



## ASSIGNMENT OF MASTER'S THESIS

<b>Title:</b>	Application supporting re-decentralization of the Web: Photo manager
<b>Student:</b>	Bc. Karel Dvořák
<b>Supervisor:</b>	RNDr. Jakub Klímek, Ph.D.
<b>Study Programme:</b>	Informatics
<b>Study Branch:</b>	Web and Software Engineering
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	Until the end of summer semester 2019/20

### Instructions

A lot of data on the current Web is stored in a centralized way, e.g. at Facebook or Google. However, the Web was originally meant to be decentralized. The current trend in Web research is therefore re-decentralization of the Web using state of the art Web technologies.

- Acquaint yourself with the Linked Data principles [1], the RDF data model [2], the recent W3C Social Web recommendations and the Solid framework [4].
- Based on the survey, evaluate technologies and protocols supporting the re-decentralization of the Web.
- Design, implement, test and evaluate a Solid based application supporting uploading and viewing images, which will include user authentication using WebID-TLS [5] and WebID-OIDC [6].
- Publish the project as open-source, hosted on a public repository.

### References

- [1] Linked Data, W3C, <https://www.w3.org/standards/semanticweb/data>
- [2] RDF, W3C, <https://www.w3.org/TR/rdf11-concepts/>
- [3] Social Web Working Group, W3C, <https://www.w3.org/wiki/Socialwg>
- [4] The Solid Project, MIT, <https://solid.mit.edu/>
- [5] WebID-TLS, W3C, <https://dvcs.w3.org/hg/WebID/raw-file/tip/spec/tls-respec.html>
- [6] WebID-OIDC, <https://github.com/solid/webid-oidc-spec>

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

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

Prague November 28, 2018





**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

Master's thesis

## **Application supporting re-decentralization of the Web: Photo manager**

*Bc. Karel Dvořák*

Department of Software Engineering  
Supervisor: RNDr. Jakub Klímek, Ph.D.

May 6, 2019



---

## **Acknowledgements**

I would like to thank my supervisor RNDr. Jakub Klímek, Ph.D. for his willingness, valuable advice, and mentoring provided throughout writing this thesis. I would also like to thank my family and friends for their support and help.



---

# 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. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 6, 2019

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2019 Karel Dvořák. 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**

Dvořák, Karel. *Application supporting re-decentralization of the Web: Photo manager*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.



---

## Abstrakt

Tato práce pojednává o návrhu a implementaci aplikace podporující re-decentralizaci Webu, která spolupracuje s technologií Solid. Na základě požadavků a výsledků analýzy je navržena aplikace zaměřena na správu fotografií. V práci je uvedena analýza, návrh, realizace, testování, a nasazení aplikace, společně se zpětnou vazbou komunity. Výstupem implementační části je webová aplikace umožňující správu fotografií na frameworku Solid.

**Klíčová slova** Solid, propojená data, decentralizace, manažer fotografií

---

## Abstract

This thesis deals with analysis and implementation of an application supporting re-decentralization of the Web, which utilizes the Solid technology. The application focused on photo management is designed based on the requirements and analysis. The thesis contains analysis, design, realization, testing, and deployment of the application, along with the feedback from the community. The output of the implementation part is the Web application supporting management of pictures on the Solid framework.

**Keywords** Solid, linked data, decentralization, photo manager



---

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 State-of-the-art</b>	<b>5</b>
1.1 Available Decentralization Solutions . . . . .	5
1.2 Solid Framework . . . . .	11
1.3 Linked Data . . . . .	16
1.4 RDF . . . . .	21
1.5 WebID . . . . .	31
1.6 Current Photo Managing Applications Built on Solid . . . . .	35
<b>2 Analysis</b>	<b>37</b>
2.1 Product Statement . . . . .	37
2.2 Business Requirements . . . . .	37
2.3 Requirements Definition . . . . .	38
2.4 Use Cases Definition . . . . .	40
2.5 Scenarios Definition . . . . .	41
2.6 Domain Model . . . . .	46
<b>3 Design</b>	<b>49</b>
3.1 User Interface Design . . . . .	49
3.2 Solid Framework Libraries Description . . . . .	64
3.3 Access Lists . . . . .	68
3.4 Class Diagram . . . . .	70
<b>4 Realization</b>	<b>73</b>
4.1 Used Technologies . . . . .	73
4.2 Application's Structure . . . . .	74
4.3 Implementation Focuses . . . . .	74
4.4 Final Version of the Pixolid Application . . . . .	91

<b>5</b>	<b>Testing</b>	<b>101</b>
5.1	Testing Scenarios . . . . .	101
5.2	Automated Testing . . . . .	106
<b>6</b>	<b>Release and Feedback</b>	<b>107</b>
6.1	Release . . . . .	107
6.2	Deployment . . . . .	107
6.3	Feedback . . . . .	108
	<b>Conclusion</b>	<b>111</b>
	<b>Bibliography</b>	<b>113</b>
<b>A</b>	<b>Acronyms</b>	<b>117</b>
<b>B</b>	<b>Developer Guide</b>	<b>121</b>
B.1	Installation . . . . .	121
B.2	Running the Development Mode . . . . .	122
B.3	Running Tests . . . . .	122
<b>C</b>	<b>Administrator Guide</b>	<b>123</b>
C.1	Installation . . . . .	123
C.2	Building the Application . . . . .	124
C.3	Deploying the Application . . . . .	124
<b>D</b>	<b>User Guide</b>	<b>127</b>
D.1	Functionality . . . . .	127
D.2	Start Using Pixolid . . . . .	127
D.3	Individual Screens' Description . . . . .	128
<b>E</b>	<b>Contents of enclosed CD</b>	<b>135</b>

---

# List of Figures

1.1	Network types . . . . .	6
1.2	Solid POD server . . . . .	15
1.3	The LOD Cloud . . . . .	17
1.4	The RDF Triple . . . . .	22
1.5	Example: Spider-man vs. Green Goblin . . . . .	24
1.6	Timeline . . . . .	36
2.1	Use Case Diagram . . . . .	42
2.2	Domain Model . . . . .	47
3.1	Task Graph . . . . .	52
3.2	WF1 Login Provider . . . . .	54
3.3	WF2 Login WebID . . . . .	54
3.4	WF3 Login Provider Redirect . . . . .	55
3.5	WF4 Application Folder Selection . . . . .	55
3.6	WF5 Friends Images Screen . . . . .	56
3.7	WF6 Tab Bar Logout . . . . .	56
3.8	WF7 Upload Image Screen . . . . .	57
3.9	WF8 Image Selection for Upload . . . . .	57
3.10	WF9 Image Selected for Upload . . . . .	58
3.11	WF10 Private Sharing Toggled . . . . .	58
3.12	WF11 Private Sharing Friends Selection . . . . .	59
3.13	WF12 Private Sharing Friends Selected . . . . .	59
3.14	WF13 Users' Images . . . . .	60
3.15	WF14 Image Detail . . . . .	60
3.16	WF15 Comment Typing . . . . .	61
3.17	WF16 Comment Adding . . . . .	61
3.18	WF17 Giving a Like . . . . .	62
3.19	WF18 User Profile . . . . .	62
3.20	WF19 Application Folder Change . . . . .	63

3.21	WF20 Application Folder Input . . . . .	63
3.22	Pixolid Logo . . . . .	65
3.23	Pixolid Black and White Logo . . . . .	65
3.24	Pixolid Favicon . . . . .	65
3.25	Class Diagram . . . . .	71
4.1	Pixolid’s directory structure . . . . .	75
4.2	Login Provider . . . . .	91
4.3	Login WebID . . . . .	92
4.4	Login Provider Redirect . . . . .	92
4.5	Application Folder Selection . . . . .	93
4.6	Friends Images Screen . . . . .	93
4.7	Tab Bar Logout . . . . .	94
4.8	Upload Image Screen . . . . .	94
4.9	Image Selection for Upload . . . . .	95
4.10	Image Selected for Upload . . . . .	95
4.11	Private Sharing Toggled . . . . .	96
4.12	Private Sharing Friends Selection . . . . .	96
4.13	Private Sharing Friends Selected . . . . .	97
4.14	Users’ Images . . . . .	97
4.15	Image Detail . . . . .	98
4.16	Comment Typing . . . . .	98
4.17	Comment Adding . . . . .	99
4.18	Giving a Like . . . . .	99
4.19	User Profile . . . . .	100
4.20	Application Folder Change . . . . .	100
C.1	Deployment Diagram . . . . .	125
D.1	Login Screen . . . . .	128
D.2	Application Folder Selection . . . . .	129
D.3	Upload Image . . . . .	130
D.4	Friend’s Images . . . . .	131
D.5	User’s Images . . . . .	131
D.6	Image Detail . . . . .	132
D.7	User Profile . . . . .	133
D.8	Application Folder Change . . . . .	133

---

## List of Tables

1.1	Scores of the decentralization solutions . . . . .	11
2.1	Use cases coverage for the requirements . . . . .	41
3.1	Wireframe coverage for use cases . . . . .	53





---

# Introduction

In the present modern world, almost everyone uses a certain kind of technology. Technology is embedded in nearly every aspect of people's lives. Most would not dare to imagine a life without any. Many industries changed rapidly in the last century, mainly in pursuit of achieving more advanced goals and tasks. Over the last few decades, there has been a massive revolution in the way that people communicate, work, interact, and get entertained. Technology has now truly become a stable part of our daily lives.

The word *technology* represents a vast variety of many different tools, instruments, knowledge, and protocols. If anyone asked people what their most used technology is, many of them would probably answer: "The Internet." According to the usage and population statistics [1], it is estimated that 55.1 percent of the world's population is using the Internet. Moreover, in Europe and North America, 85.2 percent and 95.0 percent of the residents use the Internet. People use some parts of the Internet through their mobile phones, other parts via personal computers and notebooks, or even when watching the latest television shows on demand. *Everything* is accessible via the Internet. From the recent past, the Internet has become the key to connecting all sorts of services, users, devices, and data, all together.

With the Internet mentioned above, it is crucial to specify which particular technologies make up the famous worldwide connected network. There are plenty of them. Some of which have a purpose for connecting the physical devices together, managing the network communication, translating domain names to IP addresses, running and maintaining servers, etc.

One of those many important technologies is the World Wide Web. The Web might as well be one of the most important ones. While it was primarily created to enable exchanging documents over the Internet, the Web has now become a unified and standardized way to consume and produce all sorts of data. With that in mind, the more people exchange data, each year even more than the year before, the more cluttered the world of data becomes. Moreover, other problems arise as time progresses in the way that many different services

divide the space of the Web further and further. Companies have vendor locked-in their users. The key to success in today's world of technology is to bait the user into the corporation's courtyard which is divided by a brick wall and keep him there for as long as possible.

That being said, the Web was intended to be used in quite the opposite manner. It was a solution to the never-ending creations of new data formats, to the inaccessibility of various data all over the world, to the disconnected pile of individual documents. Yet, over 30 years later, here we stand with problems such as the inability to share or access data across different social platforms, the inability to maintain a platform independent way to communicate with each other, the inability to possess what truly is ours, that being, the data.

Large companies are competing over data more than anything else. Although many people may not realize it at first, data equals money, especially if the content is being offered free of charge. Even in paid services, data have become a significant part of many companies' knowledge base and income. Those companies, quite understandably, try as much as possible to keep users just for themselves. The presence of separating users and their data is evident more than ever before. Many people have most of their private data centralized on Facebook, others have their business information stored on LinkedIn, while YouTube has become a hub for video consumption. If one decided to take his data somewhere else, he would most likely come to a realization of not having much ability to do so.

There are many efforts to keep the Web decentralized though. The inventor of the World Wide Web himself — Sir Tim Berners-Lee — has led the initiative to keep the Web re-decentralized again. His latest project is the Social Linked Data (Solid) framework which enables users to have their data saved on their chosen storage, independently of the application or service that is using the data. This method is massively different from the vendor locked-in approach. Principles for storing the data on Solid storages are based on the Resource Description Framework (RDF) model.

That being said, Solid framework is mainly a technological specification with a proof of concept implementation, rather than a complete commercial solution. Therefore, in the field of mainstream services, it cannot withstand a direct comparison with the current proprietary social networks quite yet. Users are used to specific standards of functionality which is arguably valued by masses much more than any users' privacy concerns. In addition to that, people tend to be lazy to switch or change. Therefore, it is essential to have a feature-rich and complete solution which is at least somewhat comparable with other commercial products. We are going to contribute to those efforts by implementing a photo manager because it is a common and frequent activity.

In this thesis, we are going to analyze and evaluate current technologies supporting re-decentralization of the Web. That being the Solid framework and its competitors, linked data, the RDF data model, and underlying protocols for communication and data retrieval – Hypertext Transfer Protocol

---

(HTTP) and Representation State Transfer (REST). Also, the possibilities of user authentication on the Web will be discussed, which namely includes WebID – Transport Layer Security (WebID-TLS) and WebID – Open ID Connect (WebID-OIDC).

The gained knowledge and researched technologies are going to be applied to build a standalone application for managing photographs, with the goal of keeping the data decentralized. Therefore, an analysis of the said domain will be presented. Based on the analysis, we will assess all the information and design the application itself. After that, the whole process of implementing the application will be described in detail. The whole solution is going to be tested and the testing methodology is going to be mentioned as well. Next, we will discuss the deployment of the application and the feedback from the Solid community. Finally, at the end of the thesis, we will conclude and evaluate our efforts of making the Web a little bit more accessible and decentralized place than it is now.



---

# State-of-the-art

This chapter introduces the core technologies used in the goal of achieving decentralized applications. Several different approaches to the issue of decentralization are discussed in the first section.

The Solid framework plays a central part in this effort. Its section tries to introduce further what Solid provides as a whole, what exactly can applications based on this framework achieve, and gives an overlook on the underlying principles, protocols, and routines.

Then, Linked Data and their principles are described, following with a section focused on the RDF, which Solid itself uses as a way to store data. The main RDF concepts are discussed as well as various data representations and syntaxes.

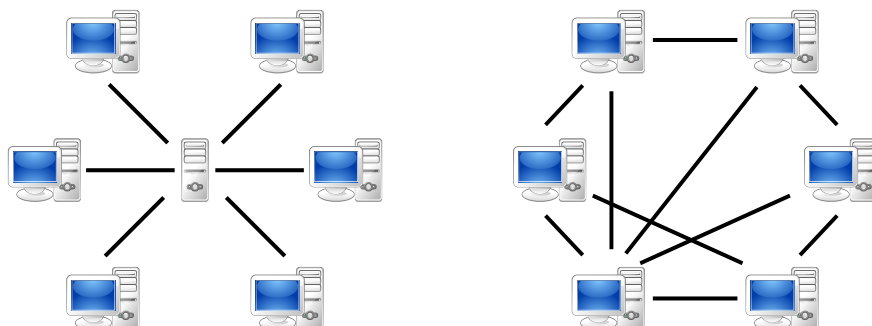
Web Identity and Discovery (WebID) is further explained as it plays a vital role in the authentication on the Web. Getting a WebID is also the first thing a user interested in Solid has to do to get started.

Last but not least follows a summary of the current photo managing applications built on the Solid platform.

## 1.1 Available Decentralization Solutions

There are various projects which are trying to achieve the goal of making the Web a more decentralized place. Many of them have been spurred by the breakthrough of Bitcoin and its Blockchain technology. However, some of them are using a Torrent based way of exchanging data between the nodes. Because of that, let us at first begin with briefly familiarizing ourselves with the BitTorrent and Bitcoin technologies.

At first, we describe said common decentralization technologies. Then, we define a set of comparison measures for concrete decentralization solutions. Then follows a comparison of the concrete decentralization solutions.



(a) A centralized (single server based) network [2]      (b) A decentralized (peer-to-peer based) network [3]

Figure 1.1: Network types

### 1.1.1 Common Decentralization Technologies

In this section, we are going to cover two decentralization technologies (BitTorrent and Bitcoin) and one anonymity providing technology (Tor), all of which have met a great success in their respective fields. A basic overview of them will be covered for further understanding of the thesis, as these technologies are also used in some of the re-decentralizations solutions.

#### 1.1.1.1 BitTorrent

BitTorrent<sup>1</sup> is a file-sharing protocol. It was designed and built as a *peer-to-peer* system. A peer is called an individual node that is sharing a file (so a concrete computer or server). Peer-to-peer means that every node which is sharing files is actively participating in the file redistribution. This redistribution is performed directly among the peers, without a need for a centralized server. The classic centralized server network is illustrated in Figure 1.1a. We can see that all of the traffic goes through the server. In Figure 1.1b, we can see that the peers communicate directly with each other, eliminating the need for the centralized server node.

Every set of shared data is defined in its *Torrent* file (ending with a `.torrent` suffix). This file is containing information about the files, size, names, and Uniform Resource Locator (URL) of a *tracker* [4]. Trackers help all of the sharing participants to discover each other. The actual file sharing is then achieved by dividing individual files (which are tied to the Torrent) into small

---

<sup>1</sup><https://www.bittorrent.com/>

chunks or bits (hence BitTorrent). Those bits are then a subject of exchange among all of the peers. Every peer can download its missing bits from others while uploading already downloaded bits to others in exchange. Peers who have completely downloaded the Torrent content are called *seeders*.

Individual Torrent files can be discovered on various forums such as the famous Pirate Bay<sup>2</sup>. The BitTorrent network is strong in its decentralized file sharing system. Being decentralized means spreading the bandwidth (download and upload) across multiple peers, without a need for a centralized server (which is often a bottleneck of the throughput). Also, it is quite problematic to control by governments as there is no centralized entity which could be addressed. Although, it is still possible to press the Torrent sharing forums and shut them down (although some disappear and others can reappear).

The decentralized design of the protocol is the reason why it is a subject of many current re-decentralization platforms.

### 1.1.1.2 Bitcoin

Bitcoin<sup>3</sup> is a purely digital currency intended for making transactions over the Internet. It is designed in a way that every transaction in the system is possible without a need for third-party authority. That means that every two users can transfer payments with each other directly in a decentralized manner [5].

Information about transactions is stored in blocks which are chained together. Every new transaction is tied to the previous transaction history. Each block creates a hash from the previous transaction and the new owner's public key. The transaction itself is signed by the previous owner. The hash creation itself can be a demanding job to do for the hardware. The creation difficulty is variable according to the frequency of making new blocks. This behavior ensures that the transaction environment can outrun any possible malicious attempts of forging transactions [5].

Bitcoins (more precisely the information about transactions) are stored on physical hard drives. There are multiple software solutions which store and exchange Bitcoin. This software is called a Bitcoin wallet. A user then installs the given application and begins to trade Bitcoin funds.

### 1.1.1.3 Tor

Tor<sup>4</sup> is a system of hiding the identity of the initiator of the network traffic. The identity is the Internet Protocol (IP)<sup>5</sup> address of the originating party.

---

<sup>2</sup><https://thepiratebay.org/>

<sup>3</sup><https://bitcoin.org/>

<sup>4</sup><https://www.torproject.org/>

<sup>5</sup><https://tools.ietf.org/html/rfc791>

When a Transmission Control Protocol (TCP)<sup>6</sup> request is made, Tor client initiates routing of this request through the randomly selected set of Tor routers (in a path of the length of 3). The traffic is all encrypted. Only the first router (which is called an entrance router) can observe the originator of the request. Also, only the last router (which is called an exit router) can know the final destination of the request [6]. The Tor network is very tough to trace and is often used as an anonymity providing service.

### 1.1.2 Comparison Measures for Decentralization Solutions

Let us introduce a set of measures for evaluating individual decentralization solutions. Each solution then will be awarded a score in the respective measure. This score will be on a scale of 1 (lowest rating) to 5 (highest rating). The ratings for solutions in respective categories are defined as follows:

- 1 - a category needs significant improvements,
- 2 - a category is dissatisfied, but could be worked on,
- 3 - a category is satisfied on a basic level,
- 4 - a category is satisfied with minor exceptions,
- 5 - a category is satisfied fully; no further improvements are needed.

Further follows a list of measures (categories), which are based on the main goals of true decentralization technologies:

- ease of use (as less additional software needed as possible),
- freedom (resistance towards censorship and content blocking),
- security (content encryption, theft proofing)
- privacy (third party avoidance, non-traceability),
- control (ability to access content, and give access rights to others),
- ownership (data should be in the hands of users, users should be able to control where the data physically is),
- major players backing (projects should have backing from the major people in the world of Web technologies as well as in the private sector).

---

<sup>6</sup><https://tools.ietf.org/html/rfc793>



### 1.1.3 Decentralization Solutions Comparison

Some of the leading decentralization solutions are described in this section. The ZeroNet, MaidSafe, Dat/HyperCore/Beaker, FreeNet, BlockStack, Diaspora, and Solid are discussed.

#### 1.1.3.1 ZeroNet

ZeroNet<sup>7</sup> is a project which is focused on decentralizing websites. Its goal is to fight against censorship. A Bitcoin wallet is used as an identity for publishing and verifying site content. BitTorrent trackers are used for exchanging the data of the websites themselves.

Each site is essentially a single Torrent which is transferred between the peers. Moreover, Tor is supported by ZeroNet, so it is possible to hide the IP address. It is necessary to have a local web-based ZeroNet application installed, in order to access ZeroNet sites. Browsing of the websites is then done by using normal Web browsers such as Mozilla Firefox<sup>8</sup>, Google Chrome<sup>9</sup>, Microsoft Edge<sup>10</sup>, and others. Each site has its Javascript Object Notation (JSON)<sup>11</sup> manifest with all the resources of which the site consists. The resources are then downloaded and further *seeded* to others like normal Torrents [7].

#### 1.1.3.2 MaidSafe

MaidSafe<sup>12</sup> is heavily focused on the privacy, security, and control of personal data. It is based on the SAFE Network<sup>13</sup>, in which users pay for uploading the content while browsing it for free. SAFE Network is a completely distributed data sharing network. It has its market based on SafeCoin cryptocurrency. The user data are divided into small parts, which are individually encrypted (using hashes from other parts), and then distributed across various peers in the network. Everyone is then essentially hosting some parts of everyone's data.

Single nodes in the network are called Vaults. The Vaults are rewarded via SafeCoin for hosting the data. In order to access the SAFE Network, a specific browser called SAFE Browser is needed. The browser is used as a gateway to the network, as well as a manager for storing passwords, data, and apps [7].

---

<sup>7</sup><https://zeronet.io/>

<sup>8</sup><https://www.mozilla.org/en-US/firefox/>

<sup>9</sup><https://www.google.com/chrome/>

<sup>10</sup><https://www.microsoft.com/en-us/windows/microsoft-edge>

<sup>11</sup><https://tools.ietf.org/html/rfc7159>

<sup>12</sup><https://maidsafe.net/>

<sup>13</sup><https://safenetwork.org/>

### 1.1.3.3 Dat/HyperCore/Beaker

Dat<sup>14</sup> is a community-driven peer to peer hypermedia protocol. Data are provided via public-key-addressed file archives (such as .zip), which can be browsed. It also tracks the history of files. HyperCore<sup>15</sup> consists of storage, content, and underlying network protocols of the Dat. Beaker<sup>16</sup> is a browser which is supporting the Dat protocol. The browser supports seeding, forking, editing, and publishing content [7].

### 1.1.3.4 FreeNet

FreeNet<sup>17</sup> provides access to the websites within its network. It is a peer to peer based platform which is focused on users in repressive regimes. On the platform, everything is anonymous. Specific software is needed for accessing the network. All of the communication is routed through peers, similarly to how Tor network works. Data are in an encrypted format and are stored across all peers. It cannot be discovered which data each node in the network stores precisely. This fact brings up many concerns about accountability for storing such unknown material. Although, the network supports limiting the circle of storing to purely known contact peers [7].

### 1.1.3.5 BlockStack

BlockStack<sup>18</sup> is promoting itself as *the easiest way to start building decentralized blockchain applications*. Presumably, it is not that far from the truth. It is a platform focused on data ownership, privacy, and security. BlockStack provides open-source developers with tools to build decentralized applications. The network is based on the Bitcoin blockchain technology. The data is accessed via a specific browser. BlockStack also provides a BlockStack ID as an identity, decentralized personal data storing, and a development platform [7].

### 1.1.3.6 Diaspora

Diaspora<sup>19</sup> is a service providing decentralized social media Web server. Its social network supports various things such as posting and sharing images. The users' data can be stored on any of the available nodes (called *Pods*), or it is possible to run your own Diaspora pod. The network is built on Ruby on Rails platform. Everyone can choose what to share with whom and where the data is stored while keeping the ownership of the author [8].

---

<sup>14</sup><https://datproject.org/>

<sup>15</sup><https://github.com/mafintosh/hypercore>

<sup>16</sup><https://beakerbrowser.com/>

<sup>17</sup><https://freenetproject.org/>

<sup>18</sup><https://blockstack.org/>

<sup>19</sup><https://diasporafoundation.org/>

measure technology	ease of use	free- dom	secu- rity	pri- vacy	con- trol	owner- ship	back- ing	SUM
ZeroNet	3	4	4	3	4	3	3	24
MaidSafe	3	4	5	4	4	3	3	26
Hypercore	3	4	4	3	3	3	3	23
Freenet	2	5	4	4	3	3	3	24
BlockStack	3	4	4	4	4	4	4	27
Diaspora	4	3	4	4	4	5	4	28
Solid	5	3	3	3	5	5	5	29

Table 1.1: Scores of the decentralization solutions

### 1.1.3.7 Solid

Solid<sup>20</sup> brings an ability to separate applications from its data. In fact, users can choose where they want to host their data. Content is stored on one of the many providers' storages called Personal Online Datastore (POD). Applications then can be developed to access the PODs' content. One can decide what each user can access and which applications have which rights. Access to the content is done just like in any other Web service via browsers or dedicated applications, without a need for any additional software. Solid is led by the inventor of the World Wide Web—Sir Berners-Lee.

### 1.1.4 Results of the Solutions' Comparison

In Table 1.1 we can see the given scores to the individual solutions. While many of the previously mentioned solutions either have the privacy, data distribution, or development tools very well perfected, the one thing that all of them except the last two fall short in is the necessity of additional software/browser for data access. To truly achieve the desired shift in the world of the Web, it is highly needed to be as less of a hassle as possible for general users. Solid has the potential to be the breakthrough project for masses, not only because it is backed by the inventor of the World Wide Web himself. In the next section, let us further introduce what the Solid framework stands for.

## 1.2 Solid Framework

This section contains information about the Solid framework. We will further introduce Solid's brief history, its main purpose, and basic terminology. Following with a basic user workflow, used technologies, and currently developed applications built on Solid.

<sup>20</sup><https://solid.inrupt.com/>

### 1.2.1 Background

Social Linked Data (Solid) framework is a project led by Sir Tim Berners-Lee, the inventor of the World Wide Web. The project has been developed by a team of people based at Massachusetts Institute of Technology (MIT). The development has been boosted in 2015, when Computer Science and Artificial Intelligence Lab (CSAIL) has received \$1 million donation from MasterCard [9]. The money has gone towards the research efforts of a Decentralized Information Group (DIG), which Sir Tim Berners-Lee is co-leading together with Lalana Kagal.

In September 2018, a company called Inrupt, Inc. was co-founded by Sir Berners-Lee and John Bruce [10]. The company's vision is to support Solid based work even further and try to spread the commercial development of Solid based applications. "Inrupt will be the infrastructure allowing Solid to flourish. Its mission is to provide commercial energy and an ecosystem to help protect the integrity and quality of the new web built on Solid," stated Sir Berners-Lee in [10]. Commercial energy is genuinely what the project needed, especially if Solid is aspiring to change the world of today's data management. As of writing this thesis, we are yet to see how exactly will Inrupt affect the workflow and development of applications using Solid.

### 1.2.2 Purpose

Main goals of the Solid framework are to give genuine data ownership back to the hands of users while improving user privacy as well. Solid as a whole consists of proposed conventions and tools for developing decentralized applications. These proposed conventions are tightly knitted with Linked Data principles which are applied to social networking applications. The base of the framework highly relies on various World Wide Web Consortium (W3C) standards and protocols. Solid's fundamental principles are as follows according to [11]:

- true data ownership,
- modular design,
- reusing existing data.

Firstly, data ownership is a problematic aspect of today's Web applications and services. Solid addresses this issue by decoupling content (user data) from the underlying applications presenting the data. Moreover, all data should be stored where a user wishes them to reside. Solid introduces a way of choosing various storage providers while supporting and encouraging users to choose their very own custom storage based solution.

Secondly, modular design denotes the possibility of people being free to switch different applications without losing their data. For example, users

can stop using one video sharing application and start using another one seamlessly. The same goes for switching the storage providers themselves. This behavior eliminates the vendor lock-in mentioned in the second part of Introduction. Companies then have to compete over offering the best possible user experience and usability for the given domain, rather than forcing users to stay imprisoned inside their restricted ecosystem.

