The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Systems and Society

Exploring Fine-Grained Access Control in Solid Pods: Implementation Perspectives on Access Control Policies (ACP)

— 6 pt research project (S2 2025)

A report submitted for the course
COMP3740, Individual Computing Project

By:
Gusti Rais

Supervisors:
Dr. Graham Williams
Dr. Anushka Vinadage

September 2025

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the Academic Integrity Rule;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

September (2025), Gusti Rais

Acknowledgements

If you wish to do so, you can include some Acknowledgements here. If you don't want to, just comment out the line where this file is included.

There is absolutely no need to write an Acknowledgement section, so only do so when you *want* to – it's always important to stay sincere. One reason for including an acknowledgement could be to thank your supervisor for extraordinary supervision (or any other reason you deem noteworthy). Some supervisors sacrifice a lot, e.g., are always available, meet on weekends, provide multiple rounds of corrections for theses reports, or the like (keep in mind that writing a thesis is special for you, but not for them, so they do actually not have any reason to sacrifice their private time for this!). Seeing acknowledgements in this report can feel like a nice appreciation of this voluntary effort. For large works that form the end of some studies (like an Honours or Master thesis), it is also not uncommon to read acknowledgements to one's parents or partner. But again, completely optional!

Abstract

This project presents the design and implementation of *SolidTasks*, a decentralised task manager built on the Solid Ecosystem, with a focus on fine-grained data sharing using Access Control Policies (ACPs) that are pre-defined in the Solid Ecosystem. SolidTasks allows users to create and manage personal tasks within their Solid Pods, ensuring that individuals retain ownership and control over their data. The project investigates the limitations of the traditional Access Control Lists (ACL) model, particularly its resource-attached and individualistic nature in the context of Solid resources, and contrasts this with the more flexible matcher-based ACP framework. Through refactored code in SolidTasks' codebase, the system enables task storage in individual Turtle files with automatic generation of Access Control Resources (ACR). Findings suggest that ACP not only streamlines access management in Solid applications, but is also established for future-ready decentralised collaboration tools.

Contents

Chapter 1

Introduction

The Solid protocol is a draft specification for a new age of the Web as designed by Sir Tim Berners-Lee. The initiative aims to standardise how user information can be decentralised and stored in Personal Online Data Stores (PODs). These PODs live inside Solid servers which are decentralised points where identity providers give users one WebID per POD that they have on that Solid server. A Solid server supports at least one of two authorisation specifications outlined by the Solid Foundation. These protocols are the Web Access Control (WAC) specification as well as the Access Control Policy (ACP) language. The point of these specifications is so that a user can declare access control rules over resources in their pod. Access Control Lists (ACLs) are the underlying model, while Web Access Control (WAC) is a decentralised system that uses the ACL model to set authorisation conditions on HTTP resources.

ACP is a novel implementation that aims to be more user-friendly for the end user and implement finer-grained control mechanisms. Unlike WAC, which relies on static ACLs directly attached to resources, ACP introduces the concept of policies and matchers. Policies define the conditions under which access is granted, while matchers allow those policies to be flexibly applied across groups of resources or users. This separation of concerns makes ACP more expressive, scalable, and adaptable to real-world scenarios where access control often extends beyond simple read–write–append rules.

In practice, both WAC and ACP empower individuals to regain ownership of their data by enabling them to specify who can access, use, or share particular resources. This decentralised design embodies the broader vision of Solid: to shift the web from a platform where data is siloed by service providers to one where individuals maintain sovereignty over their personal information. By standardising identity, storage, and access control, Solid not only redefines user privacy and interoperability but also lays the groundwork for a new generation of decentralised applications.

This project aims to design and develop a user interface for Solid Pods that enables fine-grained access control whose effectiveness can be evaluated through established usability metrics. In particular, the System Usability Scale (SUS) will be employed as a standardised measure of usability, providing a reliable method for benchmarking user experience (Brooke, 1996). By integrating ACP into the interface, the project not only explores the technical implementation of fine-grained authorisation but also examines the human factors that determine its practical adoption. Prior research emphasises that the usability of security mechanisms is a critical factor in ensuring user acceptance and long-term sustainability of decentralised systems (Cranor & Garfinkel, 2005). Thus, by combining interface design with systematic usability evaluation, this project contributes to understanding how Solid can become a more user-friendly platform for managing personal data in a decentralised web ecosystem.

Chapter 2

Background

2.1 Solid and Personal Online Data Stores

Solid (Social Linked Data) is a web decentralization project led by Tim Berners-Lee, the inventor of the World Wide Web, originally developed collaboratively at the Massachusetts Institute of Technology (MIT). The project "aims to radically change the way Web applications work today, resulting in true data ownership as well as improved privacy" by developing a platform for linked-data applications that are completely decentralized and fully under users' control rather than controlled by other entities.

