FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Altynbek Orumbayev

Decentralized Web-based Data Storage for LinkedPipes Applications using Solid

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Jakub Klímek, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2020

I want to dedicate this thesis to my family and their support and motivation. This work wouldn't have been possible without them.

I am also grateful to my friends from the Faculty of Mathematics and Physics at Charles University. Esteban, Memduh, Abishek, Yigit, and Vinit, with whom we had a great two years of fruitful discussions.

Finally, I would like to thank my supervisor RNDr. Jakub Klímek, for his insightful guidance, support, and reviews.

Title: Decentralized Web-based Data Storage for LinkedPipes Applications using Solid

Author: Altynbek Orumbayev

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Klímek, Ph.D., department

Abstract: The following thesis work is focused on developing a decentralized web-based storage solution based on Solid project for the LinkedPipes Applications platform and demonstrate the full benefits of decentralized storage and the Solid technology. The Solid project, started by Sir Tim Berners-Lee, is focused on decentralizing the World Wide Web and decoupling data from applications on the Internet. The results of the practical work satisfy all defined functional and non-functional requirements stated by the LinkedPipes Applications project. The results of the work are integrated into LinkedPipes Applications platform as a set of several software libraries allowing interaction with Solid Servers.

Keywords: solid, linked data, client, server, storage, web

Contents

Introduction

The World Wide Web as we know it started on March 12, 1989, by Sir Tim Berners-Lee. What began as a proposal, eventually ended up being one of the most important technological achievements of a century. Nowadays, the ability to access information online is often an effortless process. People can easily use a search engine to look up articles, connect with anyone in a matter of seconds using social network platforms, consume gigabytes of media [1]. Furthermore, they can upload, store, and share any data online. When it comes to storing data online, an average Internet user will most probably rely on companies providing their storage solutions in the *Cloud*. The majority of popular cloud storage providers like *Google*, *Dropbox* or *Microsoft OneDrive* are centralized. Centralization is in no means a matter of a concern for an average Internet user.

On the contrary, it usually provides a better user experience when the user has all of his relevant data stored in a centralized *data silo*. The term *data silo* often describes a fixed repository of data entirely under control of a single department while being isolated from the rest of the organization. On the other hand, it raises a lot of privacy concerns when it comes to explaining who owns the data stored under such cloud storage. With the growth of large software corporations and an increase in the dominance of their services and technologies where billions of users upload, store and share data under their centralized silos, a lot of examples of disadvantages of centralization were demonstrated for the past several years. For example, an infamous *Facebook–Cambridge Analytica* data scandal in early 2018 [1], demonstrated how millions of *Facebook* profiles were analyzed without any consent and later targeted for political advertising. In a certain sense, the fact that data was under control of a single organization and stored in centralized fashion played an important role in raising privacy concerns and making people more cautious about relying on centralized providers to own their data.

On August 10, 2016, another ambitious project was launched by Sir Tim Berners-Lee called Solid [2]. The goal of the project is to make World Wide Web decentralized, improve data ownership on the Internet, and give the full control over the data back to the users by providing an alternative to dominating storage technologies relying on centralized data silos. Even though the idea might be appealed as too ambitious, over the years, the project has grown from several proof-of-concept proposals to a large community of developers actively contributing and expanding the project every day. The Solid project, by definition, represents multiple things at once. In formal terms, it is a set of specifications, principles, conventions, and tools for building *decentralized social applications* while relying on principles of *Linked Data*. The term Solid by itself stands for *Social Linked Data*, where the term *Linked Data* stands for yet another concept that will be heavily utilized in this thesis project. Linked Data is a method for publishing structured data in a way that preserves the semantics of the data. This semantic description is implemented by the use of *vocabularies* [3], which are usually spec-

[1]https://en.wikipedia.org/wiki/Facebook\OT1\textendashCambridge_Analytica_data_scandal

[2]https://solidproject.org

[3]https://www.w3.org/standards/semanticweb/ontology

ified by the W3C as web standards. However, anyone can create and register their vocabulary, for example, in an open catalog like *Linked Open Vocabularies (LOV)* [4]. Linked data is usually dispersed across many sites on the Internet. Each site usually contains only a part of the entire data available. Thus a machine or a person trying to interpret the data as a whole needs to link this incomplete information together using unique entity identifiers shared across the data stores. Another common term usually associated with Linked Data is *Linked Open Data*, in contrast with Linked Data, it follows the *five star Linked Open Data* model [5]. It represents Linked Data that is publicly accessible to everyone.

In November 2018, a group of five Master students, including the myself, at Faculty of Mathematics and Physics at Charles Univesity in Prague, started a university software project called *LinkedPipes Applications (LPA)*. The project was focused around development of a web app that would simplify the interactions with Linked Data and other concepts of *Semantic Web* [6] for average *lay* users and provide an intuitive and straightforward way to visualize Linked Data expressed in *RDF* format for various needs. The initial goals of that project did not involve any plans to rely on any decentralized storages, such as ones represented by Solid. However, over time the set of functional and non-functional requirements related to dealing with massive amounts of Linked Data expressed in RDF format, ability to share the applications created with the platform while maintaining the control over them, and most importantly storing and sharing the LPA platform data became specific enough to become a perfect fit to use Solid as a primary technology for storing the data.

Goal of the thesis

The main goal of the following thesis is to provide a decentralized web-based storage solution based on Solid project for the LPA platform and demonstrate the full benefits of decentralized storage and the Solid technology. The results of the practical work must satisfy all defined functional and non-functional requirements stated by LPA project. The complete implementation of the solution implementing the requirements must be provided and described in detail as a part of this thesis. It is important to note that the Solid project is still in the early stages of development, and its specifications are being updated regularly. Therefore, aside from fitting the needs of LPA requirements, the implemented toolset must be generic enough for the possibility to be extended for usage in any application using Solid. Further mentions of the practical part of this thesis are going to be called as *LinkedPipes Storage (LPS)*.

Structure

The structure consist of nine main chapters that can be described as follows:

1. *Preliminaries*, Basic introduction to core concepts of *Semantic Web* and Solid that will be appearing throughout the rest of the thesis. The remain-

[4]`https://lov.linkeddata.es/dataset/lov/`

[5]`https://www.w3.org/community/webize/2014/01/17/what-is-5-star-linked-data/`

[6]`https://www.w3.org/standards/semanticweb/`

ing sections of the chapter are dedicated to general description and main concepts of LPA. That project defines the core requirements for this thesis, therefore it is important to make sure that all LPA specific terms and definitions are explained before proceeding to further chapters.

2. *Overview of decentralized web-based storage technologies*, the chapter provides an overview of decentralized software technologies that are alternative or opposed to concepts of Solid. Additionally, the chapter also provides the core ideas on why Solid turned out to be a perfect fit for LPA.

3. *Analysis*, chapter is an overview of all functional and non-functional requirements stated by LPA as they are the main users of the LPS.

4. *Architecture*, chapter continues the *Analysis* chapter by describing the architecture of the solution. Additionally, the first section of the chapter provides a review of technologies, frameworks, and libraries that were under assumption during the design of the architecture and that later used in *Implementation* chapter.

5. *Implementation*, chapter continues the *Architecture* chapter by diving into implementation details and how both specifications and architecture from the previous chapters were implemented into an actual project.

6. *Evaluation*, chapter demonstrates the benefits of choosing Solid as a main technology for decentralized storage in LPA project. The results, recognition, and notable achievements are also described in detail in this chapter.

7. *Testing*, chapter describes everything related to testing in LPS starting from generic conventions and finalizing on specific implementation details.

8. *Documentation*, the chapter describes the various documentation resources written during the implementation of LPA and provides references to access them.

9. *Future work* chapter provides an overview of how the project planned to be improved in future.

The summary of results achieved is provided in *Conclusion* chapter, briefly covering the significant points from the leading nine chapters as well as the main challenges met during the implementation.

1. Preliminaries

The chapter is going to provide the introduction to the core concepts of *Semantic Web*, Solid and LPA platform. As the concepts explained here will be often referred to in further chapters, it is important to get familiar with them for a complete understanding of every consecutive chapter.

1.1 Semantic Web

The Semantic Web represents the next significant iteration in connecting data and information over World Wide Web. It adds an ability for a data to be linked from any source to any other source allowing computers to understand the semantics of it and perform increasingly complex computational operation on them. The major idea that makes Semantic Web technology and other data related technologies such as relational databases, is that it focused on preserving the meaning of the data rather than structuring it in efficient manner.

There are three main technical specifications defining the Semantic Web technologies:

- Resource Description Framework (RDF), is a data modelling language designed for Semantic Web. Every information in Semantic Web is represented in the RDF.
- SPARQL Protocol and RDF Query Language (SPARQL), is a query language for Semantic Web. It is mainly designed for querying data across different data sets represented in RDF [2].
- Web Ontology Language (OWL) is a knowledge representation language for Semantic Web. It allows defining entities and concepts in a way that enables high reusability for many different applications and purposes. Additionally, it also can be represented in a corresponded Multipurpose Internet Mail Extensions (MIME) type, also known as a *OWL* file [3].

In other words, the usage of the technology stack mentioned above is what defines a Semantic Web application and differentiates it from any other technologies related to data.

1.1.1 RDF

The Resource Description Framework is a language for representing resources in World Wide Web [4][5]. Every element of information in RDF is expressed as relation between entities. The entity in RDF is usually referred to as *resource* and identified by Uniform Resource Identifier (URI), in other terms anything identified by Uniform Resource Identifier (URI) is a resource. The relation itself is expressed using *triple* notation. A *triple* notation is a statement written as follows, **`subject predicate object`**.

Consider the following example on Listing 1. A *subject* is a person identified by a URI, a *predicate* identified by a URI describes the kind of relationship and lastly the *object* is a resource similar to *subject* and identified by a URI. In other words, the example demonstrating a sentence *Person A knows Person B* expressed in RDF triple notation. A typical RDF document consists of many triple statements, together they can be represented as a directed *graph*, where relation is presented as a directed edge between two vertices. The term graph will be often referred in Chapter 4 and Chapter 5.

```
http://example.name#Altynbek http://xmlns.com/foaf/0.1/knows
↪  http://example.name#Tim
```

Listing 1: An example of an RDF expressed in triple notation.

Common RDF serialization formats

RDF has many commonly used serialization formats. The formats heavily utilized in this project can be described as follows:

- Turtle (syntax) (TTL), a commonly used textual syntax for RDF. It allows an RDF graph to be written in a compact and easily understandable text form. It supports abbreviations for common usage patterns and datatypes.
- JavaScript Object Notation for Linked Data (syntax) (JSON-LD), a textual syntax for RDF that adds the benefits of JSON. It is a perfect serialization format for development environments due to the minimal effort needed to transform any JSON objects into JSON-LD.

RDF ontologies

Referring back to example in Listing 1, the resources starting with `example.name` URIs are chosen at random only for the sake of demonstrating a triple notation. However, the predicate URI is not a random identifier. On the contrary, this URI belongs to a so-called *RDF Vocabulary*. The term usually stands for a grouping of resource identifiers that represent well-knows entities and resources for various domains. The predicate from example belongs to an RDF Vocabulary called Friend of a Friend (FOAF). This vocabulary, also usually referred to as *ontology*, provides a set of resource identifiers describing persons, their activities and the relations to other persons [6]. The term ontology and vocabulary will be often used in Chapter 4 and Chapter 5, when describing a process of design and implementation of a custom ontology for expressing configurations of applications published with LPA platform.

1.1.2 SPARQL

As mentioned at the beginning of Section 1.1, SPARQL is yet another important part of Semantic Web technology stack. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware [2][7]. SPARQL allows a query to consist of *triple patterns*, *conjunctions*, *disjunctions* and *optional patterns*. Another important term related to SPARQL is a *SPARQL endpoint*. It is a web service at which two or more different devices build a connection with each other on an HTTP network that is capable of receiving and processing SPARQL requests.