Thirdly, reusability of existing data ensures that anyone is free to enhance current applications or even decide to develop their own. A new application then can reuse the same data, and present them in its very own way. This principle opens the doors for developers to keep on trying to make the best possible experience for users. In the end, everyone gains from this competitive environment, with nothing to miss or give up on in exchange.

### 1.2.3 Terminology

As mentioned in 1.2.2, Solid, among other things, offers a storage functionality. This functionality is provided via Solid servers. A Solid server is used to store user-specific data. A user can either use community or other providers' servers running the Solid server software [12].

On those servers, user's space or storage is called a Solid Personal Online Datastore (POD). All user's data are stored in the POD. Also, the user has the ability to choose to move the POD from one provider to another [12].

Individual applications are given access rights to the Solid POD. The user can also define what to share with the application precisely and what to allow the application to change on his POD. All of the changes are kept in the user's POD. Data elements are kept throughout every application supporting a given format that the user is using. There is no need for any synchronization between them, unlike on today's services [12].

It is literally up to the users how they decide to use their POD storage. They can store any files such as pictures, text files, or contacts. It is similar to Dropbox or Google Drive in a way, except with Solid, users decide where their data are hosted. The main benefit being that this storage is manipulatable through several different applications. Therefore, much more than just static user content can be saved – social posts, comments on them, likes, or videos. It is also possible to have more than one POD per user, which gives users the flexibility to keep their personal and professional content apart [12].

The Solid POD provides an identity as well. Users then can log into their favorite applications via PODs. In this way, users can have just one identity, not tied to any specific vendor. Similarly to what Google is doing, but without the vendor lock-in [12].

### 1.2.4 Technologies

As stated in [13], Solid is:

- A set of standards and data formats providing social application (Facebook, etc.) capabilities, such as identity, authentication and login, authorization and permission lists, contact management, messaging and notifications, feed aggregation and subscription, comments and discussions, and more.
- A Specification document describing REST Application Programming Interface (API) extending existing standards and describes design notes of individual components used.
- "A set of servers that implement this specification."
- "An ecosystem of social apps, identity providers and helper libraries (such as solid-auth-client) that run on the glsSolid platform."
- "A community providing documentation, discussion (see the solid forum), tutorials and talks/presentations."

Solid uses and extends many other accustomed standards. The framework highly depends on RDF, which is used as a data model for storage. RDF supports various serialization formats, such as Turtle, A JSON Serialization for Linked Data (JSON-LD), and Rich Structured Data Markup for Web Documents (RDFa). All of them are supported by Solid, although Turtle is the one preferred. Another standard is WebID. WebID provides universal IDs to Solid applications. Also, it is used for referring to unique Agents (people, organizations, devices). WebIDs themselves provide WebID Profile documents. Those documents are using Friend of a Friend (FOAF) vocabulary, which is also used in Access Control lists. Access lists themselves are using Basic Access Control ontology. For authentication, WebID-TLS is used, while it is also possible to use WebID-OIDC. Linked Data Platform (LDP) standard is used as a way of reading and writing Linked Data resources [13].

### 1.2.5 Basic Workflow

As previously mentioned in 1.2.4, Solid offers authentication via WebID (will be described in detail later). Therefore, a user has to obtain one. To achieve this, a user has to register with an identity provider of his choosing. This identity is usually provided together while registering with a POD provider. The identity is stored as a profile document containing the necessary data for authentication [14].

Once registered with a POD provider, the user is then able to use his POD with other applications. Applications store their data into users' PODs, which run on Solid servers. For accessing data on the server, applications communicate with the server via REST [14].

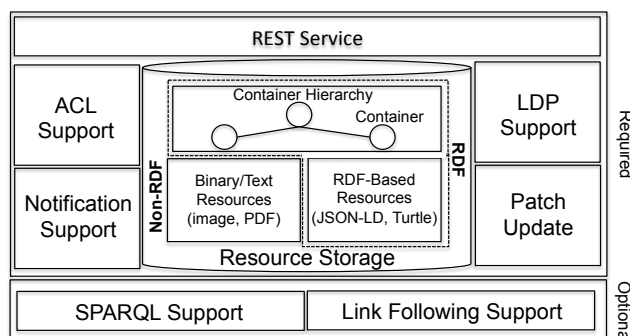


Figure 1.2: Solid POD server architecture overview [14]

Data on the server are organized in a hierarchical structure. Each node in this structure is represented as a container (or collection, directory...), described by LDP. Data can be structured or unstructured. Structured data are defined using RDF. RDF data are capable of marshaling to and unmarshaling from the Turtle format (or other formats) for storing or further processing correspondently. How RDF data are saved exactly is up to the server implementation. It is possible that the server utilizes a file system, key-value store, or a relational or graph database. Unstructured data are everything else, typically raw byte data such as images or videos. Each container and its data elements are described with an Uniform Resource Identifier (URI). Therefore, data are accessed and handled via HTTP protocol using its well-known methods such as GET, POST, PUT, PATCH, UPDATE. LDP also enables data to be accessed through link following inside retrieved data items. Servers can also support additional SPARQL Protocol and RDF Query Language (SPARQL) to retrieve RDF data. SPARQL enables applications to offload heavy data retrieval from various PODs using link following principles [14].

As the user is using various applications, access lists are needed to determine which parts of the data are available to whom and to which application. Solid servers have to implement this functionality as well. Access lists will be discussed later in detail. As applications communicate with each other, it is also key to support notifications. Solid servers support notifications according to the W3C specification stated in [15].

Figure 1.2 shows an overview of the Solid server’s supported functionality.

### 1.2.6 Solid Applications

At the time of writing this thesis, there are several applications built on the Solid framework. For example applications for managing personal data such as file browsers (Warp<sup>21</sup>, and Solside<sup>22</sup>) for managing files stored on the POD,

<sup>21</sup><https://linkeddata.github.io/warp/>

<sup>22</sup><https://jeff-zucker.github.io/solid-ide/>

a contacts manager<sup>23</sup> for organizing contacts, a profile editor<sup>24</sup> for managing users profile information, and others. As far as social media applications are concerned, among already developed applications are Cimba<sup>25</sup> (a micro blogging application), dokieli<sup>26</sup> (an article editor), timeline<sup>27</sup> (a social network), and others [16].

More applications are in development, most notably it is highly anticipated which applications or platforms are the Inrupt going to release.

### 1.3 Linked Data

In this section, a term Linked Data is introduced and further explained. Its main principles and ways of treating data are described as well. At first, basic terminology and the purpose of Linked Data are covered. Then follows a description of various identifiers which are used by Linked Data. Afterwards, Linked Data principles are discussed. Last but not least a few examples of datasets are shown.

#### 1.3.1 Terminology

The World Wide Web provides an easy way to share and access various Web documents. Those documents are primarily supposed to contain any human-readable information. On the other hand, there has to be a way to share and access any computer readable information as well. The so-called *Web of Data* contains numerous technical attributes, physical quantities, analytical properties, and countless other structured data [17].

The Web of Data needs a set of technologies that further describe a concrete way of storing, annotating, accessing, and sharing this massive stack of data. The term *Semantic Web* is a set of technologies by W3C, which supports and precisely describes all the previously mentioned activities. This set of technologies for example contains RDF, SPARQL, Web Ontology Language (OWL), and Simple Knowledge Organization System (SKOS) [17].

Those available technologies, used for querying and operating upon various data, are highly dependent on the standardized data format. Additionally, to create a Web of Data, it is crucial to store relations among data. Data which suffice all of those criteria are called the Linked Data.

---

<sup>23</sup><https://linkeddata.github.io/contacts/>

<sup>24</sup><https://linkeddata.github.io/profile-editor/>

<sup>25</sup><https://github.com/linkeddata/cimba>

<sup>26</sup><https://dokie.li/>

<sup>27</sup><https://solid-social.github.io/timeline/>



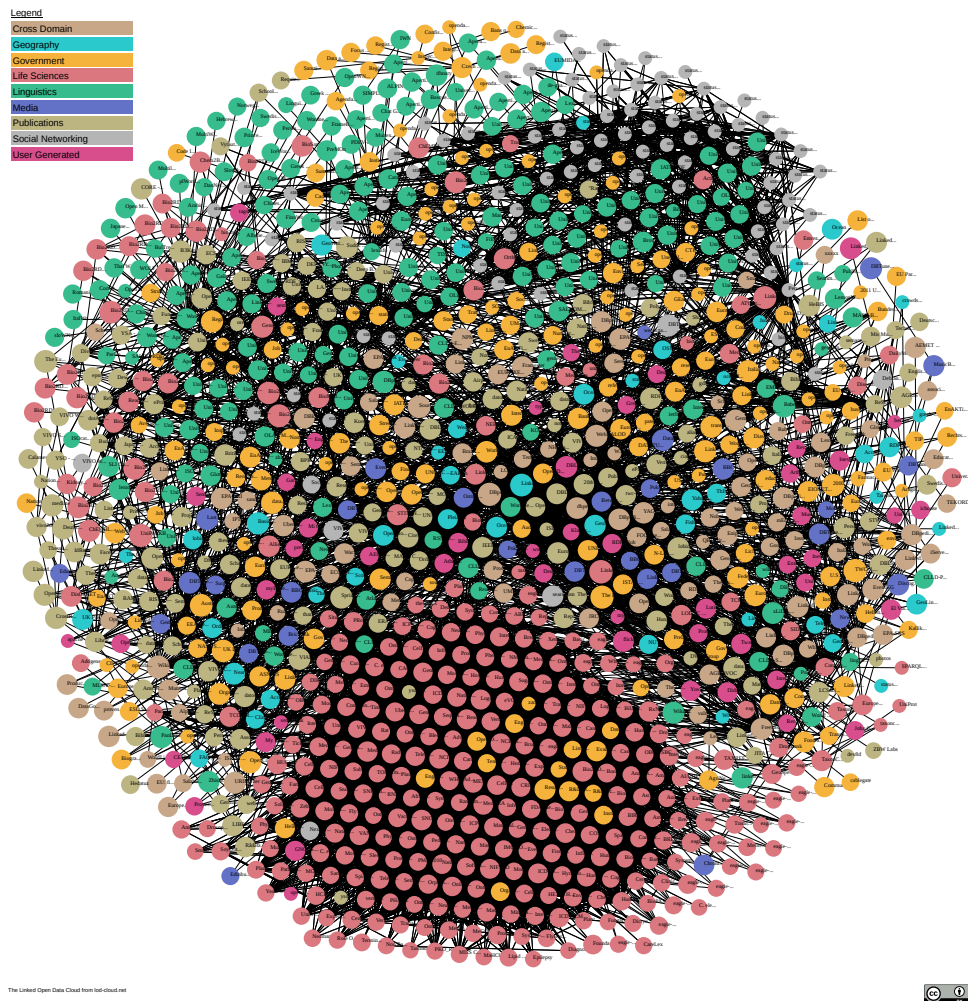


Figure 1.3: The LOD Cloud [19]

### 1.3.2 Purpose

Linked Data are an essential part of the Web of Data. They allow people and machines to browse and explore the Web of Data. As the name suggests, this behavior is made possible via links. Those links provide a bridge to finding other related data [18]. This technique is quite well known in the world of hypertext documents. Analogically, in the world of Linked Data, one can follow from a single resource to another while gaining more knowledge about the particular topic or dataset.

The real power of knowledge is hidden in the vast interconnected Web of Data. Figure 1.3 shows a simplified yet overwhelming view on the interconnected Linked Open Data (LOD) Cloud.

Each connected party contributes to this Web by making its data accessible

to others, while being well described. Providing data in a correct form and structure is not an easy task to do. It is crucial to strictly follow some of the well known and established rules to make the data appropriately linked.

### 1.3.3 IRI, URI, URN, URL

These terms are often mixed up with each other, so let us clear up their meaning all at once.

#### 1.3.3.1 URI

A Uniform Resource Identifier (URI) is a string of characters used as a universal identifier which can be assigned to any resource. It follows certain syntax rules which are tied to a proper grammar [20]. The resource identifier is limited to an American Standard Code for Information Interchange (ASCII) character set [21].

For the purpose of this thesis, we are going to further introduce a URI with an example. According to [20], a URI can be any of the following:

- `ftp://ftp.is.co.za/rfc/rfc1808.txt`,
- `http://www.ietf.org/rfc/rfc2396.txt`,
- `ldap://[2001:db8::7]/c=GB?objectClass?one`,
- `mailto:John.Doe@example.com`,
- `news:comp.infosystems.www.servers.unix`,
- `tel:+1-816-555-1212`,
- `telnet://192.0.2.16:80/`,
- `urn:oasis:names:specification:docbook:dtd:xml:4.1.2`.

Basically, URI describes a resource name, a resource locator, or both [21]. A resource name is an identifier given to the resource, while resource locator also gives information about how to access said resource.

#### 1.3.3.2 IRI

Internationalized Resource Identifier (IRI) is a generalized form of URI, which was previously mentioned in 1.3.3.1. IRIs permit use of extended Unicode characters in comparison to limited ASCII [22]. The two examples of IRIs with Czech diacritics follow next:

`https://www.hackycarky.cz/čepice.pdf`  
`https://www.háčkyčárky.cz/cepice.pdf`

The first IRI has non-ASCII characters in part after the domain. It can be transformed into URI using percent encoding. Each extended character is then rewritten according to its percent prefixed value. The following URI is equivalent to the first-mentioned IRI:

```
https://www.hackycarky.cz/%C4%8Depice.pdf
```

The second IRI has non-ASCII characters in the domain part, so Punycode encoding is used. This encoding keeps information about which extended characters are used, as well as their positions in the text. Equivalent URI for the second IRI encoded in Punycode:

```
https://www.xn--hkyrky-ptac70bc.cz/cepice.pdf
```

Both approaches are used together if the IRI has non-ASCII characters both in the domain and the part after the domain.

### 1.3.3.3 URN

Uniform Resource Name (URN) is a URI providing a resource name identifier. URN's location is arbitrary or non-existing. URN has a scheme prefix "*urn*" [21]. Example of a URN:

```
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

### 1.3.3.4 URL

URL is a URI or IRI, which also provides a location of the resource along with means of how to access the resource [21]. Example of a URL:

```
http://www.ietf.org/rfc/rfc2396.txt
```

After accessing said URL via HTTP, we should be able to retrieve the given resource, in this case, a text file.

## 1.3.4 Linked Data Principles

Sir Tim Berners-Lee stated the four rules of Linked Data in [18]. The rules are shortly described as follows:

1. "Use URIs as names for things."
2. "Use HTTP URIs so that people can look up those names."
3. "When someone looks up a URI, provide useful information, using the standards (RDF\*, SPARQL)."

4. "Include links to other URIs. so that they can discover more things."

The first rule requires using URIs as names for objects. This rule is usually quite simply satisfied by the fact that the described data are placed in the world of Semantic Web.

The second rule goes along with the first one. A HTTP URI is just a specific URI which uses HTTP as its scheme, or in other words "prefix" (the "http" part). This URI is then used as a name for a resource accessible via HTTP.

The third rule is usually covered by using already defined sets of vocabularies/ontologies. These vocabularies define specific naming conventions, attribute specifications, and relationships between many different objects or entities. There are various ontologies available to use when describing Linked Data. For the purpose of this thesis, it is important to know that when those URIs are dereferenced, vocabularies offer human-readable information about them. The same should go with URIs that are being described via those vocabularies.

The fourth rule is arguably the most important of them all. To achieve a fully interconnected Web of Data, it is crucial to have links (in form of a URL) pointing from one dataset to another. So when new information is released, it should be a standard for the publisher of the information to make initial connections to the related, already existing, data.

### 1.3.5 Examples of Datasets

After we established what Linked Data are and what are their rules, it is crucial to introduce some examples of datasets which are contributing to the Web of Data.

#### 1.3.5.1 DBpedia

One of the most well-known datasets is the DBpedia<sup>28</sup>. DBpedia is extracting structured data from several Wikimedia projects—e.g., Wikipedia. It has a vast number of entries. The English version describes 4.58 million things. Localized versions of DBpedia describe 38.3 million things altogether. Given the nature of Linked Data and its massive span of information, DBpedia is a great starting source for making a whole lot of various data queries [23].

#### 1.3.5.2 GeoNames

Another well known dataset is called GeoNames<sup>29</sup>. GeoNames contains various geographical data. There are cataloged many countries, mountains, islands, cities, including various points of interest such as football stadiums,

---

<sup>28</sup><https://wiki.dbpedia.org/>

<sup>29</sup><https://www.geonames.org/>

theatres, hotels, public transport stops, etc. It has over 25 million geographical names. Information about population, elevation, etc. is connected with those places. For example information about the Czech Republic is available at:

<https://www.geonames.org/countries/CZ/czechia.html>

On the given link we can gain knowledge about its capital city, population, area, currency, or alternative names for the country in other languages.

### 1.3.5.3 Others

There are plenty of more Linked Data sets examples. For the purpose of this thesis, we are going to list a few more examples shortly. FOAF search engine is a decentralized social network<sup>30</sup>. Many states' institutions provide so-called *open data*. For example, many Czech institutional datasets are available through its Open Data portal<sup>31</sup>. It contains data about cities, public transport, local regions, etc.

## 1.4 RDF

RDF stands for a Resource Description Framework. In the context of Linked Data, RDF is a framework used for describing such data. This section focuses on covering the said framework and its data model. Firstly, its purpose is explained. Secondly, the main RDF concepts along with abstract and concrete data representations are discussed. Lastly, a Turtle syntax is introduced in detail.

### 1.4.1 Purpose

RDF is used to represent various data which can be accessed through the Web. It is capable of describing relationships between various entities and resources. Individual relationships between entities can form a vast graph. RDF can be viewed as a kind of a graph database [22], to form a better understanding of what it is. For the scope of the entire RDF section, let us consider the following statement:

*"Alice knows Bob."*

This simple statement represents the fact that a person called Alice knows another person called Bob. With that in mind, let us further elaborate on this example.

---

<sup>30</sup><https://www.foaf-search.net/>

<sup>31</sup><https://data.gov.cz/>

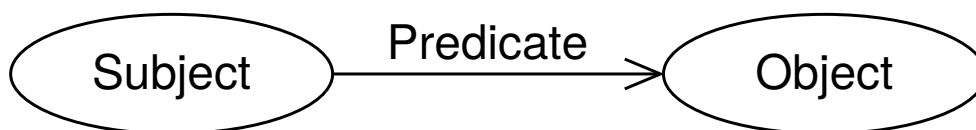


Figure 1.4: The RDF Triple [22]

### 1.4.2 Abstract Data Representation

The RDF data model is a graph consisting of multiple different relationships, such as "*Alice knows Bob.*" These relationships consist of subject-predicate-object *triples*. A *subject* is the left hand side entity of the relationship ("*Alice*"). Similarly, the right hand side entity of the relationship is called an *object* ("*Bob*"). The entire statement "*Alice knows Bob.*" is also implicitly expressing the third part of the relationship, a *predicate*. The predicate is the named relation itself ("*knows*"), which is connecting the subject ("*Alice*") with the object ("*Bob*"). A set of such triples is called an RDF graph.

It is also possible to have a fourth statement along with a triple, which provides additional information about the graph containing the said triple. This extended triple is then called a *quad*.

#### 1.4.2.1 Triple

For better visualization of the triple structure, Figure 1.4 shows an RDF graph, which can be viewed as any other graph with oriented arcs [22]. An individual node in a graph can be:

- an IRI,
- a literal,
- a blank node.

In an RDF triple, the *subject* is an IRI or a blank node, the *predicate* is an IRI, and the *object* is an IRI, a literal, or a blank node [22].

#### 1.4.2.2 IRIs

IRI is a generalized form of URI, which was previously mentioned in 1.3.3. IRIs permit use of extended Unicode characters [22]. In our example, we could have IRIs for Alice and Bob as follows:

```
http://example.org/#Alice
http://example.org/#Bob
```

Also, the relationship itself could be described as:

```
http://xmlns.com/foaf/0.1/knows
```

### 1.4.2.3 Literals

Literals are plain values like strings or numbers. A literal is represented by its *lexical form* which is a Unicode string. For example *"a cat"*, or *"1"*. It is also needed to provide a *datatype IRI*, which is an IRI identifying a datatype tied to a literal value. Although some syntaxes may support omitting the datatype [22].

### 1.4.2.4 Blank Nodes

RDF does not precisely specify blank nodes. Their structure is up to concrete implementations of RDF. Blank nodes are used for identifying a resource, which is neither identified by an IRI nor represented as a literal.

The critical information here is a knowledge of the existence of some entity tied to the adequately identified subject or object. A blank node can, for example, be representing a *"dog"*, which a particular person has. However, we do not know any more information about it, its name, breed, let alone an IRI [22].

### 1.4.2.5 Datasets

A collection of RDF graphs is called an *RDF dataset*. This dataset contains exactly one *default graph*, which may be empty and is not named. The dataset also contains zero or more *named graphs*, which consist of an IRI or a blank node and the graphs themselves. Concrete examples of datasets can be one of the examples previously mentioned in 1.3.5 [22].

## 1.4.3 Concrete Data Representations

The abstract RDF data representation, as described in 1.4.2, has many different serialization variants. All of the following concrete data representations reflect the abstract principles and offer unique serialization types. Primary individual syntaxes are named as follows:

- RDF/XML,
- N-Triples,
- Turtle,
- TriG,
- N3,
- JSON-LD,
- RDFa.

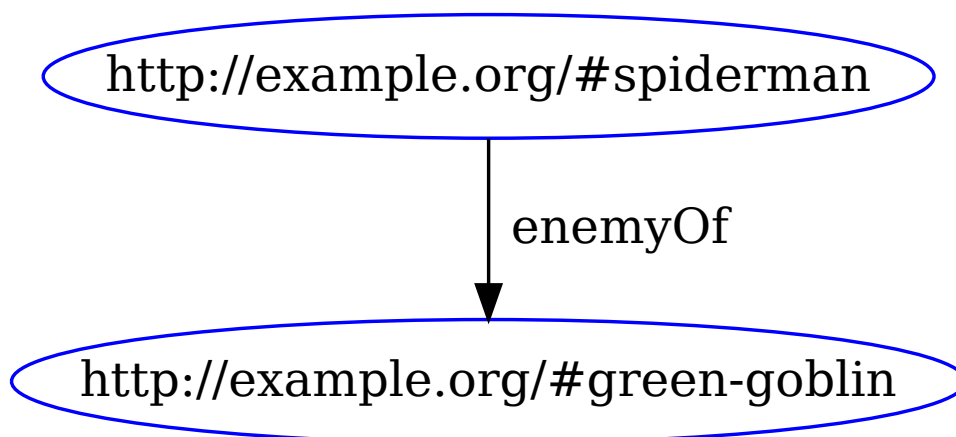


Figure 1.5: Example: Relationship between Spider-man and Green Goblin

For the purpose of this section, let us introduce the following statement:

*"Spider-man is an enemy of Green Goblin."*

For better visualization of this statement, it is illustrated in Figure 1.5. We can see that the *"Spider-man"* is a *subject*. The *"Green Goblin"* is an object. The relation between the two (*"enemyOf"*) is a predicate.

We are now going to discover how this statement is serialized in every syntax mentioned above. Some of the syntaxes share the same base syntax. Mainly they are based on Extensible Markup Language (XML)<sup>32</sup>, Notation3 (N3)<sup>33</sup>, and JSON.

#### 1.4.3.1 RDF/XML

RDF/XML is the first and still said to be a one of the most common RDF serializations [24]. It incorporates the RDF principles straight into the XML format. The main disadvantage may be the fact, that it is not very human readable, and lengthy. The example sentence can be formulated as such [25]:

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  ↔ xmlns:rel="http://www.perceive.net/schemas/relationship/">
3   <rdf:Description rdf:about="http://example.org/#spiderman">
4     <rel:enemyOf rdf:resource="http://example.org/#green-goblin"/>
5   </rdf:Description>
6 </rdf:RDF>
```

---

<sup>32</sup><https://tools.ietf.org/html/rfc4825>

<sup>33</sup><https://www.w3.org/TeamSubmission/n3/>



### 1.4.3.2 N-Triples

N-Triples is probably the most straightforward form of describing triples. It only supports sets of fully stated RDF triples. It is effortless to use, it is also fast to interpret, but the individual statements can quickly get very long. The example sentence can be formulated as such [25]:

---

```

1   <http://example.org/#spiderman>
    ↪ <http://www.perceive.net/schemas/relationship/enemyOf>
    ↪ <http://example.org/#green-goblin> .

```

---

### 1.4.3.3 Turtle

Terse RDF Triple Language (Turtle) is a popular RDF serialization. It is very human readable. Turtle is a superset of N-Triples. In addition to N-Triples, it introduces various shortcuts and syntactic sugars which can rapidly shorten the number of typed statements. The example sentence can be formulated as such [25]:

---

```

1   @prefix rel: <http://www.perceive.net/schemas/relationship/> .
2
3   <http://example.org/#spiderman> rel:enemyOf
    ↪ <http://example.org/#green-goblin> .

```

---

### 1.4.3.4 TriG

Historically, TriG is an extension of Turtle. It brings support for multiple named graphs within the document. Although, RDF 1.1 makes every Turtle document also a TriG document [25]. Our example would be same as in Turtle (see 1.4.3.3).

### 1.4.3.5 N3

Notation3 (N3) is a superset of Turtle. Its power lies in further widening the syntax capabilities. For example it offers shortcuts for implications (" $=>$ "), statement lists (" $\{...\}$ "), and other enhancements [25]. N3 can also express non-RDF statements. For our example though, the statement is the same as in Turtle (see 1.4.3.3).

### 1.4.3.6 JSON-LD

A JSON Serialization for Linked Data (JSON-LD) is a serialization based on the JSON format. Its use is pretty straightforward. There are various

keywords for linking RDF vocabularies and IRIs within the JSON content. That being said, the number of brackets can get huge quickly. The example sentence can be formulated as such [25]:

```
1 {
2   "@context": {
3     "enemyOf": "http://www.perceive.net/schemas/relationship/enemyOf"
4   },
5   "@graph": [
6     {
7       "@id": "http://example.org/#green-goblin"
8     },
9     {
10      "@id": "http://example.org/#spiderman",
11      "enemyOf": "http://example.org/#green-goblin"
12    }
13  ]
14 }
```

#### 1.4.3.7 RDFa

Rich Structured Data Markup for Web Documents (RDFa) is an RDF serialization which supports embedding RDF statements straight into Hypertext Markup Language (HTML) elements. This behavior enables making classic websites more computer readable. Every HTML element can be assigned with special attributes. Those attributes further specify the purpose of a given element (for example a name of the author of the article) [25]. The example sentence can be formulated as such:

```
1 <div prefix="foaf: http://xmlns.com/foaf/0.1/ rel:
  ↳ http://www.perceive.net/schemas/relationship/">
2 <ul>
3 <li typeof="foaf:Person">
4 <a property="foaf:homepage"
  ↳ href="http://example.org/#green-goblin"><span
  ↳ property="foaf:name">Green Goblin</span></a>
5 </li>
6 <li typeof="foaf:Person">
7 <a property="foaf:homepage" href="http://example.org/#spiderman"><span
  ↳ property="foaf:name">Spider-man</span></a>
8 <a property="rel:enemyOf"
  ↳ href="http://example.org/#green-goblin"><span>Enemy</span></a>
9 </li>
```

---

```

10 </ul>
11 </div>

```

---

### 1.4.3.8 Syntax Conclusion

As we discovered, there are plenty of RDF serializations. All of them have their advantages, and everyone can pick one of their choosing to use. For the purpose of this thesis, because of its ease of readability, we are going to cover the Turtle syntax further to illustrate the capabilities of RDF.

## 1.4.4 Turtle, a Concrete Data Representation

As mentioned above, Terse RDF Triple Language (Turtle) is a concrete language representing RDF graphs. It describes a textual syntax for such graphs. Let us further explore its syntax.

### 1.4.4.1 Main Structure

The simplest form of a Turtle statement is the following one [26].

---

```

1 <http://example.org/#spiderman>
2 <http://www.perceive.net/schemas/relationship/enemyOf>
3 <http://example.org/#green-goblin> .

```

---

In the example above we can see that IRIs are enclosed by a pair of pointed brackets ("`<>`"). A subject is declared on the first line. On the second line, a predicate is shown. An object is on the third line. Any triple like this one can be separated by whitespace in between and is terminated by a closing dot ("`.`"). In summary, the above example says that "*Spider-man*" is an enemy of "*Green Goblin*" [26].

### 1.4.4.2 @base and @prefix

The basic structure mentioned in 1.4.4.1 can be further shortened as follows.

---

```

1 @base <http://example.org/> .
2 @prefix rel: <http://www.perceive.net/schemas/relationship/> .
3
4 <#spiderman> rel:enemyOf <#green-goblin> .