Solid fundamentally restructures how web applications handle data by separating data storage from application logic. Users have personal online data stores called Pods. Solid applications can read and write documents directly from your Pod. You choose which applications to use with your data rather than being locked into specific platforms. Your data remains yours even when you change providers or try new services or applications.

In Solid, each user stores their data in an online storage space that we call a personal online datastore (pod). Pods are Web-accessible storage services, which can either be deployed on personal servers by the users themselves, or on public servers by pod providers similar to current cloud storage providers (e.g., Dropbox). A user may distribute personal information among several pods; for example, different pods might contain personal profile data, contact information, financial information, health, travel plans, or other information.

The platform provides authenticated access through standard web protocols. Solid applications read and write data stored in users' pods via RESTful HTTP operations. Besides Linked Data Platform (LDP) support, Solid servers may offer optional SPARQL support. This architectural approach ensures that the user retains complete ownership and control of data in the user's pods: what data each pod contains, where each pod is stored, and which applications have permission to use the data.

2.2 Linked Data Platform (LDP)

Solid builds on the W3C Linked Data Platform (LDP) model for read/write RDF resources over HTTP: LDP defines containers and resource semantics that enable RESTful creation, modification and deletion of linked-data documents, which Solid uses as a foundation for resource layout and client interactions.

2.3 Access control in Solid: WAC, ACP and ACRs

Solid's access-control ecosystem has evolved from Web Access Control (WAC) to a richer Access Control Policy (ACP) model on September 29th 2022. ACP introduces Access Control Resources (ACRs) and explicit policy constructs (policies, matchers, and access modes) that are stored as RDF alongside resources; these allow resource-centric, queryable policy defini-

tions that can express fine-grained allowances such as agent matchers and temporal constraints. A practical specification and examples for ACP/ACR are available in the Solid authorization panel.

2.4 Why Solid?

Solid addresses fundamental problems with the current centralized web architecture, offering a compelling vision for data sovereignty and user empowerment. The platform's importance stems from several converging factors that highlight the urgent need for web decentralization.

2.4.1 Data Ownership and Privacy Crisis

Sir Tim-Berners Lee was saddened by the Facebook-Analytica scandal, when Facebook in 2012 conducted psychological experiments on nearly 700,000 users in secret, became public as of 2016. From then on, he stressed to companies like Facebook the importance of personal privacy.

"When I invented the World Wide Web, I envisioned technology that would empower people and enable collaboration. Somewhere along the way, we lost that human-first approach; today's web often prioritises profits over people. Solid returns the web to its roots by giving everyone direct control over their own data." – Sir Tim Berners-Lee

Chapter 3

Related Work

The emergence of decentralized web technologies has sparked significant research interest in personal data sovereignty, access control mechanisms, and user-centric system design. This section reviews the foundational literature that informs the design and evaluation of SolidTasks, examining work across three key domains: the Solid ecosystem and decentralized data management, access control frameworks for web resources, and usability evaluation of security systems.

3.1 Decentralized Personalized Data Management

The Solid project represents a paradigm shift toward data decentralization, with Berners-Lee et al. (2016) outlining the foundational vision of re-decentralizing the web through Personal Online Data Stores (PODs). This work established the core principles of data ownership, interoperability, and user agency that underpin the Solid specification. Subsequent research by Sambra et al. (2016) formalized the technical architecture through the Linked Data Platform (LDP) specification, demonstrating how RESTful interactions with RDF resources could enable seamless data sharing across applications.

Verborgh and Dumontier (2016) expanded on these foundations by examining the implications of decentralized data architecture for application development, highlighting both the opportunities for user empowerment and the challenges of managing distributed resources. Their analysis of the trade-offs between centralized and decentralized systems provides crucial context for understanding why fine-grained access control becomes essential in POD-based applications. More recently, Taelman et al. (2018) conducted empirical studies on Solid's performance characteristics, revealing that while decentralization offers privacy benefits, it introduces complexity in data management that necessitates sophisticated access control mechanisms.

3.2 Other paradigms for decentralised data management.

Two systems that are relevant when discussing ACl and ACP in Solid are object-capability systems and federated protocols. Object-capability and capability0based access systems present a fundamentally different approach to authorizaion by issuing references that serve as the permission itself, such as bearer capabilities that can be attenuated and delegated. These systems are directly comparable to Solid's ACP model because they offer an alternative to identity-bound ACLs, with capabilitiy URIs encoding *least-privilege rights* including methods, paths, and expiry dates while supporting transitive delegation without requiring global policy lookup. Compared to Solid's ACP approach using policies and matchers, capability systems simplify enforcement by eliminating ambient authority, though they introduce conmplexities in discovery, capability lifecycle management, and revocation *at scale.*

Federated protocols like ActivityPub, Matrix, and IMAP/JMAP represent another relevant approach to decentralized access control that contrasts meaningfully with Solid's model. These protocols decentralize access control across autonomous servers while preserving account mobil-

ity, with authorization mediated through server-local policies and cross-server trust mechanisms such as signed activities and server ACLs. It has been recorded that they achieve strong interoperability for domain-specific data like social timelines and chat messages, they offer less uniform, cross-application semnatics for per-resource polivies compared to Solid's Linked-Data native model. Unlike Solid's resource-centric ACL/ACP approach that enables fine-grained, queryable policies, federated protocols rely more on protocol compatibility rather than per-resource RDF contracts for data portability, highlighting the trade-offs between different authorization philosophies in decentralized systems.