An example of Listing 2 demonstrates a sample SPARQL query that retrieves the name and email of the person. The first line on the example is a declaration of prefixes for abbreviating URIs, and it uses the FOAF vocabulary that we already demonstrated earlier on Listing 1. A *SELECT* keyword is a form of a query used to extract raw values from a SPARQL endpoint. The *WHERE* clause provides a generic graph pattern to match against the data graph. The variable name `?person` representing the subject is chosen to improve readability. On the contrary, the variable name can be any arbitrary string. The result of the query is rows with raw values representing name and email of the person. Assuming that a person has multiple email addresses, the response of the query can contain multiple rows representing unique names per each of the email addresses.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
       ?email
WHERE
  {
    ?person  a          foaf:Person .
    ?person  foaf:name  ?name .
    ?person  foaf:mbox  ?email .
  }
```

Listing 2: An example of a SPARQL query to retrieve the name and email values from a person resource.

Other notable query forms of SPARQL syntax can be described as follows:

- ***CONSTRUCT***, used to extract and transform data from SPARQL endpoint and construct valid RDF results.
- ***ASK***, used to provide a boolean answer for a query on a SPARQL endpoint.
- ***DESCRIBE***, used to extract a whole RDF graph from the SPARQL endpoint.

Multiple section in Chapter 4 and Chapter 5 will refer to SPARQL as it is a technology actively used and utilized by Solid.

1.2 Solid

To start with, one of many definitions of SOLID project, for developers, it can be referred to as a set of conventions and building tools for decentralized social applications based on Linked Data concepts. The open-source community actively maintains the project, and at the moment of writing this thesis project, the latest iteration of the specification [1] is equal to version `v0.7.0`. On the other hand, for an average Internet user, the term Solid is intended to be associated with a whole ecosystem of personal private storages where data is stored in a secure and decentralized fashion. The essential concepts of Solid are defined as follows:

- Solid Personal Online Dataspace (POD), is a personal website that serves as an online storage per individual user. The user has full control over his POD and can upload and store any data inside it. The data is either expressed as an RDF, if possible or has a generated RDF metadata attached to a file to preserve the semantics of it. In other words, one of the distinct features states in Solid specification is the requirement to have all data in PODs to be represented in RDF. PODs created and hosted by instances of a Solid servers.

- Solid Server, is an instance of a web server that is compliant to a Solid specification. It is responsible for managing users and creating, updating, or deleting instances of PODs belonging to users.

- WebID is a way to uniquely identify a person, organization, or any other entity using a URI. The URI leads to a *WebID Profile Document*, which is an RDF document expressed in TTL format and contains information describing the agent attached to that WebID. Solid servers manage the creation of such WebID Profile Documents and provide URIs to them that are commonly referred to as *Solid WebID* because it is a WebID with a URI that is hosted inside a POD inside an instance of a Solid server.

- Solid Provider, is essentially any organization or entity that hosts a public instance of a Solid server. Everyone, whether it is an individual person or an organization, can either create and host their instance of a Solid provider by the following documentation on the official website of *Inrupt* [2]. Inrupt is a startup company founded in 2017 by Sir Berners-Lee that is focused on further improvement and enrichment of Solid project ecosystem. They also serve as one of the most popular public Solid providers where everyone can create a WebID and have a POD hosted in Solid servers managed by Inrupt.

- Access-Control List (ACL), a list of permissions attached to an object in the filesystem. Solid project is using a specification called *Web Access Control (WAC)* [8]. It applies the generic concepts of ACL to decentralized cross-domain access control systems and describes the management of so-called *ACL resources* in Solid PODs. The term ACL, in consecutive chapters, is going to be referred to as an RDF resource in Solid conforming to WAC specification, and controlling the control privileges of resources in POD.

The core terms and concepts summarized above will be actively mentioned, and references in the following chapters and will provide a more in-depth technical overview and definitions of Solid specification.

[1]`https://github.com/solid/solid-spec/blob/master/CHANGELOG.md`
[2]`https://inrupt.com`

1.3 LPA

As mentioned in Introduction, the LPA project is a web platform that aims to provide a comfortable and intuitive user experience to generate interactive Linked Data based visualizations by *Domain Experts*. It is also important to recap that the LPA was developed as a software project at Charles University in Prague and is a separate contribution made by an author. The visualization is referred to as an application after being published, and it can then be embedded in an online article or on another web page or perhaps accessed as a standalone page.

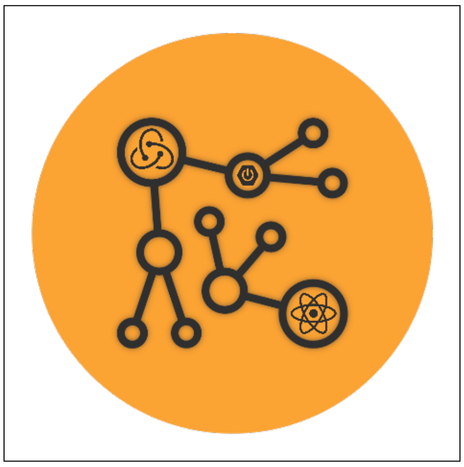


Figure 1.1: Official LPA logo designed by author as a part of separate contribution into the software project.

There are two main user groups targeted by LPA platform:

- *Domain experts*, is a special set of users identified as Semantic Web enthusiasts and people familiar with basic concepts of Linked Data. They have some basic understanding of terms such as RDF, and are able to provide data sources to LPA platform for creating visualizations.
- *Lay users*, represents an set of average Internet users with no particular knowledge of Semantic Web and Linked Data. They can access and interact with applications created with LPA platform by *domain experts*.

1.3.1 Platform description

The following subsection provides a better formal understanding of the platform and expands the definition demonstrating a simple user story on Figure 1.2. As mentioned earlier, the platform serves as a web-app that allows Domain experts to provide Linked Data sources and visualize the data on a set of visualizers supported by the platform. At the moment of writing this work, LPA supports:

- *Maps* visualizer is a generic maps visualization with markers and markers clustering.
- *Chord* visualizer is a diagram representing inter-relationships between data in a matrix.

- *TreeMap* visualizer is a generic visual representation of a data tree.
- *Timeline* visualizer is a generic diagram that allows displaying usage of resources over time.

A simple user story demonstrate on Figure 1.2 describes the platform as follows:

1. *John creates an application.* Assume a Data Journalist user named John that has some Linked Data that he wants to visualize and publish on his website. He uses the LPA and creates his application by following the platform instructions and providing the Linked Data sources to visualize.
2. *LPA extracts and visualizes data.* After inputs from John are provided, the platform is extracting and parsing his data into a format convenient for a visualize that the user selected.
3. *Application is published on website.* After the application is created and published, John uses the generated published application and embeds it into his website.
4. *Bob observes the visualized application.* Bob browses through the website and explores the published visualization without being required to log in or have an account in LPA.

This user story represents the most generic interaction flow with LPA platform from the standpoint of its target users. The consecutive sections will dive into more technical details, components, and functional aspects of this platform.

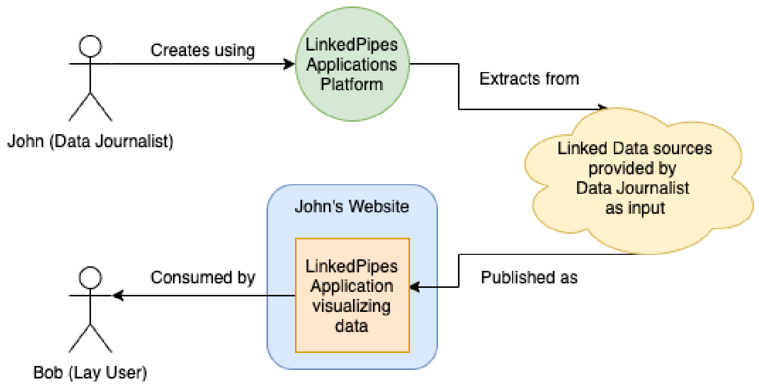

Creates using
LinkedPipes Applications Platform
Extracts from
John (Data Journalist)
Linked Data sources provided by Data Journalist as input
John's Website
LinkedPipes Application visualizing data
Consumed by
Published as
Bob (Lay User)

Figure 1.2: A sample user story describing usage of LPA platform

It is important to note that the creation and publishing aspects of the LPA are one of the requirements for LPS that will be described in detail in Chapter 3.

1.3.2 Components overview

At the lower level, the LPA is a bundle of various complex services and database solutions communicating actively communicating between each other.

Generally, we can categorize it into three main parts:

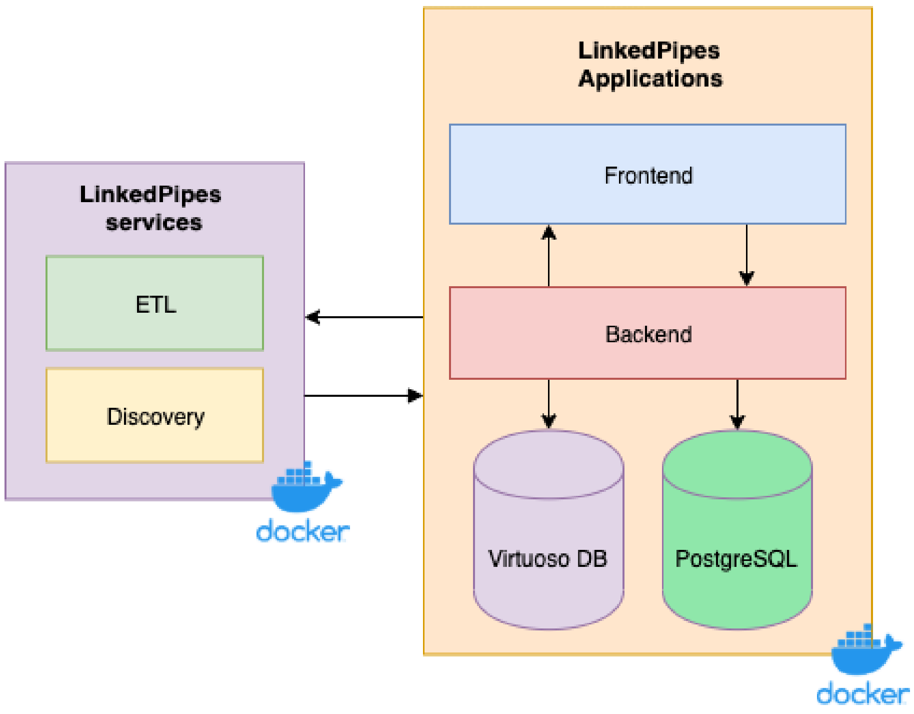


Figure 1.3: High-level overview of LPA platform

- **LinkedPipes Applications** - the main platform from LinkedPipes bundle that defines the functional and non-functional requirements for for LPS project. Combines multiple database solutions for Linked Data, conventional SQL for storing user related records and implementation of a backend and frontend for creating applications.
- **LinkedPipes Services** - a set of external services provided from LinkedPipes bundle that LPA heavily utilizes. Provides a toolset for identifying how linked data can be discovered, extracted, transformed and loaded into an RDF file for further processing.

Frontend

As mentioned at the beginning of the chapter, the frontend provides a way for the user to interact with the LPA. The Redux [3] and React [4] are the leading technologies to be used during implementation. The main terms related to LPA frontend that are commonly reffered to in Chapter 4 and Chapter 5 are described as follows:

- **React Component** - a JavaScript class or function that optionally accepts inputs, i.e., properties(props), and returns a React element that describes how a section of the UI (User Interface) should appear.
- **Redux** - represents a container that stores various states of the web application per individual webpage.

[3]`https://redux.js.org`
[4]`https://reactjs.org`

State - refers to the single state value that is managed by the store and returned by getState(). It represents the entire state of a Redux application, which is often a deeply nested object.

Reducer - specifies how the application's state changes in response to actions sent to the store.

Actions - are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using store.dispatch().

Selector - is simply any function that accepts the Redux store state (or part of the state) as an argument, and returns data that is based on that state.

Additionally, the LPA frontend is the main component that is going to be improved by LPS. This is due to specifics of a Solid toolset where most of the open-source libraries are implemented in JavaScript programming language and are intended for usage in web based projects.

Backend

The function of the backend component is to provide a RESTful Application Programming Interface (API) that is used by the frontend component to execute user-requested actions and can also be used by other developers to create their user applications. The backend then implements the communication protocols with external services from LinkedPipes Services bundle.

PostgreSQL

The LPA platform is storing all information related to users of the platform in an instance of a PostgreSQL database [5].

PostgreSQL, often Postgres, is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards compliance. It can handle workloads ranging from small single-machine applications to large Internet-facing applications with many concurrent users.

The entities stored in the database are described as follows:

- User accounts.
- Running instances of LinkedPipes discovery processes. The LinkedPipes Discovery component is described in detail Subsection 1.3.3.
- Running LinkedPipes ETL pipelines. The LinkedPipes ETL component is described in detail in Subsection 1.3.3.
- Custom templates of data sources generated by users.

[5] `https://www.postgresql.org`

Virtuoso DB

Aside from PostgreSQL, another important database technology is OpenLink Virtuoso[6] which is used for storing all Linked Data sources processed by LinkedPipes ETL pipelines, and that is later used by LPA visualizers. The important note to mention is that in contrast with PostgreSQL that is only used for all user-related information that needs to be stored, Virtuoso is only used for storing Linked Data.

In general, Virtuoso is a middleware and database engine hybrid that combines the functionality of a traditional Relational database management system (RDBMS), Object-relational database management system (ORDBMS), virtual database, RDF, XML, free-text, web application server and file server functionality in a single system. Rather than having dedicated servers for each of the functionality mentioned above, Virtuoso serves as a universal server instance.

1.3.3 LinkedPipes Services

The LinkedPipes Services bundle demonstrated on Figure 1.3 consists of two open-source software solutions developed under the Faculty of Mathematics and Physics at Charles University in Prague. They serve as the providers of core Linked Data processing functionality for LPA platform. Since a significant technical knowledge in Semantic Web is required to use the tools independently, the LPA platform provides a layer on top of them to simplify interactions with those components and make it available for any generic *lay* user.

ETL

LinkedPipes ETL is an RDF-based, lightweight Extract, Transform and Load (ETL) tool for Linked Data [9]. It is not only a stand-alone application, but it also exposes an API through which third-parties can execute the ETL process. In formal terms, a service performs the core computational work by querying the graphs of data sources provided by a user and attempting to transform that data into a format that can be later converted into an input for visualizations.

Discovery

LinkedPipes Discovery is a service used by LPA platform to discover whether provided Linked Data sources can be processed and visualized by the platform [10]. After a successful request sent to the Discovery service, it executes a session called *Discovery session*. Upon successful completion of a Discovery session, the service generates specific files in JSON-LD format. Those files are called *pipelines*. A pipeline describes how LinkedPipes ETL needs to extract the whole Linked Data set and store the data into a Virtuoso database, which is a component used by the LinkedPipes Applications platform described earlier.

[6] `https://virtuoso.openlinksw.com`

2. Related work

This chapter provides an overview of currently available alternatives to Solid project, as well as research projects that share similarities with the core concepts of decentralized social platforms. The first set of sections provides a general description of alternative technologies, the last Section 2.7 provides a comparison between the alternative software solutions and Solid.

2.1 Diaspora

Out of all currently available solutions, the conceptually closest software platform to Solid is Diaspora [11][12]. In general, it is a decentralized social network platform that enables users to choose the server where their data is hosted and even run their own data hosting server. In that sense, it is similar to Solid. However, the main focus in Diaspora is to act as a social network, where social data is shared between users using specific APIs, and not running diverse applications on stored data. Unfortunately, it does not offer a well-defined way to use the same data with different applications. Note that Diaspora uses the term pod to refer to a data hosting server. A Diaspora POD is what Solid would refer to as a Solid server.

2.2 WebBox

WebBox is a decentralized social network platform that decouples data from applications [13]. As Solid, it stores user's data as Linked Data in a decentralized way. Also similar to Solid, system rely on WebID for decentralized identity, authentication, and access control. In WebBox, each data storage service exposes a SPARQL endpoint, and applications manipulate the data via SPARQL queries and updates, or via HTTP GET requests. In contrast, Solid offers the full power of LDP for simple data interactions (e.g., hierarchical data organization, fine-grained manipulation) and additionally allows the use of link-following SPARQL for complex data retrieval. It also works as a generic storage platform upon which a significantly larger number of applications can be built. As a last remark, WebBox was mainly mentioned due to the similarities in functionality with Solid project, however, the platform was never released out of the bounds of a research prototype.

2.3 OwnCloud

OwnCloud [1] is a self-hosted open source file sync and share server. In regards to previously defined requirements, this solution is somewhat too generic to the goals of the project, but due to certain features such as decentralization of the storage using manually setup instances, and various scalability features, building a platform based on the provided API is possible. Similar to Dropbox, Google

[1]`https://owncloud.org`

Drive, Box, and others, ownCloud lets you access your files, calendar, contacts, and other data. You can synchronize everything between your devices and share files with others. In contrast with Solid, the solution is still a generic cloud storage. It does not support Linked Data out of the box and does not provide a trivial way to decouple data from applications using the data.

2.4 Mastodon

Mastodon is an online, self-hosted social media, and social networking service [14]. It allows anyone to host their own server node in the network, and its various separately operated user bases are federated across many different sites (called *instances*). These instances are connected as a federated social network, allowing users from different instances to interact with each other seamlessly. Mastodon is a part of the wider Fediverse, allowing its users to also interact with users on other platforms that support the same protocol, such as PeerTube, Misskey, Friendica and Pleroma.

Mastodon has microblogging features similar to Twitter, or Weibo, although it is distinct from them, and unlike a typical software as a service platform, it is not centrally hosted. Each user is a member of a specific, independently operated instance. Users post short messages called ”toots” for others to see, and can adjust each of their post’s privacy settings. The specific privacy options may vary between sites, but typically include direct messaging, followers only, public but not listed in the public feed, and public and posted to the public feed. The Mastodon mascot is a brown or grey woolly mammoth, sometimes depicted using a tablet or smartphone.

Because there is no central server for Mastodon, each instance has its own code of conduct, terms of service, and moderation policies. This differs from traditional social networks by allowing users to choose an instance which has policies they agree with, or to leave an instance that has policies they disagree with, without losing access to Mastodon’s social network.

2.5 Hubzilla

Hubzilla [2] is a modular webserver based operating system which includes technologies for publishing, social media, file sharing, photo sharing, chat and more (including the ability to develop custom modules). These services are accessed and connected across server and administrative boundaries through the communication protocol Zot which provides a high level of privacy and security customization and a nomadic identity for the users. A webserver running Hubzilla is called a ”hub”.

2.6 Centralized cloud storage solutions

Modern commercial and enterprise cloud storage solutions provide a great set of features as platforms for any generic use cases when reliable file storage is needed.

[2] `https://fediverse.party/en/hubzilla/`

For instance, Google Cloud Storage [3] and Amazon AWS Storage [4]. However, in contrast with Solid, this approach is the most distant from the main concepts of LinkedPipes Applications project's requirements. As a main disadvantage of the approach is the fact that even though the developer is offered a great set of flexibility within the platform, the data storing aspects are not fully decentralized. It also means that the end-user is uploading his data into a centralized data silo where commercial terms of services policies regulate ownership of his data. Another issue is platform dependency meaning that migrating, sharing, and interchanging the data between platforms is not trivial since all interactions with data within storage are defined by a set of APIs and SDKs specific to the selected platform.

2.6.1 Pitfals of centralized cloud storages

In the past, having control over content or information people store and access online was a common case for many technologies on the Web. However, during the past 20 years, that situation changed significantly. Various tech giants and media platforms like Facebook and Google gain control over the personal data of millions of users using their services. The data that is under the control of the fixed authority separated from the rest of the organization or community is often referred to as data silos. From the standpoint of the company, having centralized control over the data of all users of their platforms brings many benefits. It simplifies the development of services within the organization, allows to perform various analytics processes and understand the users better. However, from the users perspective, it significantly reduces the control of their data stored within centralized platforms.

2.7 Comparison of technologies

A significant difference between Solid and services mentioned below is that Solid sets a standard way of operating through a data model (RDF) and does not dictate how instances should behave as part of a "federation". In addition to that, Solid is not just a software technology, and it represents specifications built on top of open web standards, an extensive set of guidelines for developers, a community of active researchers and contributors. When it comes to direct comparison, Solid could be defined as a lower-level abstraction than federated services like Diaspora and Mastodon in the sense that Solid can serve as an infrastructure for building those platforms. For instance, Diaspora and Mastodon could be integrated into Solid by allowing their users to sign in using their WebID, and then allow users to store the data produced by them on the services on their POD.

The Table 2.1 provides a detailed comparison between Solid and technologies alternative to it. The columns indicate questions representing main features required from a technology to be able to use the benefits of Semantic Web, Linked Data, and RDF at full scale while preserving security and decentralized storing aspects. Both Diaspora and WebBox come very close to Solid in terms of

[3]https://cloud.google.com/storage/
[4]https://aws.amazon.com/products/storage/

Technology	**Provides Personal Data Store?**	**Is stored data easily reusable?**	**Has decentralized infrastructure?**	**Has Linked Data support?**	**Is Open Source?**	**Is used in production?**
Solid	Yes	Yes	Yes	Yes	Yes	No
Diaspora	Yes	No	Yes	Yes	Yes	Yes
WebBox	Yes	Yes	Yes	Yes	No	No
OwnCloud	No	No	No	No	Yes	Yes
Mastodon	No	Yes	Yes	No	Yes	Yes
HubZilla	Yes	Yes	Yes	No	Yes	Yes
Popular centralized cloud storages	No	No	No	No	No	Yes

Table 2.1: A comparison table between Solid and alternative technologies with similar concepts.

provided functionality. However, as mentioned in their descriptions, Diaspora does not focus actively on storage and data decoupling aspects, and WebBox is a proof of concept project that is not available in production or has an open-source community. Similar to Diaspora, Mastodon and Hubzilla position themselves as platforms providing various social networking features to users. And finally, solutions like OwnCloud or popular centralized cloud storage solutions from Google, Microsoft, or Amazon provide convenient developer APIs for using their technologies. However, the technology itself does not preserve the ideas of decentralization and privacy-oriented data management. In contrast, as mentioned before, it provides an opposite functionality where data gets aggregated inside centralized silos under control of the organization providing the storage. To sum up, this section provided a brief comparison overview of alternatives to Solid. The following Chapter 3 will give detailed reasoning for choosing Solid as a core technology for providing storage capabilities for LPA and what makes it a better fit than its alternatives.

3. Analysis

The following chapter will start by providing a detailed set of functional and non-functional requirements from LPA in regards to LPS. Afterward, an analysis of the currently available development tools within the Solid ecosystem will be demonstrated. Additionally, as a continuation of comparison in Section 2.7, a reasoning for choosing Solid as a core technology for implementing LPA requirements will be explained. The majority of the functionality of LPA and LPS were implemented by the author while working in parallel as a part of LPA and as an individual contribution to the thesis. Therefore, to simplify the understanding of work on both projects and the difference between both contributions, the whole chapter is structured under the assumption that LPA is a stakeholder with a working solution and LPS is a technology yet to be implemented and integrated into LPA. The consecutive chapter on architecture in Chapter 4 and the implementation in Chapter 5, will guide the reader through established software development cycles where the result will be provided as a summary of implementation details in regards to requirements defined in this chapter.

Aside from definitions of specific requirements, it is important to note that there are three main *actors* involved in statements of requirements. An actor is an individual entity interacting with the software components in requirements. Specific types are defined as follows:

- *Data journalists*, this category of actors are one of main target users of LPA platform. They are defined as people familiar with, at least, basics of Semantic Web. They can provide Linked Data sources to LPA for further visualization and publishing of apps. They are also assumed to have an actual account in the LPA platform.

- *Lay users*, this is a second target category of users of LPA platform. They do not have any prior experience with Semantic Web and in most cases, browse the published visualizers created by Data journalists. They are not obliged to have an actual account in the LPA platform.

- *LinkedPipes Applications (LPA) Developers*, a developers implementing on LPA platform codebase and main users of LPS features, APIs and functionality to be developed. The majority of details in Chapter 5 are focused on developing a developer-friendly software for this category of users.

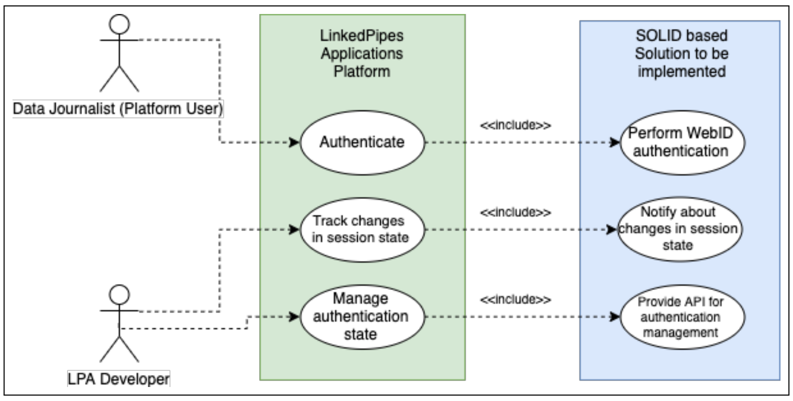
3.1 Functional requirements

In general, every functional requirement is defined as a description of services or features that software must offer. In this case LPA defines a set of features expected to be handled properly by the LPS project. The requirements are provided as a set of UML use-case [1] diagrams as well as formal textual descriptions.

[1] `https://www.uml-diagrams.org/use-case-diagrams.html`

3.1.1 User authentication

The user of the platform should be able to register an account in the application, log in, and log out. The requirement might seem unfitting to the purposes of LPS. However, if Solid is considered to be used, it provides the fully functional authentication mechanisms based on WebID that can be used for generic platform authorization.



Data Journalist (Platform User)
LPA Developer
LinkedPipes Applications Platform
Authenticate
Track changes in session state
Manage authentication state
SOLID based Solution to be implemented
Perform WebID authentication
Notify about changes in session state
Provide API for authentication management
<<include>>
<<include>>
<<include>>

Figure 3.1: A UML use-case diagram describing user authentication requirement.

The UML diagram on Figure 3.1 is described as follows:

- *Authenticate*, a platform must provide a way for Data Journalists (Platform Users) to authenticate via LPA platform. This assumes that LPA platform has an ability to communicate with the storage solution either integrated into LPA codebase or presented as entirely separate service and *perform WebID authentication.*

- *Track changes in session state*, a platform must provide an API or a developer library for tracking any changes in the state of the authenticated platform user session. This assumes that the storage solution can inform the LPA platform about such changes.

- *Manage authentication state*, the platform must provide an API or a developer library for managing the authentication state of the users. This assumes that the core functionality is provided by the storage solution that exposes an API or a developer library toolset for developers to perform such interactions.

3.1.2 Create, Store and Publish Application

Creation, storing, and publishing of the application are the core features provided by the LPA itself. However, the storage aspects are involved a lot when it comes to storing the created app. Consequently, publishing also consists of interacting with storage since an end-user expects the storage solution to have an identifier for his stored application.

Creating and storing application

User should be able to create an application within the LPA platform and store it in his personal space in an authenticated storage. The storage solution can either implement a solution that stores entire application including the visualizer or generates a metadata configuration file that allows to re-create the entire application within LPA platform from scratch.

Publishing an application

Another important requirement is an ability to publish the application and make it publically available for sharing the visualization to anyone via the permanent link. Furthermore, when a third party accesses this *permalink*, the browser should open the LPA website with the respective application opened, as configured by the publisher. The tool will also need to provide the ability to embed the published view into a data journalist's web page, for example, using an HTML `iframe` [2].

A diagram overview

The Chapter 5 describes the specific of creation, storing and publishing application requirements from perspectives of different actors involved.

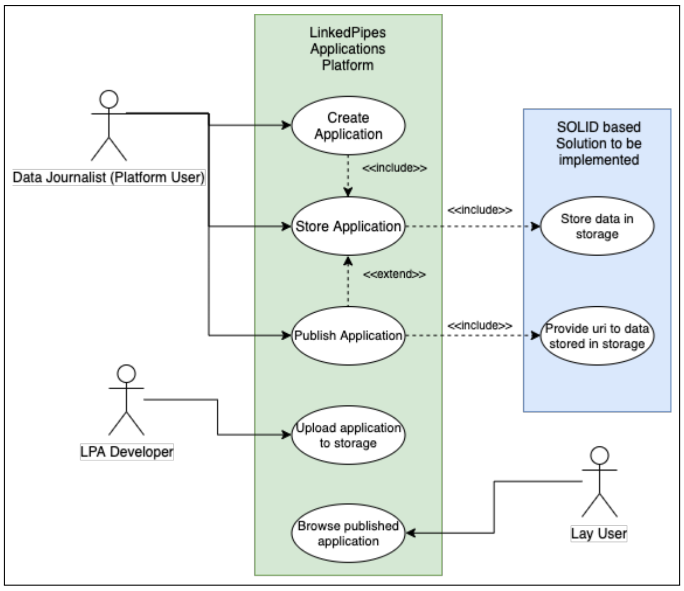

LinkedPipes Applications Platform
Data Journalist (Platform User)
Create Application
<<include>>
Store Application
<<extend>>
Publish Application
LPA Developer
Upload application to storage
Browse published application
SOLID based Solution to be implemented
<<include>>
Store data in storage
<<include>>
Provide uri to data stored in storage
Lay User

Figure 3.2: A UML use-case diagram describing creation, storing and publishing application requirements.

The diagram is described as follows:

- *Create application*, the platform provides an ability for Data Journalists to create an application. The process of creation of an application within LPA also assumes that it is consequently *Stored* and *Published* via a set of interactions with storage. The publishing process is extended from storing an application because to publish an application, a configuration needs to be saved, and a URI for a stored resource in a Solid POD is extracted.

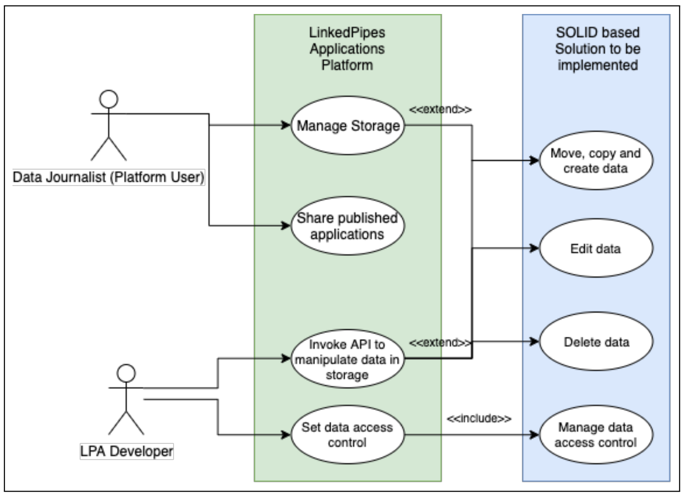
[2] `https://html.com/tags/iframe`

- *Upload application to storage*, a platform provides an API or a library for LPA Developers to perform uploading of an application into the storage solution.

- *Browse published application*, a platform provides a way for any Lay User to browse the published application created by Data Journalists. This is included as a part of the requirement since publishing is directly interacting with storage, and implementation of the requirement applies to both LPS and LPA platform.

3.1.3 Managing storage and sharing published applications

The last set of requirements is related to storage management and the sharing of published applications. The storage management requires a functionality that allows platform users to *move*, *copy* and *create* data in their authenticated spaces in storage. The sharing requirement describes the functionality to allow users to share the applications they have published and control the access control settings to them. Since published applications require to be publically accessible to anyone, the platform needs to have a functionality to persist the configurations of the applications in a secure way. Moreover, the configurations for most of the published applications are represented as Linked Data. Therefore the persistent storage tooling needs to be optimized for files in RDF format.



LinkedPipes Applications Platform
SOLID based Solution to be implemented
Data Journalist (Platform User)
Manage Storage
<<extend>>
Move, copy and create data
Share published applications
Edit data
Invoke API to manipulate data in storage
<<extend>>
Delete data
LPA Developer
Set data access control
<<include>>
Manage data access control

Figure 3.3: A UML use-case diagram describing requirements on storage management and sharing of published application.

As demonstrated on Figure 3.3, the main elements are described as follows:

- *Manage storage*, the functionality of the platform to allow Data Journalists (Platform Users) to interact and manipulate data in their storage. In the case of LPA, it is limited to visualizer application-related data only.

- *Share published applications*, an ability for Data Journalists to share the application with other users of the LPA platform and set access control to the published applications.

- *Invoke API to manipulate data in storage* assumes that LPA provides tooling for their developers to interact with storage solution and invoke its APIs or libraries for manipulating data.
- *Set data access control*, assumes that LPA provides tooling for their developers to interact with storage solution and invoke its APIs or libraries for editing access control privileges to data.

3.2 Non-functional requirements

In general, every non-functional requirement is described as the quality attributes of a software system, which contrasts with functional requirements that are specific to the exact behavior of the system. Within the scope of LPA platform, the stated non-functional are rather an informal set of terms mainly related to *reusability*, *testability* quality attributes.

3.2.1 Compatibility with latest tools

This general non-functional requirement states that an implemented software solution should be compatible with the latest tools of its technological ecosystem. For instance, if the designed software is a library using third-party packages, it needs to be generic enough to be compatible or have mechanisms to be resilient to any significant changes in third-party packages that it relies on.

3.2.2 Clean APIs and libraries

The solution needs to be implemented with code readability and reusability in mind. The primary users of the solution are developers of LPA platform. Therefore it is essential to have an implementation based on a design that minds the established best practices and patterns of software engineering.

3.2.3 Continuous Integration and Delivery

The implemented solution needs to maintain an optimized and fully automated Continuous Integration and Delivery pipelines. This ensures better possibilities in gaining contributors and planning future work and improvements on the solution as well as an infrastructure to debug and fix errors more effectively.

3.2.4 Easy integration with LPA

The final solution needs to be able to integrate with the LPA easily. Another important aspect is the solution to be integrated should not be entirely tied to the specified LPA platform and should have the potential to be reused in other Linked Data based applications.

3.2.5 Decentralized storage

The Data Journalists, one of the primary target users of LPA platform, should be able to choose any storage that is compliant with the technology stack of the solution. In other words, the platform should be able to support storing data in any personal space of such storage instances that are created, hosted, or owned by users directly or by the third-party providers.

3.3 Solid development toolset

Now, as the main requirements of the LPA platform are defined, let us dive into the analysis of the Solid technology and tooling that it provides to understand better how it can be used to cover all stated requirements.

At the core Solid is a set of open specifications. At the current state of the project implementation, the community is most concerned about the persistence and representation of resources. However, such aspects as identity, authentication, and authorization are also vital parts of Solid.

A set of these standards are undergoing active implementation stage. Main contributors consisting of the core Solid development team, as well as the open-source community, develop the standards into various servers and tools. For instance, a `node-solid-server` library that implements a Solid server in Node and `rdflib.js` that allows you to manipulate RDF programmatically.

In addition to this, there is work done on supporting tools, such as Solid React SDK, that is created to allow developers to more easily get into developing apps with Solid. As part of this is the style guide that can be reused by others, not wanting to use the React parts.

3.3.1 The Solid servers

Solid itself represents a tech stack of complementary standards and data format vocabularies that are currently only available in centralized social media services. It also represents a Specification Document, serving as the main guideline for developers building their own apps and services. However, aside from that Solid also refers to a set of servers that implement its specification.

gold

`gold` is a reference Linked Data Platform server for the Solid platform. The implementation is written in Go, based on initial work done by William Waites [15].

node-solid-server

Following solution is an implementation of a server based on Solid specifications in Node.js [16]. One of the main advantages is that it could be launched as a Solid server on top of the local file-system. Interaction with server can be performed as follows:

- Command line tool.

- Via `node-solid-server` library.

The provided server implementation was used as a main Solid server for the project due to compliance with main requirements for choosing the server. Those can be described as follows:

- **Easy integration with current LPA project**, specifically the frontend web project. This allows usage of provided 'node-solid-server.js' library.
- **Active maintenance by open-source community**. Aside from that, this server implementation is considered to be a default option suggested by creators of Solid.
- **Support of *WebID-TLS* and *WebID-OIDC***. This implies majority of WebID communications protocols, crucial to the user authentication and security aspects of the storage within LPA. The WebID-TLS stands for a WebID authentication method over Transport Layer Security (TLS), providing an efficient and user friendly authentication on the Web [17]. The WebID-OIDC is an authentication delegation protocol suitable for WebID-based decentralized systems such as Solid [18].

3.3.2 The Solid React development stack

The main frontend library used in LinkedPipes Applications is React. In order to incorporate easier integration of aspects of Solid into the project, an analysis of React related Solid frameworks and libraries was performed. Even though, Solid community is yet to grow become stable and mature, it already provides a convenient set of libraries for React that were used as a main tools during implementation of the thesis project.

solid/react

The main purpose of this library is to provide the following functionality:

- Provide React developers with components to develop Solid apps.
- Enable React developers to build their own components for Solid.

@inrupt/solid-react-components

This libray is an official SDK provided by Inrupt for developing React Web applications for Solid. The package include various dependencies allowing:

- Provide React developers with a set of easily customizable components interacting with Solid specifications.
- Provide a standardized visual design conventions based on Atomic Style.
- A set of cli commands for generating a template Solid projects.

solid-auth-client

The main purpose of this low level library is to provide ability to easily perform Authentication operations while interacting with Solid pods and servers.

At its core it is a browser library that allows your apps to securely log in to Solid data pods and read and write data from them.

solid-file-client

This library provides a simple interface for logging in and out of a Solid data store, maintaining a persistent session, and for managing files and folders. It may be used either directly in the browser or with node/require. The library is based on solid-auth-client and solid-cli, providing an error-handling interface and some convenience shortcuts on top of their methods and providing a common interface to the two modules.

rdflib

JavaScript RDF library for browsers and Node.js. One of the main contributors is Sir Tim Berners-Lee himself. The library is going to be used and described in detail in Chapter 5.

@solid/query-ldflex

The following library adds support of the LDflex language to Solid by:

- Providing a JSON-LD context for Solid.
- Binding a query engine (Comunica).
- Exposing useful data paths.
- LDflex expressions occur for example on Solid React components, where they make it easy for developers to specify what data they want to show. They can also be used as an expression language in any other Solid project or framework.

3.4 Why Solid?

Even though it is presented as a flawless dominating technology in Table 2.1, it is, in fact, far from being perfect. Some of the disadvantages include:

- *Still under active research and development*, the project itself started back in 2016. Still, the real active traction and improvements had happened only within the past two years after community grew bigger, and the project gained popularity in the open-source community. It also means that there are still major changes in specifications introduces in every new iteration.

- *Lack of actively maintained community libraries*, even though Solid organization on GitHub [3] actively updates and maintains its libraries for interacting with Solid. There is a large number of libraries developed by the community with tools that, in some cases, are more useful and popular. However, due to the Solid project still being actively developed, a lot of libraries outside of the scope of the organization become obsolete and outdated quickly.
- *Still a lot of room for improving infrastructure for developers.* The project does not provide intuitive and user-friendly documentation. An average developer is required to have basic knowledge in Semantic Web, Linked Data, and working with RDF and SPARQL. However, the improvements were made in 2019, and eventually, the onboarding process into the technology will become easier for any developer.

Despite the disadvantaged mentioned above, the main reason for choosing Solid as a core for providing storage capabilities to LPA was the fact that it is a very flexible and low-level technology. Firstly, as mentioned earlier, Solid represents multiple things at once. It is both a toolset with libraries for developers, a set of specifications, and an ambitious idea to a better World Wide Web where people own their data, and all information is linked together, preserving the semantic meaning. Of course, many attempts on decentralization were made before, but Solid differs from the others in the way that it does not target individuals in specific fields or domains of the Internet, it attempts to change the Internet itself, making it a better online space for everyone. And lastly, it demonstrated itself as a perfect fit for any application that requires any social aspects and involves dealing with any form of Linked Data, where LPA platform and its requirements are compliant to all of the statements mentioned earlier.

[3]`https://github.com`

4. Architecture

The following chapter provides an overview of the architecture of LPS. Explains the main components of the Storage as well as how it is being integrated into the LinkedPipes Applications platform.

4.1 High-Level Overview

While the majority of components were described in detail in Chapter 1, the overview architecture provided in this section shows more specific details on how all the external and internal components of the system are interacting. One of the essential details represented on the Figure 4.1 is the separation between codebases of LPA and LPS. The LPA Frontend imports the LPS as an npm [1] package and performs all interactions with Solid using the provided functionality of the package. In some sense, LPS is being treated as an additional rudimentary backend and database layer on top of existing internal components inside LPA. This is due to several functional requirements that LPS package implements, such as *user authentication*, *operations to manipulate resources inside storage* and etc.

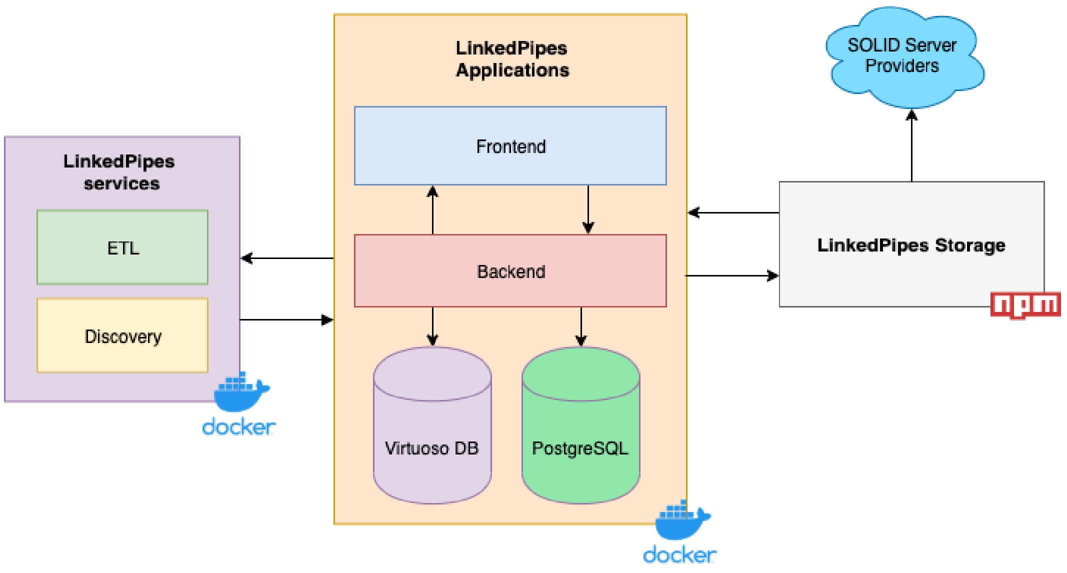


Figure 4.1: High level overview of LPA and LPS interactions

As additionally demonstrated, the Figure 4.1 also has Docker icons displayed under some modules. The icons indicate where the production-ready service is hosted. For instance, in the case of the LPS, the end goal is the npm registry, while LinkedPipes Services and LPA are all hosted in the Docker registry where each internal component is a Docker container. The generic interaction flow usually involves direct communication between the LPA frontend and LPS package. The internal frontend component has various React components implemented using the LPS package that provides navigation and interaction with Solid Pods. While the package is designed under the assumption that the LPA is the only

[1] https://www.npmjs.com

user of the package, some abstractions are generic enough and have the potential to be used outside of the scope of the main functional requirements. The implementation chapter will also cover a generic use-case on how quick start quickly with application development based on Solid.

The evolution of solid specifications

At the moment of writing this chapter, the official Solid specification reached version `0.7.0` [2]. Introducing many changes and improvements, it also adds an extra layer of complexity with every release, changing some of the conventions, or updating some fundamental paradigms. As a result, when LPA was initially implemented, it was based on an older specification and older versions of `node-solid-server`, that had a simpler and more extensive ability to manipulate ACL files. The work within this project, however, was focused around putting efforts into making it generic enough so that people could use it as a guideline for any started Solid apps, while the implementation will fit all LPA requirements.

The main note to mention for the further section in this chapter is that the architecture design was derived from relying on the official specification of Solid while also referring to the provided functionality of `NSS` that does not always strictly follow the guidelines in the specification.

4.2 Storage

The initial architecture and implementation draft of LPS were different from what is presented in this chapter. Solid related logic was firstly a part of the LPA codebase. Therefore, significantly complicating unit testing and making it hard to define the scopes of the LPA and LPS projects implementation. Later on, a decision was made to separate the logic of the LPS and move it in the separate codebase. The majority of abstractions that were initially designed to be inside the LPA codebase, consisted of various wrappers and Create Read Update Delete (CRUD) functions to interact with Solid servers. Their design was refined and aggregated into specific abstractions, each responsible for covering the functional requirements from LPA project.

Let us start by describing the main abstractions providing the functionality of LPS.

4.2.1 Authentication Manager

The Authentication Manager is responsible for wrapping Solid WebID based authentication logic into a simple and developer-friendly abstraction At the moment of writing this, the official Solid specification states to support the following protocols:

- WebID-TLS - is one of the primary authentication protocols that rely on WebIDs instead of usernames. The passwords are replaced with certain cryptographic certificates as bearer tokens and are stored within the user's browser.

[2] `https://github.com/solid/solid-spec/blob/master/CHANGELOG.md`

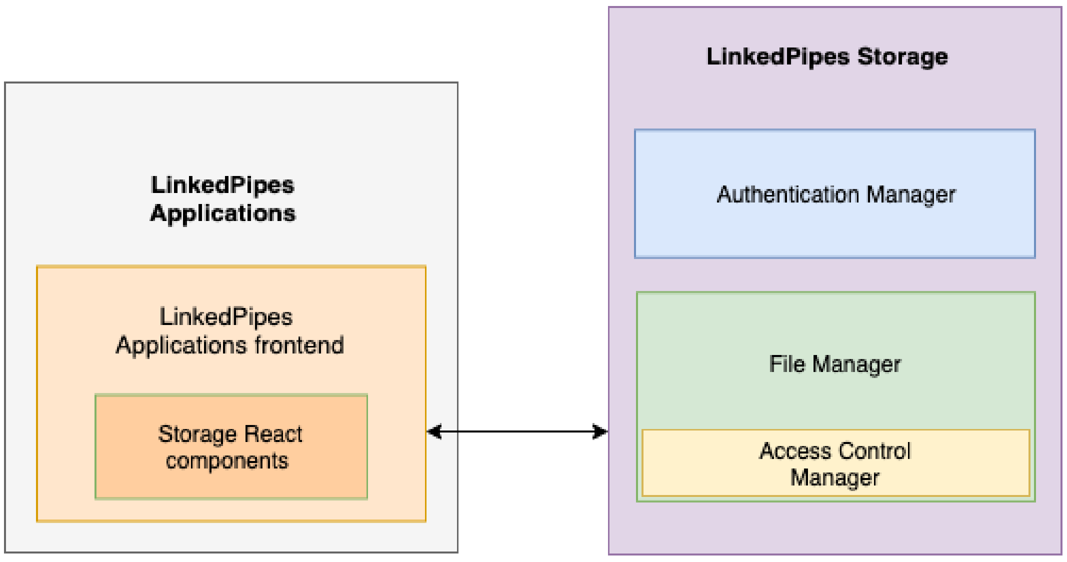


Figure 4.2: Main abstractions of LPS.

- WebID-OIDC - alternative authentication protocol based on OAuth2 and OpenID Connect protocols and adjusted to support the concept of WebID. This is, in fact, the first authentication option provided by LPS, later chapters will provide the reasoning behind why this ended up being a more intuitive option to cover the authentication requirement for LPA.

As already mentioned in Chapter 3, the usage of *WebID* protocols is one of many benefits of Solid, in contrast with popular authentication protocols used in centralized silos, it is completely agnostic to specific authentication mechanisms, allowing our single abstraction to support any arbitrary *WebID-OIDC* compliant Solid provider.

Interacting with frontend

Sequence diagram on Figure 4.3 demonstrate an example on how the Authentication Manager will be used within the LPA Frontend and how it will interact with Solid Providers. The user agent entity represents a typical lay LPA user interacting with the platform through the frontend component. The sequence flow consists of the following steps:

1. The user clicks on the 'Authenticate' button. This is the starting point of this sequence diagram that serves as a trigger for LPA to invoke the LPA package.
2. The frontend component calls the Authentication Manager and awaits for a callback. As the main input, it supplies the information about the Solid provider that the user would like to use.
3. The Authentication Manager contacts the Solid provider and requests the provider authentication web page. Each provider conforming to Solid specification should contain that page.

4. Depending on the browser environment of the user, he gets redirected to the provider's authentication web page either in a new browser tab or a popup dialog.

5. The user selects the preferred authentication and inputs the credentials. At the moment of writing this thesis, popular Solid server implementations like node-solid-server support both WebID+TLS and WebID-OIDC specifications.

6. The Authentication Manager receives the callback from the provider and sets the authenticated user session token.

7. Frontend component receives the callback from the Authentication Manager, indicating that the user is authenticated.

8. As the last step, the frontend redirects the user into the homepage dashboard of LPA platform.

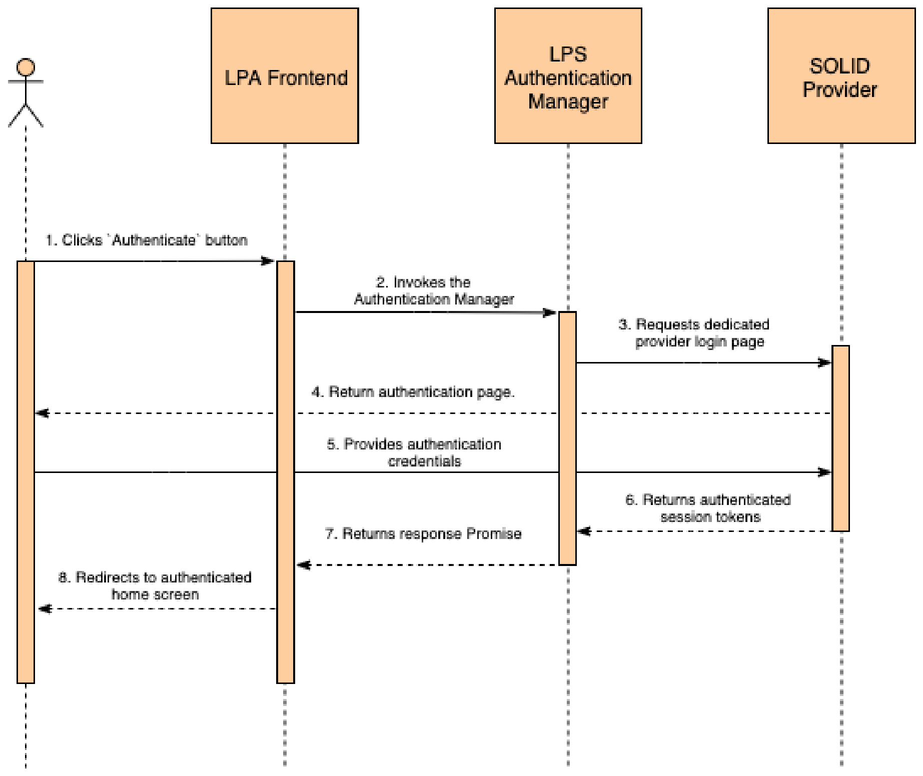

LPA Frontend
LPS Authentication Manager
SOLID Provider
1. Clicks `Authenticate` button
2. Invokes the Authentication Manager
3. Requests dedicated provider login page
4. Return authentication page.
5. Provides authentication credentials
6. Returns authenticated session tokens
7. Returns response Promise
8. Redirects to authenticated home screen

Figure 4.3: Sequence diagram for *authenticate* operation invoked from LPA frontend.

4.2.2 File Manager

The File Manager is responsible for implementing the CRUD operations for the Solid providers that are compliant with Linked Data Platform specification. The Linked Data Platform (LDP) is a specification that is reused and extended in the Solid specification to describe the REST API for interacting with LDP Resources and LDP Containers [19]. LDP Resources and LDP Containers are, in some sense, the basic building blocks of any Solid POD, since they allow users to create any files and folders.

As mentioned earlier, the Solid specification is re-using the LDP specification to provide a RESTful set of operations to interact with any compliant implementation of that specification.

The LDP has an extensive set of resource types that are suited for defining the folders and files. However, the architecture provided is relying on two common resource terminologies that were simple enough to cover the requirements without complicating the design of the architecture. Those resources are defined as follows:

- Linked Data Platform Resource (LDPR) is an HTTP resource whose state is represented in any way that conforms to the simple lifecycle patterns and conventions, in other words any resource that can be *created*, *updated*, *deleted* and *red*. The LDP servers process the CRUD operations to manipulate the lifecycle of LDPRs.

- Linked Data Platform Basic Container (LDP-BC) is a Linked Data Platform Basic Container. An LDP-RS representing a collection of linked documents that respond to client requests for creation, modification, and enumeration of its linked members and documents, and that conforms to the simple lifecycle patterns and conventions.

Creating resources

The essential operation that is at the core of most functional requirements defined earlier is the ability to create a resource in a POD. Using the NSS server as a basis, the general convention for creating LDP resources is a POST HTTP request providing a link to the POD using the path where resource needs to be created. As demonstrated on Figure 4.4 the creation consist of the following steps:

1. LPA frontend invokes the *FileManager* abstraction with a request specified in `ResourceConfig` abstraction. The configuration provided contains information on the type of resource, whether it is a folder or a file. Additionally, it describes the required information to identify the resource within the POD.

2. LPS constructs an HTTP POST request to create the resource in POD in Solid server, the response is forwarded asynchronously.

Reading resources

Reading the resources demonstrated on Figure 4.5 is a straightforward set of interactions with an Solid specification compliant server, and it can be described as follows:

1. LPA frontend invokes the *FileManager* abstraction with a request specified in `ResourceConfig` abstraction. The configuration provided contains information on the type of the resource, whether it is a folder or a file, as well as all required information to identify the resource within the POD.

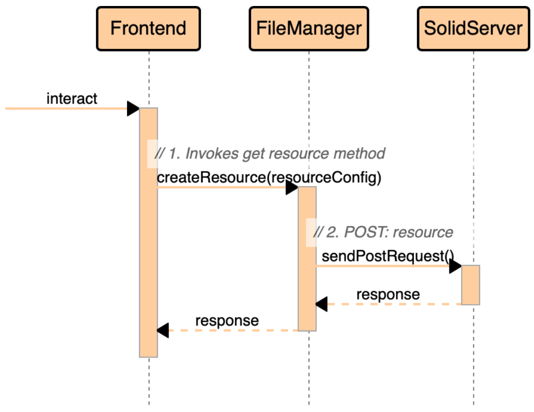


Figure 4.4: Sequence diagram for POST resource operation invoked from StorageFileManager.

2. LPS constructs an HTTP GET request to obtain the information from Solid server from the provider, the response is forwarded asynchronously.

As demonstrated in the steps above, the LDP specification provides advantages by narrowing down the amount of resource lifecycle related calls as a set of elementary CRUD operations. Another important detail is that every single resource created by LPS is an RDF file. We will cover more information and demonstrate why it is implemented in that way in the proceeding chapter.

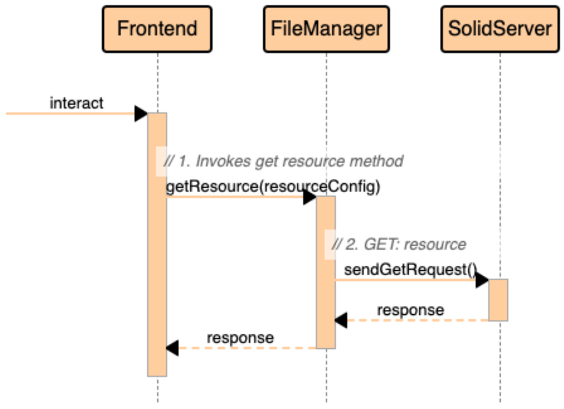


Figure 4.5: Sequence diagram for GET resource operation invoked from StorageFileManager.

Renaming resources

The rename operation covers one of the functional requirements on the LPA platform. It is based on a simpler CRUD operations such as *create* and *read* resource described in earlier sections. The goal of this function is to provide the ability for a LPA developer to implement a functionality to let LPA platform users to choose and manipulate their storage configuration folders.

The flow in the sequence diagram displayed in Figure 4.6 consists of the following steps:

1. LPA frontend invokes the *FileManager* abstraction with a request specified in the `ResourceConfig` abstraction, which is similar to the input steps in the previous sequence diagrams.
2. LPS has a conditional check to see whether the new provided resource configuration is not accidentally the resource under the same title. This step is followed if the new path and title for a configuration are not equal to an old configuration.
   (a) Invoke the copy method that will, depending on the resource, either copy it directly as a file, or copy it recursively if it is a folder. For the sake of reducing the unnecessary details, the internals of the copy resource call is not displayed on this diagram.
   (b) After copying the content of configuration into a new destination, the old resource is removed using the delete operation.
3. LPS returns a successful promise since no renaming is invoked in that case.

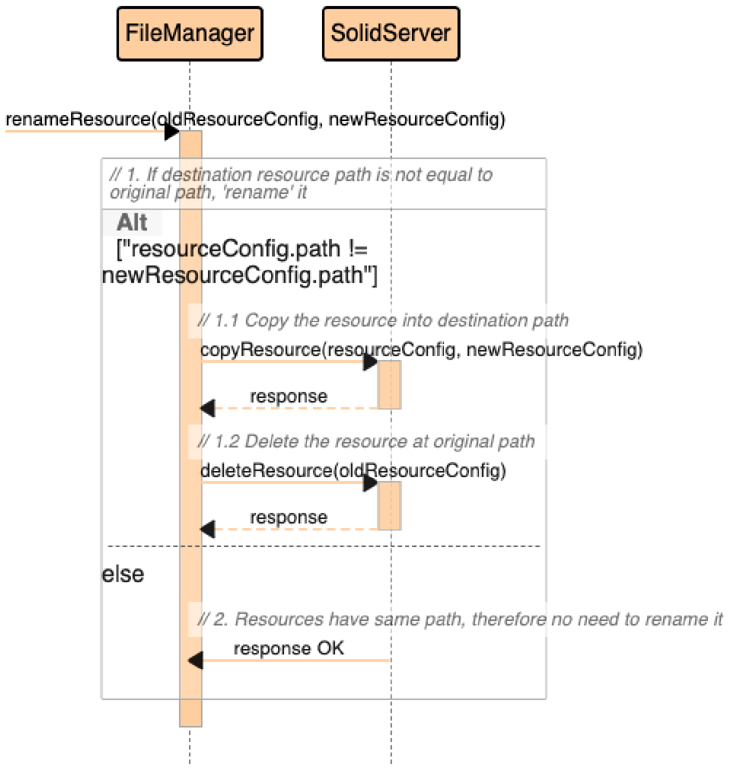

FileManager
SolidServer
renameResource(oldResourceConfig, newResourceConfig)
// 1. If destination resource path is not equal to original path, 'rename' it
Alt
["resourceConfig.path != newResourceConfig.path"]
// 1.1 Copy the resource into destination path
copyResource(resourceConfig, newResourceConfig)
response
// 1.2 Delete the resource at original path
deleteResource(oldResourceConfig)
response
else
// 2. Resources have same path, therefore no need to rename it
response OK

Figure 4.6: Sequence diagram for a complex operation to rename a particular resource in StorageFileManager.

Deleting resources

The deletion of resources is slightly more complicated in comparison with the trivial GET, POST, PUT, and PATCH operations described in earlier sections. The main goal here is to differentiate the type of the resource, and depending on whether it is an LDP Basic Container or generic LDP Resource, act accordingly to delete all files under that resource. The flow from Figure 4.7 can be described as follows:

1. Similar to the previous sequence flows, the action is triggered by an input request from LPA frontend
2. If the resource is and LDP Resource, *FileManager* trivially sends a single DELETE request to the server and returns the response
3. If the resource is an LDP Basic Container:
   (a) Invokes a method for recursively deleting the contents of a folder.
   (b) Within that method, it performs a call that fetches the raw RDF describing the LDP Basic Container and parses the resources contained within using.
   (c) Iterate over files, remove them individually as in the first step.
   (d) Iterate over folders, remove them individually as in the first step

One of the assumptions made in this recursive operation is the interactions with ACL files when a recursive delete is performed. As mentioned in the introduction to this chapter, Solid community is still at its beginnings, and there are many significant improvements yet to introduce. In order to make the delete operation more generic, the DELETE interactions with ACL files had to be removed due to a more strict access policy established between providers of the server and Solid apps. The following section will dive deeper into Access Control Managements and related functionality.

Classes overview

File manager is the biggest of all three abstractions displayed on the Figure 4.2. Therefore, in order to better describe the architecture of it, a more detail class diagram is provided on Figure 4.8.

- The core class available to LPA developers is the *StorageFileManager*. All of the functions inside are intended to be public, static and asynchronous.
- *ResourceConfiguration* is a wrapper for LDP Containers and LDP Resources. It allows a developer to specify *title*, *path* and *type* of a resource. Most of the operations within the *StorageFileManager* are operated with the resources encapsulated into *ResourceConfiguration* classes. It also provides a few helper getters that allow to generate an absolute path to a resource within a pod.
- The *AccessControlConfig* is a subclass of *ResourceConfig*. It allows to specify the access control modes to individual resources. Additionally, it introduces a few extra functions to generate the absolute path that includes the ACL file extension.
- The *SolidResource* is a simple interface that includes various details specific to the resource. Some of the fields provided are also directly used by *StorageFileManager* abstraction during construction of CRUD calls to Solid providers.

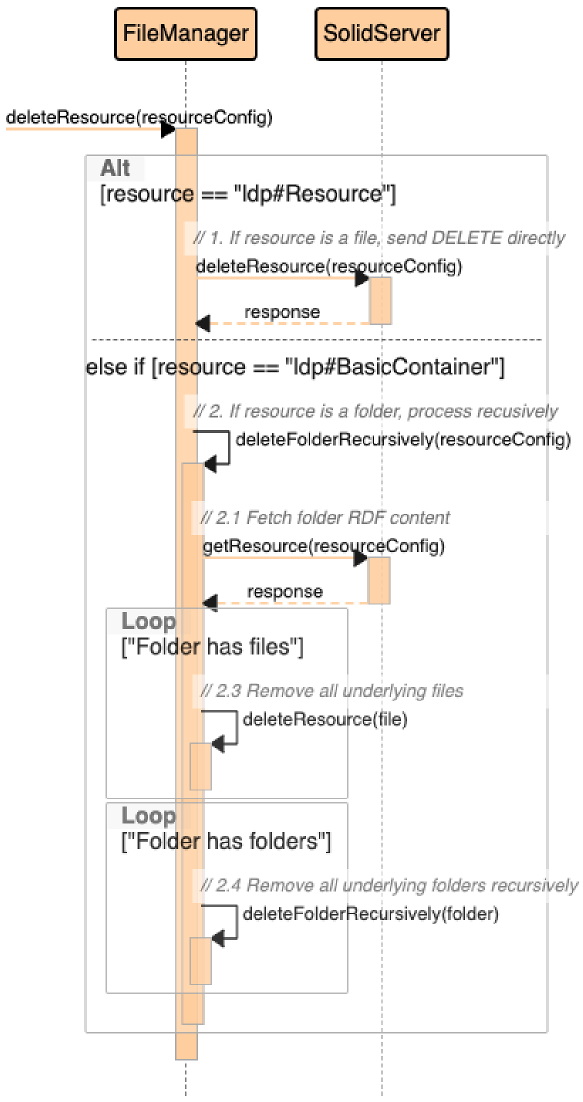

FileManager
SolidServer
deleteResource(resourceConfig)
Alt
[resource == "ldp#Resource"]
// 1. If resource is a file, send DELETE directly
deleteResource(resourceConfig)
response
else if [resource == "ldp#BasicContainer"]
// 2. If resource is a folder, process recusively
deleteFolderRecursively(resourceConfig)
// 2.1 Fetch folder RDF content
getResource(resourceConfig)
response
Loop
["Folder has files"]
// 2.3 Remove all underlying files
deleteResource(file)
Loop
["Folder has folders"]
// 2.4 Remove all underlying folders recursively
deleteFolderRecursively(folder)

Figure 4.7: Sequence diagram for a complex operation to delete a particular resource in StorageFileManager.

4.2.3 Access Control Manager

The primary responsibility of the Access Control Manager is a subset under File Manager abstraction. It is designed to support the File Manager entities and provide an ability to wrap them with Web Access Control compliant settings. In other words, this allows a developer of LPA to programmatically control the Read and Write access to any resource inside an arbitrary Solid POD by utilizing the developer-friendly interfaces and classes defined within the scope of this abstraction. Essentially, every ACL file is nothing more than yet another RDF resource with few extra features. Hence, the abstraction was placed under the File Manager since the core logic is concerned with similar resource lifecycle manipulations.

It is yet another wrapper on top of the functionality provided by any Solid compliant servers. In this case, being Solid compliant also assumes conforming to Web Access Control or WAC specification. It defines the so-called Access Control Resources, which are entities serving as the declaration of access control

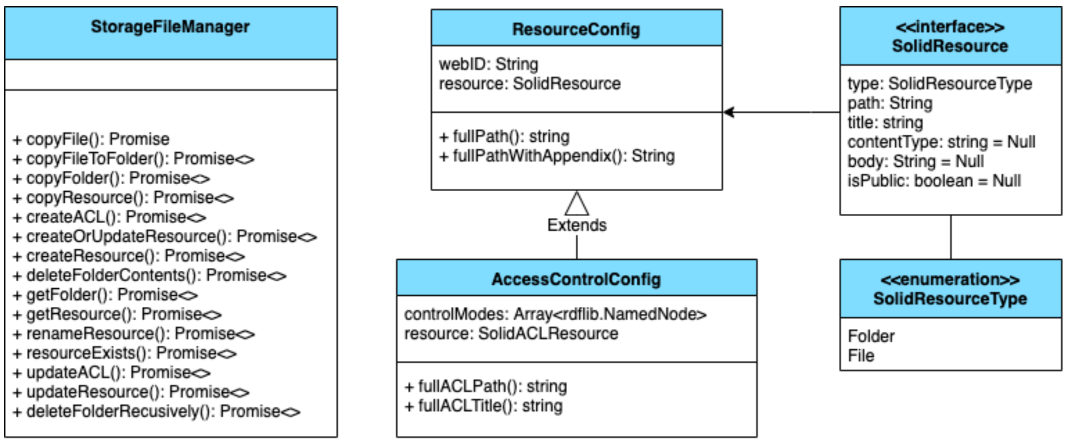

StorageFileManager
+ copyFile(): Promise
+ copyFileToFolder(): Promise<>
+ copyFolder(): Promise<>
+ copyResource(): Promise<>
+ createACL(): Promise<>
+ createOrUpdateResource(): Promise<>
+ createResource(): Promise<>
+ deleteFolderContents(): Promise<>
+ getFolder(): Promise<>
+ getResource(): Promise<>
+ renameResource(): Promise<>
+ resourceExists(): Promise<>
+ updateACL(): Promise<>
+ updateResource(): Promise<>
+ deleteFolderRecusively(): Promise<>
ResourceConfig
webID: String
resource: SolidResource
+ fullPath(): string
+ fullPathWithAppendix(): String
Extends
AccessControlConfig
controlModes: Array<rdflib.NamedNode>
resource: SolidACLResource
+ fullACLPath(): string
+ fullACLTitle(): string
<<interface>>
SolidResource
type: SolidResourceType
path: String
title: string
contentType: string = Null
body: String = Null
isPublic: boolean = Null
<<enumeration>>
SolidResourceType
Folder
File

Figure 4.8: A higher level class diagram of classes contained within StorageFileManager abstraction.

privileges for a specific resource. Within the context of Solid specification this means managing access rights to resources in Solid PODS for various WebIDs.

The main function provided by the abstraction is an ACL file generator. The flow on the Figure 4.9 demonstrated the process of generation, parsing, and serialization of ACL files:

1. Create Access Control triples for specified input resource configuration files.
2. Create individual triples defining ACL configuration for a resource owner.
3. Create individual triples for public access if the resource itself is marked as public.
4. Gather and parse those triples into a rdflib abstraction representing an RDF Graph.
5. Serialize this graph instance into a string representing a TTL file.
6. Finally performs an HTTP call, that uses PUT to create an ACL file attached to a specified resource.

This concludes the section demonstrating the essential abstractions within the LPS package. The consecutive chapters dedicated to Documentation will cover and provide more details regarding less significant classes and utilities available inside the package.

4.3 LinkedPipes Applications Ontology

Relying on LDP specification extended by Solid was not the only goal while designing an architecture to satisfy the requirements of LPA. It was also essential to use the advantages of Linked Data in general. As stated in introductory chapters, as well as the original paper, one of the significant benefits of any Solid compliant server is that everything is either an RDF file or has the metadata expressed as an RDF file [20][21].

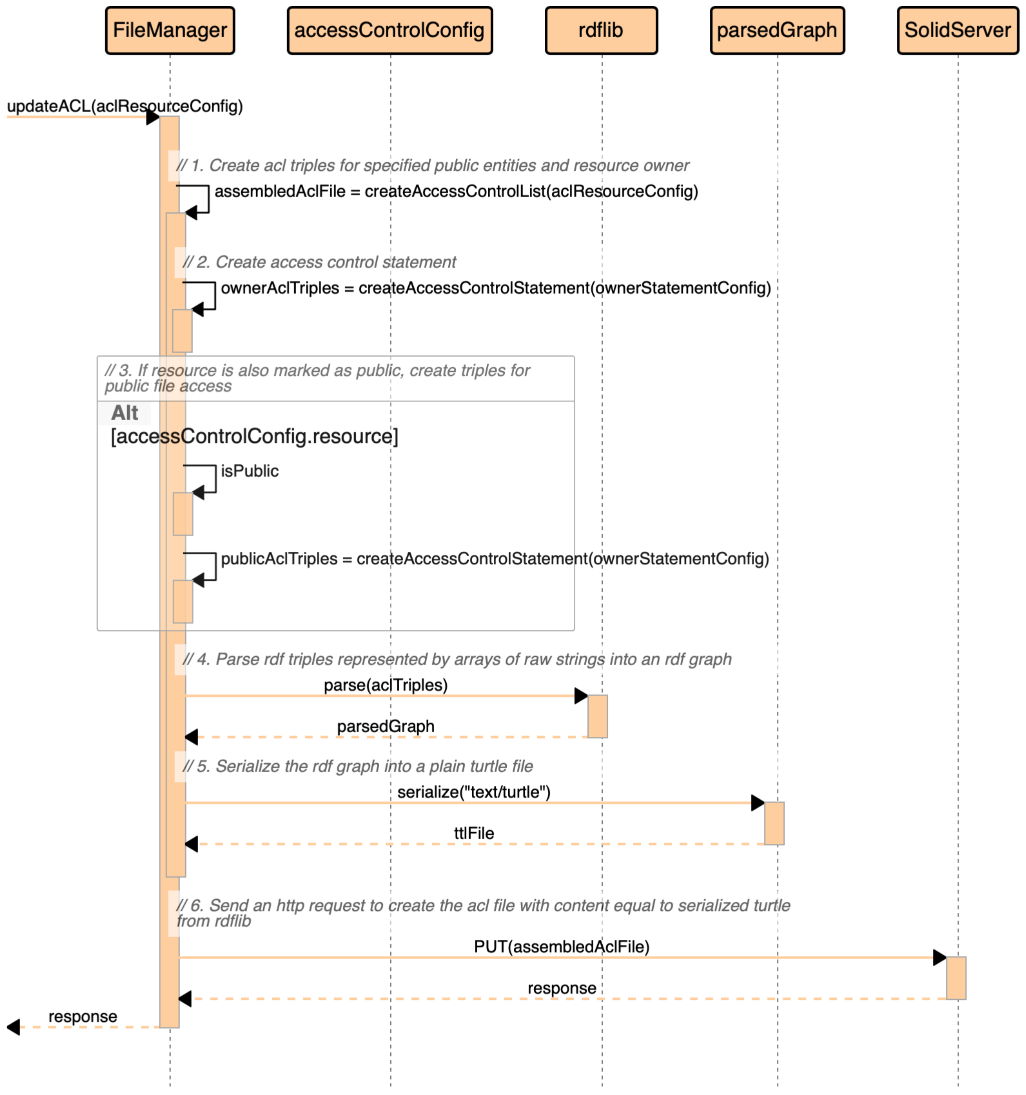

FileManager
accessControlConfig
rdflib
parsedGraph
SolidServer
updateACL(aclResourceConfig)
// 1. Create acl triples for specified public entities and resource owner
assembledAclFile = createAccessControlList(aclResourceConfig)
// 2. Create access control statement
ownerAclTriples = createAccessControlStatement(ownerStatementConfig)
// 3. If resource is also marked as public, create triples for public file access
Alt
[accessControlConfig.resource]
isPublic
publicAclTriples = createAccessControlStatement(ownerStatementConfig)
// 4. Parse rdf triples represented by arrays of raw strings into an rdf graph
parse(aclTriples)
parsedGraph
// 5. Serialize the rdf graph into a plain turtle file
serialize("text/turtle")
ttlFile
// 6. Send an http request to create the acl file with content equal to serialized turtle from rdflib
PUT(assembledAclFile)
response
response

Figure 4.9: A higher level class diagram of classes contained within StorageFileManager abstraction.

The first task was to identify and analyze the kind of data LPA is storing. The initial implementation on LPA codebase was a simple application configuration JavaScript object that assembles all required configuration information for an application by the time a user of LPA hits the *Publish* button. In other words, we were given a JavaScript object to operate. Let us describe how this input requirement was taken into consideration while designing a solution that is both optimized for Solid and satisfies the requirement.

4.3.1 Using Web Ontology Language

The Web Ontology Language or OWL, is a commonly used knowledge representation language used for:

- Designing ontologies.
- Formalizing domains.
- Defining domain specific classes and properties.

As a first step, the JavaScript object that was used to represent LPA configurations was formalized into a JSON Schema [3]. Based on that Schema, the initial OWL ontology was designed using Stanford's *Protégé* [4], which is a convenient open-source ontology editing tool. The class hierarchy on Figure 4.10 demonstrates the draft of the ontology created based on that Schema.

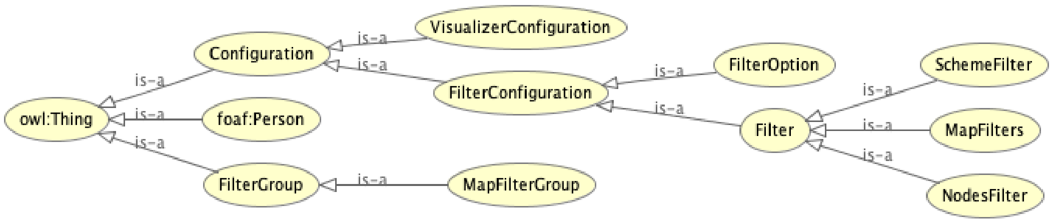


Figure 4.10: A class hierarchy visualization of LPA vocabulary.

foaf:Person

The *foaf:Person* node on Figure 4.10 refers to a WebID profile of an author of this configuration. Due to the generality of the WebID, there was no need to make a specific subclass from the foaf:Person class for that use-case.

Configuration

The *Configuration* class is the main generic abstraction for all LPA configurations. Class hold the generic object properties that can be described as follows:

- *author*, this refers to the foaf:Person class stated before, and identifies the person with his WebID as an author of this LPA configuration.
- *published*, a date timestamp used to identify when configuration was created and published.
- *title*, represents the title given to that LPA visualizer.

The class is also a parent class for two subclasses titled *VisualizerConfiguration* and *FilterConfiguration*. More details on them provided in the implementation chapter. However, at this point, it is sufficient to understand that there are two main configuration types. One of them tied to the visualization, and the other is to filters that allow filtering information displayed on visualizers.

FilterGroup

The *FilterGroup* class is closely related to Filter and FilterConfiguration and is used to reference the aggregation of visualizer specific filters. Since it does not necessarily need to inherit the object properties of FilterConfiguration, it is inheriting from generic *owl:Thing* class.

[3]`https://json-schema.org`
[4]`https://protege.stanford.edu`

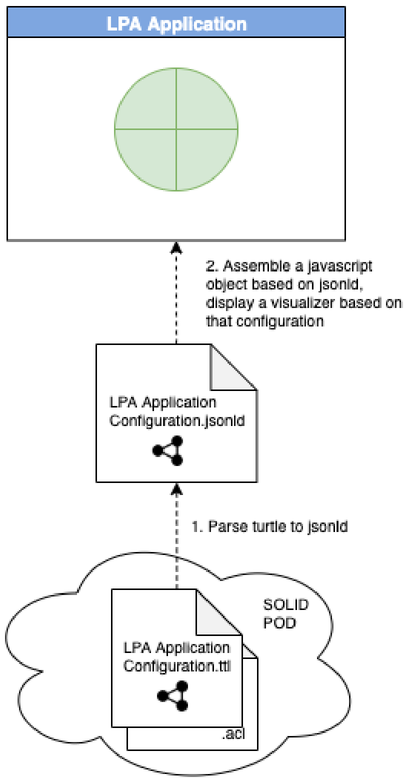

LPA Application
2. Assemble a javascript object based on jsonld, display a visualizer based on that configuration
LPA Application Configuration.jsonld
1. Parse turtle to jsonld
SOLID POD
LPA Application Configuration.ttl
.acl

Figure 4.11: A formal representation of LPA configurations expressed as RDF files using the LPS vocabulary.

In order to simplify adoption of this ontology in LPA frontend, the resulting ontology was converted into a JSON-LD Schema [5], as shown on Figure 4.11. However, due to some limitations in NSS, the individual `jsonld` configurations had to be converted into TTL files when stored in Solid. For more details on adoption and implementation of this ontology refer to Chapter 5.

4.4 Storage Component Design

It is important to note that the LPS is not just the external package completely isolated from the LPA frontend, it is also a set of React components that attempt to blend in into the user interface guidelines of the frontend. Now when the major components of the LPS package as well as the LPS vocabulary are described, let us dive deeper into the design considerations done on the frontend side inside the LPA codebase.

The LPA frontend codebase was implemented using React[6] and was utilising a modern stack of frontend development tools, all of those had to be taken into consideration to design React Component responsible for interactions with the storage. This section will cover the design conventions inherited from LPA as well as a detailed overview of User Interfaces conforming to Material Design[7] conventions that were strongly utilized in LPA frontend. In addition to that, it

[5]`https://json-ld.org`
[6]`https://reactjs.org`
[7]`https://material.io/design`

will also revisit the functional requirements and provide UI design proposals that are later implemented in the implementation section.

4.4.1 Designing React Components

At the root, LPA frontend identifies two main types of components that are logically separated into folders titled as follows:

- `Components` folder, these usually contain elements that are used in more than one webpage throughout the project, such as buttons, switches, image wrappers and etc.
- `Containers` folder, represent complex react components that are basically rendering individual webpages or sub-elements of webpages that deal with complex user interaction scenarios.

Simple components

Whenever an individual component needs to be implemented and it will be used in multiple webpages throughout the project, it is being placed into `Components` folder.

There are two main types of components that can be placed into `Components` folder and have different design conventions:

- Simple stateless component responsible for plain rendering.
- A complex component that needs to aggregate multiple sub-components, manage external state, internal states and etc.

This is not a strict guideline defined by LPA developers. However, if a component becomes too complex, as demonstrated on Figure 4.12 the intent is to split component into separate component responsible for rendering and component that manages states of the stateless component. This allows easier navigation within frontend codebase as well as faster code debugging.

Therefore, the logical decomposition of components by their complexity is the concludes the only major design convention that was required by LPA frontend. Let us go over the details of each individual component in the following sections.

4.4.2 Authentication View

Authentication is the entry for the LPA platform, and LPS covers the design and implementation of that view since LPA relies on Solid to perform the WebID authentication.

As previously demonstrated in Subsection 4.2.1, the LPS package handles the authentication by redirecting the requests and responses between the browser of the user and the Solid provider server. The mock user interface on Figure 4.13 demonstrates a basic mock for a authentication webpage. There are several ways to authentication available to a LPA user with a WebID profile in any Solid-provider:

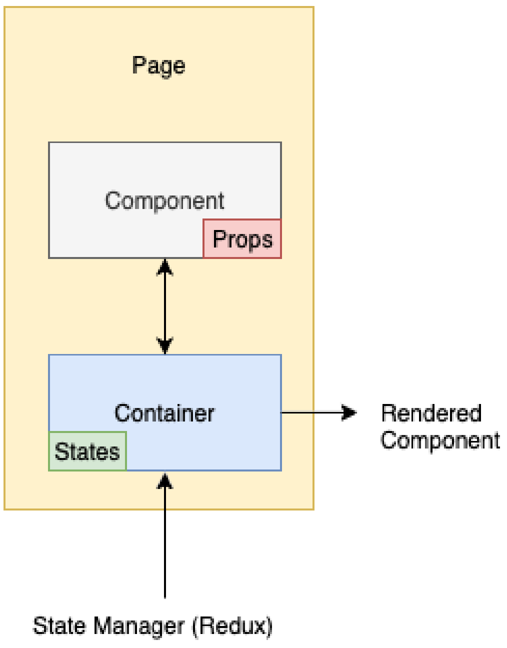


Figure 4.12: React container abstraction decomposition following LPS design conventions.

- *Provider authentication*, user clicks on a dropdown pane and selects the name of the default providers. The default providers supported by LPS are *inrupt.net* [8] , *solid.community* [9] and a self-hosted LinkedPipes server available at *lpapps.co:8443* [10].
- *WebID authentication*, similar to previous option but instead user is able to provide his WebID and be redirected right into the login page of his provider.

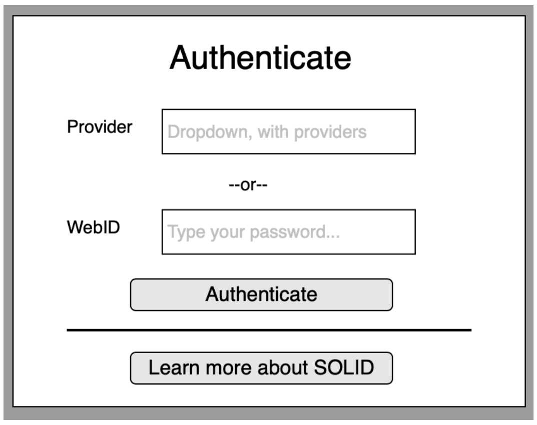


Figure 4.13: Mock UI for Authentication webpage in LPA Frontend.

The additional user interface elements are defined as follows:

[8]`https://inrupt.net`
[9]`https://solid.community`
[10]`https://lpapps.co:8443`

- *Authenticate* button executes the authentication sequence depending on the options that users have chosen, which are either Provider or direct WebID authentication.

- *Lean more about Solid* redirects users not familiar with concepts of Solid directly into the home page of the inrupt project.

4.4.3 Storage Dashboard

Referring back to the functional requirements stated in the first chapters, the ability to interact with the LPS is an essential feature allowing users of LPA platform to manipulate their Applications. The mock displayed on 4.14 is a webpage accessible via the home dashboard. There are two main display modes:

- The *My apps* tab, is a React component that fetches all RDF resources in root LPS folder containing applications created by a user.

- The *Shared* tab is a React component fetching all RDF resources in a shared LPS folder containing applications created and shared by a particular user with other users of LPS platform.

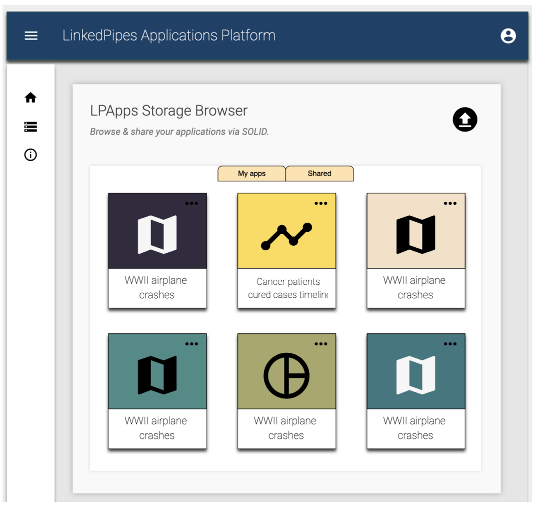


Figure 4.14: Mock UI for Storage Dashboard webpage in LPA Frontend.

The idea behind the dashboard is that each card is a visual representation of LPA configuration. As mentioned earlier in Subsection 4.3.1, each configuration is expressed using the LPS ontology and stored as an RDF file inside Solid. The card collection view pulls each of the configurations from the container, storing them in Solid and populates the content.

The user of LPA has a set of straightforward interactions that can be performed on card:

- *Clicking on card*, redirects the user to the webpage displaying the visualizer.

- *Clicking on sub-menu icon*, reveals a popup where users can choose to delete, rename, or share the visualizer.

4.4.4 Storage Control Panel

An ability to authenticate, create, and publish an application using Solid are essential requirements stated by LPA. However, users also need to have basic functionality to manipulate the data stored by the LPS within their PODs. Therefore a mock design demonstrated on Figure 4.15 provides the basic functionality described as follows:

- `Update` folder, allows users to switch their root folder into any other folder within their POD.
- `Copy` folder allows users to copy all content from the current root configurations folder into a new or existing folder.
- `Move` folder allows users to move all content from the current root configurations folder into a new or existing folder.

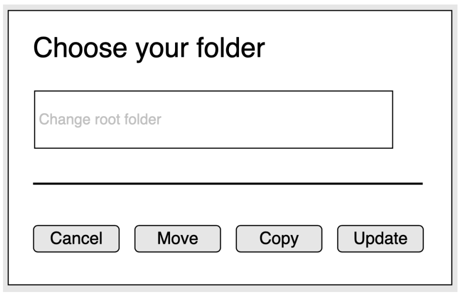


Figure 4.15: Mock UI for Storage Dashboard webpage in LPA Frontend.

The diagrams in Section 4.2 demonstrate the exact sequence of interactions between LPA, LPS and Solid providers when operations like *move* or *copy* folder are invoked.

To sum up, this chapter provided an overview of three major aspects of LPS:

- The npm package contains the core abstractions architectured to be separated from the LPA with the intent to improve Solid related code maintainability and testing. The sequence diagrams of all main CRUD operations performed on RDF resources in Solid PODs.
- The LPS ontology, an RDF vocabulary designed specifically for LPA configurations, taking full advantage of Solid. In other words, giving an ability not simply to store the LPA configurations, but also perform any complex querying on them using SPARQL.
- The frontend React components, the UI mocks of components inside LPA frontend, conforming to conventions of the LPA.

In the next chapter, a detailed review of the implementation of the architecture will be provided.

5. Implementation

The following chapter is going to cover the implementation of LPS package, the frontend components as well as the ontology. The first part of the chapter dedicated to the implementation of the package will provide a detailed overview of the decisions made on the development stack, the main challenges invoked in refactoring the original LPA codebase, and making the Solid related functionality more generic. The frontend components section will dive deeper into the implementation of the mocks provided in Chapter 4, main decisions, and challenges while developing under React. The ontology Section 5.2 will describe how the designed LPS vocabulary was converted into a OWL file, converted into JSON-LD Schema and later integrated into the LPA frontend. Lastly, an overview of the implementation results will be presented by reiterating over the defined LPA requirements and how they were satisfied by the implementation.

5.1 Storage Package

The initial implementation of LPA frontend was written in `JavaScript ES6` [1] and `React` framework. The development stack also included tools such as `Babel` [2] compiler and `Webpack` [3] package bundler.

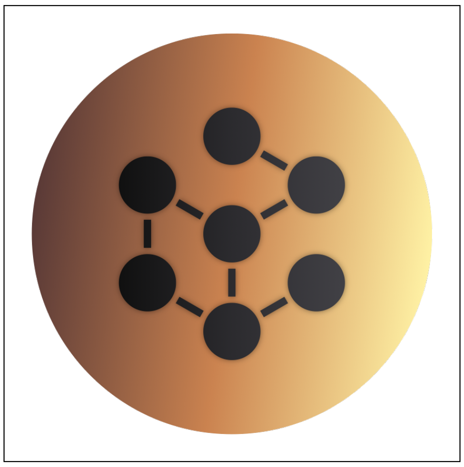


Figure 5.1: Official LPS package logo designed by author.

As the number of features and functionality to cover was increasing, the decision was made to separate the Solid storage-related functionality into a separate npm package and call it LPS package. This section will provide an overview of preliminaries chosen for the implementation of LPS package as well as the specifics of implementations of each abstraction defined in Section 4.2.

[1]`http://es6-features.org`
[2]`https://babeljs.io`
[3]`https://webpack.js.org`

5.1.1 Preliminaries

As briefly mentioned earlier, there are several main libraries used inside the LPS package:

- `rdflib` is a low-level RDF library, that mainly provides the functionality to Read and Write RDF in many popular formats, a querying store and an ability to use SPARQL queries.
- `solid-auth-client`, a browser library that implements Solid specifications for providing authentication. This is the main library used to enable the authentication into LPA platform.

Due to the complexity of the usage of the stated libraries, specifically `rdflib`, having an environment and a language that provides a fully-featured object-oriented programming and static type-checking would directly affect the code maintainability and usage. Hence, the codebase of LPS was implemented using `TypeScript` [4]. TypeScript is a strict syntactical superset of JavaScript that provides optional static typing and better object-oriented programming capabilities. When referred to files containing TypeScript code, the consecutive sections and chapters will refer to as *TS* files.

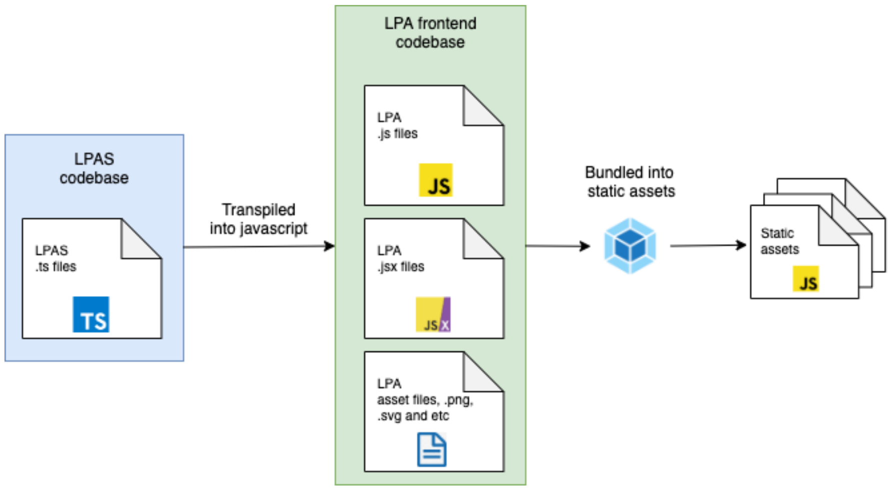


Figure 5.2: A diagram demonstrating the process of transpilation of LPS package and bundling LPA frontend with Webpack

As demonstrated on Figure 5.2, integrating the package into the LPA codebase was done using the TypeScript compiler that allows transpilation into ES6 compatible JavaScript syntax. The package, as well as the rest of the LPA codebase, is later bundled into a set of static assets using Webpack. The assets mainly consist of a set of media files such as PNG and SVG files and a large JS file that contains the whole LPA frontend. One of a few disadvantages with that approach is that the initial loading of the frontend might take a few seconds to load in the browser. Afterward, the interaction with the platform is seamless and does not involve any additional loading.

[4]`https://www.typescriptlang.org`

Project structure

The structure of the LPS package is simple and straightforward and can be demonstrated as follows:

