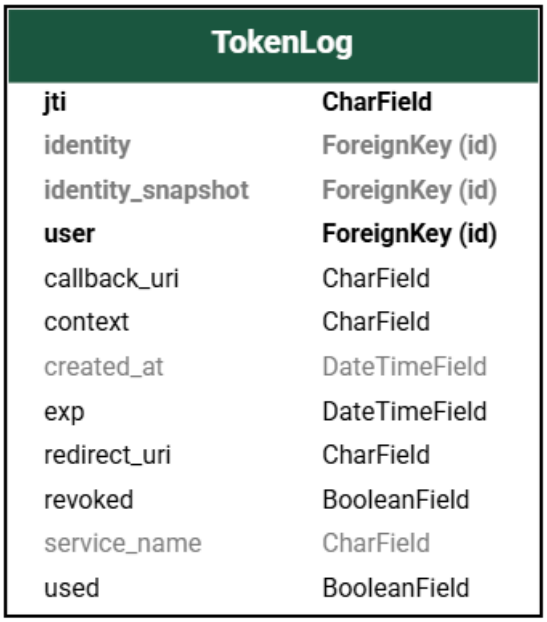


Figure 3.4: TokenLog model schema.

Key Fields:

- `jti`: JWT ID used as a primary key.
- `identity`: A foreign key to the identity related to the record.
- `identity_snapshot`: A foreign key linking the record to the immutable record of data for which sharing was consented.
- `service_name`: The name of the third-party service.
- `context, callback_uri, redirect_uri, created_at`: Metadata stored at **time of consent.**

- `used, revoked`: Flags used to prevent reuse and enable revocation.
- `exp`: Token expiration timestamp for auditing.

IdentitySnapshot Model

This model creates an immutable record of an identity at the exact moment a user consents to share it. The data is stored as a JSON object and is linked to the `TokenLog` model to provide an audit trail of the information disclosed. This ensures that even if the identity is modified or deleted, the record of what was shared remains.

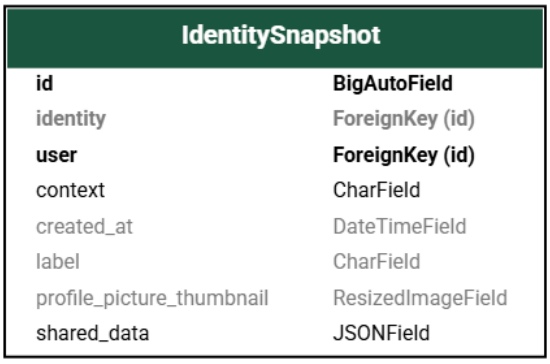


Figure 3.5: IdentitySnapshot model schema.

Key Fields:

- `identity`: A nullable foreign key to the original shared `Identity`.
- `shared_data`: A JSON field containing an exact copy of the data fields that was shared with the third party.
- `profile_picture_thumbnail`: A resized image field storing a small thumbnail of the profile picture to preserve the record without repeatedly storing the full size image.

Full ERD Overview

The complete ERD is shown in Figure 3.6.

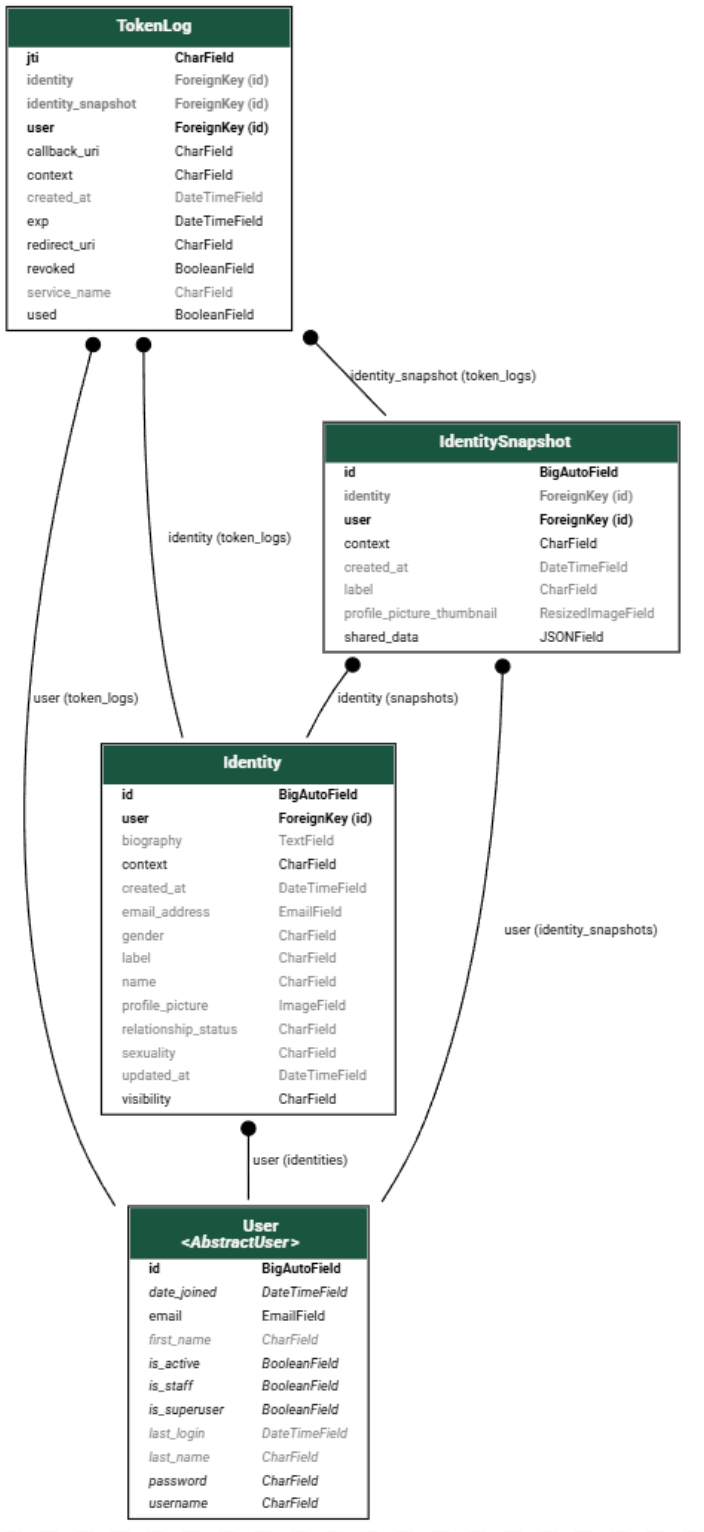


Figure 3.6: Full database ERD with relationships.

3.8 Site Structure

Figure 3.7 summarises the designed sitemap.

Table 3.7: Frontend Route Sitemap.

Page	**Route URL**	**Purpose**
Home	`/`	Landing page acting as a marketing page for unauthenticated users.
Login	`/login`	Login form with next parameter preservation.
Register	`/register`	Registration form.
Dashboard	`/dashboard`	Main authenticated area, overview of identities as editable (update, delete) cards and an option to add more identities.
Consent Prompt	`/consent`	Shown to user when third-party requests access to a specific contextual identity. Lets the user choose an identity to share. Displays requested data and offers explicit accept/reject buttons and redirects back to the RP accordingly.
Audit Log	`/audit-log`	Shows previously granted accesses and gives the option to revoke the access tokens.
Snapshot	`/snapshot/<id>`	Shows a snapshot of an identity at the time of consent.
User documentation	`/docs/user-guide`	Explains core functionalities to users: registration, identity management, visibility, access grants, auditing, revocation and account management
Developer documentation	`/docs/dev-guide`	Explains integration. Includes flow diagram, how to redirect users for consent, how to verify and consume the tokens.
Demo	`/docs/demo`	Explains how a user or a developer can use the Toy RP to simulate the full consent flow.
Account Management	`/account`	Enables users to change their account email and password or optionally to delete their account.
404 Not Found	`*`	Catch-all route for undefined paths.

The wireframes used for the design of the sitemap are available in Appendix A.1.

Chapter 4

Implementation (2458/2500 words)

4.1 Overview of the Implemented System

The application was implemented following the design of the system. It consists of a decoupled web application (React SPA + DRF backend). Users can create accounts and manage multiple contextual identities using CRUD operations. Each contextual identity has a visibility flag that the user can set. The frontend acts as a user facing wrapper over the underlying API calls, meaning the user is able to use the API instead. Explicit, privacy-preserving, contextual identity sharing functionality was created and shown to work through the creation and use of a toy RP acting as a proof-of-concept. All core functionalities have been completed. The system is live and functional at https://identity-project-frontend.onrender.com/. The API documentation is available at https://identity-project-backend.onrender.com/docs. The Toy RP is available at https://identity-project-toy-rp.onrender.com and can be used to test the SSO flow.

4.2 Backend Implementation

The backend has been created using Django and DRF. The result is a RESTful API for all operations. The database has been updated to PostgreSQL for deployment, as it is better suited for production environments than SQLite (DRF default) due to its better security and support for concurrency and scale. DRF-SimpleJWT is used for authentication, and RS256-signed scoped tokens are issued to RPs upon user consent.

Architecture and Models

In line with best practices on modularity and separation of concerns, the backend uses a Model–Serializer–View (MSV) structure typical of DRF REST APIs. The models encapsulate the data layer; serializers act as the interface layer through handling validation and JSON conversion; and views (actually controllers) handle requests and provide responses.

To reduce boilerplate, most views inherit from DRF's generic class-based views (e.g., `ListCreateAPIView`, `RetrieveUpdateDestroyAPIView`). These provide built-in handling for common operations such as listing, creating, updating, and deleting records. The behaviour is customised by selectively overriding the default methods.

Actions and resource requests are scoped to the `request.user` in almost all views. This design ensures users can only access and modify their own data. I implemented this security pattern in two distinct ways depending on the view's purpose:

1. **Filtering the Queryset:** For views that operate on a collection of objects, such as listing or retrieving a specific identity from a user's set, the `get_queryset()` method is overridden. This modifies the database query to only search records owned by the current user before DRF's default lookup logic attempts to find an object by its primary key. An example of this is shown in Listing 4.1.
2. **Overriding the Lookup Logic:** For views that operate on a single object tied to the user (like their own account), the `get_object()` method is overridden. This replaces the default behaviour of looking for the object from a primary key in the URL, ensuring the view can only ever operate on the authenticated user. This more direct approach is shown in Listing 4.2.

This double approach robustly protects against access and privilege vulnerabilities by eliminating any reliance on URL parameters for user-specific data retrieval.

```
class IdentityRetrieveUpdateDestroyView(
                    generics.RetrieveUpdateDestroyAPIView):
    """View for retrieving, updating, and deleting a specific
       identity."""

    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        return Identity.objects.filter(user=self.request.user)
```

Listing 4.1: Scoping access by filtering the queryset.

```
class AccountUpdateView(generics.RetrieveUpdateDestroyAPIView):
    """View for retrieving, updating, and deleting the current
       authenticated users account."""

    permission_classes = [permissions.IsAuthenticated]

    def get_object(self):
        """RetrieveUpdateDestroyAPIView expects to retrieve an
           object from a url parameter because it uses the
           RetrieveModelMixin. Using the current authenticated
           user is more secure."""
        return self.request.user
```

Listing 4.2: Scoping access by overriding the object lookup method.

The database was implemented exactly as in the design (see section 3.7). Each model adheres to the designed schema, and, additionally, several model-level utility

methods were implemented to encapsulate model-specific logic and for separation of concerns. Listing 4.3 demonstrates four such utilities in the `TokenLog` model:

```python
class TokenLog(models.Model):
    # ... field definitions ...

    def is_expired(self):
        """Check if the token has expired."""
        return self.exp < timezone.now()

    def mark_as_used(self):
        """Mark the token as used."""
        self.used = True
        self.save()

    def mark_as_revoked(self):
        """Mark the token as revoked."""
        self.revoked = True
        self.save()

    def is_valid(self):
        """Check if the token is valid
        (not used or revoked or expired)."""
        used = self.used
        revoked = self.revoked
        expired = self.expired()
        return not used and not revoked and not expired
```

Listing 4.3: Utility methods in the `TokenLog` model

Authentication and Authorisation

Authentication is handled using DRF-SimpleJWT, which provides a secure and stateless token-based login mechanism. I used two ready-made endpoints from SimpleJWT:

- `/api/token/`: Issues access and refresh tokens upon successful login.
- `/api/token/refresh/`: Refreshes access tokens using a valid refresh token.

All authenticated API routes are protected using the DRF `IsAuthenticated` permission class as shown in Listing **??**. These tokens are kept in local storage and automatically refreshed by the frontend when necessary.

A second type of token is issued for the purpose of identity sharing with third-party services as per R-Share-3. These tokens are signed using an RS256 private key and are valid for 5 minutes. They are also scoped to a specific contextual identity and marked as used on successful consumption (single-use). The context surrounding the generation and use of the custom token is available in the dedicated flow walkthrough in section 4.4.

Rate Limiting and Throttling

The backend implements request throttling using DRF's throttling classes to mitigate abuse such as brute-force login attempts and spam registrations. Global

user and anonymous request limits were set at 400 requests/hour for authenticated users and 100 requests/hour for unauthenticated users. Additionally, sensitive endpoints such as registration and login were given custom throttle classes (10 registrations/hour, 20 login attempts/hour) This is shown in Listing 4.4.

```
class RegistrationThrottle(AnonRateThrottle):
    """Throttle registration to prevent spam accounts."""
    scope = "registration"

class LoginThrottle(AnonRateThrottle):
    """Throttle login attempts to prevent brute force attacks."""
    scope = "login"
```

Listing 4.4: Custom throttling classes for registration and login

These classes were applied to the corresponding views (e.g., `UserRegisterView`), ensuring that requests exceeding the threshold are automatically rejected with a `429 Too Many Requests` response.

API Endpoints

Table 4.1 describes the endpoints implemented and their methods and purpose.

Table 4.1: Implemented API Endpoints.

Endpoint and Method	Purpose	Access
POST `/api/register/`	Register a new user	Public
POST `/api/token/`	Obtain Simple-JWT access + refresh tokens	Public
POST `/api/token/refresh/`	Refresh access token	Public
GET, POST `/api/identities/`	List or create identities for the authenticated user	Authenticated
GET, PATCH, DELETE `/api/identities/<id>/`	Retrieve, update, or delete an identity by ID	Authenticated
POST `/api/authorize/`	Generate scoped JWT token after user consent	Authenticated
GET `/api/token_identity/`	Retrieve identity via scoped JWT (3rd party access)	Public (token required)
GET `/api/token_log/`	Retrieve a list of token records	Authenticated
GET `/api/snapshot/<id>/`	Retrieve immutable snapshot of previously shared data	Authenticated
PATCH `/api/revoke_token/<jti>/`	Revoke an access token by its JTI	Authenticated
GET, PATCH, DELETE `/api/account/`	Retrieve, update, or delete the user's account	Authenticated
GET `/api/contexts/`	Retrieve list of contexts	Authenticated
GET `/api/public_key/`	Fetch the public RSA key for verifying tokens	Public
GET `/api/schema/`	OpenAPI schema for the API	Public
GET `/api/docs/`	Swagger UI for API documentation	Public

4.3 Frontend implementation

The interface was built with React and React router for client-side navigation. Chakra UI was used for consistent, accessible styling and to provide support for dark mode and toast notifications. All operations such as registration, login, identity and account management, the consent flow, and auditing are powered by API communication with the DRF backend. All user-generated content is retrieved via Django serializers, which ensure proper escaping by default. This behaviour has been verified through attempts to insert HTML tags into the biography field of an identity.