3.3 Access Control in Solid

The evolution of access control mechanisms in Solid represents a critical transition from traditional centralized authorization models toward decentralized, resource-centric control systems. The Solid ecosystem has developed two primary access control models: Web Access Control (WAC) and Access Control Policy (ACP), each representing different approaches to managing fine-grained permissions in decentralized web environments.

3.3.1 Web Access Control (WAC) Model

Web Access Control (WAC) emerged as Solid's foundational authorization framework, based on Tim Berners-Lee's 2009 proposal for decentralized access control systems (Capadisli et al., 2022). WAC provides "a decentralized cross-domain access control system providing a way for Linked Data systems to set authorization conditions on HTTP resources using the Access Control List (ACL) model" (Solid Community Group, 2024). The system operates through Authorization statements that explicitly link resources to access permissions, using RDF-based policies stored in Access Control Lists.

The WAC model follows traditional ACL patterns where authorization statements contain explicit wac:accessTo predicates that directly reference the protected resources. This approach creates a bidirectional relationship where ACL documents point to resources, and resources link back to their controlling ACL documents through HTTP Link headers with rel="acl" attributes (Berners-Lee et al., 2024). WAC supports four fundamental access modes: Read, Write, Append, and Control, which map directly to HTTP operations and enable fine-grained permission management (Capadisli Berners-Lee, 2022).

A significant feature of WAC is its hierarchical inheritance mechanism, where "a member resource inherits Authorizations from the closest container resource (heading towards the root container)" (Solid Community Group, 2024). This recursive property simplifies permission management by allowing default policies to cascade down directory structures while supporting resource-specific overrides when needed. However, this inheritance model introduces complexity in policy discovery, as determining the effective ACL resource requires traversal up the container hierarchy until an ACL document is found (Story, 2021).

Chapter 4

Design And Architecture

SolidTasks is a flutter (Riverpod) client that stores resources as a first-class Linked Data resource in the user's Solid Pod. The app authenticates via Solid OIDC, performs LDP-compliant CRUD over HTTP per task turtle files, and manages fine-grained access using a choice of Access Control List or Access Control Policy. by writing Access Control Resources (ACR) alongside each task file.

4.1 Storage Layout and Naming

SolidTasks organises user-owned data under a stable container (e.g., {pod}/public/solidtasks/ or {pod}/private/solidtasks/). Each task is a separate Turtle file, enabling independent lifecycle and access control per task:

4.2 Task Data Model (RDF/Turtle)

Each task includes title, description, due date, status, and timestamps. We use existing vocabularies where possible (schema:, dct:) with a small ex: extension.

```
@prefix dct:   <http://purl.org/dc/terms/> .
@prefix schema:<http://schema.org/> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix ex:    <https://example.org/vocab/solidtasks#> .

<#it>
  a               schema:Action ;
  schema:name     "Submit COMP3740 draft" ;
  schema:description
                  "Write and upload the first draft of the report" ;
  schema:actionStatus  schema:ActiveActionStatus ;  # or
    schema:CompletedActionStatus
  schema:target   <#target> ;
  schema:startTime "2025-09-21T09:30:00Z"^^xsd:dateTime ;
  schema:endTime   "2025-09-28T23:59:00Z"^^xsd:dateTime ;
  dct:created     "2025-09-21T09:31:45Z"^^xsd:dateTime ;
  dct:modified    "2025-09-24T00:12:10Z"^^xsd:dateTime .

<#target>
  a               schema:Thing ;
  schema:dateDue  "2025-09-28"^^xsd:date ;
  ex:priority     "high" ;
  ex:labels       "uni,writing" .
```

Listing 4.1: Per-task Turtle file ({uuid}.ttl)

Design notes.

- A per-task file keeps resources small and enables independent sharing/revocation and time-limited access.
- Using `schema:Action` with `schema:target` avoids inventing a custom task ontology while staying interoperable.
- `dct:created`/`dct:modified` enable conflict detection and UI freshness indicators.

4.3 Client Architecture (Flutter + Riverpod)

UI Layer `TodoHomePage` renders task lists and a weekly calendar, and opens `AddTaskWidget`/`EditTaskDialog`

State Layer Riverpod providers (`tasks_provider.dart`) expose streams or futures of task view models; they mediate optimistic updates and error states.