```

---

With *@base* we can specify a base IRI which is used for resolving relative IRIs. The relative IRI "`<#spiderman>`" is then resolved to:

`http://example.org/#spiderman`

With *@prefix* we can create a shortcut for a frequent IRI. Prefix declarations consist of a *prefix label* and an IRI which is described by the given label. A *prefixed name* ("*rel:enemyOf*") consists of a prefix label ("*rel*") and a local part ("*enemyOf*"), separated by a ":". A complete IRI is constructed by concatenating the IRI defined by the prefix and the local part. That being said, the above example is equivalent to the one mentioned in 1.4.4.1 [26].

#### 1.4.4.3 Predicate Lists

For examples in this subsection, let us consider the following setting.

---

```
1 @base <http://example.org/> .
2 @prefix rel: <http://www.perceive.net/schemas/relationship/> .
3 @prefix foaf: <http://xmlns.com/foaf/0.1/>
```

---

If we want to express multiple properties of a subject, we can list them repeatedly as in the following example [26].

---

```
1 <#spiderman> rel:enemyOf <#green-goblin> .
2 <#spiderman> foaf:name "Spiderman" .
```

---

It is apparent that by doing this, we repeat ourselves by stating the subject twice (or more depending on the number of triple statements for each property). However, Turtle brings an option to mitigate this problem by using *predicate lists* [26].

---

```
1 <#spiderman> rel:enemyOf <#green-goblin> ;
2     foaf:name "Spiderman" .
```

---

After the initial triple statement, instead of terminating it by a dot ("."), we can use a semicolon (";"). By doing this, we can then continue listing more predicates and objects tied to the first subject. It is also possible to repeat using the semicolon after every predicate and object, effectively making the predicate list longer. The semicolon effectively repeats the subject of a series of triples with different predicates and objects [26].

#### 1.4.4.4 Object Lists

For the examples in this subsection, let us consider the following setting.

---

```

1 @base <http://example.org/> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/>

```

---

Similarly, if we want to state multiple objects, we can do so by stating the triple multiple times [26].

---

```

1 <#spiderman> foaf:name "Spiderman" .
2 <#spiderman> foaf:name "Pavoučí muž"@cs .

```

---

Object lists enable us to use a comma (",") behind each object in order to input multiple values. By doing this, the above statement can be rewritten as follows [26].

---

```

1 <#spiderman> foaf:name "Spiderman", "Pavoučí muž"@cs .

```

---

The comma effectively repeats the subject and predicate of multiple triples with different objects [26].

#### 1.4.4.5 Literals

RDF literals are simple value objects without an IRI identifier. In Turtle, literals can be specified as follows [26].

---

```

1 @base <http://example.org/> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/>
3
4 <#spiderman> foaf:name "Spiderman", "Pavoučí muž"@cs .
5 <#green-goblin> foaf:name "Green Goblin" .

```

---

In the case above, the literals are the *"Spiderman"*, *"Pavoučí muž"* (with the *cs* tag) and the *"Green Goblin"*. The said literals are used to specify a *foaf:name* object to their respective subjects [26].

Quoted literals are represented by its *lexical form*, followed by a *language tag*, *datatype*, or neither. A lexical form is a string between a pair of quotes (" "). In the above example, it is the string specifying the name of the given subject. A language tag is a part behind an at sign ("@"). It is used to specify a language in which the literal is written. The tag can be *"@en"*, *"@cs"*, *"@fr"*, etc. A datatype value is used to determine the literal datatype. It is preceded by double caret signs ("^^"). The datatype value itself is an IRI referring to the given datatype. Its IRI may be absolute, relative, or in the form of a prefixed name. If the datatype is omitted, the following URI is used:

`http://www.w3.org/2001/XMLSchema#string`

The three following statements (each on lines 5, 6, 7) are equivalent [26].

---

```
1 @base <http://example.org/> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/>
3 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
4
5 <#spiderman> foaf:name
  ↪ "Spiderman"^^<http://www.w3.org/2001/XMLSchema#string> .
6 <#spiderman> foaf:name "Spiderman"^^xsd:string .
7 <#spiderman> foaf:name "Spiderman" .
```

---

Numbers are represented as any other literals with a specified datatype, e.g., `"-5.0"^^xsd:decimal`. Turtle also offers a shortcut syntax for integers, decimals, and doubles. So we can only state the value as `-5` without the datatype specified explicitly [26].

Also booleans can be written as plain *true* or *false* values, their datatype is then deduced as `"xsd:boolean"` [26].

---

```
1 @base <http://example.org/> .
2 @prefix ex: <http://example.org/>
3
4 <#spiderman> ex:height 185 ;
5           ex:weight 90 ;
6           ex:isSpider true .
```

---

#### 1.4.4.6 Blank Nodes

Blank nodes in Turtle are represented with a preceding underscore and a colon (`"_:"`) followed by a blank node label [26].

---

```
1 @prefix rel: <http://www.perceive.net/schemas/relationship/> .
2
3 _:spiderman rel:enemyOf _:goblin .
4 _:goblin rel:enemyOf _:spiderman .
```

---

#### 1.4.4.7 Collections

Turtle also supports the use of collections. A collection is enclosed by a set of round brackets `"( )"`. In those brackets, a list of RDF terms (even an empty one) is expected. A collection can either be a subject or an object [26].

---

```

1 @base <http://example.org/> .
2 @prefix ex: <http://example.org/>
3
4 <#superheroes> ex:hasMember (<#spiderman> <#superman> <#deadpool>) .

```

---

## 1.5 WebID

Web Identity and Discovery (WebID) is a technology used for identifying various entities by using a URI. In this section, we are going to discover what does WebID further mean and learn its connections to Solid. At first, its purpose is discussed. Then follows a description of WebID-TLS and WebID-OIDC.

### 1.5.1 Purpose

WebID defines a process of verifying entities (persons, organizations, etc.) on the Web. These entities are called *agents*. WebID uses a classic URI as an identifier. This URI has to dereference to a WebID *profile* document containing further information about the referring entity. A profile is an RDF document containing user's information (e.g., name, picture, friends, etc.) which can look like this [27]:

---

```

1 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
2
3 <> a foaf:PersonalProfileDocument ;
4   foaf:maker <#me> ;
5   foaf:primaryTopic <#me> .
6
7 <#me> a foaf:Person ;
8   foaf:name "Bob" ;
9   foaf:knows <https://example.edu/p/Alice#MSc> ;
10  foaf:img <https://bob.example.org/picture.jpg> .

```

---

WebID is also used as a way of Web authentication. There are two major ways of WebID based authentications:

- WebID-TLS,
- WebID-OIDC.

Let us further introduce them in the next two sections.

## 1.5.2 WebID-TLS

WebID – Transport Layer Security (WebID-TLS) is a protocol providing user authentication on the Web. The authentication itself uses WebID and its associated profiles along with X.509<sup>34</sup> certificates. A certificate is a document containing information about the identity of the claimant of the cryptographic public key. The protocol avoids a need for third-party certificate authority. Therefore, any Website can generate such certificates. At first, it is needed to obtain a certificate, which is linked to a corresponding WebID. A user then publishes a public key given by the certificate. The following example shows (lines 10 to 13) published public key information [28]:

---

```
1 @prefix cert: <http://www.w3.org/ns/auth/cert#> .
2 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
3 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
4
5 <> a foaf:PersonalProfileDocument ;
6   foaf:maker <#me> ;
7   foaf:primaryTopic <#me> .
8
9 <#me> a foaf:Person;
10   cert:key [ a cert:RSAPublicKey;
11             cert:modulus "00cb24ed85d64d794b..."^^xsd:hexBinary;
12             cert:exponent 65537
13             ] .
```

---

Once the certificate is publicly referenced in the user's WebID profile document, the user then saves the certificate into his Web browser.

The verification is then done by a server which prompts the user for a certificate. The user chooses the certificate from his browser and provides it to the server. The authentication flow then consists of dereferencing a WebID profile. Afterwards, a private key from the certificate is compared towards the public key from the WebID profile document [28]. If the information from the certificate checks with the public key, the user is then authenticated. All of the communication between the user and the server is done under a Transport Layer Security (TLS)<sup>35</sup> connection, which provides a secure and encrypted data transfer.

---

<sup>34</sup><https://www.ietf.org/rfc/rfc5280.txt>

<sup>35</sup><https://tools.ietf.org/html/rfc5246>



### 1.5.3 WebID-OIDC

WebID – Open ID Connect (WebID-OIDC) is another authentication protocol using WebIDs. It is based on the OpenID Connect<sup>36</sup> authentication protocol. OpenID Connect is an extension of Open Authentication (OAuth) 2.0<sup>37</sup> authorization protocol. The WebID-OIDC authentication flow is the following [29]:

1. A user initiates a request to access a resource from a server. If it is mandatory to be logged in, the user is shown a provider selection screen and prompted to choose a provider with which he wishes to log in.
2. The user picks a WebID provider by choosing from the list of providers, or by typing the provider's URI, or by typing his full WebID URI.
3. The user is redirected to the provider's login screen. The preferred method of signing in can either be typing in a username and password or choosing a WebID-TLS certificate, etc.
4. After successful authentication at the provider, the user is redirected back to the initial resource which started the authentication process. The server containing the resource is given a Identity Token (ID Token) which contains the profile information about the authenticated user.
5. The ID Token gets validated by the server containing the resource. The WebID of the user trying to access the resource is then extracted from the ID Token itself.
6. The WebID from the token is then additionally verified (to make sure that the WebID truly belongs to the user).

The additional steps are required to make WebID-OIDC possible. At first, let us cover the extraction of the WebID from the ID Token. Then, the verification of the authentication provider follows.

#### 1.5.3.1 Extraction of the WebID from the ID Token

The WebID can be extracted directly from the ID Token by checking for a property called *"webid"*. If the property is not available, it is possible to check the *"sub"* property of the token. If the property is set and contains a valid HTTP URI, it is then considered and used as a WebID. If neither of the mentioned methods return a valid WebID, it is possible retrieve the WebID from a *OpenID Connect UserInfo Request*. The UserInfo response then contains a *"website"* property, which is then used as a WebID [29].

---

<sup>36</sup>[https://openid.net/specs/openid-connect-core-1\\_0.html](https://openid.net/specs/openid-connect-core-1_0.html)

<sup>37</sup><https://tools.ietf.org/html/rfc6749>

### 1.5.3.2 Verification of the Authentication Provider

Because various malicious identity providers may falsely claim that the user successfully authenticated and is the valid owner of the WebID, it is needed to be able to verify the correct provider for a given identity. The WebID is considered as verified and no further actions are needed, if and only the WebID URI suffices either of the following conditions [29]:

- WebID has a same base URI as the identity provider.
- WebID is a subdomain of the identity provider's URI.

URIs which suffice those conditions can, for example, be specified as the following [29]:

```
Identity provider: https://example.com
WebID subdomain: https://alice.example.com/profile#me
WebID shared base: https://example.com/profile#me
```

If none of the above cases are met, while trying to verify an identity provider, it is needed to take additional steps. The approved identity provider for a given WebID has to be discovered. This can be done by obtaining the information from either [29]:

- the Link request headers,
- the WebID profile document.

The first method requires making a HTTP OPTION request towards the WebID URI. Afterwards, the *"Link:"* header is parsed and the following property is checked: `"http://openid.net/specs/connect/1.0/issuer"`. The said URI with the said relation is then considered as an authorized identity provider.

The second method requires dereferencing the WebID URI (obtaining a WebID profile document). Afterwards, the profile document is queried for a statement containing `"http://www.w3.org/ns/solid/terms#oidcIssuer"` as a predicate [29]. The following example states the identity provider on line 9.

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
2 @prefix solid: <http://www.w3.org/ns/solid/terms#> .
3
4 <> a foaf:PersonalProfileDocument ;
5 foaf:maker <#me> ;
6 foaf:primaryTopic <#me> .
7
```