Authentication and State Management

The backend issues JWTs for authentication. The frontend uses a custom `AuthContext` to encapsulate authentication logic and store tokens in React state and browser local storage. On login, the access and refresh tokens are stored and attached to subsequent API calls.

All authenticated API calls are made through a custom `fetchWithAuth()` utility function exposed by `AuthContext`. This function refreshes the token if expired and attaches the access token to the request headers, as shown in Listing 4.5. If any errors occur, they are thrown up to the caller, who has the appropriate context to handle them.

```
const fetchWithAuth = useCallback(
  async (url, options = {}, allowRetry = true) => {
    // Get fresh token from ref (prevents race conditions)
    let access = currentTokenRef.current;

    // Refresh token if expired
    if (isTokenExpired()) {
      access = await refreshToken();
      if (!access) {
        throw new Error('Failed to refresh token');
      }
    }

    const res = await fetch(url, {
      method: options?.method ?? 'GET',
      headers: {
        ...options?.headers,
        Authorization: `Bearer ${access}`,
        'Content-Type': 'application/json',
      },
      body: options?.body ?? null,
    });

    // Handle 401 with automatic retry
    if (res.status === 401 && allowRetry) {
      return fetchWithAuth(url, options, false);
    }

    // Parse response data
    const isJson = res.headers
      .get('content-type')
      ?.includes('application/json');
    const data = isJson && res.status !== 204 ? await res.json() :
        null;

    if (!res.ok) {
      const error = new Error(`API Error: ${res.status}`);
      error.status = res.status;
      error.details = data || { detail: res.statusText };
      throw error;
    }

    return data;
  },
  [isTokenExpired, refreshToken]
);
```

Listing 4.5: The fetchWithAuth utility for authenticated API requests.

Routing and Access Control

Navigation is handled via React Router. Access control is implemented through conditionally rendering routes based on the users authentication state. Protected routes such as the dashboard, audit log, account management, and consent page, verify that the user is logged in using the `isLoggedIn` flag provided by `AuthContext`. Users are redirected to the login page when necessary. Public routes such as the home page, login, registration, and documentation are always accessible.

The system preserves the intended destination when redirecting users using the `location.state` object. This means that users are returned to their target page after a successful login. This is essential for the function of the SSO flow, where query parameters such as `callback_uri` must be preserved. A custom `RedirectWithToast` component was created to provide information to the user on redirection due to authentication failure or invalid routes (e.g., the login page while already authenticated).

Dashboard and Identity Management

The dashboard provides an overview of all of the users' contextual identities, which are fetched on mount. Users can manage their identities through a form with full CRUD functionality. The form behaves dynamically depending on whether it is in create or update mode. This includes the pre-filling the form and appropriate headings. All state updates are preformed reactively, and toast notifications are displayed for successful and failing actions using Chakras `useToast()` utility function. The dashboard is shown in Figure 4.1.

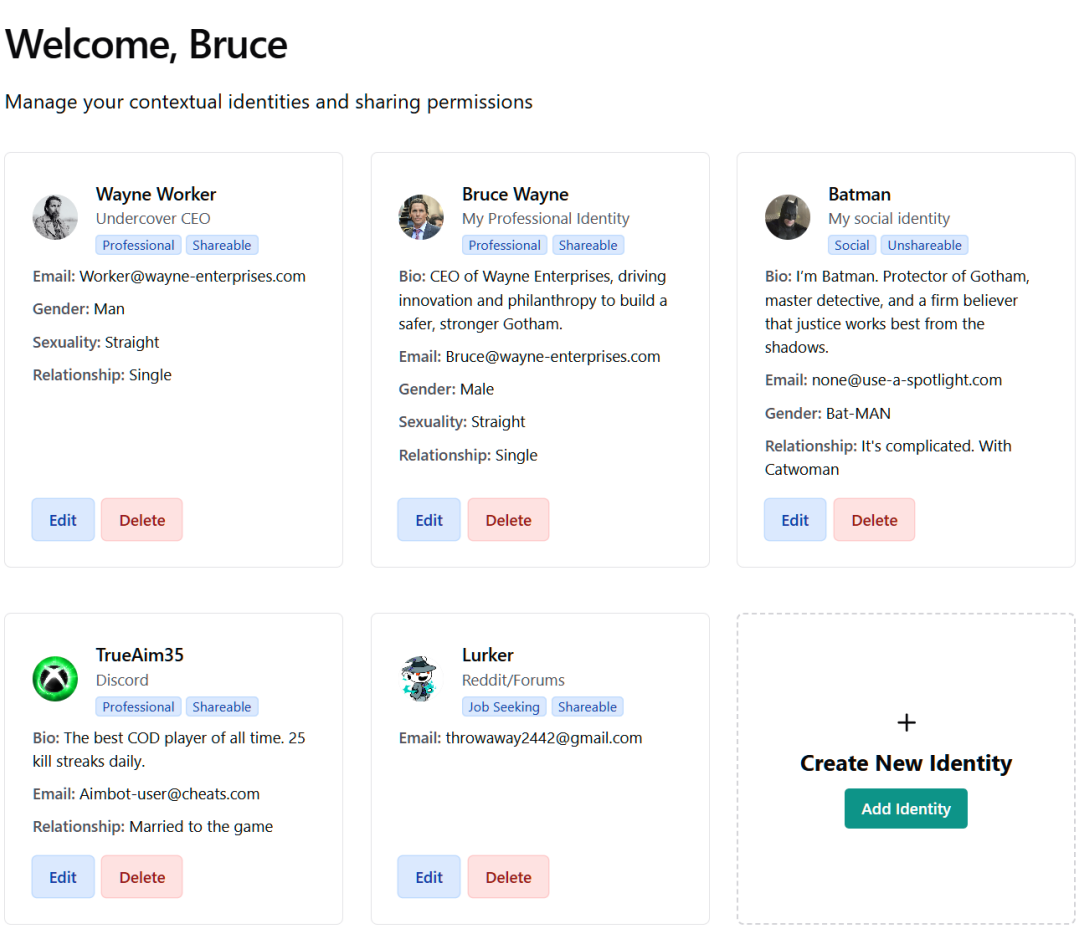


Figure 4.1: Dashboard Interface for Identity Management.

Styling and User Experience

The frontend was built with a mobile-first design and is fully responsive across devices. Chakra UI components such as `Container`, `Stack`, and `Grid` provided the basis for flexible layouts and breakpoints ensured proper scaling from small screens to large desktops. Accessibility was prioritised through dark mode and link underline toggles, as well as ARIA attributes and proper semantic elements. User experience was improved through field-error display and toast notifications to provide clear, contextual feedback.

4.4 Single-Sign-On flow walk-through

This section will go through the entire SSO flow. Figure 4.2 shows a simplified representation of the implemented flow for consent-based identity sharing.

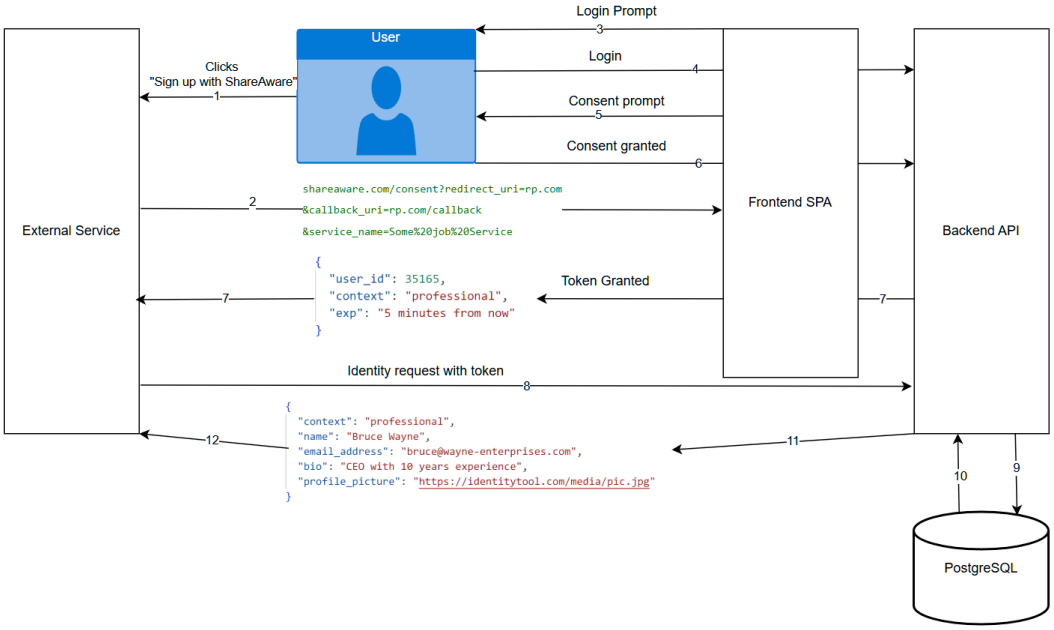


Figure 4.2: Overview of the implemented flow.

The Request

The third-party service includes an SSO button on their registration/login pages. This button acts as a link to ShareAware and has the third-party service name and callback and redirect URI's as URL parameters. This is explained to the RP developer in the developer guide page. An example SSO button is shown in Figure 4.3.

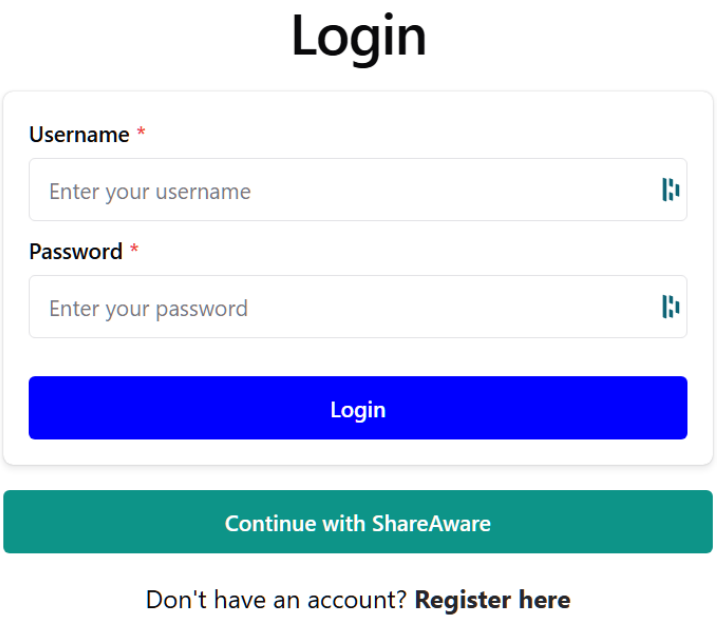


Figure 4.3: An example SSO button.

The Consent

When the user clicks the SSO button, they are redirected to the consent page. The system prompts the user to login if needed, preserving the next parameter. The page performs several important functions before allowing identity sharing. The page verifies the request contains the required parameters. If the `redirect_uri` is missing, a toast notification and a back button are displayed. If another parameter is missing then the user is redirected to the provided URI with an error message appended to the URI. These URI encoded error messages are explained in the developer documentation and it is the RPs responsibility to decide how to handle them. The case of a missing `redirect_uri` is shown in Figure 4.4

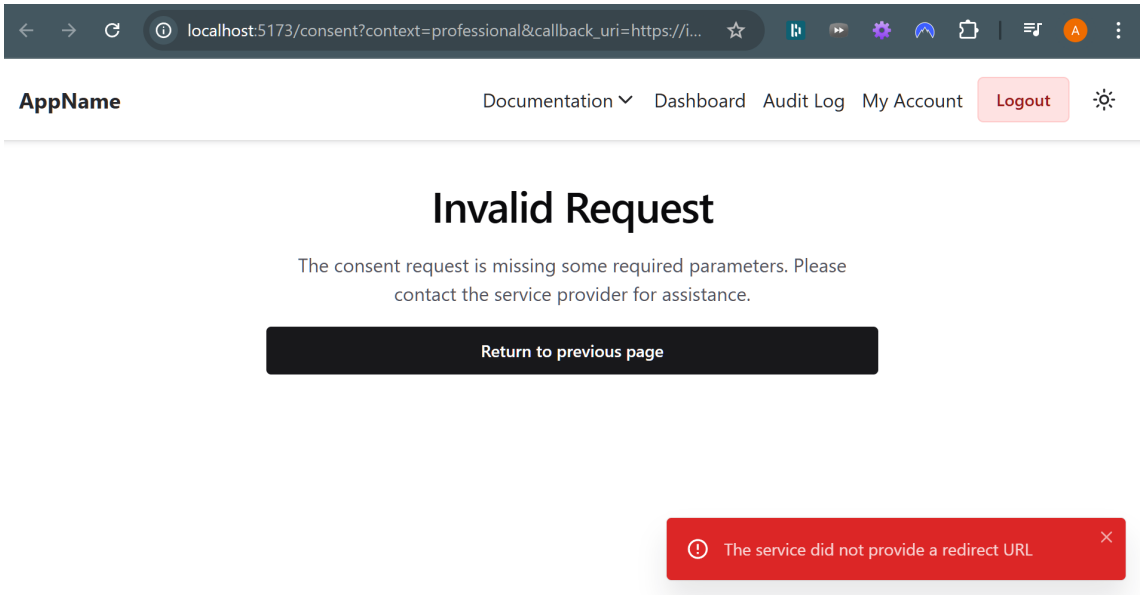


Figure 4.4: An invalid request.

