



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF MASTER'S THESIS

Title: Linked Data Notifications and ActivityPub Client and Server
Student: Bc. Antonín Karola
Supervisor: RNDr. Jakub Klímek, Ph.D.
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of summer semester 2019/20

Instructions

The student will get familiar with Linked Data, the RDF data model, the recent W3C Recommendations [1][2][3] and the Solid project [4], a recent activity of the inventor of the Web, Sir Tim Berners-Lee. The student will implement a client and a server supporting decentralized messaging on the Web according to the Linked Data Notifications [2] and ActivityPub [3] W3C Recommendations in support of the Web re-decentralization.

The client part will be a new, user friendly messaging application.

Based on the analysis of existing Solid server implementations, the student will determine what is missing in the existing implementations for the given task.

The missing features will be implemented either as a new Solid server, or an existing implementation will be enhanced.

The client and the server will be documented, evaluated, tested and published as open-source on GitHub.

The tests will consist of unit tests and tests of compatibility with existing tools implementing the Recommendations.

References

- [1] Linked Data Platform 1.0, W3C Recommendation, 2015, <https://www.w3.org/TR/ldp/>
- [2] Linked Data Notifications, W3C Recommendation, 2017, <https://www.w3.org/TR/ldn/>
- [3] ActivityPub, W3C Recommendation, 2018, <https://www.w3.org/TR/activitypub/>
- [4] Solid, MIT, <https://solid.mit.edu/>

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 18, 2018



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Linked Data Notifications and ActivityPub Client and Server

Bc. Antonín Karola

Katedra softwarového inženýrství
Supervisor: RNDr. Jakub Klímeck, Ph.D.

April 26, 2021

Acknowledgements

I would like to thank my supervisor, RNDr. Jakub Klímek, Ph.D. for his guidance, valuable insight and patience.

Besides my supervisor, I would like to thank my family for their endless support, not just throughout my studies. Furthermore, I would like to extend a big thank you to my friends for their moral support and believing in me, especially Petr, Petra, Míša and Radim. My sincerest thanks also goes to Nathaniel, Vratislav, Ema, Honza and Mike for their help.

Last but not least, thanks to IDC CEMA for their flexibility and meeting me halfway with my needs.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on April 26, 2021

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2021 Antonín Karola. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Karola, Antonín. *Linked Data Notifications and ActivityPub Client and Server*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

„Získejme web zpět!“ - Sir Tim Berners-Lee, tvůrce World Wide Webu.

Pro podporu re-decentralizace webu, principů otevřených dat a skutečného vlastnictví dat, buduje tato práce aplikace na základě technologií od Web Consortium (W3C).

Práce prozkoumává nejnovější W3C protokoly a doporučení: Linked Data (propojená data), RDF datový model, Linked Data Platform, Linked Data Notifications (LDN), Activity Streams (AS), ActivityPub (AP) a projekt Solid. Pro důkaz použitelnosti těchto technologií jsou vytvořeny a publikovány tři proof-of-concept aplikace (aplikace na ověření konceptu).

Hlavním cílem této práce je vytvořit uživatelsky přívětivou webovou aplikaci podporující decentralizovanou komunikaci. Jako první je provedena analýza existujících LDN a AP aplikací. Na základě této analýzy je vybrán *solid-server* jako server pro LDN část a *Pleroma* server pro AP část. Jako klient je vytvořena nová aplikace *Inbox*.

Inbox je webová aplikace napsaná ve frameworku *Angular*. Tento klient je otestován pomocí automatických unit a end-to-end testů. Uživatelské rozhraní aplikace je podrobena uživatelskému testování kognitivním průchodem. Na základě výsledků je pak aplikace vylepšena.

Nakonec je *Inbox* klient zdokumentován, publikován jako open-source na GitHubu a instance aplikace je nasazena na web.

Klíčová slova webová aplikace, propojená data, návrh webové aplikace, implementace webové aplikace, Linked Data Platform, Linked Data Notifications, ActivityPub, Activity Streams, RDF, JavaScript, REST, JavaScript Notifications API, JavaScript Push API, node.js, Solid, Angular

Abstract

”Reclaim the web!” - Sir Tim Berners-Lee, the inventor of the World Wide Web.

To support Web re-decentralization, open data principles and true data ownership, this thesis builds applications on top of the Web Consortium (W3C) technologies.

This thesis investigates the current W3C protocols and recommendations: Linked Data, the RDF data model, the Linked Data Platform, Linked Data Notifications (LDN), Activity Streams (AS), ActivityPub (AP) and the Solid project. To prove their applicability, three proof-of-concept applications are created and published.

The main goal of the thesis is to create a user-friendly web application supporting decentralized messaging. First, an analysis of existing LDN and AP applications is conducted. Based on this analysis, *solid-server* is selected as the server for the LDN part and the *Pleroma* server for the AP part. As a client, a new *Inbox* application is created.

Inbox is a web application written in *Angular* framework. This client is then tested using automated unit and end-to-end tests. Application’s user interface is subjected to the cognitive walk-through user testing. Based on the results, the application is enhanced.

Finally, the client is documented and published as open-source on GitHub and its instance deployed on the web.

Keywords Web application, Linked Data, Linked Data Platform, Linked Data Notifications, ActivityPub, Activity Streams, RDF, JavaScript, REST, JavaScript Notifications API, JavaScript Push API, node.js, Solid, Angular

Contents

Introduction	1
Goals of this work	3
1 State-of-the-art and available technology	5
1.1 Current technologies that address the centralization problem	5
1.1.1 RDF	6
1.1.2 Linked Data	7
1.1.3 Linked Data Platform	8
1.1.4 Linked Data Notifications	10
1.1.5 Activity Streams	13
1.1.6 ActivityPub	14
1.1.7 Solid	16
2 Analysis	19
2.1 Requirements	19
2.1.1 Actors	19
2.1.2 List of requirements	20
2.2 Use cases	21
2.2.1 List of use cases - consumer	21
2.2.2 List of use cases - sender	23
2.3 Analysis of existing solutions - LDN clients and Solid servers	25
2.3.1 Criteria for analysis of existing solutions	25
2.3.2 Overview of all analysed applications - LDN	26
2.3.3 Detailed analysis of selected applications	29
2.4 Analysis of existing solutions - ActivityPub applications	34
2.4.1 Criteria for analysis of existing AP solutions	35
2.4.2 Overview of analysed servers - AP	35
2.4.3 AP servers analysis result	37
2.5 Implementation analysis	37

2.6	Analysis results	38
3	Design	39
3.1	System architecture: in-browser web application + server back-end	39
3.2	Client	39
3.2.1	Programming languages and frameworks	39
3.2.2	Application architecture: MVC	40
3.2.3	User interface design - wireframes	40
3.2.4	Linked Data Notifications part	46
3.2.5	ActivityPub part	49
3.3	Server	51
3.3.1	Linked Data Notifications part - solid-server	52
3.3.2	ActivityPub part - Pleroma	52
4	Implementation	53
4.1	Client	53
4.1.1	Linked Data Notifications part	53
4.1.2	ActivityPub part	59
4.2	Server	61
5	Testing	63
5.1	Unit tests	63
5.1.1	Coverage	64
5.2	E2E tests	64
5.2.1	Coverage	65
5.3	Continuous integration	65
5.4	Usability testing	66
5.4.1	Cognitive walk-through	66
5.4.2	How the cognitive walk-through was conducted	66
5.4.3	Test cases	66
5.4.4	Cognitive walk-through testing results	68
5.5	Tests of compatibility with existing tools	70
5.5.1	LDP, LDN, Solid - Solid-server	70
5.5.2	ActivityPub - Pleroma	71
6	Documentation	73
6.1	User documentation	73
6.1.1	Login	73
6.1.2	Reading messages	74
6.1.3	Sending a message	76
6.1.4	Inbox monitoring and notifications	77
6.1.5	Pleroma connection	79
6.2	Administrator documentation	80

6.2.1	Source code	81
6.2.2	Live version	81
6.2.3	Requirements	81
6.2.4	Installation and build	81
6.2.5	Running the application	81
6.2.6	Tests	82
6.2.7	Deployment	82
6.2.8	Continuous integration (CI)	83
6.3	Developer documentation	83
6.3.1	Development environment	83
6.3.2	Project structure	83
6.3.3	Contributing to Inbox	84
Conclusion		85
	Problems encountered	86
	Future work	87
Bibliography		89
A Glossary		93
B Technical research - proof-of-concept applications		95
B.1	LDN-inbox - LDN proof-of-concept	95
B.1.1	Architecture	95
B.1.2	Technologies	96
B.1.3	Implementation	97
B.1.4	Documentation	100
B.2	LDN-target	101
B.2.1	Architecture	101
B.2.2	Technologies	102
B.2.3	Documentation	102
B.3	js-notification-poc	103
B.3.1	Architecture	103
B.3.2	Technologies	103
B.3.3	Implementation	104
B.3.4	Documentation	105
B.4	inbox-client	106
B.4.1	Architecture	107
B.4.2	Technologies	107
B.4.3	Implementation	107
B.4.4	Testing	110
B.4.5	Documentation	111
C Reported GitHub/GitLab issues		113

C.1	Problems with solid-client-authn-js library	113
C.1.1	Problem using the library in Angular	113
C.1.2	Library producing unsolicited request with 404 error	113
C.1.3	session.info.webId not available in onLogin callback	113
C.1.4	Library does not stay logged in after page reload	114
C.2	Solid server issues	114
C.2.1	Server sends phantom WebSocket pub messages	114
C.2.2	GET https://tonda.inrupt.net/inbox times out	114
C.2.3	Solid sends WebSockets messages for private resources without authentication	114
C.2.4	Solid uses incompatible WebSockets protocol version	114
C.2.5	POSTing ActivityPub message to Solid produces wrong content-type	115
C.3	Other repositories	115
C.3.1	LDN tests page unavailable	115
C.3.2	Cannot verify Pleroma OAuth token	115
C.3.3	Mastodon offers little to no ActivityPub client-to-server support	115
C.3.4	solid-auth-fetcher - missing method implementation	116
C.3.5	Questions in forums	116
D	Complete results of cognitive walk-through	117
D.1	TC1 - Read list of messages from all available inboxes	117
D.2	TC2 - Read list of messages from selected inbox	118
D.3	TC3 - Read detail of a received message	118
D.4	TC4 - Reply to message	119
D.5	TC5 - Send a simple message	119
D.6	TC6 - Send an AP message	120
D.7	TC7 - Start monitoring arbitrary inbox	120
D.8	TC8 - Stop monitoring arbitrary inbox	121
D.9	TC9 - Receive a system notification on a new message	121
D.10	General comments from the testing	121
E	Attached medium content	123

List of Figures

0.1	Example of centralized social networks	2
1.1	Example of a connected graph that can be represented with RDF	6
1.2	Structure of Linked Data Platform Resources	8
1.3	Overview of Linked Data Notifications	10
1.4	Illustration of a solid pod with application	17
2.1	UML diagram of Consumer use cases	23
2.2	UML diagram of Sender use cases	25
2.3	Solid inbox - list of messages. Screenshot of the current official inbox client application.	31
2.4	Solid inbox - message detail. Screenshot of the current official inbox client application.	32
2.5	Solid inbox - message content. Screenshot of the current official inbox client application.	33
3.1	Screen 1 - welcome page.	40
3.2	Screen 2 - login using Solid authentication - step 1.	41
3.3	Screen 3 - login using Solid authentication - step 2.	41
3.4	Screen 4 - start monitoring inbox - UC1.	42
3.5	Screen 5 - stop monitoring inbox - UC2.	42
3.6	Screen 6 - Read list of all messages - UC3, default screen. All messages from all monitored inboxes combined. This is the main screen user will see when he logs in.	43
3.7	Screen 7 - Empty list of all messages. All messages view, when no messages are available.	43
3.8	Screen 8 - Read list of messages from one inbox. All messages from chosen inbox.	44
3.9	Screen 9 - Read list of messages from one inbox - empty. Detail of inbox, when there are no messages available.	44
3.10	Screen 10 - Message detail - UC4.	45

3.11	Screen 11 - Send a notification - UCs 6 - 9, empty	On the screen, there is an option to send a notification to an either person from Solid contact list, or directly using IRI.	45
3.12	Screen 12 - Send a notification - UCs 6 - 9, filled	User can specify multiple recipients.	46
3.13	Application architecture	- web application running in user's browser, reading data from Solid POD hosted on a solid-server . .	47
3.14	UML diagram of application entities		48
3.15	Receiving system notifications for incoming messages	- sequence diagram illustrating User A receiving notification for new message from User B.	49
3.16	Communication with Pleroma server	- sequence diagram illustrating Client application communicating with Pleroma server. .	51
5.1	Message sent by Inbox test, consumed directly in the Solid data browser at Inrupt.net.		71
5.2	Message sent by Inbox test, consumed in the Solid POD "OhMyPod!" browser.		71
5.3	Post sent from Mastodon social network to the Pleroma test user, consumed in the Inbox client.		72
6.1	Login screen		74
6.2	Choosing login provider		74
6.3	List of all messages		75
6.4	Message content		75
6.5	Message content - detail		76
6.6	Send message - Activity Streams		76
6.7	Selecting recipient from contacts	- application offers list of user contacts	77
6.8	Selecting multiple recipients	- user can send a message to multiple recipients simultaneously	77
6.9	Application notification	for new message	78
6.10	System notification	- example of a system notification in OS MS Windows 10	78
6.11	Add inbox for monitoring		79
6.12	Step 1 - login to Pleroma		80
6.13	Step 2 - user's Pleroma statuses		80
6.14	Project structure.	Only notable files and folders are shown (e.g. ones that are not standard part of Angular or are important for development).	84
B.1	inbox - index screen.	Screenshot of the index page	98
B.2	inbox-client screen 1 - watched inboxes		109
B.3	inbox-client screen 2 - added watched inbox		110

B.4 inbox-client screen 3 - send message to a friend 110

Introduction

The Internet was designed from the start as a decentralized network. It began as the military's ARPANET, where in the case of one network node being incapacitated, technologies like network packets and dynamic routing would still allow for the rest of the nodes to communicate [1]. The internet infrastructure has since become very robust, and it is virtually impossible to take it down by disabling even multiple nodes.

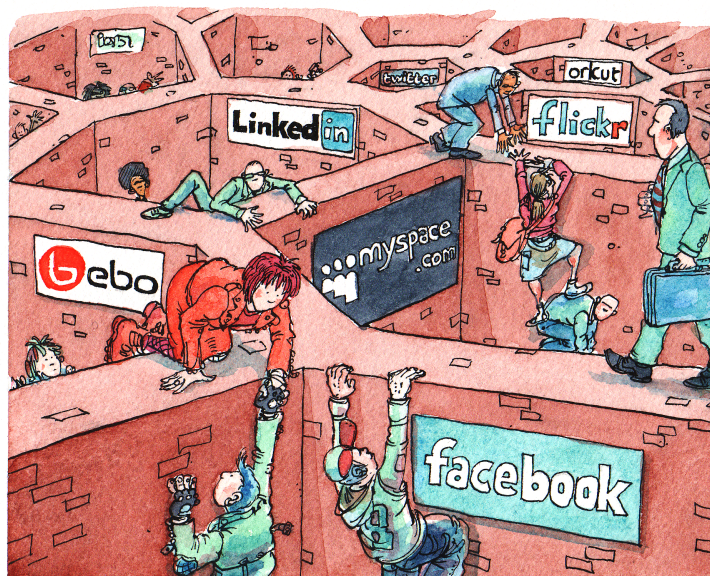
Meanwhile, market monopolization has introduced a new problem - **web centralization** [2]. Tech giants like Google (with YouTube) and Facebook (owning Instagram), have made users dependent on them for information or entertainment [3]. As a result, the internet has become very centralized regarding ownership and services. Furthermore, web applications are incapable of intercommunication because they are closed systems (e.g. a Facebook user cannot comment on YouTube, YouTube cannot send you notifications to the app of your choice, etc.).

An important part of this problem is data ownership. For example, when you upload your photograph to the Facebook platform, you are granting Facebook many rights:

"Specifically, when you share, post, or upload content that is covered by intellectual property rights ..., **you grant us** a non-exclusive, transferable, sub-licenseable, royalty-free, and worldwide **license to** host, use, distribute, modify, run, copy, publicly perform or display, translate, and **create derivative works of your content** ... This means, for example, that if you share a photo on Facebook, you give us permission to store, copy, and share it with others ... This license will end when your content is deleted from our systems." ¹

¹From Facebook terms of use: <https://www.facebook.com/terms.php>

Figure 0.1: Example of centralized social networks. Original at [4].



As we can see from recent events in the US², based on these user data, the tech giants can influence politics, access private messages, and delete user content without any justification.

Recently, these problems have become the focus of multiple re-centralization initiatives, e.g. Redecentralize.org³ or the Solid project⁴. The World Wide Web Consortium (W3C) together with the web creator Tim Berners-Lee are working to address these problems with technical solutions. They have proposed protocols such as Linked Data (LD), LD Platform (LDP) and Notifications (LDN), its extension ActivityPub (AP) and ActivityStreams (AS), so web developers can build their applications without these problems. Users would then be able to choose e.g. their data provider and applications could intercommunicate.

We can see this development already taking place with social networks such as the decentralized platform mastodon⁵ or AS video streaming service PeerTube⁶. In fact, a whole platform of interconnected, federated, open-sourced applications that are making use of the ActivityPub and other open protocols has emerged - *fediverse*⁷.

²<https://www.theguardian.com/technology/commentisfree/2019/oct/23/facebook-influence-next-election-democratic>

³<https://redecentralize.org/>

⁴<https://solidproject.org/>

⁵<https://joinmastodon.org/>

⁶<https://peertube.video/>

⁷Home page: <https://fediverse.party/>, about page: <https://fediverse.party/en>

Goals of this work

This work aims to provide a Linked Data Notifications and ActivityPub implementation that is easy to use and is not merely a proof-of-concept. The assignment has divided the implementation into two parts - a client and a server.

The client should be a user-friendly messaging application that takes advantage of system notifications.

The server should be an AP and LDN compliant implementation, either an enhanced existing one or a new implementation.

Both client and server should be documented, evaluated, tested and published as open-source on GitHub.

State-of-the-art and available technology

This chapter introduces current technologies that are being used to address the centralization problem. It covers protocols developed by the World Wide Web Consortium (W3C), mainly by its Linked Data Platform Working Group and Social Web Working Group. These include Linked Data, Linked Data Platform and Notifications.

The main building block is Linked Data (LD) with its RDF representation. It allows resources to be more than just a heap of binary data. LD introduces data semantics, it gives data meaning and allows the resources to be interlinked. Furthermore, this enables data to be computer-readable and allows automated querying and processing of the data.

On top of Linked Data and RDF, W3C has developed protocols such as LD Platform (LDP) and LD Notifications (LDN). These protocols specify data formats and communications methods, so compliant applications can work together and exchange data. This allows e.g. various web applications to interchange notifications and messages. More specifically, with a LDN-compliant social network, a user could post e.g. comments on a video from another LDN-compliant video application.

1.1 Current technologies that address the centralization problem

World Wide Web Consortium (W3C) is trying to address the centralization problem with various technologies. The technologies described in this section were not necessarily created to address the centralization problem, but they are being used to do so.

1.1.1 RDF

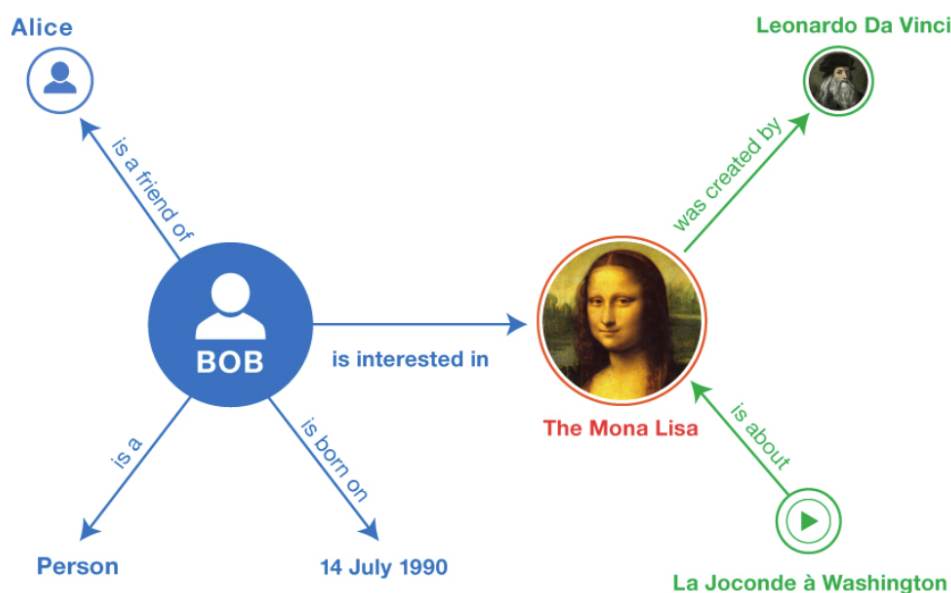
Resource Description Framework (RDF) is a standard graph data model created for data interchange on the web. It was created as a W3C specification [5] and is used for modelling information like web resources. RDF can be understood as a language for describing statements about things/entities. It consists of triples: subject + predicate + object:

Listing 1.1: RDF triple example

```
<subject> <predicate> <object>
```

These together create an oriented labeled multigraph, where subjects and objects represent nodes and predicates represent edges. Additionally, RDF supports named graphs, thus creating quads: subject + predicate + object + graph label.

Figure 1.1: Example of a connected graph that can be represented with RDF. Original at [6]



RDF supports a big variety of data serialization formats, Turtle/N-Triples being the most human-readable:

Listing 1.2: **RDF Turtle/N-Triples serialization example.** Turtle representation of the main subset of the graph at Figure 1.1

```
<http://example.org/#bob>
<http://perceive.net/schemas/relationship/isInterestedIn>
```


1.1. Current technologies that address the centralization problem

```
<http://example.org/#mona-lisa> .
```

In web applications, JSON-LD (JSON for Linking Data) [7] serialization is easier to use because JSON-LD is still valid JSON, which has robust support among web application technologies:

Listing 1.3: **JSON-LD serialization example.** JSON-LD representation of list of LDP notifications.

```
{
  "@context": "http://www.w3.org/ns/ldp#",
  "@id": "http://localhost:5001/API/notifications/",
  "@type": "ldp:Container",
  "ldp:contains": [
    {
      "@id": "http://localhost:5001/API/notifications/0"
    },
    {
      "@id": "http://localhost:5001/API/notifications/1"
    },
    {
      "@id": "http://localhost:5001/API/notifications/2"
    },
    {
      "@id": "http://localhost:5001/API/notifications/3"
    },
    {
      "@id": "http://localhost:5001/API/notifications/4"
    }
  ]
}
```

1.1.1.1 Relation of RDF to this thesis

RDF is the basic data model that the technologies used in this thesis build on. All linked data technologies like LDN, JSON-LD and ActivityPub build on RDF. Furthermore, it is essential part of open data and e.g. the Solid initiative.

1.1.2 Linked Data

Linked Data are structured data that are interlinked. More specifically, it is a term defined by Tim Berners-Lee in his 2006 design note "Linked Data" [8]. He outlines four basic principles of Linked Data:

1. use URI (IRI) to identify entities
2. use HTTP URI to access data

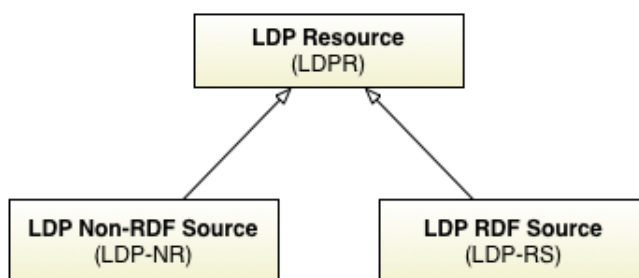
3. use RDF + SPARQL to retrieve useful information on entities
4. include links to other entities for discovery

1.1.3 Linked Data Platform

Linked Data Platform (LDP) is a W3C recommendation [9] from the 26 of February 2015, published by the Linked Data Platform Working Group⁸. It is a set of rules that applications must follow to exchange Linked Data resources.

LDP differentiates between a client and a server. They communicate using defined HTTP methods and exchange Linked Data in specific format, typically RDF. More specifically, LDP defines LDP Resource (LDPR) as a HTTP resource that conforms to the LDP patterns and conventions [10]. LDPR can be either RDF or a non-RDF resource (see Figure 1.2).

Figure 1.2: **Structure of Linked Data Platform Resources.** Original at [10].



Furthermore, LDP introduces an important concept for LDN - Linked Data Platform **Containers** (LDPC). Simply put, an LDP Container is an RDF resource where the subject is the container, the predicate is `ldp:contains`, and the object is the real data resource:

```
<LDPC URI> <ldp:contains> <document-URI>
```

Listing 1.4: **Simple LDP Container.** Example of a Linked Data Platform Basic Container in an RDF Turtle serialization format [11].

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.

<http://example.org/c1/>
  a ldp:BasicContainer;
  dcterms:title "A very simple container";
```

⁸https://www.w3.org/2012/ldp/wiki/Main_Page

1.1. Current technologies that address the centralization problem

```
ldp:contains <r1>, <r2>, <r3>.
```

The LDP Container concept is further extended in the Linked Data Notifications protocol - LDN Inbox is based on LDP Basic Container.

The following code is an example of a full LDP exchange. It represents an LDP-conformant client's request and the server's response.

Request to `http://example.org/container1/`:

Listing 1.5: **Example of LDP exchange - request.** Example of a full Linked Data Platform communication - client's request [11].

```
GET /container1/ HTTP/1.1
Host: example.org
Accept: text/turtle
Prefer: return=representation;
      include="http://www.w3.org/ns/ldp#PreferMinimalContainer"
```

And response:

Listing 1.6: **Example of LDP exchange - response.** Example of a full Linked Data Platform communication - server's response [11].

```
HTTP/1.1 200 OK
Content-Type: text/turtle
ETag: "_87e52ce291112"
Link: <http://www.w3.org/ns/ldp#DirectContainer>; rel="type",
      <http://www.w3.org/ns/ldp#Resource>; rel="type"
Accept-Post: text/turtle, application/ld+json
Allow: POST,GET,OPTIONS,HEAD
Preference-Applied: return=representation
Transfer-Encoding: chunked

@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.