```
- build       # Transpiled JavaScript code
- markdown    # Markdown assets
- src         # Root project folder
  - lib       # Main library codebase
    - common  # Utilities and helper functions
       - ...     # TypeScript tests and core abstractions
  - types     # Custom user-defined type definitions
- docs        # Static html with library documentation
_ ...         # Readme and various configuration files
```

Listing 3: LPS package folder structure description.

The proceeding sections will describe the individual abstractions mentioned in Section 4.2.

5.1.2 Authentication Manager

This section will continue the architectural description of the Authentication manager described in Subsection 4.2.1, describe the implementation and provide examples of how the abstraction is used inside LPA.

The *AuthenticationManager* is a Singleton class, instantiated only once and utilized both in the package itself as well as being invoked from LPA frontend codebase. The reason for the class being implemented as a singleton is due to the fact that it wraps the functionality of `solid-auth-client` library, and it provides a singleton instance as well. Aside from providing the WebID authentication, `solid-auth-client` implements a WebID OIDC specific *fetch* functionality. The *FETCH API* is originally a JavaScript API that provides an ability to send asynchronous HTTP calls. The implementation in `solid-auth-client`, is based on `isomorpic-fetch`, which is a third-party framework that implements the *Fetch API* both for browsers and Node.js. Hence, the abstraction is used for:

- Authentication, and ability to track the user session with callbacks.
- Sending HTTP calls to Solid server.

In other words, once the client is logged in the Solid app, consecutive interactions are performed via *fetch* function that is conveniently wrapped in the *AuthenticationManager* abstraction.

As demonstrated on Figure 5.3, the class consist of a set of public methods described as follows:

- *getInstance()*, this public method returns a singleton instance to an *AuthenticationManager*.

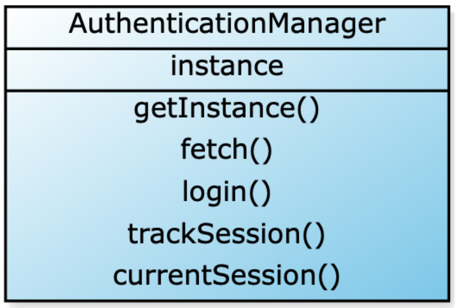


Figure 5.3: A class diagram generated directly from a TypeScript file, demonstrating an implemented *AuthenticationManager* abstraction

- *fetch()*, a wrapper redirecting the call to `solid-auth-client` fetch method.
- *login()*, a wrapper redirecting the call to `solid-auth-client` login method.
- *trackSession()*, a method with asynchronous callback notifying the listener when a logout or login operation is performed.
- *currentSession()*, a method returning the instance of a `solid-auth-client` Session object that contains relevant information about the authenticated user and his WebID.

Referring back to Figure 4.3, there are several places in LPA codebase where the *AuthenticationManager* is invoked directly. The implementation of React components will be covered in the proceeding section, but the invocation of the abstraction itself can be described as follows:

- *Component layouts* are special high-level react containers that wrap every other container inside LPA frontend. They are differentiated by *public* and *private*. The *private* components reactively monitor the authenticated session of a user and redirect them back to the authentication screen whenever the value of the session becomes `undefined`. It is important to note that the session object from *AuthenticationManger* is duplicated in LPA frontend as a Redux state. Therefore, any changes in the original session object are reflected on that state and triggers re-rendering of layout components.
- *Authentication component functions*, are the functions being invoked when user attempts to perform the authentication. In other words, this is the input that triggers the flow demonstrated earlier on Figure 4.3.
- *The App router*, the main class in LPA that serves as an entry point and utilizing the `react-router` [5] package, contains a function that invokes the *trackSession()* method in *AuthenticationManager*. This directly links to the session object and updates the changes from original session to internal Redux state.

The usage of both *currentSession()* and *login()* methods from *AuthenticationManager* can be observed below:

[5] `https://www.npmjs.com/package/react-router`