Services Layer

- `pod_service.dart` issues LDP requests (e.g., `PUT` to create a new task file).
- `pod_service_acp.dart` and `AcpService` generate and write ACR graphs for per-resource ACP policies (app-scoped writes, delegated read, time-limited access).

4.4 Create/Read/Update/Delete (CRUD) Flow

Following standard application architecture, there is a UI, Services and the Solid POD server endpoint. The basic flow starts with the UI sending in a CRUD function command into the services file. The services file will then invoke access to the Solid POD. The solid pod will in turn send an acr file that contains the policies of the file's permissions. The services receives this then will invoke refresh tasks Figure 4.1 summarises the create flow. Reads and updates follow the same path with `GET`/`PATCH` (SPARQL Update) or full `PUT`.

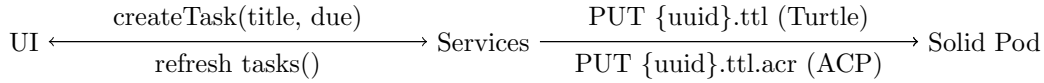


Figure 4.1: Create task sequence: write task file then its ACR; UI refreshes state.

Chapter 5

Use Cases and Application Demonstration

The following use cases demonstrate when to employ different access control mechanisms in Solid applications. We utilize WAC/ACL for simple "user X can read/write" scenarios, while ACP is employed when we require client/app constraints, claim/issuer verification, composable policies, temporal or lifecycle rules, or clearer container inheritance patterns.

5.1 App-scoped Access Control

5.1.1 Scenario and Actors

User Alice stores tasks in her Pod. Anyone she authorizes (e.g., a family member or manager) may read tasks, but only the official SolidTasks application (mobile or web client) should be permitted to create or update task files. This prevents other applications, even those using the same user credentials, from performing write operations.

5.1.2 User Experience

- Official application: create/edit/delete operations function normally
- Third-party application attempting to write the same task: PUT request returns forbidden status
- Shared readers maintain read access permissions

Why ACP?

WAC only gates access by WebID (person identity). ACP enables inclusion of client/azp/DPoP identity checks, allowing permission grants to specific named clients for Write operations.

5.1.3 TTL ACR Example

```
@prefix acp: <http://www.w3.org/ns/solid/acp#>.
@prefix acl: <http://www.w3.org/ns/auth/acl#>.

<> a acp:AccessControlResource;
   acp:accessControl <#owner>, <#appWrite>, <#readers>.

<#owner> a acp:AccessControl; acp:apply <#ownerPolicy>.
<#ownerPolicy> a acp:Policy;
   acp:allow acl:Read, acl:Write, acl:Control;
   acp:anyOf ( <https://pods.example.org/users/alice#me> ).

<#appWrite> a acp:AccessControl; acp:apply <#clientPolicy>.
```

```
<#clientPolicy> a acp:Policy;
   acp:allow acl:Write;
   # Server should match the client id / azp for the official app:
   acp:anyOf ( <https://clients.example.org/solidtasks-web#client> ).

<#readers> a acp:AccessControl; acp:apply <#readerPolicy>.
<#readerPolicy> a acp:Policy;
   acp:allow acl:Read;
   acp:anyOf ( <https://pods.example.org/users/alice#me>
               <https://pods.example.org/users/bob#me> ).
```

5.1.4 Implementation Notes

The Solid server must evaluate azp/client identity in tokens or DPoP cnf. If not supported, implement server-side middleware that denies writes unless azp equals the official client identifier. In the `saveTasks` function (`pod_service.dart`), ACRs are already written; switch to `AcpService.writeAcrForResource(fileUrl, owner, allowWriteWebIds: [], ...)` or utilize allow options to encode client ID as metadata.

5.2 Delegated Sharing Workflows

5.2.1 Scenario and Actors

Owner Alice possesses a set of tasks for a project. She grants a project lead (Manager) the ability to delegate read-only access to short-term contractors for specific tasks or folders. Delegation must be auditable and revocable by Alice.

5.2.2 User Experience

- Manager can add a temporary Read policy for `contractor@example.org` on `task_77.ttl` for 7 days
- Alice can view who created the delegation and revoke it instantly

5.2.3 Why ACP?

ACP allows multiple named policies with metadata. Owners can author policies that permit managers to create limited ACRs, providing a cleaner approach than constantly editing ACLs for each contractor.