```
8 <#me> a foaf:Person;  
9   solid:oidcIssuer <https://provider.com> .
```

---

### 1.5.4 WebID Summary

We covered the two authentication protocols which are used on the Web. They are both used by the Solid framework. At first, Solid supported the WebID-TLS protocol. Later on, the WebID-OIDC support was added.

As of writing this thesis, WebID-TLS is slightly being put in the background, as major Web browsers (e.g., Chrome, Firefox, etc.) dropped support<sup>38</sup> for a *"keygen"* Hypertext Markup Language 5 (HTML5) element. This element was used for creating certificates by the Websites. This deprecation means that users are less able to create certificates conveniently. They have to use local programs to create a correct certificate, rather than leaving the process to an easy-to-use Web service.

As the WebID-OIDC delegates the authentication process to the identity provider, it enables bigger flexibility on the part of possible authentication methods. So it is still possible to sign in via WebID-TLS or various other future methods (possibly a biometric technology, etc.).

## 1.6 Current Photo Managing Applications Built on Solid

In this section, the current available photo managing applications based on Solid are discussed. Then, a summary of the current situation is presented.

### 1.6.1 Photo Managing Applications Description

There is only one application based on Solid which meets the description of a *"photo managing"* application based on Solid. It is called *Timeline*. Let us further introduce it.

Timeline<sup>39</sup> is a social media Web application focused on the type of content similar to Facebook. In Figure 1.6, we can see the application itself. Users can share posts containing text or photos. It supports writing comments to the posts, liking, etc. A logged in user can see posts from other people, if the user does have any friends linked in his Solid POD, and the users use Timeline as well.

Timeline was developed by Melvin Carvalho from November 2015 to January 2016. The application currently supports logging in only via WebID-TLS.

---

<sup>38</sup><https://github.com/solid/solid/issues/134>

<sup>39</sup><https://github.com/solid-social/timeline>

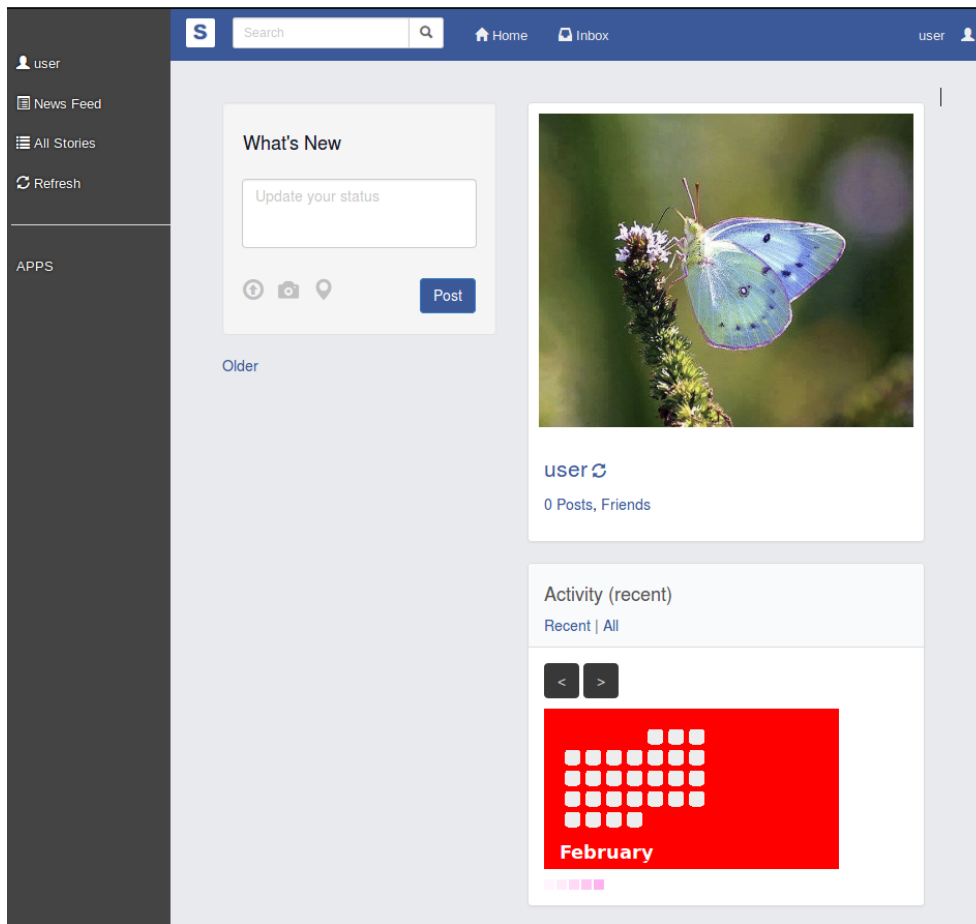


Figure 1.6: Timeline

It is required to have a *"timeline"* statement with appropriate container created in the user's POD, but this functionality seems to be non-functioning at the moment.

### 1.6.2 Photo Managing Applications Summary

As previously shown in 1.2.6, there are not many applications for Solid as a whole. Some are in the first stages of development, and for some social domains, there are no alternatives. There is a need for developers to continue and build new applications to take Solid to a broader spectrum of users. As we can see, the current situation of the photo managing application shows, that there is a void which can be filled with a new photo managing application. Moreover, that is one of the main goals of this thesis.

---

# Analysis

This chapter describes an analysis relating to a domain of the photo managing application.

Firstly, we define a product statement of said application, containing a base specification of what the application should stand for. Along with that goes the definition of the business requirements, which are partly inherited from the assignment of the thesis.

Secondly, precise definitions of the requirements for the application are mentioned. Functional and non-functional requirements are identified and further specified.

Thirdly, we extend such requirements and continue with the definitions of individual use cases.

Then, concrete scenarios for the said use cases are described in detail, along with definitions of individual actors.

Lastly, we identify and examine significant entities of the application, and further capture connections among them in a domain model.

## 2.1 Product Statement

An application which enables essential photo management will be built. This application will be named Pixolid (a wordplay on *Pictures on Solid*). It is expected to support core functionality such as uploading photos to a Solid POD, viewing uploaded photos, viewing other people's photos, basic commenting, and liking. This application will be focused on the Web desktop environment.

## 2.2 Business Requirements

This application will be built to support the Solid framework. The application needs to provide a way of authentication and interoperability with Solid PODs.

By doing this, we will contribute to the cause of re-decentralization of the Web, and support the development of Solid and related technologies.

### 2.3 Requirements Definition

In this section, functional and non-functional requirements are defined for the photo managing application.

#### 2.3.1 Functional Requirements

At first, let us define key functional requirements as follows:

- **FR1 Uploading New Images**
  - The application is expected to enable uploading new photos to a user's POD storage.
  - Priority: high.
- **FR2 Adding an Image Description**
  - The application should support adding a textual description to the image which is being uploaded.
  - Priority: medium.
- **FR3 Setting Access Rights**
  - The application should support setting access rights (public, private, selected users) to the newly uploaded images.
  - Priority: high.
- **FR4 Viewing User's Images**
  - The application needs to support viewing user's images from the POD storage.
  - Priority: high.
- **FR5 Viewing Friends' Images**
  - The application will support viewing images posted by the user's friends.
  - Priority: high.
- **FR6 Viewing an Image Detail**
  - The application enables viewing an image in a full sized detail.
  - Priority: medium.

- **FR7 Viewing Image Information**
  - The application will show the additional information about an image such as description, date of posting, author.
  - Priority: medium.
- **FR8 Posting Comments**
  - The application should support basic commenting on images.
  - Priority: high.
- **FR9 Liking Images**
  - The application may support liking user's photos.
  - Priority: medium.
- **FR10 Viewing Comments**
  - The application has to support viewing comments linked to the specific image.
  - Priority: high.
- **FR11 Viewing a Count of Likes**
  - The application may support showing the count of likes which a given image received from other users.
  - Priority: medium.
- **FR12 Choosing an Application Folder**
  - The application has to enable the user to choose where he wants to store the application data on his POD (in which folder).
  - Priority: high.
- **FR13 Viewing User's Information**
  - The application shows basic information about the user.
  - Priority: medium

### 2.3.2 Non-functional Requirements

Secondly, let us specify the main non-functional requirements below:

- **NFR1 Usage of the Solid Framework**
  - The application needs to support and utilize the Solid framework as a server storage service.

- Priority: high.

- **NFR2 Authentication of the Account**

- The application must authenticate the user’s Solid POD account via WebID-TLS and WebID-OIDC.
- Priority: high.

## 2.4 Use Cases Definition

In this section, we further specify use cases which shall cover the requirements, which were previously mentioned in 2.3. A user of the application initiates such use cases. A list of individual use cases follows next:

- **UC1 Upload a New Image**

- Enables the user to select and upload a new image to his Solid POD storage.

- **UC2 Set Access Rights**

- Enables the user to select public or private access rights to a given image.

- **UC3 View User’s Images**

- Enables the user to show his images stored on the Solid POD storage.

- **UC4 View Friends’ Images**

- Enables the user to show his friends’ posted images.

- **UC5 View Image Detail**

- Enables the user to view a detail of a single image containing further information.

- **UC6 Post a Comment**

- Enables the user to post a comment to a chosen image.

- **UC7 Like an Image**

- Enables the user to like a chosen image.

- **UC8 Choose an Application Folder**

- Enables the user to set an application folder location which is used to store data on the *POD*.



requirement	FR1	FR2	FR3	FR4	FR5	FR6	FR7	FR8	FR9	FR10	FR11	FR12	FR13	NFR2
use case														
UC1	X	X												
UC2			X											
UC3				X										
UC4					X									
UC5						X	X			X	X			
UC6								X						
UC7									X					
UC8												X		
UC9													X	
UC10														X
UC11														X

Table 2.1: Use cases coverage for the requirements

- **UC9 View User’s Information**

- Enables the user to view his brief information (name, image, etc.) and settings (application folder location).

- **UC10 Login**

- Enables the user to authenticate with his Solid POD.

- **UC11 Logout**

- Enables the user to log out of the application.

In Table 2.1, we can see the use cases covering the individual requirements which were defined in 2.3. In Figure 2.1, we can see a use case diagram showing relations among the individual use cases.

## 2.5 Scenarios Definition

In this section, we will precisely define how the use cases mentioned in 2.4 will be executed. This will be done in a form of use case scenarios defining each step of the use case.

### 2.5.1 Scenarios’ Actors Definition

At first, let us introduce the use case scenarios’ actors:

- **User**

## 2. ANALYSIS

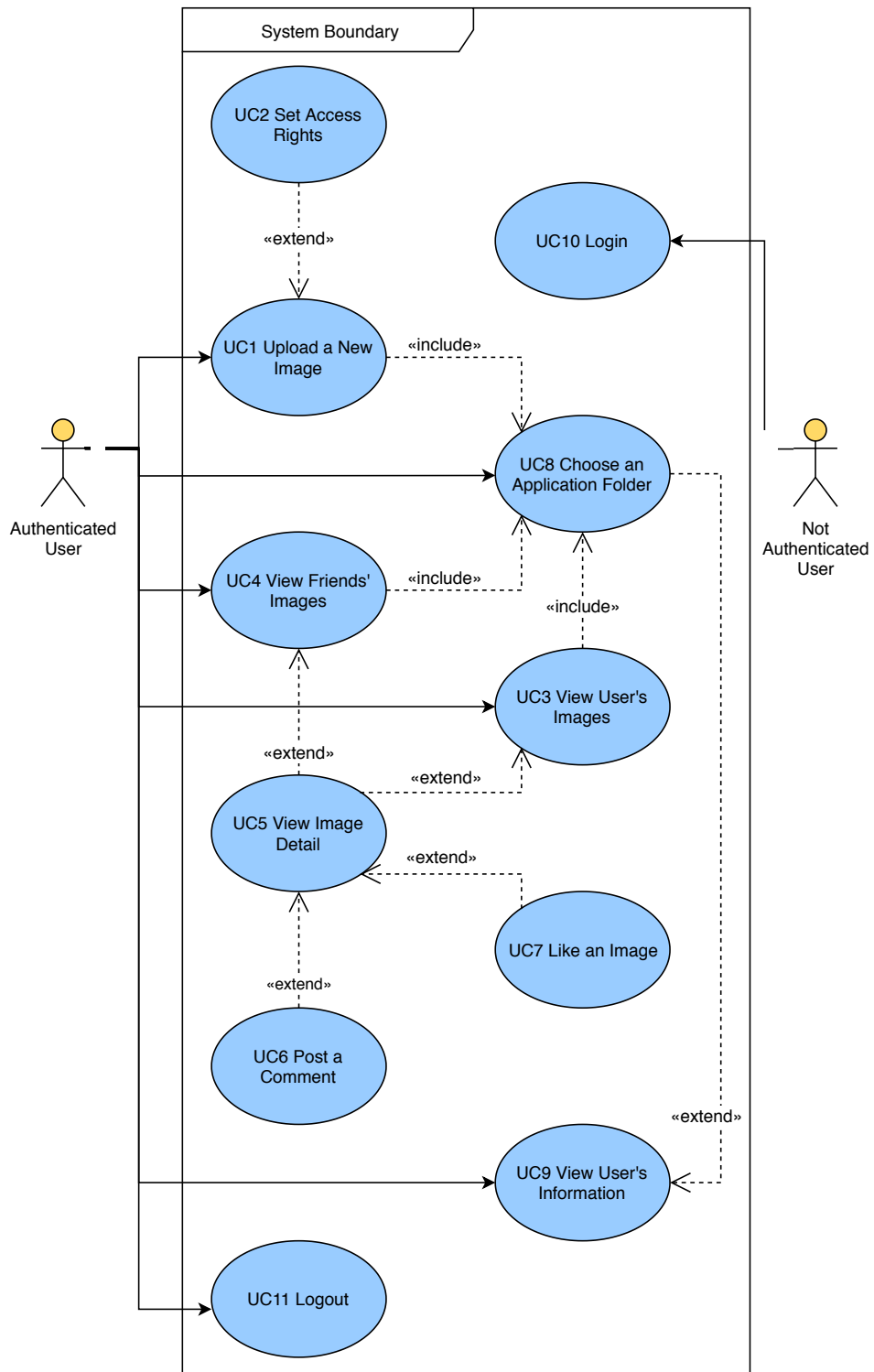


Figure 2.1: Use Case Diagram: Relations among the use cases.

- An actor which uses and controls the *system* (application) in the role of a user.
- **System**
  - An actor which substitutes the client-side application, which is executing *user's* tasks and commands. The *system* is directly communicating and exchanging data with the *server*.
- **Server**
  - An actor which is emulating the Solid POD storage server. The server handles requests made by the *system*, which are originally initiated by the *user*.

### 2.5.2 Scenarios Description

A description of the individual use cases' scenarios continues further below:

- **Scenario UC1 Upload a New Image**

**Input state:** The *user* is logged in accordingly to *UC10*.

1. The *user* clicks on an *upload image* tab button.
2. The *system* shows an *upload image* page.
3. The *user* clicks on a *select image* button.
4. The *system* shows an *image selection* dialog.
5. The *user* chooses an image he wishes to upload.
6. The *user* types an *image description* text into an *image description* text area.
7. The *user* may or may not wish to set specific access rights for the image accordingly to *UC2* from this extension point.
8. The *user* confirms the upload by clicking on an *upload image* button.
9. The *system* uploads the image to the *server*.

**Output state:** The *system* uploaded the user's selected image with its metadata to the *server*.

- **Scenario UC2 Set Access Rights**

**Input state:** The *user* used an extension point from *UC1*.

1. The *user* toggles an *access rights* toggle (checkbox).
2. According to the *access rights* toggle value, the *system* shows appropriate *access rights* setting, either a case *a)* or *b)* apply.

- a) If the *access rights* toggle is set to *public*, the *system* shows no further interaction.
- b) If the *access rights* toggle is set to *private*, the *system* shows a *user selection* input control.
  - i. The *user* chooses with whom he wishes to share the image by selecting multiple users (or none) from the *user selection* input control.
3. The *system* sets the chosen *access rights* for the image.

**Output state:** The *system* set the access rights for the image from *UC1*.

- **Scenario UC3 View User's Images**

**Input state:** The *user* is logged in accordingly to *UC10*.

1. The *user* clicks on a *my images* tab button.
2. The *system* fetches user's images from the *server*.
3. The *system* lists previews of the images on the *my images* page.

**Output state:** The *user* is shown an overview of his uploaded images.

- **Scenario UC4 View Friends' Images**

**Input state:** The *user* is logged in accordingly to *UC10*.

1. The *user* clicks on a *friends' images* tab button.
2. The *system* fetches images uploaded by his friends from multiple *servers*.
3. The *system* lists previews of the images on the *friends' images* page.

**Output state:** The *user* is shown an overview of his friends' uploaded images.

- **Scenario UC5 View Image Detail**

**Input state:** The *user* is shown an overview of his or his friend's uploaded images accordingly to *UC3* or *UC4*.

1. The *user* clicks on any *image preview* element.
2. The *system* shows a full image in a new window with additional information.

**Output state:** The *user* is shown a detail page for the selected image.

- **Scenario UC6 Post a Comment**

**Input state:** The *user* is shown an image accordingly to *UC5*.

1. The *user* types a comment into a comment text area.
2. The *user* clicks on an *add comment* button.
3. The *system* saves the comment to the *server*.

**Output state:** The *system* shows the new comment under the related image.

- **Scenario UC7 Like an Image**

**Input state:** The *user* is shown an image accordingly to *UC5*.

1. The *user* clicks on a *like* button.
2. The *system* saves the like to the *server*.

**Output state:** The *system* shows a thumb and increases the like counter under the related image.

- **Scenario UC8 Choose an Application Folder**

**Input state:** The *user* is logged in accordingly to *UC10*.

1. Folder selection initiation, either *a)* or *b)* apply.
  - a) The *user* initiates the folder change from a *profile* page by clicking on a *change folder* button.
  - b) The *system* checks whether the *application folder* is set correctly.
    - i. If the *application folder* is set and valid, the use case ends here.
2. The *system* shows a *set application folder* dialog.
3. The *user* types a desired folder into the *application folder* text area and confirms it by clicking on a *Save* button.
4. The *system* creates the *application folder* on the *server*.
5. The *system* sets the *application folder* to the *user's* profile document on the *server*.

**Output state:** The *system* created and set the *application folder* on the *user's server*.

- **Scenario UC9 View User's Information**

**Input state:** The *user* is logged in accordingly to *UC10*.

1. The *user* clicks on a *profile* tab button.

2. The *system* fetches the user's information from the *server*.
3. The *system* shows the user's information.

**Output state:** The *user* is shown the settings page for his account.

- **Scenario UC10 Login**

**Input state:** The *system* is on a main page for non-authenticated users.

1. The *system* shows a login screen.
2. Provider selection, either the case *a)* or *b)* applies.
  - a) The *user* clicks on a provider from a *provider menu*.
  - b) The *user* types a WebID URI into a WebID text area.
3. The *user* clicks on a *login* button.
4. The *system* redirects the *user* to his provider's login page.
5. The *user* logs in at the provider (may type in a username and a password, or via certificate, etc.).
6. The *user* is redirected back to the *system* to a *friend's images* page.

**Output state:** The *user* is authenticated.

- **Scenario UC11 Logout**

**Input state:** The *user* is logged in accordingly to *UC10*.

1. The *user* clicks on a *logout* button.
2. The *system* logs the user out of the application.
3. The *system* is redirected to the main page for non-authenticated users.

**Output state:** The *user* is logged out of the application.

## 2.6 Domain Model

In this section, we will describe key entities concerning the photo managing application's domain. Based on the requirements, the related use cases and the use cases' scenarios we identify following entities:

- a person,
- an image,
- a comment,
- a like.

In Figure 2.2, we can see relations among the said entities. Let us further describe them in detail.

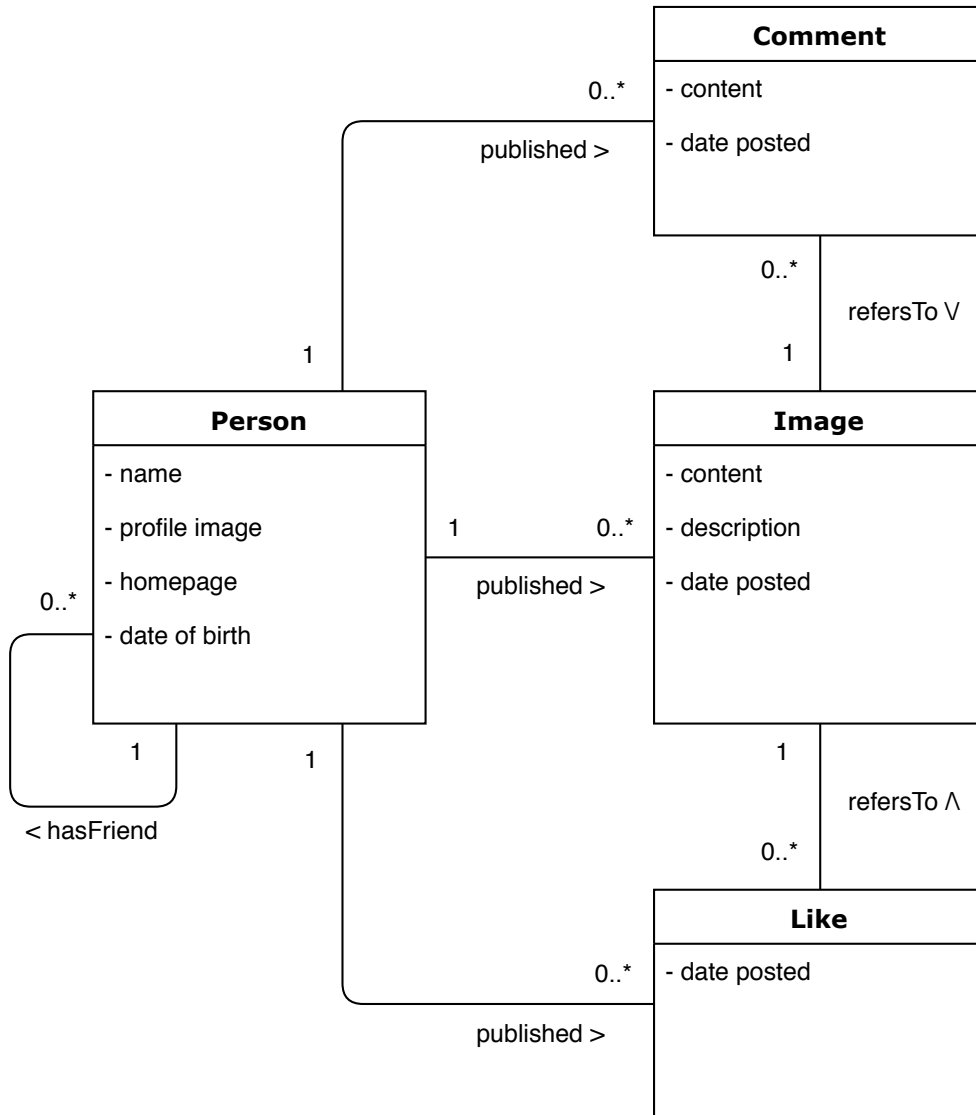


Figure 2.2: Domain Model: Relations among the entities.

### 2.6.1 Person

A *person* entity represents any user of the application. The person is the owner of his uploaded images, comments, or likes. The person is distinguished by a WebID, which can provide additional details such as name, profile image, age, homepage, etc. Persons can have other persons listed as friends.

### 2.6.2 Image

An *image* entity represents a single photo uploaded by a *user* to the application. The image entity contains the photo itself, as well as additional information such as date posted, description, etc. The image is tied to the concrete *person* which uploaded it.

### 2.6.3 Comment

A *comment* is a short text referred to the *image*. Comments are posted by *persons*. Each comment contains information about its content, date of posting, etc.

### 2.6.4 Like

A *like* entity is a form of acknowledgment of enjoyment. The like is linked to an *image*. Likes are posted by *persons*.



---

# Design

In this chapter, the user interface design is discussed, along with the task list and graph, wireframes, and branding. Then follows a description of several Solid framework libraries. Afterwards, for controlling access to individual resources, access lists are introduced and discussed. Finally, the individual application's classes are defined and commented, as well as visualized in a class diagram.

## 3.1 User Interface Design

In this section, the User Interface (UI) of the Pixolid application is discussed and described. At first, a task list along with task groups and graph are defined. Then, individual wireframes are shown and commented. Lastly, the proposed branding of the Pixolid application is introduced.

### 3.1.1 Tasks

In this section, we identify key task groups and their tasks. Individual tasks are then visualized in a task graph.

#### 3.1.1.1 Task Groups

According to the use cases, which were previously defined in 2.4, we identify the following groups for the tasks within the application:

- ■ Landing Page,
- ■ Friends' Images,
- ■ My Images,
- ■ Upload Image,

### 3. DESIGN

---

- ■ Image Detail,
- ■ User Information,
- ■ Application Folder.

The *Landing Page* is responsible for the authentication process. The *Friends' Images* page shows images uploaded by the user's friends and is shown after the authentication. The *My Images* page is responsible for showing the user's images. The *Upload Image* page provides the functionality to upload new images. The *Image Detail* shows an image in full size along with its description, comments, and likes. In *User Information*, it is possible to see the user's basic info. The *Application Folder* is responsible for setting the correct application folder by a user.

#### 3.1.1.2 Task List

For the task groups, the following tasks are defined:

- ■ Landing Page,
  - Register,
  - Login,
- ■ Friends' Images,
  - Show friends' images,
- ■ My Images,
  - Show user's images,
- ■ Upload Image,
  - Select an image,
  - Add image description
  - Set private/public sharing,
- ■ Image Detail,
  - Show an image in full size,
  - Show image description,
  - Show comments,
  - Show a number of likes,
  - Add a comment,

- Give a like to the image,
- ■ User Information,
  - Show a user name,
  - Show a user image,
  - Show an application folder,
  - Change the application folder,
  - Show welcome information,
  - Logout,
- ■ Application Folder,
  - Show an application folder selection,
  - Select an application folder.

### 3.1.1.3 Task Graph

In Figure 3.1, we can see relations among the said tasks within the groups.

### 3. DESIGN

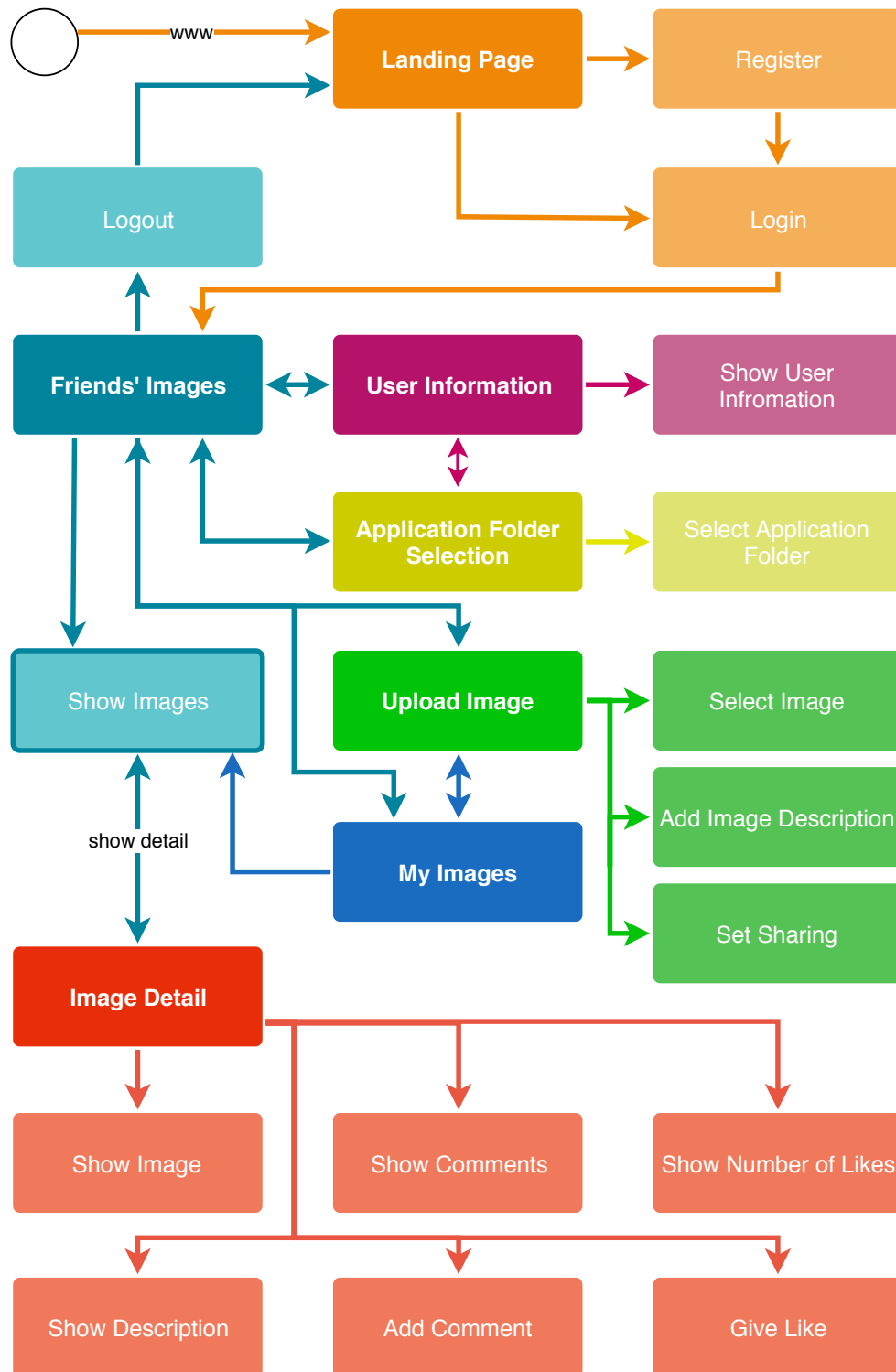


Figure 3.1: Task Graph: Relations among the tasks within groups.

### 3.1.2 Wireframes

In this section, we can see individual wireframes which were created based on the use cases (see 2.4) and the task list (see 3.1.1). In Table 3.1 we can see coverage of the use cases by the individual wireframes.

use case ——— wireframe	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11
WF1 (fig. 3.2)										X	
WF2 (fig. 3.3)										X	
WF3 (fig. 3.4)										X	
WF4 (fig. 3.5)								X			
WF5 (fig. 3.6)				X							
WF6 (fig. 3.7)											X
WF7 (fig. 3.8)	X										
WF8 (fig. 3.9)	X										
WF9 (fig. 3.10)	X										
WF10 (fig. 3.11)		X									
WF11 (fig. 3.12)		X									
WF12 (fig. 3.13)		X									
WF13 (fig. 3.14)			X								
WF14 (fig. 3.15)					X						
WF15 (fig. 3.16)						X					
WF16 (fig. 3.17)						X					
WF17 (fig. 3.18)							X				
WF18 (fig. 3.19)								X	X		X
WF19 (fig. 3.20)								X			
WF20 (fig. 3.21)								X			

Table 3.1: Wireframe coverage for use cases

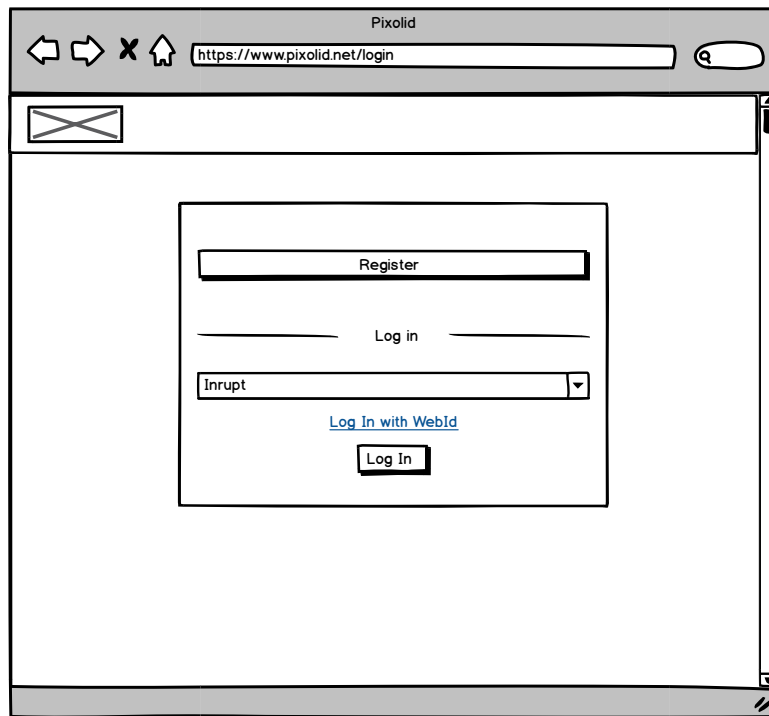


Figure 3.2: Login screen with a provider.

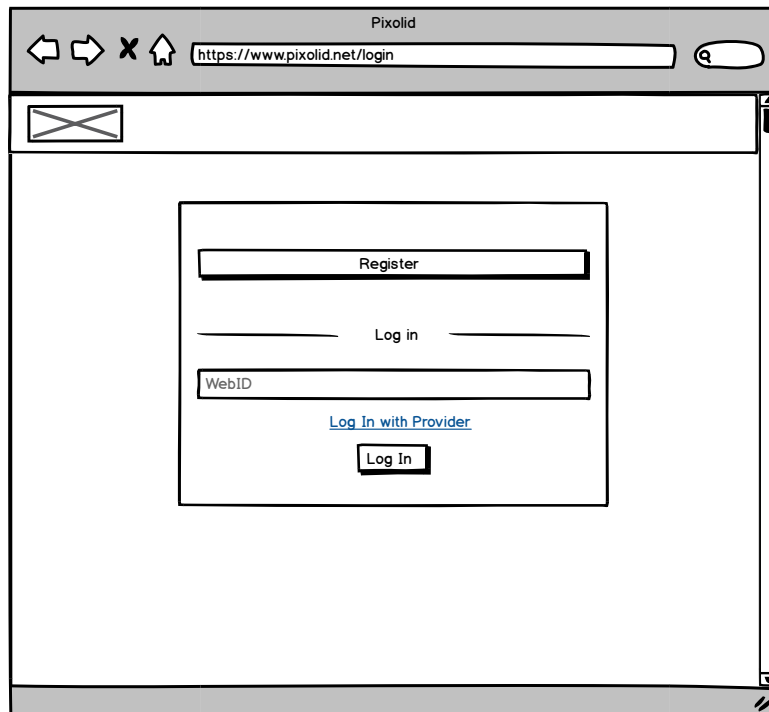


Figure 3.3: Login screen with a WebID.

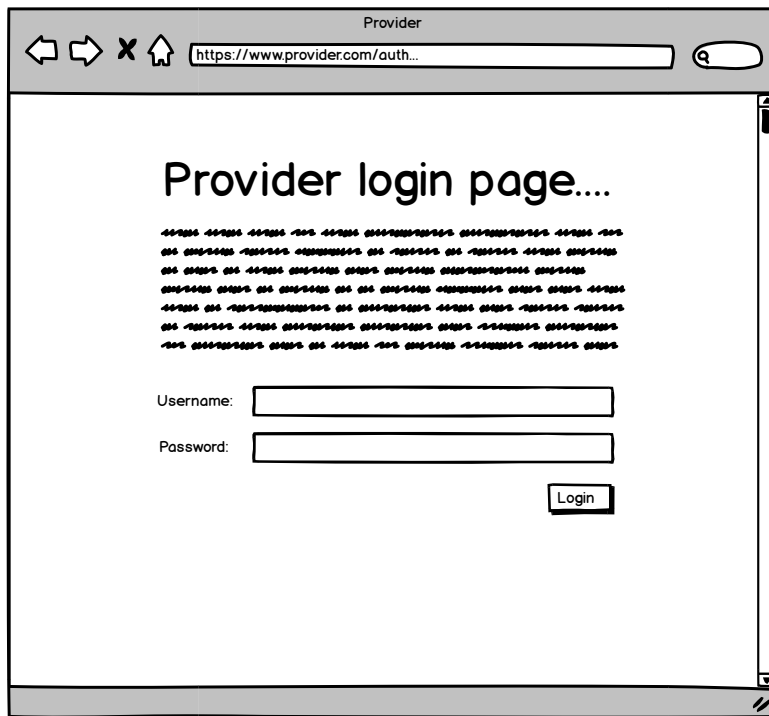


Figure 3.4: Login screen at the provider.

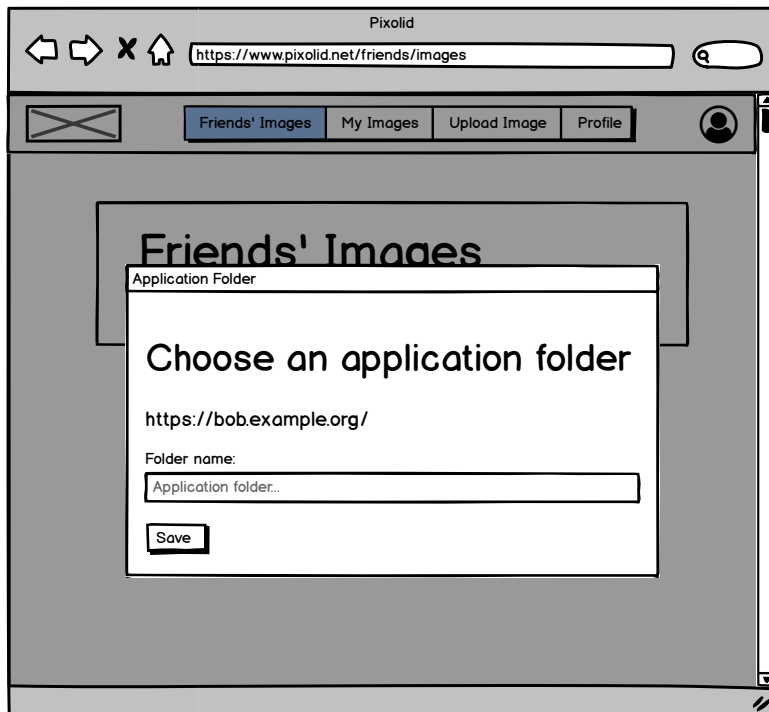


Figure 3.5: The user is prompted to choose an application folder.

### 3. DESIGN

---

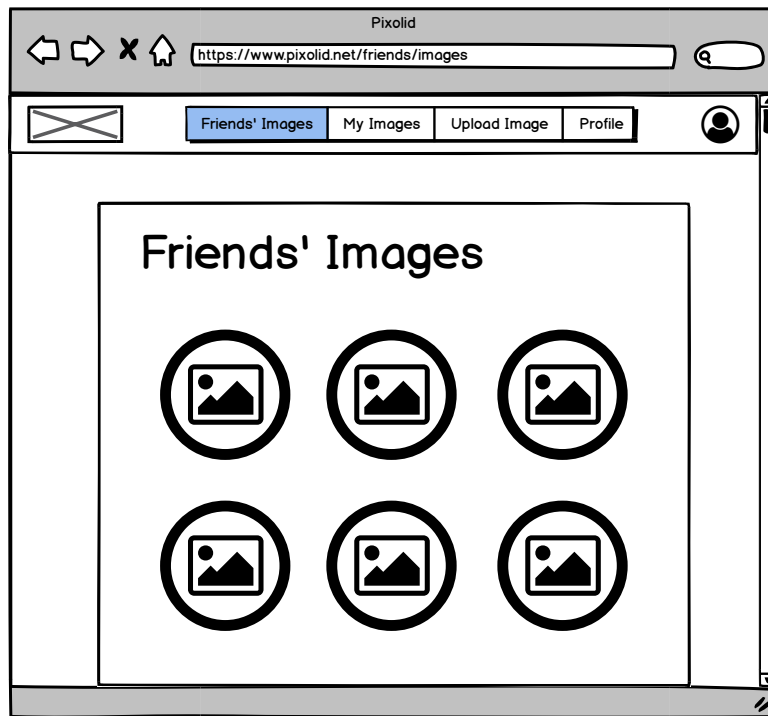


Figure 3.6: An overview of images posted by the user's friends after login.

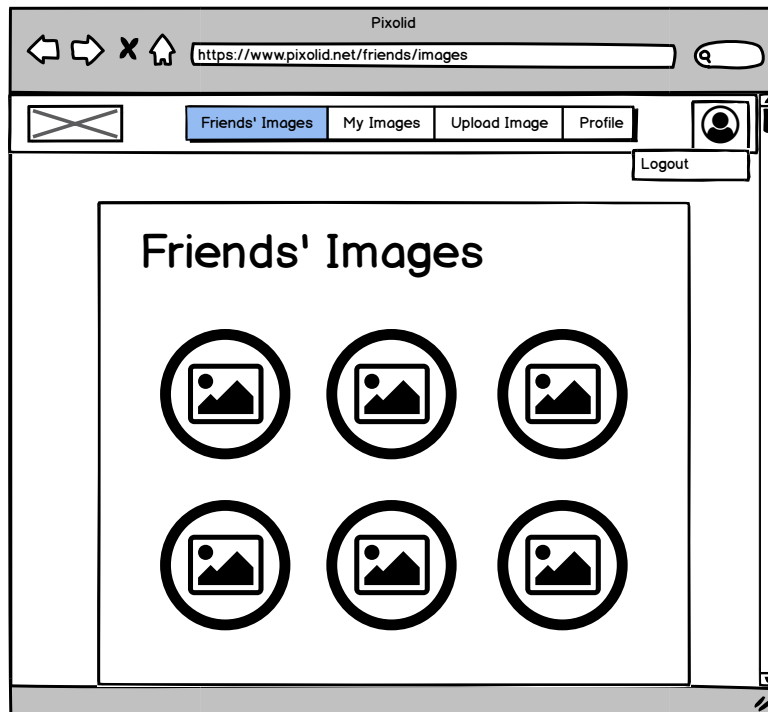


Figure 3.7: Tab bar showing a logout option.



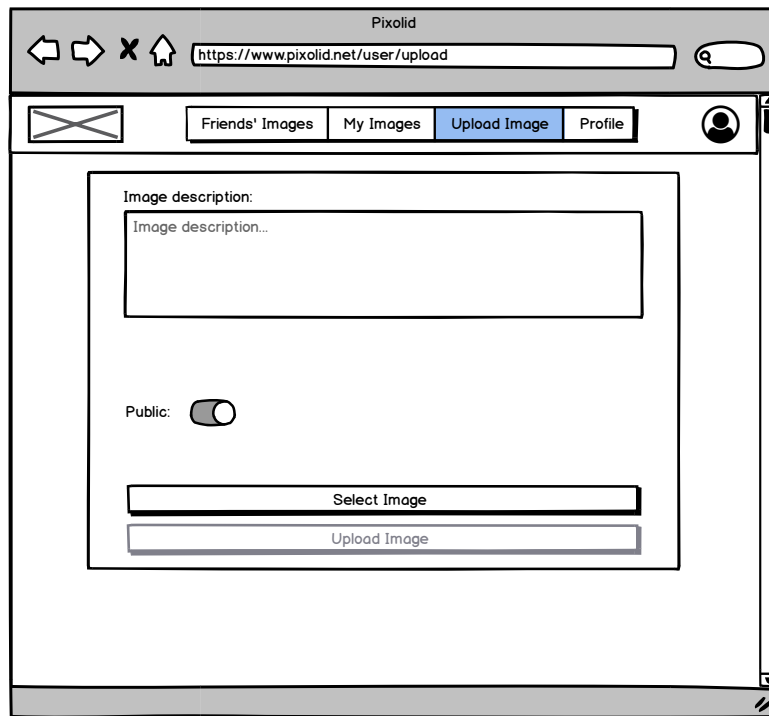


Figure 3.8: Upload image screen providing upload functionality.

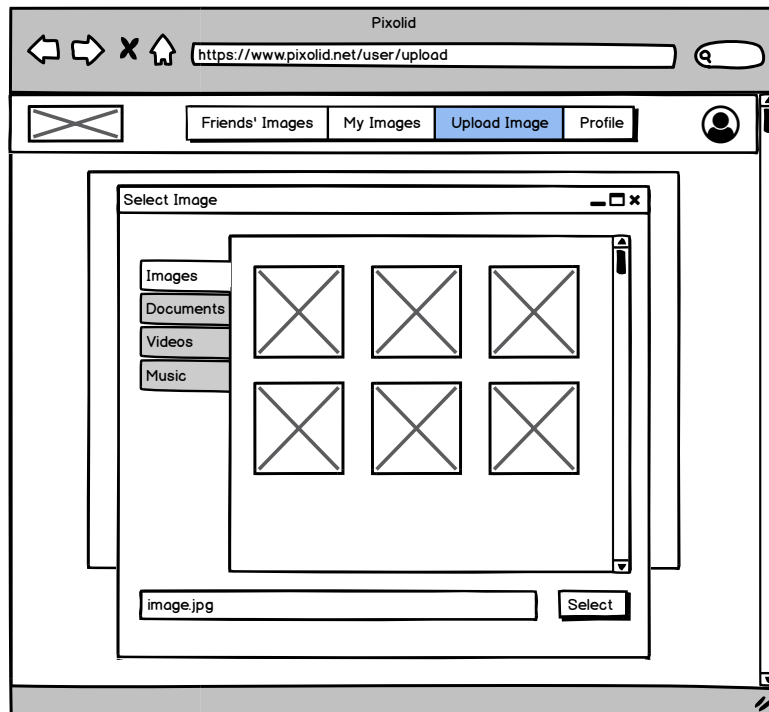


Figure 3.9: Dialog shown during the image selection for upload.

### 3. DESIGN

---

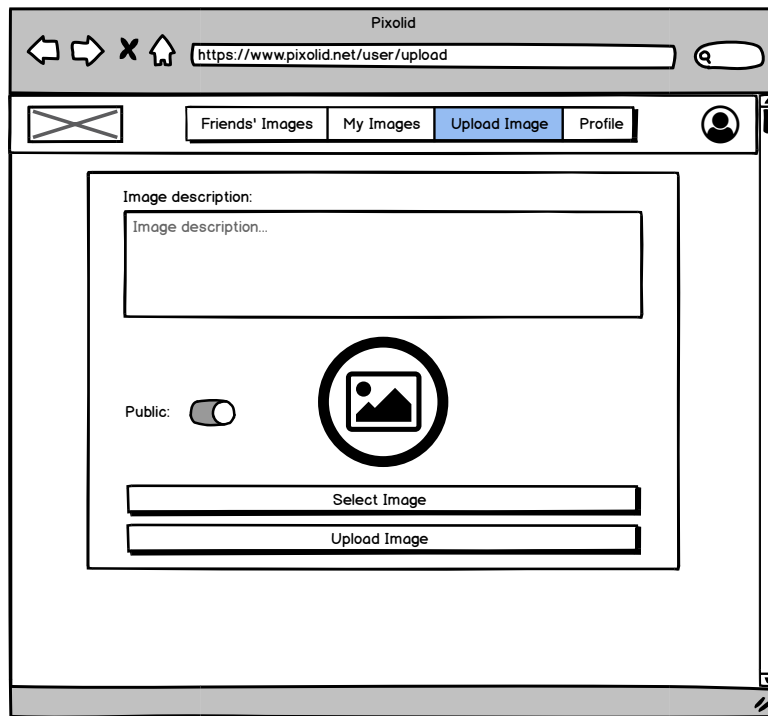


Figure 3.10: An image preview is shown after the selection of the image.

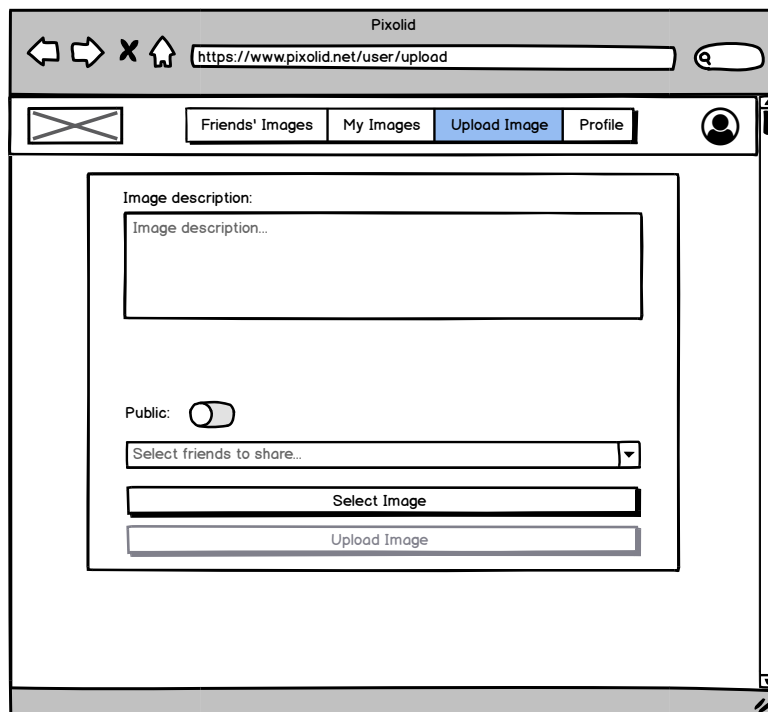


Figure 3.11: If the public sharing is off, the user can choose individual friends.

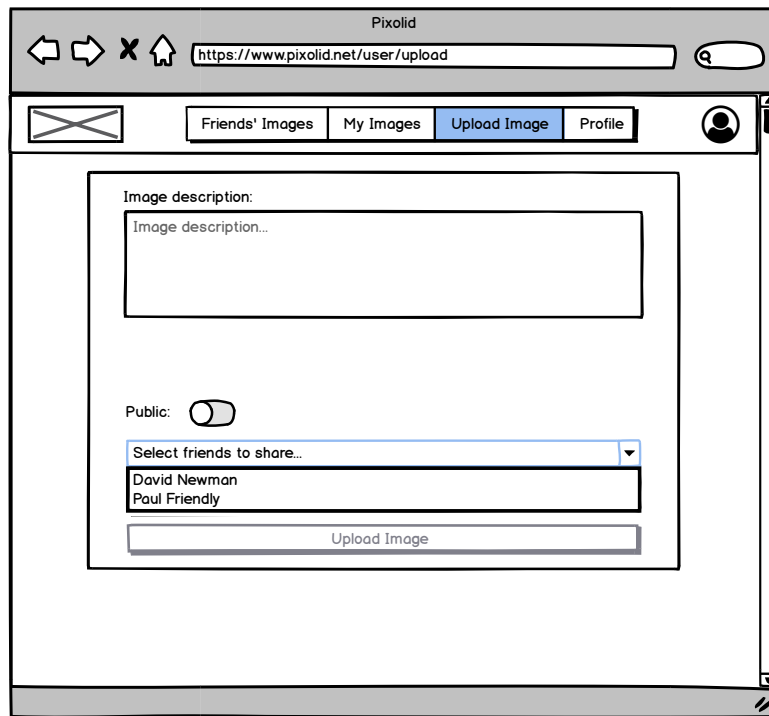


Figure 3.12: The user picks multiple friends (or none) to share the image with.

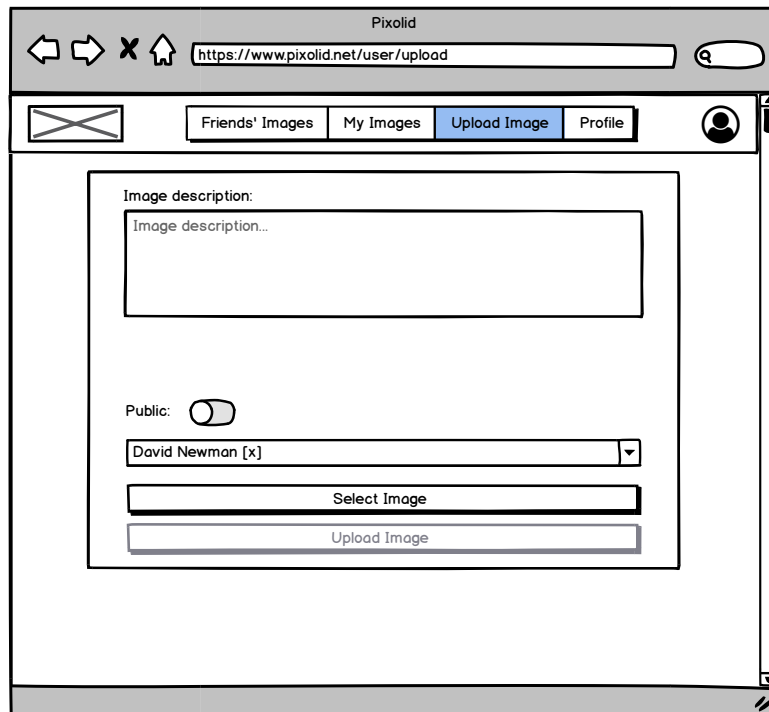


Figure 3.13: After the selection, the selected users are shown for sharing.

### 3. DESIGN

---

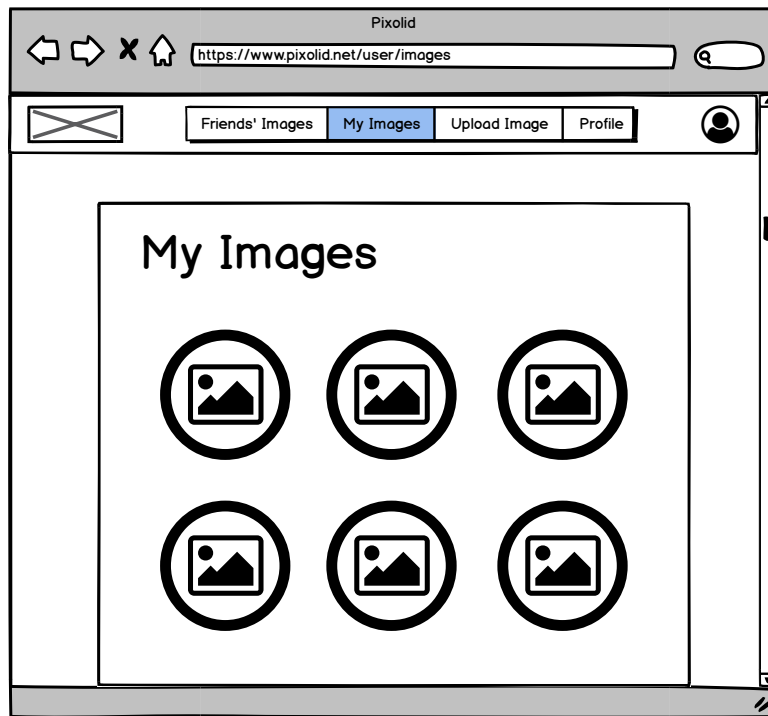


Figure 3.14: An overview of images posted by the user.

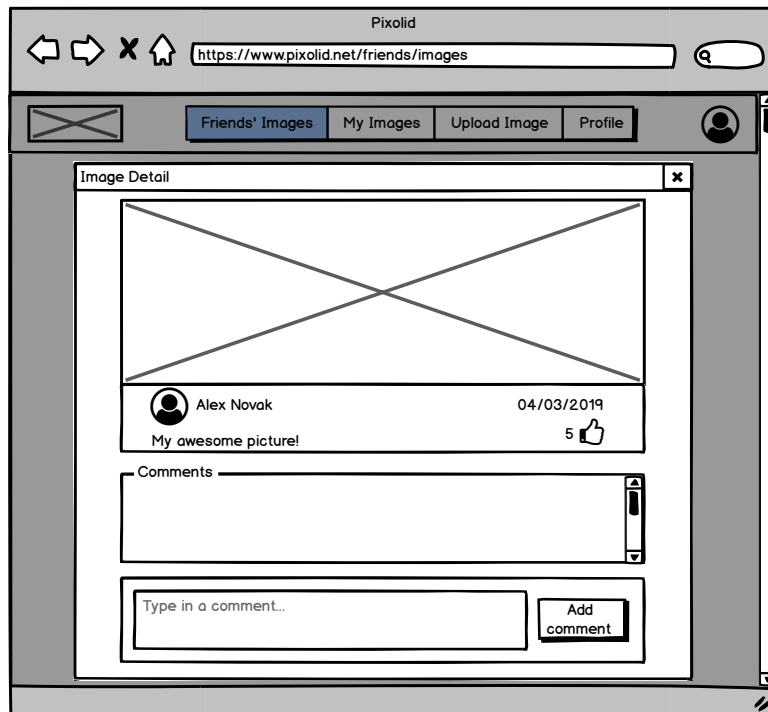


Figure 3.15: After a click on any image, an image detail is shown.

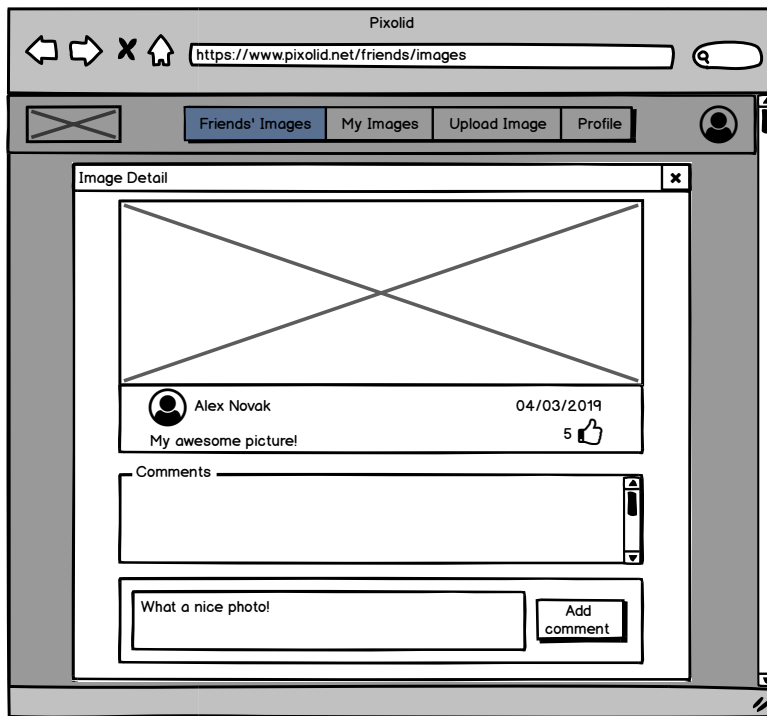


Figure 3.16: The user can type in a comment related to the picture.

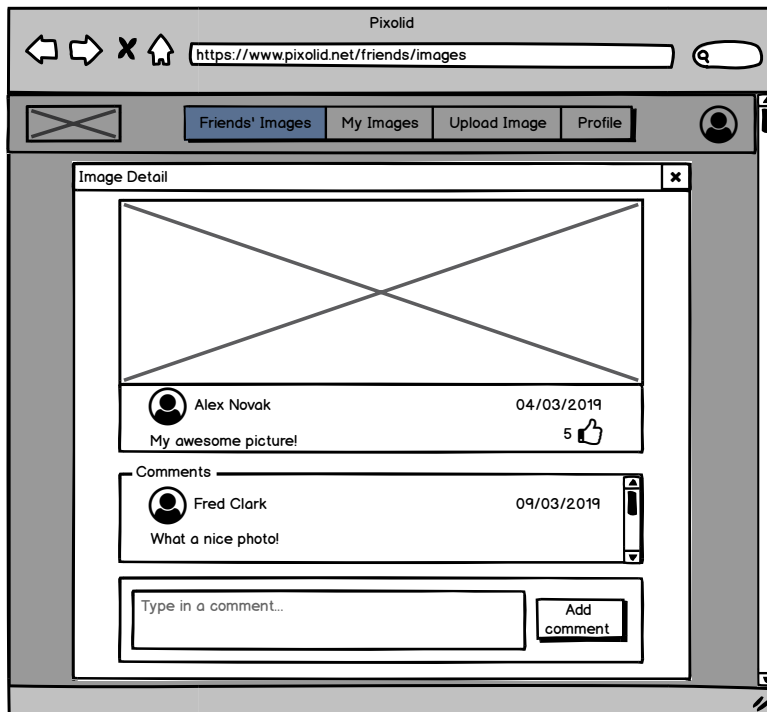


Figure 3.17: The comment is added by clicking on the *Add comment* button.

### 3. DESIGN

---

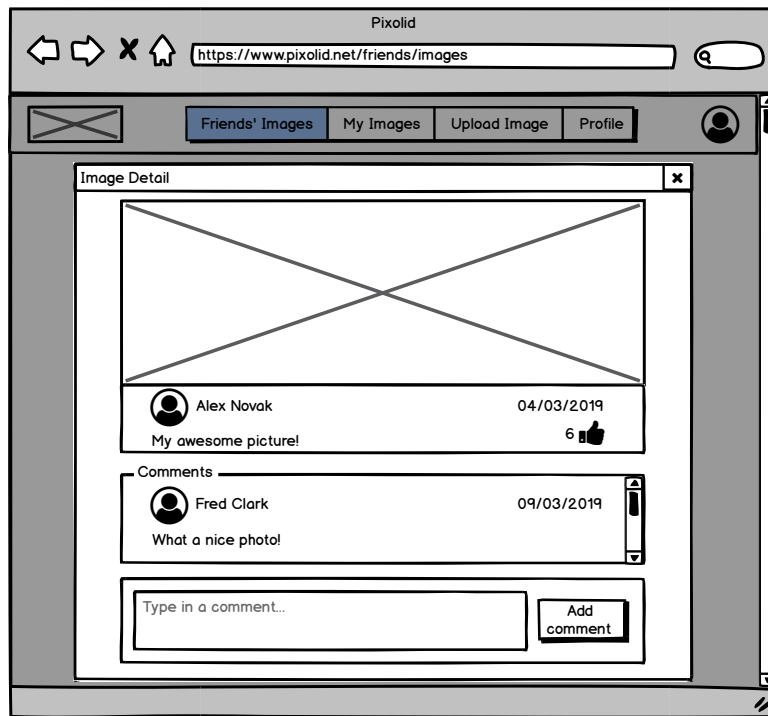


Figure 3.18: The like has been added by clicking on the like button.

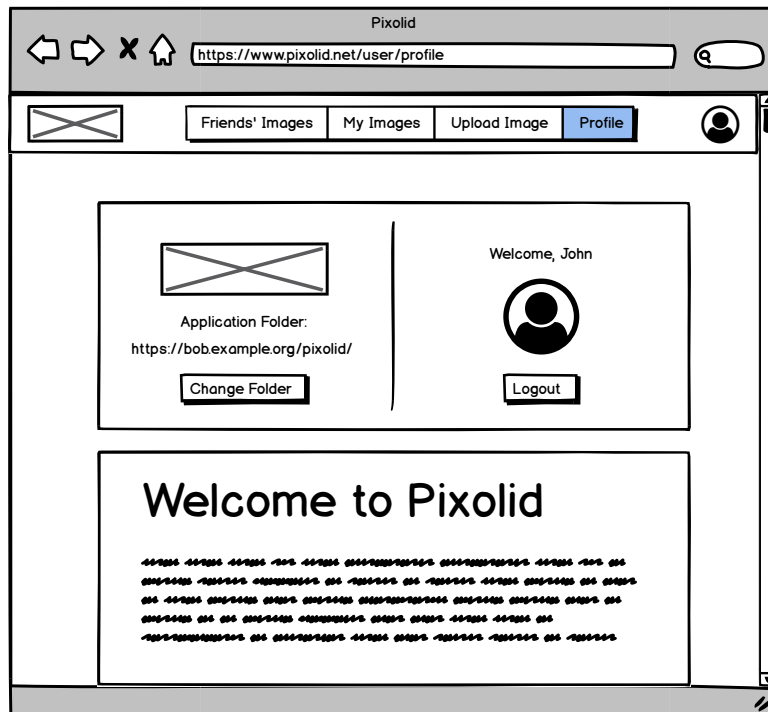


Figure 3.19: The profile shows info about the user, and the logout button.

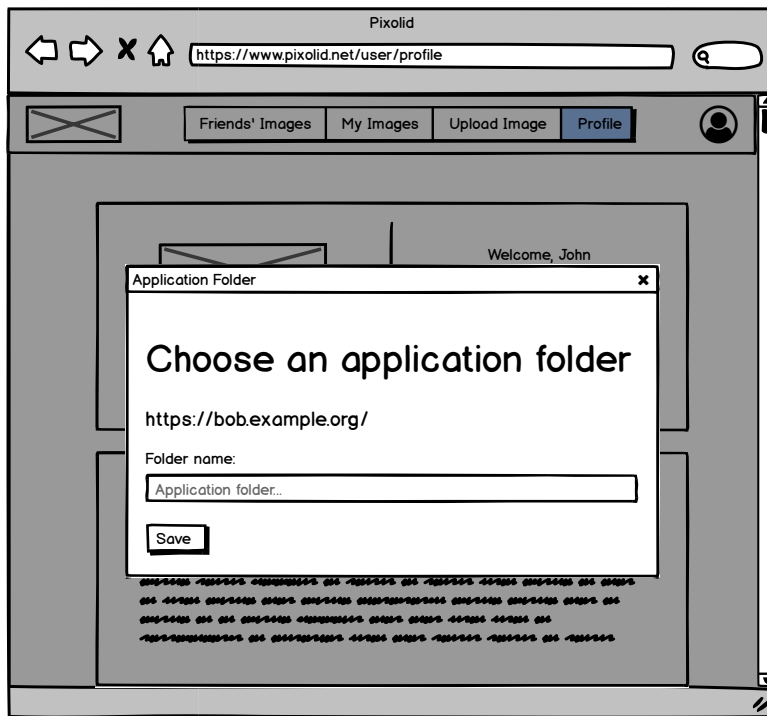


Figure 3.20: Application folder change after the *change folder* button click.

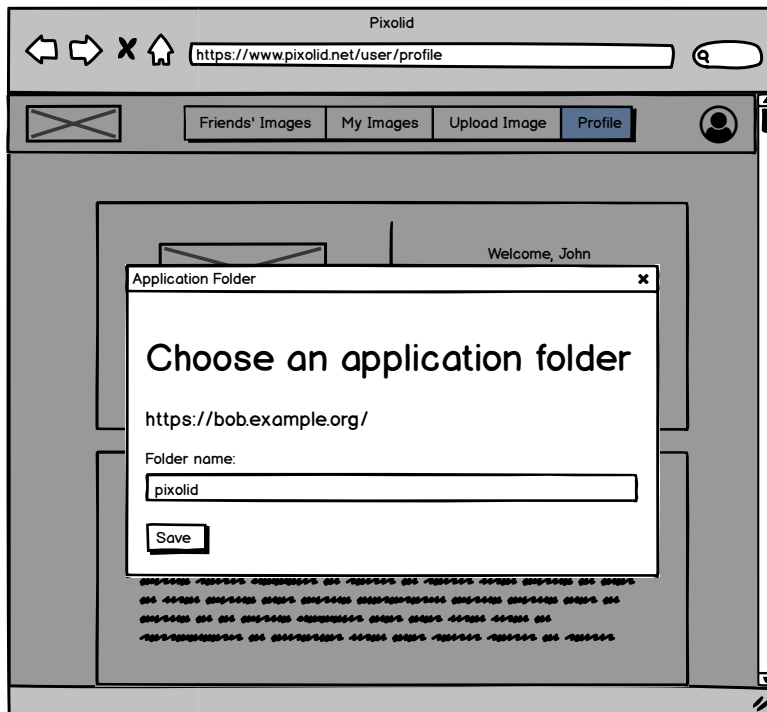





Figure 3.21: The application folder can be typed into the text area and saved.

### 3.1.3 Pixolid Branding

In this section, the proposed branding of the Pixolid application is introduced. The first part showcases an application logo. In the second part, a favicon for the website is unveiled. All of the proposed designs are created as a vector graphic, ready to be used on the Web and in prints.

#### 3.1.3.1 Pixolid Logo

Pixolid shall use a unified brand identity throughout the application. We can see the logo design in Figure 3.22. Its color scheme is based on the design recommendations mentioned in [30]. Specifically, the colors are listed as follows:

- Royal Lavendar (#7C4DFF )
- Cerulean (#18A9E6 )
- Deep Sky Blue (#01C9EA )

The idea behind the logo is to capture the connection between individual pictures shared via Pixolid. The three circles are overlaying each other with a minor opacity. The font which is used for the brand name is called "*DejaVu Sans*". In Figure 3.22, we can also see the design of the logo in black and white colors.

#### 3.1.3.2 Pixolid Favicon

The favicon for the Web application is derived from the logo itself. It is using a shortened *PIX* name with just one circle. That is due to filling the browser's favicon area, of which the dimensions typically are 32x32 pixels. The icon design is shown in Figure 3.24 in various sizes.

## 3.2 Solid Framework Libraries Description

In this section, various Solid framework and linked data libraries are described. The libraries mainly provide a way of accessing and modifying the RDF and other data from HTTP resources. This includes the Solid POD storage. The following libraries are further described:

- rdfib.js,
- LDflex,
- Solid File Client.





Figure 3.22: The Pixolid logo in a colored design.



Figure 3.23: The Pixolid logo in a black and white design.



Figure 3.24: The Pixolid favicon design in various sizes.

#### 3.2.1 rdflib.js

Javascript RDF library for browsers and Node.js (rdflib.js)<sup>40</sup> provides an easy way to retrieve and modify RDF data in Javascript. The primary data access point is a *store*. A store is an object representing the RDF graph. This graph then can be queried via various data access functions such as [31]:

---