<http://example.org/container1/>
  a ldp:DirectContainer;
  dcterms:title "A Linked Data Platform Container of
    Acme Resources";
  ldp:membershipResource <http://example.org/container1/>;
  ldp:hasMemberRelation ldp:member;
  ldp:insertedContentRelation ldp:MemberSubject;
  dcterms:publisher <http://acme.com/>.
```

To summarize, LDP is introducing important concepts like LDP Containers, it defines communication roles (client/server), methods (HTTP) and formats (RDF). Together it represents an important building block for further applications like LDN.

1.1.3.1 Relation of LDP to this thesis

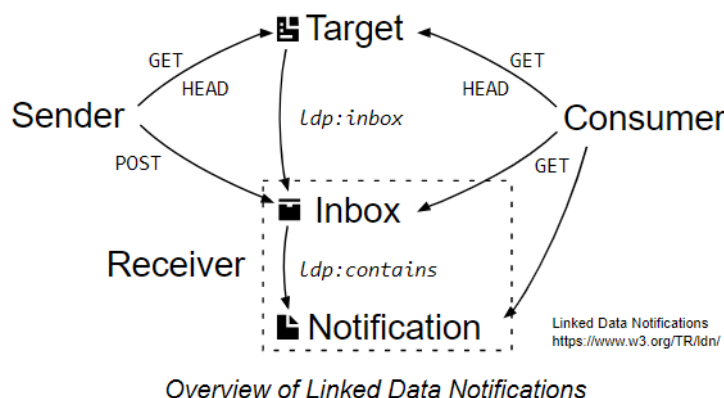
Linked Data Platform is based on RDF and it is a superset of the Linked Data Notifications (LDN). LDP concepts like Containers, its communication roles (client/server) and other specifications are essential for the technologies and concepts used in this thesis.

1.1.4 Linked Data Notifications

Linked Data Notifications (LDN) is a W3C recommendation [12] from the Social Web Working Group. It is a subset of Linked Data Platform. LDN is a protocol that specifies generic notification format for sharing between various web applications.

It defines the following roles: target (for inbox discovery [13]), server with inbox = receiver and client = consumer/sender: Figure 1.3

Figure 1.3: **Overview of Linked Data Notifications.** LDN overview with distinct roles - Consumer, Sender and Receiver. Original at [12].



Consumer and **Sender** is typically one client web application. User A sends a notification using his application (LDN sender) to the user B's inbox on his receiver application. User B can then access the notification on his receiver using his application - LDN consumer. **Inbox** is an endpoint on the Receiver, to which the notifications are sent by the Sender and from which they are being accessed by the Consumer. Furthermore, LDN specifies a Target.

Target provides a way for a user to discover another person's Inbox [13]. There are two ways to present the Inbox's URL:

- as a response to an HTTP request using the Link header with rel value `http://www.w3.org/ns/ldp#inbox`,

1.1. Current technologies that address the centralization problem

- or as a predicate `<http://www.w3.org/ns/ldp#inbox>` in an RDF graph, where the subject is the requested resource and the object is the Inbox: `<http://localhost:3000/>`
`<http://www.w3.org/ns/ldp#inbox>`
`<http://localhost:5001/API/notifications/>`

The first option is a way to present the Inbox by the server's response headers, the second way can be embedded in the content's body, e.g. RDF, JSON-LD or even embedded in the HTML (e.g. on a blog post):

1. HTTP Link

- a) HEAD > Link: `<http://example.org/inbox/>`;
`rel="http://www.w3.org/ns/ldp#inbox"`
- b) GET > Link: `<http://example.org/inbox/>`;
`rel="http://www.w3.org/ns/ldp#inbox"`

2. RDF

- a) JSON with relation of type `http://www.w3.org/ns/ldp#inbox`
- b) HTML `<a>` tag with `rel="http://www.w3.org/ns/ldp#inbox"`
- c) HTML `<link>` tag with `rel="http://www.w3.org/ns/ldp#inbox"`
- d) HTML `<section>` tag with
`property="http://www.w3.org/ns/ldp#inbox"`
- e) text/turtle with `<http://www.w3.org/ns/ldp#inbox>` relation

Simple discovery example:

Listing 1.7: **Example of LDN discovery.** Example of a Linked Data Notifications discovery of a Inbox using HTTP request to a LDN Target.

```
GET / HTTP/1.1
Accept: */*
Cache-Control: no-cache
Host: localhost:3000

HTTP/1.1 200 OK
Link: <http://localhost:5001/API/notifications/>;
      rel="http://www.w3.org/ns/ldp#inbox"
Content-Type: text/html; charset=utf-8
Content-Length: 249
Date: Sat, 18 Jul 2020 10:02:35 GMT

<!DOCTYPE html>
<html>
  ...
</html>
```

LDN also specifies the message format (RDF, preferably JSON-LD) and defines the application communication using HTTP protocol. Notifications can contain any data. The following HTTP dump is example of a simple LDN communication. The client, which is called "consumer" in LDN, sends a GET request to the Receiver to access his notifications. The server with the LDN's receiver role responds with RDF data (see Listing 1.8):

Listing 1.8: Example of LDN exchange - request and response. Example of a Linked Data Notifications communication - consumer's request and receiver's response.

```
GET /API/notifications/ HTTP/1.1
Host: localhost:5001
Accept: application/ld+json

HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Content-Type: application/ld+json; charset=utf-8
Content-Length: 390
Date: Sat, 18 Jul 2020 08:43:20 GMT
{
  "@context": "http://www.w3.org/ns/ldp#",
  "@id": "http://localhost:5001/API/notifications/",
  "@type": "ldp:Container",
  "ldp:contains": [
    {
      "@id": "http://localhost:5001/API/notifications/0"
    },
    ...
    {
      "@id": "http://localhost:5001/API/notifications/4"
    }
  ]
}
```

To summarize, LDN is a protocol for a universal notification exchange between LDN-compliant web applications. It uses RDF as data format and HTTP as communication protocol.

1.1.4.1 Relation of LDN to this thesis

Linked Data Notifications is one of the communication protocols of the resulting client application. It is used to communicate with e.g. Solid server, read user inboxes and other.

1.1.5 Activity Streams

Activity Streams 2.0 is a W3C data format specification [14]. It is basically a way of representing an activity in JSON. AS is specified with `application/activity+json` MIME media type.

Listing 1.9: **Basic AS example.** Very simple example of an Activity Streams data format.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "summary": "A note",
  "type": "Note",
  "content": "CTU FIT is awesome."
}
```

Using Activity Vocabulary⁹, AS defines entities that are necessary to represent an Activity. For example `summary`, `type` of an activity (e.g. "Like", "Create"), `actor` and others. Furthermore, it defines collections, pagination and other useful constructs. The five core objects are:

- Object
- Actor
- Activity
- Collection
- CollectionPage

In the following example, you can see an example of a `Person` adding an object of type `Article` to his blog, located at the `target: id` URL:

Listing 1.10: **Detailed AS example.** Example of an Activity Streams with additional details [14].

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "summary": "Martin added an article to his blog",
  "type": "Add",
  "published": "2015-02-10T15:04:55Z",
  "actor": {
    "type": "Person",
    "id": "http://www.test.example/martin",
    "name": "Martin Smith",
  }
}
```

⁹<https://www.w3.org/TR/activitystreams-vocabulary/>

```
"url": "http://example.org/martin",
"image": {
  "type": "Link",
  "href": "http://example.org/martin/image.jpg",
  "mediaType": "image/jpeg"
},
"object" : {
  "id": "http://www.test.example/blog/abc123/xyz",
  "type": "Article",
  "url": "http://example.org/blog/2011/02/entry",
  "name": "Why I love Activity Streams"
},
"target" : {
  "id": "http://example.org/blog/",
  "type": "OrderedCollection",
  "name": "Martin's Blog"
}
}
```

Activity Streams is used as the data format of the protocol ActivityPub.

1.1.5.1 Relation of AS to this thesis

Activity Streams data format is the base of the ActivityPub (AP) protocol. It defines the core objects of AP and is essential to understand the communication between federated AP applications.

1.1.6 ActivityPub

ActivityPub¹⁰ is a protocol for decentralized social networks, which can also be extended to create all kinds of federated apps [15]. It is a W3C recommendation that provides two kinds of API:

- **C2S API** - client-server protocol for AP clients for creating, updating and deleting content;
- **S2S API (server-server, federation protocol)** for delivering notifications and content between AP applications.

ActivityPub application can implement only one or both of them, based on the application's scope. Based on the implementation, we differentiate three kinds of AP applications (three "conformance classes" based on the AP specification):

- **C2S Client** - a client application that implements C2S API (= ActivityPub conformant Client), e.g. mobile application that connects to a AP server.

¹⁰<https://www.w3.org/TR/activitypub/>

1.1. Current technologies that address the centralization problem

- **C2S Server** - a server that implements C2S API (= ActivityPub conformant Server), e.g. web server that allows mobile clients to read user data.
- **S2S Server** - a server that implements S2S API (= ActivityPub conformant Federated Server), e.g. server that connects to another server (like the whole <https://fediverse.party/>).

AP uses Activity Streams 2.0¹¹ as its data format. It adds couple of extra requirements. Only `Link` and `Object` entities are allowed. In the `Object` entity, `id` and `type` fields are required [15]. Additionally, the *Actor* object must have `inbox` and `outbox`. An `inbox` is a collection of all messages received by the Actor. Similarly, an `outbox` is a collection of messages produced by the Actor.

An *Actor* is not only a person/human user, but it can be any fitting object, such as a company, a website, software, city and others. Typically, it is one of the AC core types:

- Application
- Group
- Organization
- Person
- Service

However, it can also be of another type, made with ActivityStreams extension¹² (= type not defined by the Activity Vocabulary¹³). Furthermore, ActivityPub extends AS addressing with `to`, `bto`, `cc`, `bcc` and `audience` fields:

Listing 1.11: AP example - Submitting an Activity to the Outbox. Example of an ActivityPub Like with additional details [14].

```
POST /outbox/ HTTP/1.1
Host: dustycloud.org
Authorization: Bearer XXXXXXXXXXXX
Content-Type: application/ld+json; profile="https://www.w3.org/ns/
    ↪ activitystreams"

{
  "@context": ["https://www.w3.org/ns/activitystreams",
    {"@language": "en"}],
  "type": "Like",
  "actor": "https://dustycloud.org/chris/",
```

¹¹<https://www.w3.org/TR/activitystreams-core/>

¹²<https://www.w3.org/TR/activitystreams-core/#extensibility>

¹³<https://www.w3.org/TR/activitystreams-vocabulary/>