5.2.4 TTL ACR Example

The following is an example of an ACR file in turtle form. The purpose of this specific ACR file is to establish a two-tier access control system where:

1. **Owner permissions**: Alice (`https://pods.example.org/users/alice#me`) has full control with Read, Write, and Control permissions, effectively giving her complete authority over the resource including the ability to modify access controls.
2. **Manager permissions**: A designated manager (`https://pods.example.org/users/manager#me`) has Write access only, allowing them to modify the resource content but not change the access control policies themselves.
3. **Governance structure**: The comment indicates that the manager can add subordinate policies within a governance container, with server-side enforcement ensuring proper hierarchical access management.

This ACR demonstrates a common organizational pattern where resource ownership is separated from day-to-day management responsibilities, while maintaining strict control over who can modify access permissions.

```
@prefix dct: <http://purl.org/dc/terms#>.
@prefix acp: <http://www.w3.org/ns/solid/acp#>.
@prefix acl: <http://www.w3.org/ns/auth/acl#>.

<> a acp:AccessControlResource;
   acp:accessControl <#owner>, <#manager>.

<#owner> a acp:AccessControl; acp:apply <#ownerPolicy>.
<#ownerPolicy> a acp:Policy;
   acp:allow acl:Read, acl:Write, acl:Control;
   acp:anyOf ( <https://pods.example.org/users/alice#me> );
   dct:creator <https://pods.example.org/users/alice#me>;
   dct:created "2025-09-01T10:00:00Z"^^xsd:dateTime.

<#manager> a acp:AccessControl; acp:apply <#managerPolicy>.
<#managerPolicy> a acp:Policy;
   acp:allow acl:Write;
   acp:anyOf ( <https://pods.example.org/users/manager#me> ).

# Manager can add subordinate policies in a governance container
# (server-enforced).
```

5.2.5 Implementation

The optimal pattern grants the manager Write access only to the resource's ACR container (or a designated policies subresource), not full Control everywhere. The manager's writes must be validated by the server to ensure creation of only constrained policies (e.g., `acl:Read` for specified WebIDs with limited duration).

5.3 Time-limited Access

5.3.1 Scenario and Actors

Provide temporary access to a dataset for examination duration, or temporary contractor access for 48 hours. After the window expires, access is automatically denied.

5.3.2 User Experience

- A user or client is granted Read access between 2025-09-25T08:00 and 2025-09-25T12:00
- Before or after this window, access is denied without manual cleanup

5.3.3 Why ACP?

ACP enables storage of conditional metadata (expiry) in ACRs. A server can evaluate `validUntil` rather than requiring manual ACL modifications.

5.3.4 TTL-style ACR Example

```
@prefix ex: <http://example.org/ns#>.
@prefix acp: <http://www.w3.org/ns/solid/acp#>.
@prefix acl: <http://www.w3.org/ns/auth/acl#>.

<> a acp:AccessControlResource;
   acp:accessControl <#owner>, <#guestTemp>.

<#owner> a acp:AccessControl; acp:apply <#ownerPolicy>.
<#ownerPolicy> a acp:Policy;
   acp:allow acl:Read, acl:Write, acl:Control;
   acp:anyOf ( <https://pods.example.org/users/alice#me> ).

```

```
<#guestTemp> a acp:AccessControl; acp:apply <#guestPolicy>.
<#guestPolicy> a acp:Policy;
   acp:allow acl:Read;
   acp:anyOf ( <https://pods.external.org/users/guest#me> );
   ex:validUntil "2025-10-10T12:00:00Z"^^xsd:dateTime .
```

5.4 Role-based Team Permissions with Auditability

5.4.1 Scenario and Actors

A large organization utilizes the application. Users possess roles issued by an Identity Provider (role=admin, role=reviewer, role=contributor). Administrators can edit all tasks, reviewers can read all tasks but not edit, and contributors can edit only their own tasks.

5.4.2 User Experience

- Admin login: can edit any task
- Reviewer login: read-only view for all tasks
- Contributor: can create and edit only their own tasks

5.4.3 Why ACP?

ACP can incorporate role matchers and attach `dct:creator` metadata for policy changes, enabling role/claim evaluation and auditable policy history. WAC cannot incorporate token claims cleanly.

5.4.4 TTL ACR Example

```
@prefix acp: <http://www.w3.org/ns/solid/acp#>.
@prefix acl: <http://www.w3.org/ns/auth/acl#>.
@prefix dct: <http://purl.org/dc/terms#>.

<> a acp:AccessControlResource;
   acp:accessControl <#admin>, <#reviewer>, <#author>.

<#admin> a acp:AccessControl; acp:apply <#adminPolicy>.
<#adminPolicy> a acp:Policy;
   acp:allow acl:Read, acl:Write, acl:Control;
   acp:anyOf ( <https://org.example.org/roles/admin> );
   dct:created "2025-09-01T09:00:00Z"^^xsd:dateTime.

<#reviewer> a acp:AccessControl; acp:apply <#reviewerPolicy>.
<#reviewerPolicy> a acp:Policy;
   acp:allow acl:Read;
   acp:anyOf ( <https://org.example.org/roles/reviewer> ).

<#author> a acp:AccessControl; acp:apply <#authorPolicy>.
<#authorPolicy> a acp:Policy;
   acp:allow acl:Read, acl:Write;
   # server must map resource's author metadata (or owner WebID)
   # to allow editing
   acp:anyOf ( <https://pods.example.org/users/<author>#me> ).
```

5.4.5 Implementation Notes

The server must map Identity Provider role claims (verify issuer) to matchers such as `<https://org.example.o`
Add `dct:creator` and `dct:created` to each policy to enable audit trail display in the user interface.