Next, the system ensures that the user has a shareable identity (visibility marked as `sharable`). If they do not, they are notified and prompted to create one or return to the RP. This is shown in Figure 4.5

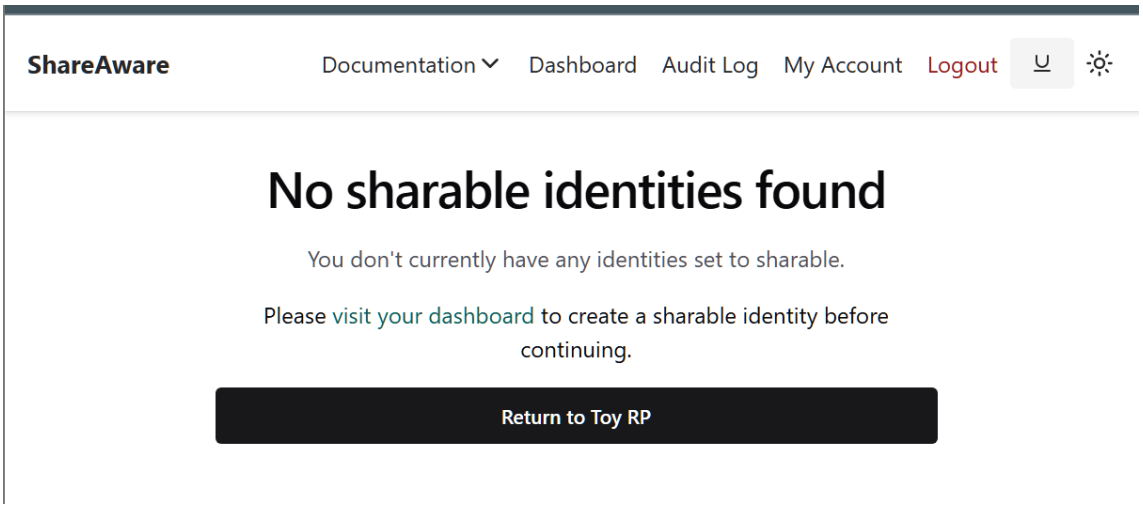


Figure 4.5: No Identity Found.

If the request is valid, the user is shown their shareable identities with the exact data they will share upon consent for each, along with the name of the service making the request. The user is prompted to select an identity to share or reject the request as shown in Figure 4.6.

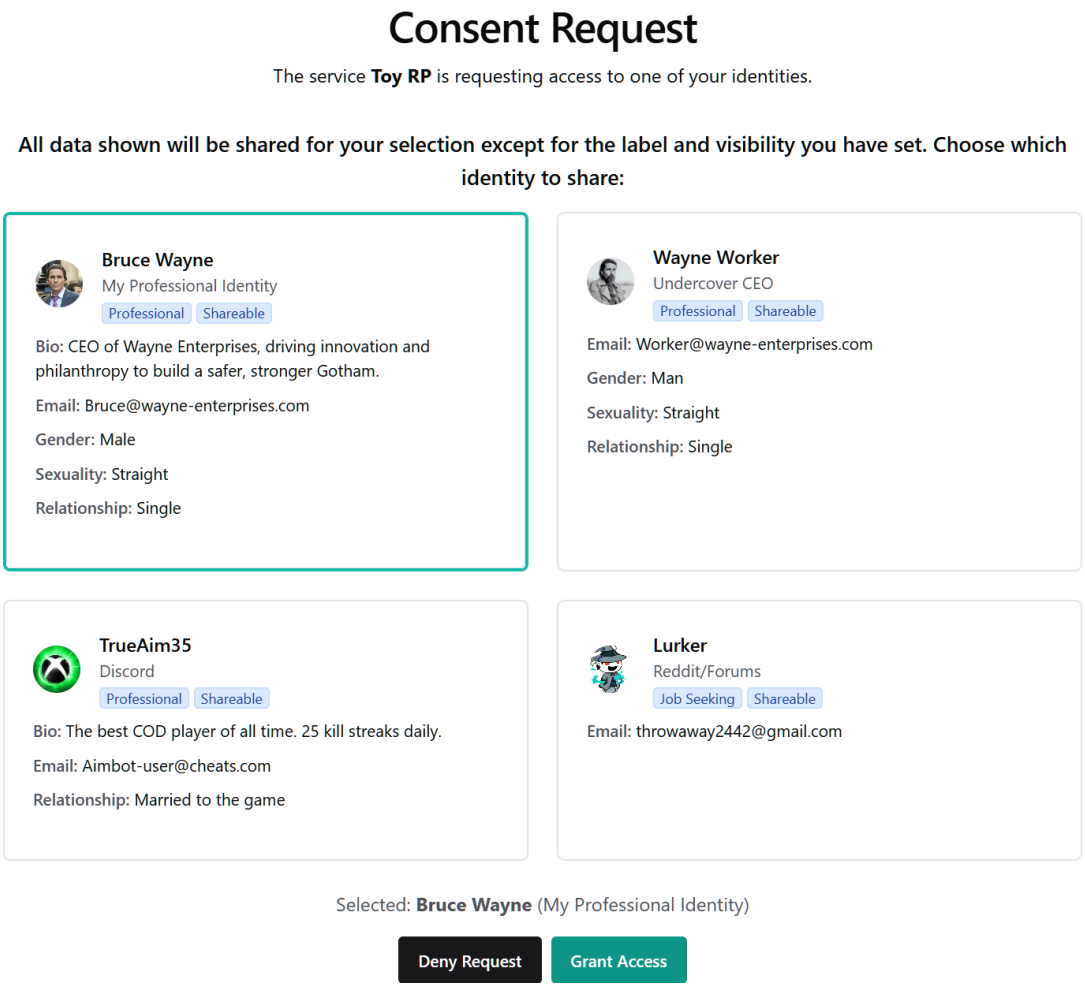


Figure 4.6: The Consent Page.

If the user rejects, they are redirected with `?error=consent_rejected` appended to the URL. If they accept, then the SPA sends a POST request to the `/authorize` endpoint with the selected identity ID and provided parameters.

Token Creation and Transmission

The backend validates the request and creates a signed, scoped JWT as per R-Share-3. It also creates new `TokenLog` and `IdentitySnapshot` instances for auditing and revocation purposes, as shown in Listing 4.6.

```
class ConsentGrantedView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def post(self, request):
        # ... validation logic for request data (identity_id,
            callback_uri etc.) skipped for brevity ...

        try:
            identity = Identity.objects.get(id=identity_id,
                user=request.user, visibility="token_only")
            # "token_only" == "shareable"
        except Identity.DoesNotExist:
```

```
            return Response({"detail": "Identity not found"},
                status=status.HTTP_404_NOT_FOUND)

        # 1. Create an immutable snapshot of the identity
        snapshot_serializer =
            IdentitySnapshotSerializer(data={"identity":
            identity.id})
        if snapshot_serializer.is_valid():
            snapshot = snapshot_serializer.save()
        else:
            return Response(status=
                            status.HTTP_500_INTERNAL_SERVER_ERROR)

        # 2. Create the JWT payload, including the snapshot ID
        payload = {
            "user_id": request.user.id,
            "identity_id": identity.id,
            "snapshot_id": snapshot.id,
            # ... other payload fields (exp, jti, etc.) ...
        }
        token = jwt.encode(payload, settings.PRIVATE_KEY,
            algorithm="RS256")

        # 3. Log the transaction for auditing
        TokenLog.objects.create(
            user=request.user,
            identity=identity,
            identity_snapshot=snapshot,
            jti=jti,
            # ... other log fields ...
        )

        # 4. Return the token to the frontend
        return Response({"token": token},
            status=status.HTTP_200_OK)
```

Listing 4.6: Token creation, snapshotting, and logging in `ConsentGrantedView`.

The token is received by the SPA as a response to its POST request. The SPA creates a hidden form, appending the token as an input field within, and POSTs the form to the provided `callback_uri`.

Token Verification and Consumption

The RP can then validate the token using the public key retrievable from the `/api/public_key/` endpoint and the verification method of a JWT library. Figure 4.7 shows a decoded token.

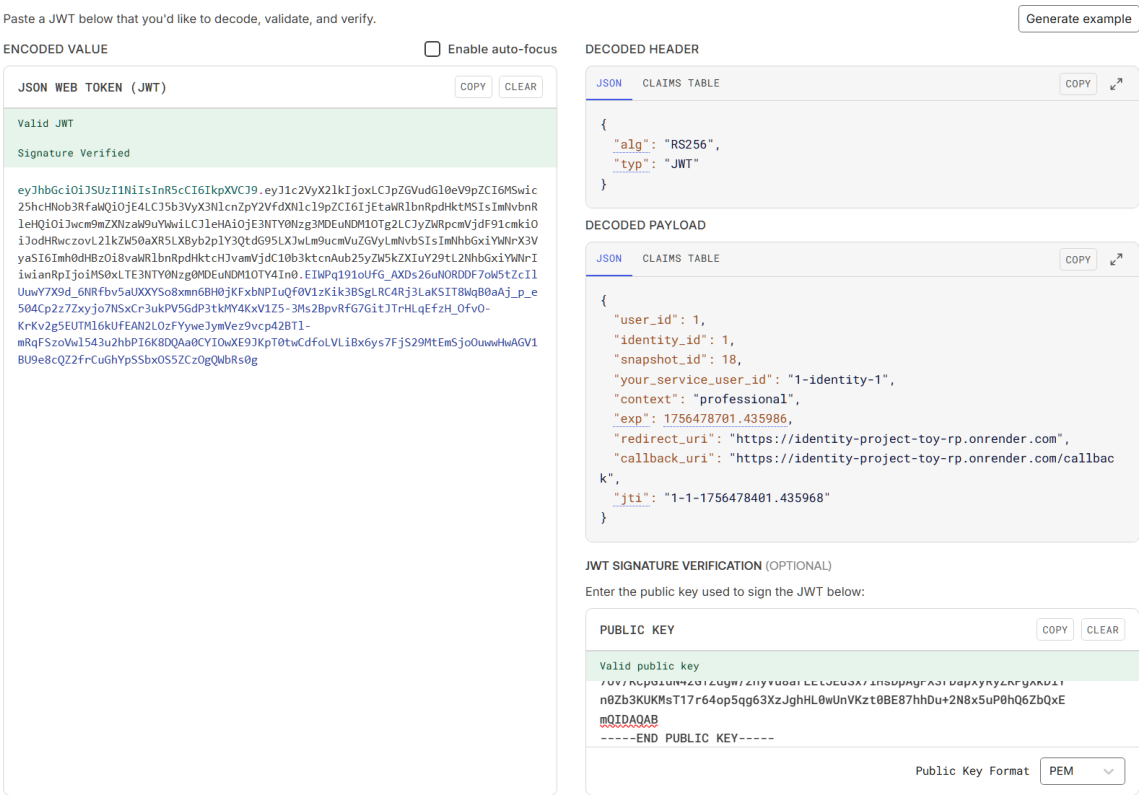


Figure 4.7: Generated using jwt.io

The token payload contains a `your_service_user_id` field which the RP can use to look up the user in their database. If present, they can log them in, optionally querying the `token_identity` endpoint for updated user data. Otherwise, the RP is expected to attempt to create an account, including the assigned user id for their service. If fields are missing, they are expected to generate a partially populated sign-up form. If the RP would like to consume the token to retrieve the data, they should make a GET request to the `token_identity` endpoint. An example of this is shown in Listing 4.7, which is a snippet from the toy RP.

```
app.post('/callback', async (req, res) => {
  // Get the token from the request body
  const token = req.body.token;

  // Validate the token using the public key
  const publicKeyResponse = await fetch(GET_PUBLIC_KEY_URL);
  const publicKey = await publicKeyResponse.text();

  const decodedToken = jwt.verify(token, publicKey, {
    algorithms: ['RS256'],
  });

  /*
  Use the decoded tokens "your_service_user_id" field to log the
     user in. This is skipped in this demo, but you would use this
     value to look up the user in your database and create a
     session.

  You can optionally fetch the identity using the token to update
     the user's profile on your service's database.
```

```
  */
  const response = await fetch(GET_IDENTITY_URL, {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });
  const data = await response.json();

  res.render('profile', { identity: data });
});
```

Listing 4.7: Token verification and data consumption in the Toy RP.

The backend validates the request, verifying that the token has not been tampered with and is not used, revoked, or expired. The identity is then serialised (converted to JSON) and returned in the response. This process is shown in Listing 4.8.

```
class TokenIdentityView(APIView):
    permission_classes = [permissions.AllowAny]

    def get(self, request):
        # ... token extraction from author header ...

        try:
            payload = jwt.decode(token, settings.PUBLIC_KEY,
                algorithms=["RS256"])
        except (jwt.ExpiredSignatureError, jwt.InvalidTokenError):
            return Response({"detail": "Invalid or expired
                token"}, status=status.HTTP_403_FORBIDDEN)

        # Check against the TokenLog to prevent reuse
        token_log =
            TokenLog.objects.filter(jti=payload.get("jti")).first()
        if not token_log or not token_log.is_valid():
            return Response({"detail": "Invalid or expired
                token"}, status.HTTP_403_FORBIDDEN)

        # Mark the token as used to prevent replay attacks
        token_log.mark_as_used()

        # Retrieve the live identity via the snapshot reference
        try:
            id = payload.get("snapshot_id")
            snapshot = IdentitySnapshot.objects.get(id)
            if snapshot.identity:
                serializer = IdentitySerializer(snapshot.identity)
                return Response(serializer.data,
                    status=status.HTTP_200_OK)
            # Identity was deleted: treat as revoked
            return Response({"detail": "Identity access has been
                revoked by the user"}, status=status.HTTP_410_GONE)
        except IdentitySnapshot.DoesNotExist:
            return Response(status=status.HTTP_404_NOT_FOUND)
```

Listing 4.8: Backend view for serving identity data to third parties.

This JSON response can then be used by the RP to create an account or update user profile information. The images are sent over the API as Base64 strings with

the Mime type dynamically detected and prefixed within the identity serializer, therefore, they can be directly used in the `src` attribute of an image tag. An example consumption of the response to generate a profile is shown in Listing 4.7 and the resulting profile is shown in Figure 4.8.

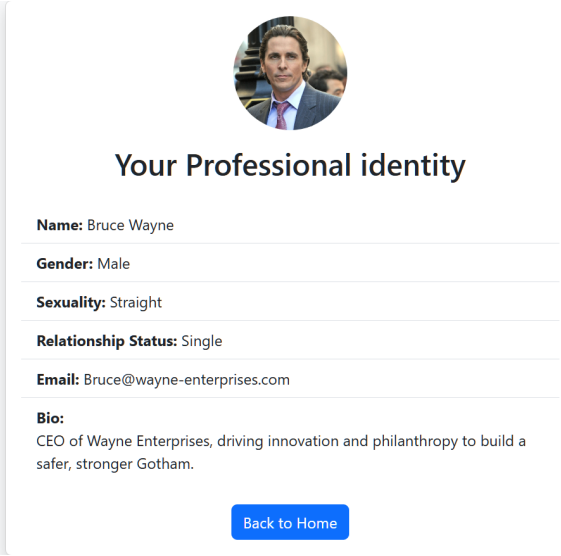


Figure 4.8: Demonstration of Retrieved Data Use.