```
1 store.any(subject, predicate, object, graph);
2 store.match(subject, predicate, object, graph);
```

---

The *any* function takes an RDF *Quad* statement, and returns any single node satisfying the given statement. The *match* function takes an RDF *Quad* statement, and returns an array of statements matching the given statement.

Also, for the data retrieval from distant resources (which are available through HTTP), there is a *fetcher* object. The fetcher is tied to the store, as the store is passed to the constructor of the fetcher. The fetcher offers the following main function [31]:

---

```
1 fetcher.load(resource);
```

---

This function takes as an argument a desired URL of the resource to be fetched. The resource is then loaded into the store object.

To update local data with their online counterparts, there is a *UpdateManager* (further referred as an *updater*). The updater is equipped with the following main function [31]:

---

```
1 updater.update(deletions, insertions, callback);
```

---

The update function takes an array of deleting and inserting *quad* statements, along with the callback function to be executed after the completion of the update. The distant HTTP resource is then updated with the new information [31].

#### 3.2.2 LDflex

Linked Data flex (LDflex)<sup>41</sup> provides an easy way to query RDF data. There are various ways of accessing such data. The main entry point for data retrieval is a *data* object. Through the *data*, it is possible to retrieve information about a currently logged in user, for example [32]:

---

<sup>40</sup><https://github.com/linkedin/rdf-lib.js/>

<sup>41</sup><https://github.com/solid/query-ldflex>

```
1 data.user.name;  
2 data.user.friends;
```

---

The first one gives us a user's name, and the second one gives us an array of the people listed as friends.

There is also a way to query any URL entry point:

```
1 data['https://example.org/tom#me'].name;
```

---

The above example shows how to get the name predicate of the document on a specific URL.

If we wanted to get a specific predicate from the document, it could be accomplished as follows [32]:

```
1 data.user['http://xmlns.com/foaf/0.1/name'];  
2 data.user['foaf:name'];  
3 data.user.foaf_name;  
4 data.user.foaf$name;  
5 data.user.name;
```

---

The above examples are mutually equivalent.

### 3.2.3 Solid File Client

The Solid File Client<sup>42</sup> is a library which provides an easy way of manipulating files and folders stored on a Solid POD. The library provides a high-level way of file and folder management on a Solid POD, which makes it easy to use. The main file manipulation functions are [33]:

```
1 createFile(url, content, contentType);  
2 readFile(url);  
3 updateFile(url, content, contentType);  
4 deleteFile(url);  
5 copyFile(oldUrl, newUrl);
```

---

The above functions are used for creating, reading, updating, deleting and copying files on a Solid POD storage.

For folder operations there are following functions [33]:

---

<sup>42</sup><https://github.com/jeff-zucker/solid-file-client>

### 3. DESIGN

---

```
1 createFolder(url);
2 readFolder(url);
3 deleteFolder(url);
4 copyFolder(oldUrl, newUrl);
```

---

The above functions enable creating, reading, deleting, and copying of folders stored on a Solid POD.

## 3.3 Access Lists

In this section, an access management of resources stored in Solid is explained. Firstly, a basic workflow of the access management in Solid is introduced. Secondly, access list locations are specified. Thirdly, concrete examples of access lists used by Solid are discussed.

### 3.3.1 Access Control Workflow

A principle of the resource access control in Solid is based on a specification which is further described in [34]. Access to individual resources is dependant on following crucial entities [35]:

- an agent,
- a mode.

Basically, an agent is an identifier of those who have access to a given resource. The agent can be a user or an application identified by a WebID.

A mode is a type of access that a particular agent possesses over a resource. Each mode represents an ability to read, write, append, or control a resource.

It is possible to describe as many authorizations (sets of agents and modes) as desired. By doing this, it is possible to give multiple agents or groups multiple permissions.

Such said entities are then recorded and preserved in access lists. An access list is a file containing information about which agent owns which access rights over a resource [35].

### 3.3.2 Access List Location

Each resource can obtain its specific access list. If the resource does not have a specific access list, its access rights are then inherited from the parent container. In this way, it is possible to mix specific access rights for individual resources with default access rights for a particular directory [35].

Therefore, when a resource is accessed, a server first checks whether or not the resource has a specific access list file. If it does, then the list is used to

determine the access rights. If it does not, the server recursively tries to find an access list by looking up in the directory tree. Eventually, some node in this structure has to have the access list [35] which is then used.

Let us say that we have a photo with the following URI.