```
login = async (idp, callbackUri) => {
    const session = await
     ↪  AuthenticationManager.currentSession();
    if (!session)
      await AuthenticationManager.login(idp, {
        storage: localStorage
      });
    else {
      Log.info(`Logged in as ${session.webId}`);
      return session;
    }
};
```

Listing 4: An implementation of *login()* wrapper.

To sum up, the *AuthenticationManager* is a simple and straightforward singleton abstraction that wraps the `solid-auth-client` library and only necessary functions from the wrapped library to be used inside LPS package. The examples of invocation from within the LPS package are limited to directly calling the *fetch()* method whenever an HTTP request is assembled and needs to be executed, more details on that will be described in a section dedicated to *FileManager* abstraction.

5.1.3 File Manager

In this subsection, we will continue on the *FileManager* abstraction described in Subsection 4.2.2, provide the specifics on implementation as well as a detailed overview of each method inside the abstraction.

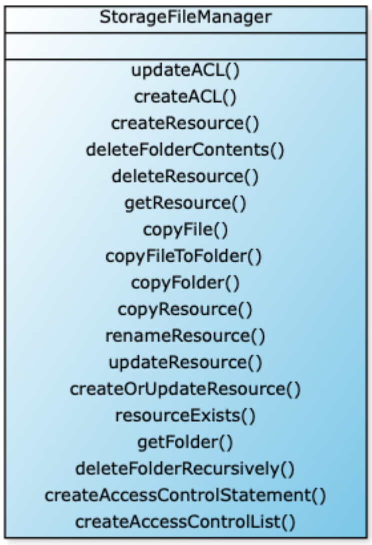


Figure 5.4: A class diagram generated directly from a TypeScript file, demonstrating an implemented *FileManager* abstraction

Similar to the *AuthenticationManager*, the *FileManager* abstraction is implemented as a TypeScript class providing a set of public static methods. It is responsible for all core CRUD operations with Solid resources. It is important to note that all HTTP requests to Solid server are made using the *fetch()* method invoked via *AuthenticationManager*. In other words *FileManager* relies on *AuthenticationManager* when performing HTTP requests. To follow up a class representation on Figure 5.4, the methods can be described as follows:

- *createResource()*, a function is responsible for creating new Solid resources.
- *deleteFolderContents()*, a function is mainly used for cleaning up the content of a particular folder. It iterates the contents and recursively deletes underlying folders and files.
- *deleteResource()*, a function responsible for deleting individual resources from a Solid POD. Following the sequence flow presented on Figure 4.6, depending on the type of the resource it will either remove it directly or attempt to clean it recursively if the resource is an *LDP-BC*.
- *getResource()*, a basic function executing a GET call to obtain a resource content from a Solid server.
- *copyFile()*, a basic function performing a copy operation on an *LDPR*.
- *copyResource()*, a generic method performing a copy operation on a resource. Depending on the type of the resource it either invokes the *copyFile()* directly or, if resource is an *LDP-BC*, it performs a recursive copying.
- *renameResource()*, a method responsible for changing the title of the resource inside the Solid POD. In that case it simply means changing the absolute path to resource, where the name is the last element in the path. This function is described in detail on Section 4.2.2. The operation involves invocation of several simpler methods from the *FileManager* abstraction.
- *updateResource()*, a generic function executing an HTTP PUT request for a particular resource.
- *createOrUpdateResource()*, a method used in cases when a resource needs to be updated even if a different resource exists under that path. If a nothing exists under supplied path, resource will be created, if an object exists under particular path then it will be removed first.
- *resourceExists()*, a method that executes the HTTP GET call to check if anything exits under specified path. The response is a simple boolean value.
- *getFolder()*, a method that parses the contents of a particular folder and returns it as an array of underlying files and folders wrapped into a simple configuration abstraction. This is mainly used a part of any recursive operations involving folders.
- *deleteFolderRecursively()*, a method used to execute recursive deletion of a particular folder. This is demonstrated in detail on Figure 4.7 under a conditional block that is executed when resource is a folder.

- *createAccessControlStatement()*, *createAccessControlList*, *updateACL()*, *createACL()*, following methods can be found described in Subsection 5.1.4, as they relate to interacting with ACL resources.

It is important to note that the class is not a singleton in contrast with *AuthenticationManager*. This is because all classes are exposed as public and static. The design makes the abstraction generic for many use cases within and outside the bounds of requirements of LPA.

The Solid resource within the LPS package is represented by *ResourceConfiguration* abstraction that is a basic required input for most of the operations in *FileManager* that involve execution of HTTP requests. The proceeding Section 5.1.3 will provide more details on implementation of this class and the relation to *FileManager*.

Resource Configuration

One of the challenges when LPA frontend initially contained the code for Solid interaction inside was the lack of any abstraction representing a particular Solid resource. There were several refactoring attempts to add basic ES6 classes, but maintaining the codebase was not trivial at all. Therefore, after introducing the LPS package, the implementation of abstractions to represent the Solid resources was one of the first challenges. The Figure 5.5 consist of two main entities:

1. *ResourceConfig*, a main class representing *ResourceConfiguration* abstraction.
2. *SolidResource*, an interface required for each object representing the Solid resource to be conforming to.

```
enum SolidResourceType {
  Folder = '<http://www.w3.org/ns/ldp#BasicContainer>;
  ↪  rel="type"',
  File = '<http://www.w3.org/ns/ldp#Resource>; rel="type"'
}
```

Listing 5: Definition of *SolidResourceType* enumerator.

The *SolidResource* interface consist of the following:

- *type*, an enumerator property represented on Listing 5. Type is either a *Folder* or a *File*.
- *path*, a `string` property representing a full absolute path excluding the filename itself. The reason why filename is excluded is to simplify the interactions within *FileManager* class and provide more flexibility when operating with resources.

- *title*, a `string` property representing the tile of the resource. The extension of the resource is not included.
- *contentType*, an optional `string` property representing the content type header to be passed when constructing and HTTP request to manipulate this resource. If value is not provided, a TTL extension is used by default.
- *body*, an optional `string` property holding a content of the resource. The property is marked optional because there are cases when an empty folder resource need to be created, from an LPA developer point, he does not need to provide any content to create an empty folder resource.
- *isPublic*, an optional *boolean* indicating whether the file is controlled only by the creator, owner or can have public read access. This is closely related to *AccessControlManager* abstraction described in Subsection 4.2.3. The proceeding section will describe the implementation of that abstraction in detail.

The fact that *SolidResource* is represented as the interface allows the objects conforming to eat to be constructed effortlessly and straightforwardly, similar to simply creating a `dictionary` object. The *ResourceConfig*, on the other hand, wraps the object by providing additional information and functionality on top of the original resource object. The *ResourceConfig* class consist of the following:

- *webID*, is a `string` property. The value should be assigned to the creator or owner of the resource. Within LPA frontend, this value is usually holding the *WebID* of the authenticated platform user.
- *resource*, is an `object` property conforming to *SolidResource* interface.
- *fullPath*, a method returning a concatenated `string`, representing an absolute path to a resource.
- *fullPathWithAppendix()*, a method returning a concatenated `string`, representing an absolute path to a resource with a `'/'` symbol. This is required in cases when an operation needs to be performed on a resource a developer wants to be sure that the absolute path will be constructed correctly for the underlying type. The *folder* resource requires the symbol to be appended when dealing with the construction of ACL files. This will be described in more detail in the proceeding section dedicated to *AccessControlManager* abstraction.

The *FileManager* class is extensively used throughout the whole LPA frontend codebase. The examples below demonstrated various examples of invocations of the *FileManager* abstraction from within the LPA frontend codebase. It is important to note that the provided examples are generic enough to be applied to the development of any Solid application supporting the current iteration of `node-solid-server`.

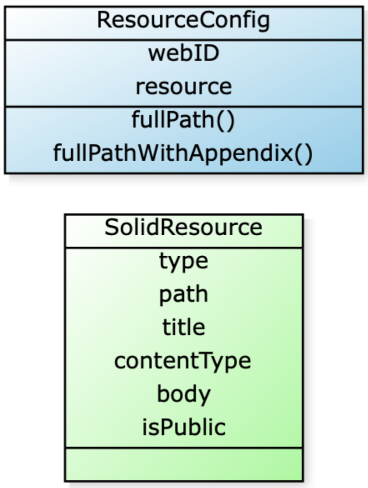


Figure 5.5: A class diagram generated directly from a TypeScript file, demonstrating implemented *ResourceConfig* class and *SolidResource* interface

An example on creating a resource

The example below demonstrated a simple use case on how LPS package can be used to create a Solid resource using the *FileManager* abstraction.

```
const folderConfig: ResourceConfig = new ResourceConfig(
        {
          path: rootResourceConfig.fullPath(),
          title: 'configurations',
          type: SolidResourceType.Folder
        },
        webId
      );

const response = await
↪  StorageFileManager.createResource(folderConfig)
```

Listing 6: An example usage of *FileManager* abstraction to create a folder in Solid pod.

In this particular case, a folder resource *ResourceConfig* class instance is being created named *configurationsFolderConfig*. Afterwards it is being passed to a public static *createResource()* method that creates the resource as described on Figure 4.4. This concludes the description of the implementation of *FileManager* abstraction. The proceeding section is dedicated to an *AccessControlManager* abstraction, following up the first introduction in the Subsection 4.2.3.

5.1.4 Access Control Manager

This subsection provides details on implementation of the *AccessControlManager* abstraction from Subsection 4.2.3 section. Despite being called a separate abstraction, it is in fact implemented as a set of additional public static methods inside the *FileManager* abstraction as seen on Figure 5.4 diagram. Essentially every ACL resource is yet another resource represented as *LDPR* in Solid. However the functional implication of those resource are more specific and require extra information and functionality to operate with them. That is why it is separated conceptually as a different abstraction but is implemented inside the same class called *FileManager*.

Referring back to Figure 5.4, the main methods related to the abstraction are described as follows:

- *updateACL()*, a method constructing the array of RDF triples that is later serialized into a TTL and send as a body in HTTP PUT request to Solid server, replacing the previous resource under that path.
- *createACL()*, a method constructing the array of RDF triples that is later serialized into a TTL and send as a body in HTTP PUT request to Solid server.
- *createAccessControlStatement()*, a method constructing the array of RDF triples expressing access control for a particular WebID.
- *createAccessControlList()*, a method assembling an array of access control statements for multiple requested WebIDs using *createAccessControlStatement()*, and as result serializes the array of statements into a raw TTL `string` using `rdflib`.

The functionality implemented by *AccessControlManager* is not conforming to entire Web Access Control specification, since it was not needed and not required to satisfy the LPA requirements. Therefore, methods described earlier such as *updateACL()* and *createACL()* as seen on Figure 4.9, were based on the default ACL files that `NSS` generates. At the moment of writing this thesis, the default ACL files generated by `NSS v5.2.0` looks as follows:

```
@prefix : <#>.
@prefix n0: <http://xmlns.com/foaf/0.1/>.
@prefix n1: <http://www.w3.org/ns/auth/acl#>.
@prefix test: <./>.
@prefix c: </profile/card#>.

:owner
    n1:accessTo test:;
    n1:agent c:me;
    n1:default test:;
    n1:mode n1:Control, n1:Read, n1:Write.
:public
n1:accessTo test:; n1:agentClass n0:Agent; n1:default test:;
 ↪  n1:mode n1:Read.
```

Listing 7: An example ACL resource in TTL describing access control for a folder

As seen on Listing 7, the example describes access control privileges to a folder named *test*. The first semantic triple contains a subject named `:owner`, and this refers to the WebID of the owner of this Solid POD. The predicates of the Owner subject can be described as follows:

- *accessTo*, the information resource to which the access is being granted. In this case, we are granting access to a folder named *test*
- *agent*, a person or an entity to whom the rights are being given. Since this is an owner of the Solid pod, the reference is given to himself.
- *default*, this is a special predicate that behaves as follows. If the underlying resources have no ACL files specified, they will keep referring to parent resources until it will reach the resource containing the ACL file with *default* predicate. In this case, the statement says that the underlying resources without explicitly set ACL files will have the same access control rights as the *test* folder.
- *mode*, a predicate describing the access control modes. In this case, the object is defined as follows:

  Control, a semantic object describing full read and write access to an ACL of the resource.

  Read, a semantic object, is giving full read access to a resource.

  Write, a semantic object is giving full write access to a resource.

On the other hand, the subject named *public* only has an ability to *Read* the folder and its content. The *public* subject is just a generic simplification for cases when there is no need to set specific WebIDs explicitly. However, using ACL easily allows listing particular users as well, giving a lot of flexibility to have a very sophisticated access control privileges setup per any resource in a Solid POD. As mentioned earlier, the ability to manipulate access control to any resource and own your data is one of the core benefits proposed by Solid project.

Access Control Configuration

Aside from the main *AccessControlManager* abstraction, as demonstrated on Chapter 1, the original *ResourceConfig* class from *FileManager* required to be extended by introducing several subclasses called:

- *AccessControlConfig*, a subclass of *ResourceConfig* that adds a property listing available control modes to a resource for specified WebID, and adds methods to construct the proper absolute path to a resource. The only difference between those methods and methods described in Section 5.1.3 such as *fullPath()* and *fullPathWithAppending()*, is that the hardcoded keyword ACL for files and hardcoded keyword `/.acl` for folder are being appended.

- *AccessControlStatementConfig*, a subclass of *AccessControlConfig*, contains additional references to `rdflib` nodes to simplify construction of ACL triples in *AccessControlManager* abstraction.

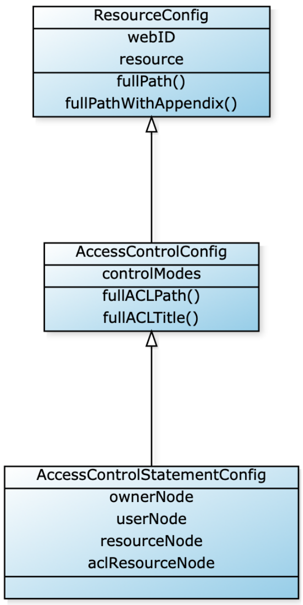


Figure 5.6: A class diagram generated directly from a TypeScript file, demonstrating implemented *AccessControlConfig* and *AccessControlStatementConfig* extending *ResourceConfig*

An example on creating an ACL file for a Solid resource

The example below demonstrated a simple use case on how LPS package can be used to create an ACL file for a Solid resource using the *FileManager* abstraction.