```
"name": "Chris liked 'Minimal ActivityPub update client'",
"object": "https://rhiaro.co.uk/2016/05/minimal-activitypub",
"to": ["https://rhiaro.co.uk/#amy",
       "https://dustycloud.org/followers",
       "https://rhiaro.co.uk/followers/"],
"cc": "https://e14n.com/evan"
}
```

1.1.6.1 Relation of AP to this thesis

ActivityPub is one of the communication protocols of the resulting client application. It is used to communicate with e.g. Pleroma social network. The whole Fediverse federated network¹⁴ is based on this protocol.

1.1.7 Solid

Solid is a set of open specifications, built on existing open standards like LDN and RDF, that describes how to build applications in such a way that users can conveniently switch between data storage providers and application providers. [16]

1.1.7.1 WebID

A WebID is a unique identifier of an agent (e.g. user, organization). It is an Internationalised Resource Identifier (IRI) and can be dereferenced as a FOAF profile document [17]. An example is <https://tonda.solid.community/profile/card#me>.

The owner can set sharing preferences of his WebID to the WebID of third parties [17]. In Solid, WebIDs are also used to manage access rights through Web Access Control [17].

1.1.7.2 Pod

"A Pod is where data is stored on the Web with Solid. A user may store their data in one Pod or several Pods, and applications read and write data into the Pod depending on the authorisations granted by the user or users associated to that Pod." [17] (see Figure 1.4).

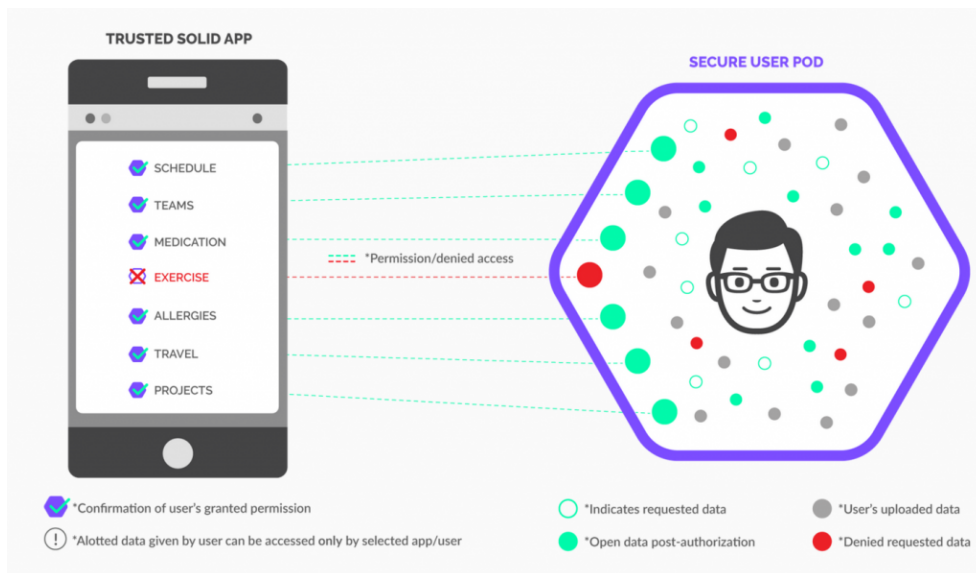
1.1.7.3 Relation of Solid to this thesis

Solid is the main server that the resulting client application is intended to communicate with. It is used as the data storage for the user inboxes.

¹⁴<https://fediverse.party/>

1.1. Current technologies that address the centralization problem

Figure 1.4: **Illustration of a solid pod with application.** An illustration of interaction of a web/mobile application with user's solid pod. Original at [18].



Analysis

This chapter deals with the analysis of the topics relevant to the thesis. First, based on the Linked Data Notifications and ActivityPub protocols, the system actors are identified. Second, the system requirements are specified, distinguishing between functional and non-functional ones. Third, use cases are derived to further specify the desired system behavior.

With this system specification, an analysis of existing applications was conducted in two phases. The first phase is a broad search for existing solutions with quick analysis to determine whether the application meets the basic requirements (sources, documentation are available) and is applicable for a more detailed analysis. The second phase goes into details of the system and studies if the application not only meets desired requirements but can support all the use cases.

Lastly, the analysis result is presented.

2.1 Requirements

First, requirements and use cases were specified for use in further analysis. Requirements cover required functionality on an abstract level. They are constructed based on the required technologies and represent boundaries of the system.

2.1.1 Actors

User roles can be divided into two roles as defined in LDN: consumer and sender (see Figure 1.3).

2.1.1.1 Consumer

Consumer is a person who can access his inbox and notifications. He understands what LDN and AP are and he wants to try communication using these

protocols.

2.1.1.2 Sender

Sender is a person who can post notifications to other people's inboxes. He understands what LDN and AP are and he wants to try communication using these protocols.

2.1.2 List of requirements

In this section, a list of requirements is presented, divided into functional and non-functional requirements.

2.1.2.1 Functional requirements

List of functional requirements with short descriptions.

- F1 Support LDN.** Application supports communication using Linked Data Notifications protocol.
- F2 Support AP.** Application supports communication using ActivityPub protocol.
- F3 Support LDP inbox monitoring.** Application supports monitoring of Linked Data Protocol inboxes that the user has access to.
- F4 Support AP inbox monitoring.** Application supports monitoring of ActivityPub inboxes that the user has access to.
- F5 Support JSON-LD.** Application is able to exchange data with another application using JSON-LD.
- F6 Support system notifications.** Application supports system notifications, such as pop-up information on incoming message to the monitored LDP inbox.
- F7 Support WebID login.** Users can authorize and authenticate using WebID ¹⁵.
- F8 List of incoming messages.** Application can show list of incoming messages.
- F9 List of sent messages.** Application can show list of sent messages.
- F10 Show message content.** Application can show message content.
- F11 List of contacts.** Application can show list of user contacts.
- F12 Show contact detail.** Application can show contact detail.

¹⁵<https://www.w3.org/wiki/WebID>

2.1.2.2 Non-functional requirements

List of non-functional requirements with short descriptions.

N1 Web application. System is implemented as a web application.

N2 Git versioning. Application sources are versioned using Git VCS ¹⁶, publicly hosted on GitHub¹⁷.

N3 Security. User can access only messages he has access to.

2.2 Use cases

A use case is a description of the specific ways a user interacts with a system. Use cases are a more specific view of system requirements.

List of use cases is divided into two parts based on actors: consumer and sender.

2.2.1 List of use cases - consumer

UC1 Start monitoring inbox. Consumer sets application so it monitors an LDP inbox he has access to.

- a) User logs in using WebID.
- b) User clicks on action "add inbox for monitoring".
- c) System shows form to add inbox.
- d) User inputs IRI of a resource and submits.
- e) System discovers resource's inbox.
- f) System starts monitoring messages coming to the inbox.

UC2 Stop monitoring inbox. User can turn off monitoring of an inbox he has previously selected for monitoring.

- a) User logs in using WebID.
- b) System shows list of monitored inboxes.
- c) User chooses inbox to stop being monitored.
- d) System stops monitoring incoming messages to the chosen inbox.

UC3 Read list of all messages. Consumer can read a list of all incoming messages that he has access to in all monitored inboxes.

- a) User logs in using WebID.

¹⁶<https://git-scm.com/>

¹⁷<https://github.com/>

2. ANALYSIS

- b) System shows list of incoming messages.

UC4 Read list of messages from selected inbox. Consumer can read a list of incoming messages that he has access to in a selected inbox.

- a) User logs in using WebID.
- b) System shows list of monitored inboxes.
- c) User chooses inbox.
- d) System shows list of incoming messages.

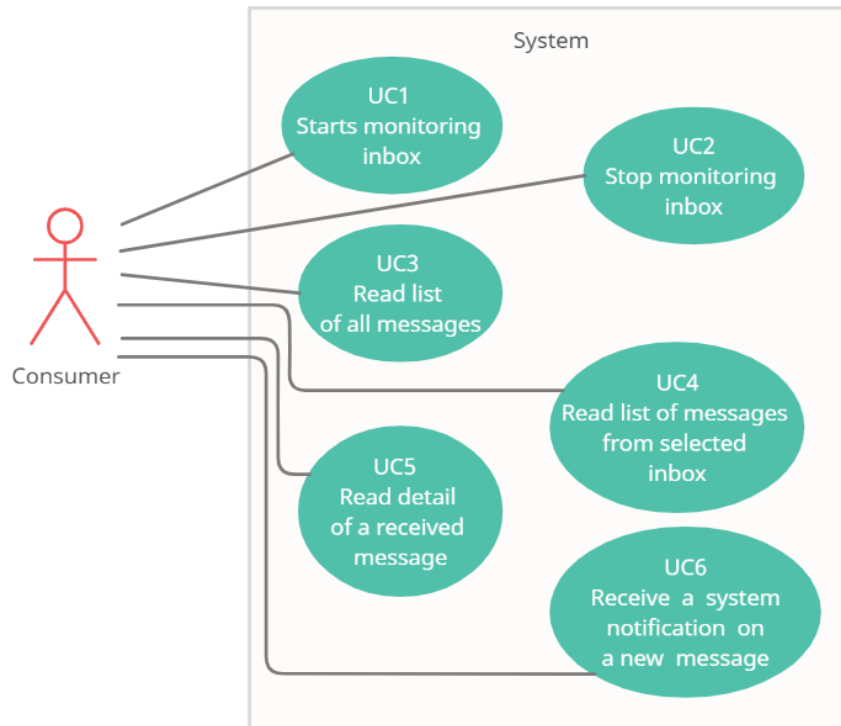
UC5 Read detail of a received message. Consumer can read the complete content of a received message that he has access to.

- a) User logs in using WebID.
- b) User sees list of his incoming messages.
- c) User can open and read the full content of the incoming message.

UC6 Receive a system notification on new message. Consumer gets a system notification, such as pop-up message, when he receives a new message to one of his monitored inboxes.

- a) User logs in using WebID.
- b) System receives a message for the user to a monitored inbox.
- c) System shows a pop-up system notification to the user.

Figure 2.1: UML diagram of Consumer use cases



2.2.2 List of use cases - sender

UC7 Send a message to a person/resource from contact list using LDN. Sender sends a message to an actor from contact list using Linked Data Notifications protocol. The actor can be a person or some other resource (like company, website, etc.) identified by IRI.

- a) User logs in using WebID.
- b) User sees list of his contacts.
- c) User clicks on the action "send message".
- d) User enters a content of the message.
- e) User submits the message.
- f) System sends the message to the resource's inbox.

UC8 Send a message to a person/resource from contact list using AP. Sender sends a message to an actor from contact list using ActivityPub protocol. The actor can be a person or some other resource (like company, website, etc.) identified by IRI.

- a) User logs in using WebID.

2. ANALYSIS

- b) User sees list of his contacts.
- c) User clicks on the action "send message".
- d) User enters a content of the message.
- e) User submits the message.
- f) System sends the message to the resource's inbox.

UC9 Send a message to an unknown person/resource using LDN.

Sender sends a message to an actor that is not in his contact list using LDN protocol. The actor can be a person or some other resource (like company, website, etc.) identified by IRI.

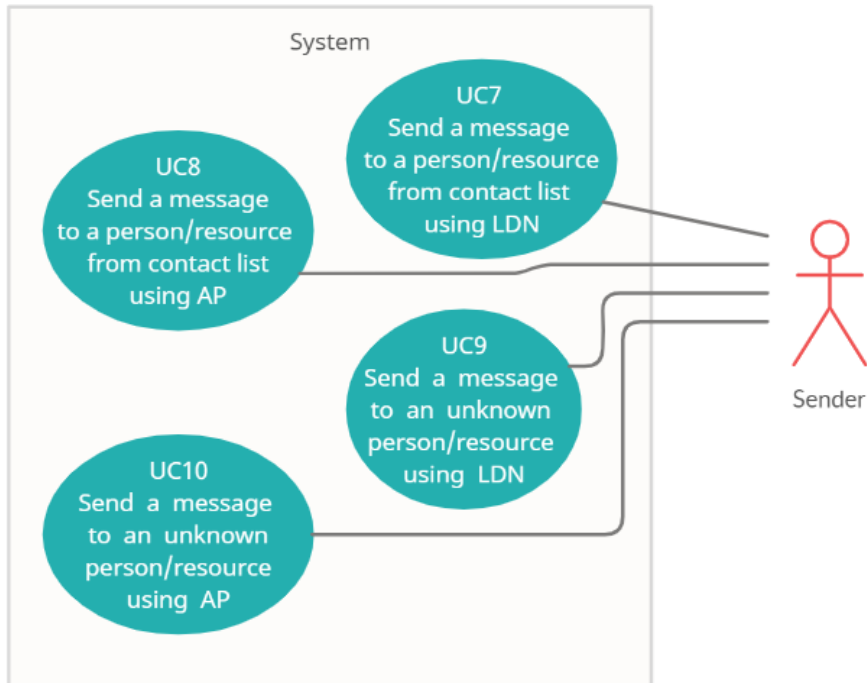
- a) User logs in using WebID.
- b) User opens a new message form.
- c) User enters the receiver's IRI.
- d) User enters a content of the message.
- e) User submits the message.
- f) System sends the message to the resource's inbox.

UC10 Send a message to an unknown person/resource using AP.

Sender sends a message to an actor that is not in his contact list using AP protocol. The actor can be a person or some other resource (like company, website, etc.) identified by IRI.

- a) User logs in using WebID.
- b) User opens a new message form.
- c) User enters the receiver's IRI.
- d) User enters a content of the message.
- e) User submits the message.
- f) System sends the message to the resource's inbox.

Figure 2.2: UML diagram of Sender use cases



2.3 Analysis of existing solutions - LDN clients and Solid servers

An analysis of existing LDN clients and Solid server implementations was conducted to determine whether they could be used for the thesis goal (a client and a server supporting decentralized messaging). Applications were searched on the web using Google with terms like "ldn inbox", "solid server" and similar. Also, existing implementations linked at "LDN Test Reports and Summary" <https://linkedresearch.org/ldn/tests/summary> and <https://solidproject.org/use-solid/apps> were examined.

2.3.1 Criteria for analysis of existing solutions

The following criteria were considered:

- application has available sources;
- license permits extending the application;
- application has sufficient documentation on how to run it, or it is runnable without the need for documentation;

2. ANALYSIS

- application is live or with active development - sources without a commit within 1 year were excluded;
- application support for linked data, LDP, LDN, AP, RDF and system notification;
- application must be extensible with our cause - LDN/AP notifications. Applications without the possibility of being extended with our use cases were excluded.

Based on these criteria, three applications were selected for more detailed analysis (see section Detailed analysis below).

2.3.2 Overview of all analysed applications - LDN

This section contains list of all analysed applications with short description, result of the analysis and link for access.

2.3.2.1 SCTA inbox receiver

Only a simple LDN app to pass the LDN test suite¹⁸. No information available on how to run this. No extension possible.

Accessible at <https://github.com/scta/scta-inbox>.

2.3.2.2 Sloph/DIY Inbox

Sample/POC LDN demonstration in PHP. No extension possible. Made only to pass the LDN test suite¹⁹.

Accessible at <https://rhiaro.co.uk/2017/08/diy-ldn>.

2.3.2.3 gold

Reference Linked Data Platform server for the Solid platform. Not maintained anymore - latest commit on Oct 10, 2018.

Accessible at <https://github.com/linkedata/gold>.

2.3.2.4 ldn-streams

Implementation of Linked Data Notifications for RDF streams. Not maintained anymore - latest commit on Jun 11, 2018.

Accessible at <https://github.com/jpcik/ldn-streams>.

¹⁸LDN test suite, available at <https://linkedresearch.org/ldn/tests/>

¹⁹see footnote above

2.3.2.5 Carbon LDP

Carbon LDP is "an enterprise-class Linked Data Platform that helps artists and engineers create and extend web applications with ease" [19]. Based on its broad capabilities, it has been selected for a detailed analysis, see below.

Accessible at <https://carbonldp.com/>, <https://github.com/CarbonLDP>.

2.3.2.6 solid-server in Node

Solid-server is the main candidate for the possible solution. It is implemented in NodeJS on top of the file system. Solid supports Linked Data Platform, Web Access Control, WebID+TLS Authentication, real-time live updates (using WebSockets) and other features. It is available both as a NodeJS project and as a Docker container. Detailed analysis is available below.

Accessible at <https://github.com/solid/node-solid-server>.

2.3.2.7 pyldn

Pyldn is a lightweight Linked Data Notifications (LDN) receiver implemented in python. As it does not include other LDN parts, it was not selected for further analysis.

Accessible at <https://github.com/albertmeronyo/pyldn>.

2.3.2.8 Virtuoso Universal Server

Commercial Data Virtualization platform. Sources not available.

Accessible at <https://virtuoso.openlinksw.com/#this>.

2.3.2.9 maytkso

"HTTP server and command-line RDF tool to get/send, serialise data." [20] Written in one JS file, merely an LDN server/receiver. Not acceptable for extension.

Accessible at <https://github.com/csarven/maytkso>.

2.3.2.10 Apache Marmotta

Open platform for LDP, implemented as a Java web application. Has to be run on a JavaEE application server. Based on its capabilities, it has been selected for a detailed analysis below.

Accessible at <https://marmotta.apache.org/>.

2.3.2.11 IndieAnndroid/ blog-a-loosh

IndieAnndroid is a Indieweb blog platform. It has been developed for personal use and not applicable for our purpose.

Accessible at <https://github.com/Kongaloosh/blog-a-loosh>.

2.3.2.12 LDP-CoAP

LDP for the Constrained Application Protocol. Provided mapping of LDP over HTTP to CoAP (RFC 7252 Constrained Application Protocol [21]). Not applicable for our purpose.

Accessible at <http://sisinflab.poliba.it/swottools/ldp-coap/>, <https://github.com/sisinflab-swot/ldp-coap-framework>.

2.3.2.13 distbin.com

Application similar to pastebin - for easy copy, paste and share of text. Not useful for messaging. Not available anymore.

Accessible at <https://distbin.com/>.

2.3.2.14 Fedora Repository

Big repository system for management and dissemination of digital content (digital libraries and archives). It does not support use cases such as decentralized messaging for a common user.

Accessible at <https://wiki.duraspace.org/display/FF/Fedora+Repository+Home>.

2.3.2.15 SNS

SNS is a social network based on Solid. It is built using JavaScript. No sources available, so the project cannot be used for this thesis.

Accessible at <https://electrapro-pk.github.io/SNS/>.

2.3.2.16 Solidarity

Chat application written in node.js. It is an online chat with channels. No license information available.

Accessible at <https://github.com/scenaristeur/solidarity>, <https://scenaristeur.github.io/solidarity/>.

2.3.2.17 OChat

Simple chat application written with React. Not maintained anymore (last commit on Jul 25, 2019).

Accessible at <https://github.com/jaxoncreed/o-chat>, <https://chat.o.team/>.

2.3.2.18 Friend Requests Exploration

Exploration into how Solid could be used for sending friend requests. Not extensible.

Accessible at <https://github.com/inrupt/friend-requests-exploration>.

2.3.2.19 solid-inbox

Inbox for processing notifications. It is just a single JavaScript file application. It is not maintained anymore.

Accessible at <https://github.com/solid/solid-inbox>.

2.3.3 Detailed analysis of selected applications

Applications selected in the previous analysis were subjected to a more detailed analysis based on the requirements and support of possible use cases. Table 2.1 is an overview of the analysis, details follow below.

Table 2.1: **Detailed analysis.** This table shows application support of requirements and use cases. Only applications selected for detailed analysis are shown. See requirements section and use cases section for details. ✓ means full support, ✗ means no support, - means that the support could not be verified or that it is not applicable in the application’s scope.

requirement/ use-case ID	application name	
	solid-server	Apache Marmotta
F1	✓	✓
F2	✓	✗
F3	✓	✓
F4	✓	✗
F5	✓	✓
F6	✓	-
F7	✓	✗
F8	✓	-
F9	✓	-
F10	✓	-
F11	✓	-
F12	✓	-
N1	✓	✓
N2	✓	✓
N3	✓	✓
UC1	✓	-
UC2	✓	-
UC3	✓	-
UC4	✓	-
UC5	✓	-
UC6	✓	-
UC7	✓	-
UC8	✓	-
UC9	✓	-

2.3.3.1 solid-server

Solid is a project led by Prof. Tim Berners-Lee, inventor of the World Wide Web, taking place at MIT [22]. Solid-server is a server implemented in NodeJS on top of the file system. It supports Linked Data Platform, Web Access Control, WebID+TLS Authentication, real-time live updates (using WebSockets) and other features. It is available both as a NodeJS project and as a Docker container.

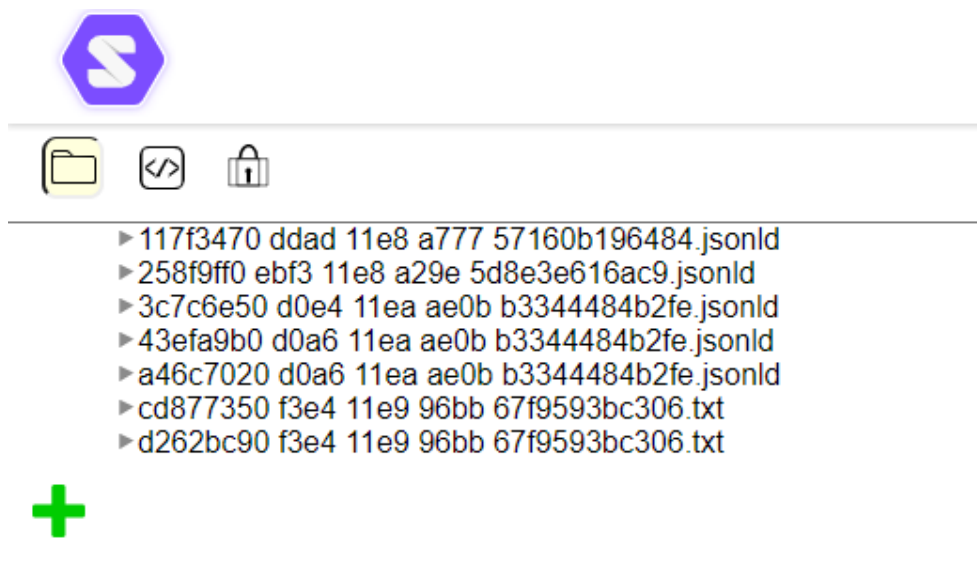
Solid-server is still a developing prototype. The main instance is running at <https://solid.community/>, however it is designed so anybody can host their instance. It is published under the free MIT license [23], so it is available for extension.

Source code and documentation is available at <https://github.com/solid/node-solid-server>.

2.3.3.2 Solid inbox client

As a part of the public Solid server instance at <https://solid.community/>, there is an existing inbox client application. The problem is that its user interface (UI) is very un-intuitive and cumbersome. Figure 2.3 is the UI for the use case UC4 - read a list of inbox messages:

Figure 2.3: **Solid inbox - list of messages**. Screenshot of the current official inbox client application.



The following two screenshots Figure 2.4 and Figure 2.5 capture the solid UI for message detail - UC4.

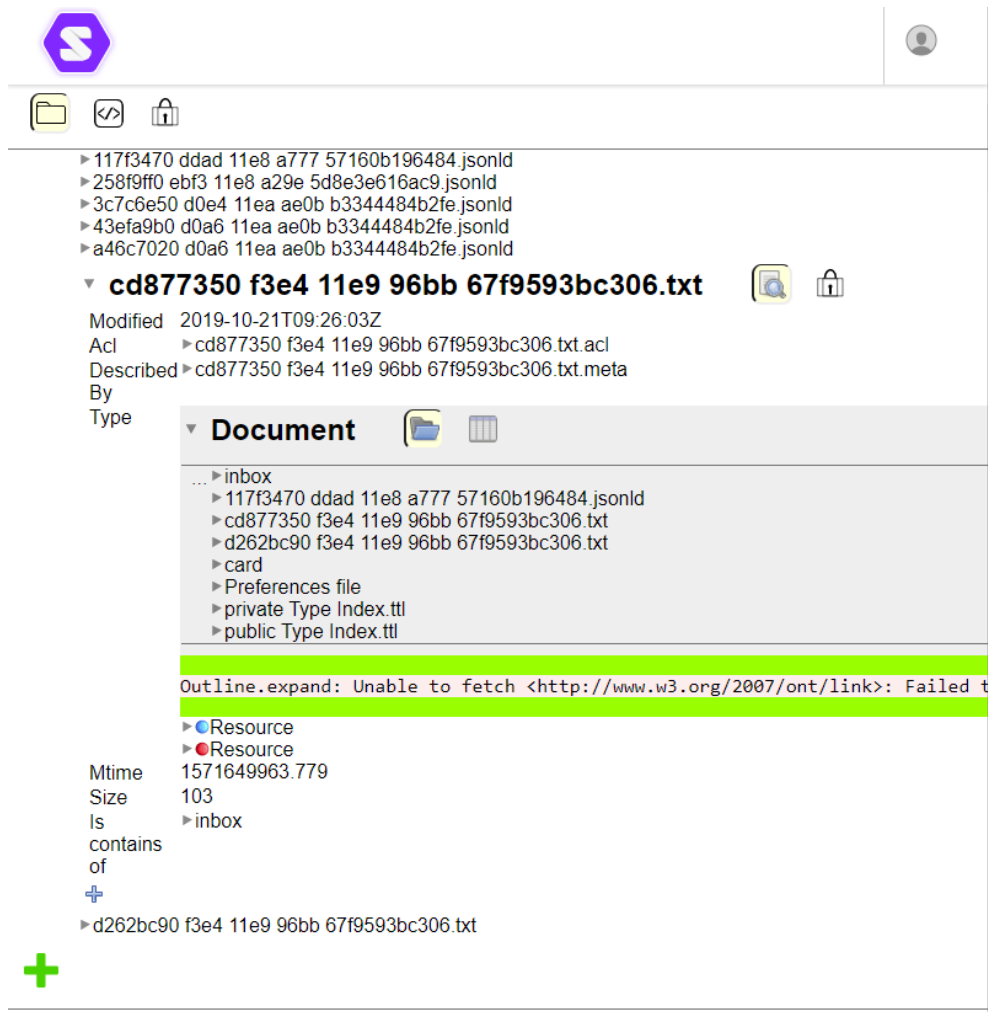
2. ANALYSIS

Figure 2.4: **Solid inbox - message detail**. Screenshot of the current official inbox client application.



As you can see, the current UI makes it impossible to access the detail content and the user is forced to use another solution (such as HTTP GET the RDF document representation).

Figure 2.5: **Solid inbox - message content.** Screenshot of the current official inbox client application.



2.3.3.3 Apache Marmotta

Apache Marmotta is an Open Platform for Linked Data [24]. Based on the Linked Media Framework project, it is an implementation of a Linked Data Platform. It is implemented as a Java Web Application [24].

Marmotta features Linked Data server for the Java EE stack, LDP, SPARQL and LDPPath querying, caching and basic security mechanisms. The installation comes as a Java Web Archive (.war) file that has to be deployed on an application server (such as Tomcat) [24].

Marmotta is a robust, well-documented platform. However, based on the version list [25] and issue tracker [26], it does not appear to be under active development. The last version was published in June 2018 [25] and there are unresolved open issues from 2018 [26].

Apache Marmotta is published under the open-source Apache Software License, Version 2.0: <https://marmotta.apache.org/license.html>.

2.3.3.4 Carbon LDP

Carbon LDP is an enterprise server implementation of LDP. It aims to help engineers and artists to create web applications supporting LDP. It provides R/W access to RDF graph data using RESTful HTTP. Homepage: <https://carbonldp.com/>, GitHub page: <https://github.com/CarbonLDP>.

Carbon LDP supports REST API requests over HTTP. It supports JSON-LD, Turtle and RDF XML serializations. Data are stored in native RDF format - RDF triples in a NoSQL database. It also supports querying documents using SPARQL.

At first glance, Carbon LDP appears to be open source with commercial support (like e.g. Spring framework²⁰). However, after a more thorough analysis Carbon LDP emerges as **a commercial product** without sources available and without an option for extensions. It does contain open source modules, such as a graph/document visualisation tool Workbench²¹ and JavaScript SDK²². But **these modules do not meet requirements** for this theses and thus Carbon LDP is disqualified from further use.

2.4 Analysis of existing solutions - ActivityPub applications

This section deals with the analysis of existing ActivityPub applications. The main goal was to find an ActivityPub C2S Server (a server that supports AP client API - C2S protocol, or "Social API", see subsection 1.1.6 for details).

²⁰<https://spring.io/>

²¹<https://github.com/CarbonLDP/carbonldp-workbench>

²²<https://github.com/CarbonLDP/carbonldp-js-sdk>

As a part of this analysis, also a search for existing clients was conducted. Results are available in the Table 2.2.

The analysis is mostly based on the lists of ActivityPub implementations available at <https://activitypub.rocks/implementation-report/> and <https://en.wikipedia.org/wiki/ActivityPub>.

2.4.1 Criteria for analysis of existing AP solutions

The following criteria were considered:

- application has available sources;
- license permits extending the application;
- application has sufficient documentation on how to run it, or it is runnable without the need for documentation;
- application is live or with active development - sources without a commit within 1 year were excluded;
- **server supports ActivityPub client to server API**

2.4.2 Overview of analysed servers - AP

This section contains a table with the analysis result for a quick overview and a list of analysed applications with short description, result of the analysis and link for access.

Table 2.2: **AP API support.** This table shows application support of ActivityPub API. The table columns (C2S Client, C2S Server, S2S Server) are based on the subsection 1.1.6.

application name	ActivityPub API part		
	C2S Client	C2S Server	S2S Server
distbin.com	(✓) ²³	(✓)	(✓)
Go-Fed	✗	✓	✓
Express ActivityPub Server	✗	✗	✓
Mastodon	✗	✗	✓
Pleroma	✗	✓	✓
AndStatus	✓	✗	✗

²³distbin only supports parts of AP API

2.4.2.1 distbin.com

Distbin is a distributed service similar to a more widely known application pastebin²⁴ - users can paste any text and share it using URL. According to the activitypub.rocks implementation report, Distbin is supporting C2S API. However no public instance of this service is available, the proclaimed website <http://distbin.com/> is not running.

Sources are available at <https://github.com/gobengo/distbin>.

2.4.2.2 Go-Fed

Go-Fed is a suite of libraries for writing Fediverse applications. It supports both CS2 and S2S API. As its name suggests, Go-Fed is written in the "Go" programming language²⁵. This exotic language, together with the lack of publicly running servers was the main reason not to use Go-Fed as a server for our solution.

Accessible at <https://go-fed.org/>.

2.4.2.3 Express ActivityPub Server

Express ActivityPub Server is a simple ActivityPub implementation written in Node.js. It was created as a sample implementation and is not intended for production usage. No public instances of this server are available.

Accessible at <https://github.com/dariusk/express-activitypub/>.

2.4.2.4 Mastodon

Mastodon is a social network that was built in support of web re-decentralization. It is an open-source federated network and it is using ActivityPub as the federation protocol. User can choose an existing network with various topics or create his own instance. Mastodon instances can be part of Fediverse²⁶.

Accessible at <https://joinmastodon.org/>.

2.4.2.5 Pleroma

Pleroma is an open-source social network. It is a fork of the Mastodon project with aim for lightweight devices, such as a Pi computer. Pleroma has also capabilities to be part of the Fediverse²⁷.

²⁴<https://pastebin.com/>

²⁵<https://golang.org/>

²⁶<https://fediverse.party/>

²⁷<https://fediverse.party/>

Accessible at <https://pleroma.social/>.

2.4.2.6 AndStatus

AndStatus is an open-source mobile application that works as a client for various social networks including Mastodon, Twitter, ActivityPub (Client to Server), GNU social and Pump.io. It also implements the C2S AP API.

AndStatus is written in Java and runs on the Android mobile platform. It is not a web-based application and thus is not usable for the purpose of this thesis.

Accessible at <https://github.com/andstatus/andstatus>.

2.4.3 AP servers analysis result

Pleroma was chosen as the application server because it is the only ActivityPub implementation that meets all criteria - supports CS2 API, has running public servers available and is open-source.

2.5 Implementation analysis

As a part of this thesis, before designing the final application, several proof-of-concept applications were created in order to explore the W3C recommendations and protocols:

1. **inbox - LDN proof-of-concept** - web application created to investigate the LDN protocol, test the architecture, technologies and the LDN, RDF libraries.
2. **LDN-target** - simple web application separated from the original inbox POC in order to document how to use the LDN target.
3. **js-notification-poc** - implementation of the JavaScript Notification API and Push API. Its development was intended to get familiar with the specifications and test the APIs.

Detailed description of the applications is available at Appendix A. Based on this analysis, `solid-client-authn-js`²⁸ was chosen as the only usable library for WebID authentication. Similarly, `solid-client`²⁹ was chosen as the best library for connecting to the Solid POD.

²⁸<https://github.com/inrupt/solid-client-authn-js>

²⁹<https://github.com/inrupt/solid-client-js>

2.6 Analysis results

In the analysis chapter, first, requirements for the solution were formulated. Use cases were derived from requirements to specify user actions. Based on the requirements and use cases, a broad analysis of existing solutions was conducted.

Only solid-server and Apache Marmotta matched the criteria of the first research for Linked Data Notifications (LDN) solution and were selected for a more detailed review.

After a more detailed analysis, the **solid-server was selected as a sufficient server** solution to support the LDN client part of the thesis. There is no need to implement a new server solution.

It was determined that there is **no usable messaging client supporting LDN protocol and a new one should be implemented.**

Based on the implementation analysis, the **solid-client-authn-js and solid-client libraries were selected** for client implementation.

The research for ActivityPub (AP) clients and servers concluded that only **Pleroma** supports the client-to-server (C2S) API and can be used as the AP server. Apart from the mobile application AndStatus, **no web client implementing AP C2S API has been found.** Thus it was determined that a **new client should be implemented.**

Design

First, a technical research, three proof-of-concept applications (POC) were developed. Then, based on these applications, the final client application Inbox was implemented. This chapter describes design of the final Inbox application. For details about the POC applications, see Appendix A.

3.1 System architecture: in-browser web application + server back-end

The goal of this thesis is to "implement a client and a server supporting decentralized messaging on the Web". Based on this requirement, the client needs to be a web application, meaning user can access its interface in a web browser. It connects to a server where the user data is stored.

3.2 Client

In this section, the Inbox client application's design is introduced.

3.2.1 Programming languages and frameworks

Open-source web application framework **Angular**³⁰ was chosen as the application base. It allows to create modern web applications and take advantage of the existing solid libraries for authentication and communication with Solid server.

Angular Material³¹ components were chosen as the best fit for developing user interface.

³⁰<https://angular.io/>

³¹<https://material.angular.io/>

Both Angular framework and Material components are written in **TypeScript**. TypeScript is a language that allows type-safe programming and direct compilation to JavaScript.

3.2.2 Application architecture: MVC

As the application is written in Angular, it takes advantage of the Model-View-Controller architecture. The Model is represented by entities and services. The View is represented by HTML templates with CSS styles sheets. The Controller is represented by components.

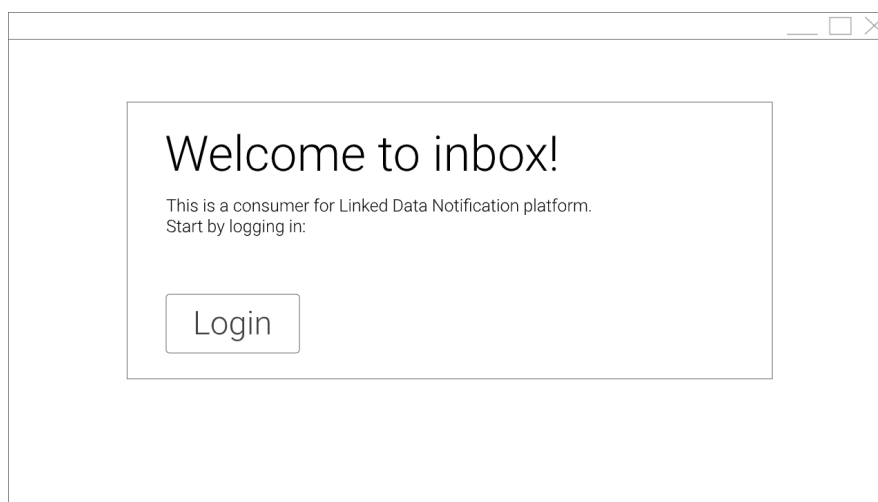
More details on the model are provided in the application LDN and AP parts below.

3.2.3 User interface design - wireframes

User interface was designed using wireframes - a low-fidelity prototypes of application web pages. The web pages are based on the use cases. The wireframes were created using free online tool Photopea³².

Please note that the solid.community login pop-up windows in the screens 2 and 3 are screenshots of actual 3rd party component.

Figure 3.1: Screen 1 - welcome page.



³²<https://www.photopea.com/>

Figure 3.2: Screen 2 - login using Solid authentication - step 1.

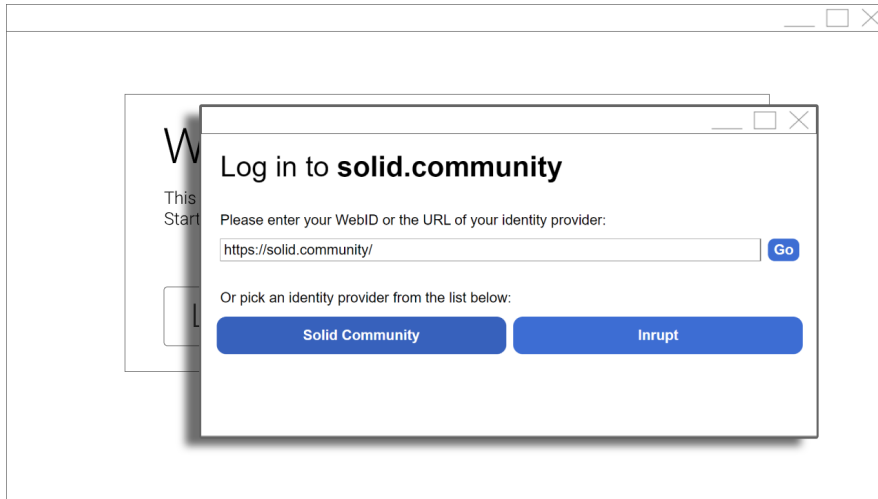
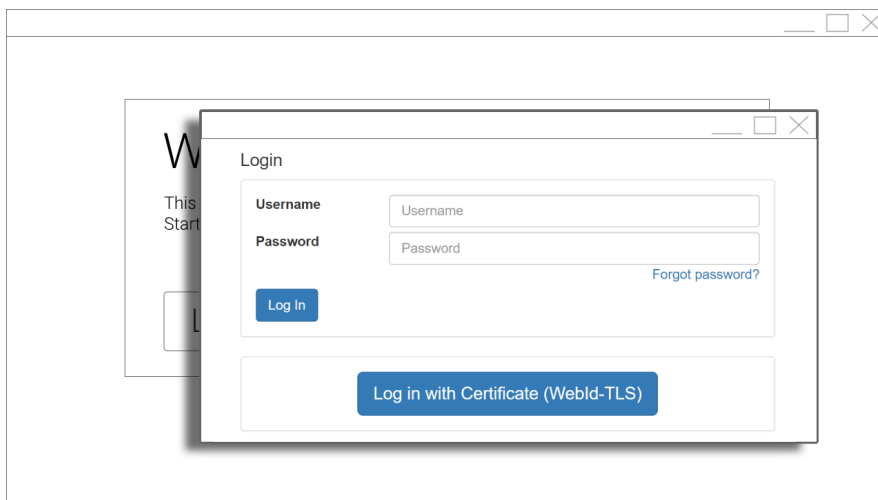


Figure 3.3: Screen 3 - login using Solid authentication - step 2.



3. DESIGN

Figure 3.4: **Screen 4 - start monitoring inbox - UC1.**

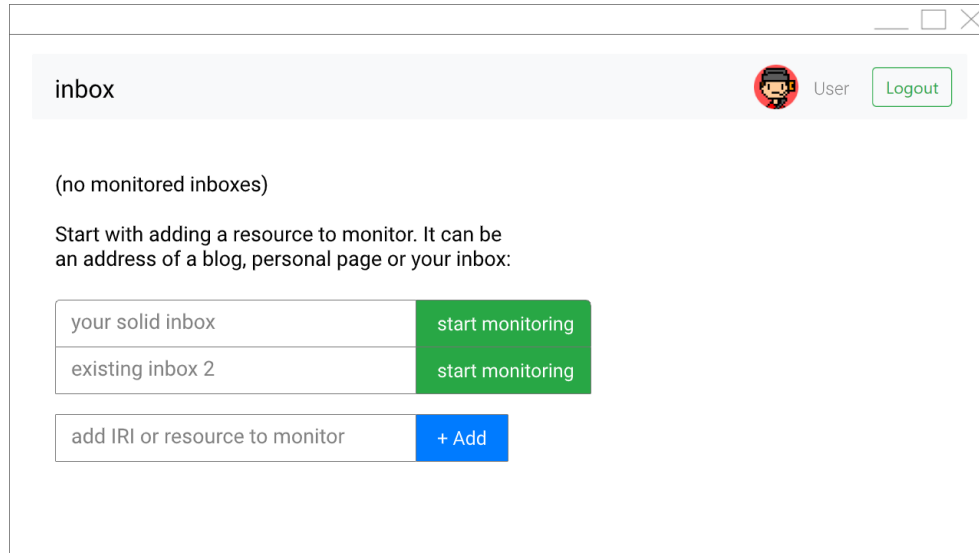


Figure 3.5: **Screen 5 - stop monitoring inbox - UC2.**

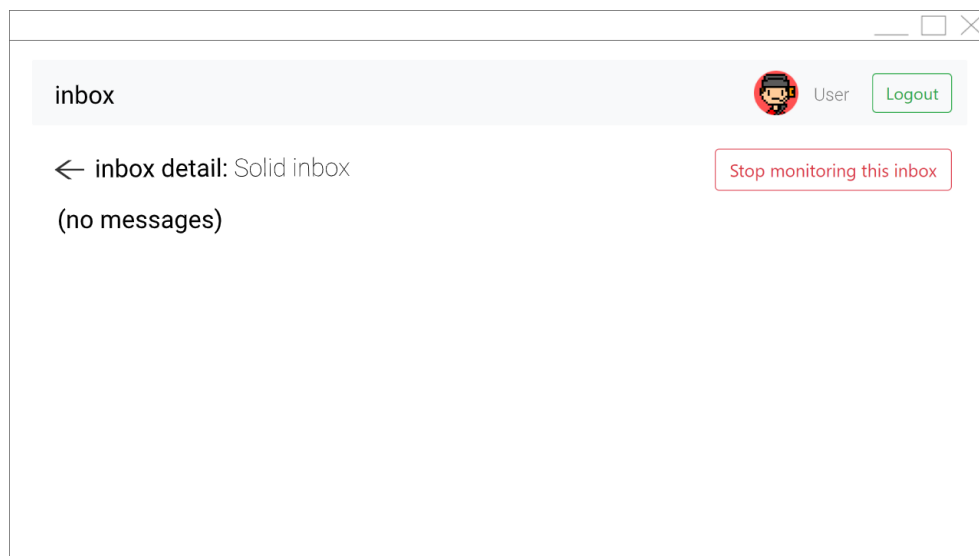


Figure 3.6: **Screen 6 - Read list of all messages - UC3, default screen.** All messages from all monitored inboxes combined. This is the main screen user will see when he logs in.

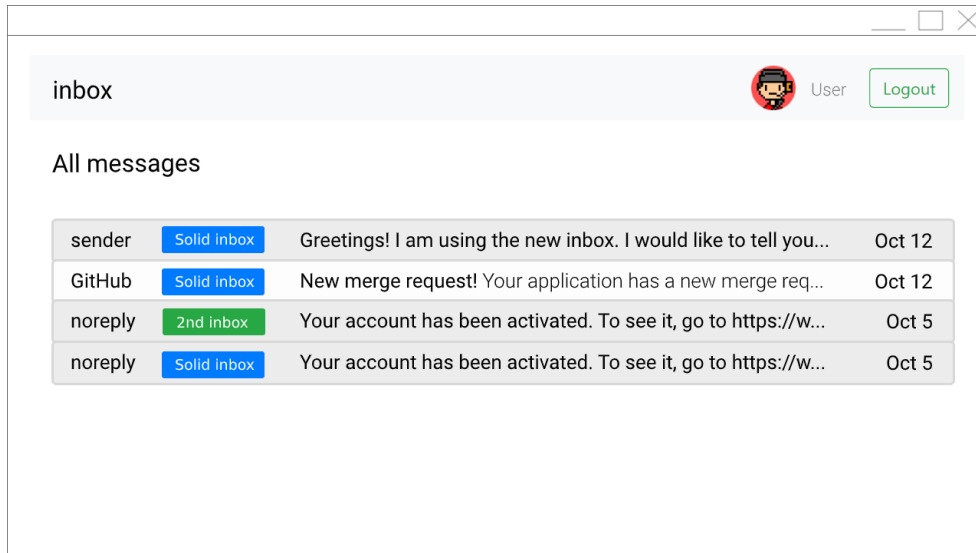
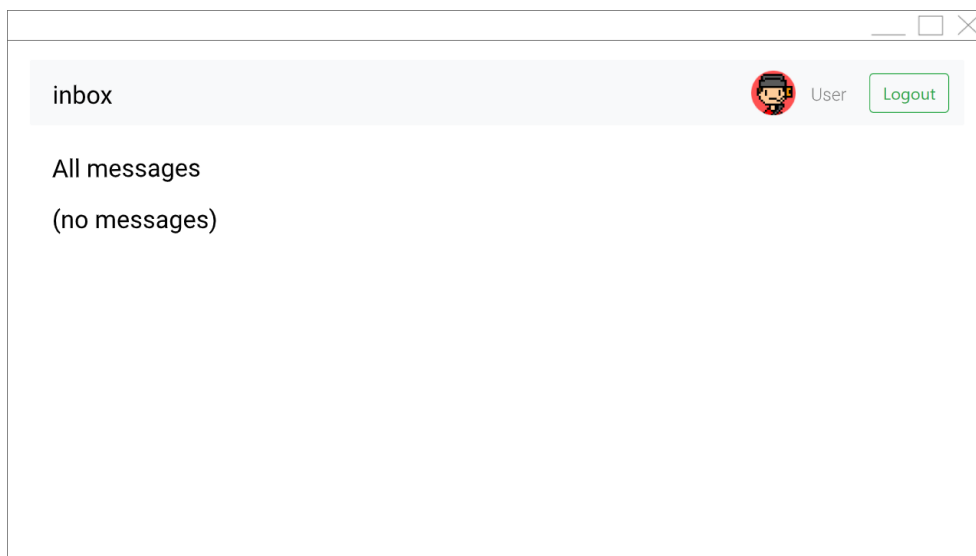


Figure 3.7: **Screen 7 - Empty list of all messages.** All messages view, when no messages are available.



3. DESIGN

Figure 3.8: **Screen 8 - Read list of messages from one inbox.** All messages from chosen inbox.

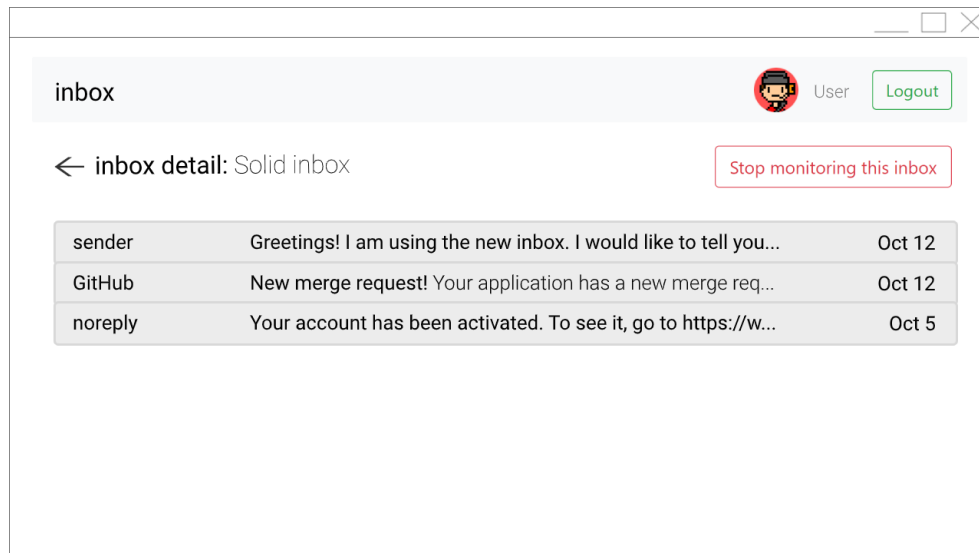


Figure 3.9: **Screen 9 - Read list of messages from one inbox - empty.** Detail of inbox, when there are no messages available.

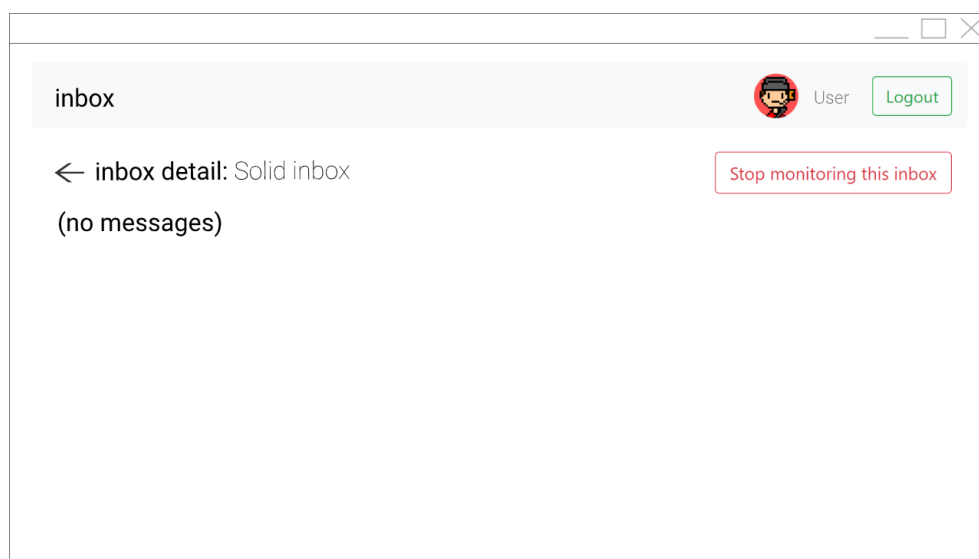


Figure 3.10: Screen 10 - Message detail - UC4.

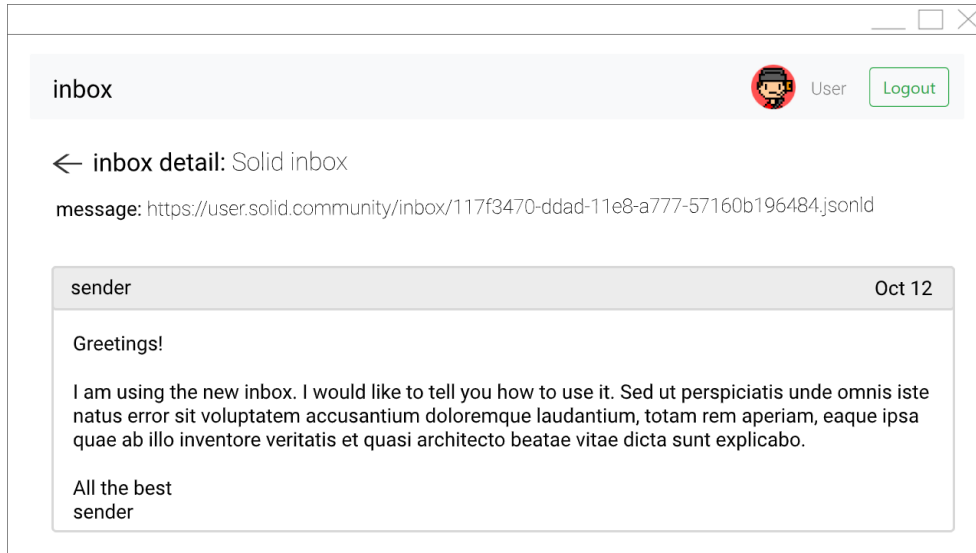


Figure 3.11: Screen 11 - Send a notification - UCs 6 - 9, empty On the screen, there is an option to send a notification to an either person from Solid contact list, or directly using IRI.

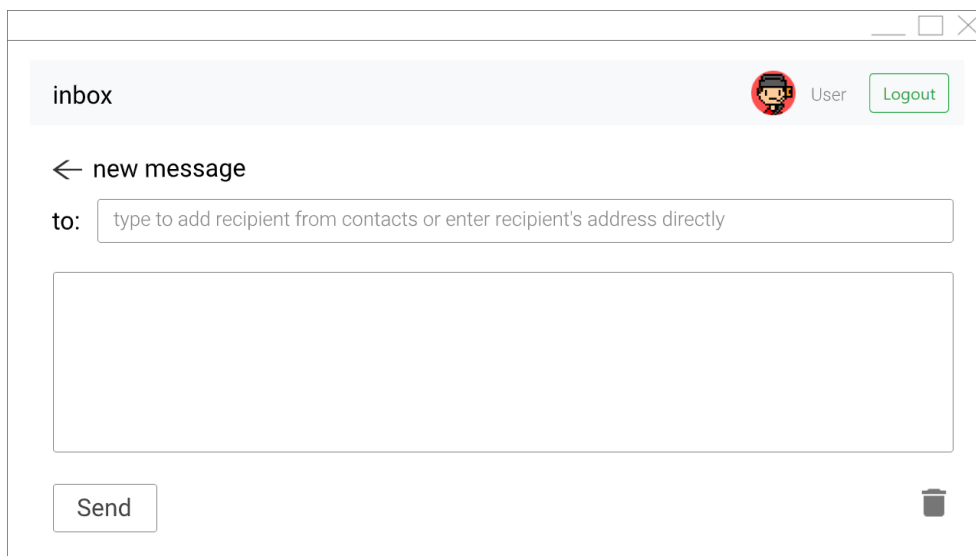
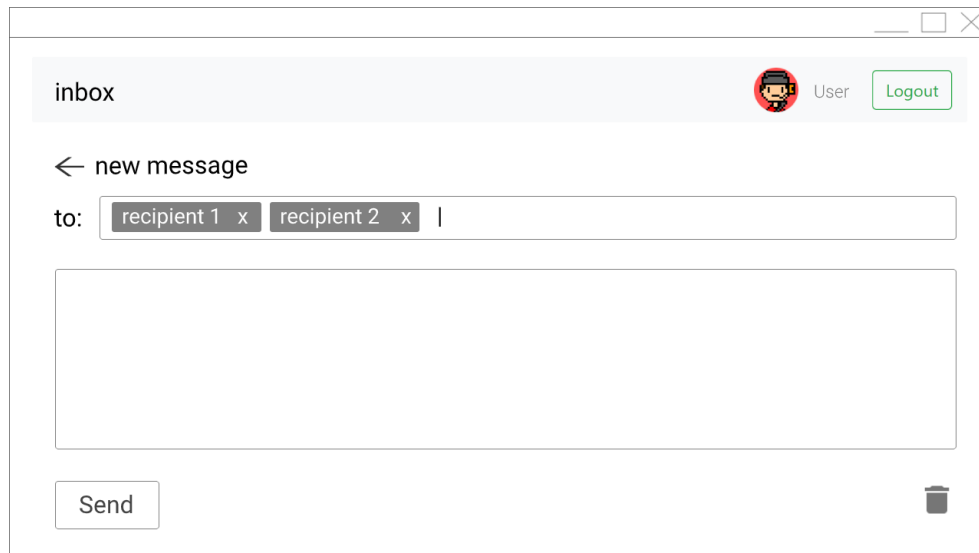


Figure 3.12: **Screen 12 - Send a notification - UCs 6 - 9, filled** User can specify multiple recipients.



3.2.4 Linked Data Notifications part

Based on the requirements from section 2.1, namely the requirement "F7 Support WebID login", the web application needs to rely on the existing libraries for WebID authentication. Implementing custom ones would be out of scope of this thesis.

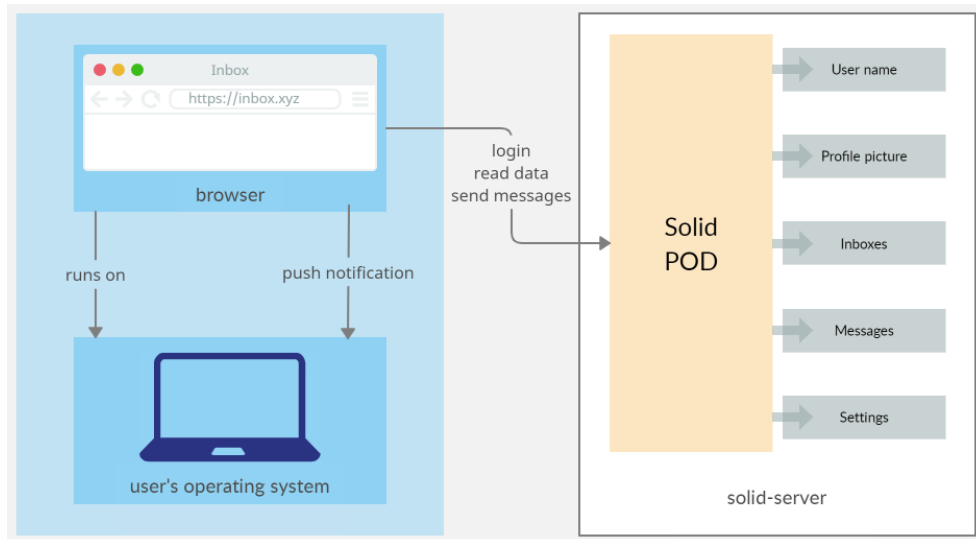
As a part analysis (see section 2.5), several proof-of-concept applications were created in order to explore the W3C recommendations and protocols. With the experience from these implementations, the `solid-client-authn-js`³³ was chosen as the only usable library for WebID authentication. Also, `solid-client`³⁴ was chosen as the best library for connecting to the Solid POD.

The libraries and experience with POC implementation determined the final application architecture - in-browser web application without back-end, written in JavaScript. Thanks to the usage of Solid PODs, there is no need for custom data storage and all users data can be stored in their Solid profile.

³³<https://github.com/inrupt/solid-client-authn-js>

³⁴<https://github.com/inrupt/solid-client-js>

Figure 3.13: **Application architecture** - web application running in user's browser, reading data from Solid POD hosted on a solid-server

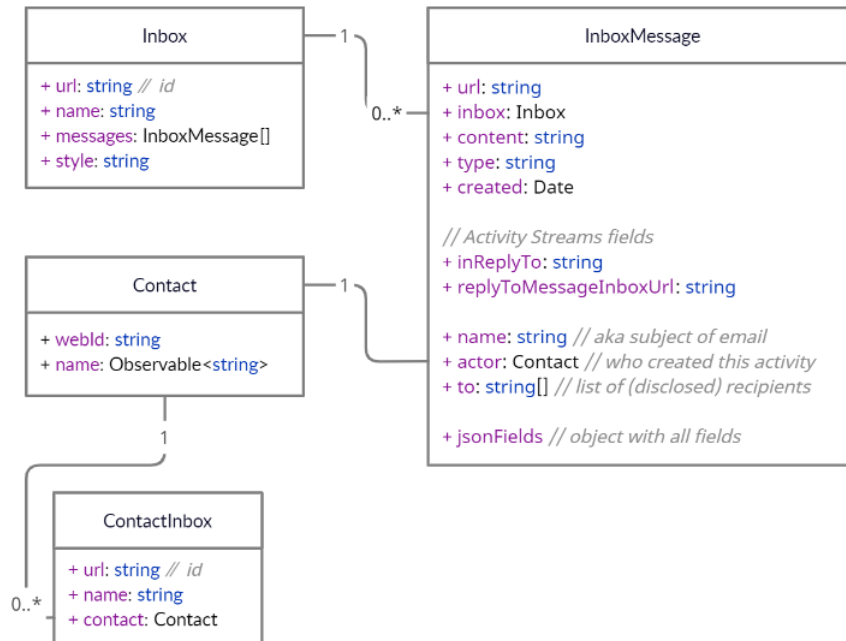


3.2.4.1 Model

The application is using data from the Solid POD directly, so it does not need an extensive model. But few entities are necessary in order to manipulate data in the application:

- **Inbox** - each instance represents single `http://www.w3.org/ns/ldp#inbox` of the logged-in user, identified by its `url`. Class member `messages` holds all messages of that inbox. `style` is merely a helper for CSS inline style (color of inbox label).
- **InboxMessage** - each instance of `InboxMessage` class represents a single message in the user's inbox. Identified by its `url`, it holds all information about the message, such as `created` date, message `type` (`plaintext/json/rdf...`, based on the message metadata). When the message is of an Activity Streams type, its JSON gets parsed and fields such as its sender (`replyToMessageInboxUrl` field) and original's message URL (`inReplyTo` field) are filled.
- **Contact** represents contacts from user's profile and contacts added in the send new message form.
- **ContactInbox** - as each user represented by his/her WebID can have multiple inboxes, `ContactInbox` holds information about every single such inbox (much like the `Inbox` class).

Figure 3.14: UML diagram of application entities



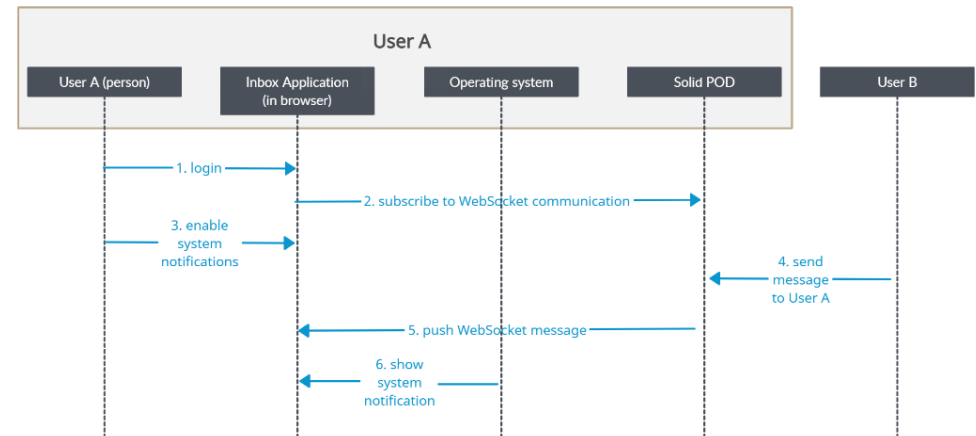
3.2.4.2 Technical details - notifications using WebSockets

To enable notifications for incoming messages, the specification of the solid-server offers WebSockets API³⁵. WebSockets is a JavaScript API³⁶ that enables "push" notifications without server polling. See Figure 3.15 for details.

³⁵<https://github.com/solid/solid-spec/blob/master/api-websockets.md>

³⁶https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Figure 3.15: **Receiving system notifications for incoming messages** - sequence diagram illustrating User A receiving notification for new message from User B.



3.2.5 ActivityPub part

Based on the analysis, the ActivityPub part of the application should use Pleroma as its back-end server. The social network Pleroma and its original version Mastodon are using a different authentication system than the solid-server: OAuth and WebFinger (as opposed to the solid-server's WebID). There are ongoing discussions in the community about interconnecting these two enterprises³⁷, however there is no connection for now. Because of this, the LDN and AP client parts need to be logically separated.

The client connection to the Pleroma is designed using client-to-server API (see subsection 1.1.6 for details) Thanks to this, there is no need for special application model. All data are loaded directly from Pleroma servers and stored in browser memory.

The integration was designed after discussion with community³⁸ and Pleroma developers³⁹. The following steps are required in order to login user, register the client and retrieve data from server:

1. In client, user submits his Pleroma account's username with URL of the server instance, e.g. `https://greenish.red/users/<username>`;

³⁷<https://forum.solidproject.org/t/discussion-solid-vs-activitypub/2685> and <https://socialhub.activitypub.rocks/t/which-links-between-activitypub-and-solid-project/529/22>

³⁸<https://socialhub.activitypub.rocks/t/want-to-build-an-activitypub-client-where-to-start/993/17>

³⁹<https://gitlab.com/vpzomtrrrft/c2sdemo>

3. DESIGN

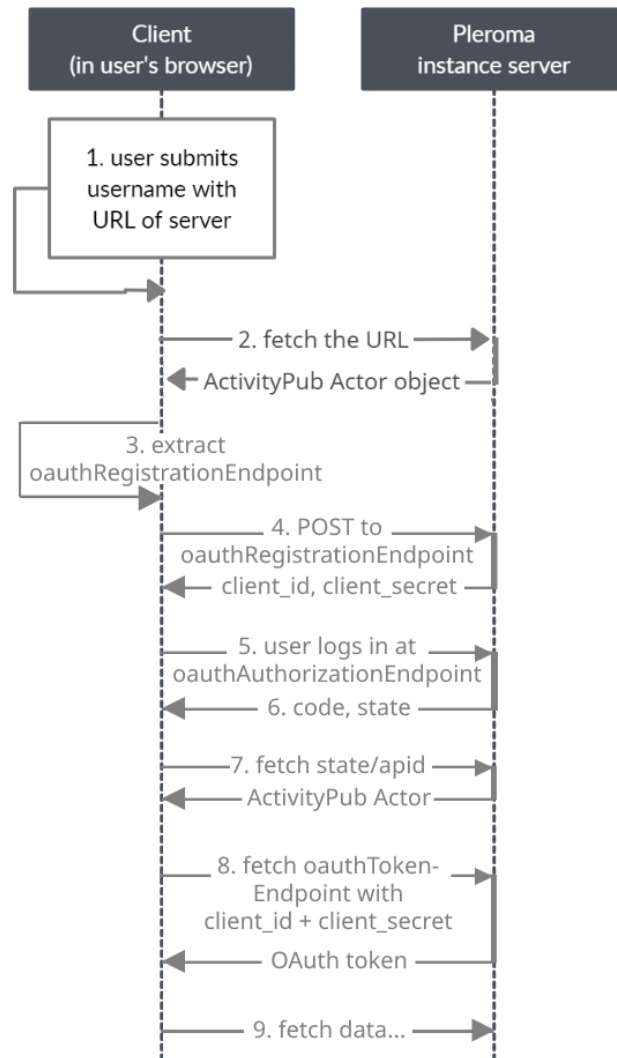
2. from the URL, the client application fetches user data - ActivityPub Actor object⁴⁰, typically in JSON-LD serialization;
3. from the AP Actor object, the `endpoints.oauthRegistrationEndpoint` is extracted. This is an URL at which the CS2 client application should register.
4. To register, the client application POSTs to this URL with required information:
 - client name (e.g. Inbox),
 - redirect URL (callback address, URL that the server redirects to after a successful registration),
 - requested permission scope - e.g. 'read write follow'.

The response from server includes `client_id` and `client_secret` parameters. Client should save this data.

5. When the client is successfully registered, it can now proceed to login the user. The user is redirected to the `endpoints.oauthAuthorizationEndpoint` URL from the previously received AP Actor object. The URL is Pleroma login page, where user enters his/her credentials. This way user never provides his sensitive data to the client.
6. After a successful login, the Pleroma server redirects user back to the client's callback URL, while providing two additional parameters:
 - "code" - unique identifier to retrieve OAuth token,
 - "state" ("apid") - URL with user identification (typically equal to URL user submits in the first step, e.g. `https://greenish.red/users/<username>`).
7. Client extracts the state/apid parameter, e.g. `https://greenish.red/users/<username>`, and fetches it (similar to the first two steps).
8. From the response (ActivityPub Actor) the `endpoints.oauthTokenEndpoint` URL is extracted. Client then fetches an OAuth token from this URL, using the `client_id` and `client_secret` parameters from previous communication (stored in the application memory).
9. Finally, this token can be used to retrieve some data, e.g. `fetch user.inbox` (e.g. `https://greenish.red/users/<username>/inbox`).

⁴⁰<https://www.w3.org/TR/activitypub/#actor-objects>

Figure 3.16: **Communication with Pleroma server** - sequence diagram illustrating Client application communicating with Pleroma server.



Because of the limitations in the Pleroma's implementation of the C2S API [27], the Inbox client is designed to only access the user's Pleroma inbox that contains public statuses.

3.3 Server

The analysis (see section 2.6) concluded that there is no need to implement a new server, neither for Linked Data Notifications, nor ActivityPub part.

3.3.1 Linked Data Notifications part - solid-server

As described at subsection 3.2.4 and Figure 3.13, the application's LDN client part is using Solid POD hosted at solid-server as the back-end.

3.3.2 ActivityPub part - Pleroma

As described in the analysis results, the client's ActivityPub part is using Pleroma server as its back-end, taking advantage of its C2S server API implementation.

Implementation

This chapter covers implementation specifics of the thesis main application Inbox. For implementation details about the proof-of-concept applications, see Appendix A.

The main application of this thesis was developed in two iterations. The first iteration was written using plain JavaScript with no application framework. However, it was deemed insufficient at the thesis defense. Detailed description of the application is available at the Appendix A.

Implementation details of the second iteration are described below.

4.1 Client

In this section, the implementation details of the user-facing client application are described. The client is split to two conceptual parts:

1. Linked Data Notifications - connecting to solid-server;
2. ActivityPub - connecting to Pleroma server.

4.1.1 Linked Data Notifications part

This subsection describes implementation details of the LDN client part.

4.1.1.1 Authentication

Inbox application relies on the `solid-client-authn-js`⁴¹ library for authentication. Application itself does not store any private user data. Only a list of arbitrary inboxes manually added for monitoring is stored in user's browser memory. However this is not private data that would be insecure to store. All requests for private user data are secured with session information. This way the application transfers any security concerns to the Solid server.

⁴¹<https://github.com/inrupt/solid-client-authn-js>

4. IMPLEMENTATION

Parts of the application that are accessible only to a logged user are secured with `auth-guard.service.ts`:

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):  
  ↪ UrlTree | boolean {  
  if (this._inruptService.isLoggedIn()) {  
    return true;  
  } else {  
    return this._router.parseUrl('/login');  
  }  
}
```