4.5 Audit Logging and Revocation

Each time consent is granted, the system takes a snapshot of the identity data and stores it as an instance of the `IdentitySnapshot` model, as shown in Listing 4.6. This model stores profile pictures in a resized image field (100 x 100 pixels) to reduce storage requirements from having to save the image each time. The token generation metadata is logged in a `TokenLog` instance along with a foreign key to the snapshot, forming a permanent audit trail. This metadata includes the token's JTI (unique identifier), timestamps for issue and expiry, used and revoked flags, nullable identity ID, redirect and callback URIs, and the name of the requesting service.

Upon user consent, the `ConsentGrantedView` validates the request and triggers the creation of an `IdentitySnapshot` instance via the `IdentitySnapshotSerializer`. This process creates an immutable record of the shared identity's metadata at that exact moment. The unique ID of this new snapshot is then embedded directly into the payload of the JWT issued to the third party.

The primary trade-off of the snapshot is image quality. For a user's own audit trail, the `/api/snapshot/<id>/` endpoint returns the exact `IdentitySnapshot`, whose resized thumbnail is fine for logging purposes. However, for third parties the `TokenIdentityView` retrieves the live `Identity` object. This is a deliberate design choice to provide RPs with the original, high-resolution profile picture instead of the low-resolution thumbnail stored in the snapshot. This ensures the RP receives a high-quality image but means profile fields may not match the snapshot if the user made edits before the RP made a claim within the 5-minute window. This is discussed further in the Evaluation chapter.

The `/api/token_log/` endpoint returns all tokens issued by the authenticated user. These are displayed in the audit interface shown in Figure 4.9, where users can view records and revoke tokens that are still unused.

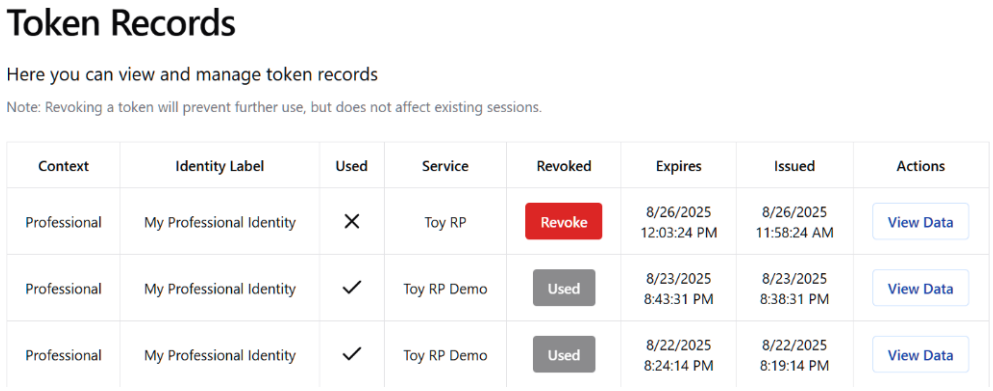


Figure 4.9: Token Record Management Interface.

The revoke button calls the `/api/revoke_token/<jti>` endpoint. This action marks the token as revoked in the database, preventing it from being consumed by third-party services. Tokens that have already expired or have been used are marked appropriately and cannot be revoked.

Due to the way the toy RP is implemented, issued tokens are always used immediately upon generation. This limits the ability to demonstrate revocation in the live interface, as there isn't a window in which a valid but unused token exists. I considered tweaking the Toy RP such that it would be possible to approve access but not use it until user action but decided against it for now due to time constraint. The interface in 4.9 above shows a record in which a used token may still be revoked due to commenting out consumption code in the toy RP.

Despite this demonstration limitation in the frontend, the backend enforces revocation correctly as validated in automated tests. Listing 4.9 shows a test which confirms that a revoked token cannot be used to access the identity endpoint:

```
def test_using_revoked_token_fails(self):

    # Create an identity and request a token
    self.create_identity()
    token = self.request_token()
    revoked_token = token
    response = self.client.get(self.token_log_url, format="json")
    jti = response.data[0]["jti"]

    # Revoke the token
    revoke_url = reverse("revoke_token", kwargs={"jti": jti})
    response = self.client.patch(revoke_url, format="json")
    self.assert_equal_status(response, status.HTTP_200_OK)
    self.assertTrue(response.data["revoked"])

    # Attempt to use the revoked token
    self.assert_equal_status(
        self.use_token(revoked_token), status.HTTP_403_FORBIDDEN)
```

Listing 4.9: Test for token revocation

The token log can also be used to view the snapshots for a particular consented sharing instance, as shown in Figure 4.9. The snapshot view is shown in Figure 4.10.

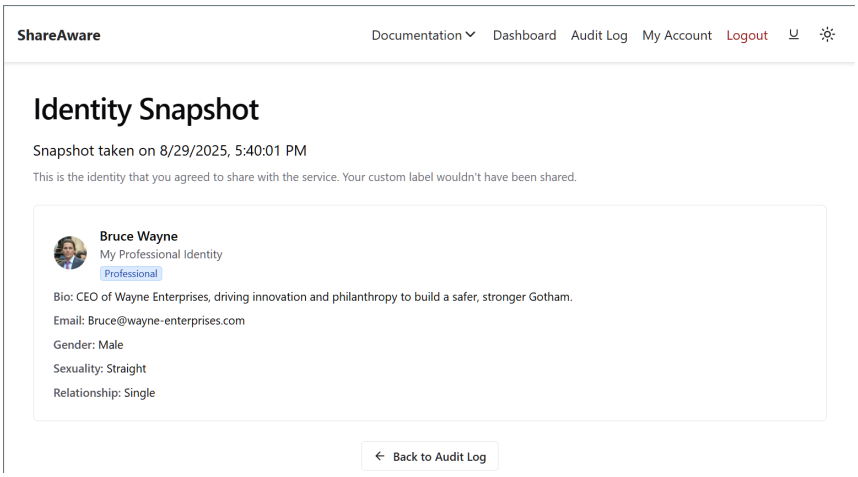


Figure 4.10: The snapshot generated upon consent in Figure 4.6.

4.6 Deployment

The backend, frontend, and toy RP were deployed using Render for simplified HTTPS hosting and continuous deployment from GitHub. PostgreSQL was hosted using Amazon RDS, and environment variables (such as RSA keys and database credentials) were configured using Render's secure environment variables functionality. Each service was hosted from its own repository with unique build and deploy pipelines due to their differing tech stacks.

Chapter 5

Evaluation (2286/2500 words)

5.1 Overview and Approach

This chapter provides a comprehensive evaluation of the ShareAware system. It assesses its success in meeting the project's core objectives and requirements. The approach combines both qualitative and quantitative methods to provide an effective view on the system's performance, security, and usability. The evaluation begins with a review of functional and non-functional requirements coverage, and then continues with an analysis of the system's security and privacy. The technical evaluations are followed by a critical reflection on the implementation trade-offs and an analysis of user feedback gathered through surveys.

5.2 Requirements Coverage

Table 5.1 maps implemented functionality to the requirements defined in Chapter 3.

Table 5.1: Requirement Coverage Summary. Legend: ✓= Fully implemented, ∼ = Partially implemented, ✗= Not implemented.

Requirement	Status	Notes
R-Core-1	✓	SimpleJWT used for secure login and session management. User model implemented as per DB design.
R-Core-2	✓	Users can create as many contextual identities as they wish, with labels to differentiate them.
R-Core-3	✓	Identity fields are optional and free-text format. A custom combobox element allows for either free text or selection from common options where appropriate.
R-Core-4	✓	Visibility flag implemented. Enforced before data is shared.
R-Share-1	✓	Consent screen on UI prompts user before data is released to RP.
R-Share-2	✓	Consent page allows the user to select an identity to share and displays the data that would be shared.
R-Share-3	✓	Token payload implemented as designed.
R-Share-4	✓	Tokens are sent as a response to the frontend. The frontend then POSTs to the provided callback URI.
R-Share-5	✓	Dedicated identity retrieval endpoint present and requires valid JWT.
R-Aud-1	✓	Dedicated token log retrieval endpoint present. SPA displays log in a UI table.
R-Aud-2	✓	Revocation available through the UI table or a dedicated endpoint.
R-Aud-3	✓	Snapshots are available through the token log UI table.
R-Acc-1	✓	Available on the SPA and also via a dedicated endpoint.

All requirements have been met, including those derived from the user feedback.

Sitemap coverage

Table 5.2 below maps the implemented routes to the design sitemap.

Table 5.2: Sitemap Coverage Summary (By Page). Legend: ✓= Fully implemented, ∼ = Core logic implemented (UI improvements needed), ✗= Not implemented.

Page	Status
Home Page	✓
Login Page	✓
Registration Page	✓
Dashboard	✓
Consent Page	✓
Audit Log	✓
Snapshot Page	✓
User Documentation	✓
Developer Documentation	✓
Demo Walkthrough	✓
Demo Page	✓
404 Not Found	✓
Account Management	✓

All pages have been created and meet the design specifications.

5.3 Security and Privacy

The security and privacy of the system was treated as a central design constraint driven by GDPR principles, threat models and the shortcomings of OIDC deployments identified in the literature. These design intentions were carefully considered in the implementation and evaluated through automated testing and exploratory usage.

Access Control via `request.user`

Most views restrict data access by scoping queries through `request.user`. This prevents users from accessing or modifying data that does not belong to them. For example, identity views use:

```
Identity.objects.filter(user=self.request.user)
```

Even when a primary key is supplied in the URL, this ensures users cannot retrieve resources they do not own. The exception is during the consumption of the sharing tokens to make claims. In that case the user ID is taken from the token payload as the request comes from the RP. This is not a vulnerability, however, as the users consent prompts the creation of the payload. Therefore, the payload user ID is the ID of the `request.user` that consented to the identity sharing.

Scoped, Signed Token Access

The system uses single-use, RS256-signed JWT tokens scoped to a specific context (later to become identity ID). These tokens are:

- Cryptographically signed using asymmetric keys (mitigating algorithm spoofing [7]).
- Valid for 5 minutes and single-use (effectively eliminating replay attacks [6]).
- Only usable if the identity has `visibility=token_only` (respecting user intention and preventing accidental sharing).

These tokens are POSTed to a `callback_uri` (instead of passed as URI fragments), avoiding exposure via referer headers or browser history.

Consent and Visibility Enforcement

Each token is issued only after explicit, user consent on a per-request basis. This consent is informed, in that the user is shown exactly what data will be shared and with whom. If the user has not marked an identity as shareable, the identity wont be shown as an option on the consent request page. The backend uses an additional check to further enforce this by explicity ensuring that the candidate identity to be shared is marked with the correct visibility option. This layered defence model eliminates accidental sharing.

Additional Security Measures

Beyond the core token logic and access control patterns, the system incorporates additional safeguards:

- All sensitive API endpoints are protected using DRFs `IsAuthenticated` permission class.
- Django's database interface and serializers ensure proper escaping of user provided data, preventing XSS and SQL injection.
- JWTs are stored securely in the frontend and automatically refreshed using the token refresh endpoint.

CSRF Protection

CSRF mitigation is required for cookie-based session authentication. This system utilises stateless JWT authentication passed in request headers. Since no cookies are involved and requests require an explicit authorisation header, CSRF protections are unnecessary, and thus deliberately not included.

CORS Configuration

The CORS (Cross-Origin Resource Sharing) policy has intentionally been set to public through `CORS_ALLOW_ALL_ORIGINS = True`. This is because the system is designed to support a publicly accessible and functional API, in addition to requests from the frontend and unregistered RPs. Its possible to create a stricter system wherein only the SPA frontend can make use of browser-based access. However, CORS is a browser-based security feature and does not affect server-to-server or non-browser access. As such, API requests made from backend services (e.g., via Python, Node.js) are unaffected by CORS restrictions. Actual access enforcement

would require client registration and whitelisting callback URIs, both of which are impractical and out of scope.

Limitations in Design

While the system's security is robust, the implementation involved specific trade-offs and contains limitations worth noting.

Snapshot Immutability

The `IdentitySnapshot` model was introduced to ensure that an immutable record of consented data is kept and is viewable by the user to increase the usefulness of the audit log. As explained in the Implementation chapter, the final design still fetches the live identity object when a third party makes a claim. This was an intentional compromise to enable serving high-resolution profile pictures rather than the resized thumbnails stored in the snapshot. This could have been completely avoided by storing the full profile picture in the snapshot and serving the snapshot to the third party. However, this would result in the picture being saved for each SSO instance, massively ballooning storage use. The implemented solution instead relies on the mitigation provided by the revocation capabilities, treating identity deletion as revocation, and the short Time-To-Live (TTL).

Arbitrary Token Expiration

The five-minute TTL for third-party access tokens was a rational choice, but not a researched one. It was selected as a balance between providing a reasonable window for an RP to consume the token and minimising the risk of misuse if a token was leaked. A more evidence-based approach would involve researching typical SSO completion times to determine an optimal TTL that improves security and usability.

5.4 Test Coverage

A comprehensive test suite was created using DRFs `APITestCase`. The goal of the tests is to validate the correctness, security, and privacy properties of the system. Table 5.3 provides a high-level summary of the areas covered.

Table 5.3: Summary of Automated Test Coverage.

Feature Area	**Validation Summary**
User and Account Management	Confirms successful registration, login, and secure management of user accounts. Prevents authenticated users from re-registering and ensures account updates and deletions are functional.
Identity Management (CRUD)	Validates the full functionality of identity management: create, retrieve, update, and delete.
Token-based Identity Sharing	Verifies the end-to-end process of generating and consuming the sharing tokens. Confirms tokens are correctly scoped and that only a valid token (not revoked, used or expired) can retrieve only the intended identity data.
Security and Privacy Enforcement	Confirms the system's protection against token replay and tampering. Tests that visibility rules, like `unshareable`, are correctly enforced.
Audit Trail and Logging	Ensures that every token issued and used is logged. Verifies that the audit log is accessible to the user and correctly reflects the status of tokens.
Rate Limiting and Throttling	Validates that rate limits are correctly applied to sensitive endpoints like registration and login to prevent abuse. Also confirms that global request limits are enforced for both anonymous and authenticated users.