```
const configurationsAclResourceConfig: AccessControlConfig = new
↪  AccessControlConfig(
  {
    ...configurationsFolderConfig.resource,
    isPublic: true
  },
  [AccessControlNamespace.Read, AccessControlNamespace.Write],
  webId
);

const response = await StorageFileManager.updateACL(
  configurationsAclResourceConfig
);
```

Listing 8: An example ES6 code on creating ACL files for Solid resource in LPA frontend

In this case, a folder resource is expressed as a *ResourceConfig* class instance under *configurationsFolderConfig* constant and *configurationsAclResourceConfig* constant is created based on it to express and ACL file. Using the ES6 `spread` operator we populate the description of the folder resource as follows '`...configurationsFolderConfig.resource`'. Afterwards, the access control modes are supplied in an array giving the ability to *Read* and *Write* to that resource to anyone by default. In the last step the specify the webId of the owner of the resource and the ACL file to be created and supply the constructed *AccessControlConfig* into *updateACL()* method that is demonstrated in detail on this sequence Figure 4.9.

To sum up, the section provided an overview of three main abstractions inside LPS package. The *AuthenticationManager* used as a main class to deal with WebID based authentications for LPA platform. The *FileManager*, responsible for managing all HTTP requests to manipulate resources in a Solid resource. *AccessControlManager* abstraction that significantly simplified the interactions with Solid inside LPA frontend significantly and gave an ability to implement more advanced access control related features to configure the published LPA visualizers. This will be demonstrated in detail in Section 5.3. The consequitive section will describe the LPS Ontology originally described in Figure 4.9, its implementation and hosting.

5.2 Hosting Storage Ontology

This section is going to continue the details on implementation of the LPS ontology, firstly described in Section 4.3. As mentioned earlier, the intent to design the ontology was to utilize the benefits of Solid better and the ways in which every entity is represented as an RDF resource.

The ontology itself was implemented using a set of open-source tools that will be described in the first part of this section.

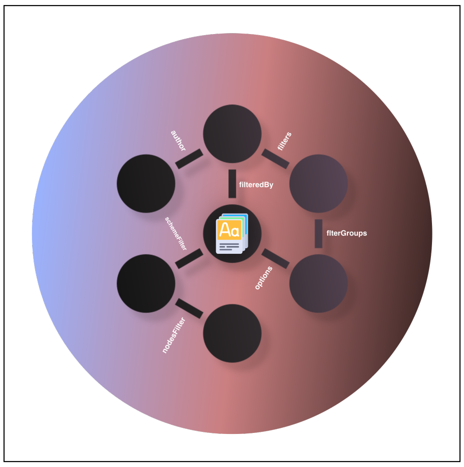


Figure 5.7: Official LPS Ontology logo designed by author.

5.2.1 Preliminaries

Two open-source frameworks were used to implement and publish the LPS ontology:

- `Protégé` is an open-source ontology editing framework developed at Stanford University and written in Java programming language [22]. It provides an intuitive graphical user interface for defining and developing ontologies. The version of the software used at the moment of implementing the ontology is `v5.5.0`.

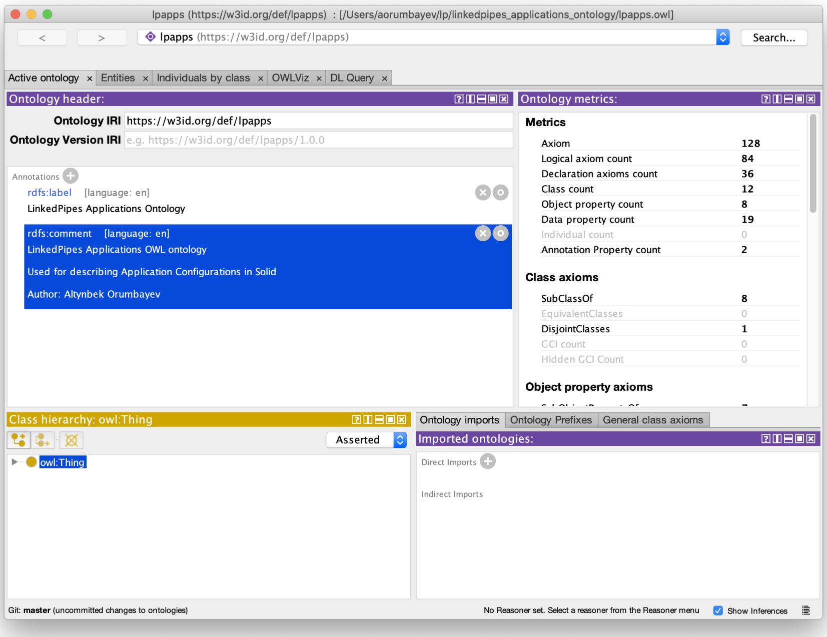
- `Ontoology`, is an open-source software solution for collaborative development of ontologies on GitHub [23]. However, in the scope of implementing LPS ontology, it was mainly chosen for its additional features, such as an ability to publish ontologies under permanent `w3id.org` URL, an ability to host the ontology as a static HTML webpage under GitHub Pages.

5.2.2 Using `Protégé`

After defining the main entities and designing the hierarchical structure of the ontology in Section 4.3, the ontology was implemented using `Protégé` ontology editor. The editor allows defining the ontology entities by specifying them under:

- *Classes*, tab represents the *asserted* and *inferred* class hierarchies as a tree, where each node represents a particular class. The *asserted* class view is a default and primary navigation device for browsing class hierarchies in *Protégé*, the *inferred* classes view on the other hand differs from *asserted* by requiring a reasoner to be setup to render the complete hierarchy view. If no reasoner is provided the *inferred* classes view will be empty.

- *Object properties* tab, represents the graphical user interface to create and define relations between instances of the classes. The view under that tab is similar to the *Classes* tab, the properties are represented as a tree with nodes identifying individual object property. For instance, each *VisualizerConfiguration* can be filtered by a *FilteredConfiguration*, this relation can be expressed by defining an object property calls *filteredBy*.


lpapps (https://w3id.org/def/lpapps) : [/Users/aorumbayev/lp/linkedpipes_applications_ontology/lpapps.owl]

lpapps (https://w3id.org/def/lpapps)

Ontology header:

Ontology IRI https://w3id.org/def/lpapps

Ontology Version IRI e.g. https://w3id.org/def/lpapps/1.0.0

Annotations

rdfs:label [language: en]
LinkedPipes Applications Ontology

rdfs:comment [language: en]
LinkedPipes Applications OWL ontology

Used for describing Application Configurations in Solid

Author: Altynbek Orumbayev

Ontology metrics:

Metrics

Axiom	128
Logical axiom count	84
Declaration axioms count	36
Class count	12
Object property count	8
Data property count	19
Individual count	0
Annotation Property count	2

Class axioms

SubClassOf	8
EquivalentClasses	0
DisjointClasses	1
GCI count	0
Hidden GCI Count	0

Object property axioms

Class hierarchy: owl:Thing

Imported ontologies:

Direct Imports

Indirect Imports

Git: **master** (uncommitted changes to ontologies)

No Reasoner set. Select a reasoner from the Reasoner menu

Figure 5.8: Example UI of `Protégé` ontology editor at *Active Ontology* tab.

- *Data properties* tab, represents the graphical user to create and define relations between instances of classes and RDF literals or datatypes. For instance, a *VisualizerConfiguration* class has a *backgroundColor* data property which is a *String* datatype.

The resulting ontology created in `Protégé` was represented as an OWL, which is a format for Web Ontology Language described earlier in Subsection 4.3.1. The following subsection will explain the process of publishing the ontology using the *Ontoology* software.

5.2.3 Using *Ontoology*

The *Ontoology* itself is an open-source project that can be instantiated and configured manually. It is, however, also available as a standalone running instance [6]. The publishing of the LPS Ontology was done using the available instance instead of manually configuring one, this significantly reduced the development time needed to publish and host the vocabulary to be used in LPA frontend.

The initial configuration or startup guide to *Ontoology* consist of the following steps:

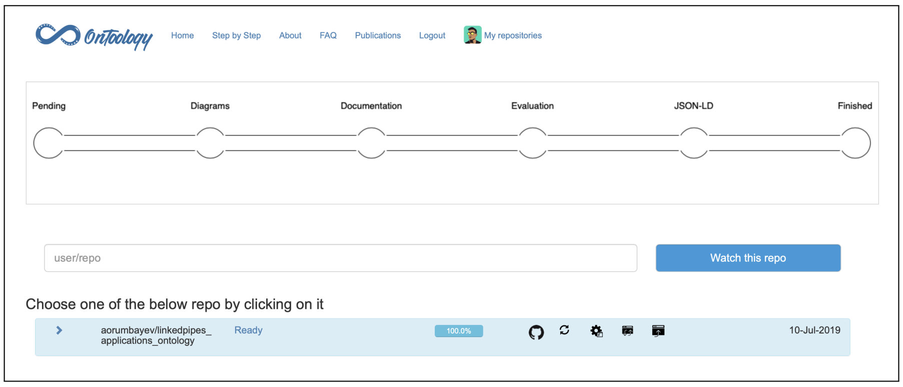
1. *Setup a GitHub repository*, an exported OWL file from `Protégé` was hosted on a regular public GitHub repository [7] and placed at the root.

2. *Add repository to Ontoology*, add the repository by name into Ontology, and press *Watch this repo* button. This is pretty much the last step involving

[6]`http://ontoology.linkeddata.es`
[7]`https://github.com/aorumbayev/linkedpipes_applications_ontology`

any manual work. The following steps are all invoked in an automated fashion, demonstrating the full list of benefits provided by this software.

3. *Merge initial PR from Ontoology*, once the repository is connected, an automated PR will be opened that will contain a following list of elements:
    (a) Diagrams, a folder with pre-generated class hierarchy visualizations of the ontology.
    (b) Documentation, an automatically generated ontology documentation hosted on GitHub Pages on the same repository, and available at *w3id.org* permanent Uniform Resource Locator (URL) redirecting users to GitHub Pages.
    (c) Evaluation report, *Ontoology* generates a report that checks various aspects of the implementation of the ontology, providing hints when, for instance, some parts in the description of an entity is missing.
    (d) JSON-LD, an original OWL, the file is automatically converted into a JSON-LD file.



Choose one of the below repo by clicking on it

aorumbayev/linkedpipes_applications_ontology Ready 100.0% 10-Jul-2019

Figure 5.9: UI of Ontoology displaying the processed repository with LPS Ontology.

To sum up, the LPS ontology was implemented using two open-source frameworks called `Protégé` and `Ontoology`. The first tool was used as an editor to implement the ontology designed in Section 4.3. The second tool was used to publish and host applications. The detailed documentation will be mentioned in Chapter 8 and is also available online as hosted documentation.

5.3 Storage Frontend

The following section will continue the Section 4.4 by providing the details on implementation of individual components interacting with Solid POD.

5.3.1 Preliminaries

To simplify the understanding of this section, it is important to recap the conventions and specifics of LPA frontend implementation by providing some extended details, as they were strictly taken into consideration while developing LPS components inside the LPA frontend codebase.

As mentioned earlier in Subsection 1.3.2, the frontend provides a way for the user to interact with the LPA. As demonstrated on the Figure 5.10, the frontend uses Redux and React as a primary framework for building the components and managing states.

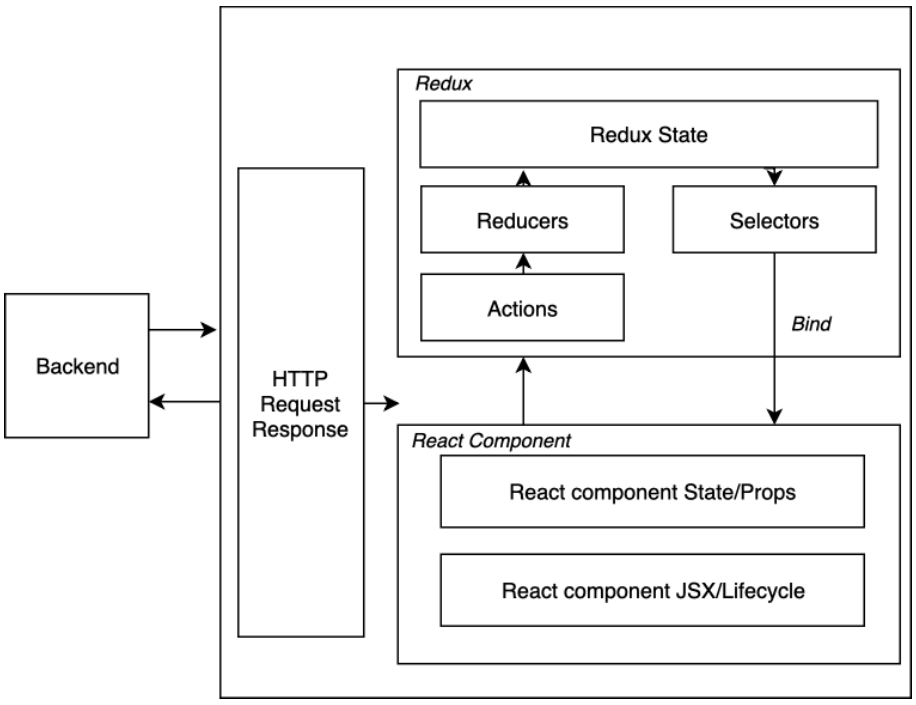


Figure 5.10: General architecture of Redux store and React components within *frontend*

The main entities displayed on 5.10 can be described as follows:

- *React Component*, as described in Section 1.3.2, it represents a JavaScript class or function that accepts optional inputs and returns a React element that describes how a section of the User Interface should appear.
- *Redux* as described in Section 1.3.2, it represents an internal state management framework for frontend web frameworks like React.

Frontend code structure

The implementation of the LPA frontend is located at the `src/frontend` path at official LPA repository [8] and structured as follows:

- *constants* - contains all constants used throughout the frontend component implementation.
- *utils* - contains various handy utility classes, variables, and methods.

[8] `https://github.com/linkedpipes/applications`

- *ducks* - contains all Redux reducers, actions and selectors. The title *duck* comes from a convention proposed by Erik Rasmussen in [24], it suggests a way to bundle Redux entities into folders that are easier to manage and maintain as they contain reducers, actions, and selectors.
- *layouts* - contains a global setup for various components as well as the specification for the MaterialDesign theme used in the project.
- *components* - contains all components with JSX file type, visualizers and various UI elements implementations not related to interactions with Solid. The JSX is a syntactic extension for JavaScript that adds the ability to combine ES6 syntax with HTML tags.
- *containers* - contains complex layouts for specific web-pages of the web app not related to interactions with Solid. Each folder is named after the individual webpage.
- *storage* - contains all contributions and implementations made within the scope of LPS project. It has a substructure mimicking the global folder structure to improve code maintainability and signify the difference between implementations of LPA and LPS.
- *configuration files and entry points*, contains a set of configuration files at the root, such as linter configurations, the file with global Redux store and a file called *AppRouter*, which is the main controller of the whole frontend since it manages the user sessions and establishes the socket connections with LPA backend.

5.3.2 Storage folder structure

As mentioned in Section 5.3.1, let us describe the internals of *storage* folder inside LPA frontend codebase.

- *utils* - contains various handy utility classes, variables, and methods for storage specific React Components.
- *ducks* - contains all Redux reducers, actions and selectors for storage specific React Components.
- *models* - contains classes representing *VisualizerConfiguration* and other LPA specific classes that did not have to be split into the LPS package.
- *components* - contains all components, visualizers and various UI elements implementations not related to interactions with Solid.
- *containers* - contains complex layouts for storage specific web-pages. Each folder is named after individual webpage.
- *StorageBackend* - represents a class wrapping various utilities on top of basic LPS storage abstractions. The logic related to fetching the LPA configurations is implemented in this file.

- *StorageToolbox* - represents a class wrapping the *StorageBackend* utilities into simple methods to be invoked directly from internal methods of Storage React Components. In other words, it serves as a middleware layer between React Components and low level and generic functionality of LPS package abstractions.

- *StorageSparqlClient* - a class containing a single generic method that utilises the *fetch()* function from *AuthenticationManager* to submit `SPARQL` requests modifying certain parameters of LPA configuration RDF files.

5.3.3 Authentication View

This section continues the architecture and mocks provided in Section 4.4, and describes the implementation of the React Component in detail. To comply with conventions of LPA frontend, the webpage is represented by two separate React Components, one of which is stateful and the other is stateless and manages the views only. The implementation is located at `src/frontend/containers-/AuthenticationPage` path on LPA repository. The folder is named after the webpage and consist of:

- *AuthenticationContainer*, a file containing the stateful React Component. In other words, it manages local views specific states as well as global states related to Redux.

- *AuthenticationComponent*, a file containing implementation of stateless React Component. In other words, it is only responsible for passing the React *props* and rendering simple views.

- *SolidProviderComponent*, a file located in a sub-folder called *children*. The component renders the drop-down menu that allows users to pick specific provider from a list of supported Solid providers.

```
const mapStateToProps = state => {
  return {
    webId: state.user.webId
  };
};

export const AuthorizationContainer =
↪  connect(mapStateToProps)(Authorization);
```

Listing 9: An example of mapping Redux state to a props of React Container

Following that pattern, the *render()* function of *AuthenticationContainer* expects an instance of *AuthenticationComponent* while passing both internal and Redux state as props to it. The global structure of all Redux states in LPA is too specific to be described within the scope of the LPS project. However, examples of states relevant to Authentication View are going to be provided. For instance, the code example at Listing 9 demonstrates a typical example of an operation that maps the Redux states to *props* of a React Component, in this case, we are mapping the global *webID* value that is used by React Router to identify whether user is authenticated and can render the dashboard or should be redirected back to the screen. As it was mentioned earlier in Section 5.3.1 and Subsection 5.1.2, the *AppRouter* is one of the main entry points to the frontend web app, and it directly utilizes both *AuthenticationManager* abstraction from Authentication React component. An example code on Listing 10 demonstrates a code snippet from *AppRouter* that is invoked every time the user session is being authenticated, or user is logging out.

```
AuthenticationManager.trackSession(session => {
  if (session) {
     handleSetUserWebId(session.webId);

     self.startSocketListeners();
     ...
  }
})
```

Listing 10: An example of *trackSession()* callback listening for changes in authentication state

The User Interface strictly follows initial mock design demonstrated at Figure 4.13. User has an ability to perform authentication either by choosing an individual Solid provider or use a supply his WebID directly if his Solid provider is not available in the list of default providers. At the moment of writing this part, the project provides support for `inrupt.net` [9], `solid.community` [10] and self-hosted provider made specifically for LinkedPipes available at *lpapps.co* [11]. After performing the authentication use is redirected into the Dashboard webpage of LPA frontend.

Overview of states and methods

As general details on the structure of the implementation are defined now, let us dive deeper into the description of main states and functions of Authentication View components. As *AuthenticationComponent* is a stateless React component, it is sufficient to provide definitions for states and methods in *AuthenticationContainer* as this represents the main functionality of the whole webpage. The states can be described as follows:

[9]`https://inrupt.net`
[10]`https://solid.community`
[11]`https://lpapps.co:8443`

- *webIdFieldValue*, a `string` value holding input at the *textfield* for supplying WebID value.
- *withWebIdStatus*, a `boolean` value representing the current input state and whether is chosen to use available providers or entered his WebID manually.
- *session*, an `object` representing current session returned from *AuthenticationManager* abstraction.
- *providerTitle*, a `string` value holding the title of selected provider. This is needed to be supplied into *login()* method of *AuthenticationManager* abstraction that invokes the `solid-auth-client` library.

The set of methods inside the *AuthenticationComponent* class can be described as follows:

- *handleProviderChange()* a methods that is passed as a `prop` to *AuthenticationComponent*, and invoked every time user selects or update his selection of default providers.
- *handleSignIn* a method that is invoked when is pressing the *Authenticate* button, this redirects a call to *AuthenticationManager* abstraction that initiates the sequence flow described at Figure 4.3.
- *handleWebIdFieldChange* a method invoked when user inputs or modifies the input at the *textfield* used for WebID.
- *isWebIdValid* a method that uses *regex* to validate whether provided WebID is in valid format.
- *onSetWithWebId* a method invoked when the *withWebIdStatus* status is being changed.

The final rendered UI of this implementation will be provided in the of this chapter under Section 5.4, where a detailed overview of a final implementation satisfying each of the original LPA requirements are provided.

5.3.4 Storage Dashboard

The Storage Dashboard webpage described at Subsection 4.4.3 was a challenging component to implement. To simplify the understanding of the structure in which components and containers are invoked, the Figure 5.11 is provided. The components demonstrated on diagram follow the usual pattern of splitting React Components into stateful and stateless and can be described as follows:

- *StoragePageController* is a root React component that wraps both individual and shared application views. It stores states representing the tab index and depending on the selected index it switches the components to be rendered in *render()* function.
- *StorageAppsBrowser**, both container and component represent a view that fetches and displays LPA configurations. The stateless part relies on the *Grid* layout system from `MaterialUI` framework.

- *StorageSharedAppsBrowser**, both container and component represent a view that fetches and displays LPA configurations that were shared with the User. The stateless part relies on the *Grid* layout system from `MaterialUI` framework.

- *StorageAppsBrowserCardComponent* is a child component that represents a reusable card representing an individual LPA configuration in a grid of cards.

The user interface is inspired by the mock at Figure 4.14. However, it is more refined and optimized to fit the general style of LPA frontend. Similar to the Authentication view, the final rendered UI of this implementation will be provided at the end of this chapter under Section 5.4.

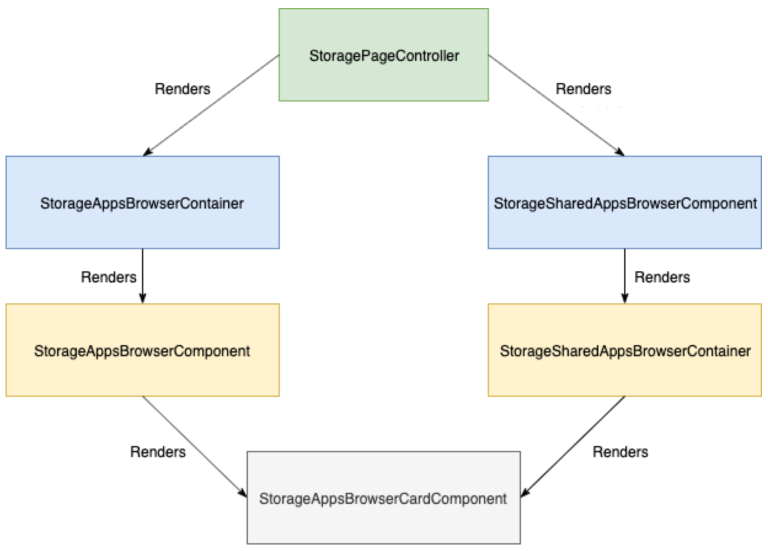

StoragePageController
Renders
Renders
StorageAppsBrowserContainer
StorageSharedAppsBrowserComponent
Renders
Renders
StorageAppsBrowserComponent
StorageSharedAppsBrowserContainer
Renders
Renders
StorageAppsBrowserCardComponent

Figure 5.11: Diagram representing the chain of *render()* method invocations among components representing Storage Dashboard.

Overview of main functionality

In contrast with Authentication View, the structure of sub-components is rather complex to describe in detail. Therefore a set of primary states and methods responsible for the core functionality of the webpage will be described instead.

In general both *StorageAppsBrowser** and *StorageSharedAppsBrowser** components and containers share similar states and functions the only difference is the way stored LPA configurations are fetched. The fetching itself is done by invoking the corresponding methods called *getAppConfigurationsMetadata()* and *getSharedApplicationsMetadata()* methods from *StorageToolbox* class. In general, the *StorageToolbox* invokes lower-level wrappers in *StorageBackend* that utilize the `rdflib` library from LPS package and converts the RDF resource representing the LPA configuration into a JavaScript object that is processed by *StorageApps-BrowserCardComponent* view.

5.3.5 Storage Control Panel

The Storage Control Panel is rather a simple set of React Components that heavily rely on *FileManager* abstraction from LPS package. As initially described in Subsection 4.4.4, the functionality should provide user options to perform basic operations with the root folder for all LPA configurations in his Solid POD. Those operations include changing the title of the root folder, moving within the pod, or copying within the pod. The implementation consist of the following classes:

- *StorageAccessControlDialog* is a stateful React Component based on *Dialog* Component from *Material UI* framework. Due to simplicity of the elements to be rendered, it did not require splitting into stateful and stateless components.

- *SettingsPageComponent* is a stateless React Component that renders the simple label with the current path to the root storage folder in *Solid* and a simple button to change it by invoking the *StorageAccessControlDialog*.

- *SettingsPageContainer* is a stateful React Component that serves to connect the *StorageAccessControlDialog* and *SettingsPageComponent*.

Overview of main functionality

In general, the main functionality is located in the *StorageAccessControlDialog* class, it strictly follows the mock at Figure 4.15 and can be described as follows:

- *handleFolderConfirm()* a method invoked when user clicks on *Update* button. The component then invokes the *StorageToolbox* class that communicates with *FileManager* abstraction in LPS package. For more detailed description of how LPS package performs the CRUD operations on resources in Solid PODs refer back to Chapter 4.

- *handleFolderCopy()* a method invoked when user clicks on *Copy* button. The rest is similar to *handleFolderConfirm()* where *StorageToolbox* is invoked and it communicates with *FileManager* abstraction in LPS package.

- *handleFolderMove()* a method invoked when user clicks on *Move* button. The proceeding invocation chain is identical to methods before.

The final rendered UI of this implementation will be provided in Section 5.4, demonstrating the *StorageAccessControlDialog*, *SettingsPageComponent* and *SettingsPageContainer*.

5.4 Implemented functional requirements

This section will iterate over the requirements from LPA stated in Chapter 3, and demonstrate how each of the functional requirement was implemented by classes and abstractions described in previous sections of this chapter. For requirements that directly involved the implementation of React components, detailed renders of final User Interfaces will be provided. It is important to note that this section offers the implementation overview of all stated functional requirements defined in Section 3.1.

5.4.1 User Authentication

To recap the User Authentication functional requirement, the user of the platform should be able to register an account in the application, log in, and log out. To fulfill the requirement, the Authentication View component described at Chapter 1 was implemented. The final render on Figure 5.12 provides an intuitive User Interface that allows authentication by selecting a default Solid provider, as demonstrated on Figure 5.13 or by specifying the WebID for authentication directly. The *Learn more about WebID, and SOLID* button serves as a guide for users new to Solid, clicking the button will get them redirected to the official `inrupt` website.

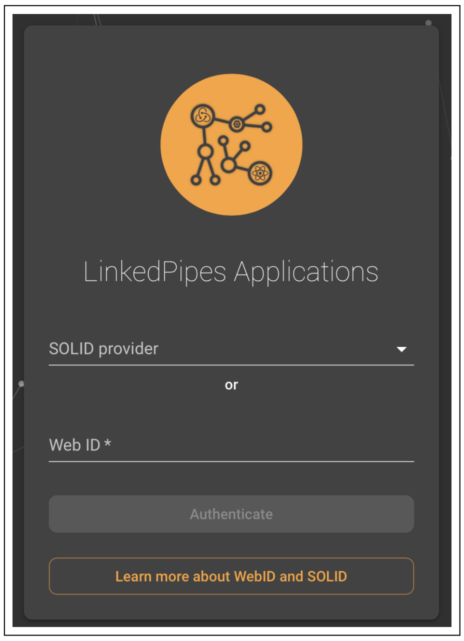


Figure 5.12: The final render of an Authentication View webpage

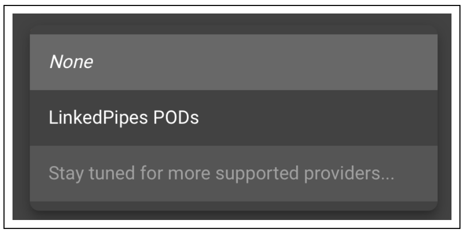


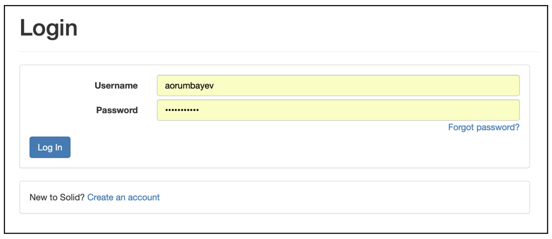
Figure 5.13: And example of the *providers* dropdown in expanded state.

After user provides the authentication inputs and clicks on *Authenticate* button, the default Solid server authentication popup will appear as demonstrated on Figure 5.14 and Figure 5.15. Depending on whether user has a WebID and a Solid POD or if user selected the authentication option using default providers he will be given an option to either *Login* or *Register* on specified Solid server.

The *logout* functionality is available under several components in LPA. Those can be described as follows:

- *User Profile Page*, this page is located under settings tab in LPA platform. The intent is to provide the information about the current user name and his WebID under which the authentication was performed. It also provides buttons to either *Reset Password* or *Logout* from the platform.

- *Logout toolbar item*, this is a toolbar element always available in the top right corner of the platform. Clicking on the button invokes the *logout*, and the user is redirected back to the Authentication View webpage.



Login

Username aorumbayev

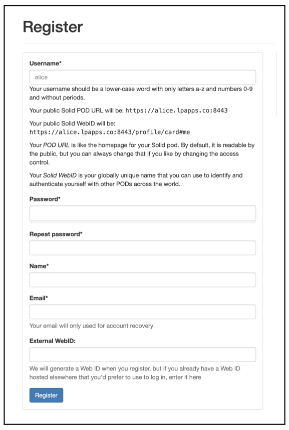
Password ••••••••••••

Forgot password?

Log In

New to Solid? Create an account

Figure 5.14: Example of a login popup provided by Solid server.



Register

Username*

alice

Your username should be a lower-case word with only letters a-z and numbers 0-9 and without periods.

Your public Solid POD URL will be: `https://alice.lpapps.co:8443`

Your public Solid WebID will be: `https://alice.lpapps.co:8443/profile/card#me`

Your *POD URL* is like the homepage for your Solid pod. By default, it is readable by the public, but you can always change that if you like by changing the access control.

Your *Solid WebID* is your globally unique name that you can use to identify and authenticate yourself with other PODs across the world.

Password*

Repeat password*

Name*

Email*

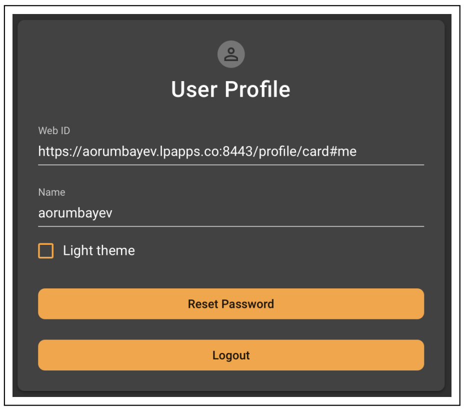
Your email will only used for account recovery

External WebID:

We will generate a Web ID when you register, but if you already have a Web ID hosted elsewhere that you'd prefer to use to log in, enter it here

Register

Figure 5.15: Example of a register popup provided by Solid server



User Profile

Web ID
https://aorumbayev.lpapps.co:8443/profile/card#me

Name
aorumbayev

☐ Light theme

Reset Password

Logout

Figure 5.16: The User Profile webpage with options to reset or logout from Solid provider.

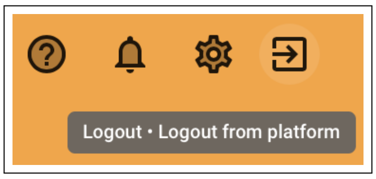


Figure 5.17: The global control toolbar of LPA platform. The logout button allows to quickly logout from authenticated Solid provider.

5.4.2 Create, Store and Publish Application

The Create and Publish Application requirements are one of the core requirements that are both applicable to LPS and LPA frontend itself. The separation of code contributions to both LPA and LPS was a challenging task. From the standpoint of LPS, an application is created when its configuration is stored as an RDF resource inside Solid POD. Having a stored configuration also implies that the application described by this configuration is published because the URI to configuration inside the POD is publicly available by default. In other words, storing creation and publishing of the application are closely related requirements that are easier to cover at once. The section will also focus on the parts of these functional requirements that directly related to interactions with Solid server. Additionally, It is important to mention that there is no actual application being stored in the Solid pod, the app itself is a React Component that is re-rendered from scratch based on the LPA configuration that is loaded from the storage.

The LPA platform has a process called *Application data preparation workflow*. It consists of four steps during which a user of the platform provides data sources to visualize, and if successful, the LPA redirects him to a *Create Application* webpage. At this point, the platform assembles a visualizer configuration object in memory until the user provides a title for his application and decides to publish it. The diagram on Chapter 1 demonstrates the user interface component on *Create Application* that consist of a text field and two buttons:

- *Publish* a button that invokes a chain of methods that parse the prepared visualizer object into TTL and uploads the file into his Solid POD.
- *Embed* a button invoking the same process of storing and publishing an application but additionally displays a dialog popup to quickly generate the HTML `iframe` to incorporate the visualizer into a webpage.

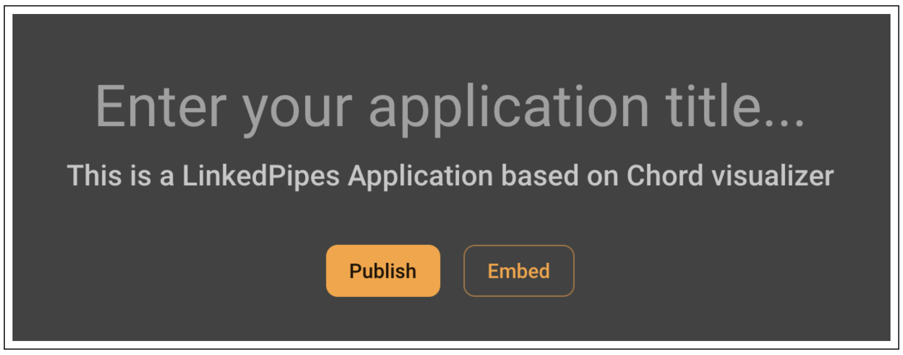

Enter your application title...
This is a LinkedPipes Application based on Chord visualizer
Publish
Embed

Figure 5.18: A part of *Create Application* page that invokes creation and publishing of an application.

A method source code demonstrated on Listing 11 is invoked by *CreateVisualizerPage* and its underlying sub components. As the first step, it simply checks whether the `webId` is provided. Afterwards, the `applicationConfigurationObject` is assembled which is a JavaScript *object* with fields corresponding to LPS Ontology. When passed into *StorageBackend* the object is parsed into an `rdflib` graph, serialized into TTL and uploaded into Solid using LPS package.