Then, in route definition at `app-routing.module.ts`, the secured routes are specified with `canActivate`:

```
{ path: 'monitor', component: MonitorInboxesComponent, canActivate: [  
  ↪ AuthGuardService]}
```

4.1.1.2 Working with Solid POD

To access user data stored on a Solid POD, the `solid-client-js` library is used. It supports two basic fetch modes: authenticated and unauthenticated. The latter is available in static context and provides an easy way to work with public RDF data. For example this is the way to find all inboxes associated with a WebID:

```
static async retrieveInboxUrlsFromWebId(webID: string): Promise<string[]> {  
  const myDataset: SolidDataset = await getSolidDataset(webID);  
  const profile = getThing(myDataset, webID);  
  return getUrlAll(profile, LDP.inbox);  
}
```

Minor inconvenience of the library is, as shown in the example above, that it works with JavaScript promises, as opposed to Angular's RxJS Observables⁴². Developer then can either mix Promises and Observables in the application, or wrap the Promise into Observable:

```
private _getObservableInboxes$() {  
  return new Observable<Inbox[]>((subscriber) => {  
    InboxDiscoveryService.retrieveInboxUrlsFromWebId(this.getSessionWebId()  
    ↪ )  
    .then(inboxUrls => {  
      this._prepareInboxes(inboxUrls).then(inboxes => {  
        subscriber.next(inboxes);  
      });  
    });  
  });  
}
```

⁴²<https://rxjs-dev.firebaseapp.com/>


```

        subscriber.complete();
      })
    });
  }).pipe(shareReplay(1));
}

```

The `shareReplay(1)` also removes redundant calls of the same endpoint, multiple observers can read the same data without extra HTTP request to the server.