Chapter 6

Implementation and Code Artifacts

This section describes how Access Control Policies are implemented in the SolidTasks application. We detail the code structure, tokened HTTP interactions with Solid PODs, policy-template generation for four use cases, and integration points with the Flutter client and the Riverpod state.

6.1 Code Structure and Responsibilities

The ACP implementation is encapsulated in two Dart classes:

- `AcpService`: a low-level service that (i) generates Turtle ACR bodies for specific use cases, and (ii) performs authenticated `PUT`/`GET` requests to the resource's `.acr` endpoint. It also exposes lightweight helpers for reading ACRs and evaluating time-limited access.
- `AcpPresets`: a thin, developer-friendly façade that binds common application scenarios ("app-scoped access", "manager delegation", "exam access", and "team roles") to the underlying `AcpService` methods with sensible defaults.

This separation keeps the HTTP/authn mechanics and policy–template generation in one place (`AcpService`), while keeping call-sites terse and intention-revealing via `AcpPresets`.

6.2 Authenticated ACR I/O

To write an Access Control Resource (ACR), the client derives the ACR URL by appending `.acr` to the resource URL and performs a `PUT` with a Turtle body. Authentication uses DPoP-bound access tokens obtained per request method and URL:

```
final (:accessToken, :dPopToken) = await getTokensForResource(acrUrl, 'PUT');
final res = await http.put(
  Uri.parse(acrUrl),
  headers: {
    'Content-Type': 'text/turtle',
    'Authorization': 'DPoP $accessToken',
    'DPoP': dPopToken,
  },
  body: acrBody,
);
```

Listing 6.1: Writing an ACR with DPoP-bound tokens

This service treats HTTP responses `200/201/202/205` as successes. In exception, the application surfaces the server status and body to be used for debugging by developers. This explicit method in addition to url binding ensures the DPoP proof matches the request semantics.

6.3 Policy Template Generation

Each use case in SolidTasks is realised through a parameterised Turtle template. These templates declare the Access Control Resource (ACR) and its associated `acp:AccessControl`s, bind them to `acp:Policy` nodes, and encode allowances such as `acl:Read`, `acl:Write`, or `acl:Control` through agent matchers collected under `acp:anyOf`.

In the first use case, *App-Scoped Access*, only the official SolidTasks client is authorised to write. The owner retains full control with read, write, and control permissions, while a dedicated client-specific policy, keyed by the client identifier IRI, grants write privileges to the official client. Optional collaborators can be added as readers, spliced into the `acp:anyOf` matcher alongside the owner.

```
static String _generateAppScopedAcr(String ownerWebId, {
  List<String>? allowReadWebIds, required String allowedClientId,
})
```

Listing 6.2: App-scoped write; optional readers

The second use case, *Delegated Sharing*, models a chain of trust from the owner to a manager and then to contractors. The owner delegates write permission to the manager, who can extend read access to a defined set of contractors. To enhance auditability, policies incorporate provenance metadata such as `dct:creator` and `dct:created`, which record the author and creation timestamp, enabling both UI surfacing and log-based traceability.

```
static String _generateDelegatedSharingAcr(
  String ownerWebId, String managerWebId, {
  List<String>? contractorWebIds, String? createdBy,
})
```

Listing 6.3: Delegation with `dct` metadata

The third use case, *Time-Limited Access*, provides temporary read permissions to guests until a specified UTC timestamp expressed as `ex:validUntil`. Once the timestamp passes, the grant is treated as expired by the UI. A lightweight client-side utility checks validity by parsing the stored timestamp and comparing it to the current system time. If no expiry is defined, the access is considered valid indefinitely.

```
static String _generateTimeLimitedAcr(
  String ownerWebId, { List<String>? tempAccessWebIds, DateTime? validUntil }
)
```

Listing 6.4: Time-boxed read via `ex:validUntil`

Finally, the fourth use case, *Role-Based Access*, applies policies to organisational role IRIs, such as `https://org.example.org/roles/reviewer`, thereby supporting RBAC. The resource author receives both read and write permissions, administrators are allowed read, write, and control, and reviewers are restricted to read-only access. This approach allows access rules to be defined at the level of organisational roles rather than individual users, simplifying policy management while retaining flexibility.

```
static String _generateRoleBasedAcr(
  String ownerWebId, {
    List<String>? adminRoles, List<String>? reviewerRoles,
    List<String>? contributorRoles, String? resourceAuthor,
})
```