The tests confirm that the backend behaves as intended, from basic user operations to the critical logic of token generation and consumption. For example, token tampering and single-use validation tests are critical for proving the system's resistance to common attack vectors discussed in Section 3.5. Figure 5.1 shows the output of these tests.

```
System check identified no issues (0 silenced).
test_account_delete (accounts.tests.AccountTests.test_account_delete)
Deletes the authenticated user's account. ... ok
test_account_get (accounts.tests.AccountTests.test_account_get)
Gets the authenticated user's account details. ... ok
test_account_patch (accounts.tests.AccountTests.test_account_patch)
Updates the authenticated user's account details. ... ok
test_authenticated_user_cannot_register_again (accounts.tests.AccountTests.test_authenticated_user_cannot_register_again)
Test that an authenticated user cannot register a new account ... ok
test_deleted_identity_returns_410 (accounts.tests.IdentitySnapshotTests.test_deleted_identity_returns_410)
Test that deleted identity results in 410 Gone response. ... ok
test_snapshot_created_on_token_generation (accounts.tests.IdentitySnapshotTests.test_snapshot_created_on_token_generation)
Verify identity snapshot is created when token is generated. ... ok
test_snapshot_view_access (accounts.tests.IdentitySnapshotTests.test_snapshot_view_access)
Test that users can view their identity snapshots. ... ok
test_create_identity (accounts.tests.IdentityTests.test_create_identity)
Creates an identity successfully. ... ok
test_delete_identity (accounts.tests.IdentityTests.test_delete_identity)
Deletes an identity successfully. ... ok
test_multiple_identities_same_context_allowed (accounts.tests.IdentityTests.test_multiple_identities_same_context_allowed)
Users can create multiple identities with the same context. ... ok
test_read_identities (accounts.tests.IdentityTests.test_read_identities)
Retrieves identities successfully. ... ok
test_update_identity (accounts.tests.IdentityTests.test_update_identity)
Updates an identity successfully. ... ok
test_anon_user_limits (accounts.tests.RateLimitingTest.test_anon_user_limits)
Ensures that anonymous users have a limit of 100 requests per hour. ... ok
test_authenticated_user_higher_limits (accounts.tests.RateLimitingTest.test_authenticated_user_higher_limits)
Test that authenticated users have higher rate limits (400/hour) than anonymous users (100/hour). ... ok
test_login_rate_limiting (accounts.tests.RateLimitingTest.test_login_rate_limiting)
Ensures that login is rate limited to 20 attempts per hour. ... ok
test_registration_rate_limiting (accounts.tests.RateLimitingTest.test_registration_rate_limiting)
Ensures that registration is rate limited to 10 attempts per hour. ... ok
test_throttle_scopes_are_separate (accounts.tests.RateLimitingTest.test_throttle_scopes_are_separate)
Test that registration and login throttles are separate. ... ok
test_revoke_token (accounts.tests.TokenLogTests.test_revoke_token)
Tokens can be revoked. ... ok
test_token_log_list (accounts.tests.TokenLogTests.test_token_log_list)
Lists token logs for the authenticated user. ... ok
test_used_token_shows_in_log (accounts.tests.TokenLogTests.test_used_token_shows_in_log)
Used tokens are visible in the token log. ... ok
test_access_without_token (accounts.tests.TokenTests.test_access_without_token)
Ensures access to /token_identity/ is forbidden without a token. ... ok
test_missing_callback_uri_or_redirect_uri_or_service_name (accounts.tests.TokenTests.test_missing_callback_uri_or_redirect_uri_or_service_name)
Ensures a missing callback_uri, redirect_uri, or service_name results in an error. ... ok
test_self_only_identity_hidden_from_token_access (accounts.tests.TokenTests.test_self_only_identity_hidden_from_token_access)
Ensures identities with visibility 'self_only' are not accessible even with user approval. ... ok
test_tampered_with_token_fails (accounts.tests.TokenTests.test_tampered_with_token_fails)
Ensures tampered tokens are rejected. ... ok
test_token_access_specific_identity (accounts.tests.TokenTests.test_token_access_specific_identity)
Ensures token scoped to one identity cannot access another. ... ok
test_token_is_single_use (accounts.tests.TokenTests.test_token_is_single_use)
Ensures tokens are single-use. ... ok
test_using_revoked_token_fails (accounts.tests.TokenTests.test_using_revoked_token_fails)
Using a revoked token fails. ... ok

----------------------------------------------------------------------
Ran 27 tests in 40.467s

OK
Destroying test database for alias 'default' ('file:memorydb_default?mode=memory&cache=shared')...
```

Figure 5.1: The output of backend tests.

CLI Tool (Moved to Appendix)

A custom Node.js CLI tool was created to automate and test the full identity sharing flow. It was especially useful early in development to simulate consent and token usage. However, the tool uses the flow where the user is expected to have a single identity per context. This has since been superseded by the updated requirement to allow the user to have multiple identities for each context and to choose the one to share. Additionally, the frontend now POSTs tokens directly to callback URIs, rendering the partially and fully manual CLI modes obsolete. For completeness, the tool and its capabilities are fully documented in Appendix A.2.

5.5 Reflections on Chosen Technologies

Django REST Framework was chosen as it significantly improves productivity by reducing boilerplate code. The class-based views and mixins in particular serve as base classes that provide much of the needed functionality through inheritance. In many cases, the inheritance hierarchies are complex and deep. For example,

`RetrieveUpdateDestroyAPIView` inherits from the `RetrieveModelMixin`, which expects an object to be retrieved from a URL parameter. In this project, account update and deletion were scoped to the currently authenticated user rather than a URL-specified object. This required overriding the `get_object()` method as shown in Listing 4.2. The deep inheritance hierarchies mean that it can be easy to introduce subtle bugs or vulnerabilities without a thorough grasp of the underlying logic. This highlights a key trade-off of using mature, object-orientated frameworks like DRF: the rapid development gained from deep inheritance hierarchies comes at the cost of a steeper learning curve to use its advanced features securely and correctly.

On the frontend, I initially decided to use both React and Tailwind CSS for the first time. Learning these two new technologies, while also developing a novel and secure SSO flow, caused significant delays and difficulties. The utility-first approach of Tailwind, in particular, proved to have a steep learning curve. This led to a pragmatic decision to switch to Chakra UI. While also a new technology, its component-based design and pre-built features (such as toasts, cards, and dark mode integration) made it much easier and faster to build a clean and consistent user interface.

Similarly, I used a custom implementation for handling token lifecycles in the frontend `AuthContext`, to maximise learning. This led to complex asynchronous challenges, including token refresh logic and the mitigation of race conditions. In hindsight, delegating this task to an established specialised library like `react-auth-kit` would have been more robust and sensible.

5.6 User Feedback and Usability Testing

Two rounds of user testing were conducted using the System Usability Scale (SUS) [9] to formally assess the usability of the ShareAware system. The surveys included instructions to go through the the applications core features, including registration and login, identity creation and sharing via the use of the Toy RP. The surveys also included targeted and open-ended questions to gather qualitative feedback on specific design choices and the overall user experience. The full surveys are available in Appendix A.3.

5.6.1 First Round of Testing

The initial survey was administered to peers on Slack, of which four became respondents.

Quantitative Findings (SUS Score)

The SUS scores were 75, 80, 60 and 65, resulting in a mean score of 70. For the SUS system, a score above 68 is considered above average and indicates good usability [9]. Whilst this is encouraging, the wide spread of scores (from 60 to 80), and the the small sample size of four respondents meant that the statistical significance is questionable. Instead, it was taken as an initial indicative finding. The quality of the qualitative insights is unaffected.

Qualitative Insights and Actionable Feedback

The open-ended feedback which was invaluable in identifying what the respondents felt worked well and areas that needed improvement. Respondents praised the "clean nice design" and the core concept of having "different profiles for different contexts", they also identified several areas for improvement:

- **Clarity of Terminology:** One user stated, "Self only vs Token only I do not understand those terms.", referring to the previous visibility labels. This directly prompted the update to the more intuitive labels: "Shareable" and "Unshareable".
- **Input Fields and User Experience:** One respondent found the free-text fields for some identity data confusing, asking "i don't get what do you mean by sexuality? seems not clear at all." This feedback, combined with another suggestion of adding more dropdowns, was addressed by adding a custom combobox component on the frontend. This component now provides users with common options for sexuality, relationship status, and gender, while still allowing them to enter a custom value.
- **Core Functionality:** All respondents expressed they would like to be able to have multiple identities per context. This was previously restricted to a single identity per context. Additionally, respondents wished they could select the identity to share, rather than the RP selecting by requesting a specific context (e.g., professional or social) through a URL parameter. Both requests were fully integrated as core requirements, significantly shaping the final design.

This first round of testing was very useful. It identified key usability issues and validated the project's core concept, allowing for targeted improvements that were validated in the second round of testing.

5.6.2 Second Round of Testing

Following the implementation of the first round feedback, a second, larger survey was conducted with 7 participants. This round tested the refined system, with the added functionality of users selecting the identity to share, multiple identities per context and improvements to the terminology and input fields.

Quantitative Findings (SUS Score)

The SUS scores were 100, 100, 100, 70, 62.5, 100, and 70. This resulted in a strong mean score of 86.07. This score is well above the average of 68, placing the system in the excellent tier for usability (80.3+) and suggests a high user satisfaction and ease of use.

Qualitative Insights and Actionable Feedback

The qualitative feedback from the second round reinforced the SUS score and validated the design improvements.

- **Overall Experience:** Users described the system as a "user friendly experience, intuitive and clean," praising its "simplicity" and "Clear, uncluttered

interface." The core feature of multiple contextual identity profiles was again highlighted as a major strength: "I like being able to select specific profiles for specific use cases."

- **Consent and Trust:** The consent mechanism, a critical component of the project's design, was received positively. Most respondents found the consent prompt "very clear," confirming it effectively explained exactly what data was being shared and with whom. Consequently, a majority of users stated they would "trust this system" for use with real services.
- **Validation of Design Choices:** One question targeted my assumption that adding the user defined label field to the identities to differentiate multiple identities with the same context category renders the context category redundant. All but one respondent felt that this is incorrect and that the context field should be kept. This emphasised to me the importance of gathering user feedback instead of relying on developer assumptions.
- **Reported Issues and Suggestions:** A user reported the system refusing a "auto generated OSX strong password" for being too long. Another user suggesting adding a filter for the contexts to help manage identities more easily.

In conclusion, both rounds of testing were crucial. The first round provided actionable feedback which led to significant improvements. The second round validated these changes and demonstrated a high level of usability and user trust in the final system.

5.7 Summary

The evaluation confirms that the ShareAware system successfully meets all of its core functional and security requirements. The comprehensive backend test suite provides strong evidence of the system's technical robustness and its resistance to threats such as token replay and tampering. Furthermore, the second round of user testing demonstrated a high degree of usability, with the final mean SUS score of 86.07 placing the system in the excellent tier. Qualitative feedback validated that the core concept of contextual identity is both well-understood and highly valued by users. While trade-offs in the snapshot implementation and token TTL were identified, the overall evaluation concludes that the project provides a strong, positive answer to the central research question:

> How can an identity system be designed and implemented to support secure, transparent, consent-driven, contextual identity sharing through a simplified SSO-like flow?

Chapter 6

Conclusion (658/1000 words)

The primary aim of this project was to address the limitations of existing federated identity protocols by designing and developing a system that supports secure, transparent, and consent-driven identity sharing. The research problem was derived from the observation that whilst protocols like OAuth 2.0 and OpenID Connect (OIDC) offer theoretical security, their real-world implementations are often complex, prone to privacy leaks and lack explicit user consent mechanisms. The ShareAware system was created as a targeted solution for this gap, focusing on simplicity, user control and the management of multiple contextual identities.

The project began with a critical literature review that synthesised academic findings to inform the design. This research highlighted recurring vulnerabilities in the establish protocols and a widespread lack of genuine user consent, which directly informed the core design principles of ShareAware. Including the use of short-lived, single-use, scoped, and signed JWTs, to its explicit, per-request consent flow.

The design phase resulted in a system with clear requirements aligned with key security and privacy principles, particularly GDPR. The foundational idea that identity is not monolithic was realised by allowing users to create distinct identities for different contexts. This feature that directly addresses the context collapse problem prevalent in mainstream SSO providers. By combining this with a per-request, explicit consent model, the system presents an architecture that successfully mitigates many of the privacy and security challenges identified in traditional IdPs.

The development phase had its challenges, and the evaluation chapter provides reflections on these. For instance, the delays and difficulties faced due to learning React and Tailwind simultaneously, whilst also developing a custom SSO flow. This led to a pragmatic decision to switch to Chakra UI. This pivot significantly improved development speed and UI consistency and illustrates that a successful project is not about perfect planning and flawless execution, but instead relies on pragmatic problem solving and a willingness to adapt.

The final evaluation confirmed that the system successfully achieved its objectives. The comprehensive test suite provided strong evidence of a robust and secure implementation, and two rounds of user testing validated the system's usability. The final mean SUS score of 86.07 places the system in the excellent tier, and qualitative feedback confirmed that users found the consent mechanism clear and trustworthy.

This process validated the core concepts of the project and demonstrated a high level of user satisfaction.

In conclusion, the ShareAware project successfully demonstrates that it is possible to build a simplified, privacy-respecting alternative to entrenched federated identity protocols. By placing explicit informed consent at the heart of its flow, it offers a strong proof-of-concept for a more user-focused approach to identity management.

Future Work

Although the project provides a solid foundation, several opportunities for future work exist to enhance its capabilities and move it closer to a production-ready system. Important improvements include implementing a registration and verification system for relying parties to mitigate phishing risks and incorporating additional security functionality such as Multifactor Authentication (MFA) and password reset.

Furthermore, user control could be made even more granular by implementing field-level visibility controls. This would introduce an intermediary step in the consent flow where, after selecting an identity, the user could check or uncheck individual fields before granting consent. However, for this feature to be truly effective, it would be a prerequisite to first solve the snapshot immutability challenge discussed in the evaluation. Saving a full resolution copy of the image at the time of consent would be necessary to ensure that the identity data shared with the relying party is identical to that for which the user approved. With that prerequisite met, snapshots could then be created that contain only the user-approved fields. These snapshots would then be sent instead of live identity data, ensuring better adherence to the principle of data minimisation and improving auditing accuracy. However, this introduces an additional step for the user. A future implementation should be preceded by user testing to validate that the benefits of field-level visibility outweigh the potential inconvenience.

Appendix A

Appendix

A.1 Wireframes

The following are wireframes generated using uxpiliot.ai and then heavily modified to suit the project.