4.1.1.3 Retrieving inbox name

The default Solid POD's inbox, the container for messages, does not have any name. But since users can have multiple inboxes (like email addresses), it is probable that they might want to name them, e.g. "school inbox", "work inbox" and similarly. RDF natively supports this. However there are multiple ways of adding name to the resource, so reading the name from RDF must accept at least the best-practise options. The Inbox application accepts three options, all use `http://purl.org/dc/terms/` predicate:

- string with locale - application default language (English) tag,
- string with any locale,
- string with no locale.

Using the `solid-client-js` library, this is the way to implement it:

```

private _findInboxName(inboxUrl): Promise<string> {
  return new Promise<string>(async (resolve, reject) => {
    try {
      await getSolidDataset(inboxUrl, {fetch: this.session.fetch}).then(
        inboxDataSet => {
          const inbox = getThing(inboxDataSet, inboxUrl);

          const titleEn = getStringWithLocale(inbox, DCTERMS.title, "en");
          if (titleEn) {
            resolve(titleEn);
            return;
          }

          const titleSomeLocale = getStringByLocaleAll(inbox, DCTERMS.title
            ↪ );
          if (titleSomeLocale && titleSomeLocale[0]) {
            resolve(titleSomeLocale[0]);
            return;
          }

          const titleNoLocale = getStringNoLocaleAll(inbox, DCTERMS.title);

```

```
        if (titleNoLocale && titleNoLocale[0]) {
            resolve(titleNoLocale[0]);
            return;
        }

        reject();
        return;
    });
} catch (error) {
    console.error("Error when finding inbox name: ", error);
    reject();
}
});
}
```

4.1.1.4 Inbox monitoring using WebSockets

As described at subsection 3.2.4.2, WebSockets (WS) are used for inbox monitoring. In order to receive WS message, the application has to open the connection with correct WS protocol supported by the server. During the development, problem with the WS protocol version has occurred, however it has been fixed in the meantime (see subsection C.2.4 for the details). The most important part is what happens after receiving WS message - see the `socket.onmessage` callback:

```
private connect(inboxUrl: string, connectNew: boolean = false) {
    let socket = new WebSocket(MonitorInboxesService.getWsUrlFromInboxUrl(
        ↪ inboxUrl), MonitorInboxesService.WS_SOLID_PROTOCOL);
    socket.onopen = this.onopenCallback(inboxUrl, this._snackBar, connectNew)
        ↪ ;

    socket.onmessage = (msg) => {
        if (msg.data && msg.data.slice(0, 3) === 'pub') {
            this.zone.run(() => {
                this._snackBar.openFromComponent(MessageSnackbarComponent, {data:
                    ↪ inboxUrl});
                this._systemNotificationsService.inboxNotification(inboxUrl);
            });
        }
    }
    this.sockets.push(socket);
}
```

If the message contains data and the data is string 'pub' (as published), the application notifies the user, using both in-browser and system notification (if enabled). The `this.zone.run` is needed in order to run the code in Angular context.

However using WebSockets for inbox monitoring has its limitations. According to the specification⁴³, the server sends WS messages on all CRUD (POST, PUT, PATCH, DELETE) operations on the subscribed resource. This means that if e.g. message deletion would be implemented, there is no way to distinguish between new incoming message (POST) and the DELETE request. In order to keep getting notifications on new messages, a subsequent message reload and comparison would need to be implemented, using WS messages only as a trigger for this check.

Furthermore, the current solid-server implementation has another bug limiting the WS usage - it is sending phantom WS messages even when no CRUD operation has occurred on the resource. See Appendix C for details and link on this issue.

4.1.1.5 Problems encountered

During development of the applications, many problems were encountered. This section describes the most affecting issues and problems that led to implementation changes. Part of these problems were bugs in the used third-party libraries. These bugs were reported to the library authors and part of them was already fixed. For list of these bugs see Appendix C.

Cannot use solid libraries in Angular with npm Angular uses npm⁴⁴ as its default package manager. Based on the analysis, solid-client-js library was chosen to connect to the Solid POD. However after including this library in the Angular application, the build fails.

The problem lays in the library dependencies and missing polyfills (supplying new JavaScript functions in old browsers) in the new npm builds. After reporting the issue to the library authors (see <https://github.com/inrupt/solid-client-js/issues/608>) and working with them on a fix, a workaround had to be found in order to use this library.

A custom webpack configuration is needed for the build to work. This is not supported by the default npm package manager. However its alternative, the yarn package manager⁴⁵, does support this feature. To use yarn instead of the default npm, the following command is used:

```
ng set --global packageManager=yarn
```

Using yarn, a custom webpack configuration can be supplied to the application build and polyfills can be added. But first, custom angular builder

⁴³<https://github.com/solid/solid-spec/blob/master/api-websockets.md>

⁴⁴<https://www.npmjs.com/>

⁴⁵<https://yarnpkg.com/>

4. IMPLEMENTATION

must be used. After trying out `ngx-build-plus`⁴⁶, a better alternative `@angular` `↔ -builders`⁴⁷ was used instead:

```
yarn add @angular-builders/custom-webpack --dev
```

More details can be found on StackOverflow⁴⁸. Required polyfills are listed in the `webpack.partial.js` file (in the application root):

```
module.exports = {
  resolve: {
    fallback: {
      crypto: require.resolve("crypto-browserify"),
      stream: require.resolve("stream-browserify"),
      util: require.resolve("util/"),
      buffer: require.resolve("buffer/"),
    }
  },
}
```

In order to add the polyfill file to the build, the following custom config option must be used at every build specification in `angular.json` file, path `projects.inbox.architect.build.options`:

```
"customWebpackConfig": {
  "path": "./webpack.partial.js",
  "replaceDuplicatePlugins": true
},
```

Additionally, after a successful application build, another error occurred. In the browser, the application did not start up and `global not defined` error occurred in the console. After further research, the solution was adding the following code to the main HTML file `\src\index.html`:

```
<script>
  global = globalThis // fix for "global not defined" error
</script>
```

With this setup, solid libraries can be used in angular

⁴⁶<https://github.com/manfredsteyer/ngx-build-plus>

⁴⁷<https://github.com/just-jeb/angular-builders>

⁴⁸<https://stackoverflow.com/questions/51068908/angular-cli-custom-webpack-config/51130803#51130803>

solid-client-authn-js library does not stay logged in After integrating authentication in Inbox with Solid using the `solid-client-authn-js`, a problem occurred - the user does not stay logged in after page refresh. Before reporting this issue, an existing GitHub issue was found⁴⁹. This is a known problem of the library. Possible fixes and workarounds were discussed by thesis author on the GitHub issue (<https://github.com/inrupt/solid-client-authn-js/issues/423#issuecomment-741646705>). As suggested in the discussion, using another library `solid-auth-fetcher`⁵⁰ has been tried. Unfortunately, this library cannot be used as a workaround, mainly because it is missing implementation of key method (see Appendix C for details).

As a result, no solution has been found to this issue and it remains as an UX problem of the Inbox application.

4.1.2 ActivityPub part

In the design chapter, the Inbox client communication with the Pleroma backend is explained. Here, the implementation details are described.

Pleroma source code is located in the `pleroma.component.ts` component and its service `pleroma.service.ts`. When the `/pleroma` page is loaded, the `initPage` \rightarrow () method is called. It determines whether the page has been loaded as a callback from the Pleroma server (= when the `code` and `state` URL parameters are present), or the page has been requested by user - `loadInbox()` method is called in that case:

```
private initPage(): void {
  this.route.queryParamMap.subscribe(
    queryParamMap => {
      this.code = queryParamMap.get('code');
      this.apid = queryParamMap.get('state');
      if (this.code && this.apid) {
        this.submitted = true;
        this.connectPleroma(this.code, this.apid);
      } else {
        this.loadInbox()
      }
    }
  );
}
```

But before any user data can be loaded, first they must submit their Pleroma username with the URL of their account's server, e.g. `https://greenish.red/users/<username>`. When the URL is submitted, the steps 1. - 5. from the Pleroma connection can be executed:

⁴⁹<https://github.com/inrupt/solid-client-authn-js/issues/423>

⁵⁰<https://github.com/solid/solid-auth-fetcher/>

4. IMPLEMENTATION

```
submit() { // user submitted - step 1.
  this.submitted = true;
  this.pleromaService.fetchUser(this.idInput).subscribe( // step 2.
    user => {
      BrowserStorageService.saveToLocalStorage(PleromaService.
        ↪ STORAGE_KEY_USERS, [this.idInput])
      this.pleromaService.registerApp(user.endpoints.
        ↪ oauthRegistrationEndpoint).subscribe( // steps 3., 4.
        app => { // step 5.
          this.pleromaService.logUserIn(user, app);
        });
    })
}
```

After a successful login, the Pleroma server redirects user back to the Inbox client application with the `code` and `state` parameters. As a result, the `'if (this ↪ .code && this.apid) {..'` condition from the `initPage()` method is satisfied and the `connectPleroma()` is executed.

The `connectPleroma()` function then does steps 6. - 8. from the subsection 3.2.5 - it receives the `code` and `state` parameters, fetches the user's Actor ActivityPub object in JSON-LD and retrieves the OAuth token:

```
private connectPleroma(code, apid) {
  this.pleromaService.fetchUser(apid).subscribe(user => {
    this.pleromaService.getOAuthToken(code, apid, user.endpoints).
    ↪ subscribe(
      tokenInfo => {
        const token = tokenInfo.access_token;
        this.pleromaService.saveToken(token);
        BrowserStorageService.saveToSession('userInbox', user.inbox)

        // remove the 'code' and 'state' parameters from URL
        this.router.navigate([],
          {queryParams: {'code': null, 'state': null},
            queryParamsHandling: 'merge'});
      });
  })
}
```

In order to retrieve token, the client has to do a POST request with specific parameters:

```
getOAuthToken(code: string, apid: string, userEndpoints): Observable<any> {
  const storageKey = PleromaService.STORAGE_PREFIX_APP + btoa(userEndpoints
    ↪ .oauthRegistrationEndpoint);
  const app = BrowserStorageService.loadFromLocalStorage(storageKey);

  const params = {
    grant_type: 'authorization_code',
```

```
code: code,
client_id: app.client_id,
client_secret: app.client_secret,
redirect_uri: this.CALLBACK_URI,
scope: 'read write follow',
};

return this.http.post(userEndpoints.oauthTokenEndpoint, params, {headers:
  ↪ {'Content-Type': 'application/json'}});
}
```

Finally, the Inbox client application now has all data that is required to fetch user data from the server. The application proceeds with fetching user ActivityPub inbox, that contains the user's public statuses in the Pleroma context. The statuses are then displayed to the user.

4.2 Server

As concluded in the analysis, no server implementation is needed for this thesis. The Inbox client is using existing server implementations (see section 3.3 for details).

Testing

This chapter describes how the Inbox application was tested. To avoid ambiguities, first the testing terms are defined and then information about how the testing methods were applied is presented. For details on testing the proof-of-concept applications and the first client application iteration see Appendix B.

5.1 Unit tests

Unit tests are automated tests that verify behavior of the application's isolated methods. For example for a simple method `sum (a, b) {return a + b;}`, a unit test should verify the method's output. Unit tests should run with each application build. A build should fail when the tests don't pass. This way a developer can be informed about code changes that broke the desired behavior as soon as possible.

The Inbox application was built using the Angular framework which comes with unit testing capabilities - it takes advantage of Karma and Jasmine:

- **Karma**⁵¹ is a JavaScript test runner. It provides an environment in which the unit tests can be executed.
- **Jasmine**⁵² is a general JavaScript test framework that provides test syntax.

To summarize, Jasmine was used to write unit tests and Karma is used to run them. A `package.json` script goals `test` and `test-headless` were created for easy test execution. See subsection 6.2.6 on how to run the tests.

In the Angular framework, unit tests are typically located in the same folder as the class being tested. So for example the class `MonitorInboxesService` is located in the `\src\app\services\monitor-inboxes`

⁵¹<https://karma-runner.github.io/>

⁵²<https://jasmine.github.io/>

5. TESTING

folder, the file `monitor-inboxes.service.ts` contains the class code and the file `monitor-inboxes.service.spec.ts` contains the unit tests, such as:

Listing 5.1: **Example of unit test.** Testing method for parsing WebSocket URL from user inbox URL.

```
it('get correct WS URL from inbox URL', () => {
  const wsUrlFromInboxUrl = MonitorInboxesService.getWsUrlFromInboxUrl('
  ↪ https://tonda.solidcommunity.net/inbox/');
  expect(wsUrlFromInboxUrl).toEqual("wss://solidcommunity.net/");
});
```

5.1.1 Coverage

All custom service methods with application logic such as sorting, parsing and similar, were covered with unit tests. Methods that use library functions to orchestrate application logic were not tested, as testing 3rd party libraries is not in scope of unit testing. Inbox contains 31 unit tests. Code coverage statistics as generated by Angular's CLI:

```
> ng test --no-watch --code-coverage

===== Coverage summary =====
Statements   : 33.12% ( 210/634 )
Branches     : 22.6% ( 33/146 )
Functions    : 31.4% ( 65/207 )
Lines        : 31.69% ( 187/590 )
=====
```

5.2 E2E tests

End-to-end (E2E) tests are automated comprehensive tests of the whole system. They should simulate behavior of a typical user of the tested system, they are usually based on the user scenarios. E2E tests of web applications are typically run with testing frameworks that control a testing instance of a browser. Principally they run longer than unit test, so they are usually run periodically and/or with each release candidate.

The Angular framework also provides E2E testing capabilities - apart from the aforementioned Jasmine test framework, it utilizes Protractor⁵³ - an E2E testing framework. Jasmine is used to write the tests, Protractor is used as the test runner (just like Karma for unit tests).

A `package.json` script goal `e2e` is used for E2E test execution. See subsection 6.2.6 on how to run the tests.

⁵³<https://www.protractortest.org/>

5.2.1 Coverage

E2E tests cover integration with the Solid server and the Pleroma social network. They test main use cases derived from section 2.2 and from test cases defined at subsection 5.4.3 - e.g. reading message list, detail; sending message, form validation and other UI parts. More specifically:

- integration with Solid PODs: `AppPage.loginToInrupt`. For application to work, it needs to read data from 3rd-party server (this is not a separate test, but methods executed before most tests to ensure login state)
 1. test logs in using preset credentials (existing test user, see subsection 6.2.6)
 2. if the application has not been authorized in the Solid POD yet, test authorizes it
 3. test waits for return back to application and ensures user has been successfully logged in.
- further login checks with UI test (OIDC auto-complete suggestion): `login.e2e-spec.ts`
- consume messages: `messages.read.e2e-spec.ts`
 - read list of messages
 - read message detail
- sending messages: `messages.send.e2e-spec.ts`
 - form validation (cannot send empty email)
 - reading contact from user Solid POD
 - UI checks - recipients picker
 - sending simple message (integration with Solid POD)
- UI navigation: `homepage.e2e-spec.ts`
- integration with Pleroma: `pleroma.login.e2e-spec.ts`

5.3 Continuous integration

To avoid introducing bugs during development, a continuous integration (CI) was set up. Taking advantage of the GitHub actions⁵⁴, unit tests and E2E tests are executed with every Git Push to the hosted repository. The application is installed, built and tests are executed in the headless mode. For details see the subsection 6.2.8.

⁵⁴<https://github.com/features/actions>

5.4 Usability testing

Usability testing is manual testing of the application. Its goal is to determine how usable for a typical user the system is. There are several types, such as cognitive walk-through, heuristic evaluation and user testing [28]. The key part of the thesis is to prove that the solid inbox application can be user-friendly. To ensure this, a usability testing was conducted.

Because the application's typical user is an experienced user, the cognitive walk-through was selected as the most suitable usability testing method.

5.4.1 Cognitive walk-through

Cognitive walk-through testing is typically conducted by a developer/UX expert. This person uses user scenarios to walk through the application and uses his expertise to identify system's UX defects [28].

More specifically, the goal is to answer the following questions [29]

1. Will the user try and achieve the right outcome?
2. Will the user notice that the correct action is available to them?
3. Will the user associate the correct action with the outcome they expect to achieve?
4. If the correct action is performed; will the user see that progress is being made towards their intended outcome?

5.4.2 How the cognitive walk-through was conducted

The tester, an experienced front-end developer, followed the steps defined in test cases subsection 5.4.3. At each step, the tester was trying to answer the questions defined at subsection 5.4.1. Afterwards, the tester summarized his findings and presented the results.

The complete testing records with the findings is available in the Appendix D.

5.4.3 Test cases

Test cases were derived directly from the use cases (defined at section 2.2). There are only minor differences, with the biggest one being the order - test cases are ordered to follow the natural user flow through the application.

TC1 Read list of messages from all available inboxes

- a) User logs in using WebID
- b) System shows list of all received messages

TC2 Read list of messages from selected inbox. Users can have multiple inboxes in their Solid profile. Application shows list of all user inboxes in the left menu.

- a) On the Incoming page, choose an inbox in the left menu and click on it
- b) System shows list of all messages in that inbox

TC3 Read detail of a received message

- a) On the list of messages, choose some message from the list and click on it
- b) System shows you its content with details

TC4 Reply to message

- a) select a message that has a recipient (there's some name in the "From" column)
- b) open it
- c) click on "reply" button
- d) send your reply

TC5 Send a simple message. Send a simple message to an unknown person.

- a) click on "Send message"
- b) choose simple message
- c) add recipient "<https://inbox4.inrupt.net/profile/card#me>"
- d) send him a message

TC6 Send an AP message. Send an Activity Streams message to a person from your contacts.

- a) on Send message, choose Activity Streams message
- b) click on recipient and choose one of the shown contacts
- c) fill all message fields
- d) send

TC7 Start monitoring arbitrary inbox. Users can add any other inbox for monitoring (e.g. inbox for their article where people send their comments).

- a) go to Monitored inboxes

5. TESTING

- b) add “`https://inbox3.inrupt.net/profile/card#me`” for discovery and monitoring
- c) add “`https://inbox4.inrupt.net/inbox/`” directly for monitoring

TC8 Stop monitoring arbitrary inbox

- a) at Monitored inboxes, stop monitoring previously added inboxes

TC9 Receive a system notification on a new message

- a) enable system notifications (please note they don't work in incognito browser mode)
- b) (test administrator sends a message)
- c) user receives system notification

5.4.4 Cognitive walk-through testing results

The cognitive walk-through has not found any serious UX problems. All test cases answers to the testing questions (defined at subsection 5.4.1) were answered positively. The findings are presented below:

Table 5.1: Table with test findings and reactions to them. Ordered by their seriousness, from most serious to least serious.

Finding	Reaction
The “Messages from all inboxes” option could be reduced to “All inboxes” and be visually separated from the single inboxes.	Fixed - used suggested text and visual separation.
Show message subject in the message list.	Fixed - added subject to the message list.
Inbox labels should not be visually prominent in the message list - this information is not important to the user.	Fixed - made inbox labels (badges) smaller.
Monitoring arbitrary inbox: Users can get confused by two different fields and two different buttons (and other confusion in TC7, TC8)	Fixed - added visual separator and added informative text
Logo “inbox” in the header should redirect the user into the inboxes overview, now it logs the user out.”	Fixed link.

<p>The notification logic does not seem stable. Sometimes only a toast message appears, sometimes only a browser system notification and sometimes both.</p>	<p>Reported - this is a bug of the Solid WS implementation. A bug was reported.</p>
<p>Consider adding a button icon to each item in the inbox overview. It can be visible on hover only. In result it will be a nice shortcut for a quick reply.</p>	<p>Not fixed - action is not easy to implement in the list of messages.</p>
<p>Message details should be part of the header and not below the message content.</p>	<p>Not fixed - unlike email client, the message details are important part of the message, because parsing of the message is not common and user needs simple access to the content.</p>
<p>The form has inconsistency in background colors. Keep it the same as in inbox (main grey background and for content use white background to make it step out of the rest of the page).</p>	<p>This is a good design suggestion. Added to the "future work".</p>
<p>It might take few second before user finds the "send message" option in the top navigation. Change the label to "New message" and add an icon to it to make it more visible as it is one of the primary user actions. Also there can be new message icon in the same container as reload button (can be a shortcut to send a message from actual inbox)</p>	<p>This is a good design suggestion. Added to the "future work".</p>
<p>A success toast can contain green color which represents success (and the errors should be red).</p>	<p>This is a good design suggestion. Added to the "future work".</p>
<p>When loading list of messages, spinner should be placed in the messages container, not in the "reload" button.</p>	<p>Not fixed - spinner size and placing is only a small design issue.</p>
<p>The purple loading spinner could be smaller.</p>	<p>Not fixed - spinner size and placing is only a small design issue.</p>

The simple message misses the subject field. Is it by purpose?	Yes, simple message over LDN does not support subject field.
--	--

5.5 Tests of compatibility with existing tools

One of the thesis goals was to test compatibility of the new client with existing tools implementing the LDP, LDN and AP W3C recommendations. This section describes how the compatibility is ensured.

5.5.1 LDP, LDN, Solid - Solid-server

The Inbox client application is built on the solid-server, using Solid POD as its data storage. The client compatibility with solid-server is being tested using the E2E tests - integration with its main implementation Inrupt.net is ensured. This inherently tests the LDP, LDN and Solid compatibility.