```
async saveAppToSolid(
  applicationConfiguration,
  filtersConfiguration,
  webId,
  appFolder
): Promise<ApplicationMetadata> {
  if (!webId) {
    Log.error('No webID available', 'StorageToolbox');
    return;
  }

  const applicationConfigurationObject =
   ↪  ApplicationConfiguration.fromRawParameters(
    applicationConfiguration,
    filtersConfiguration,
    webId
  );

  return StorageBackend.uploadApplicationConfiguration(
    applicationConfigurationObject,
    appFolder,
    webId
  );
}
```

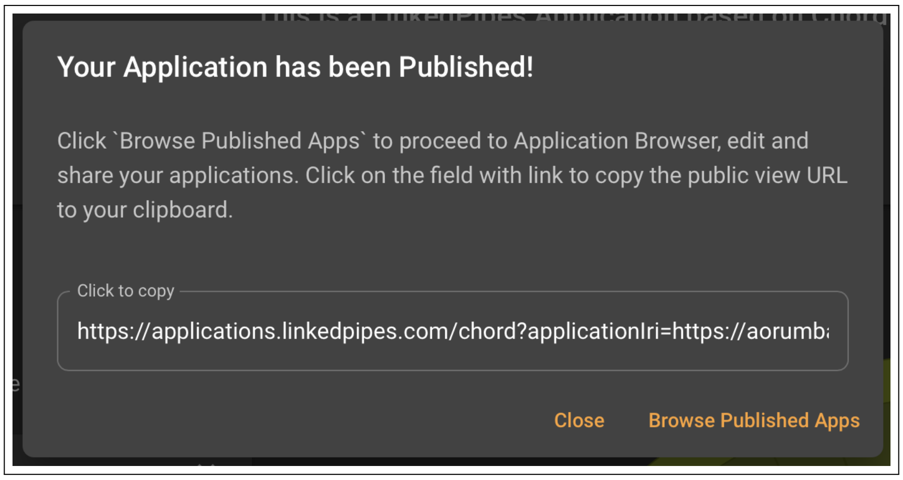
Listing 11: A method from *StorageToolbox* class inLPA frontend, that assembles configuration object and saves it to Solid

The resulting LPA configuration uploaded to solid with the method from Listing 11 is demonstrated on Listing 12. The fields with unique identifiers were replaced by sample text to improve readability of the example.

```
@prefix : <#>.
@prefix lp: <https://w3id.org/def/lpapps#>.
@prefix c: </profile/card#>.

<>
    a lp:VisualizerConfiguration;
    lp:applicationData "undefined";
    lp:author c:me;
    lp:backgroundColor "#106368";
    lp:configurationId "sample id";
    lp:endpoint "chord";
    lp:etlExecutionIri "sample etl iri";
    lp:filteredBy
            [
                a lp:FilterConfiguration;
                lp:enabled "true";
                lp:filterGroups [];
                lp:visible "true"
            ];
    lp:graphIri "sample graph iri";
    lp:published "2019-12-05T13:11:37.788Z";
    lp:title "My cool visualizer";
    lp:visualizerType "CHORD".
```

Listing 12: An example of stored LPA configuration for *CHORD* visualizer in TTL

As the last step of creating and publishing the LPA application, the popup at Figure 5.19 is presented, indicating the successful publishing of the application.


Your Application has been Published!

Click `Browse Published Apps` to proceed to Application Browser, edit and share your applications. Click on the field with link to copy the public view URL to your clipboard.

Click to copy

https://applications.linkedpipes.com/chord?applicationIri=https://aorumba

Figure 5.19: A popup presented after configuration is stored and a published URL is ready to be shared.

This concludes the demonstration of components and functionality of both

LPA and LPS package that implement the functional requirements on storing, publishing and creating LPA configurations.

5.4.3 Configuring Application

The ability to configure the published applications had a clear and straightforward set of terms to implement. After publishing an application LPA users to *rename*, *delete* the configuration as well as configure and modify the filters available for the visualiser.

Renaming application

Renaming the published application is performed by executing a simple `SPARQL` query. The *fetch()* method available in *AuthenticationManager* abstraction from LPS package is used to send the constructed request.

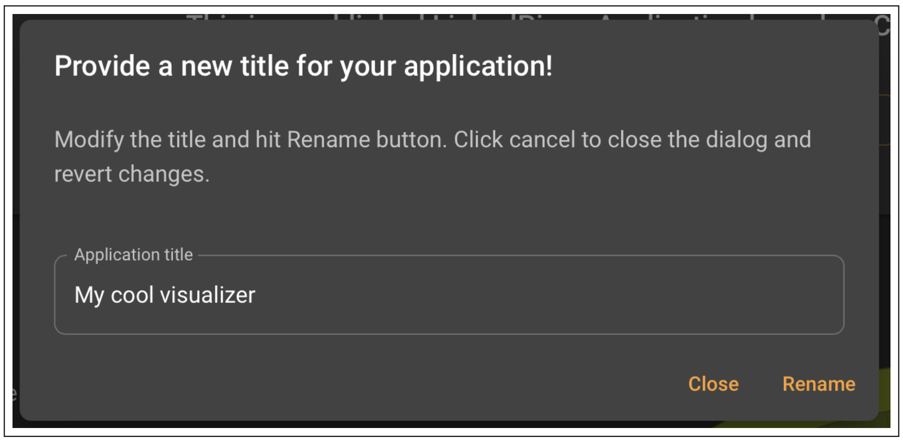
```
const sparqlQuery = `
        @prefix lpa: <https://w3id.org/def/lpapps#> .

        DELETE
        { ?configuration lpa:title ?titleValue . }
        INSERT
        { ?configuration lpa:title "${newTitle}" .}
        WHERE
        { ?configuration lpa:title ?titleValue . }
`;
```

Listing 13: An example of `SPARQL` query to update the application title in configuration stored in Solid.

Example on Listing 13 demonstrate the `SPARQL` query used for renaming the title of LPA configuration. The source code on Listing 14 demonstrate how the query is constructed and submitted to Solid using a class called *StorageSparqlClient*.

```
patchFileWithQuery = async (url, query) => {
  try {
    await StorageAuthenticationManager.fetch(url, {
      method: 'PATCH',
      body: query,
      headers: {
        'Content-Type': 'application/sparql-update'
      }
    });
    return true;
  } catch (error) {
    if (error instanceof Response && error.status === 404)
     ↪  return false;
    throw error;
  }
};
```

Listing 14: The *patchFileWithQuery* method in *StorageSparqlClient* class is used for executing the PATCH requests to Solid servers.


Provide a new title for your application!

Modify the title and hit Rename button. Click cancel to close the dialog and revert changes.

Application title: My cool visualizer

Figure 5.20: A popup presented after user pressed *Rename* button.

On the frontend side, the rename is invoked by pressing the *Rename* button from the *Application Control and Setup* webpage. The user is presented with a simple popup dialog demonstrated on Figure 5.20, where he gets an option to choose the new title of his application. This invokes the execution of the `SPARQL` query mentioned earlier.

Deleting application

Deleting an application that was previously published is a trivial task from the standpoint of LPS. When user wants to delete the application he simply invokes the *Delete* button either from individual application card in *Storage Dashboard* webpage or inside the *Application Control and Setup* webpage. Executing the deletion invokes the *StorageToolbox* class that uses the *deleteResource()* function from *FileManager* abstraction in LPS package described in Subsection 5.1.3.

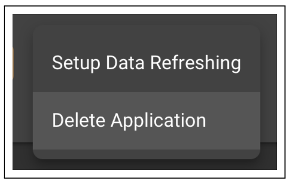


Figure 5.21: Option to invoke the deletion confirmation popup on *Application Control and Setup* web page

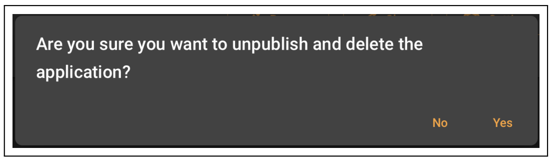

Are you sure you want to unpublish and delete the application?

Figure 5.22: Popup displayed before removing published application configuration.

The popups that appear to user when he presses the *Delete* button on Figure 5.21 is presented on Figure 5.22.

Updating filters

The ability to modify the filters on a published visualizer is another important part of the *Configure Application* functional requirement. The process of updating filters on published applications is invoked when a user modifies available *node* or *scheme* filters. For example, the UI elements on Chapter 1 demonstrate a *CHORD* visualizer containing multiple *node* filters. User has an option to control *state* of the filter, *visibility* to the end user and *interactivity* or the ability for public viewers to interact with the visualizer using the individual filters.

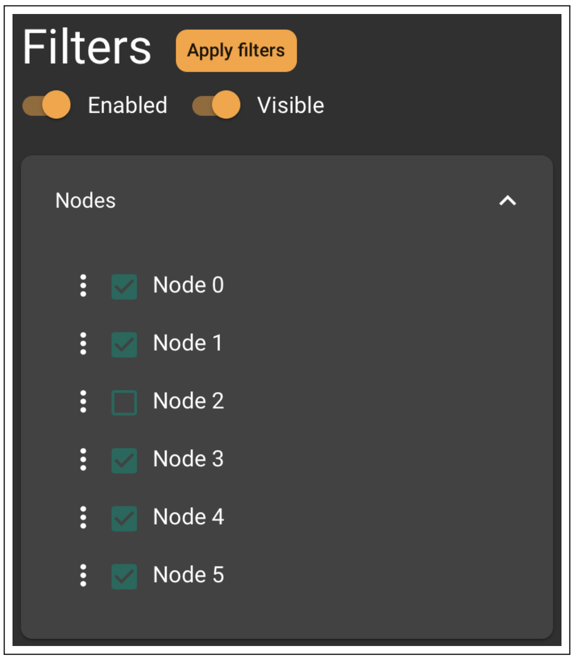


Figure 5.23: A user interface components to interact with filters available for a visualizer.

The implementation of methods processing these operations are done in *StorageToolbox*, *StorageBackend* and *StorageSparqlClient* classes, and similar to *Re-*

naming Application procedure, the core logic simply constructs the required SPARQL queries and executes them using HTTP PATCH request. Example on Listing 15 demonstrates the construction of SPARQL query that processes multiple filter selections in a single PATCH call.

```
let sparqlQuery = '@prefix lpa: <https://w3id.org/def/lpapps#>
↪  .';
const deleteStatements = [];
const insertStatements = [];
const whereStatements = [];

for (const node of nodes) {
  deleteStatements.push(`?selectedOption${cnt} lpa:uri
   ↪  "${node.uri}" .
  ?selectedOption${cnt} lpa:selected ?selected${cnt} .
  ?selectedOption${cnt} lpa:visible ?visible${cnt} .
  ?selectedOption${cnt} lpa:enabled ?enabled${cnt} .`);

  insertStatements.push(`?selectedOption${cnt} lpa:uri
   ↪  "${node.uri}" .
  ?selectedOption${cnt} lpa:selected "${node.selected}" .
  ?selectedOption${cnt} lpa:visible "${node.visible}" .
  ?selectedOption${cnt} lpa:enabled "${node.enabled}" .`);

  whereStatements.push(`?selectedOption${cnt} lpa:uri
   ↪  "${node.uri}" .
  ?selectedOption${cnt} lpa:selected ?selected${cnt} .
  ?selectedOption${cnt} lpa:visible ?visible${cnt} .
  ?selectedOption${cnt} lpa:enabled ?enabled${cnt} . `);
}

sparqlQuery += `
  DELETE { ${deleteStatements.join('\n')} }
  INSERT { ${insertStatements.join('\n')} }
  WHERE { ${whereStatements.join('\n')} }
`;
```

Listing 15: An example of SPARQL query to update the state of multiple *node* filters selected by user.

One of the additional features implemented within the bounds of that requirement is the ability to reflect the applied changes in any field of the application configuration RDF file using socket listeners. In other words, if a user is editing filters on his visualizer and some other user is looking at his visualizer on some webpage where it is published, he will see the changes being applied in real-time. This is implemented using the rdflib.js library that is accessed via

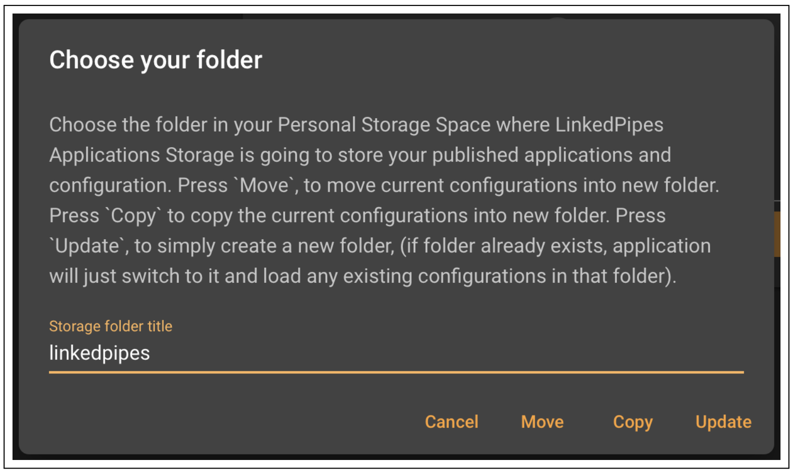
LPS package. The Listing 16 demonstrates a simple utility function available at *StorageBackend* and it is invoked when application configuration is being fetched for the first time.

```
registerChanges(url: string, callbackOnRefresh: Function =
↪  undefined) {
  if (this.alreadyAddedDownstreamListeners.indexOf(url) === -1)
   ↪  {
    const doc = $rdf.sym(url).doc();
    this.updater.addDownstreamChangeListener(doc,
     ↪  callbackOnRefresh);
    this.alreadyAddedDownstreamListeners.push(url);
  }
}
```

Listing 16: Implementation of helper utility that uses feature of `rdflib` to invoke any callback when a change in a specified RDF resource is detected.

5.4.4 Storage Management

The Storage Management functional requirement defines an implementation request to have an ability to *move*, *rename* or *copy* the root folder with all configurations inside the storage. From the perspective of LPS it implies controlling the root LDP Basic Container that stores all configurations.


Choose your folder

Choose the folder in your Personal Storage Space where LinkedPipes Applications Storage is going to store your published applications and configuration. Press `Move`, to move current configurations into new folder. Press `Copy` to copy the current configurations into new folder. Press `Update`, to simply create a new folder, (if folder already exists, application will just switch to it and load any existing configurations in that folder).

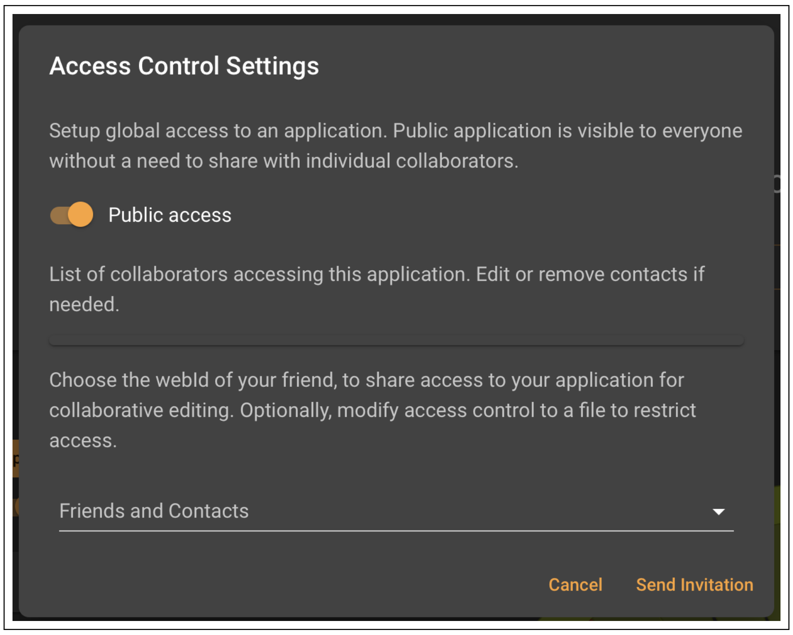
Storage folder title
linkedpipes

Figure 5.24: A final render of Storage Control Panel component.

Since the description in Subsection 5.3.5 already provided all details on implementing this requirement, this section will demonstrate the final renders of implemented components to perform the operations stated in the requirement. The user interface demonstrated on Figure 5.24, performs the functionality as requested in the definition of the requirement. It is invoked via *Settings* webpage under *Application Storage* tab when user clicks on *Change folder* tab.

5.4.5 Visualizer Access Control

The Visualizer, Access Control requirement, defines how the creator of an application can control public access to visualizer or share it with only a specified set of contacts. From the LPS standpoint, this implies controlling the ACL files of individual LPA configurations expressed in RDF.


Access Control Settings

Setup global access to an application. Public application is visible to everyone without a need to share with individual collaborators.

Public access

List of collaborators accessing this application. Edit or remove contacts if needed.

Choose the webId of your friend, to share access to your application for collaborative editing. Optionally, modify access control to a file to restrict access.

Friends and Contacts

Figure 5.25: Access control configuration popup for published application.

The implementation of *AccessControlManager* abstraction in Subsection 5.1.4 is the essential building block that was used to implement a set of React Components to allow modifying access control privileges of published visualizers. The Figure 5.25 demonstrate an final render of a React Component that is presented to user when he clicks on `Access Control` drop down item on *Application Control and Setup* webpage. The component is implemented in *StorageAccessControlDialog* file which is a simple stateful React Component. The main elements of that popup can be described as follows:

- *Public access* switch, this element controls the default public access to a published application. By default, LPA sets the *READ* public access so that anyone can access the published application by default. However, the user has an option to toggle the control element and disable default public visibility.

- *List of collaborators*, this element lists the contacts knows to user's *WebID*. The querying of knows contacts is performed using `rdflib` library accessed via LPS package.

- *Friends and Contacts* dropdown, this element is a selector for sending invitations to share the published visualizer with other users of the platform. The next subsection will provide more details on the collaborative sharing feature implemented as an additional functionality on top of the initial functional requirement. Collaborative sharing features implementation is inspired by *Linked Data Notifications (LDN)* [25] and uses the *ActivityStreams* vocabulary [26]. However, due to the specificity of LPA, it does not strictly follow all requirements of that specification.

Collaborative sharing

Collaborative sharing allows users of LPA platform to share their visualizers with other users of the platform. The sharing process implementation is inspired by Linked Data Notifications specification and involves the generation of an invitation file that send to the recipients inbox. The Figure 5.26 demonstrates the process of submitting an invitation in detail.

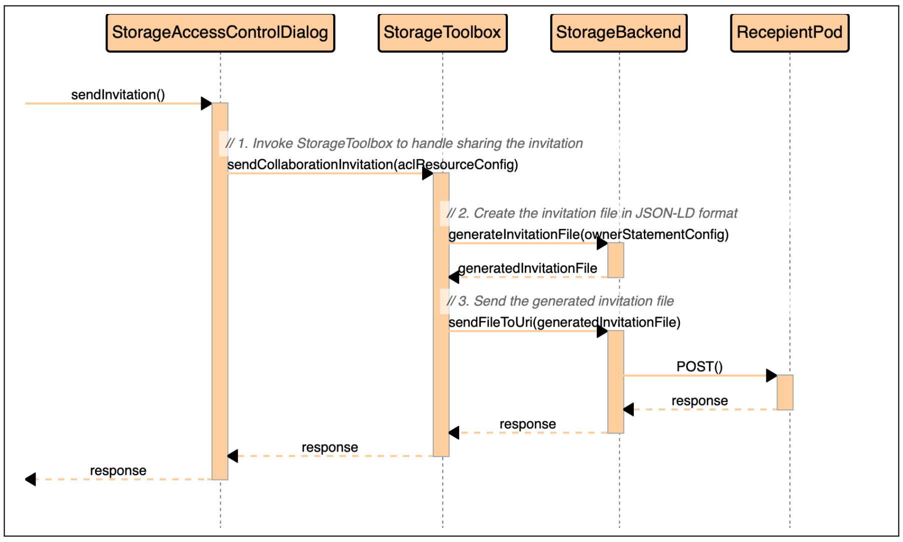

StorageAccessControlDialog
StorageToolbox
StorageBackend
RecepientPod
sendInvitation()
// 1. Invoke StorageToolbox to handle sharing the invitation
sendCollaborationInvitation(aclResourceConfig)
// 2. Create the invitation file in JSON-LD format
generateInvitationFile(ownerStatementConfig)
generatedInvitationFile
// 3. Send the generated invitation file
sendFileToUri(generatedInvitationFile)
POST()
response
response
response
response

Figure 5.26: A sequence diagram of implemented sharing functionality for collaborative editing.

The steps can be described as follows:

1. Firstly, the *sendInvitation()* function is invoked from *StorageAccessControlDialog* React Component, specifically when users chooses his available list of contacts and presses submit invitation button.

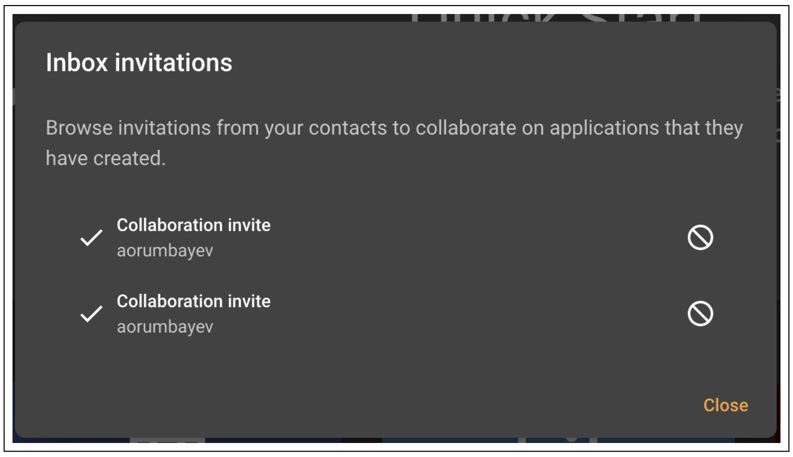
2. As a next step, *StorageToolbox* redirects the request to lower-level *StorageBackend* class.

3. The *StorageBackend* class generates the invitation RDF in JSON-LD format using `rdflib` library from LPS package.

4. The *StorageBackend* class sends the invitation file to recepients inbox folder using *FileManager* abstraction from LPS package. The exact inbox path is known by parsing the recipient's profile card associated with his WebID.

5. Asyncronous response is returned of the operations is propagated back to *StorageAccessControlDialog* for further processing.

It is important to note that the recipient is assumed to be a user of LPA platform. Once invitation is send, whenever recipient opens the platform, the


Inbox invitations

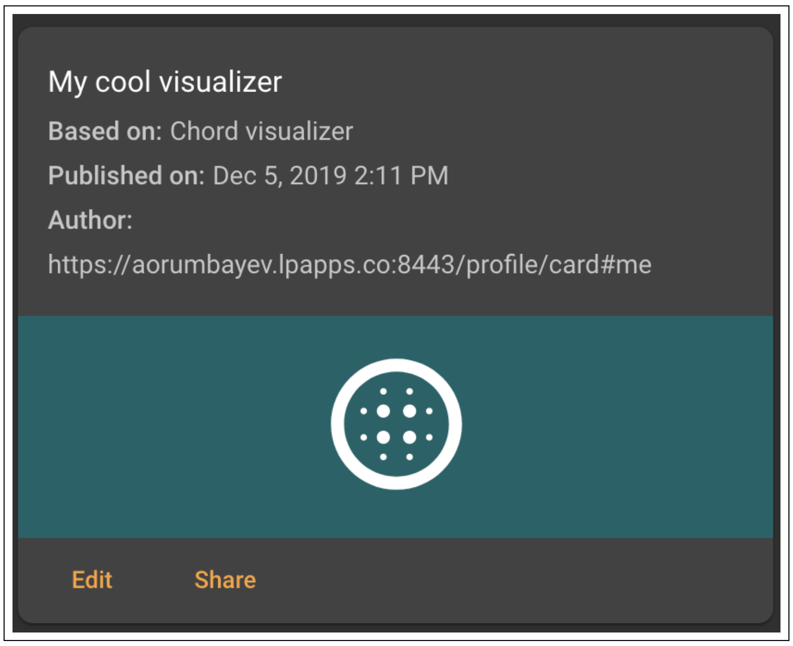
Browse invitations from your contacts to collaborate on applications that they have created.

✓ Collaboration invite
aorumbayev

✓ Collaboration invite
aorumbayev

Figure 5.27: An inbox dialog popup with two new invitations to collaborate on application.

listener inside *AppRouter* class will check the inbox for new invitations and display new invitations in inbox popup demonstrated on Figure 5.27. If user *Accepts* the invitation, the invitation JSON-LD file is transformed into a shared configuration file by extracting URI to an application and placing it under a folder named *sharedApplications* in the root storage folder and a response notification is sent back to the sender. Once sender receives the *Accept* notification, it sets the *READ* and *WRITE* access to the application configuration for the recipient. If user *Declines* the invitation, the invitation is deleted from the inbox without any further processing.


My cool visualizer

Based on: Chord visualizer

Published on: Dec 5, 2019 2:11 PM

Author:

https://aorumbayev.lpapps.co:8443/profile/card#me

Figure 5.28: A shared visualizer card displayed to recipient in his Storage Dashboard after he accepts the invitation.

The Chapter 1 demonstrates the shared application appearing in the shared applications storage dashboard of the recipient. In addition to that, the code on Listing 17 demonstrates the example of the generated invitation being send to recipient in JSON-LD format using *ActivityStreams* vocabulary.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Invite",
  "actor": "https://sender.lpapps.co:8443/profile/card#me",
  "name": "lpapps_invite",
  "object": {
    "type": "Link",
    "href": "https://sender.lpapps.co:8443/
    linkedpipes/configurations/1575551497789.473.ttl"
  },
  "published": "2019-12-06T20:04:02Z",
  "target": "https://recipient.lpapps.co:8443/profile/card#me"
}
```

Listing 17: An example invitation to collaborate on a published application.

The collaborative aspect is basic and only allows the recipient to rename the title of the application and modify and persist the selection of available filters. The changes are reflected in real-time using socket connection to Solid server as initially described in Section 5.4.3.

To sum up, the following section iterated over each functional requirement stated by LPA and demonstrated the detailed description of implementation to comply with those requirements. The section also continued the storage-related React Components implementation details provided in Section 5.3 and demonstrated finalized renders of those components that closely follow the initial mocks designed in Chapter 4.

5.5 Implemented non-functional requirements

The section provides a general overview of implementation of non-functional requirements stated in Section 3.2. Due to the majority of requirements being covered simply by the core functionality of the Solid project itself, the implementation details of individual requirements are less descriptive than descriptions in Section 5.4. It is also important to note that this section provides the implementation overview of all stated functional requirements defined in Section 3.2.

5.5.1 Compatibility with latest tools

The implemented solution consists of three parts:

- The LPS package, an npm package containing the generic abstractions performing low-level interactions with instances of Solid servers. Relies on `rdflib`, `solid-auth-client` and `solid-auth-cli` libraries. All third party packages are pinned, stable and indirectly unit tested, more details on testing is provided in Chapter 7.

- The Storage React components, a set of frontend components written in ES6 inside the LPA package. No additional third party packages are introduces to LPA with these components, except for LPS package.
- The LPS ontology, a vocabulary designed with `Protégé` and published with `Ontoology`. Used in LPA by accessing its public hosted instance, therefore no additional third party packages are introduced.