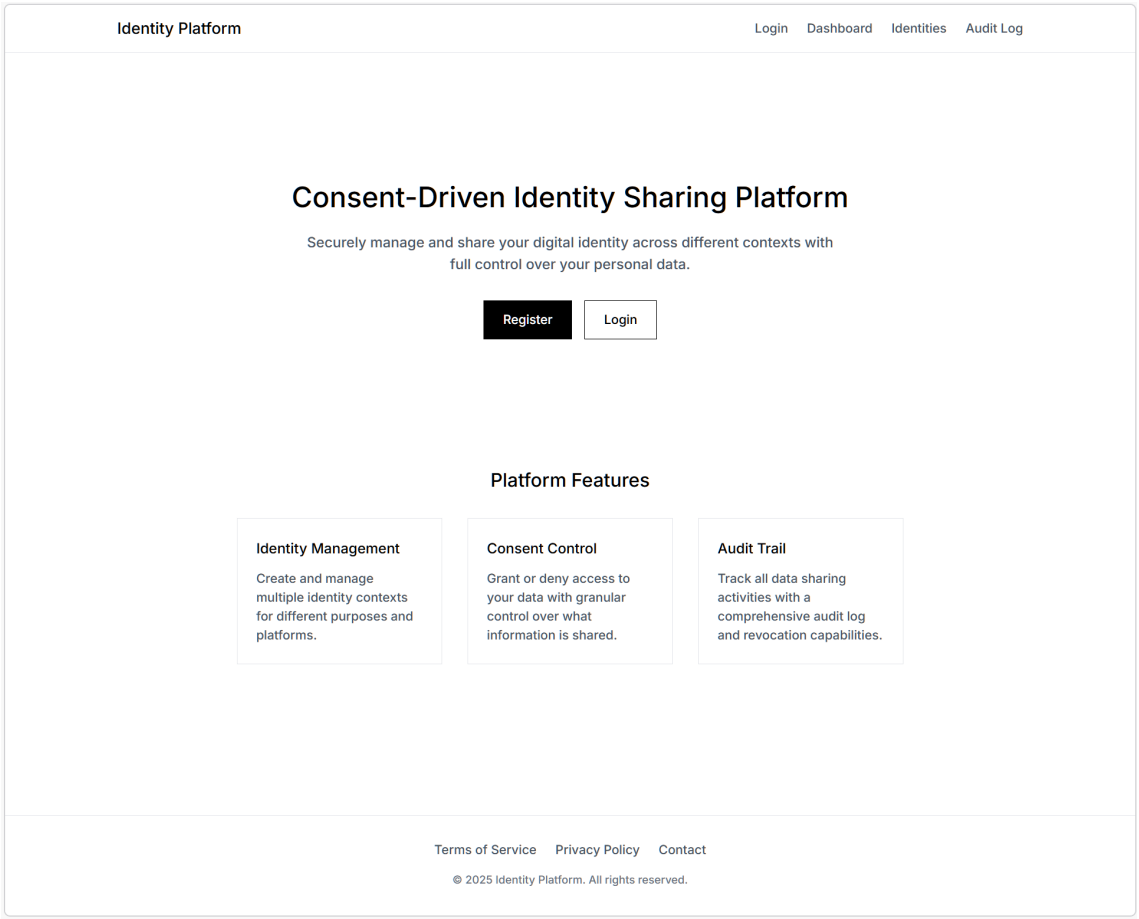


Figure A.1: landing page.

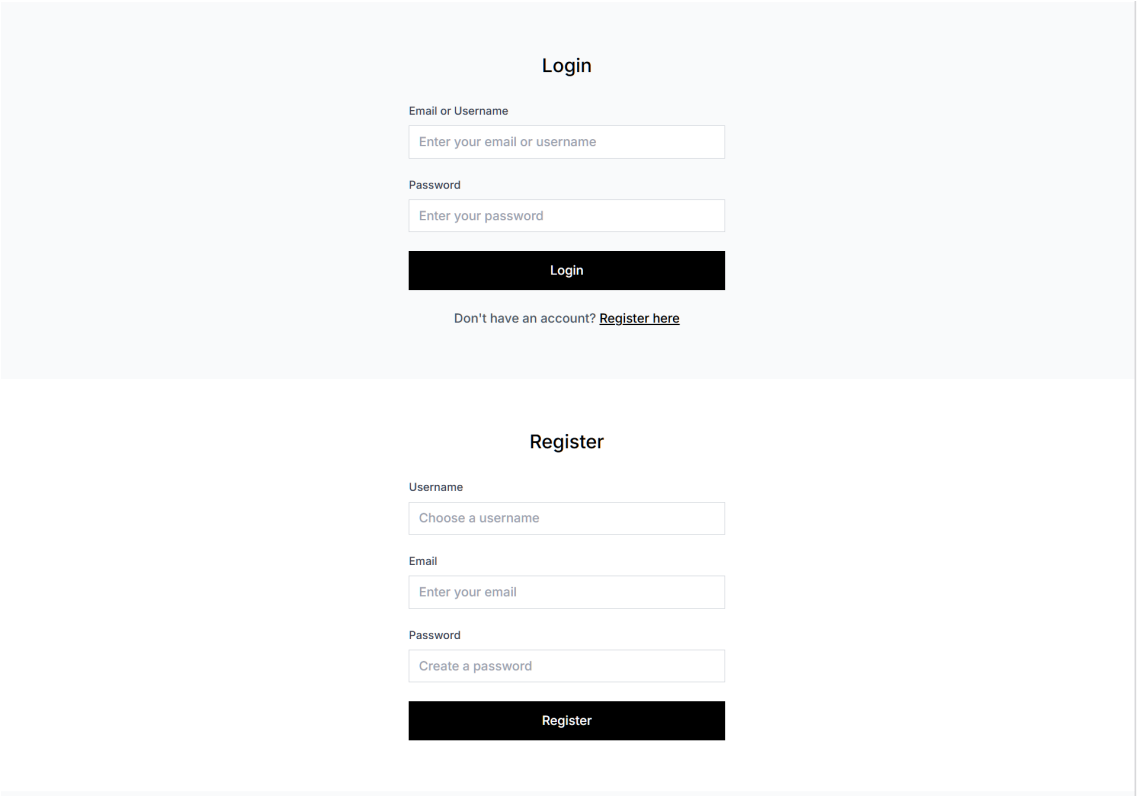


Figure A.2: Login and Registration partials.

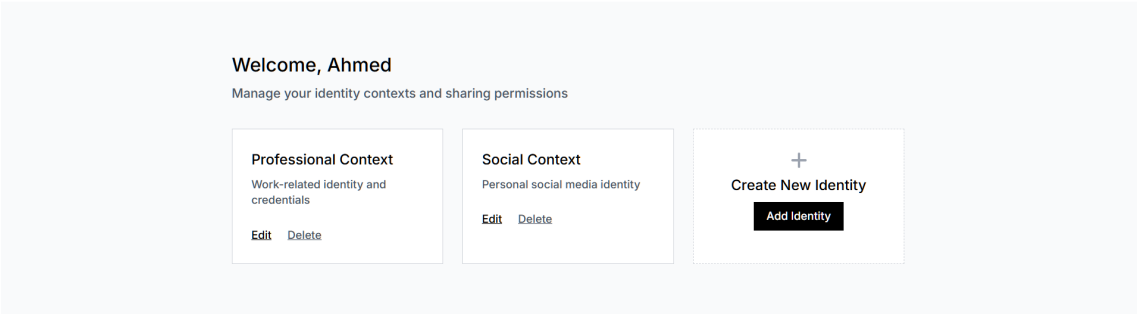


Figure A.3: Dashboard partial.

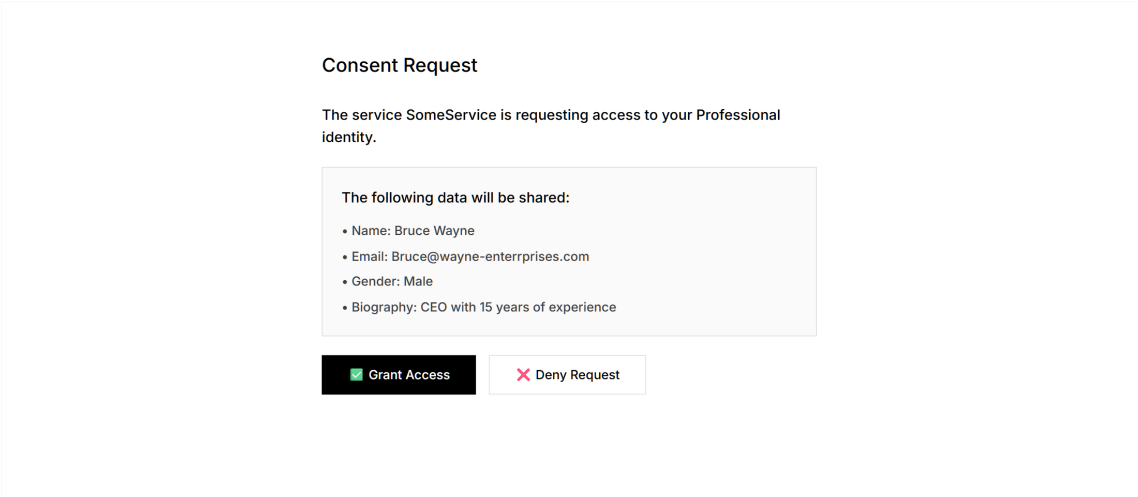


Figure A.4: Consent Page Partial.

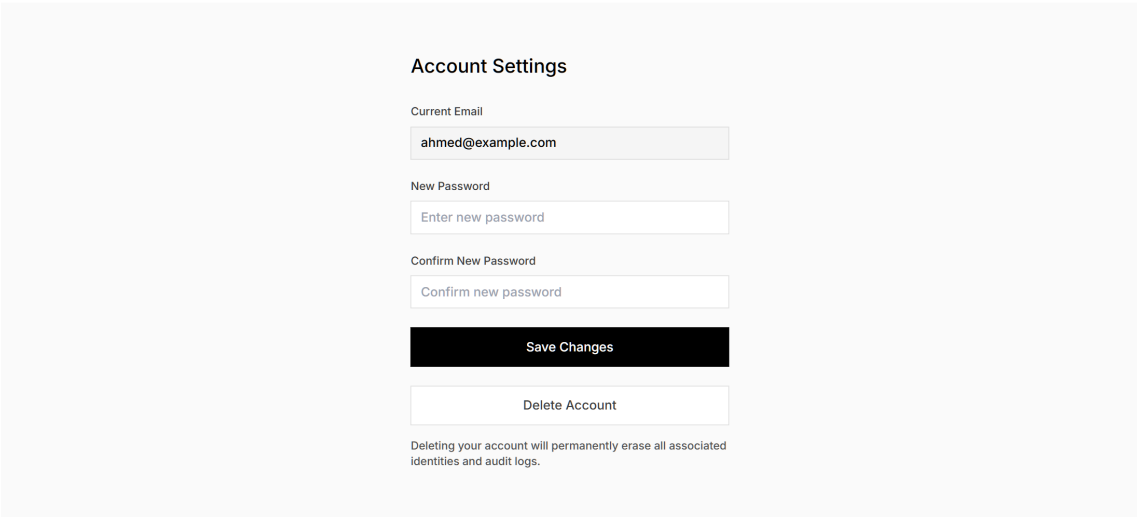


Figure A.5: Account page partial.

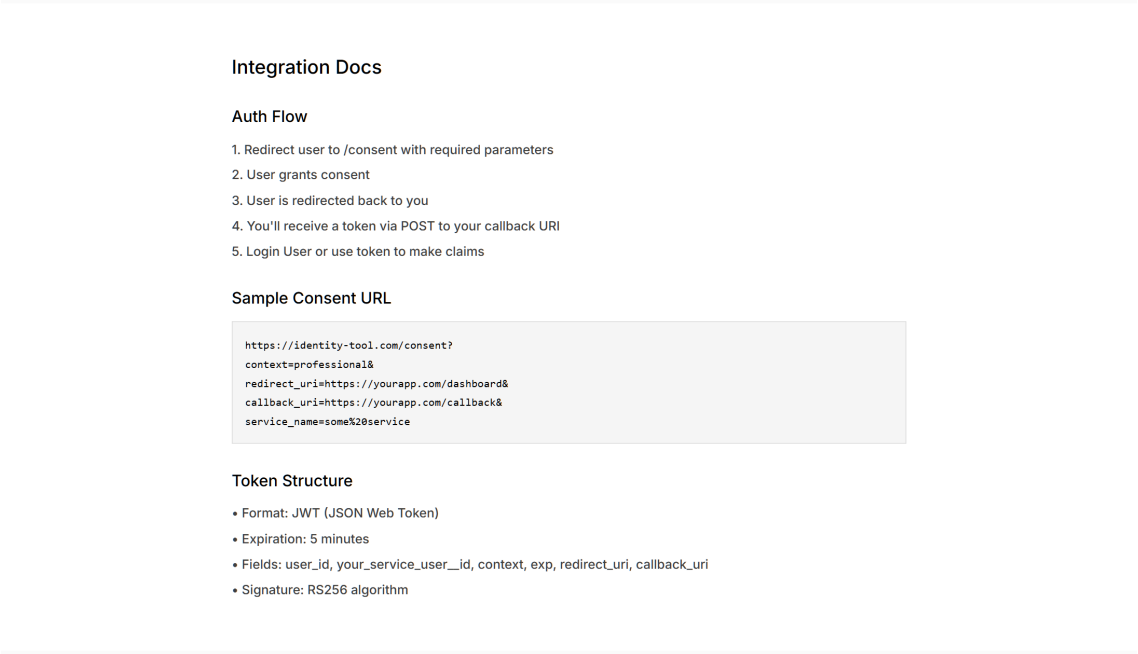


Figure A.6: Integration documentation partial.

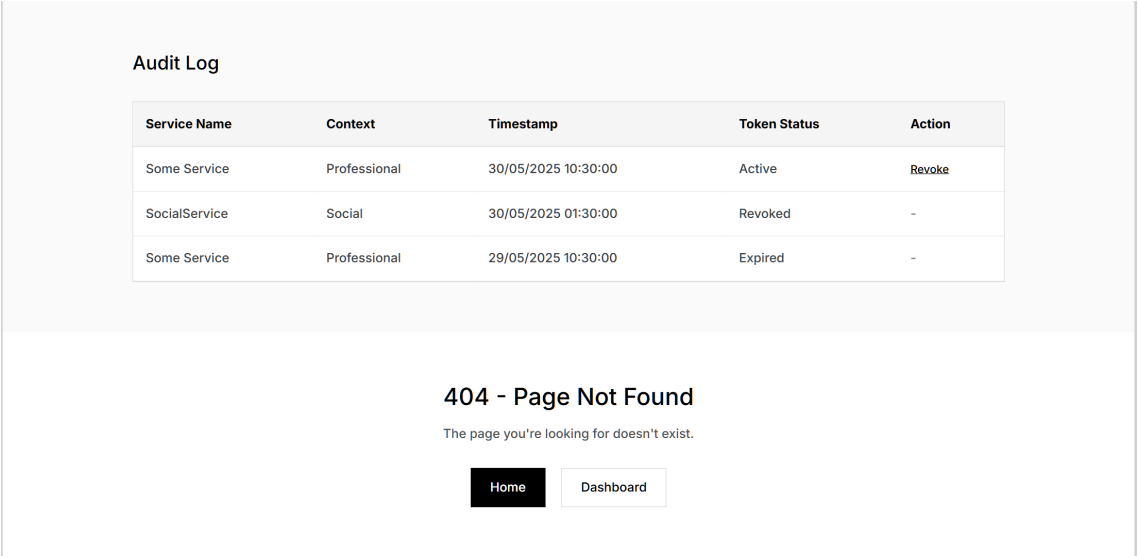


Figure A.7: Partials of the Audit log and the 404 pages.

A.2 CLI tool

A custom Node.js CLI tool was created to simplify testing of the full identity sharing flow in the early stages of development. The tool simulates this by:

- Registering or logging into a test user.
- Creating multiple contextual identities.
- Requesting a scoped token via the consent flow.
- Fetching the shared identity data using the issued JWT.