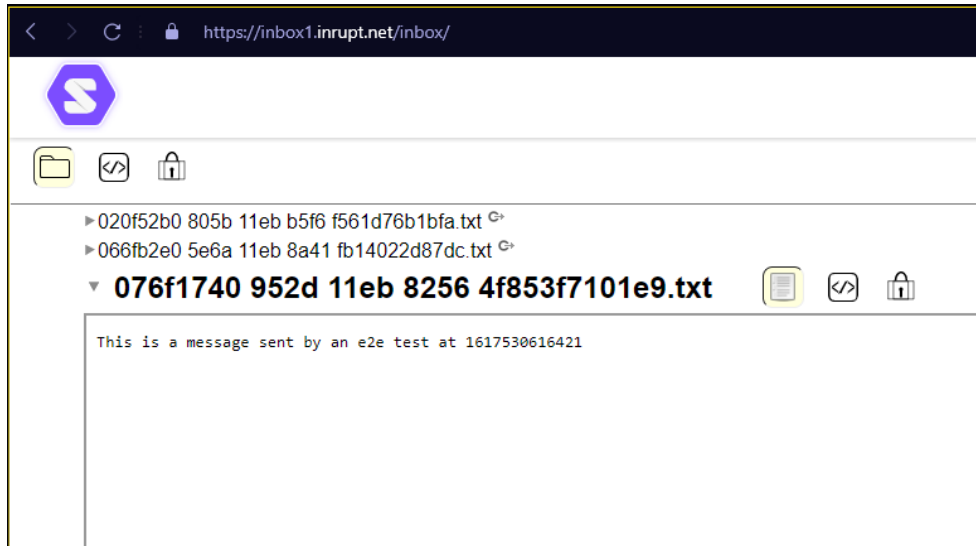
Also manual tests have been conducted. Using e.g. the HTTP POST request below, a message can be posted to the test user's inbox and then consumed in the application:

Listing 5.2: **POST ActivityPub message to user inbox.** Based on the AP specification, a message can be POSTed to the user inbox and then consumed in the Inbox client.

```
curl --location --request POST 'https://test-user1.inrupt.net/inbox/' \
--header 'Content-Type: application/ld+json' \
--data-raw '{
  "@context": ["https://www.w3.org/ns/activitystreams",
    {"@language": "en"}],
  "type": "Like",
  "actor": "https://dustycloud.org/chris/",
  "name": "This is message (activity) name",
  "object": {"type": "https://rhiaro.co.uk/2016/05/minimal-activitypub", "
    ↪ content": "Hello! I'\''m sending you an interesting message content
    ↪ !"},
  "to": ["https://rhiaro.co.uk/#amy",
    "https://dustycloud.org/followers",
    "https://rhiaro.co.uk/followers/"],
  "cc": "https://e14n.com/evan"
}'
```

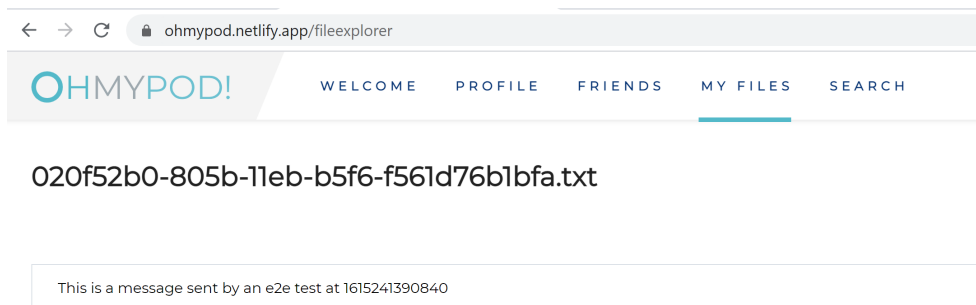
Furthermore, for every message, the Inbox client shows link to its original URL. User can consume the message directly there:

Figure 5.1: Message sent by Inbox test, consumed directly in the Solid data browser at Inrupt.net.



The message can be consumed in other Solid POD applications in similar fashion. The `https://podbrowser.inrupt.com/` and `https://ohmypod.netlify.app/` have been tried out:

Figure 5.2: Message sent by Inbox test, consumed in the Solid POD "OhMyPod!" browser.



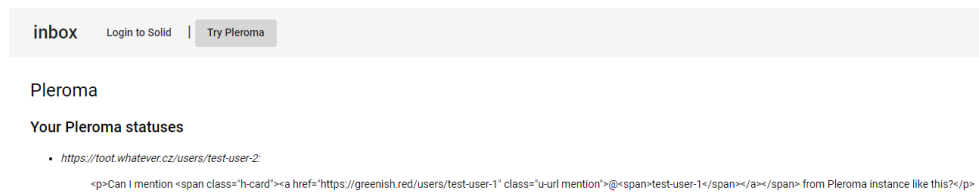
5.5.2 ActivityPub - Pleroma

The Inbox client application has capability of connecting to the Pleroma social network and access user inbox that contains paginated user statuses. The compatibility is being tested using the E2E tests.

5. TESTING

Pleroma is a part of the `https://fediverse.party/`, so e.g. users from Mastodon social network can post to Pleroma users and vice-versa. To test Inbox client compatibility, a manual test was conducted: a post was created on the Mastodon network, tagging the Pleroma user in the post. This post is now available at the tagged Pleroma user's timeline (inbox). Inbox client is able to read this post:

Figure 5.3: Post sent from Mastodon social network to the Pleroma test user, consumed in the Inbox client.



Documentation

This chapter provides documentation of the Inbox application. It is split to three main parts:

1. user documentation - how to use the application, including screenshots and user scenarios;
2. administrator documentation - including links to access source code, software prerequisites, application requirements; installation, build and run steps;
3. developer documentation - how is the project structured, how to contribute to it (where to code to add functionality).

6.1 User documentation

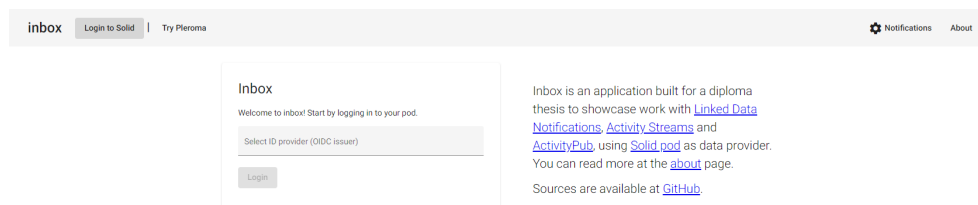
In this section, the application is described from a user perspective. It is explained how to use the application, including screenshots. Main application features are presented.

Application runs in a web browser (tested in Google Chrome and Mozilla Firefox). You can deploy your own copy using the steps below or simply access the live version at <https://whyneedtofillusername.github.io/inbox/>.

6.1.1 Login

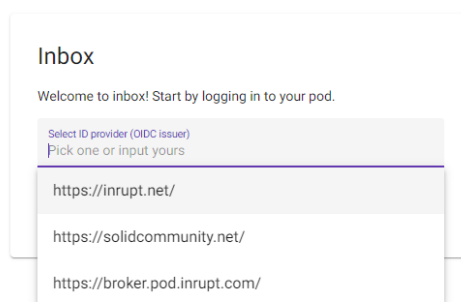
To start using the application, first you need to login:

Figure 6.1: Login screen



Use identity provider of your choice:

Figure 6.2: Choosing login provider



After clicking on the Login button, you will be redirected to the login provider's page. If you don't have an account, you can create one there.

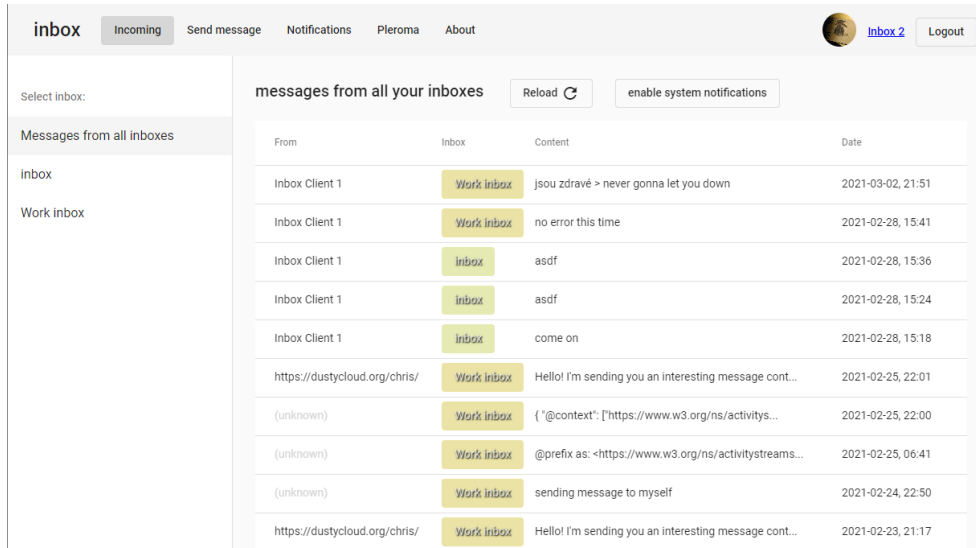
After a successful login, you might need to authorize the Inbox application's access to your pod data. Then you will be redirected back to the Inbox application.

In the top right corner, the username and profile picture is loaded from the Solid profile.

6.1.2 Reading messages

After a successful login (and application authorization), the main screen is presented - list of all messages in your profile's inboxes:

Figure 6.3: List of all messages

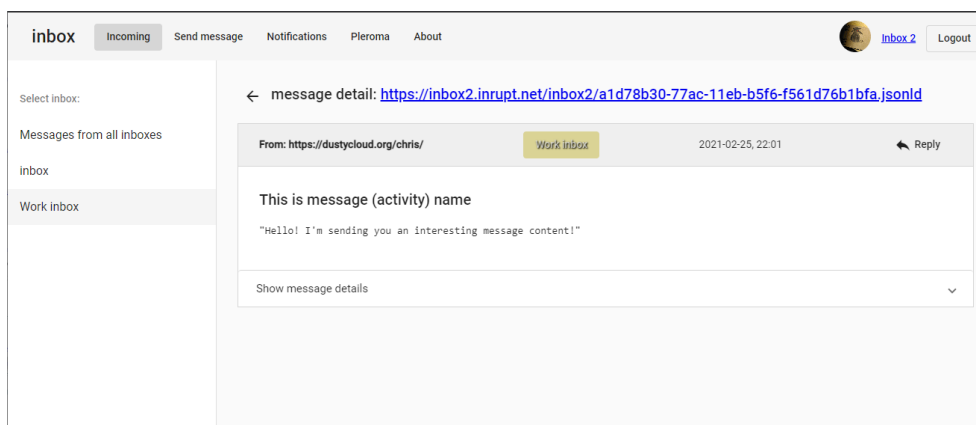


By default, Inbox application shows list of all messages from all inboxes combined. In the left column, list of all inboxes in the Solid profile is shown. You can choose specific inbox and application will show messages only from that particular inbox.

6.1.2.1 Message detail

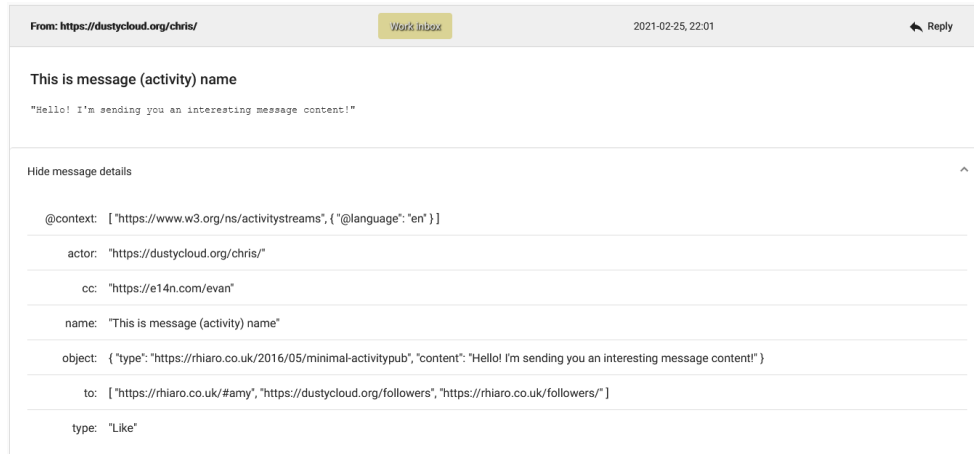
When you click on a message from the list, its content is opened:

Figure 6.4: Message content



If this is a message in the Activity Streams format, You can look at its details:

Figure 6.5: Message content - detail

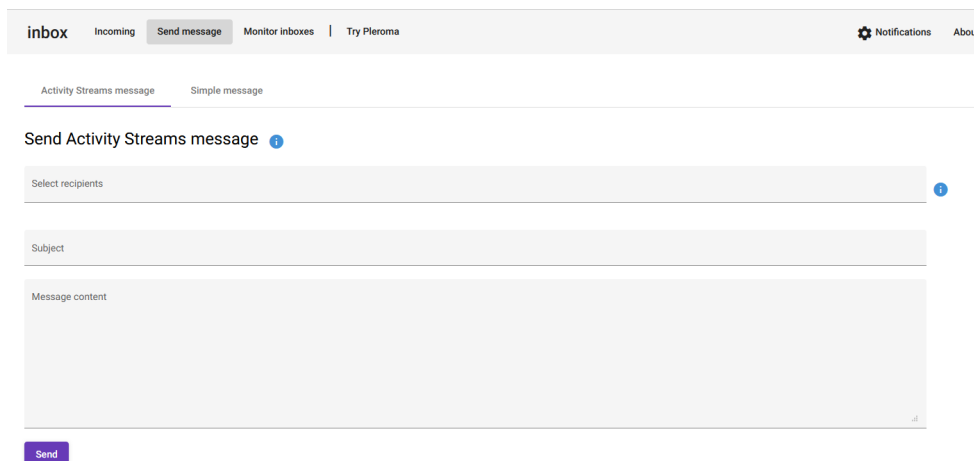


For this message format you can also use the "reply" button to send a reply message.

6.1.3 Sending a message

To send a message, either use the "reply" function, or go to the "Send message" tab. Here, you have two options. The default option is to send a message in an Activity Streams format. This allows the application to send details with the message, such as "subject", multiple recipients and original sender's id:

Figure 6.6: Send message - Activity Streams



You can also choose to send a simple text message. First, you have to select recipient(s). You have three options:

1. select recipients from your profile contacts (loaded from WebID's `foaf:knows`);
2. insert WebID - application executes inbox discovery (using `ldp:inbox`);
3. inser inbox IRL directly.

Figure 6.7: **Selecting recipient from contacts** - application offers list of user contacts

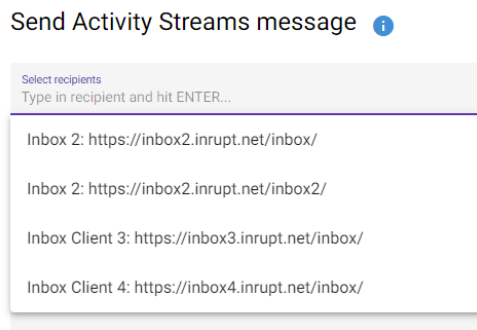
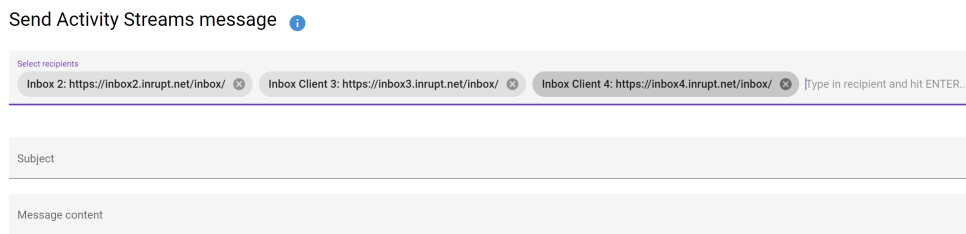


Figure 6.8: **Selecting multiple recipients** - user can send a message to multiple recipients simultaneously

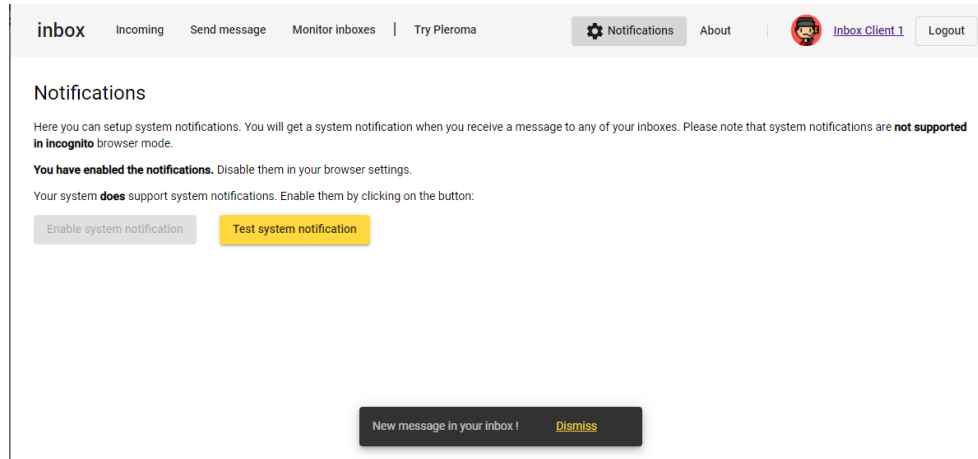


If you're using the "reply" functionality, the recipient's inbox is pre-filled automatically. Then simply type in message content (text only) and click on the "send" button.

6.1.4 Inbox monitoring and notifications

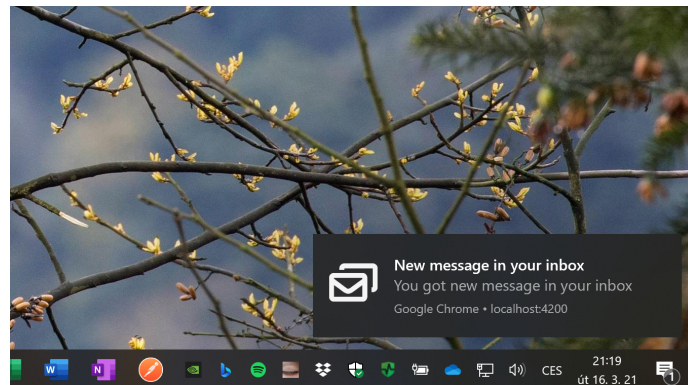
Inbox application automatically starts monitoring all user inboxes after login. This means that application creates a notification when user receives a new message to his inbox:

Figure 6.9: **Application notification** for new message



Application also supports **system notifications**. This means that when user receives a new message, Inbox creates a system notification. The system notification is shown above all windows and user is informed of a new message, even when the Inbox application is not in focus:

Figure 6.10: **System notification** - example of a system notification in OS MS Windows 10



6.1.4.1 Enabling system notifications

To receive system notifications, user must enable them in the "Notifications" tab. System notifications are also limited by user system and browser, they do not work e.g. in incognito browser mode.

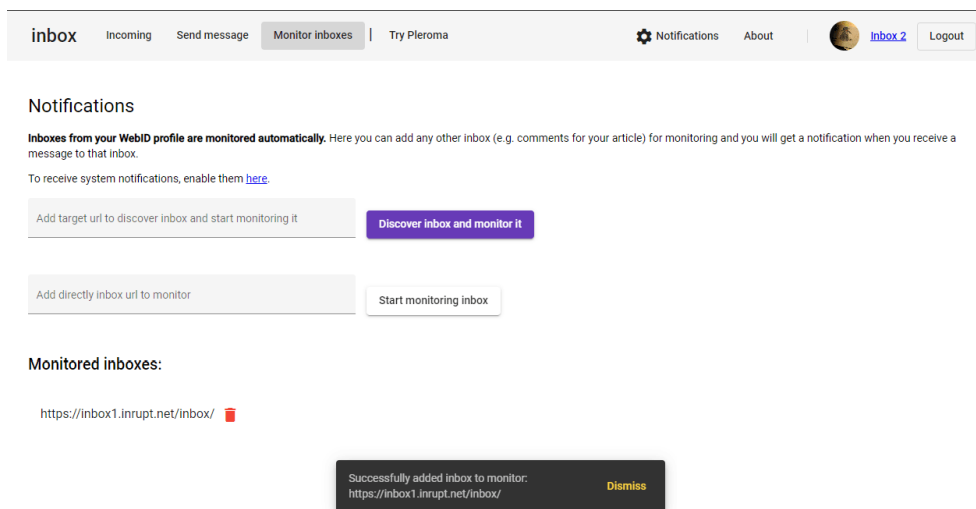
6.1.4.2 Monitoring other inboxes

Inbox application can also monitor other inboxes that user specifies. This allows user to get notifications for incoming messages that are not in his Solid

profile. To do so, go to the "Monitor inboxes" tab and add the desired inbox. There are two options:

- add target URL (e.g. WebID) and let the application discover `ldp:inbox`;
- add inbox URL directly (application checks that the inserted URL contains Link HTTP header with `<http://www.w3.org/ns/ldp#Container>; rel ↪ ="type"`).

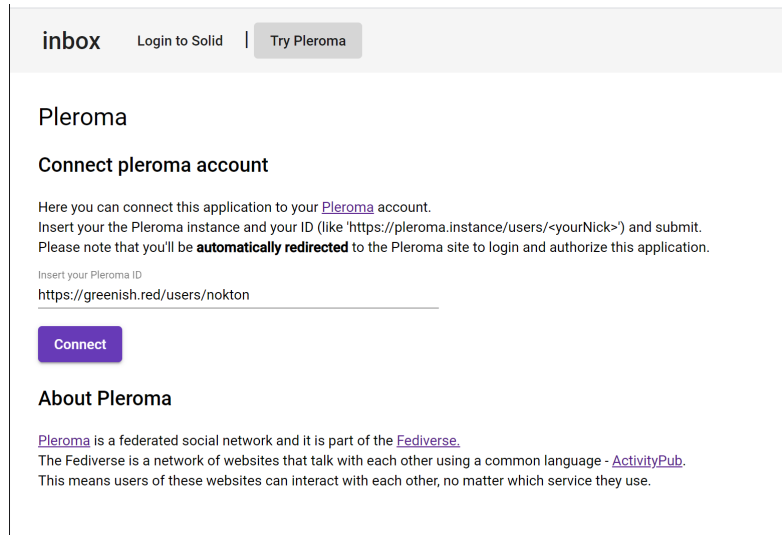
Figure 6.11: Add inbox for monitoring



6.1.5 Pleroma connection

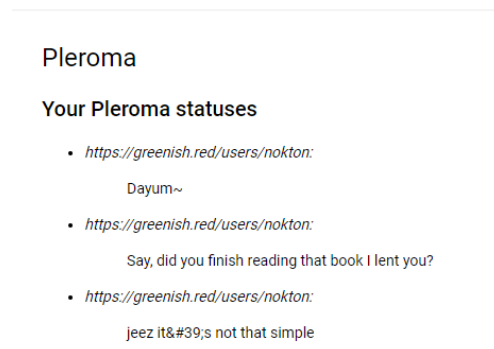
Inbox application also includes an integration of ActivityPub C2S API (for details see subsection 1.1.6):

Figure 6.12: Step 1 - login to Pleroma



Because of the limited options of Pleroma's implementation, Inbox only offers list of all user statuses on the Pleroma:

Figure 6.13: Step 2 - user's Pleroma statuses



For more details on problems with Pleroma connectivity see subsection 4.1.1.5.

6.2 Administrator documentation

This section documents application for its administrators. It contains link to the source code, presents application requirements and provides steps to build, run and deploy the application.

6.2.1 Source code

Source code is versioned using Git on GitHub, published as open-source. It is available with documentation at

<https://github.com/WhyINeedToFillUsername/inbox>

GitHub is a public Git VCS with a web interface. Please note that the source codes at GitHub are subject to change. To access the code version that comes with this thesis (see attached DVD), a Git branch "thesis-version" was made in the repository.

6.2.2 Live version

Live version of the Inbox application is deployed using GitHub pages at

<https://whyineedtofillusername.github.io/inbox/>.

6.2.3 Requirements

In this section software needed to run the Inbox application is described. You need:

- node.js (with included npm);
- Yarn package manager.

6.2.4 Installation and build

Run the following command in the **root folder**:

```
yarn install
```

It installs all project dependencies.

After install, you can use any of the "goals" specified in the `package.json` file. More specifically, to build the application use the following command in the **root folder**:

```
yarn build
```

6.2.5 Running the application

Use predefined goals in `package.json` to run the application. You can either build the application yourself and then deploy it on a web server, or use the Angular's `ng serve` to run an embedded development web server:

```
yarn serve
```

The application should be now running. Navigate to `http://localhost:4200/` in your browser.

6.2.6 Tests

As described at chapter 5, the Inbox application contains both unit tests and E2E tests.

6.2.6.1 Unit tests

As is typical for the Angular framework, unit tests for the Inbox application are located in the same folder as the class being tested, with `*.spec.ts` extension. E.g. the class `MonitorInboxesService` is located in the `\src\app\services\monitor` ↪ `-inboxes` folder, the file `monitor-inboxes.service.ts` contains the class code and the file `monitor-inboxes.service.spec.ts` contains the unit tests.

In order to run the unit tests, you can use the predefined goals from `package` ↪ `.json`. To run and debug tests, the "test" goal is useful, as it starts the Karma test runner and tests are executed with each code change:

```
yarn test
```

For continuous integration (CI), the headless test mode is more useful, as it does not start a browser window and closes after tests finish:

```
yarn test-headless
```

This is done using `--watch=false` and `--browsers=ChromeHeadless` parameters. See `\karma.conf.js` file for further unit test configuration.

6.2.6.2 E2E tests

End-to-end tests mock a typical user behavior and cover application integration with other services. E2E tests for the Inbox application are located in the `\e2e` folder. Use

```
yarn e2e
```

Please note that a valid test user is required to run this test ("test-user1" is used).

6.2.7 Deployment

In order to deploy the application, first you need to build it with a `prod` flag. You can use the prepared deploy step:

```
yarn build-prod
```

In order to deploy the application on a URL with some path (e.g. `www.application/your-path`), you need to specify the `--base-href /your-path` ↪ `/` option. See the `build-prod-github` step:

```
yarn build-prod-github
```

Then you can proceed by deploying the built application to a web server of your choice. The built application is located in the `\dist` folder.

6.2.8 Continuous integration (CI)

To avoid introducing bugs during development, a continuous integration was set up. Taking advantage of the GitHub actions⁵⁵, unit tests and E2E tests are executed with every Git Push to the hosted repository. The application is installed, built and tests are executed in the headless mode. For details see the GitHub actions configurations file located in the source code at `\.github` \leftrightarrow `\workflows\node.js.yml`. The test results are available at the repository's Action tab at <https://github.com/WhyINeedToFillUsername/inbox/actions>.

6.3 Developer documentation

This section documents application for developers and contributors. It contains information about development environment and project structure. The section informs developer on how to modify or add functionality to the Inbox application.

6.3.1 Development environment

The Inbox application is a Angular application written in web technologies TypeScript/HTML/CSS. A web development environment (IDE) is suggested for the application's development. The Inbox application and all research and POC applications in this thesis were developed in the JetBrains' IntelliJ IDEA⁵⁶ IDE.

Inbox application development doesn't require any special tools and environment setup.