```
http://bob.example.org/photos/dog.png
```

This resource could have its specific access rights described in the following access list:

```
http://bob.example.org/photos/dog.png.acl
```

Alternatively, if the resource does not have a specific access list, an access list from a parent container may be used. The parent container would have its access list saved in the following file:

```
http://bob.example.org/photos/.acl
```

### 3.3.3 Access List Example

Access lists are specific RDF files in a Turtle or any other RDF serialization format [35]. Let us say that a person called *Bob* owns the following resource:

```
http://bob.example.org/photos/dog.png
```

Moreover, the said resource would have its permissions set in a specific access list located at:

```
http://bob.example.org/photos/dog.png.acl
```

The appropriate access list could look like this:

```
1 @prefix acl: <http://www.w3.org/ns/auth/acl#> .
2
3 <#owner> a acl:Authorization;
4     acl:agent <https://bob.example.org/profile/card#me>;
5     acl:accessTo <https://bob.example.org/photos/dog.png>;
6     acl:mode acl:Read, acl:Write, acl:Control.
```

In the example above, on the fourth line, the agent statement describes Bob's identity (his WebID). The fifth line states the resource to which the access list specifies access. Line six contains individual modes for the given agent. Specifically, it is permitted to read, write, and control the resource.

## 3.4 Class Diagram

In this section, the main application classes are described, and captured in the class diagram. The following classes resulted from the design details of the Solid framework and the Pixolid application itself.

- `SolidBackend`,
- `App`,
- `Person`,
- `Image`,
- `Comment`,
- `Like`.

In Figure 3.25, we can see relations among the said classes. Let us further describe them in detail.

### 3.4.1 `SolidBackend`

*SolidBackend* is a service class responsible for the data fetching and updating from/to the Solid POD. The class will use the Javascript RDF library for browsers and Node.js (`rdfib.js`) for the data exchange. Retrieved data are going to be preserved in corresponding model classes.

### 3.4.2 `App`

*App* is a class representing the main application worker. The class uses the *SolidBackend* class for data management. It is also a starting point of the *UI* management.

### 3.4.3 `Person`

*Person* is a model class representing individual users of the application. The class has to provide user's WebID, name, and image.

### 3.4.4 `Image`

*Image* is a model class representing images uploaded via the application. The class contains member variables such as a URL of the image itself, a URL of the Turtle file describing the image, its description in a text form, an author (*Person*), and a creation date timestamp.

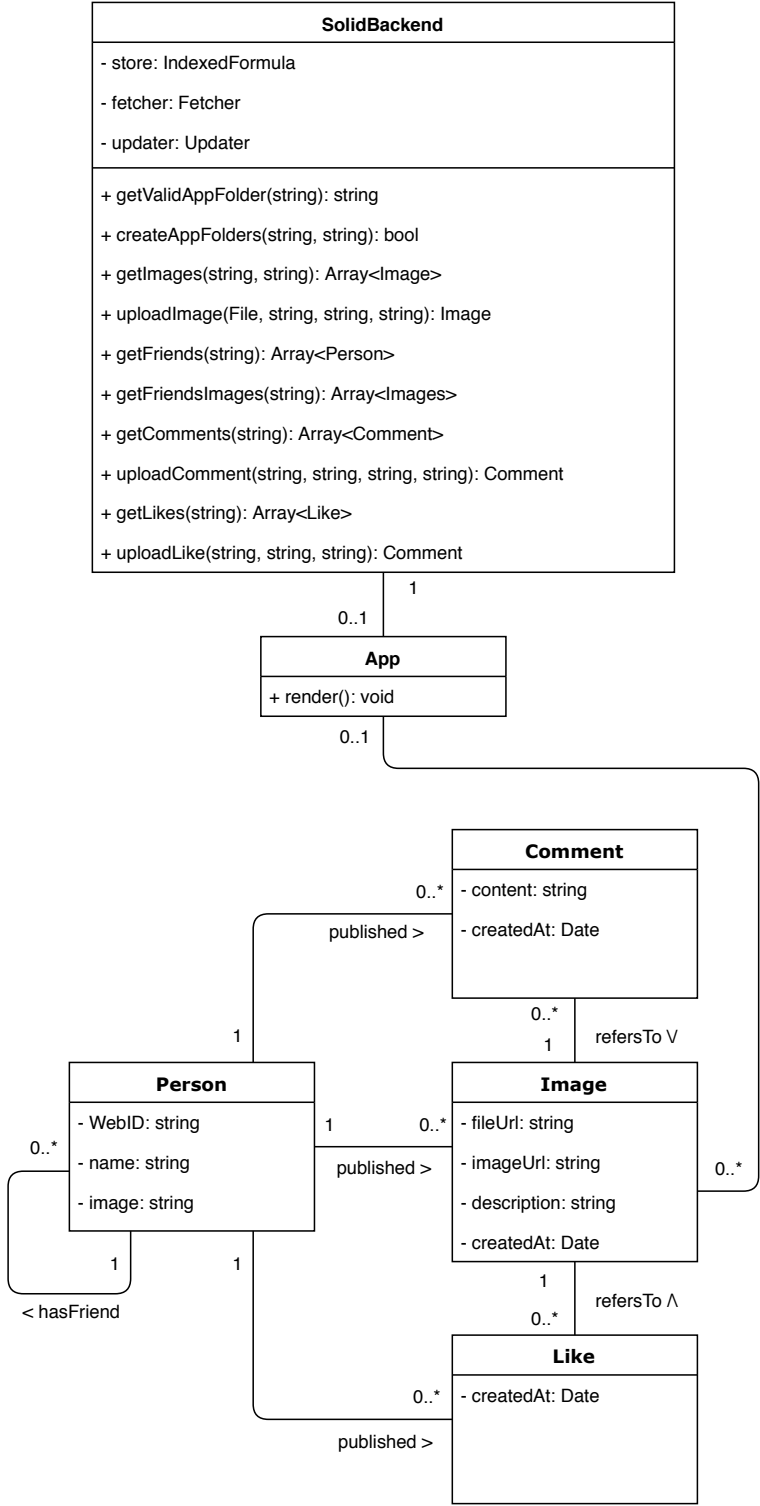


Figure 3.25: Class Diagram: Relations among the classes.

### 3.4.5 Comment

*Comment* is a model class representing each comment given to a certain image. The class contains member variables such as text content of the comment, an author (*Person*), a URL of the commented image, and a creation date timestamp.

### 3.4.6 Like

*Like* is a model class representing each like given to a specific image. The class contains an author of the like (*Person*), a *URL* of the liked image, and a creation date timestamp.



---

# Realization

In this chapter, the process of implementing the Pixolid application is described. At first, we take a look at the technologies which were used and applied in the making of the application. Secondly, the application's basic structure is discussed. Then follows a section with the description of the main implementation focuses, which were inherited both from the analysis and the design, along with the connections between the backend and the frontend. Lastly, we take a look at the final version of the implemented Pixolid application.

## 4.1 Used Technologies

In this section, the main technologies which were used in the process of creating the application are described. The technologies namely are *React* and *Solid React SDK*. Let us further outline their roles in the implementation.

### 4.1.1 React

React<sup>43</sup> is a JavaScript library (or a framework), which can be used to develop and build user interfaces. It is developed by Facebook and the React community<sup>44</sup>.

React is used by a wide variety of developers who are creating modern Web applications. Each application consists of React components, where every component is responsible for a part of the application. Components control their state, from which they are then rendered. Many community-created reusable components exist, which make React a rich and powerful tool for building applications.

---

<sup>43</sup><https://reactjs.org/>

<sup>44</sup><https://github.com/facebook/react>

### 4.1.2 Solid React SDK

Inrupt, the previously mentioned company behind the commercial efforts to spread the Solid framework technologies, are developing a Software Development Kit (SDK) aimed at creating applications based on the Solid framework. As of writing this thesis, the work on the SDK is still in progress.

The SDK provides reusable React components to use in a Solid application development [36]. The components are currently mainly focused on the authorization of the user with his WebID to the Solid POD.

The SDK also offers a *Solid React Application Generator*<sup>45</sup>, which is based on the *Create React App*<sup>46</sup> by Facebook. The generator creates a React project, similarly to Facebook's version. The main difference here is that the SDK's generator creates an app which is already equipped with the Solid authorization components and applied styling according to the design guidelines mentioned in [30]. It encourages the developers to begin developing applications on top of the functional Solid base application built on React.

The SDK also includes the best practices for the application development regarding Solid.

## 4.2 Application's Structure

In Figure 4.1 we can see the application's main directory structure, along with its basic description. Let us further inspect the inner directory structure of the *src* folder, describing the contents of some of its key directories.

In *components*, we can find standalone components such as components for uploading images, showing images, or showing the image detail. The *Containers* directory contains individual container pages which are bound to key application's routes, such as a page for showing friends' images, uploading images, or showing profile information. In *models*, there are classes used for encapsulating the application's model, such classes for images, comments, likes, and persons. The *services* folder contains the service classes providing the data such as the *SolidBackend* class, which retrieves and updates the data on the Solid POD.

For detailed information about how to obtain and run the development version of the application, please refer to Appendix B.

## 4.3 Implementation Focuses

In this section, the main implementation focuses are introduced and commented. At first, let us describe the connection between the frontend and the backend of the application. Secondly, we take a look at how the setting of

---

<sup>45</sup><https://github.com/inrupt-inc/generator-solid-react>

<sup>46</sup><https://github.com/facebook/create-react-app>

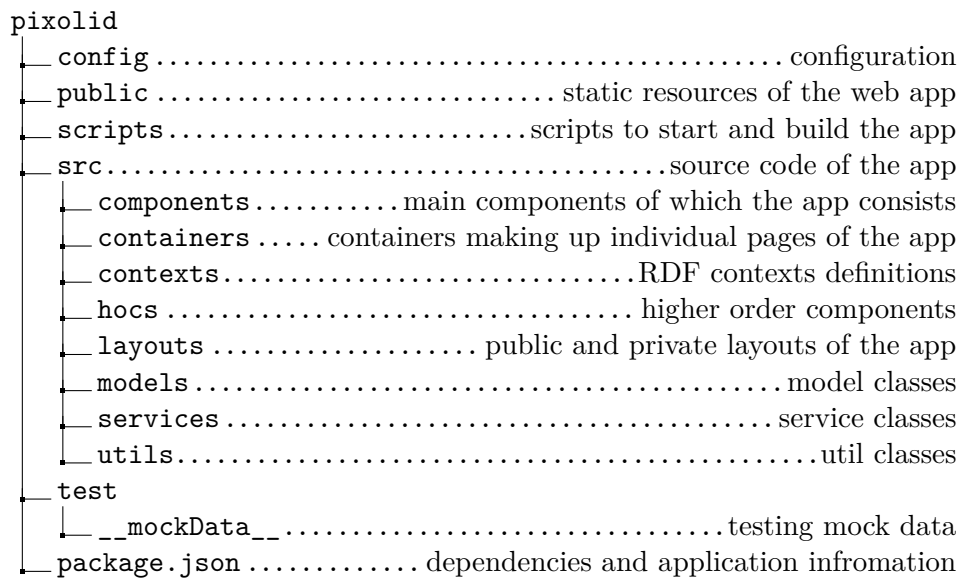


Figure 4.1: Pixolid’s directory structure

the application’s folder was handled. Then we describe uploading new images to a user’s POD. After that, let us discuss showing user’s images as well as friends’ images. In addition to that, we comment on the implementation of showing the image detail. After that, we take a look at adding likes and comments to images. Second to last, a user logging in and out of the application is described. Last but not least, the design and styling of the application are described.

### 4.3.1 Backend and Frontend Connection

The *SolidBackend* service class is responsible for retrieving data from Solid PODs, and uploading new data to Solid PODs. The class itself offers various functions for getting images, persons, comments, likes from multiple Solid POD sources. There are also functions for creating new application’s resources (images, comments, likes, etc.) while handling the later discoverability of the resources. *SolidBackend* also offers a way of specifying and giving access rights to individual images. The interface as mentioned earlier is then called from within the React frontend components.

The React components are managing their state, from which the data is rendered into the User Interface (UI). The components also handle user input, interaction, and trigger calling of the *SolidBackend*’s functions.

### 4.3.2 Choosing the Application Folder

When a user opens the Pixolid application for the first time, it prompts him to choose a location on his Solid POD storage, where the application's data will be stored. This task can be divided into the following tasks:

- detecting the current location of the application folder,
- validating the structure of the application folder,
- prompting the user to enter a location of the application folder,
- creating the application folder,
- registering a record of the new location of the application folder,
- propagating the application folder.

This set of tasks is imminent across frontend and backend as well. For each task, we will explain implementation details and a sequence of events which occur during the said task. Let us further cover the above-specified tasks in detail.

#### 4.3.2.1 Locating the Application Folder

The discovery of the application folder is achieved by looking at the user's profile document, which is tied to his WebID. User's profile document is retrieved by dereferencing his WebID. The RDF profile document is then searched for a triple stating the location of the user's timeline. The triple consists of the user's WebID (subject), the folder's location (object), and the following timeline predicate:

`http://www.w3.org/ns/solid/terms#timeline`

Next follows the example of a profile document containing a specified application folder (note: a full profile document would typically contain additional triples):

---

```
1 @prefix : <#>.
2 @prefix solid: <http://www.w3.org/ns/solid/terms#>.
3 @prefix pix: </public/pixolid/>.
4
5 :me solid:timeline pix:.
```

---

If the profile document does not contain any information about the location of the application folder, then it needs to be specified by the user and

created by the application (will be explained later). Otherwise, the retrieved application folder needs to get validated. The validation process is explained next.

### 4.3.2.2 Validating the Application Folder

After the application folder is located, its content structure needs to get validated. The application folder gets dereferenced, which is achieved by getting the container's RDF document. This document contains information about the container's content. The document is then searched for child containers which contain individual images, comments, and likes.

If the application folder contains said containers, it is considered to be a valid application folder and the application is then able to continue to run without further actions. Otherwise, the application folder needs to be specified by the user, which is explained next.

### 4.3.2.3 Specifying the Application Folder

When the application folder is not specified or considered as valid, the user needs to be prompted to type in a desired location of the folder. This prompt is shown in the form of a modal window, which overlays the current content of the parent page. The component responsible for this behavior is the *FolderModal* component (`pixolid/src/components/FolderModal/`). As a base component to show a window in a modal form, a reusable React component is used, which is called *react-responsive-modal*<sup>47</sup> by Léo Pradel. This component allows us to build further specific UI responsible for the desired functionality of choosing the application folder.

The modal shows the main storage root folder which is derived from the user's WebID. Then, there is a text area, where the user can type in any further directory structure, which in concatenation with the main root folder will be used as an application folder. The folder name then gets validated. A folder name can only consist of alphanumeric characters of non-zero length, which can be divided, prefixed, or suffixed by slashes. It is allowed to specify subdirectories as well.

If the folder name is not valid, the user is made aware of this fact by underlining the text area to red, and by an alert stating to choose a correct folder structure.

Otherwise, if the folder structure is correct, then the user can confirm it, and the folders are created in a way which is described next.

---

<sup>47</sup><https://www.npmjs.com/package/react-responsive-modal>

#### 4.3.2.4 Creating the Application Folder

The correctly specified folder is created, along with the desired subfolder structure. For the process of creating the folders, the *createFolder* function from the *solid-file-client* is used, which was described in 3.2.3. If any of the folders already existed, they would stay untouched.

With the creation of the application folder, it is needed to make sure that other users can read the contents of the folder. We create an access list which permits to read the contents of the application folder (while it is possible to select which content can be read or not later in uploading). This access list is created in the form of triple statements, which are then passed to the *rdflib.js*' store for further handling. Let us consider the following RDF prefixes:

---

```
1 @prefix : <#>.
2 @prefix acl: <http://www.w3.org/ns/auth/acl#>.
3 @prefix pix: <./>.
4 @prefix card: </profile/card#>.
5 @prefix foaf: <http://xmlns.com/foaf/0.1/>.
```

---

The important one here is the following access list vocabulary (line 2):

<http://www.w3.org/ns/auth/acl#>

The access list then looks like this:

---

```
1 :owner
2   a acl:Authorization;
3   acl:accessTo pix:;
4   acl:agent card:me;
5   acl:defaultForNew pix:;
6   acl:mode acl:Control, acl:Read, acl:Write.
7 :public
8   a acl:Authorization;
9   acl:accessTo pix:;
10  acl:agentClass foaf:Agent;
11  acl:defaultForNew pix:;
12  acl:mode acl:Read.
```

---

In the above access list, we can see that the owner has full control over the folder including reading and writing, while the public access is restricted to read-only mode. The access list is then created using the *updateFile* function, which was specified in 3.2.3.

#### 4.3.2.5 Registering the Application Folder

After the creation of the folder, it needs to get registered into the user's profile document. We simply dereference the document using the `rdflib.js` (see 3.2.1). Then, we update the RDF document by deleting the previous application folder triple and inserting the new one. After its updating, the profile document (with additional profile information) could look like:

```
1 @prefix : <#>.
2 @prefix solid: <http://www.w3.org/ns/solid/terms#>.
3 @prefix fold: </my/new/folder/>.
4
5 :me solid:timeline fold:.
```

#### 4.3.2.6 Propagating the Application Folder

When the process of detecting or setting the application folder is finished, the parent React component then saves the location to its state. Other components are then able to use it as a starting point to fetch and update additional data.

### 4.3.3 Uploading Images

When a user wants to upload a new image, he can start the process on the *upload image* page. The page component is called *Upload Image* (`pixolid/src/containers/UploadImage/`). This page contains the *Image Uploader* component (`pixolid/src/components/ImageUploader/`), which is responsible for the process of uploading new images. This component handles the responsibility for:

- setting an image description,
- selecting an image,
- setting access rights,
- uploading the image.

Let us further describe each task in detail.

#### 4.3.3.1 Setting an Image Description

The user can type in a description of the image into the *image description* text area. The description text is not mandatory. The text is being preserved in the parent's component state, as the user types.

### 4.3.3.2 Selecting an Image

The user can select an image by clicking on the *select image* button. This button opens up a native file dialog window, where the user selects the desired image file.

To achieve showing the dialog window, a reusable React component is used, which is called *react-file-reader*<sup>48</sup> by Travis Mathis. This component allows us to specify allowed file types (in our case images), and get selected files in a HTML5 *FileList*<sup>49</sup> and in *Base64*<sup>50</sup>. The *FileList* is important for later uploading of the image, while the *Base64* representation can be used to preview the image on the page directly.

The selected image is then preserved in the parent's component state and is rendered on the *upload image* page as a preview.

### 4.3.3.3 Setting Access Rights

If the user wants to specify image access rights, there is a toggle input with its two states. The toggle is labeled as *Public*. By default, the toggle is in an *on* state (set to true). The user can turn the toggle off and on by clicking on it.

Being toggled *on* denotes that the currently uploaded image will have public access rights set. That means the image will be discoverable and available to see by anyone who can access it. In an *off* state, the currently uploaded image will have private access rights set. By doing that, the image will be discoverable and available only to the further specified group of users (which are identifiable by an WebID).

When the access rights are set to private, a *user selection* select input appears. As a base component for this input, a reusable React component is used, which is called *react-select*<sup>51</sup> by Jed Watson. This component allows us to give the user the ability to choose from multiple given options. In our user selection input, the user can select multiple users which will be allowed to see the uploaded image. The users shown in the select input menu are the user's friends, which are fetched from the user's profile document. This is done by dereferencing the profile document via *rdflib.js*' store, and searching for the following statements:

```
http://xmlns.com/foaf/0.1/#knows
```

The user's friends can be listed in an RDF profile document in the following way:

---

<sup>48</sup><https://www.npmjs.com/package/react-file-reader>

<sup>49</sup><https://developer.mozilla.org/en-US/docs/Web/API/FileList>

<sup>50</sup><https://tools.ietf.org/html/rfc4648>

<sup>51</sup><https://react-select.com/>



---

```

1 @prefix : <#>.
2 @prefix foaf: <http://xmlns.com/foaf/0.1/>.
3 @prefix bob: <https://bob.example.org/profile/card#>.
4 @prefix tom: <https://tom.example.org/profile/card#>.
5
6 :me
7   foaf:knows bob:me;
8   foaf:knows tom:me.

```

---

The above statements say that the user, who is the owner of the profile document mentioned above, knows two people (Bob and Tom), who are identified by their WebIDs.

If the user wishes, it is possible not to select any users as well. The image is then accessible only to the user who uploaded the image. Otherwise, the selected users are then preserved into the parent's component state.

After the user specified suitable access rights, it is needed to create appropriate access lists. When the image is public, the following access list is then created through the `rdflib.js` statements:

---

```

1 @prefix : <#>.
2 @prefix acl: <http://www.w3.org/ns/auth/acl#>.
3 @prefix card: </profile/card#>.
4 @prefix foaf: <http://xmlns.com/foaf/0.1/>.
5
6 :owner
7   a acl:Authorization;
8   acl:accessTo <1553894090984.8718.ttl>;
9   acl:agent card:me;
10  acl:mode acl:Control, acl:Read, acl:Write.
11 :public
12  a acl:Authorization;
13  acl:accessTo <1553894090984.8718.ttl>;
14  acl:agentClass foaf:Agent;
15  acl:mode acl:Append, acl:Read.

```

---

The above access list says that the owner of the image has full control rights, whereas public access is in the mode of reading and appending. When the image is set to be private, with a non-empty set of allowed users, the following access list is then created:

---

```

1 @prefix : <#>.
2 @prefix acl: <http://www.w3.org/ns/auth/acl#>.

```

---

## 4. REALIZATION

---

```
3 @prefix card: </profile/card#>.
4 @prefix bob: <https://bob.example.org/profile/card#>.
5 @prefix tom: <https://tom.example.org/profile/card#>.
6
7 :owner
8   a acl:Authorization;
9   acl:accessTo <1553894090984.8718.ttl>;
10  acl:agent card:me;
11  acl:mode acl:Control, acl:Read, acl:Write.
12 <>
13  a acl:Authorization;
14  acl:accessTo <1553894090984.8718.ttl>;
15  acl:agent bob:me;
16  acl:mode acl:Append, acl:Read.
17 <>
18  a acl:Authorization;
19  acl:accessTo <1553894090984.8718.ttl>;
20  acl:agent tom:me;
21  acl:mode acl:Append, acl:Read.
```

The above example of the access list shows how the owner once again has the full access rights, while selected individual users (Bob and Tom), possess read and append access rights. Note that no other user will be allowed access to the objected image.

If the picture were meant to be entirely private, the authorization nodes for Bob and Tom (lines 12-16 and 17-21) would be non-existing. The access list would not change in any other way.

### 4.3.3.4 Uploading the Image

Once the image file is set, along with its description and access rights, it is ready to get uploaded to the user's POD by clicking on the *upload image* button. The actual image file (in an image file format such as Portable Network Graphics (PNG) or Joint Photographic Experts Group (JPEG)) is uploaded via the *createFile* function from the *solid-file-client*. In addition to that, an appropriate access file is created as well, with read-only rights for the selected group of users.

With the image file uploaded, it is needed to create an RDF image file, which describes the said image. In this file, we store information about the image location (URL), description, timestamp of creating, and the WebID of the author. Turtle content of the file is created with the *rdflib.js*' statements, and then uploaded as a file via the *createFile* function from the *solid-file-client*. This file utilizes the following structure in RDF Turtle:

```

1  @prefix : <#>.
2  @prefix ns: <http://rdfs.org/sioc/ns#>.
3  @prefix terms: <http://purl.org/dc/terms/>.
4  @prefix XML: <http://www.w3.org/2001/XMLSchema#>.
5  @prefix card: </profile/card#>.
6  @prefix foaf: <http://xmlns.com/foaf/0.1/>.
7
8  <>
9      a ns:Post;
10     terms:created "2019-01-15T21:14:51.493Z"^^XML:dateTime;
11     terms:creator card:me;
12     terms:description "Enjoying a wonderful day.";
13     foaf:depiction <1553894090984.8718.jpeg>.

```

The above example states that a particular user (line 11), created an image (line 13), with a given description (line 12) at the said time (line 10). Along with that, the access lists specified in the previous section are uploaded to the POD. After the upload process is complete, the user can continue by uploading another image.

#### 4.3.4 Showing Images

When a user wants to show either his or his friends' uploaded images, he starts on the *my images* or *friends' images* page. Those pages are container components, which use a same child component called *Images* (`pixolid/src/components/Images/`). This component is responsible for showing a given set of images in a grid layout. The *MyImages* (`pixolid/src/containers/MyImages/`) and *FriendsImages* (`pixolid/src/containers/FriendsImages/`) components fetch appropriate image data into their states, and render them via the *Images* child component.

The process of showing images consists of the following tasks:

- fetching images,
- showing image previews.

At first, let us specify the fetching process for the user's images, followed by the friends' images. After that, we describe the process of showing the image previews.

##### 4.3.4.1 Fetching User's Images

When we want to fetch the user's images, we start with his WebID. The container components also possess the application folder location. The *images*

subdirectory of the application folder gets searched for the images. This process is done by first dereferencing the images container document via `rdflib.js`. Image files then get individually dereferenced, and RDF triples parsed into the *Image* model class. The images then get sorted by the creation timestamp in the order from latest to oldest.

### 4.3.4.2 Fetching Friends' Images

When we want to fetch the images posted by the user's friends, we start with the list of his friends' WebIDs. The process of getting the user's friends was described in 4.3.3.3. Individual friends' WebIDs are checked for the application folder. If it exists, for each friend we further continue by searching for images as in 4.3.4.1.

Let us assume that the owner of a particular image set private access rights to specific users. If the currently signed in user is not one of them (meaning that the image cannot be accessed by the current user), the image is therefore not included in the retrieved images set. Only the accessible images for the current user are included in the final set of fetched images. The images from all of the friends are then sorted by the creation timestamp in the order from latest to oldest.

### 4.3.4.3 Showing Image Previews

When individual images are fetched, it is needed to display them. The *Images* component is responsible for this task. The images are passed from the parent component and displayed in a grid layout. For the purpose of creating a responsive grid, a reusable library is used, which is called *flexboxgrid*<sup>52</sup> by Kristofer Joseph. It enables usage of the well-known *col-xx-\** classes, where *xx* stands for a screen size (*xs*, *sm*, *md*, *lg*), and *\** stands for how much of the screen should the element take up (numbers range from 1 to 12, meaning the 1/12 of the screen, up to the 12/12 of the screen).

The individual grid items are image previews. Each image preview consists of a label stating the creator's name, and the image itself, which is a clickable circle image. The image preview items then fill up the available screen.

### 4.3.5 Showing a Detail of the Image

When a user clicks on any of the *image preview* items on the *user images* or *friends' images* page, an image detail modal window of the image is shown. As a base component for the purposes of showing a window in a modal form, a reusable React component is used, which is called *react-responsive-modal*<sup>53</sup> by

---

<sup>52</sup><http://flexboxgrid.com/>

<sup>53</sup><https://www.npmjs.com/package/react-responsive-modal>

Léo Pradel. A component responsible for showing the image detail is the *ImageDetailModal* component (`pixolid/src/components/ImageDetailModal/`). The image to be shown along with the creator gets passed to this component from the parent container component. This component also triggers fetching comments and likes related to the given image, which are then bound to the component state.

The process of showing a detail of the image can be divided into the following tasks:

- showing image information,
- fetching likes,
- showing likes,
- fetching comments,
- showing comments.

Let us further describe each task in detail.

### 4.3.5.1 Showing Image Information

All of the needed information about the image is already obtained from the parent container component. The primary source of information is an instance of the *Image* class. Also, the image creator is in an instance of the class *Person*.

In a detail modal window, the most important thing is to show the image in full size. The image takes up space according to the available screen size. It also takes into consideration to have enough space to show the description and the comment section.

The description box is located beneath the image. On the left side, it contains information about the creator, namely his name and profile image. On the right side, there is a date when the image was created. Right below the date, there is a like button, which will be described in the next section. Below all of that, there is a text of the image description.

### 4.3.5.2 Fetching Likes

To be able to show a number of likes a given image received, we need to fetch individual like statements first. Our starting point here is the URL of the image RDF file, which is in the instance of the *Image* model class. The file gets dereferenced by the `rdflib.js`' store. After that, the file is searched for like statements. An example of such like statement is the following:

## 4. REALIZATION

---

```
1 @prefix : <#>.
2 @prefix as: <https://www.w3.org/ns/activitystreams#>.
3 @prefix bob: <https://bob.example.org/public/pixolid/likes/>.
4
5 bob:1553543249465.92.ttl as:type as:Like.
```

---

We could find multiple above-mentioned statements among others in the current image RDF file. This statement (line 5) tells us, that under the given URL we can find a like from another user (located in a different Solid POD storage). Each individual like statements are then dereferenced once again. The dereferenced like statement (in the other user's POD) looks like this:

```
1 @prefix : <#>.
2 @prefix as: <https://www.w3.org/ns/activitystreams#>.
3 @prefix card: </profile/card#>.
4 @prefix XML: <http://www.w3.org/2001/XMLSchema#>.
5 @prefix tom: <https://tom.example.org/public/pixolid/images/>.
6
7 <>
8   as:actor card:me;
9   as:object tom:1553356280581.4055.ttl;
10  as:published "2019-01-25T19:47:29.465Z"^^XML:dateTime;
11  as:type as:Like.
```

---

The above statements tell us, that a given user (line 8), liked (line 11) an image (line 9), at a given time (line 10). All of these statements get parsed into an instance of the *Like* model class. All the likes then get returned for further actions.

### 4.3.5.3 Showing Likes

Fetches likes then get saved into the state of the *ImageDetailModal* component. The like button has a like counter tied to it. The number of fetched likes gets passed to the counter, and then rendered.

Also, it is needed to detect whether or not the currently signed in user has already liked the image. That is achieved by filtering the set of likes by the creator's WebID. If the user's WebID is found among the likes, we know that he liked the image already. Otherwise we conclude that he has not liked it yet. Depending on the like status, the like button is rendered either in an outlined (not liked yet), or in a solid (liked already) visual variant.

#### 4.3.5.4 Fetching Comments

To be able to show comments related to the image, it is needed to fetch them first. Once again we start with the URL of the image RDF file, which is in the instance of the *Image* class. The file gets dereferenced by the `rdflib.js`' store. Afterwards, the file is searched for the individual comment statements. A comment statement contained in the RDF image file looks like this:

```

1 @prefix : <#>.
2 @prefix as: <https://www.w3.org/ns/activitystreams#>.
3 @prefix bob: <https://bob.example.org/public/pixolid/comments/>.
4
5 bob:1553543310112.2375.ttl as:type as:Note.

```

Such statements can be found among others in the RDF image file. The statement tells us that under the given URL we can locate a comment (which can be in another Solid POD storage) related to the current image.

All of those comment statements get dereferenced again. A dereferenced comment file then can look like this:

```

1 @prefix : <#>.
2 @prefix as: <https://www.w3.org/ns/activitystreams#>.
3 @prefix card: </profile/card#>.
4 @prefix XML: <http://www.w3.org/2001/XMLSchema#>.
5 @prefix tom: <https://tom.example.org/public/pixolid/images/>.
6
7 <>
8   as:actor card:me;
9   as:content "What a nice picture, man!";
10  as:inReplyTo tom:1553356280581.4055.ttl;
11  as:published "2019-01-26T19:47:29.465Z"^^XML:dateTime;
12  as:type as:Note.

```

The above statements say that a given user (line 8), commented (line 12) on the image (line 10), with a certain message (line 9), at a given time (line 11). Those statements then get parsed into an instance of the *Comment* model class. Retrieved comments are then sorted according to the creation timestamp in the order from latest to oldest, and returned for further actions.

Furthermore, after the comments are fetched, it is needed to fetch creators of the comments, in order to be able to show their names and profile images. To achieve this, a set of comments gets mapped onto the unique creators' WebIDs. Those WebIDs get dereferenced, essentially retrieving the users' profile documents. In those documents, we are interested in the following statements:

## 4. REALIZATION

---

```
1 @prefix : <#>.
2 @prefix ns: <http://www.w3.org/2006/vcard/ns#>.
3 @prefix foaf: <http://xmlns.com/foaf/0.1/>.
4
5 :me
6   ns:hasPhoto <profile.jpg>;
7   foaf:img <profile.jpg>;
8   foaf:name "Bob".
```

The above statements mean that the owner of the profile document has a profile photo (line 6 or line 7) and is named *Bob* (line 8). Those statements get parsed into an instance of the *Person* model class. The comments' creators then get returned for further use.

### 4.3.5.5 Showing Comments

Fetches comments along with their creators are saved in the *ImageDetailModal* component's state. For each comment, it is needed to create a comment box containing the comment's data.

A single comment contains the creator's name and profile image. This information can be retrieved from the appropriate *Person* instance for a given comment. To be able to get such instance, all creators are stored in a key-value map, which is accessible via keys made from creators' WebIDs. From the *Image* instance, it is possible to get the creator's WebID, and then use it to get the appropriate *Person* instance of the creator.

On the right side of the comment box, there is a comment creation timestamp, which is retrieved from the comment instance.

The comment text is then rendered below as last.

### 4.3.6 Adding Likes to Images

Let us assume that a user wants to give a like to an image. This process is handled by the *ImageDetailModal* component, and can be divided into the following tasks:

- initiation of the like,
- uploading the like.

Let us further describe the tasks in detail.

#### 4.3.6.1 Initiation of the Like

A user can start the intention of liking an image by clicking on the *like* button located in the image detail. If the user has not liked the image yet, the button



is clickable, and the like is uploaded in a way described in the next section. If the user already liked the image, the button is not clickable and cannot be liked again.

#### 4.3.6.2 Uploading the Like

A like consists of the creator, the like target (the URL of the liked image), and the timestamp of making the action. The appropriate RDF statements get created via the `rdflib.js`' store. An example of such like statements was mentioned in 4.3.5.2. The like is then uploaded to the creator's Solid POD via the *solid-file-client*'s `createFile` function. The like itself is created in a way to be accessible publicly.

Along with the like statement itself, it is needed to somehow mention it in the image file that is being liked. For this purpose, a statement mentioned in 4.3.5.2 is created and inserted into the image file via `rdflib.js`' updater.

The like is then added into the *ImageDetailModel* component's state, causing re-rendering of the like button as already liked.

#### 4.3.7 Adding Comments to Images

Let us say that a user wants to comment on an image. This process is handled by the *ImageDetailModal* component, and separated into the following tasks:

- setting the comment content,
- uploading the comment.

Let us further describe individual tasks in detail.

##### 4.3.7.1 Setting the Comment Content

When a user wants to type a comment to a given image, an *add comment* box is the starting point of this action. The box is located below all listed comments. The user can start typing inside the text area, and the comment text gets preserved in the *ImageDetailModal* component's state.

##### 4.3.7.2 Uploading the Comment

A comment entity consists of the creator, the comment content (text), the target it replies to (the URL of the commented image), and the timestamp of creation. For the comment entity, equivalent RDF statements need to be created. This is done by using the `rdflib.js`' store. How those statements look like was mentioned in 4.3.5.4. After the statements are created, they are uploaded to the creator's Solid POD via the *solid-file-client*'s `createFile` function. The comment itself is created in a way to be accessible publicly.

Just as with the likes, it is needed to register the comment in the image file that is being commented on. For this purpose, a statement which says that a particular comment is at a certain URL (see 4.3.5.4) is created and inserted into the image file via `rdflib.js`' updater.

After the creation of the comment is complete, it is added into the *ImageDetailModel* component's state. That causes the component to render the new comment among others.

### 4.3.8 Login and Logout

The application is required to allow users to sign in via their WebIDs. The process of logging users in is handled by the *Login* component (`pixolid/src/components/Login/`). This component was contained in and created by the *Solid React Application Generator*<sup>54</sup>. It contains *ProviderLogin*, which is a reusable component from *Solid React Components Library*<sup>55</sup>

The user selects a login provider, gets redirected towards the provider's login page, and after a successful authentication is redirected back to the application.

To access the currently signed in user's WebID throughout the application, *Solid React Components Library* provides a wrapping Higher Order Component (HOC) called *withWebId*. This component is used to wrap individual components, which get the currently signed in user's WebID as their component's *prop* (component's argument). The WebID is then accessed within the component by calling:

---