The CLI supports fully automated, partially automated, and fully manual modes. Figures A.8, A.9, A.10 and A.11 show the flow for each mode and the output of the fully automated mode.

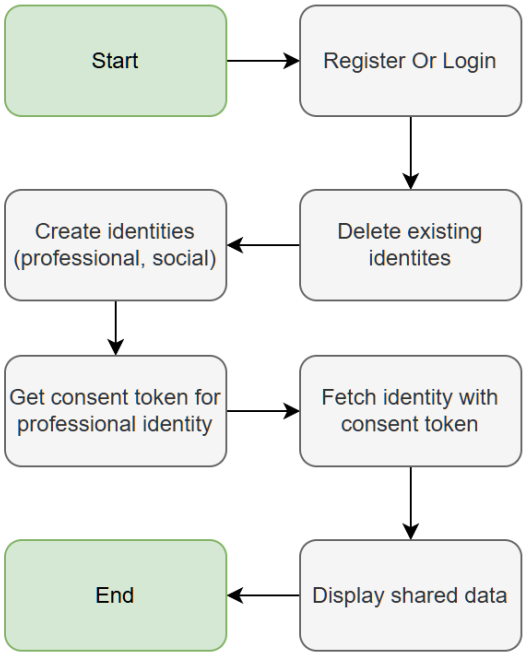


Figure A.8: Fully automated flow.

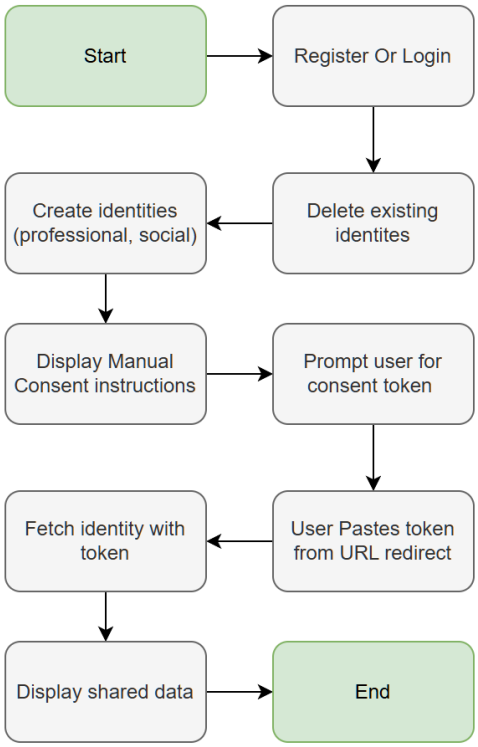


Figure A.9: Partially automated flow.

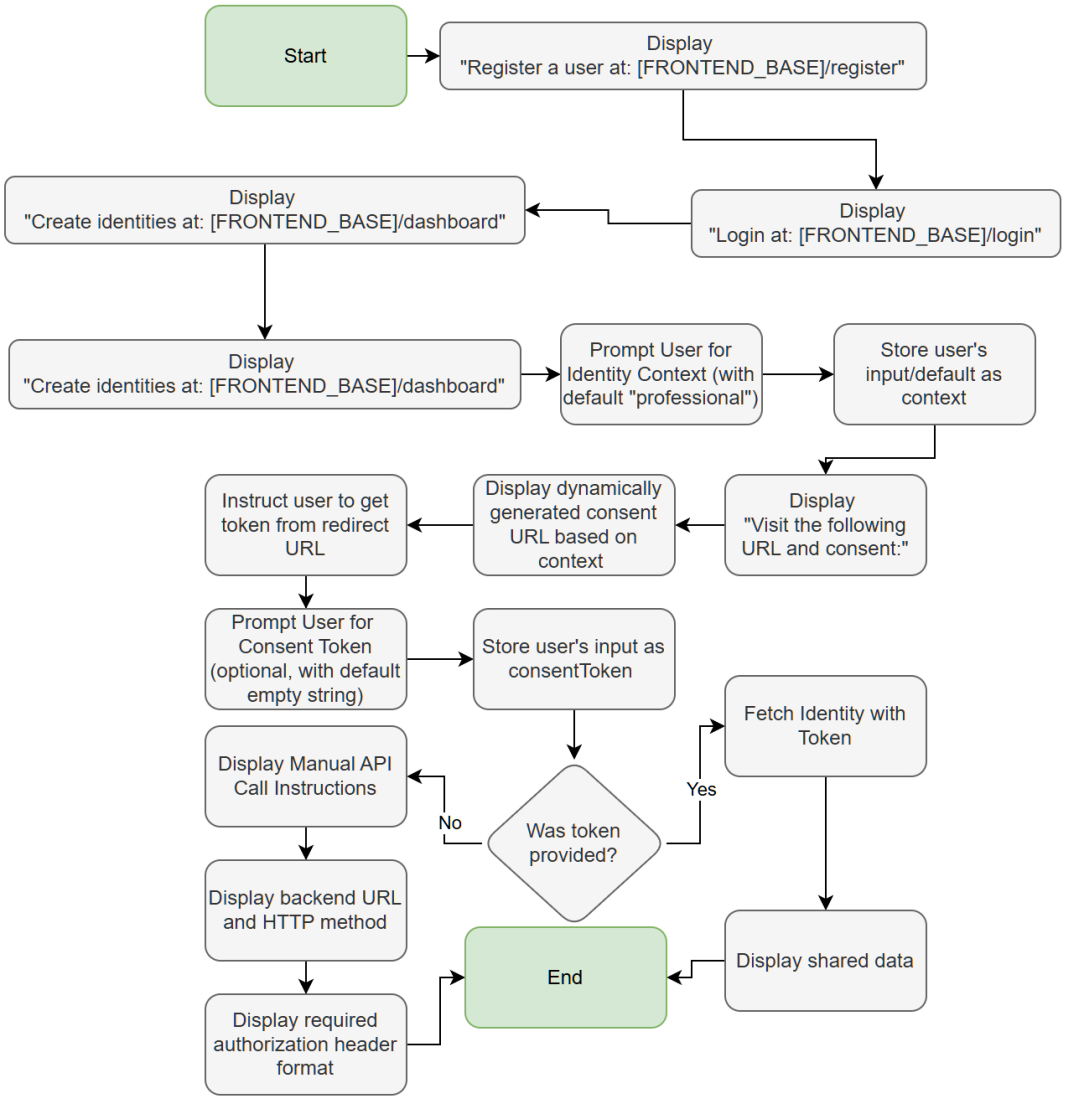


Figure A.10: Manual flow.

```
PS C:\Users\Ahmed\OneDrive\Desktop\final_project\identity_project> node .\quick_tests\auth_flow_test.js
dentity tool testing
    Press ENTER to go to next step
    Launch the backend and frontend then press ENTER
    Both have a README.md file with run instructions

Paste frontend base URL (Press ENTER to use "http://localhost:5173"):
Paste backend base API URL (Press ENTER to use "http://127.0.0.1:8000/api"):
Frontend base URL: http://localhost:5173
    Backend base URL: http://127.0.0.1:8000/api
    How do you want to continue?
    1: Fully automated
    2: Partially automated (You have to consent and get the token from the URL manually)
    3. Fully manual (You have to do everything yourself with step by step instructions)
Enter your choice (1, 2 or 3): 1
You chose fully automated mode.
Running automated tests...
 Registering...
❌ API error at http://127.0.0.1:8000/api/register/ (400):
{"username":["A user with that username already exists."],"email":["user with this email already exists."]}
User already exists. Logging in...
Logging in...
✅ Done
Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzQ5NDgxNjgwLCJpYXQiOjE3NDk
6NH0.0CtlVwtIJ9V9OpQHeR-0wwRI5tTEW6KiHuSBeT1RHX8
Deleting existing identities...
Deleted 2 identities
✅ Done
Creating professional identity...
✅ Done
Creating social identity...
✅ Done
Getting consent token for professional identity...
✅ Done
Fetching identity using consent token...
✅ Done
Shared data:
{
  context: 'professional',
  name: 'Bruce Wayne',
  email_address: 'bruce@wayne-enterprises.com',
  biography: 'CEO of Wayne Enterprises'
}
All tests completed successfully.
```

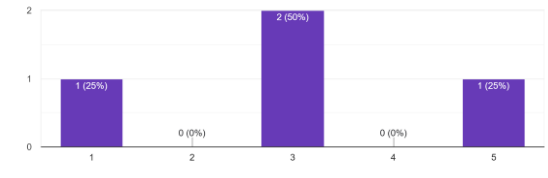
Figure A.11: CLI tool.

The manual mode of the tool acts as an end-to-end test case specification, going through each step in the flow for reproducible evaluation.

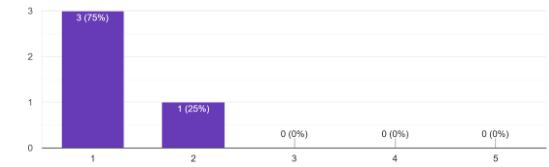
A.3 Survey Results

A.3.1 First Round

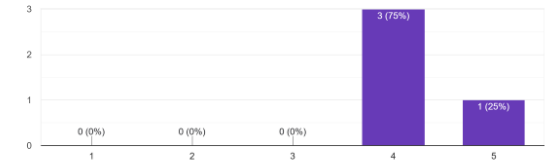
I think I would like to use this system frequently.
4 responses



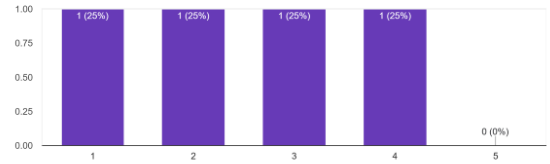
I found the system unnecessarily complex.
4 responses



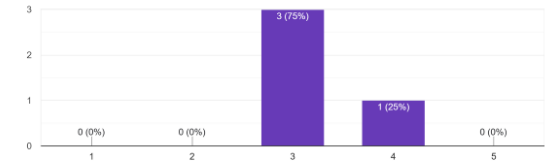
I thought the system was easy to use.
4 responses



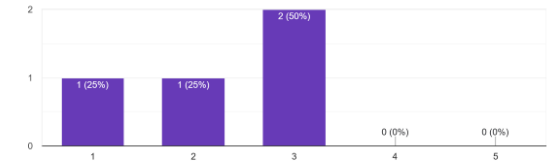
I think I would need support to use this system.
4 responses



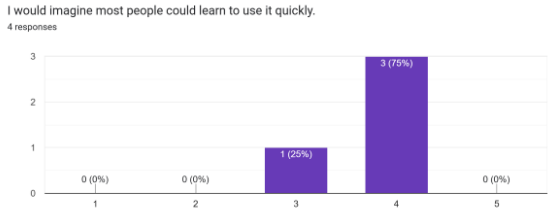
I found the various functions well integrated.
4 responses



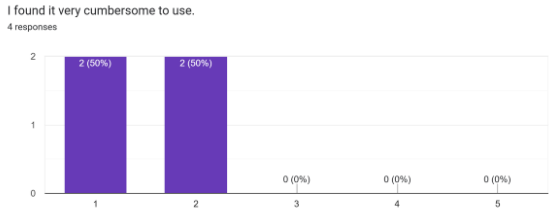
I thought there was too much inconsistency.
4 responses



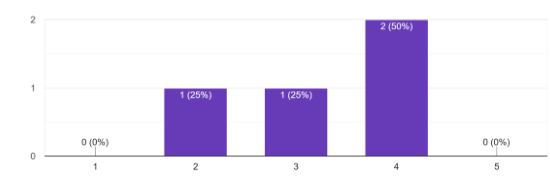
I would imagine most people could learn to use it quickly.
4 responses



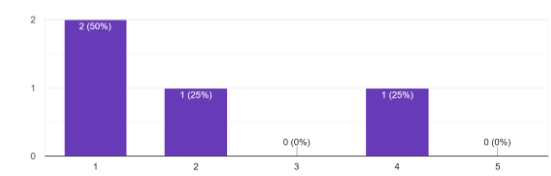
I found it very cumbersome to use.
4 responses



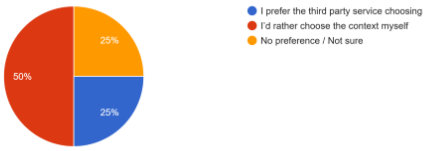
I felt confident using the system.
4 responses



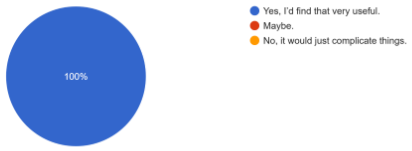
I needed to learn a lot before I could get going.
4 responses



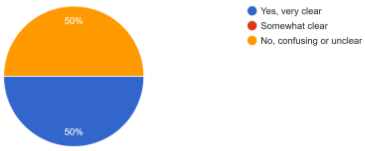
Currently, the third party service selects which identity context (professional, gaming, etc) to use. Would you prefer:
4 responses



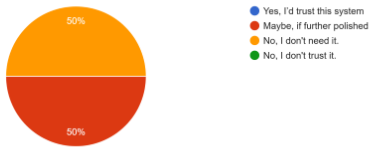
Would you find it useful to create more than one identity for the same context? (e.g., two "professional" profiles - one for freelancing, one for job-hunting)
4 responses



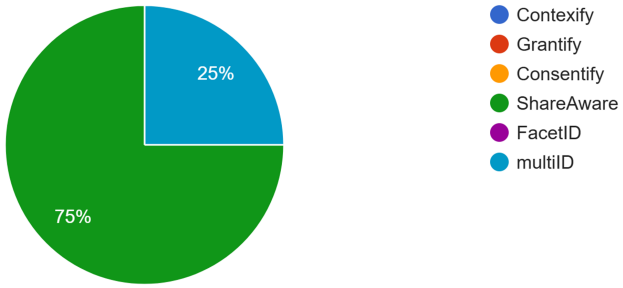
Did the consent prompt clearly explain what was being shared and to whom?
4 responses



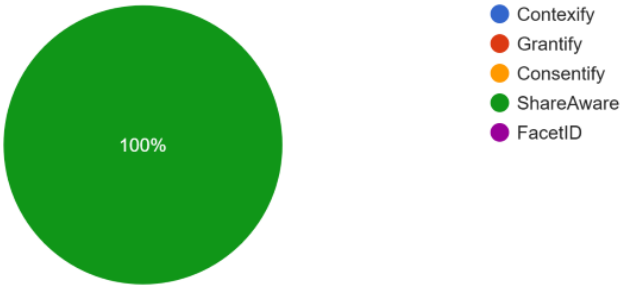
Would you feel comfortable using this for real services?
4 responses