Listing 6.5: Role-based policy groups

6.4 Presets and Developer Experience

The `AcpPresets` class exposes developer-friendly wrappers around these use cases. For example, calling `teamRoles` automatically provisions policies for *admin*, *reviewer*, and *contributor* roles without requiring the caller to manually assemble Turtle. This abstraction reduces cognitive overhead and ensures that policy bodies remain consistent across the codebase.

```
1 await AcpPresets.teamRoles(taskUrl, ownerWebId);
```

Listing 6.6: Team roles preset

This design pattern also ensures extensibility: new presets can be layered atop the same low-level ACR primitives, allowing SolidTasks to evolve towards more advanced scenarios such as nested delegation chains or revocable group memberships.

6.4.1 Feasibility and Evaluation

This subsection summarises the experimental approach used to evaluate ACP feasibility in the SolidTasks context, presents the evaluation criteria and metrics, outlines the results interpretation framework, and describes limitations discovered during integration and usability experiments. The goal of this evaluation is to determine whether ACP can be reliably (a) expressed via parameterised Turtle templates, (b) installed to Solid PODs using DPoP-bound requests, (c) surfaced to end users through Flutter UIs, and (d) reasoned about by lightweight client-side utilities (for example, expiry checks).

6.4.2 Methodology

We evaluated feasibility across two dimensions: *functional correctness* and *operational reliability*. For each dimension we executed a mix of unit tests, controlled integration experiments against developer Solid PODs, and lightweight user-facing scenarios. I wanted to test out usabiity and further UX metrics using surveys; however, we were not able to get ANU Ethics council approval, and as such that is out of the scope for this current project.

Testbed Integration experiments used developer Solid PODs (local or community pods) and an instrumented SolidTasks build. All ACR writes used DPoP-bound tokens obtained at runtime via the existing authentication flow. Tests were executed both as automated unit tests (mocking HTTP) and ad-hoc manual flows (real token exchange and network operations).

Scenarios tested

1. **App-Scoped Access:** Verify owner control and that the official client can perform writes while others cannot.
2. **Delegated Sharing:** Verify manager write privileges and contractor read privileges; confirm provenance metadata is present.
3. **Time-Limited Access:** Grant temporary read to guests, validate client-side expiry parsing, and verify expired access is not honoured in the UI.
4. **Role-Based Access:** Grant role IRIs and verify role-based membership simplifies policies when many users share a role.

6.5 ACL vs. ACP Comparisons

6.6 Performance / Complexity Trade-offs

Chapter 7

Discussion

Clearly not every project has empirical components (in particular in mathematics or theoretical topics) – though many do. So in case you were coding anything and conducted an empirical evaluation, this is where you should report the results.

The following sections are, as for the rest of this template, just *suggestions*. They might fit to your work, or they don't. Discuss this with your supervisor(s).

7.1 Benchmark Set

This is where you describe the set of benchmarks that you use for testing your hypothesis empirically. Some ideas on what information you could convey:

- What's the origin of your benchmarks, where are they from?
- What benchmark set did you use *exactly*, i.e., can you explain what they are/mean?
- Why did you choose these benchmarks, and not others? I.e., why are they appropriate for your evaluation? Are they maybe some sort of "standard" and thus also used by others?
- Could you have chosen other benchmarks? If so, which? Why didn't you do so? (Could this maybe form future work?)

In nutshell, just tell everything interesting about the set of benchmarks selected.

7.2 Evaluated Software

This is where you would describe all software or algorithms etc. that you test. For example, if later you have tables or plots with some abbreviations to denote algorithms or specific configurations of your algorithm(s), then this would be the place to define and explain them. So, all these abbreviations/acronyms or software names should be explained/introduced here, and explained in sufficient detail.

Furthermore, in most works you might not only test your own software/contributions, but you might compare it against software from the literature. If possible, then this is certainly good style, as you are usually not the first to tackle a certain problem: Others have attempted this before. So you should compare your performance against performance of the current/previous state of the art. Therefore, this software should be listed here as well. You might potentially have explained these other "competitors" before in the related work section, but there you focused on their scientific approach. Here you would list their software names and configurations, and otherwise reference back to the related work section. Explain why you compare against this software, and maybe mention other related software as well, explaining why you did not compare against it.

7.3 Hardware Setup

This section is supposed to explain all details on how to run the above-mentioned software, so that others could reproduce it, or at least can interpret your results appropriately. This is usually rather short.