5.5.2 Clean APIs and libraries

The whole intent of creating the LPS package was to refactor and redesign the abstractions interacting with solid inside LPA frontend codebase. Therefore, the final refactored implementation is easier to maintain and use by LPA developers due to usage of TypeScript, extensive automated testing coverage covered in Chapter 7, and documentation included in Chapter 8.

5.5.3 Continuous Integration and Delivery

The LPS package is located in a separate GitHub repository, has an automated continuous delivery pipeline invoking unit test, code formatting, and linting. The continuous delivery aspect publishes new npm versions in is semi-automated manner. The frontend storage components in LPA are incorporated into frontend codebase. Hence they are validated using the LPA CI and CD pipelines. Some improvements into LPA testing pipelines were introduced, and they are described in detail in Chapter 7.

5.5.4 Easy integration with LPA

The LPS package is distributed via npm, and seamlessly integrated into LPA using the package management software. The frontend storage components are implemented inside the frontend codebase. However, they also maintain a consisted structure that improves code readability and defines the logical separation between functionalities of LPA and LPS.

5.5.5 Decentralized storage

The solution for this non-functional requirement is covered by Solid specification. Since Solid servers are decentralized by design, it means that the *Authentication-Manager* from LPS package allows the users of LPA to authenticate with any arbitrary instance of their private Solid server instances or instances from third-party Solid providers. The *FileManager* abstraction also allows users to move the created LPA configurations between Solid PODs within a Solid server instance or between different instances of compatible Solid servers.

6. Evaluation

This chapter briefly describes and evaluates the results, recognition, and notable achievements obtained after evaluating final LPS solution inside the production instance of LPA platform [1].

6.1 Benefits of Solid

Usage of Solid technology significantly enhanced the initial functionality of LPA, allowing to utilize the full benefits of Semantic Web and Linked Data management. The section describes the individual advantages and their influence on the improvement of LPA platform.

6.1.1 ACL managed applications

Access Control Management is one of the core features provided by Solid. The implementation and integration of *AccessControManager* abstraction from LPS package, ability for LPA developers to easily create, and edit the ACL files. From the perspective of LPA platform users, this provided the ability to have full control and management over the data created by the LPA but stored in LPS package.

6.1.2 Everything is an RDF resource

Storing LPA applications in Solid means storing their configuration files expressed in TTL RDF format. Therefore, adding more value to the date as it can be queried, filtered, or transformed in any way possible using SPARQL querying language. Consider a use case where an experienced data journalist and developer who created and owns a set of multiple visualizers displaying markers on a map, he recently discovered a new dataset that consists of smaller datasets for applications he created previously. He can use SPARQL to analyze his configurations stored in his Solid POD and create a new configuration by aggregating the filters from smaller application configurations and LPA platform will attempt to re-create the application for a new configuration. Storing data as RDF also provides possibilities for future enhancements of LPS. For instance, consider a case when several LPA applications are embedded in a service similar to Wikipedia. One could easily create a SPARQL query to extract URIs to those visualizations and query any of the components of that configuration.

6.1.3 Provider agnostic storage

By design, Solid servers are decentralized, a single Solid POD can host multiple Solid POD, and user own the data inside each of the PODs. When an LPA platform users creates his first WebID and a Solid POD, all of LPA configurations are created inside the Solid POD in authenticated Solid server. However, it is not bound to any specific implementation of Solid servers, it is completely agnostic and supports any implementation of a server that complies to Solid specification.

[1] `https://applications.linkedpipes.com`

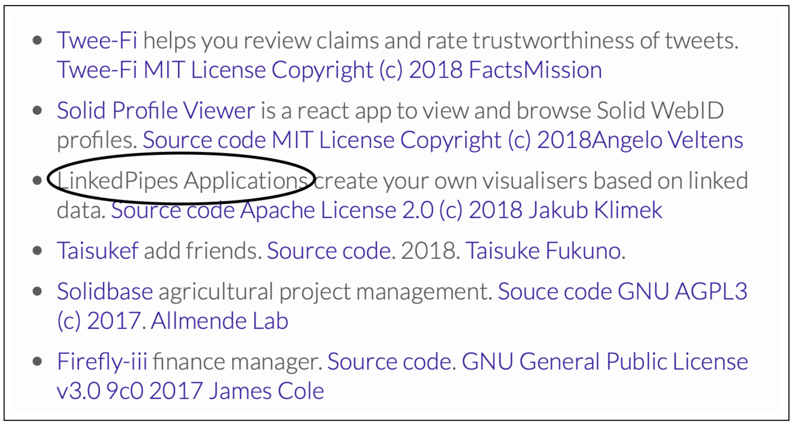
In addition to that, user has an ability to use the *Storage Control Panel* to move the created configurations within their PODs. The Solid POD webpage also allows users to move any data between different instances of Solid servers.

6.2 Results and achievements

The sections provide a brief overview of notable achievements and recognitions of LPA platform that was directly and indirectly influenced by usage of Solid technology.

6.2.1 Recognition on official Solid website

In July 2019, after the first production release of LPA platform with a partial implementation of LPS, the project was approved and added into the official list of Solid applications on the repository of Solid organization in GitHub. The screenshot of Figure 6.1 demonstrates the listing of the project on the official website.



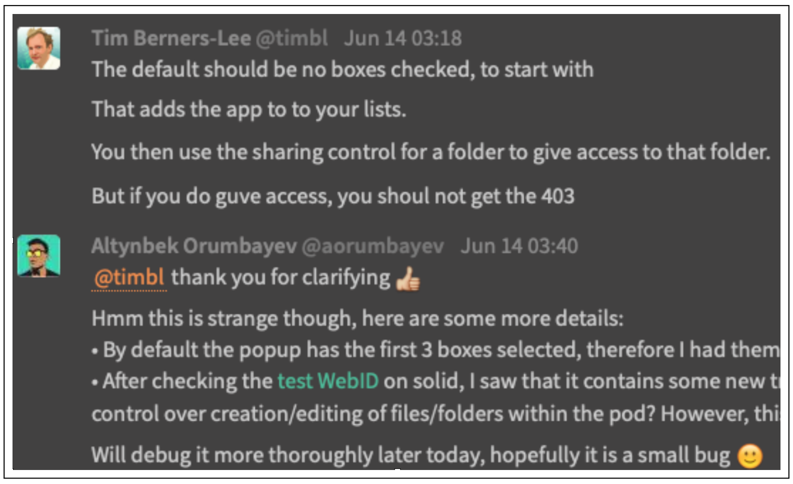
- Twee-Fi helps you review claims and rate trustworthiness of tweets. Twee-Fi MIT License Copyright (c) 2018 FactsMission
- Solid Profile Viewer is a react app to view and browse Solid WebID profiles. Source code MIT License Copyright (c) 2018Angelo Veltens
- LinkedPipes Applications create your own visualisers based on linked data. Source code Apache License 2.0 (c) 2018 Jakub Klimek
- Taisukef add friends. Source code. 2018. Taisuke Fukuno.
- Solidbase agricultural project management. Souce code GNU AGPL3 (c) 2017. Allmende Lab
- Firefly-iii finance manager. Source code. GNU General Public License v3.0 9c0 2017 James Cole

Figure 6.1: Listing of LPA platform on official Solid project website.

6.2.2 Comments from Sir Tim Berners-Lee

During the whole development lifecycle of the LPS project, active communication with the community helped to clarify many intricated nuances in the rapidly changing development ecosystem of Solid project. The creator of the Solid project is an active participant on *Gitter* [2] channels related to the project. During the implementation of *FileManager* abstraction, multiple questions asked on Solid channels on Gitter were answered by Sir Tim Berners-Lee, providing useful insights and motivation to contribute into the development of Solid as a web standard for decoupling and decentralizing the web. One of several interactions is demonstrated on a screenshot of a conversation of Figure 6.2.

[2] `https://gitter.im`



Tim Berners-Lee @timbl Jun 14 03:18
The default should be no boxes checked, to start with

That adds the app to to your lists.

You then use the sharing control for a folder to give access to that folder.

But if you do guve access, you shoul not get the 403

Altynbek Orumbayev @aorumbayev Jun 14 03:40
@timbl thank you for clarifying 👍

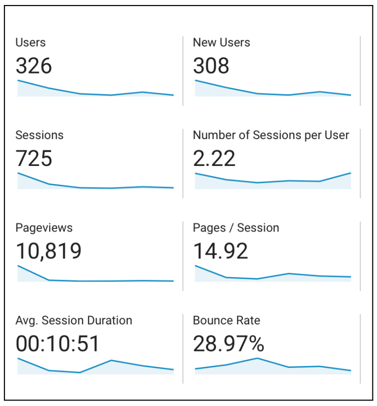
Hmm this is strange though, here are some more details:
• By default the popup has the first 3 boxes selected, therefore I had them
• After checking the test WebID on solid, I saw that it contains some new t
control over creation/editing of files/folders within the pod? However, thi

Will debug it more thoroughly later today, hopefully it is a small bug 🙂

Figure 6.2: One of multiple interactions with creator of Solid on Solid community chat.

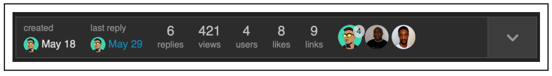
6.2.3 User traction on LPA platform

Evaluation of final release of LPA platform with finalized LPS solution received a feedback from multiple Solid community members. Additionally one of the members of Basel Register of Thesauri, Ontologies and Classifications (BARTOC) organization evaluated the application, provided a positive feedback and published an application stored in Solid using LPS on official website of organization [3]. It is important to note that the evaluators from BARTOC were using a beta staging version of the LPA platform. Therefore the live instance available on their website may not be available due to database updates after the final release of the platform in July 2019.



Users 326	New Users 308
Sessions 725	Number of Sessions per User 2.22
Pageviews 10,819	Pages / Session 14.92
Avg. Session Duration 00:10:51	Bounce Rate 28.97%

Figure 6.3: Google Analytics traction of users of test LPA platform instance over a period of six months.



created May 18 · last reply May 29 · 6 replies · 421 views · 4 users · 8 likes · 9 links

Figure 6.4: Feedback, views and comments on for LPA launch on official Solid community forum.

Additionally, screenshots on Figure 6.3 and Figure 6.4 demonstrate the traction of Solid community users. Release of the LPA platform with LPS storage was posted on official Solid community forum, where a notable positive commentary was left by one of the founders of *Virtuoso Universal Server* Kingsley Uyi Idehen

[3] `https://new.bartoc.org/node/332`

[4]. Over the period of six month starting July and ending on December 2019, over 300 users interacted with test instance of LPA as demonstrated on Figure 6.3.

[4] `http://dbpedia.org/page/Kingsley_Uyi_Idehen`

7. Testing

The following section will provide an overview of testing practices followed during the implementation described in Chapter 5. The chapter will start by describing the preliminary technologies used for implementing automated testing, integration, and delivery pipelines as well as libraries used for unit testing of TypeScript base LPS package.

7.1 Technologies used

The main development stack consisted of the following technologies:

- ***ava.js*** [1] is a Node.js unit testing library. It provides a clean and minimalistic syntax for writing tests, concurrent tests execution, includes TypeScript definitions and supports asyncronous functions. All of the aforementioned factors were considered when choosing this package as main unit testing library for LPS package.

- ***tslint*** [2] is an extensible static analysis tool that checks TypeScript code for readability, maintainability, and functionality errors. It is widely supported across modern editors and build systems and can be customized with your own lint rules, configurations, and formatters.

- ***istanbul.js*** [3] is a code coverage analyzer. It checks ES2015+ JavaScript code with line counters providing a very extensive overview of all classes and their coverage.

- *Travis CI* [4] is a hosted, distributed continuous integration service used to build and test software projects hosted at GitHub. Travis CI is used to check that newly committed code does not break the build and, consequently, the system. This ensures that developers are not disrupted and that the system remains stable. It is also able to run available automated tests to check further that the system is working correctly, even if the build isn't broken.

- *Renovate* [5] is an automated dependency updater. Multi-platform and multi-language. Due to various dependencies used in the *frontend* web component, it is crucial to keep the project up to date with the latest stable software releases. Renovate is set up and being triggered using Github Webhooks each weekend to check for updates in `package.json` and make a pull request to `master` and `develop` branches if any available.

- *Codecov* [6] is an automated code review tool that allows developers to improve code quality and monitor technical debt. Codecov automates code

[1]`https://github.com/avajs/ava`
[2]`https://palantir.github.io/tslint/`
[3]`https://istanbul.js.org/`
[4]`https://travis-ci.org/`
[5]`https://docs.renovatebot.com/`
[6]`https://codecov.io/`

reviews and monitors code quality on every commit and pull request. It reports back the impact of every commit or pull request in new issues concerning code style, best practices, security, and many others. It monitors changes in code coverage, code duplication, and code complexity. It allows developers to save time in code reviews, tackle issues efficiently, and overall makes the system maintainability much easier.

- *Slack* [7] a cloud-based proprietary instant messaging platform developed by Slack Technologies. The platform was used for automated notifications from GitHub and Travis-CI to report the states of executed CI and CD pipelines and for communication with the LPA developers.

7.2 Unit testing

Every class in LPS package was unit tested using the ava.js library. The convention used required in-place creation of a file with ava.js tests definitions named identical to a TS file to be tested but with `.spec.` prefix appended before the file extension. The code on Listing 18 demonstrates the example serial ava.js test executed in order. The asynchronous *createResource* can be reused in multiple ava.js tests improving code reusability while writing new tests.

```
async function createResource(t: any, input: any, expected:
↪  any): Promise<any> {
  const result = await StorageFileManager.createResource(input);
  logger.info(result.text());
  t.is(result.status, expected);
}

test.serial(
  'createFolderResource',
  createResource,
  folderConfigurationResource,
  201
);
```

Listing 18: An example of a reusable asynchronous test chunk and single serial ava.js test for *createResource()* method from *FileManager* abstraction in LPS package.

The istanbul.js JavaScript coverage library has integration with ava.js. The official command-line interface of istanbul.js, called `nyc` is used to invoke the unit tests to both run the tests and visualize the test results in the terminal in a convenient format.

[7] `https://slack.com/`

Unit tests are automatically executed on every push to any remote branch on LPS package repository. More details on executing the unit tests in automated pipelines are provided in next section.

7.3 Continious Integration and Delivery

The section will provide details on implementing automated integration and delivery pipelines using Travis CI. The terms Continuous Integration stands for a development practice that requires a developer to integrate his latest commits to a shared repository regularly. On the other hand, the name Continuous Delivery stands for an ability to deliver code changes of any type to production in a quick, safe, and automated way. Both of the practices were taken into consideration while implementing the LPS package.

7.3.1 Using Travis CI

Travis CI was used in pairs with the GitHub platform. As the first step, the service was connected with LPS package GitHub repository using Webhooks [8]. Secondly, a special YAML configuration file was added to all branches of LPS repository. The configuration file defined the integration and delivery pipelines to be executed inside Travis CI. Using a special feature called *build matrix*, the LPS was set to be unit tested automatically in concurrent manner on both Node.js `v10.x` and Node.js `v12.x`.

The branch protection rules were also applied to `dev` and `master` branches on LPS repository. The additional step in Travis CI pipeline configuration was added for `master` branch. Every push on remote `master` on LPS repository also involves deployment to npm. Failed build states are propagated into a special slack channel in LPA Slack workspace, channel is dedicated specifically for reporting CI and CD pipeline execution statuses for all branches on LPS repo.

7.3.2 Integration testing in LPA

The diagram on Figure 7.1 is a detail demonstration of how LPA and LPA package are integration tested in automated and semi-automated manner. It is important to note that the example on diagram considers a case when LPS developer merges changes into `master` branch of LPS package.

The operational flow can be described as follows:

1. *LPS developer pushes code to* `master` *branch*, the code is pushed to a remote GitHub repository. This triggers the Travis CI and Codecov reviews via Webhooks.

2. *Travis CI evaluates the results of pipeline.* If result is successfull it automaticall pushes the new version into npm and updates the static documentation for TypeScript classes. If failed, then the LPS developer is notified on Slack channel with a report on why the pipeline is failed.

[8]`https://en.wikipedia.org/wiki/Webhook`

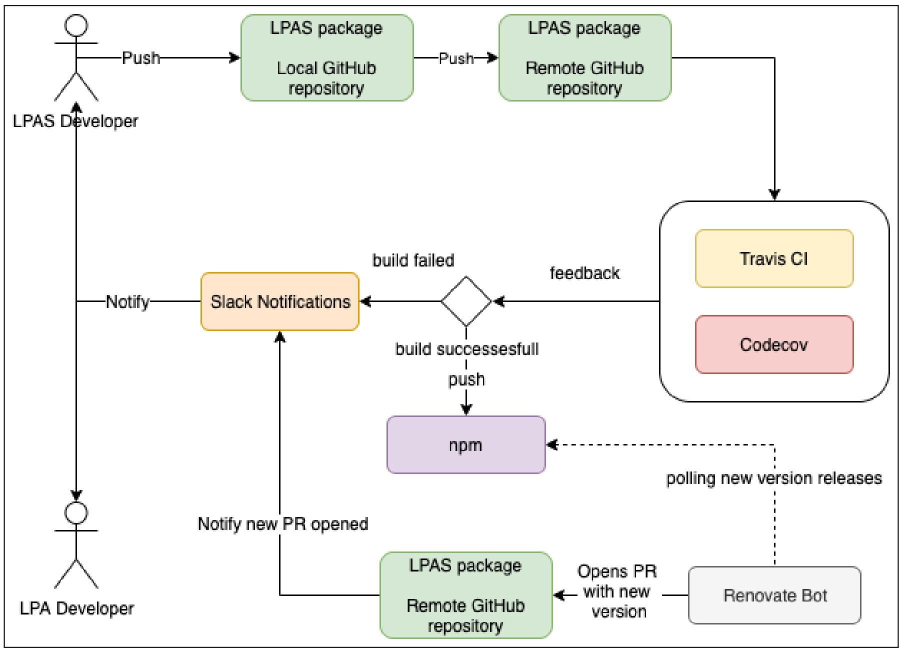


Figure 7.1: A detailed overview of CI and CD pipeline interaction for LPA and LPS.

3. *Renovate bot polls for new LPS releases*, the Renovate bot instance attached to LPA platform repository periodaclly polls npm and opens a Pull Request with new LPS package to LPA platform on a branch called `dependency-updates`.

4. *The LPA developer is notified via Slack.* When a Pull Request into LPA codebase is opened by Renovate, it notifies the LPA developer via Slack to review the Pull Request and merge new release of LPS package.

8. Documentation

The following chapter provides the information on available documentation resources implemented for both LPS package and LPA Storage Components. Selected code snippets demonstrate the core usage and provide detailed references for hosted documentations. The first section will start with user documentation covering a generic interaction flow with LPA platform using LPS package and React Storage Components. The remaining sections will dive deeper into the developer documentation. Starting from the installation guide and documentation of LPA package hosted on GitHub Pages and finalizing at an overview of how LPA Storage Components were integrated into documentation resources of LPA platform.

8.1 User documentation

This section is intended for non-developers who want to evaluate and try out the latest release of LPA platform with integrated LPS package and LPA Storage Components implemented. The documentation will serve as a tutorial expanding a generic user story first introduced in Figure 1.2. To recap, we have two target platform users named John and Bob. John attempts to use LPA platform, create and publish the application. On the other hand, Bob wants to see the resulting application created and shared by John. The documentation will cover interactions with a live instance of the latest LPA release, hosted at the Software Engineering department at Faculty of Mathematics and Physics in Charles University. The user-story based guideline will focus mainly on LPS and Solid related aspects, therefore for more specific information regarding other LPA platform features refer to the official user documentation [1] of the platform.

8.1.1 Creating Account

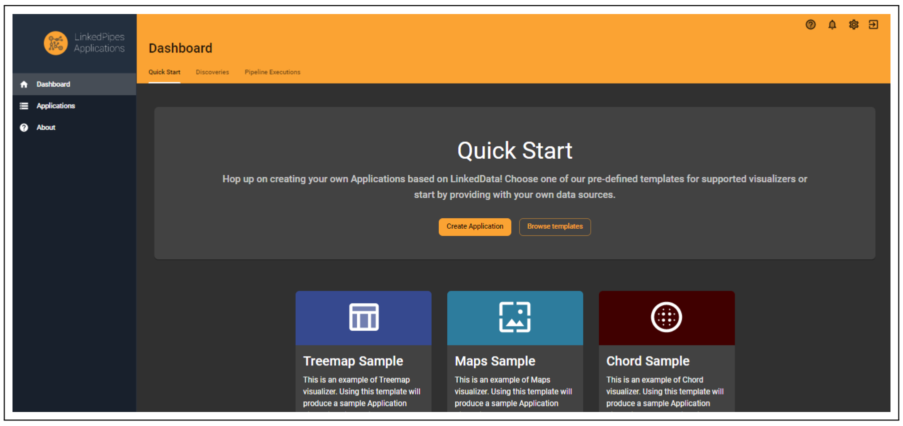
The first step for John is to create his first Solid POD and associated WebID profile. Referring back to Subsection 5.4.1 and Figure 5.12, after selecting a provider, for instance LinkedPipes PODs, he will be redirected to a Solid server login webpage demonstrated at Figure 5.14, where a *Create an account* button needs to be clicked. After that he will be redirected to a Solid server signup webpage demonstrated at Figure 5.14. Lastly after creating a Solid account he will be redirected to a his POD webpage. Performing the login at LPA instance website will now be possible with John's new WebID and Solid POD created under selected provider.

After successfull authentication an LPA Dashboard page demonstrated on Figure 8.1 will be presented.

Notes for providers using the latest NSS versions

One distinct difference in NSS releases starting from the fifth version, and higher is more strict access control settings when authenticating a Solid application

[1] `https://docs.applications.linkedpipes.com`



Quick Start

Hop up on creating your own Applications based on LinkedData! Choose one of our pre-defined templates for supported visualizers or start by providing with your own data sources.

Treemap Sample

This is an example of Treemap visualizer. Using this template will produce a sample Application

Maps Sample

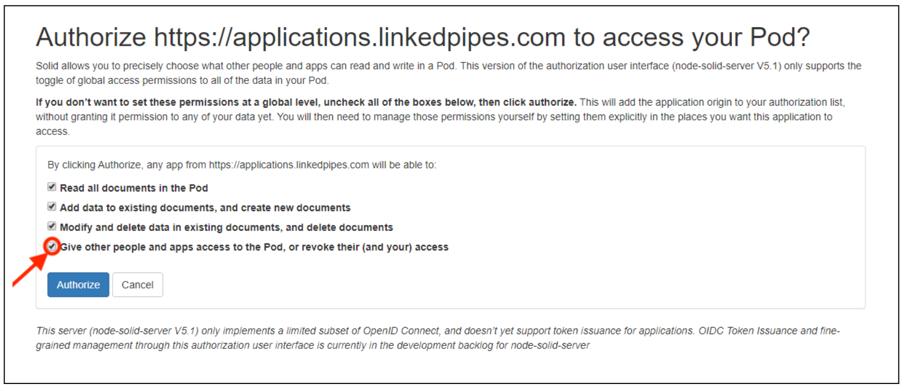
This is an example of Maps visualizer. Using this template will produce a sample Application

Chord Sample

This is an example of Chord visualizer. Using this template will produce a sample Application

Figure 8.1: The Dashboard page displayed after successfull authentication into LPA platform.

with a Solid POD. Since the majority of stable features of LPS were developed before under fourth release iteration of NSS, it was impossible to simplify the authentication flow for the newer version due to constant changes and updates in both NSS releases and Solid specifications. Therefore, for LPA users that are intended to use the providers that supply Solid servers based on the latest NSS version, there is one additional manual step required during initial signup. As demonstrated on Figure 8.2, this popup will be displayed during the initial login to LPA platform after creating a Solid account, it is crucial to enable the *Give other people, and apps access to the Pod, or revoke their (and your) access* option. This will ensure that the Solid app will have enough privileges to create and edit the ACL files associated with visualizer configurations inside the POD.



Authorize https://applications.linkedpipes.com to access your Pod?

Solid allows you to precisely choose what other people and apps can read and write in a Pod. This version of the authorization user interface (node-solid-server V5.1) only supports the toggle of global access permissions to all of the data in your Pod.

If you don't want to set these permissions at a global level, uncheck all of the boxes below, then click authorize. This will add the application origin to your authorization list, without granting it permission to any of your data yet. You will then need to manage those permissions yourself by setting them explicitly in the places you want this application to access.

By clicking Authorize, any app from https://applications.linkedpipes.com will be able to:

- ✓ **Read all documents in the Pod**
- ✓ **Add data to existing documents, and create new documents**
- ✓ **Modify and delete data in existing documents, and delete documents**
- ✓ **Give other people and apps access to the Pod, or revoke their (and your) access**

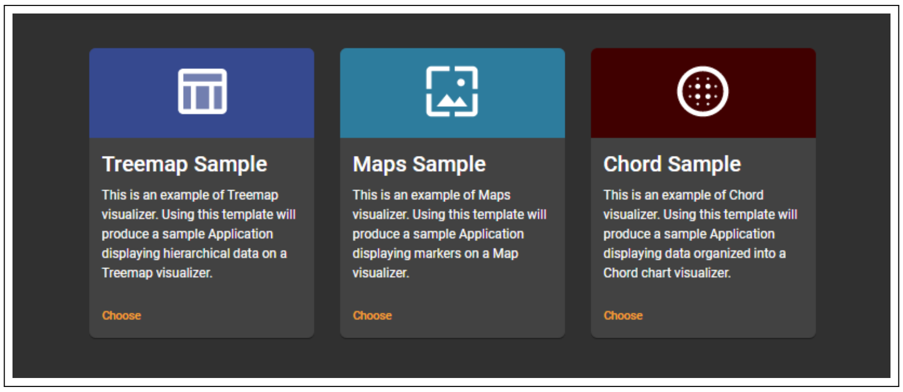
This server (node-solid-server V5.1) only implements a limited subset of OpenID Connect, and doesn't yet support token issuance for applications. OIDC Token Issuance and fine-grained management through this authorization user interface is currently in the development backlog for node-solid-server

Figure 8.2: An access control popup displayed on latest NSS instances.

8.1.2 Creating Application

To create an application, John must either provide the necessary data source information or select one of the data sample templates provided at the Dashboard page.

To use one of the data samples provided by LPA, on the Dashboard page, John needs to click the *Choose* button on the desired data sample template. This will redirect him the Create Application page, as displayed on Figure 8.4, where the data source fields will already be pre-filled, and he can proceed to click *Start Discovery* immediately. In this guide, it is assumed that John started the application creation process using templates. For more details on other ways to provide data sources, refer to the official LPA platform documentation.



Treemap Sample

This is an example of Treemap visualizer. Using this template will produce a sample Application displaying hierarchical data on a Treemap visualizer.

Choose

Maps Sample

This is an example of Maps visualizer. Using this template will produce a sample Application displaying markers on a Map visualizer.

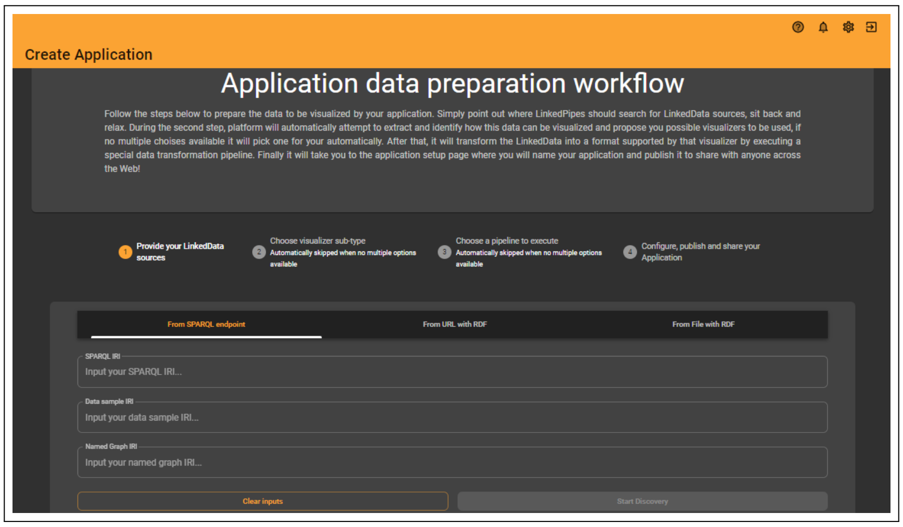
Choose

Chord Sample

This is an example of Chord visualizer. Using this template will produce a sample Application displaying data organized into a Chord chart visualizer.

Choose

Figure 8.3: Template data sources provided by LPA platform.



Create Application

Application data preparation workflow

Follow the steps below to prepare the data to be visualized by your application. Simply point out where LinkedPipes should search for LinkedData sources, sit back and relax. During the second step, platform will automatically attempt to extract and identify how this data can be visualized and propose you possible visualizers to be used, if no multiple choises available it will pick one for your automatically. After that, it will transform the LinkedData into a format supported by that visualizer by executing a special data transformation pipeline. Finally it will take you to the application setup page where you will name your application and publish it to share with anyone across the Web!

1 Provide your LinkedData sources
2 Choose visualizer sub-type — Automatically skipped when no multiple options available
3 Choose a pipeline to execute — Automatically skipped when no multiple options available
4 Configure, publish and share your Application

From SPARQL endpoint | From URL with RDF | From File with RDF

SPARQL IRI: Input your SPARQL IRI...

Data sample IRI: Input your data sample IRI...

Named Graph IRI: Input your named graph IRI...

Clear inputs | Start Discovery

Figure 8.4: Create Application page.

After the discovery process finishes, the user will be prompted to pick one of the recommended visualizers, as displayed on Figure 8.5. If only one possible visualizer is available, this step will be skipped as that visualizer will be automatically picked by LPA.

Clicking on the desired visualizer will trigger the data transformation process, as displayed on Figure 8.6. In other words, the pipeline transformations will be applied to the whole RDF data based on the data sources given. If more than one applicable pipeline is found, John will be asked to choose one before this process begins.

The resulting data will then be in the format required to be visualized using the selected visualizer.

Once this step is complete, as presented on Figure 8.7, John can click on 'Create App' to visualize the data and customize the visualization on a separate webpage called *Application Control and Setup*.