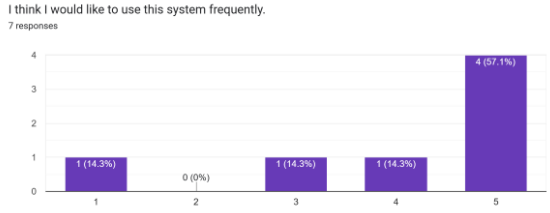
I haven't decided on the app name yet. Which do you prefer or do you have any suggestions?
4 responses

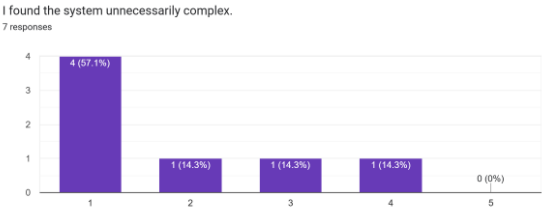


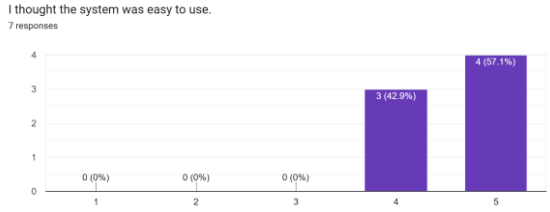
I haven't decided on the app name yet. Which do you prefer or do you have any suggestions?
3 responses

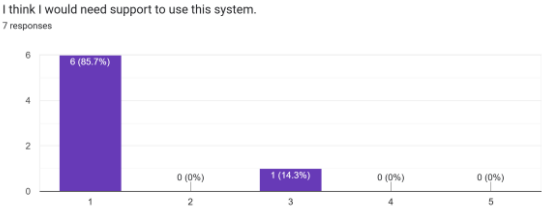


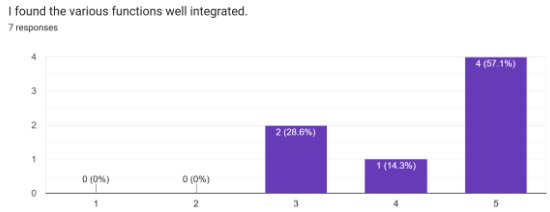
A.3.2 Second Round

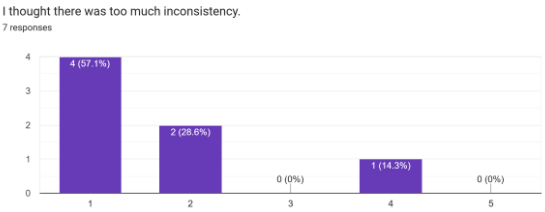


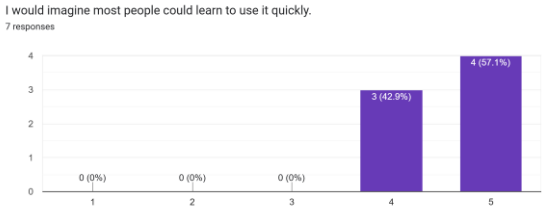


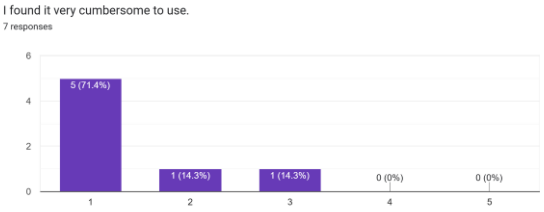


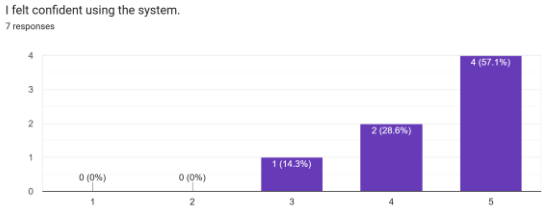


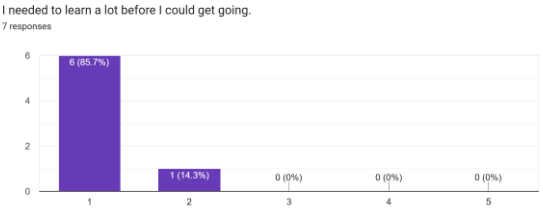




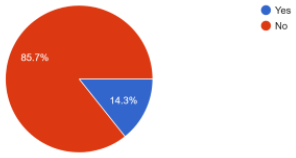




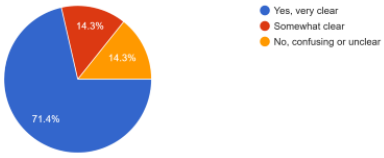




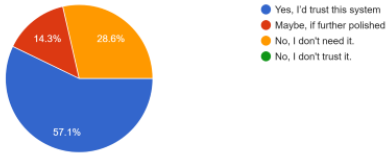
You are required to label each identity you create. This is to help differentiate two identities with the same context (e.g., professional). Do you feel this ...sulted in the predefined contexts being redundant?
7 responses

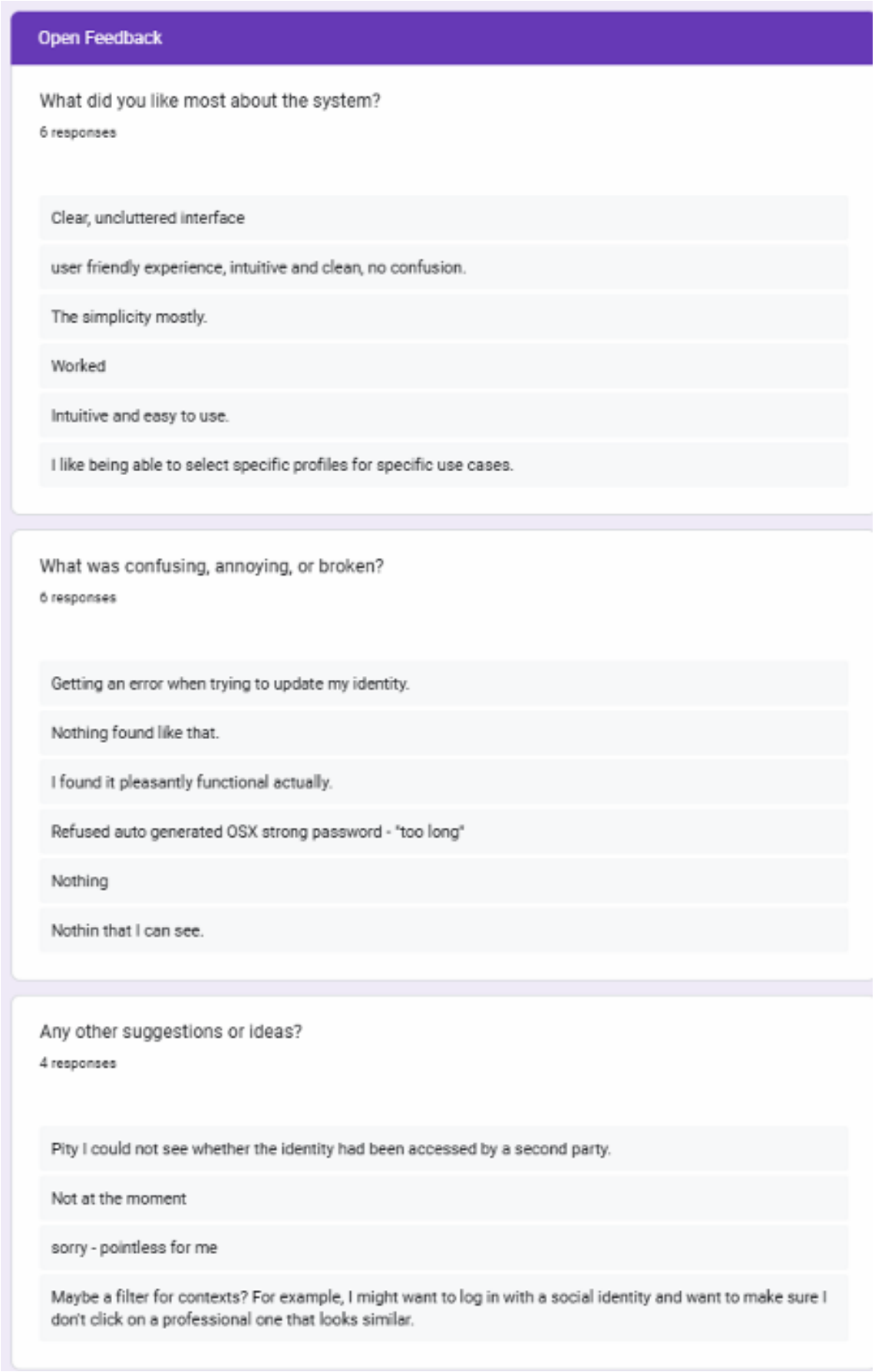


Did the consent prompt clearly explain what was being shared and with whom?
7 responses



Would you feel comfortable using this for real services?
7 responses





Open Feedback

What did you like most about the system?

6 responses

- Clear, uncluttered interface
- user friendly experience, intuitive and clean, no confusion.
- The simplicity mostly.
- Worked
- Intuitive and easy to use.
- I like being able to select specific profiles for specific use cases.

What was confusing, annoying, or broken?

6 responses

- Getting an error when trying to update my identity.
- Nothing found like that.
- I found it pleasantly functional actually.
- Refused auto generated OSX strong password - "too long"
- Nothing
- Nothin that I can see.

Any other suggestions or ideas?

4 responses

- Pity I could not see whether the identity had been accessed by a second party.
- Not at the moment
- sorry - pointless for me
- Maybe a filter for contexts? For example, I might want to log in with a social identity and want to make sure I don't click on a professional one that looks similar.

Gantt Chart

Figure A.12 illustrates the planned progression of this project, including core features, stretch goals, and delivery milestones.

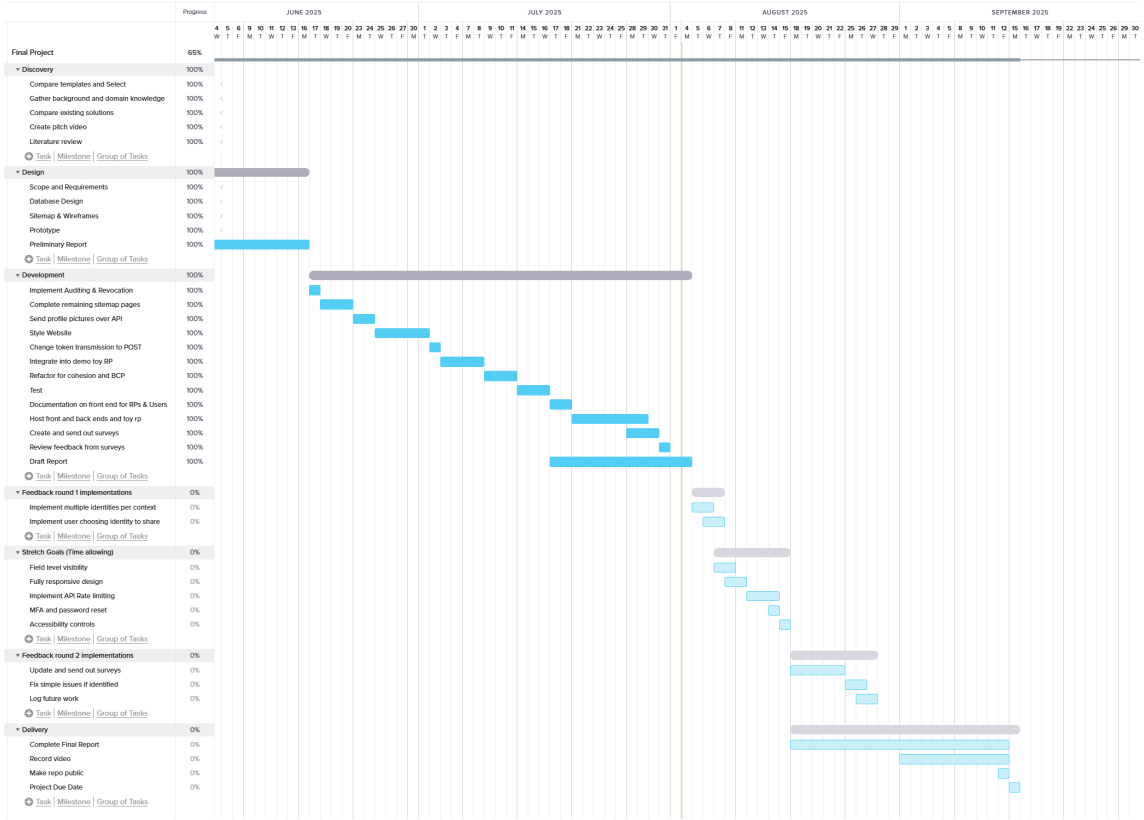


Figure A.12: Gantt Chart.

Bibliography

[1] European Union. 2024. What is GDPR, the EU's general data protection regulation? Accessed: 2025-05-27. (2024). https://gdpr.eu/what-is-gdpr/.

[2] Alice Marwick and danah boyd danah. 2010. I tweet honestly, i tweet passionately: twitter users, context collapse, and the imagined audience. *New Media and Society*, 20, (July 2010), 1–20. DOI: 10.1177/1461444810365313.

[3] Helen Nissenbaum. 2004. Privacy as contextual integrity. *Washington Law Review*, 79, (May 2004).

[4] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2017. The web SSO standard OpenID Connect: in-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, 189–202. DOI: 10.1109/CSF.2017.20.

[5] Maximilian Westers, Andreas Mayer, and Louis Jannett. 2024. Single sign-on privacy: we still know what you did last summer. In *2024 Annual Computer Security Applications Conference (ACSAC)*, 321–335. DOI: 10.1109/ACSAC63791.2024.00039.

[6] Pieter Philippaerts, Jan Vanhoof, Tom Van Cutsem, and Wouter Joosen. 2024. Is your OAuth middleware vulnerable? evaluating open-source identity providers' security. In *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, 3607–3612. DOI: 10.1109/GLOBECOM52923.2024.10901500.

[7] Adiva Fiqri Nugraha, Herman Kabetta, I Komang Setia Buana, and R Budiarto Hadiprakoso. 2023. Performance and security comparison of JSON Web Tokens (JWT) and platform agnostic security tokens (PASETO) on RESTful APIs. In *2023 IEEE International Conference on Cryptography, Informatics, and Cybersecurity (ICoCICs)*, 15–22. DOI: 10.1109/ICoCICs58778.2023.10277377.

[8] European Union. 2025. General Data Protection Regulation (GDPR) – Official Legal Text. Accessed: 2025-05-29. (2025). https://gdpr-info.eu/.

[9] John Brooke. 1996. SUS: a "quick and dirty" usability scale. In *Usability Evaluation in Industry*. P. W. Jordan, B. Thomas, B. A. Weerdmeester, and A. L. McClelland, (Eds.) Developed at Digital Equipment Corporation, UK; widely used ten-item usability scale. Taylor and Francis, London, UK, 189–194.