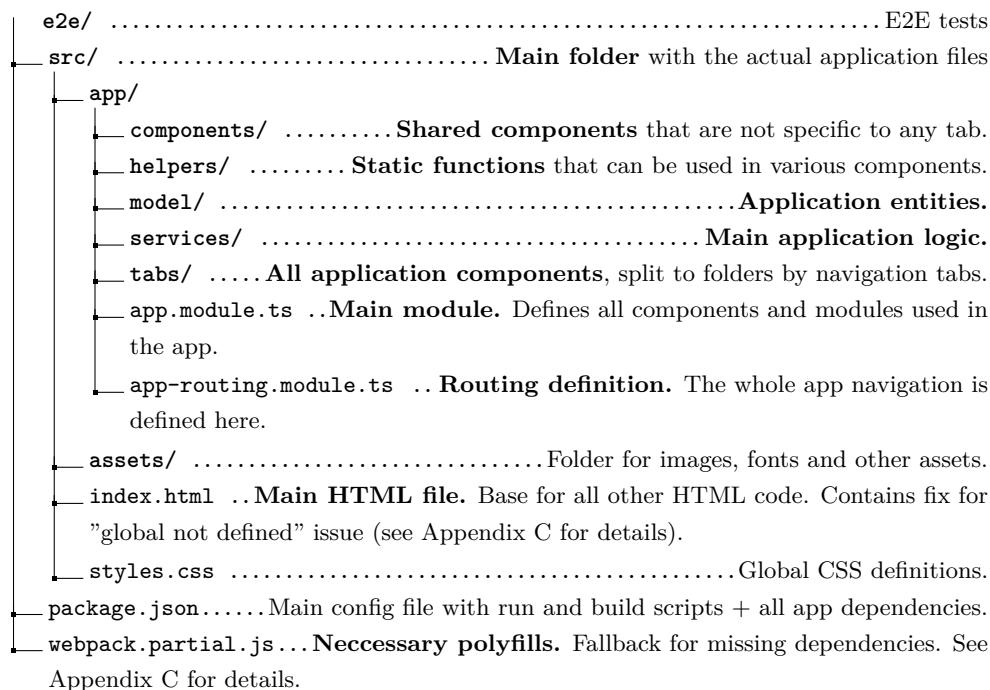
6.3.2 Project structure

The Inbox application follows standard Angular project structure:

⁵⁵<https://github.com/features/actions>

⁵⁶<https://www.jetbrains.com/idea/>

Figure 6.14: **Project structure.** Only notable files and folders are shown (e.g. ones that are not standard part of Angular or are important for development).



6.3.3 Contributing to Inbox

There are 2 options for a developer who wants to contribute to the Inbox application (see section 6.2 for the link to the GitHub repository):

1. work on a private copy of the application, obtained by forking the code on GitHub or copying the sources manually;
2. contribute to the existing GitHub repository using merge requests.

There are no special requirements in order to add and modify the application functionality.

Conclusion

In support of Web re-decentralization, the goal of this thesis was to get familiar with Linked Data, the RDF data model, the W3C Recommendations Linked Data Platform, Linked Data Notifications, ActivityPub, and the Solid project. Furthermore, the goal was to analyse current applications. Based on the analysis, an existing application was then to be enhanced or a new one implemented to produce a user-friendly messaging application.

Firstly, the current technologies that address the centralization problems were studied and described.

Secondly, requirements for the result application were formulated. Actors in the system were identified and, from the requirements, a list of use cases was derived.

Based on the requirements and use cases, a thorough analysis of existing solutions and applications was conducted. The result is that the solid-server is a sufficient solution for the application server, but no user-friendly messaging client is available. The solid-server has been improved by collaboration with its developers.

Next, the technologies to create the client application were researched and described in the Design section. Based on this research, multiple proof-of-concept applications were implemented to test these technologies for further use. The following proof-of-concept applications were created:

- **LDN-inbox** - an implementation of the Linked Data Notifications protocol. The resulting application was successfully tested with LDN test suite. The source codes were published as open-source on GitHub.
- **ldn-target-showcase** - a simple POC implementation of the Linked Data Notifications Target. Its goal is to showcase the LDN Discovery process to the community. The source code was published as open-source on GitHub and the application deployed for public access: <https://ldn-target-showcase.herokuapp.com>.

- **js-notification-poc** - an implementation of the JavaScript Notification API and Push API, developed to become familiar with the specifications and test the APIs. The source code was published as open-source on GitHub, and the application deployed for public access: <https://js-notification-poc.herokuapp.com/>.

Based on the previous analysis and technology review, the **main client application developed**:

Inbox is a new, user-friendly client for Linked Data Notifications and ActivityPub messaging. The application is an Angular application using solid-server as its back-end. It utilizes Solid WebID for user authentication and authorization, Solid PODs for data storage. The application allows its users to add their LDP inboxes for monitoring and get system notifications for new messages.

Inbox has been tested - automated tests and UX testing, documented and published as open-source on GitHub. Furthermore, the application has been deployed for a public use: <https://whyineedtofillusername.github.io/inbox/>.

Problems encountered

During the analysis and development, several major problems were encountered. The first problem occurred when developing the LDN-inbox proof-of-concept. The RDF library `rdfib.js` had problems with JSON-LD serialization/deserialization. The problem was discussed in Gitter Solid Chat and reported on GitHub. In the Inbox application, the problem was avoided by using another RDF library.

Another related problem is with the current technology and libraries for working with RDF data, Solid and Solid Pods. As the W3C recommendations for this area are relatively new, the technologies are still in the rapid development and work-in-progress phases. Documentation is sparse, and often insufficient.

This can be seen on the next major issue - Solid libraries not working in Angular framework. More specifically, the `solid-client-js` library brakes build with the Angular's default package manager `npm`. This issue has been resolved by using an alternative package manager `yarn` and using custom build options. For details see subsection 4.1.1.5.

Another case of these immaturities are various `solid-server` issues, the most pressing of them being Solid server sending phantom WebSocket messages, causing false notifications about new incoming message. For the full list of problems that were reported to the library and other software authors and community, please see Appendix C.

Lastly, the ActivityPub (AP) Client-to-server (C2S) server-side API has very few and limited implementations. For example, the C2S API is regarded as "incredibly bare-bones" by Mastodon (AP implementation) developers [27], who decided it is easier to implement a proprietary REST API. Thus the Inbox AP client part offers only basic functionality.

Future work

Future work should focus on further development of the Inbox application. The main focus should be on improving application performance - do not load all messages at once, but use pagination. Also, the application cache should be introduced in order to reduce number of HTTP requests to the Solid POD and shorten loading times of particular messages.

The next steps should also include adding new key functionalities such as keeping sent messages and introducing a private contact list (as opposed to using the public `foaf:knows` list). To make application accessible on as many devices as possible, UI responsiveness for various screen widths should be introduced.

Last but not least, reported GitHub issues should be monitored and Inbox application updated when the bugs are resolved, (most pressingly losing session after page reload and phantom WebSocket messages).

Bibliography

- [1] Abbate, J. E. *From ARPANET to Internet: A history of ARPA-sponsored computer networks, 1966-1988*. [cit. 2020-07-18].
- [2] Hindman, M. *The Internet Trap: How the Digital Economy Builds Monopolies and Undermines Democracy (2018)*. Princeton University Press, 2018, ISBN 9780691159263. Available from: <https://books.google.co.ke/books?id=hmmYDwAAQBAJ&printsec=frontcover&dq=monopoly+of+the+internet&hl=en&sa=X&ved=0ahUKEwj6483N3Zv1AhWIh1wKHTTqAIkQ6AEIJzAA#v=onepage&q=monopoly%20&f=false>
- [3] Peng, Z. *Decentralized Internet [online]*. [cit. 2020-07-18]. Available from: <https://www.cse.wustl.edu/~jain/cse570-19/ftp/decentrl/index.html>
- [4] World Wide Web Consortium (W3C). *Social Network Silos [online]*. [cit. 2020-07-12]. Available from: [https://www.w3.org/2010/Talks/0303-socialcloud-tbl/#\(2\)](https://www.w3.org/2010/Talks/0303-socialcloud-tbl/#(2))
- [5] World Wide Web Consortium (W3C). *Resource Description Framework (RDF) [online]*. [cit. 2020-03-29]. Available from: <https://www.w3.org/RDF/>
- [6] World Wide Web Consortium (W3C) Working Group. *RDF 1.1 Primer [online]*. [cit. 2020-07-17]. Available from: <https://www.w3.org/TR/rdf11-primer/#section-triple>
- [7] W3C JSON-LD Working Group. *JSON for Linking Data [online]*. [cit. 2020-07-13]. Available from: <https://json-ld.org/>
- [8] Berners-Lee, T. *Linked Data [online]*. World Wide Web Consortium (W3C), [cit. 2020-07-12]. Available from: <https://www.w3.org/DesignIssues/LinkedData.html>

- [9] World Wide Web Consortium (W3C), Linked Data Platform Working Group. *Linked Data Platform [online]*. [cit. 2020-07-17]. Available from: <https://www.w3.org/TR/ldp/>
- [10] World Wide Web Consortium (W3C), Linked Data Platform Working Group. *Linked Data Platform Resources [online]*. [cit. 2020-07-17]. Available from: <https://www.w3.org/TR/ldp/#ldpr>
- [11] World Wide Web Consortium (W3C), Linked Data Platform Working Group. *Linked Data Platform Containers [online]*. [cit. 2020-07-17]. Available from: <https://www.w3.org/TR/ldp/#ldpc>
- [12] World Wide Web Consortium (W3C). *Linked Data Notifications [online]*. [cit. 2019-09-18]. Available from: <https://www.w3.org/TR/ldn/>
- [13] World Wide Web Consortium (W3C). *Linked Data Notifications - Discovery [online]*. [cit. 2020-07-24]. Available from: <https://www.w3.org/TR/ldn/#discovery>
- [14] World Wide Web Consortium (W3C), Social Web Working Group. *Activity Streams 2.0 [online]*. [cit. 2020-07-20]. Available from: <https://www.w3.org/TR/activitystreams-core/>
- [15] Castaño, A. *What is ActivityPub? [online]*. [cit. 2020-07-20]. Available from: <https://alexcastano.com/what-is-activity-pub/>
- [16] *This Week in Solid 2019-12-12 [online]*. [cit. 2020-04-14]. Available from: <https://solidproject.org/this-week-in-solid/2019-12-12>
- [17] *What is a Pod? [online]*. [cit. 2020-07-22]. Available from: <https://solidproject.org/faqs#pod>
- [18] Moody, G. *Tim Berners-Lee unveils next step for Solid, a decentralized Web platform with privacy built-in as standard [online]*. [cit. 2020-07-22]. Available from: <https://www.privateinternetaccess.com/blog/tim-berners-lee-unveils-solid-a-decentralized-web-platform-with-privacy-built-in-as-standard/>
- [19] *About Carbon LDP implementation [online]*. [cit. 2020-06-07]. Available from: https://www.w3.org/wiki/LDP_Implementations#Carbon_LDP_.28Client_and_Server.29
- [20] *maytkso, HTTP server and command-line RDF tool to get/send, serialise data. [online]*. [cit. 2020-06-07]. Available from: <https://github.com/csarven/mayktso>
- [21] Internet Engineering Task Force (IETF). *RFC 7252, The Constrained Application Protocol (CoAP) [online]*. [cit. 2020-06-07]. Available from: <https://tools.ietf.org/html/rfc7252>

-
- [22] *Solid [online]*. [cit. 2020-07-29]. Available from: <https://solid.mit.edu/>
- [23] *Solid license [online]*. [cit. 2020-07-29]. Available from: <https://github.com/solid/node-solid-server/blob/master/LICENSE.md>
- [24] The Apache Software Foundation. *Apache Marmotta [online]*. [cit. 2020-07-29]. Available from: <https://marmotta.apache.org/>
- [25] The Apache Software Foundation. *Download Apache Marmotta [online]*. [cit. 2020-07-29]. Available from: <https://marmotta.apache.org/download.html>
- [26] The Apache Software Foundation. *Apache Marmotta - Open issues [online]*. [cit. 2020-07-29]. Available from: <https://issues.apache.org/jira/projects/MARMOTTA/issues/MARMOTTA-674?filter=allopenss>
- [27] tootsuite/mastodon developers. *ActivityPub client-to-server support - GitHub issue [online]*. [cit. 2021-04-05]. Available from: <https://github.com/tootsuite/mastodon/issues/10520>
- [28] Nielsen Norman Group. *Summary of Usability Inspection Methods [online]*. [cit. 2020-07-30]. Available from: <https://www.nngroup.com/articles/summary-of-usability-inspection-methods/>
- [29] Interaction Design Foundation. *How to Conduct a Cognitive Walk-through [online]*. [cit. 2020-07-30]. Available from: <https://www.interaction-design.org/literature/article/how-to-conduct-a-cognitive-walkthrough>
- [30] *Bootstrap (front-end framework) [online]*. [cit. 2020-07-24]. Available from: [https://en.wikipedia.org/wiki/Bootstrap_\(front-end_framework\)](https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework))
- [31] Inrupt. *Tripledoc GitLab - project page [online]*. [cit. 2020-07-24]. Available from: <https://gitlab.com/vincenttunru/tripledoc>
- [32] Inrupt. *Tripledoc GitLab - first commit [online]*. [cit. 2020-07-24]. Available from: <https://gitlab.com/vincenttunru/tripledoc/-/commit/802f3661920dddedf34120d4c07cacb8d4a49c94>
- [33] Mozilla and individual contributors. *MDN web docs - Push API [online]*. [cit. 2020-07-24]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API
- [34] Spyna, L. *Push Notifications in JavaScript? Yes, you can! [online]*. [cit. 2020-07-24]. Available from: <https://itnext.io/an-introduction-to-web-push-notifications-a701783917ce>

BIBLIOGRAPHY

- [35] Mozilla and individual contributors. *MDN web docs - Push API [online]*. [cit. 2020-07-24]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Push_API
- [36] Copes, F. *The Push API Guide [online]*. [cit. 2020-07-24]. Available from: <https://flaviocopes.com/push-api/>
- [37] *Solid-auth-client GitHub [online]*. [cit. 2020-07-29]. Available from: <https://github.com/solid/solid-auth-client>
- [38] SLANT TEAM. *JavaScript E2E test framework comparison [online]*. [cit. 2020-07-30]. Available from: <https://www.slant.co/versus/9648/20624/~nightwatch-js-vs-cypress>

Glossary

AP ActivityPub

AS Activity Streams

ČVUT České vysoké učení technické v Praze

GUI Graphical User Interface

IRI Internationalized Resource Identifier

JS JavaScript

LD Linked Data

LDN Linked Data Notifications

LDP Linked Data Platform

LDPC Linked Data Platform Container

LDPR Linked Data Platform Resources

NPM Node Package Manager

POC Proof of concept

R/W Read/Write

RDF Resource Description Framework

REST API Representational State Transfer Application Program Interface

SPARQL SPARQL Protocol and RDF Query Language

SSO Single Sign-On

URI Uniform Resource Identifier

A. GLOSSARY

UX User Experience

VCS Version Control System

VPN Virtual Private Network

W3C World Wide Web Consortium

WIP Work-In-Progress

Technical research - proof-of-concept applications

This appendix contains description of all POC applications developed as part of the technical research for the final client Inbox application. Also, the first iteration of the Inbox client application is described here.

B.1 LDN-inbox - LDN proof-of-concept

In this section, the LDN proof-of-concept (POC) application called originally "inbox" is described. First, the architecture is discussed, followed with a description of used technologies. This POC web application was created to investigate the LDN protocol. Its purpose is to test the architecture, technologies and the LDN, RDF libraries.

B.1.1 Architecture

Based on the actors of the LDN protocol, the application is divided into three modules (see Figure 1.3):

- **consumer** - LDN consumer + sender,
- **receiver** - LDN receiver,
- **target** - sample LDN target for inbox discovery.

The consumer is designed to communicate using REST API with any application compliant with the LDN specification. The receiver is a REST API server with in-memory DB implementation to receive and serve notifications using LDN protocol. The target is an implementation of the LDN target and works for the inbox discovery.

Both consumer and server are designed to have a back-end with HTML/CSS/JS front-end.

The consumer and receiver are an MVC architectures with RDF as a model and a separate service layer.

B.1.2 Technologies

After the analysis, JavaScript was chosen as the language for development. The main reasons are the requirement of a web client with desktop notification and the lack of back-end libraries. Only these four back-end libraries were discovered:

- **rdflib**⁵⁷ for Python,
- Java:
 - **Apache Jena**⁵⁸,
 - **Eclipse RDF4J**⁵⁹;
- **EasyRDF**⁶⁰ for PHP.

server For this application, the Node.js⁶¹ server was used with the Express⁶² web framework. Node.js is an asynchronous, event-driven JavaScript engine for back-end implementations. Express is a simple web framework built on top of Node.js.

front-end Front-end is made of HTML/CSS/JS pages. The Bootstrap framework was used to help with the UI design. Bootstrap is an open-source CSS/JS framework [30].

B.1.2.1 LD/RDF libraries

To read, write and manipulate Linked Data in RDF, a JavaScript library is needed. There are not many available libraries and the existing ones are still in development. For example, while working on this thesis, an issue⁶³ with the rdflib⁶⁴ was encountered - the library had troubles parsing JSON-LD.

The following LD/RDF libraries were considered and examined:

⁵⁷<https://rdflib.readthedocs.io/en/stable/>

⁵⁸<https://jena.apache.org/>

⁵⁹<https://rdf4j.org/>

⁶⁰<https://www.easyrdf.org/>

⁶¹<https://nodejs.org/>

⁶²<https://expressjs.com/>

⁶³<https://github.com/linkedin/rdflib.js/issues/364#issuecomment-546705383>

⁶⁴<https://github.com/linkedin/rdflib.js/>

rdflib *rdflib.js* is an RDF JavaScript library. It supports R/W with RDF/XML and Turtle RDF serializations. It also supports reading of JSON-LD. Furthermore, it contains a fetch API to access RDF resources and local store with API to query the result.

Rdflib.js sources and documentation is accessible at <https://github.com/linkedin/rdf-lib.js/>. It is also available as a npm⁶⁵ (JavaScript package manager) at <https://www.npmjs.com/package/rdf-lib>.

Tripledoc *Tripledoc* is an RDF JavaScript library to read, create and update documents on a Solid Pod [31]. It has a more intuitive and easy-to-understand interface than the rdflib, however fewer capabilities.

Please note that it was not available at the time of development of the inbox POC (first commit is Jul 17, 2019 [32]). For this reason it was not considered for this POC, but was later used in the inbox-client application. <https://vincenttunru.gitlab.io/tripledoc/>,

Shighl <https://github.com/scenaristeur/shighl>,

LDflex <https://github.com/LDflex/LDflex>.

The rdflib was chosen as the most mature technology. To access solid pods, the solid-auth-client⁶⁶ library is needed.

B.1.3 Implementation

Here, the application implementation details are described.

The application flow is:

1. user visits consumer+sender application
2. the welcome page is shown: Figure B.1
3. user inputs target URL or click on one from the last used list
4. consumer+sender module performs inbox discovery
5. if successful, consumer+sender reads messages from the inbox

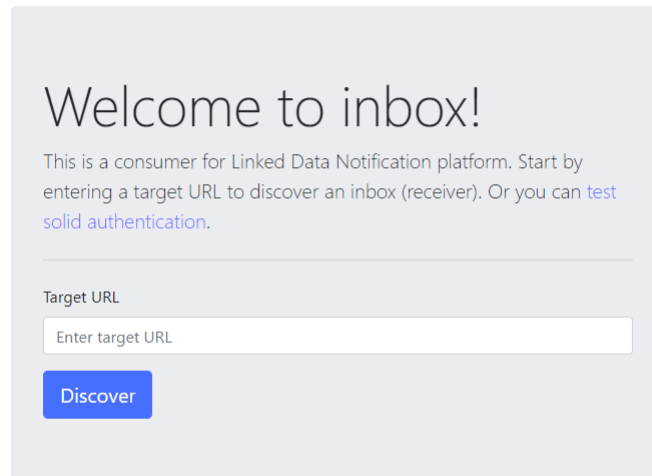
The inbox can be located anywhere. In this application, it is located at the receiver module.

⁶⁵<https://docs.npmjs.com/about-npm/>

⁶⁶<https://github.com/solid/solid-auth-client>

Figure B.1: **inbox - index screen**. Screenshot of the index page

Inbox



Last used:



B.1.3.1 consumer + sender

The main functionality is in the application services:

- `receiverServices.js`
 - `getNotifications(inboxUrl, callback)` retrieve all available notifications from the inbox/target, then execute callback
 - `getNotificationById(id, callback)` retrieve a specific notification from the inbox/target by supplied id, then execute callback
- `targetService.js`
 - `discoverInboxAt(urlToExplore, callback)` attempt to discover inbox at the urlToExplore
 - `getInboxUrlFromRDF(urlToExplore, callback)` when no link header is available, try to get the inbox url from RDF content

B.1.3.2 receiver

The main functionality is in the application service:

- notificationService.js
 - processMessage (notification)
 - createAllNotificationsResponse() produce valid JSON-LD envelope for the notifications from DB
 - getNotificationById()

B.1.3.3 target

Target is a simple web page to try and showcase LDN discovery [13]. It is implemented as a node.js/express application that responds to HTTP requests.

As described at Figure 1.1.4, there are two options for discovery - using HTTP Link header⁶⁷, for both HEAD and GET methods, or by embedding RDF into the resource content.

For the first option (HTTP Link header), the application accepts both GET and HEAD HTTP requests:

```
const LINK_VALUE = '<' + INBOX_URL + '>';
rel="http://www.w3.org/ns/ldp#inbox";

router.get('/', function (req, res, next) {
  res.set('Link', LINK_VALUE);
  res.render('index', {title: 'Inbox_discovery_demo'});
});

router.head('/', function (req, res, next) {
  res.set('Link', LINK_VALUE);
  res.status(200).end();
});
```

The second option is returning content based on the Accept header. It either returns RDF in JSON-LD or Turtle, or return HTML with embedded RDF:

```
// 2) RDF
router.get('/content', function (req, res, next) {
  // switch response based on Accept header and set the response content
  ⇨ type accordingly
  res.format({
    // a) JSON with relation of type http://www.w3.org/ns/ldp#inbox
    'application/ld+json': function () {
      res.send(
        {
          "@context": "http://www.w3.org/ns/ldp",
          "@id": "https://tonda.solid.community/",
          "inbox": "https://tonda.solid.community/inbox/"
        }
      );
    }
  });
});
```

⁶⁷<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Link>

```
    )
  },

  // b) HTML <a> with rel="http://www.w3.org/ns/ldp#inbox"
  // c) HTML <link> with rel="http://www.w3.org/ns/ldp#inbox"
  // d) HTML <section> with property="http://www.w3.org/ns/ldp#inbox"
  'text/html': function () {
    res.render('contentWithRdf');
  },

  // e) text/turtle with <http://www.w3.org/ns/ldp#inbox> relation
  'text/turtle': function () {
    res.send("<https://tonda.solid.community/>_<http://www.w3.org/
↵ ns/ldp#inbox>_<https://tonda.solid.community/inbox/>_");
  },

  default: function () {
    // log the request and respond with 406
    res.status(406).send('Not_Acceptable')
  }
})
});
```

B.1.4 Documentation

This section contains the LDN-inbox POC documentation.

B.1.4.1 Source code

Source code (with documentation) available at <https://github.com/WhyINeedToFillUsername/ldn-inbox>

B.1.4.2 Requirements

You need to install node.js (with included npm).

B.1.4.3 Install

Run the following command in the **module's root folder**:

```
npm install
```

It installs all project dependencies, for details see <https://docs.npmjs.com/cli/install>.

B.1.4.4 Run

consumer listens on local port 8000, *receiver* on 3000. You can change that in their bin/www files. Run each module separately using following command in the **module's root folder**:

```
npm start
```

The *consumer* requires the *receiver* to be running.

B.1.4.5 Usage

You can try the *consumer* in your browser at <http://localhost:8000/>. When you post

```
{"@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "to": ["https://chatty.example/ben/"],
  "attributedTo": "https://social.example/alyssa/",
  "content": "Say, do you think that Gary Webb and Jeffrey Epstein really
  ↔ killed themselves?"}
```

with `Content-Type: application/ld+json` to <http://localhost:5001/API/notifications>, the *receiver* will return 201 with location. You can then GET it at <http://localhost:5001/API/notifications/xx>. Or let the *consumer* display it in the browser at <http://localhost:8000/notification/xx>.

B.2 LDN-target

LDN-target is a simple web application that was separated from the original inbox POC. It was extracted to a new project and extended, so it is possible to document and deploy it separately. This way it can be a helpful contribution to the community.

B.2.1 Architecture

Based on the LDN target discovery protocol [13], the application has the following endpoints:

- HTTP HEAD that returns response with the Link header,
- HTTP GET that returns response with the Link header,
- HTTP GET that returns RDF content with serialization based on the HTTP content negotiation.

B.2.2 Technologies

The technology stack is the same as for the inbox POC.

server For this application, the Node.js⁶⁸ server was used with the Express⁶⁹ web framework.

front-end Front-end is made of HTML pages with CSS styling.

B.2.3 Documentation

This section contains the LDN-target POC/showcase documentation. The application is a simple Linked Data Notifications target implementation to test and showcase all discovery options. It is a node.js (express) app.

B.2.3.1 Source code

Source code (with documentation) available at <https://github.com/WhyINeedToFillUsername/ldn-target-showcase>

B.2.3.2 Requirements

You need to install node.js (with included npm).

B.2.3.3 Install

Run the following command in the **root folder**:

```
npm install
```

It installs all project dependencies, for details see <https://docs.npmjs.com/cli/install>.

B.2.3.4 Run

Run with

```
npm start
```

Application is listening on the port 3000.

⁶⁸<https://nodejs.org/>

⁶⁹<https://expressjs.com/>

B.2.3.5 Usage

Open browser at <http://localhost:3000/>. You can see all the options to try out. Use e.g. curl or Postman to make HTTP request with various options.

For example this request:

```
GET /content HTTP/1.1
Accept: application/ld+json
Host: localhost:3000
```

gets this response:

```
HTTP/1.1 200 OK
Content-Type: application/ld+json; charset=utf-8
{
  "@context": "http://www.w3.org/ns/ldp",
  "@id": "https://tonda.solid.community/",
  "inbox": "https://tonda.solid.community/inbox/"
}
```

B.2.3.6 Live version

Application is deployed to: <https://ldn-target-showcase.herokuapp.com>.

B.3 js-notification-poc

js-notification-poc is an implementation of the JavaScript Notification API and Push API. Its development was intended to get familiar with the specifications and test the APIs. The Notification API part of the application is based on the API documentation [33], the second, Push API, is directly derived from an existing Push API example [34] (its source code available at <https://github.com/Spyna/push-notification-demo/>).