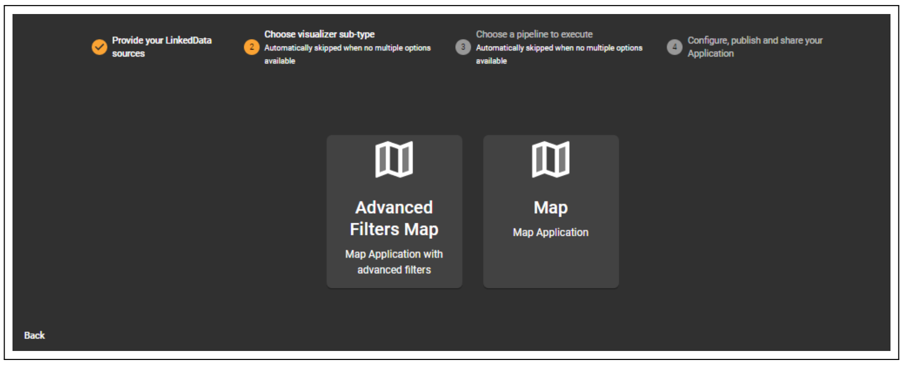
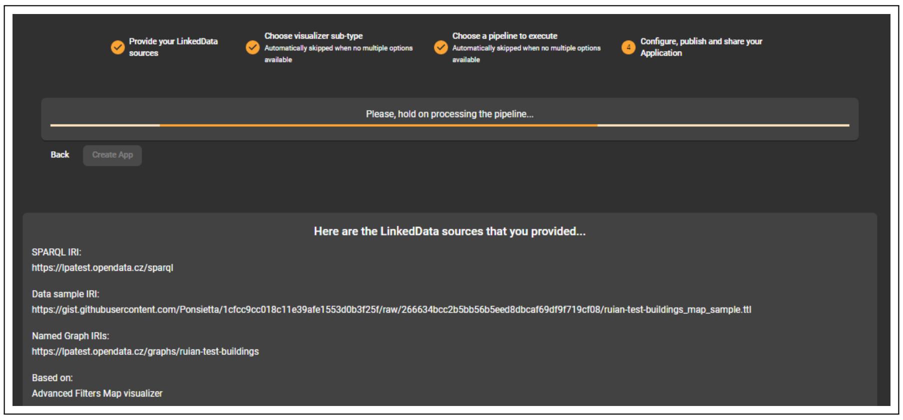

Provide your LinkedData sources
Choose visualizer sub-type
Automatically skipped when no multiple options available
Choose a pipeline to execute
Automatically skipped when no multiple options available
Configure, publish and share your Application
Advanced Filters Map
Map Application with advanced filters
Map
Map Application
Back

Figure 8.5: Create Application page - Choose Visualizer.


Provide your LinkedData sources
Choose visualizer sub-type
Automatically skipped when no multiple options available
Choose a pipeline to execute
Automatically skipped when no multiple options available
Configure, publish and share your Application
Back
Create App

Please, hold on processing the pipeline...

Here are the LinkedData sources that you provided...

SPARQL IRI:
https://lpatest.opendata.cz/sparql

Data sample IRI:
https://gist.githubusercontent.com/Ponsietta/1cfcc9cc018c11e39afe1553d0b3f25f/raw/266634bcc2b5bb56b5eed8dbcaf69df9f719cf08/ruian-test-buildings_map_sample.ttl

Named Graph IRIs:
https://lpatest.opendata.cz/graphs/ruian-test-buildings

Based on:
Advanced Filters Map visualizer

Figure 8.6: Create Application page - Executing pipeline.

8.1.3 Publishing application

As presented on Figure 8.8, the *Application Control and Setup* page consists of the preview of the visualizer to be published, a set of controls to configure the filters (if available) and a header component with elements to publish or embed the application. To publish an application by storing the visualizer configuration in Solid POD, John needs to input the title for his application and click on the *Publish* button. Alternatively he can click on *Embed* application an will be presented with a simple popup to generate the HTML `iframe`. For more specific details on publishing and storing in Solid refer to the description in Subsection 5.4.2. For more specific details on configuring the post published application refer to Subsection 5.4.3.

The screenshot on Figure 8.9 represents a webpage for the stored application configuration inside the Solid POD. Every published application configuration have their own URI that is used during the generation of a URL to share the visualization.

8.1.4 Sharing the Application

As the final step of this user-story based interaction flow, John either embeds the application into his website or shares the link to his application with Bob. The screenshot on Figure 8.10 demonstrates the resulting published application assembled from visualizer configuration stored in John's Solid POD as an RDF

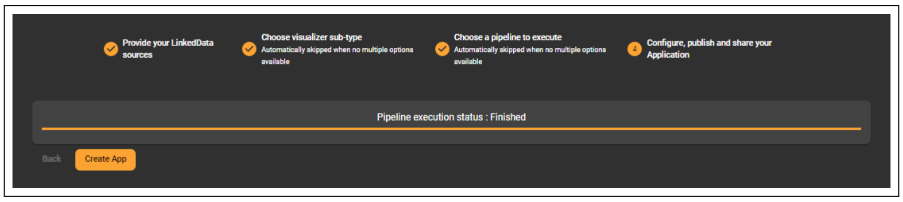

Provide your LinkedData sources
Choose visualizer sub-type
Automatically skipped when no multiple options available
Choose a pipeline to execute
Automatically skipped when no multiple options available
Configure, publish and share your Application
Pipeline execution status : Finished
Back
Create App

Figure 8.7: Create Application page - Finished pipeline execution.

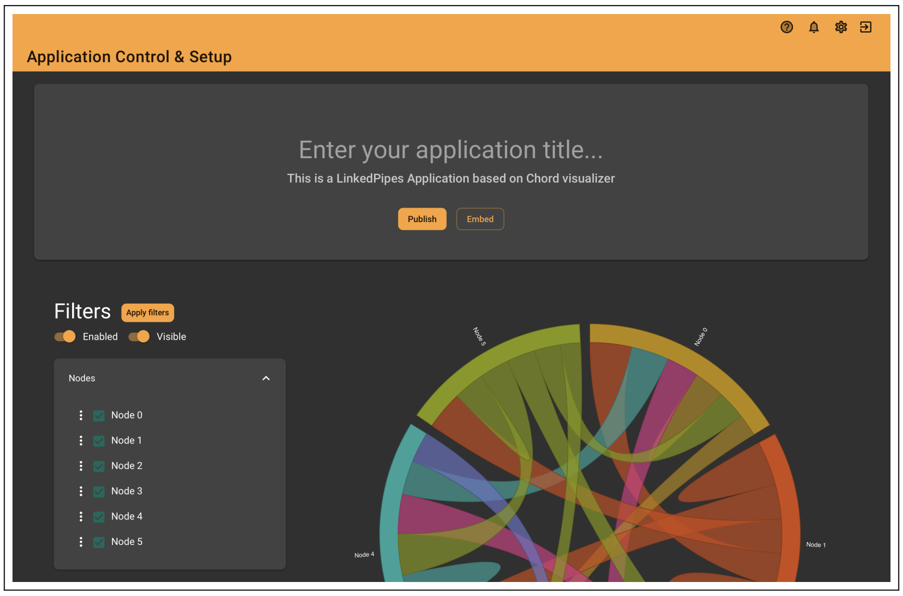

Application Control & Setup
Enter your application title...
This is a LinkedPipes Application based on Chord visualizer
Publish
Embed
Filters
Apply filters
Enabled
Visible
Nodes
Node 0
Node 1
Node 2
Node 3
Node 4
Node 5
Node 5
Node 0
Node 4
Node 1

Figure 8.8: Application Control and Setup webpage.

TTL resource. Additionally, Bob can access the webpage without creating an official profile inside the LPA platform since it is publically available.

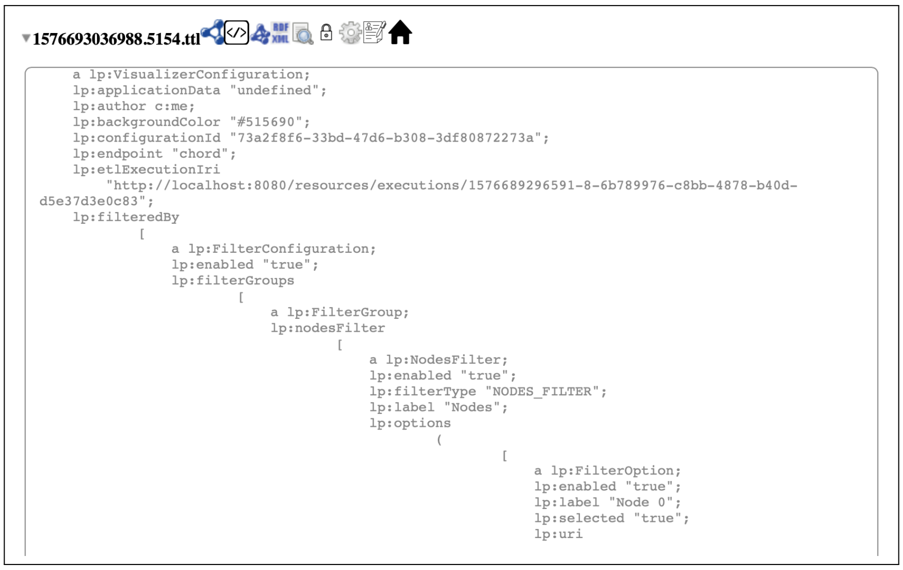
To sum up, this section concludes the user-oriented documentation demonstrating the most basic interaction flow with LPA platform using LPS package and React Storage Components.

8.1.5 LPA platform guide

The user-story based guide demonstrated in the previous subsection only covered the essential functionality of LPA made possible with the use of LPS and Solid. Another important non-developer documentation resource is the LPA platform documentation. That documentation was expanded with additional sections covering functionality, directly and indirectly, involving the interactions with Solid servers. Each section of the documentation is presented as a detailed video tutorial hosted on YouTube and can be described as follows:

- *Create, publish and embed your application* [2] a video tutorial guiding users from start of application visualization creation to the finishing step where it is stored inside Solid POD, and a published URL is generated using the URI of the resource from the POD.

[2] `https://docs.applications.linkedpipes.com/tutorials/3.creating_applications/`



```
    a lp:VisualizerConfiguration;
    lp:applicationData "undefined";
    lp:author c:me;
    lp:backgroundColor "#515690";
    lp:configurationId "73a2f8f6-33bd-47d6-b308-3df80872273a";
    lp:endpoint "chord";
    lp:etlExecutionIri
        "http://localhost:8080/resources/executions/1576689296591-8-6b789976-c8bb-4878-b40d-
d5e37d3e0c83";
    lp:filteredBy
            [
                a lp:FilterConfiguration;
                lp:enabled "true";
                lp:filterGroups
                        [
                            a lp:FilterGroup;
                            lp:nodesFilter
                                    [
                                        a lp:NodesFilter;
                                        lp:enabled "true";
                                        lp:filterType "NODES_FILTER";
                                        lp:label "Nodes";
                                        lp:options
                                                (
                                                        [
                                                            a lp:FilterOption;
                                                            lp:enabled "true";
                                                            lp:label "Node 0";
                                                            lp:selected "true";
                                                            lp:uri
```

Figure 8.9: Published application configuration inside Solid POD.

- *Configuring application and filters* [3] a video tutorial demonstrating the configuration of published applications and editing of filter settings as described in Subsection 5.4.3.
- *Adding SOLID contacts, collaborative editing* [4] a video tutorial demonstrated a process of adding new SOLID contacts in an instance of node-solid-server, and then demonstrates the process of sharing a published app with other user of the platform user as described in Section 5.4.5.
- *Managing platform and user settings* [5] a video tutorial demonstrating how to interact with the storage configuration and the Solid user profile settings popups.

More detailed and developer-oriented documentation is provided in consecutive chapters.

8.2 Developer documentation

The following section is intended for developers interested in a more in-depth overview of both LPS package and React Storage Components implemented inside LPA frontend codebase.

8.2.1 Installation

The following section will provide a set of guidelines covering various use-cases on trying out the LPS package on a local machine. The installation process depends

[3] https://docs.applications.linkedpipes.com/tutorials/4.configuring_published_applications/
[4] https://docs.applications.linkedpipes.com/tutorials/5.adding_solid_contacts_sharing/
[5] https://docs.applications.linkedpipes.com/tutorials/6.misc/

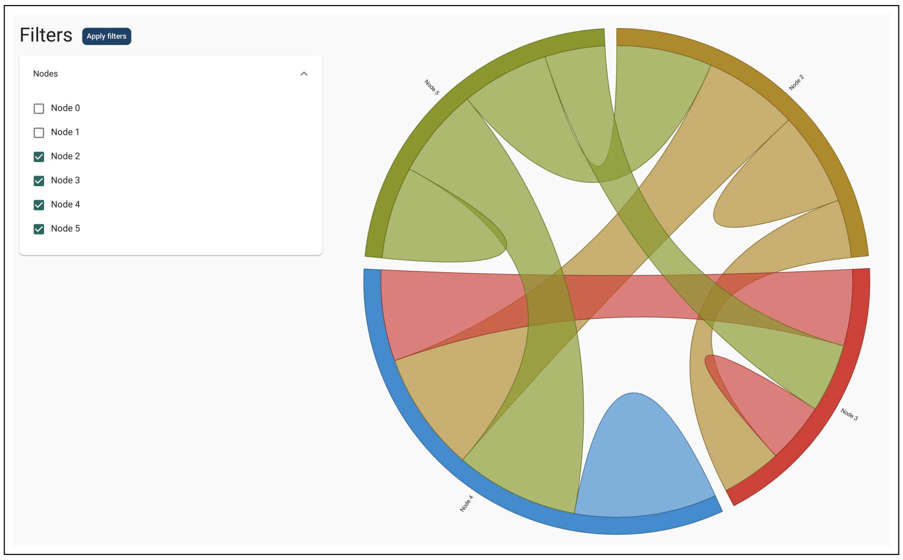


Figure 8.10: Published and publicly accessible application accessed via *Share* URL.

on two separate use-cases. First, it relies on simply trying to use the LPS npm package. Second, relies on installing the whole LPA platform to interact with Solid.

LPS package

It is important to note that this installation guide is not entirely generic for any Solid application. However, the *AuthenticationManager* and *FileManager* abstractions provide a generic functionality to be utilized in majority of Solid app development use-cases that involve authentication and interaction with resources inside the POD. The following prerequisites are required in order to install the LPS package on a local machine:

- *Node.js v10.15.x and higher* [6]. The LPS is distributed via npm, therefore a proper version of Node.js and a corresponding package manager are a required prerequisite.
- *yarn v1.19.1* [7]. Yarn is an alternative package manager similar to npm. Due to extensive usage of this package manager during development of LPS, the author recommends it to be used as an alternative to npm.

```
$ yarn add linkedpipes-storage # if using yarn package manager
# or
$ npm install linkedpipes-storage # if using npm
```

Listing 19: Installing the LPS package locally via yarn or npm.

[6] https://nodejs.org/en/
[7] https://yarnpkg.com/en/

The code on Listing 19 demonstrates the installation of the package at the folder from which the command is invoked. In other words, it is assumed that the package is being installed into a Solid based web app project.

LPA platform

Installation of entire LPA platform is even more straightforward process in contrast with LPS package. However, due to the complexity of components inside the platform, the recommended way to install it is by using Docker. Therefore, the only required prerequisite is the latest stable version of Docker and Docker Compose [8].

```
$ curl https://git.io/fjXIB -o lpa-cli.sh && chmod +x lpa-cli.sh
↪  && ./lpa-cli.sh --production-no-cloning
```

Listing 20: Installing the LPA platform locally using docker-compose.

The code snippet on Listing 20 demonstrates the local installation of entire LPA platform using docker-compose, whose invocation is conveniently wrapped into a command line interface by LPA developers.

8.2.2 LPS package

The package is source codes are available at public GitHub repository [9] and corresponding page on npm [10]. The generic documentation on repository *README*, as well as on a webpage at npm, includes the installation guide and a quick start on creating, editing, and deleting resources using a folder as an example.

More detailed and developer oriented documentation is available on repository GitHub Page [11]. The website is generated and published as a part of automated CI and CD pipeline demonstrated at Figure 7.1. The documentation is generated using `typedoc` [12], a documentation generator for TypeScript project. The codebase is annotated using `tsdoc` [13], an official TypeScript comment standard developed by Microsoft.

The typedoc documentation provides the overview of entire LPS project codebase in the following order:

1. *Enumerations* description of all enumeration types in codebase.
2. *Classes* description of all class types in codebase.
3. *Interfaces* description of all interface types in codebase.

[8] `https://docs.docker.com`
[9] `https://github.com/aorumbayev/linkedpipes-storage`
[10] `https://www.npmjs.com/package/linkedpipes-storage`
[11] `https://aorumbayev.github.io/linkedpipes-storage`
[12] `https://github.com/TypeStrong/typedoc`
[13] `https://github.com/microsoft/tsdoc`

4. *Variables* description of all variable types in codebase.

5. *Functions* description of all function types in codebase.

Every individual type is expanded into a detailed description of its sub-elements. For instance, every class type documentation is structured as follows:

1. *Constructors* description of all class constructors, input parameters and return values.

2. *Properties* description of all properties of the class, their types and inheritance hierarchy.

3. *Constructors* description of all class method, input parameters and return values.

For more details, refer to the package documentation website listed in the footnote at the beginning of this section.

8.2.3 Storage Components

The Storage Components documentation is a part of LPA documentation. Therefore the section will firstly provide a brief description of how the LPA platform is documented and finally how the Storage Components documentation was integrated.

There are three main documentation sources provided by LPA platform described as follows:

- *The platform documentation* [14], as mentioned and demonstrated earlier in Section 8.1, it contains non-developer oriented tutorials, introduces core concepts of the platform, and provides a detailed set of video tutorials demonstrating the available feature. This documentation was expanded by including interactions with storage components. The corresponding video tutorials were recorded to illustrate and guide users to use and interact with storage. The static webpage was generated using `hugo` framework [15].

- *The frontend documentation* [16] contains developer-oriented guideline over frontend components of the platform generated using `docz` [17]. Provide the main installation, quick start, and interactive documentation of selected components of the platform. Each of the interactive components can be immediately forked into `codesandbox` environment [18] and tested in an online web IDE environment.

- *The backend documentation* [19] contains developer-oriented guideline over backend components of the platform generated using `orchid` [20]. No additional documentation for LPS related code was added since no changes were required on the backend component side.

[14]`https://docs.applications.linkedpipes.com`
[15]`https://gohugo.io`
[16]`https://docs.frontend.applications.linkedpipes.com`
[17]`https://www.docz.site`
[18]`https://codesandbox.io`
[19]`https://docs.backend.applications.linkedpipes.com`
[20]`https://orchid.run`

For references to a so-called *Admin documentation* and information on hosting the LPA instance using LPS refer to project GitHub repository [21].

Expanding frontend documentation

As mentioned earlier in the section, the frontend documentation was created using docz framework, which relies on a special markdown syntax called *MDX* [22]. It allows for combining the advantages of generic markdown syntax and JavaScript code snippets. Therefore, inside the frontend codebase, each component representing the individual platform webpage had an MDX file added, having the same name as the JSX component file. Later on, the docz framework automatically assembles the static HTML pages based on the declared MDX files describing components.

The following Storage Components were added into general frontend documentation by implementing the corresponding MDX files:

- *StoragePage* represents a set of components rendered into a Storage Dashboard. The documentation includes the structure of stateful and stateless components that are associated with the webpage. The main properties passed to the components are also described. Lastly, a live instance of that webpage is rendered inside the documentation webpage, allowing developers to interact with the component.

- *SettingsPage* represents both user profile and storage control settings pages. The documentation includes the structure of stateful and stateless components that are associated with the webpage and main properties used by those components.

The frontend documentation also contains the dedicated section describing the Storage Components in general and provides several references to Solid toolset that was used as well as the LPS package. For more details, refer to the package documentation website listed in the footnote at the beginning of this section.

[21]`https://github.com/linkedpipes/applications`
[22]`https://mdxjs.com`

Conclusion

To summarize, in the following work implements a multipart decentralized storage solution for LPA platform based on Solid project. The term multipart reffers to implementation of the LPS npm package, the LPS Ontology used to represent LPA visualizer configurations as RDF files and lastly, a set of Storage Components implemented in LPA frontend.

An overview of related tools in Chapter 2 provided a set of software technologies alternative to Solid. The chapter also provided a comparison table described in Subsection 1.3.2 that demonstrated benefits of choosing Solid as a core storage technology for LPA platform.

An analysis and description of LPA requirements and Solid development toolset in Chapter 3 defined the main practical tasks to be achieved by LPS as well as to which Solid libraries and frameworks to utilize in implementatioin. The final reasoning expanding the conclusion from Chapter 2 on choosing solid was provided at the end of the chapter in Section 3.4.

Based on the performed analysis on Solid and requirements stated by LPA, Chapter 4 described a detailed overview of a designed architecture of LPS. The architecture consisted of three main parts. Firstly, the Section 4.2, provided the design of abstractions for a LPS npm package providing functionality for authenticating (Subsection 4.2.1), operating the resources inside Solid PODs (Subsection 4.2.2) and managing ACL files (Subsection 4.2.3). Afterwards, the Section 4.3 provided an overview of designed OWL Ontology for representing the LPA visualizer configurations as RDF resources inside Solid POD as well as describing the benefits of such approach to storing visualizer data. And lastly, Section 4.4 provided a set of designed user interface mocks complying to stated LPA requirements as well as a detailed overview of the functioinality provided by those user interface components.

Continuing the architecture overview, Chapter 5 described the entire implementation of the storage functionality and structured similar to the previous chapter as it implements the designed elements in order with their design and architecture. The Section 5.1 provided detailed overview of the LPS package implementation using TypeScript programming language. The Section 5.2 described the process of implementing and hosting the designed LPS Ontology using Protégé and Ontology open-source tools. The Section 5.3 described the implementation of Storage React components inside the LPA frontend codebase. The final renders of components were demonstrated in Section 5.4 by iterating of functional requirements. Lastly, in Section 5.5 an overview of implemented non-functional requirements was provided. Chapter also clearly demonstrated that all defined functional and non-functional requirements of LPA were covered and implemented, thus fulfilling the practical goals of the thesis.

The Chapter 6 demonstrated the improvements introduces to LPA after evaluating the platform with fully integrated LPS solution. The chapter also provided the main results and achievements obtained as a part of the implementation stage and evaluation process, such as:

- Recognition on official website of the Solid project (Subsection 6.2.1).

- Comments and short conversations with Sir Tim Berners-Lee in regards to questions on Solid specifications asked by author (Subsection 6.2.2).
- Demonstration of user traction over a six month period of continious delivery of the LPS solution. Positive feedback and recognition by Solid community members and member of BARTOC organization (Subsection 6.2.3).

A detailed overview of the testing of LPS solution was provided in Chapter 7. In Section 7.2 a demonstration of how the LPS package itself was unit tested and automated with Continious Integration and Delivery pipelines using Travis CI. And lastly, in Section 7.3 an overview of improvements introduced into LPA automated build pipelines and integration testing of both LPA and LPS solutions was provided.

The whole LPS solution was thoroughly documented both on the npm package side as well as by expanding the LPA documentation to include Storage Components as described in Chapter 8.

8.3 Future work

The Solid ecosystem is constantly expanding in its specifications, community, and available technological toolset. Throughout fulfilling the goals of the thesis, many challenges were faced due to the aforementioned constant changes in the Solid project. Over time once, Solid technology will mature for more advanced production level use cases, and the ecosystem of decentralized social applications will grow. The dependencies such as node-solid-server and solid-auth-client used in LPS package might require significant updates and refactor. Additionally, by the time of finishing the practical part of this thesis, several libraries were introduced by official Solid contributors, including a brand new Solid server implementation in TypeScript called `pod-server` [23] aimed to eventually replace the node-solid-server. Aside from that a library called `tripledoc` [24] was introduced, aiming to potentially become the standard library to interact and manage resources in Solid PODs.

Therefore, potential improvements and future work in LPS package and storage components include the following:

- *Gradual refactoring and replacement of node-solid-server.* Replacing the node-solid-server implementation with more stable and actively supported pod-server implementation could introduce more user-friendly experience while performing authentication and manipulation of resources inside Solid PODs.
- *Gradual integration of tripledoc into LPS package.* As tripledoc potentially offers the same functionality as the LPS package, the potential future work includes replacing certain low-level interactions with rdflib and using tripledoc instead. This can simplify the maintainability of LPS package, simplify unit-testing, and can potentially make it a generic utility for storing and managing any configuration ontologies in Solid.

[23]`https://github.com/inrupt/pod-server`

[24]`https://vincenttunru.gitlab.io/tripledoc/`

- *Improving collaborative editing.* One of the extra features implemented within the scope of LPS was the ability to share published applications within users of LPA platform and let them configure the published application collaboratively. An improved version of that can include real-time editing of the resources more robustly, while the current version discards any simultaneous real-time changes submitted by collaborating users.

- *General support of the project.* This, of course, assumes the long term general support of the solution and parts of LPA platform covering the Storage functionality to keep it up to date with all the latest improvements and changes introduces in Solid specification.

The ambitious goal of decentralizing the World Wide Web set by Sir Tim Berners-Lee is yet to demonstrate its benefits and receive a more comprehensive recognition across non-developer oriented domains and average Internet users. However, we believe that the fundamental specifications of Solid project that improves upon established Web Standards, an active and passionate community, and a developer-friendly environment and tools to builds decentralized social applications will define the next generation of Internet technologies and make it more secure and privacy-oriented.

Bibliography

[1] Sebastian Ismael Garrido Simon. 30 years on, what's next #fortheweb?, 2019.

[2] World Wide Web Consortium (W3C). SPARQL 1.1 Query Language, 2013.

[3] W3C. Web Ontology Language (OWL), Nov 2009.

[4] W3C RDF Core Working Group. RDF - Semantic Web Standards, Feb 2014.

[5] Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[6] Dan Brickley and Libby Miller. FOAF Vocabulary Specification, 2000.

[7] Andy Seaborne and Eric Prud'hommeaux. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[8] Solid. Web Access Control (WAC). Accessed: 2018-10-13.

[9] Jakub Klímek, Petr Skoda, and Martin Necaský. LinkedPipes ETL: Evolved Linked Data Preparation. In *The Semantic Web - ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers*, pages 95–100, 2016.

[10] Jakub Klímek, Jirí Helmich, and Martin Necaský. LinkedPipes Visualization: Simple Useful Linked Data Visualization Use Cases. In *The Semantic Web - ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers*, pages 112–117, 2016.

[11] Ames Bielenberg, Lara Helm, Anthony Gentilucci, Dan Stefanescu, and Honggang Zhang. The growth of diaspora - A decentralized online social network in the wild. In *2012 Proceedings IEEE INFOCOM Workshops, Orlando, FL, USA, March 25-30, 2012*, pages 13–18, 2012.

[12] Diaspora. Diaspora Decentralized Social Network. Accessed: 2018-11-03.

[13] Max Van Kleek, Daniel Smith, Nigel Shadbolt, and schraefel. A decentralized architecture for consolidating personal information ecosystems: The WebBox. 01 2012.

[14] Matteo Zignani, Sabrina Gaito, and Gian Paolo Rossi. Follow the "Mastodon": Structure and Evolution of a Decentralized Online Social Network. In *Proceedings of the Twelfth International Conference on Web and Social Media, ICWSM 2018, Stanford, California, USA, June 25-28, 2018*, pages 541–551, 2018.

[15] Read-Write Linked Data. A Reference Linked Data Platform server for the Solid platform, 2015.

[16] Solid. Solid server in NodeJS, 2015.

[17] Toby Inkster, Henry Story, and Bruno Harbulot. WebID Authentication over TLS, 2014.

[18] Solid. Specs for WebID-OIDC decentralized authentication protocol, 2016.

[19] Steve Speicher, John Arwe, and Ashok Malhotra. Linked data platform 1.0. W3C recommendation, W3C, February 2015. http://www.w3.org/TR/2015/REC-ldp-20150226/.

[20] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. Solid: A platform for decentralized social applications based on linked data, 2016.

[21] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11-15, 2016, Companion Volume*, pages 223–226, 2016.

[22] Natalya Fridman Noy, Monica Crubézy, Ray W. Fergerson, Holger Knublauch, Samson W. Tu, Jennifer Vendetti, and Mark A. Musen. Protégé-2000: An open-source ontology-development and knowledge-acquisition environment: AMIA 2003 open source expo. In *AMIA 2003, American Medical Informatics Association Annual Symposium, Washington, DC, USA, November 8-12, 2003*, 2003.

[23] Ahmad Alobaid, Daniel Garijo, María Poveda-Villalón, Idafen Santana-Pérez, and Óscar Corcho. Ontoology, a tool for collaborative development of ontologies. In *Proceedings of the International Conference on Biomedical Ontology, ICBO 2015, Lisbon, Portugal, July 27-30, 2015*, 2015.

[24] Erik Rasmussen. Ducks: Redux Reducer Bundles, 2015.

[25] Sarven Capadisli, Amy Guy, Christoph Lange, Sören Auer, Andrei Vlad Sambra, and Tim Berners-Lee. Linked data notifications: A resource-centric communication protocol. In *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I*, pages 537–553, 2017.

[26] James Snell and Evan Prodromou. Activity Streams 2.0. *World Wide Web Consortium, Working Draft WD-activitystreams-core-20150722 (July 2015)*, 2017.

List of Figures

List of Tables

Acronyms

ACL Access-Control List. 10

API Application Programming Interface. 14, 15

BARTOC Basel Register of Thesauri, Ontologies and Classifications. 86

CRUD Create Read Update Delete. 30

ETL Extract, Transform and Load. 15

FOAF Friend of a Friend. 8

JSON-LD JavaScript Object Notation for Linked Data (syntax). 8

LDN Linked Data Notifications. 80

LDP Linked Data Platform. 32

LDP-BC Linked Data Platform Basic Container. 33

LDPR Linked Data Platform Resource. 33

LOV Linked Open Vocabularies. 5

LPA LinkedPipes Applications. 5, 20

LPS LinkedPipes Storage. 5

MIME Multipurpose Internet Mail Extensions. 7

ORDBMS Object-relational database management system. 15

OWL Web Ontology Language. 7

POD Personal Online Dataspace. 10

RDBMS Relational database management system. 15

RDF Resource Description Framework. 7

SPARQL SPARQL Protocol and RDF Query Language. 7

TLS Transport Layer Security. 26

TTL Turtle (syntax). 8

URI Uniform Resource Identifier. 7

URL Uniform Resource Locator. 61

W3C World Wide Web Consortium. 5

A. Online sources

- https://github.com/aorumbayev/linkedpipes-storage - LPS GitHub repository.
- https://www.npmjs.com/package/linkedpipes-storage - LPS npm web page.
- https://aorumbayev.github.io/linkedpipes-storage/ - LPS developer documentation.
- https://git.io/JeQSk - LPS Ontology GitHub repository.
- https://w3id.org/def/lpapps - LPS Ontology documentation.
- https://github.com/aorumbayev/solid_diploma_thesis - source texts of thesis work in LaTeX, compatible with Overleaf.