On what computer was the experiment run? I.e.,

- What was the Operating System? (Name and version number.)
- What processor (CPU) was used, and how many? Single-core, or Multi-core? (In some disciplines, such as AI planning, only a single CPU is used, even if the processor has multiple cores.) Was GPU power used as well? If so, which?
- How much RAM was made available? (Note that this is typically different from the RAM the hardware has available, as we can reserve a specific amount to processes, which is lower than the total amount physically available.)
- How much runtime did you grant your processes? How did you measure it? Did you take "walltime" or "CPU time"? If you don't know what this means, google it! Explaining it might also be appropriate.
- Was your system a VM, running within another operating system (e.g., Linux within Windows), in a cluster, on a server, a personal laptop, etc.?

In a nutshell, you should simply report anything that will enable your reader to interpret your numbers that you are going to report later.

7.4 Empirical Results

This is the "core" of your section: Here you report all your findings.

A *very important* observations is that you will have to report on *two* things, where some might forget the second, although this is actually the more important item:

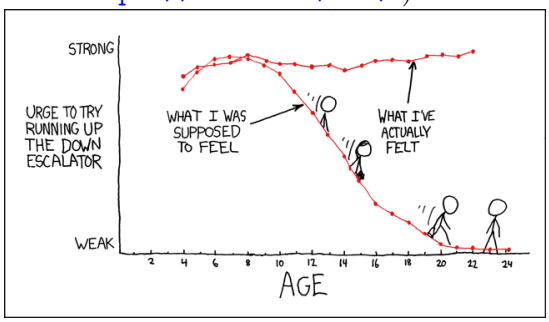
- Plain data. This is the raw data you obtained, reported in tables or plots or the like. E.g. which problems did you solve in the available time? How well did you classify the input correctly? (This is clearly content-specific.) Make sure to report your data in an appropriate way, enabling the reader to easily grasp what the data is supposed to show. It is important to discuss this early on with your supervisor, as he/she likely knows better how to appropriately report on your findings. (Also don't forget to look into important literature working on similar topics.)
- Interpretation. This is what you can infer from your results. Did the approach work well, or did it not? If it worked well, then to which extent? Does it *always* perform best (unlikely!), or is there a subset of benchmarks on which it worked well? If so, why? What's special about this subject making the approach work well there? Do your findings raise further questions and thus directions for future research/investigations? Please do not worry if your results are objectively bad. Certainly bad results can not be published, but you can still obtain very high marks. It is your job to evaluate how well (or how badly!) your approach worked, and it's (likely) not your fault if it did not. So making up ridiculous reasons why the results are great although they are clearly not is anti-scientific and will thus make an incredibly bad impression and penalized mark-wise. Simply objectively and truthfully report the findings – this is science. If the results are bad, can you explain or at least hypothesize why? (This would prove your high level of understanding.) Can you form future work based on your findings?

When reporting your results using graphs and plots, make sure to provide all information necessary to interpret the data, e.g., axis and graph labeling (cf. Figure 7.1).

When you print tables, it's good style to highlight the best results in bold. Also use the BOOKTABS package for nicely formatted tables, as explained in the previous section on LaTeX advice. (That's not a must, but it will simply look much nicer!)

[capbesideposition=left,top,capbesidewidth=.5]figure[]

Figure 7.1: A graph illustrating the importance of axis and graph labeling. (Graphic taken from https://xkcd.com/252/.)



Chapter 8

Concluding Remarks

If you wish, you may also name that section *"Conclusion and Future Work"*, though it might not be a perfect choice to have a section named "A & B" if it has subsections "A" and "B". Also note that you don't necessarily have to use these subsections; that also depends on how much content you have in each. (E.g., having a section header might be odd if it contains just three lines.)

8.1 Conclusion

This section usually summarizes the entire paper including the conclusions drawn, e.g., did the developed techniques work? Maybe add why or why not. Also don't hold back on limitations of your work; it shows that you understood what you have done. And science isn't about claiming how great something is, but about objectively testing hypotheses. Also note that every single scientific paper has such a section, so you can check out many examples, preferably at top-tier venues, e.g., by your supervisor(s).

8.2 Future Work

On top of that, you could discuss future work (and make clear why that is future work, i.e., by which observations did they get justified?).

Note that future work in scientific papers is often not mentioned at all or just in a very few sentences within the conclusion. That should not stop you from putting some effort in. This will (also) show the examiner(s)/supervisor(s) how well you understood the topic or how engaged you are.

8.3 Future Works

Future works can work on the UX side of this project. One reason for low adoption rates of the solid technology aside from high-cost overhead and the inherent technical overhead of implementing Solid technologies in their applications is the unpolished UI/UX implementation. Further research can work on evaluating general or focused audiences' perception of interfaces implemented in this project and make refinements from there.

References

- Brooke, J. (1996). SUS: A quick and dirty usability scale.
- Cranor, L. F., & Garfinkel, S. (2005). *Security and Usability: Designing Secure Systems That People Can Use.*