B.3.1 Architecture

This POC is a simple static page. It is split to two conceptual parts, first being based on the JavaScript Notification API [33] and the second on the Push API [35]. Both are a simple web page, the whole application logic is written in JavaScript.

B.3.2 Technologies

The technology stack is the same as for the inbox POC.

server For this application, the Node.js⁷⁰ server was used with the Express⁷¹ web framework.

front-end Front-end is made of HTML pages with CSS styling. The core functionality is written in JavaScript.

B.3.3 Implementation

This POC is a simple static page to test JavaScript notification api. It runs on node.js (express) server. The POC was created based on the MDN documentation. Quite simply, first the application first checks if the browser supports the Notification API:

```
// Let's check if the browser supports notifications
if (!('Notification' in window)) {
  const message = "This browser does not support notifications.";
  console.error(message);
} else {
  // yes, supports, handle user's answer
}
```

After the user confirms that he wants to receive notifications, a notification is created with the following code:

```
function createSimpleNotification() {
  var img = '/images/fit.png';
  var text = 'Text of the notification.';
  var title = 'Hello there! Cool title!';
  const options = {
    body: text,
    icon: img,
    // A vibration pattern to run with the display of the notification.
    vibrate: [200, 100, 200],
    // An ID for a given notification that allows you to find, replace,
    ↪ or remove the notification using a script if necessary.
    tag: "new-product",
    image: img,

    // URL of an image to represent the notification when there is not
    ↪ enough space to display the notification itself
    badge:
      "https://spyna.it/icons/android-icon-192x192.png"
  };
  let notification = new Notification(title, options);
  // hook events, save reference etc.
}
```

⁷⁰<https://nodejs.org/>

⁷¹<https://expressjs.com/>

This creates a system notification with the requested parameters (the whole list is accessible at the MDN web docs: <https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration/showNotification#Parameters>).

B.3.3.1 Push API

The second part, Push API implementation, is directly derived from the *Push Notifications in JavaScript* article [34].

To make a full use of the JavaScript Notification + Push API, there are 6 steps [36]:

1. Check if Service Workers are supported
2. Check if the Push API is supported
3. Register a Service Worker
4. Request permission from the user
5. Subscribe the user and get the PushSubscription object
6. Send the PushSubscription object to your server

Also, to identify application and prevent spam for the clients, a VAPID key must be generated and used in the subscription. The web-push npm package was used for this purpose: <https://www.npmjs.com/package/web-push>.

Further implementation details are available in the *Push Notifications in JavaScript* article [34].

B.3.4 Documentation

js-notification-poc is an implementation of the JavaScript Notification API and Push API. It was developed to get familiar with the specifications and test the APIs.

B.3.4.1 Source code

Source code (with documentation) available at <https://github.com/WhyINeedToFillUsername/js-notification-poc>

B.3.4.2 Requirements

You need to install node.js (with included npm).

B.3.4.3 Install

Run the following command in the **root folder**:

```
npm install
```

It installs all project dependencies, for details see <https://docs.npmjs.com/cli/install>.

B.3.4.4 Build

Run the following command in the **root folder**:

```
npm build
```

It uses browserify to build the project javascript bundle files.

B.3.4.5 Run

The node.js server is set to listen on local port 3001. You can change that in the bin/www files. Start it by this command:

```
npm start
```

Then go to <http://localhost:3001/>. Click on the "Enable notifications" button to request permission, and "Create notification" to show system notification.

Please note that request for notifications won't work in browser "private" mode.

B.3.4.6 Live version

Application is deployed to: <https://js-notification-poc.herokuapp.com/>.

B.4 inbox-client

In this section, the first iteration of the final Inbox application called "inbox-client" is described. First, the user interface was designed. It is described in the main design chapter, see subsection 3.2.3. Here, the the application's architecture is discussed, followed with a description of used technologies.

B.4.1 Architecture

As for the inbox POC, JavaScript was used as the language for development. The main reasons are the requirement of a web client with desktop notification and the lack of back-end libraries.

The application logic is on the client side, with server used only for serving HTML/JS/CSS content. This is based on the findings from the development of the inbox POC section B.1. The reason is that the solid pods require the solid-auth-client library for authentication. This is available only as a client-side browser library [37].

All data are stored in the solid pod and the client-side JavaScript is using the tripledoc library's fetch api to create authenticated HTTP requests to access the pod.

B.4.2 Technologies

The technology stack is the same as in the inbox POC (see subsection B.1.2), with the main difference of using different LD/RDF library.

server For this application, the Node.js⁷² server was used with the Express⁷³ web framework.

front-end Front-end is made of HTML/CSS/JS pages. The Bootstrap [30] framework was used to help with the UI design.

B.4.2.1 LD/RDF libraries

To read, write and manipulate Linked Data in RDF, a JavaScript library is needed. The *tripldoc* RDF JavaScript library [31] was chosen for this application. It has more intuitive and easy-to-understand interface than the *rdflib* used in the inbox POC.

To access solid pods, the *solid-auth-client*⁷⁴ library is required.

The *rdf-namespaces* library (<https://www.npmjs.com/package/rdf-namespaces>) is used to help with RDF namespaces. This allows to use predefined constants like `rdfnamespaces.foaf.knows` instead of direct URLs like `http://xmlns.com/foaf/0.1/knows`.

B.4.3 Implementation

All the application's logic is in the front-end (browser) JavaScript. To structure the application, code is divided into separate modules based on their

⁷²<https://nodejs.org/>

⁷³<https://expressjs.com/>

⁷⁴<https://github.com/solid/solid-auth-client>

functionality. During the build, JS code is processed by the browserify plugin. A separate JS bundle file is created for each page. This way only used code is being loaded.

This application uses the solid-auth-client to authenticate and *tripleDoc* RDF JavaScript library [31] to read and manipulate solid pod data.

B.4.3.1 Modules

Below is a list of the application JavaScript modules with description.

- `alerts.js` a helper module with `addAlert(level, message, autoDismiss)` function to create a user-friendly alerts
- `inbox.js` main module with the logic to add a watched inbox, retrieve notifications from it, show them to user and detect new messages. Every ten seconds, it loads notifications from the monitored inboxes and detects any new messages: `window.setInterval(loadNotifs, 1000 * 10);`
 - `loadMonitoredInboxesFromPod(webID)` retrieve monitored inboxes from the solid pod:
 1. fetch solid profile
 2. on the profile, get/create document that stores the watched inboxes list
 3. from the document, get all subjects of class `schema.URL`
 4. each has the url saved as type string; save them to memory and call `addInboxToShownList()`
- `inbox-detail.js` module for loading and formatting the inbox messages. Also contains methods for remove inbox from monitored list.
- `inbox-discover.js` module for LDP inbox discovery on the supplied LDP target
- `inbox-send.js` contains methods for sending the messages using LDN protocol (with AS format)
- `notifications.js` contains methods for handling system notifications using the JavaScript Notification API
- `pod.js` using the *tripleDoc* library [31], this module contains methods for the communication and data retrieval from the solid pods.
 - `getFriends(webID)` retrieve a list of user contacts (predicate `foaf.knows`)

- `getWatchedInboxesListDocument(profile)` retrieves the document where list of watched inboxes. If it does not exist in the client's profile, it is created by calling `initialiseWatchedInboxesList()` method.
 - `initialiseWatchedInboxesList(profile, typeIndex)` creates an empty document for storing the watched inboxes list
 - `addWatchedInbox(inbox, watchedInboxesListDoc)` stores a watched inbox into the supplied document
 - `removeWatchedInbox(inboxIRI, watchedInboxesListDoc)` removes a watched inbox from the supplied document
-
- `solid-login.js` contains logic for solid login, using the `solid-auth-client` library

 - `solid-logout.js` functionality for the logout button

B.4.3.2 User interface

By using the Bootstrap CSS/JS framework [30], the UI is fully responsive. Below are few screenshots of the UI for illustration:

Figure B.2: inbox-client screen 1 - watched inboxes

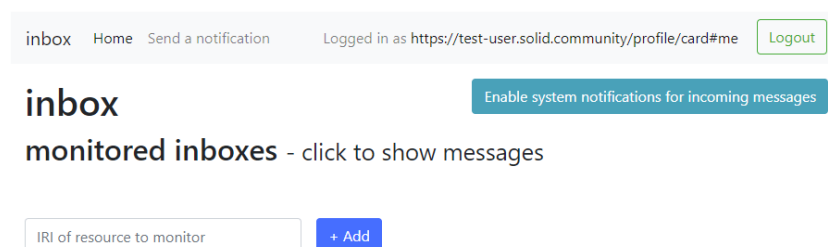


Figure B.3: inbox-client screen 2 - added watched inbox

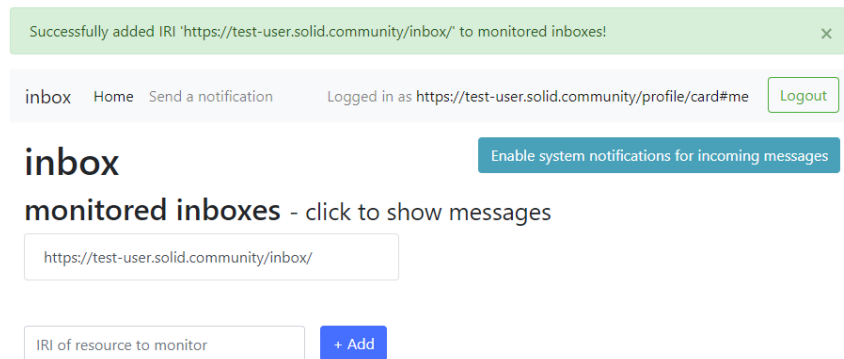
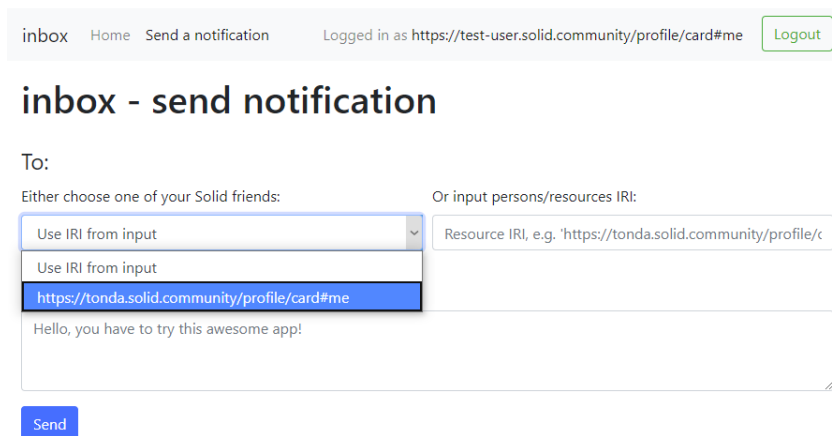


Figure B.4: inbox-client screen 3 - send message to a friend



B.4.4 Testing

This section describes testing of the inbox-client application.

B.4.4.1 Unit tests

For this application, a unit testing framework research was conducted. From the candidates the Mocha testing framework (<https://mochajs.org/>) was selected. It is a JavaScript test framework running on Node.js.

However, during the test development it occurred that the inbox-client application has no back-end services. Because of the used available solid and RDF libraries, all the application logic is on the front-end. The created front-end JavaScript methods are not suitable for unit tests - they either directly

modify HTML DOM or rely on the solid session, which is created with the external solid-client-auth library.

B.4.4.2 E2E tests

For the inbox-client, a E2E testing framework research was conducted. From the candidates, the Cypress E2E testing framework (<https://www.cypress.io/>) was selected as the most suitable framework [38]. It is an open source, JavaScript test framework running on Node.js and in a browser.

However a problem with the selected testing framework occurred. It does not support pop-up windows. And as of time of writing the thesis, a pop-up windows is the only way the solid-client-auth library is able to work. As a result, the testing framework is unable to log into the application This is an essential step in each UC and so the E2E tests are not part of the final solution.

B.4.4.3 Usability testing

The key part of the thesis is to prove that the solid inbox application can be user-friendly. To ensure this, a usability testing was conducted. Because the application's typical user is an experienced user, the cognitive walk-through was selected as the most suitable usability testing method.

B.4.5 Documentation

Documentation for inbox-client. The application is designed as a JavaScript client-side application with Node.js/express web framework back-end.

B.4.5.1 Source code

Source code (with documentation) available at <https://github.com/WhyINeedToFillUsername/inbox-client>

B.4.5.2 Requirements

You need to install node.js (with included npm).

B.4.5.3 Install

Run the following command in the **root folder**:

```
npm install
```

It installs all project dependencies, for details see <https://docs.npmjs.com/cli/install>.

B.4.5.4 Build

Run the following command in the **root folder**:

```
npm build
```

It uses browserify to build the project javascript bundle files.

B.4.5.5 Run

The node.js server is set to listen on local port 3000. You can change that in the bin/www file. Start it by this command:

```
npm start
```

Then go to <http://localhost:3000/>. Click on the "Enable notifications" button to request permission.

Please note that request for notifications won't work in browser "private" mode.

B.4.5.6 Usage

Open browser at <http://localhost:3000/>.

Please note that you have to add the running url "http://localhost:3000" (exactly like this, with no trailing slash) to your trusted applications in your solid.community profile preferences with Read, Write and Append rights.

B.4.5.7 Live version

Application is deployed to: <https://inbox-client.herokuapp.com/>.

Please note that you have to add "https://inbox-client.herokuapp.com" (exactly like this, with no trailing slash) to your trusted applications in your solid.community profile preferences with Read, Write and Append rights.

Or you can use the following test user:

- **Username** "test-user"
- **Password** "SolidCommunity@2020"

Reported GitHub/GitLab issues

This appendix contains list of issues of various libraries that have been discovered and reported during development for this thesis. Both GitHub and GitLab repositories were used for issue tracking.

C.1 Problems with solid-client-authn-js library

This section describes problems with the @inrupt/solid-client-authn-browser library (GitHub repository at <https://github.com/inrupt/solid-client-js>, npm package at <https://www.npmjs.com/package/@inrupt/solid-client-authn-browser>).

C.1.1 Problem using the library in Angular

This is a major issue for the library - including the @inrupt/solid-client-authn-browser npm package in an angular package.json breaks its build. Originally reported at <https://github.com/inrupt/solid-client-js/issues/608>, the issue has not been properly resolved and a workaround had to be found in order to use the library in the angular application.

C.1.2 Library producing unsolicited request with 404 error

Closed with "working as expected". <https://github.com/inrupt/solid-client-authn-js/issues/981>

C.1.3 session.info.webId not available in onLogin callback

This issue, reported at <https://github.com/inrupt/solid-client-authn-js/issues/955>, was swiftly resolved and a new version of the solid-client-authn-browser library was released.

C.1.4 Library does not stay logged in after page reload

The user does not stay logged in after page refresh. Before reporting this issue, an existing GitHub issue was found: <https://github.com/inrupt/solid-client-authn-js/issues/423>. The problem and possible workarounds were discussed there (see comments by the thesis author at <https://github.com/inrupt/solid-client-authn-js/issues/423#issuecomment-741646705>).

C.2 Solid server issues

This section describes problems with the Solid server (GitHub repository at <https://github.com/solid/community-server>). As the server implementations deployed at e.g. inrupt.net are using its node.js implementation, most issues were moved to the correct repository at <https://github.com/solid/node-solid-server/issues/>.

C.2.1 Server sends phantom WebSocket pub messages

The solid-server is sending phantom WebSocket messages for subscribed resource (e.g. inbox) even when no CRUD operation has occurred on the resource. Reported at <https://github.com/solid/node-solid-server/issues/1587> with no response from the developers.

C.2.2 GET <https://tonda.inrupt.net/inbox> times out

This issue occurs for the specific profile <https://tonda.inrupt.net/>. The profile seems to gotten corrupted somehow, but no informative error is presented, there is no action the user can do to fix it. There has been no reaction from the server developers. Originally reported at <https://github.com/solid/community-server/issues/546>, the issue has been moved to the node.js server implementation GitHub tracker at <https://github.com/solid/node-solid-server/issues/1558>.

C.2.3 Solid sends WebSockets messages for private resources without authentication

For example an inbox <https://inbox1.inrupt.net/inbox/> is not accessible without authentication (HTTP GET returns 401). But anybody can subscribe to it without any authentication/authorization. <https://github.com/solid/solid-spec/issues/232>

C.2.4 Solid uses incompatible WebSockets protocol version

Solid used 'solid/0.1.0-alpha' WS protocol name. As reported at <https://github.com/solid/specification/issues/163#issuecomment-759725>

245, this is not a valid WS protocol name and is incompatible with Google Chrome. Before authors fixed it (see <https://github.com/solid/solid-spec/issues/221>), using 'solid.0.1.0-alpha' version worked as a work-around. The latest version is `solid-0.1`.

C.2.5 POSTing ActivityPub message to Solid produces wrong content-type

POSTing message to solid inbox (inrupt.net) with content-type 'application/ld+json; profile="https://www.w3.org/ns/activitystreams"' (official ActivityPub content-type from its specification, at <https://www.w3.org/TR/activitypub/#client-to-server-interactions>) produces 'application/octet-stream' content-type when GETting the message. Reported at <https://github.com/solid/node-solid-server/issues/1574>.

C.3 Other repositories

This section lists issues reported in another repositories.

C.3.1 LDN tests page unavailable

The LDN test suite originally available at <https://linkedresearch.org/ldn/tests/> stopped working - it returned "503 Service Unavailable" error. The issues has been reported (<https://github.com/csarven/ldn-tests/issues/19>) and swiftly resolved.

C.3.2 Cannot verify Pleroma OAuth token

Pleroma was forked from Mastodon. The issue (originally reported at <https://git.pleroma.social/pleroma/pleroma-support/-/issues/49>, moved to <https://git.pleroma.social/pleroma/pleroma/-/issues/2282>) is that Pleroma is using different way of obtaining the OAuth token. The issue was closed with "working as expected" status.

C.3.3 Mastodon offers little to no ActivityPub client-to-server support

The social network Mastodon offers very limited ActivityPub client-to-server support. Based on this GitHub issue, the developers find it "incredibly bare-bones" and chose to implement a proprietary REST API. <https://github.com/tootsuite/mastodon/issues/10520>

C.3.4 solid-auth-fetcher - missing method implementation

The library is missing `getSessions()` method implementation. The method is promoted in the library's readme file. <https://github.com/solid/solid-auth-fetcher/issues/19>

C.3.5 Questions in forums

Various topics have been discussed in few forums, e.g. "How to get Objects based on Predicate from Thing": <https://forum.solidproject.org/t/how-to-get-objects-based-on-predicate-from-thing/3859>. This question has been quickly answered and stays as a proof that the community is active and collaboration with it is useful.

Complete results of cognitive walk-through

This appendix contains detailed results of the cognitive walk-through testing of the Inbox application. It is ordered by test cases as defined in the subsection 5.4.3. Each table contains answers (column "A" - answer) to the testing questions defined in the subsection 5.4.1 (identified in the column "Q" - question), together with tester comment and recommendation.

D.1 TC1 - Read list of messages from all available inboxes

Test the first page after user logs in.

Q	A	Comment	Recommendation
Q1	Yes	Users will expect the same behavior as in traditional e-mail inbox - to login and to see the messages immediately.	
Q2	Yes	Provider selection form with the "Login" button is clearly visible.	Add an onboarding / sign up flow for new users (this is a must before releasing it to the general public)
Q3	Yes	After successful login the user will see a list of messages. It is supported by heading "messages from all your inboxes". The list contain information about sender, first words of message's content, date and to which inbox it belongs	Information about each message's inbox should not be visually prominent. In e-mail communication users firstly check who sent the message, what is the subject and then are most interested about its content. Instead of showing part of the message content consider rendering the message's subject.

D. COMPLETE RESULTS OF COGNITIVE WALK-THROUGH

Q4	Yes	This is pretty clear as the experience is very similar to traditional e-mail inboxes.	The purple loading spinner could be smaller. Loading spinner in the inbox should be placed in the messages container to reflect that it is loading messages, and not the Reload button itself.
-----------	-----	---	--

D.2 TC2 - Read list of messages from selected inbox

Users can have multiple inboxes in their Solid profile. The application shows a list of all user inboxes in the left menu.

Q	A	Comment	Recommendation
Q1	Yes		
Q2	Yes	The inbox selection is placed in the left panel, which is similar to traditional e-mail clients (e.g. Apple's Mail)	The "Messages from all inboxes" option could be reduced to "All inboxes" and be visually separated from the single inboxes.
Q3	Yes	Each inbox item has a hover state, so the user expects it to be clickable.	
Q4	Yes	On inbox item click the list of available messages gets reloaded to show desired content	

D.3 TC3 - Read detail of a received message

Q	A	Comment	Recommendation
Q1	Yes	Users will expect to open the message's details by clicking the particular message in the inbox.	
Q2	Yes	There is a background color change hover effect on each message together with the cursor changing to a pointer. It is a standard way to tell users that the element is clickable.	
Q3	Yes		Message details should be part of the header and not to be below the message content. Generally do not reinvent the wheel, make it as similar to classic email clients as possible, so it can be adopted by the majority of users easily.
Q4	Yes		Loading spinner could be smaller and centered.

D.4 TC4 - Reply to message

Q	A	Comment	Recommendation
Q1	Yes	Users will look for a reply button on the message detail page.	Consider adding a button icon to each item in the inbox overview. It can be visible on hover only. In result it will be a nice shortcut for a quick reply.
Q2	Yes	Reply button is clearly visible on the message detail page.	
Q3	Yes	Clicking the Reply button will open a new message form with the prefilled recipient and quoted previous message. The type of message (simple message vs. Activity Streams message) respects the original message. Subject is a mandatory field. Send button is clearly visible	The form has inconsistency in background colors. Keep it the same as in inbox (main grey background and for content use white background to make it step out of the rest of the page).
Q4	Yes	A confirmation toast message is shown on the bottom of the page when the message is sent.	After sending the message, redirect the user back to the inbox as writing another reply to the same user is an improbable action. A success toast can contain green color which represents success (and the errors should be red).

D.5 TC5 - Send a simple message

Send a simple message to an unknown person.

Q	A	Comment	Recommendation
Q1	Yes		
Q2	Partly yes	It might take few second before user finds the “send message” option in the top navigation.	Change the label to “New message” and add an icon to it to make it more visible as it is one of the primary user actions. Also there can be new message icon in the same container as reload button (can be a shortcut to send a message from actual inbox)
Q3	Yes	When new message forms appear, the “Activity Streams message” is selected by default. Therefore the user has to switch it to “Simple message”. The form is the same as in UC4.	The simple message misses the subject field. Is it by purpose?
Q4	Yes	Same as in UC4	

D.6 TC6 - Send an AP message

Send an Activity Streams message to a person from your contacts.

Q	A	Comment	Recommendation
Q1	Yes	Same as in UC5	
Q2	Yes	Same as in UC5	
Q3	Yes	When new message forms appear, the “Activity Streams message” is selected by default. There is an info tooltip to tell users more about the protocol. The form is the same as in UC4.	
Q4	Yes	Same as in UC4 and 5	

D.7 TC7 - Start monitoring arbitrary inbox

Users can add any other inbox for monitoring (e.g. inbox for their article where people send their comments).

Q	A	Comment	Recommendation
Q1	Yes	Menu item “Monitor inboxes” is clearly visible	
Q2	Partly Yes	Users can get confused by two different fields and two different buttons	Visually separate both forms and describe their functionality clearly
Q3	No	Added inbox immediately appears in the list below the form. However if user goes to “Incoming” page, there is no new inbox visible as expected.	Show newly added inboxes (even those without any message) on the “Incoming” page. After adding a new inbox consider redirecting user directly to the inbox’s detail page.
Q4	Yes	Added inbox appeared in the list of monitored inboxes.	

D.8 TC8 - Stop monitoring arbitrary inbox

Q	A	Comment	Recommendation
Q1	No	User would look for this option on the inbox's detail page and will expect it in the header of the inbox.	On inbox's detail add a button to allow user to stop monitoring the arbitrary inbox.
Q2	Partly yes	If user previously added any new inbox to be monitored, he/she will notice a red trash bin icon in the list of monitored inboxes.	
Q3	Yes	If user clicks the red trash bin icon, the monitored inbox disappear from the list (Monitor inboxes page)	
Q4	Yes	The action's result is immediate	

D.9 TC9 - Receive a system notification on a new message

Q	A	Comment	Recommendation
Q1	Yes	Notifications menu item is clearly visible (thanks to the icon).	Notifications menu item does not make sense for not-logged users.
Q2	Yes	Enabling system notifications is straightforward. "Test" button is a great perk to enable users to test it immediately.	I would make more clear what a system notification is. Many users will not understand it. Consider adding a small screenshot/GIF of typical system/browser notification on Windows and Mac. Test button should show the corresponding text label in the notification (now it shows New message in your inbox).
Q3	No	The "Enable system notification" becomes disabled. User might think it does not work and the functionality is disabled	Instead of making the button disabled, show button "Disable system notification" or at least some confirmation text ("System notifications enabled") instead of the original disabled button.
Q4	Partly yes	The notification logic does not seem stable. Sometimes only a toast message appears, sometimes only a browser system notification and sometimes both.	The notifications should be deterministic.

D.10 General comments from the testing

Tester added comments about the Inbox application that are not related to any specific test case:

D. COMPLETE RESULTS OF COGNITIVE WALK-THROUGH

- "Logo "inbox" in the header should redirect the user into the inboxes overview, now it logs the user out."
- "Generally there is no visual difference between clickable text elements and simple text."
- "Before releasing the solution to a general public, the author should work on inbox client's responsivity to make it usable on cell phones."
- "Regarding accessibility - app is already optimized for Deuteranomaly (user cannot differentiate between red and green)"

Attached medium content

```
(root)
├── apps.....source codes of the developed applications
│   ├── inbox.....main inbox application source code
│   └── other
│       ├── ldn-inbox.....ldn-inbox application source code
│       ├── ldn-target.....ldn-target application source code
│       ├── js-notification.....js-notification application source code
│       └── inbox-client..inbox-client application source code (first thesis
           iteration)
├── text.....diploma thesis document
│   ├── src.....document source code in LATEX
│   └── thesis.pdf.....document in PDF
└── readme.txt.....medium content description
```