```
1  this.props.webId;
```

---

For logging the user out of the application, in *Solid React Components Library*, there is a component called *LogoutButton*. This component handles the logging out of the application.

### 4.3.9 Application's Design

In this section, we further describe the styling of the React components. Throughout the application, the styling based on the design recommendations mentioned in [30], is used.

The base from the Solid React SDK is using *styled components*<sup>56</sup> to style the application. Styled components enable creating components with custom Cascading Style Sheets (CSS) styles, and use them directly within the React application.

---

<sup>54</sup><https://github.com/inrupt-inc/generator-solid-react>

<sup>55</sup><https://github.com/inrupt-inc/solid-react-components>

<sup>56</sup><https://www.styled-components.com/>

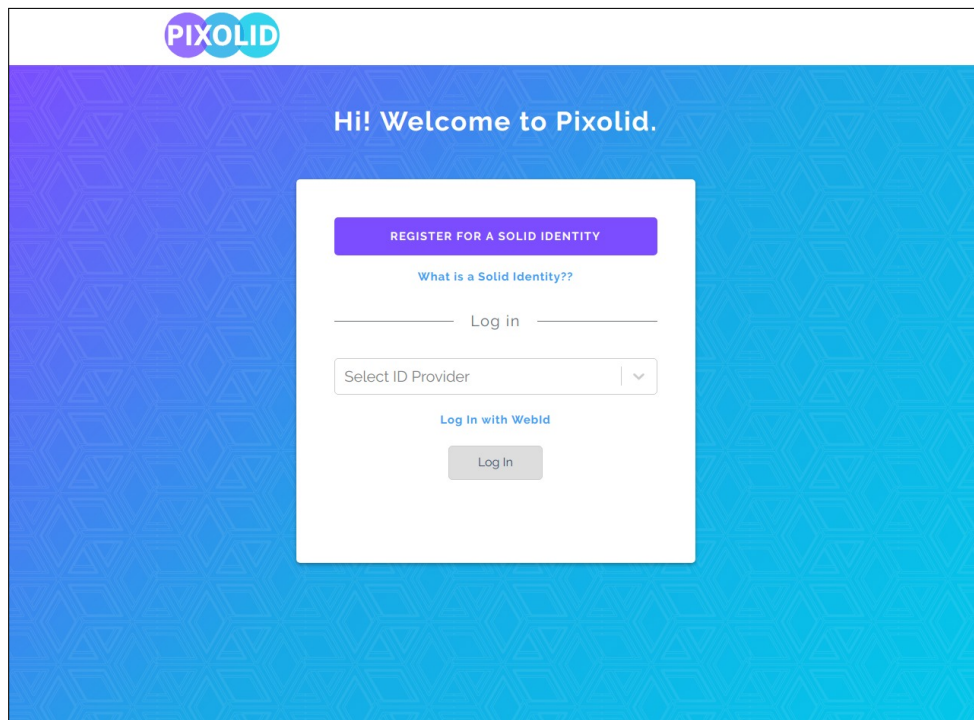


Figure 4.2: Login screen with a provider.

Main application pages such as *user images*, as well as individual *image previews*, comments, and others, were also styled with the custom styled components.

## 4.4 Final Version of the Pixolid Application

In this section, it is showcased how the application looks in its final version of the implementation.

## 4. REALIZATION

---

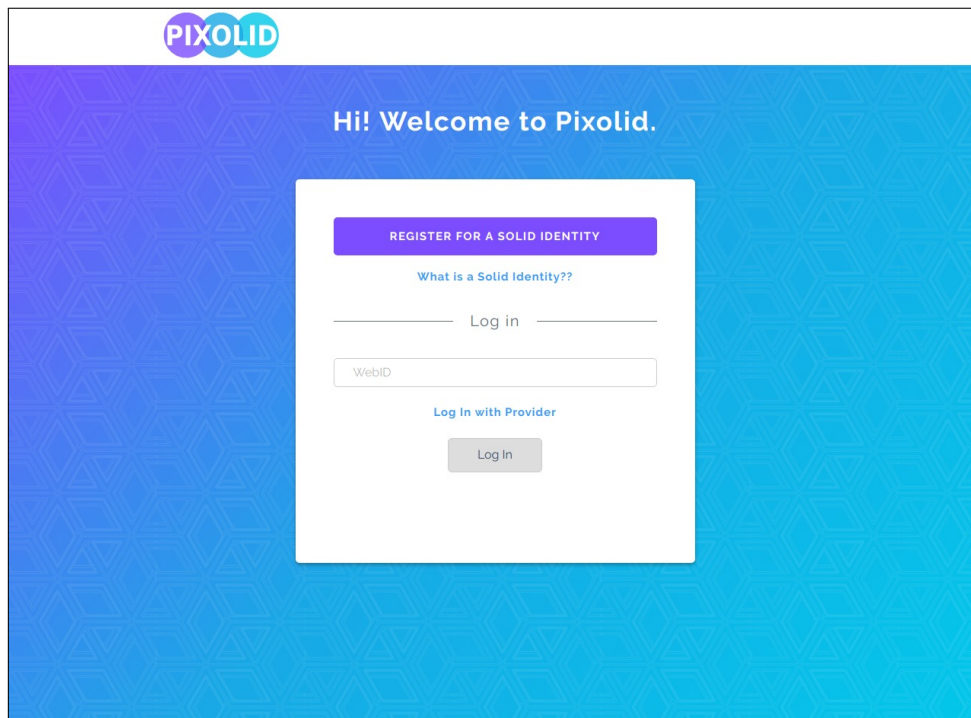


Figure 4.3: Login screen with a WebID.

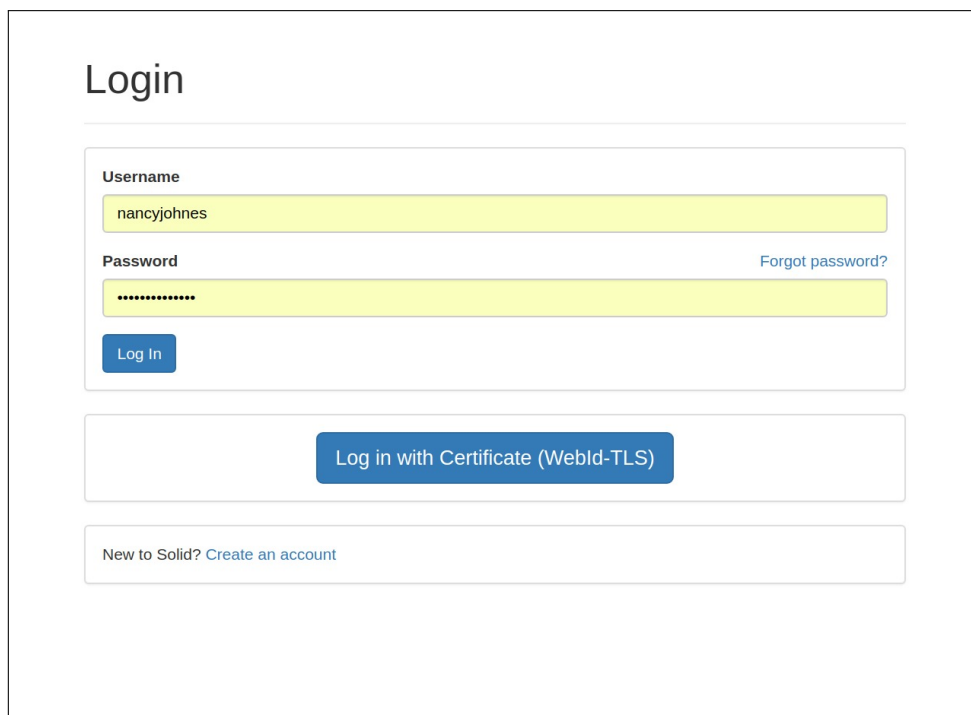


Figure 4.4: Login screen at the provider.

#### 4.4. Final Version of the Pixolid Application

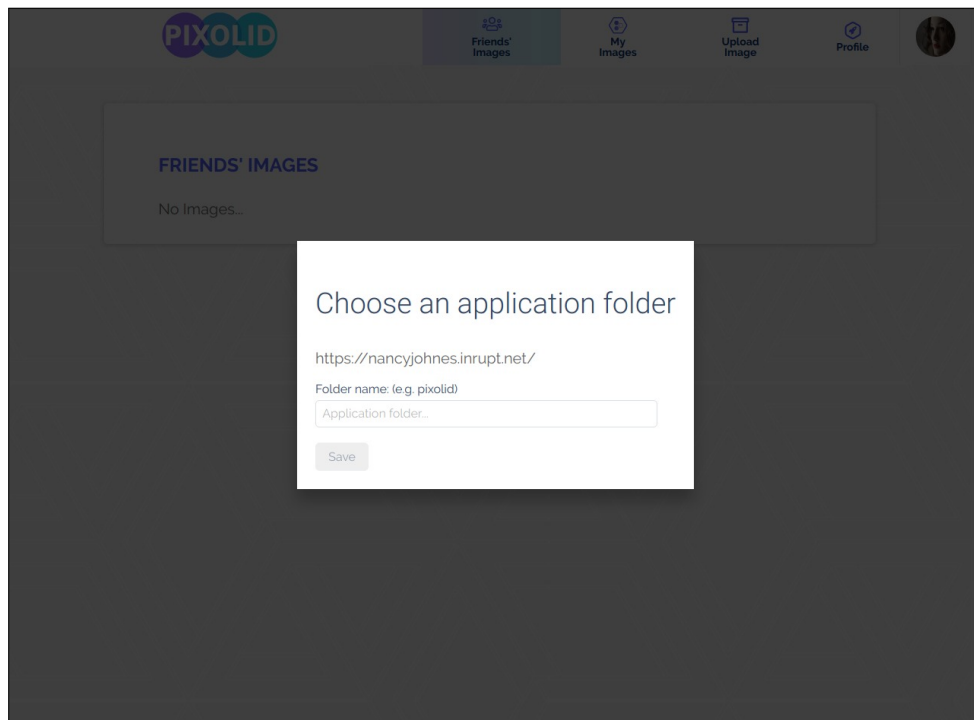


Figure 4.5: The user is prompted to choose an application folder.

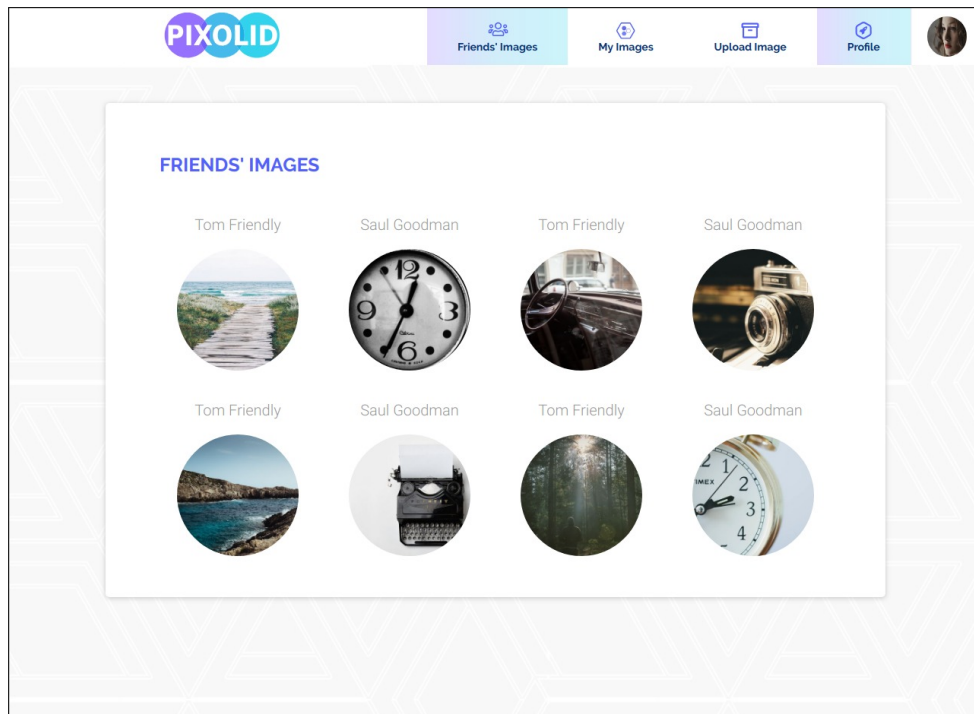


Figure 4.6: An overview of the images posted by the user's friends after login.

#### 4. REALIZATION

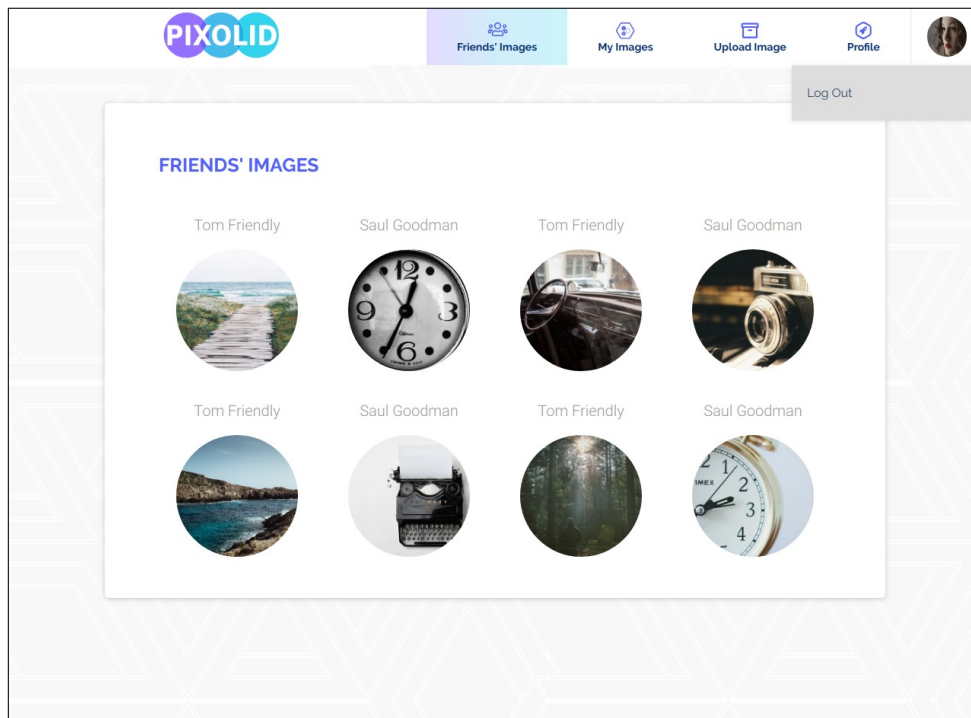


Figure 4.7: Tab bar showing a logout option.

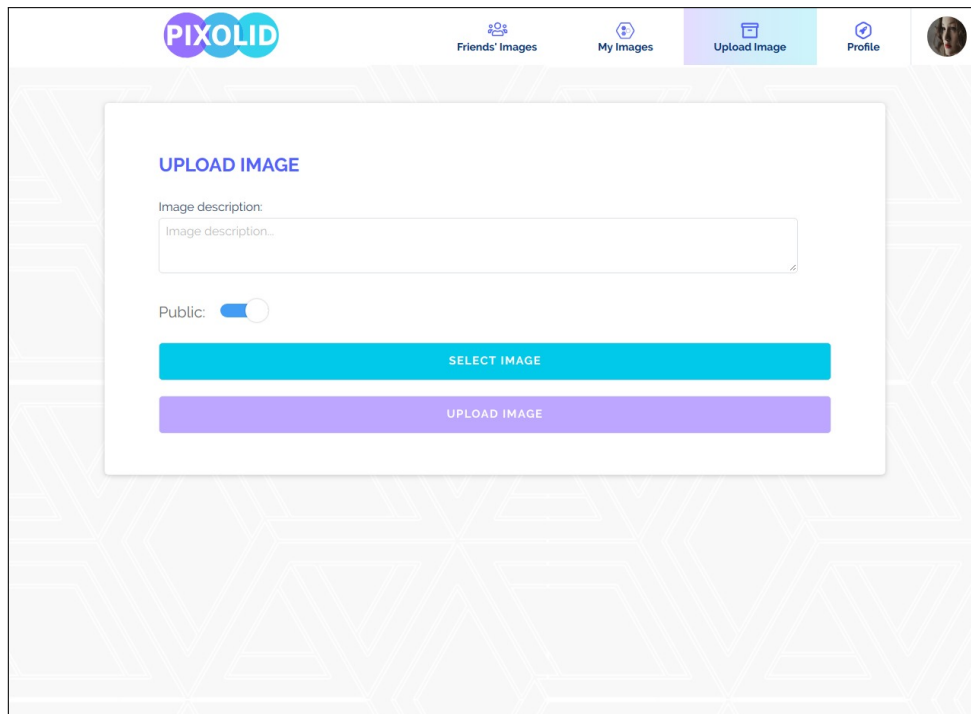


Figure 4.8: Upload image screen providing upload.

#### 4.4. Final Version of the Pixolid Application

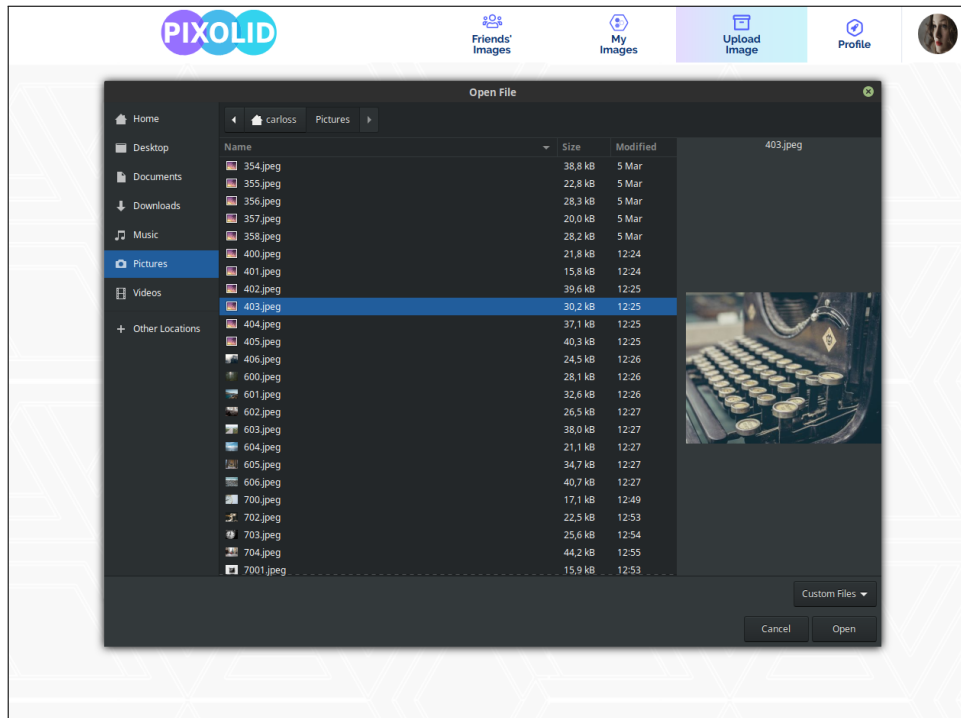


Figure 4.9: Dialog shown during image selection for upload.

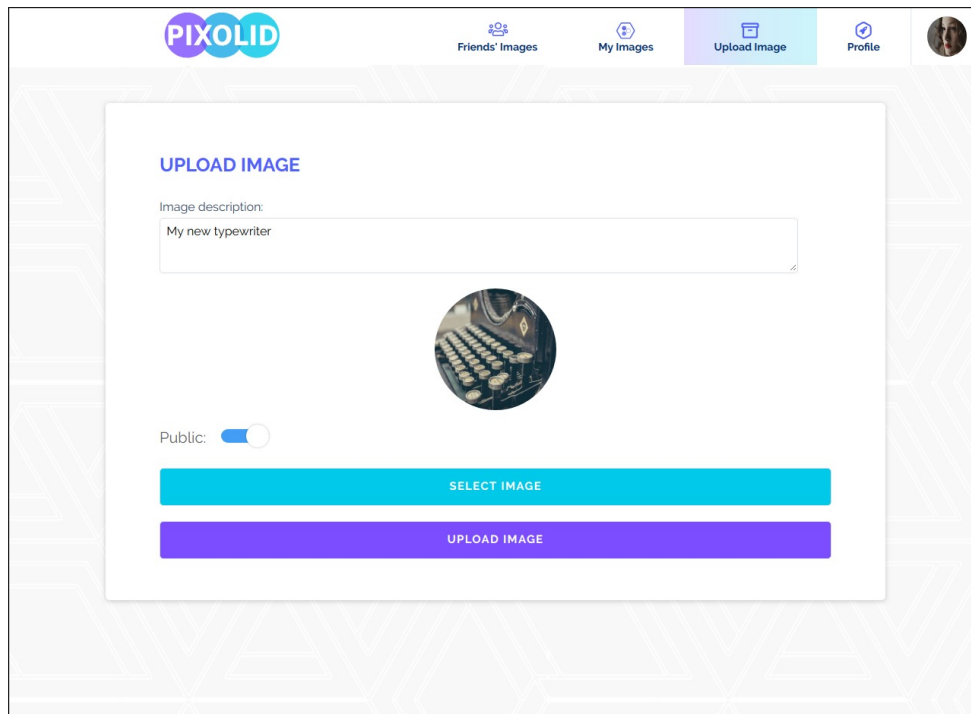


Figure 4.10: An image preview is shown after the selection of the image.

#### 4. REALIZATION

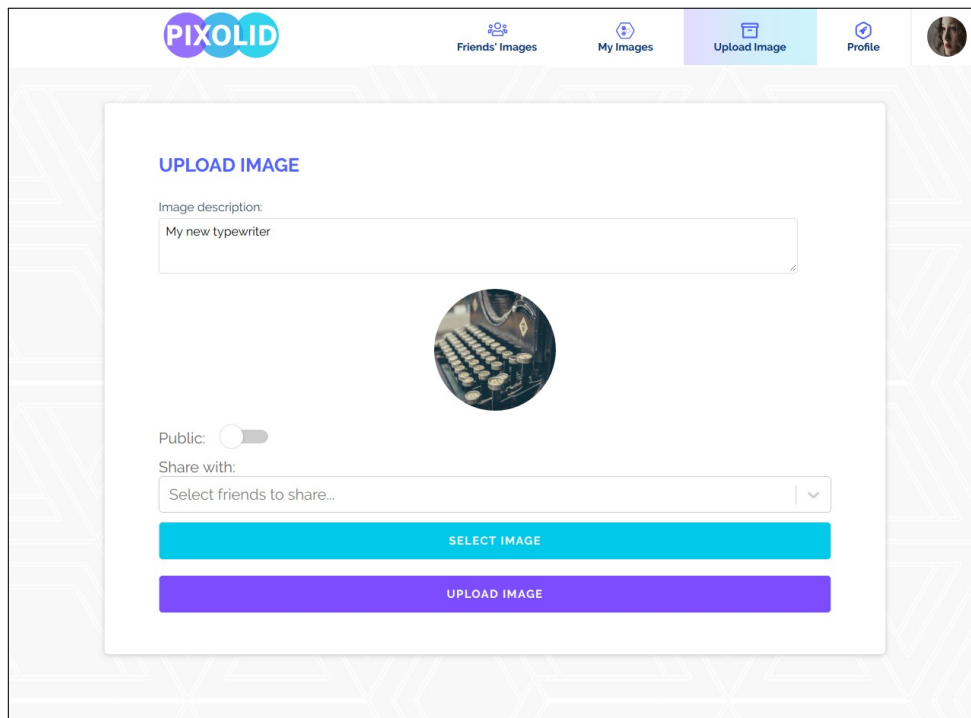


Figure 4.11: If the public sharing is off, the user can choose individual friends.

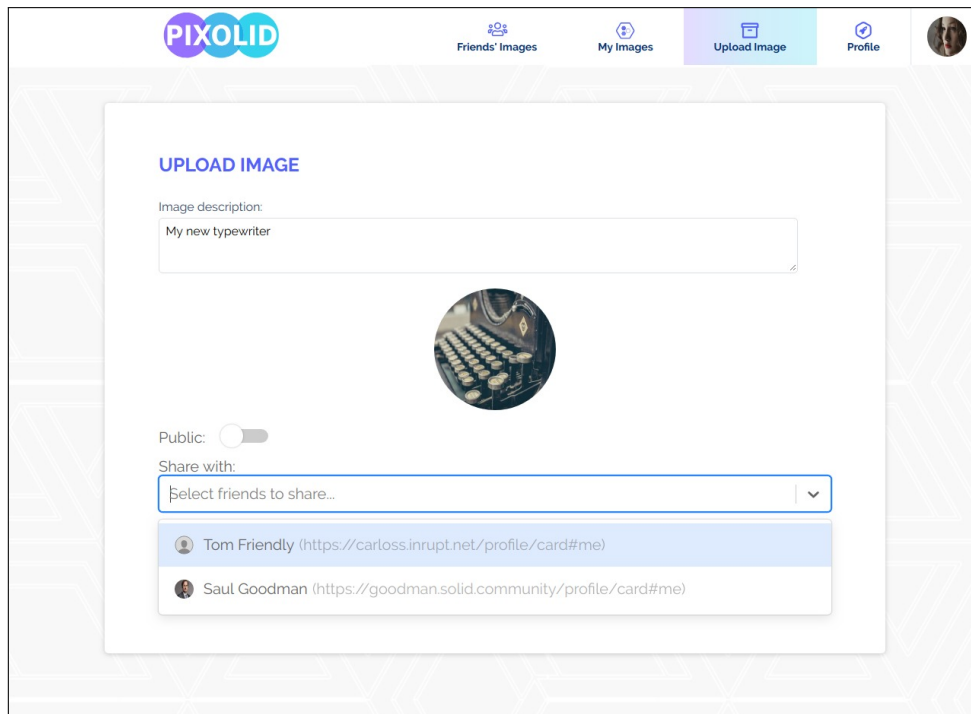


Figure 4.12: The user picks multiple friends (or none) to share the image with.



#### 4.4. Final Version of the Pixolid Application

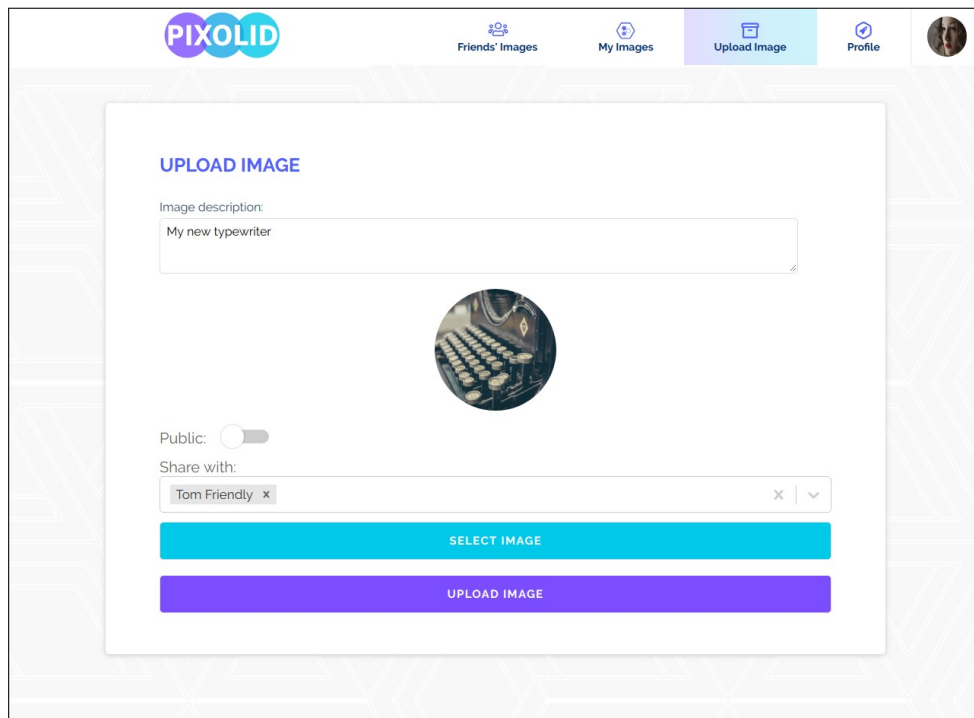


Figure 4.13: After the selection, selected users are shown for sharing.

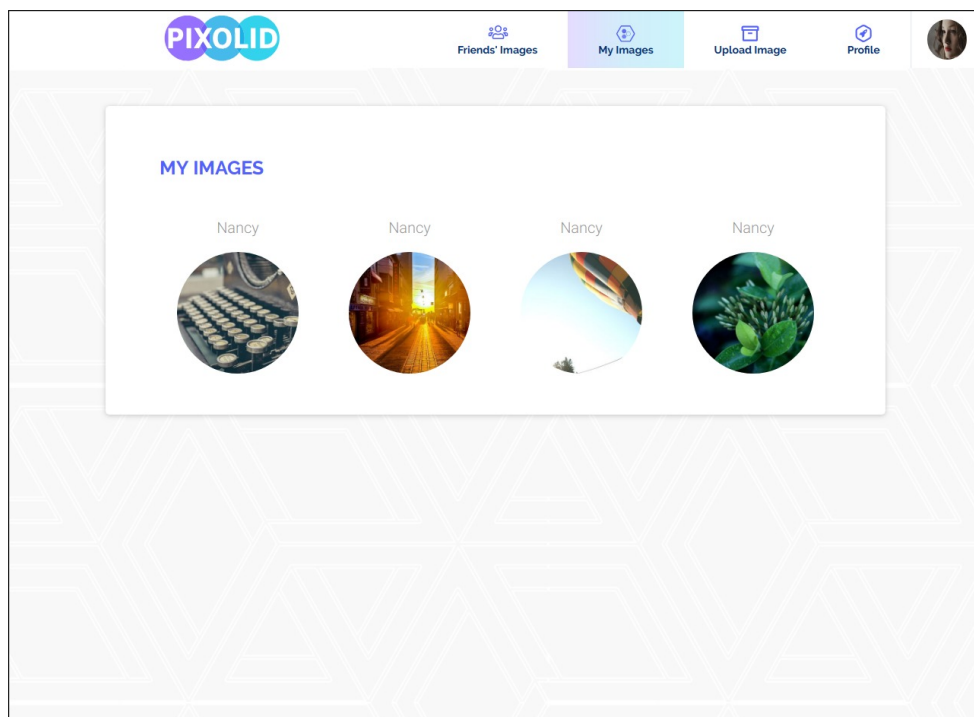


Figure 4.14: An overview of the images posted by the user.

#### 4. REALIZATION

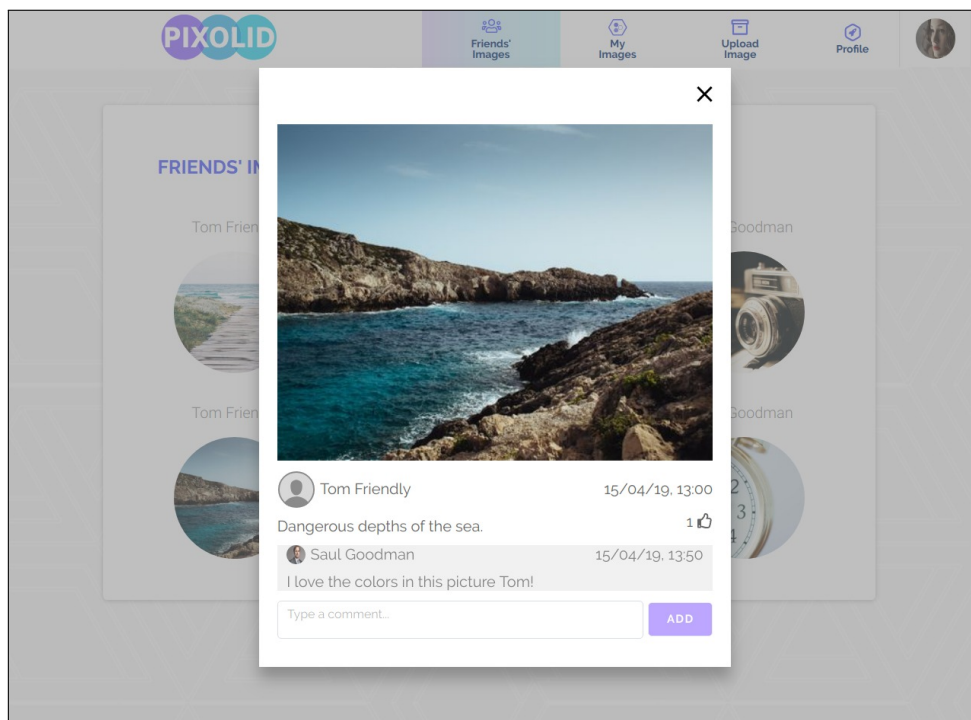


Figure 4.15: After the click on any image, the image detail is shown.

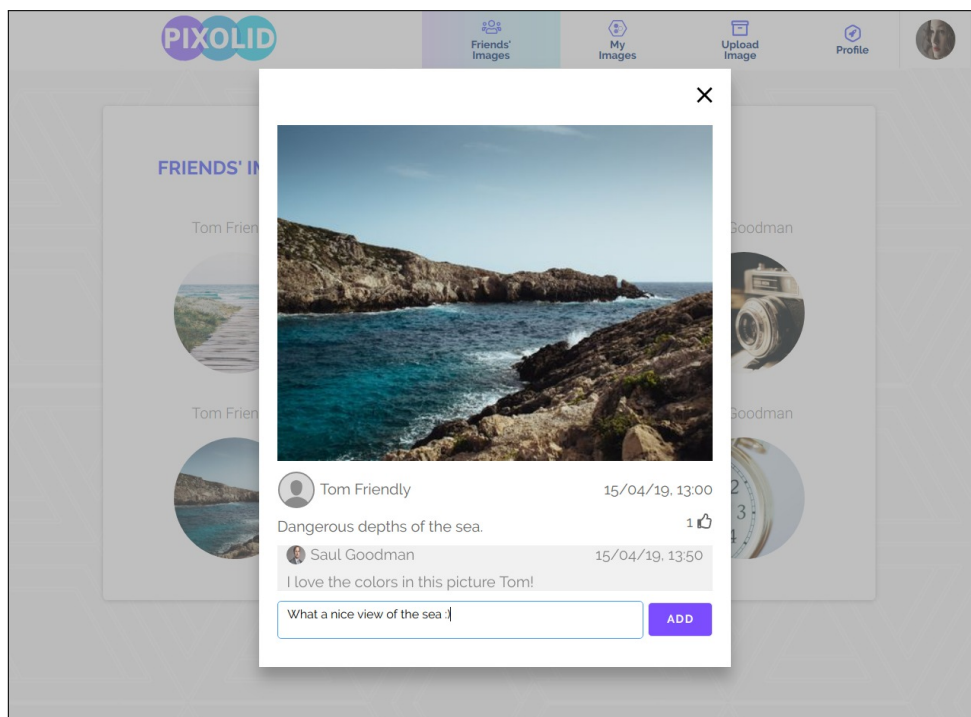


Figure 4.16: The user can type in a comment related to the picture.

#### 4.4. Final Version of the Pixolid Application

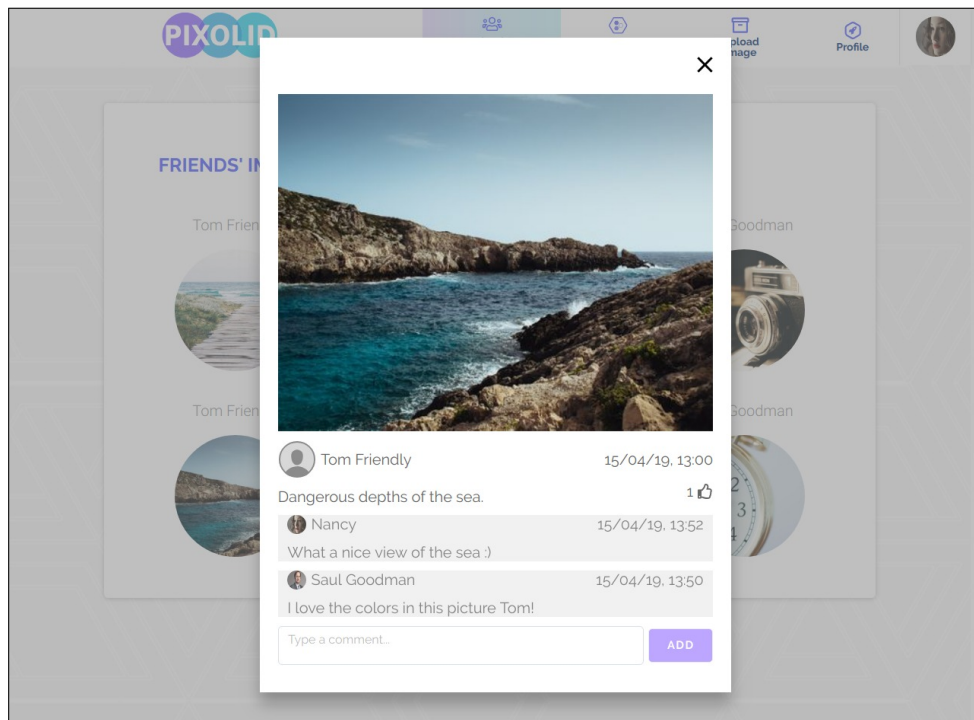


Figure 4.17: The comment is added by clicking on the *Add comment* button.

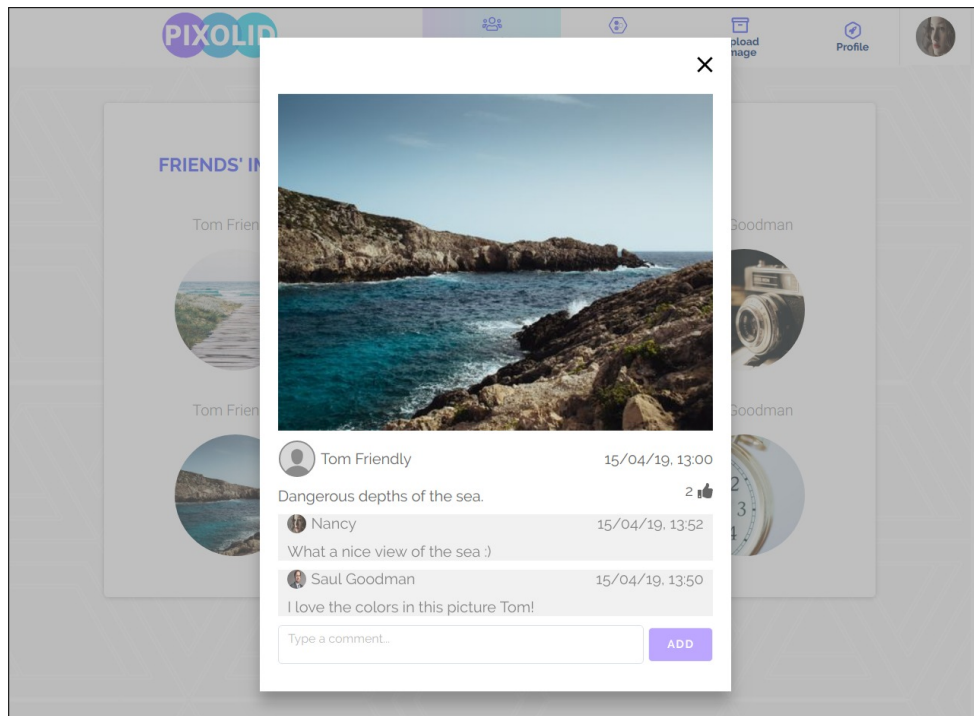


Figure 4.18: The like has been added by clicking on the like button.

## 4. REALIZATION

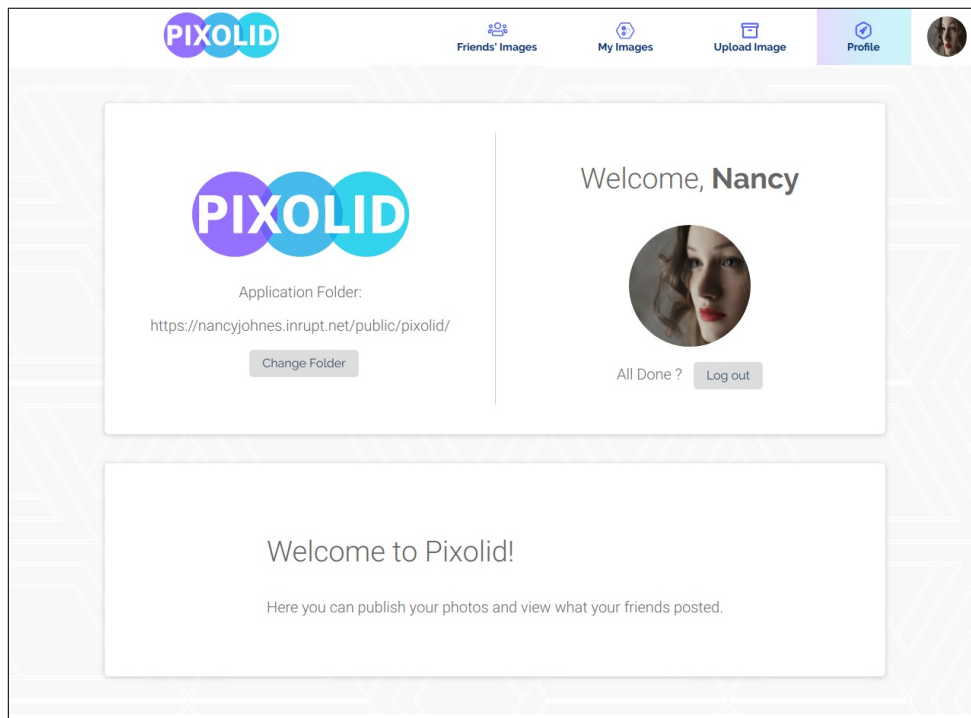


Figure 4.19: The profile shows info about the user, and the logout button.

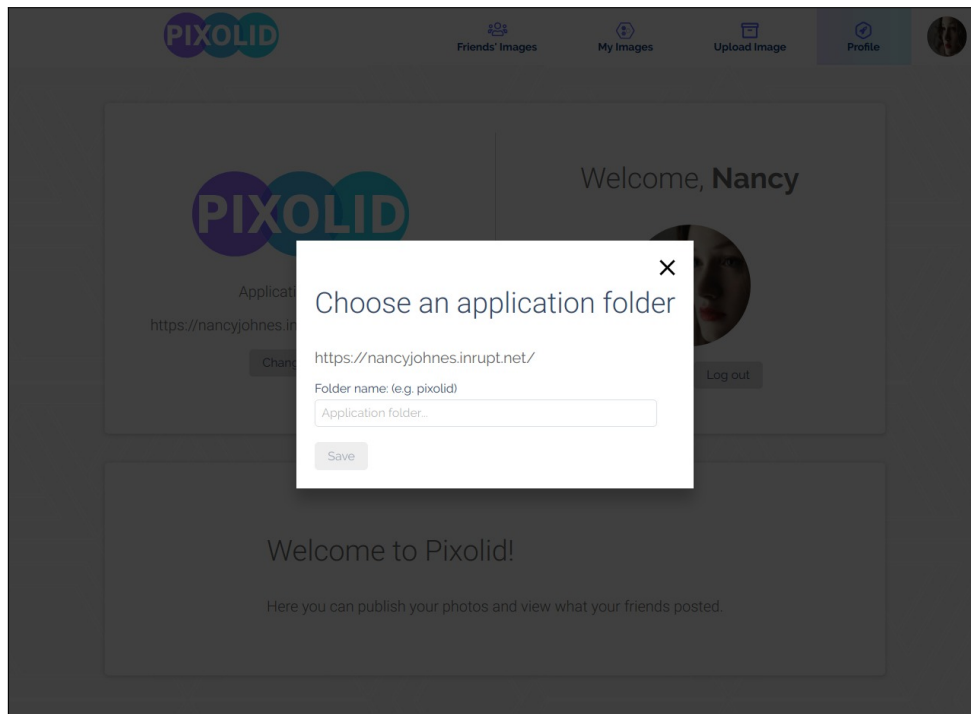


Figure 4.20: Application folder change after *change folder* button click.

---

# Testing

This chapter describes the used methods of testing the application. Firstly, testing scenarios for alpha testing are defined. Secondly, the process of creating and executing automated tests is discussed.

## 5.1 Testing Scenarios

In this section, we introduce testing scenarios which are meant to be used in alpha testing by testers. These scenarios were used to ensure the correct expected functionality of the application.

### 5.1.1 Log In

The *login* testing scenario checks whether the user can successfully log into the application.

#### 5.1.1.1 Prerequisites

- A Web browser (Chrome, Firefox, . . .) launched.

#### 5.1.1.2 Steps

1. Open a following website: `http://localhost:3000`.
2. Click on the *Select Provider ID* button, click on the *Inrupt* item.
3. Click on the *Login* button.
4. Ensure that the application gets redirected towards the following website which starts with: `https://inrupt.net/login?...`
5. Double click the username text area and select the shown pre-entered username.

6. Click on the *Log In* button.

### 5.1.1.3 Desired Outcome

- Redirect towards: `http://localhost:3000/friends/images`.
- The *friends' images* page is displayed.

## 5.1.2 Choose an Application Folder

This testing scenario checks whether the user can successfully choose an application folder.

### 5.1.2.1 Prerequisites

- Being logged in for the first time as in 5.1.1.

### 5.1.2.2 Steps

1. Ensure that the *choose an application folder* modal appears.
2. Click into the *folder name* text area, and type *public/pixolid*.
3. Click on the *Save* button.

### 5.1.2.3 Desired Outcome

- The modal disappears.
- The current page is: `http://localhost:3000/friends/images`.

## 5.1.3 Upload an Image

This testing scenario checks whether the user can successfully upload a new image.

### 5.1.3.1 Prerequisites

- Set application folder as in 5.1.2.

### 5.1.3.2 Steps

1. Click on the *Upload Image* tab button.
2. Ensure that you get redirected towards a following website: `http://localhost:3000/user/upload`.
3. Click into the *image description* text area, and type *my dog*.

4. Click on the *select image* button.
5. Select a *dog.png* image, and click on the *Open* button.
6. Ensure that the selected image is shown on the page.
7. Click on the *upload image* button.
8. Ensure that the *image uploaded* notification occurs.
9. Click on the *my images* tab button.
10. Ensure that you get redirected towards a following website: `http://localhost:3000/user/images`.

#### 5.1.3.3 Desired Outcome

- The uploaded image is among the previews of uploaded images.

#### 5.1.4 View User's Images

This testing scenario checks whether the user can successfully view his uploaded images.

##### 5.1.4.1 Prerequisites

- Set application folder as in 5.1.2.

##### 5.1.4.2 Steps

1. Click on the *my images* tab button.

##### 5.1.4.3 Desired Outcome

- Redirect towards: `http://localhost:3000/user/images`.
- Previews of the pre-posted user's images are shown.

#### 5.1.5 View Friends' Images

This testing scenario checks whether the user can successfully view his friends' uploaded images.

##### 5.1.5.1 Prerequisites

- Set application folder as in 5.1.2.

### 5.1.5.2 Steps

1. Click on the *friends' images* tab button.

### 5.1.5.3 Desired Outcome

- Redirect towards: `http://localhost:3000/friends/images`.
- Previews of the pre-posted friends' images are shown.

## 5.1.6 View an Image Detail

This testing scenario checks whether the user can successfully view an image detail.

### 5.1.6.1 Prerequisites

- The *my images* page shown as in 5.1.4.

### 5.1.6.2 Steps

1. Click on the first listed preview of the image.
2. Click on the *X* button.

### 5.1.6.3 Desired Outcome

- An *image preview* modal is shown.
- An image description is shown with the following text: *my dog*.
- A *like count* is shown with the count of 0.
- A comment section is shown with no comments.
- The *image detail* modal is closed.

## 5.1.7 Give a Like

This testing scenario checks whether the user can successfully give a like to the desired image.

### 5.1.7.1 Prerequisites

- *Image detail* modal is shown as in 5.1.6.

### 5.1.7.2 Steps

1. Click on the *Like* button.



### 5.1.7.3 Desired Outcome

- An *image liked* notification is shown.
- The *like count* has changed from 0 to 1.

## 5.1.8 Add a Comment

This testing scenario checks whether the user can successfully add a comment to the desired image.

### 5.1.8.1 Prerequisites

- *Image detail* modal is shown as in 5.1.6.

### 5.1.8.2 Steps

1. Click into the *type a comment* text area, and type *nice image*.
2. Click on the *add* button.

### 5.1.8.3 Desired Outcome

- A *comment added* notification is shown.
- The comment is added into the comments with the text *nice image*.

## 5.1.9 Log Out

The *log out* testing scenario checks whether the user can successfully log out of the application.

### 5.1.9.1 Prerequisites

- Set application folder as in 5.1.2.

### 5.1.9.2 Steps

1. Choose either a) or b) sub scenario:
  - a) Logout via profile.
    - i. Click on the *profile* tab button.
    - ii. Ensure that you get redirected towards a following website:  
`http://localhost:3000/user/profile`.
    - iii. Click on the *log out* button.
  - b) Logout via tab bar.
    - i. Hover over the *user profile image* tab button.
    - ii. Click on the *log out* button.

### 5.1.9.3 Desired Outcome

- Redirect towards: `http://localhost:3000/login`.
- The *login* page is displayed.

## 5.2 Automated Testing

In this section, the applied testing of the application is described. The application is equipped with an automated testing environment. For the purposes of testing, a Jest<sup>57</sup> testing framework is used. At first, we discuss frontend testing, followed by backend testing.

### 5.2.1 Frontend Testing

The frontend React application tests are also using Enzyme<sup>58</sup>, which is a JavaScript testing utility for React. This utility allows testing React components' output. Individual components have their own tests where it is assured that the components render correct output with desired styled components.

### 5.2.2 Backend Testing

The application is tested with the individual unit and integration tests. The tested functionality varies from creating new RDF structures to uploading new content to the Solid POD.

For the purposes of testing, testing data files (available in `pixolid/test/__mockData__`) were created. The files contain example profiles, images, comments, etc.

Some behavior of functions and dependencies is simulated with the use of testing mock objects. That mainly applies to network bound functions and operations. One of which includes the `rdflib.js`' fetcher and updater, as well as *solid-file-client*'s file and folder managing functions.

The testing data files are then returned by the mocked objects, according to the expected testing scenarios.

For detailed information about how to run tests in the interactive testing environment, please refer to Appendix B.

---

<sup>57</sup><https://jestjs.io/>

<sup>58</sup><https://github.com/airbnb/enzyme>

---

## Release and Feedback

In this chapter, we discuss the deployment process of the application along with releasing it to the public. At first, we comment on the availability of the source code, along with a description of deploying the application to a server. Then, we discuss the feedback of the Solid community concerning the application itself.

### 6.1 Release

The application's source code was released on the GitHub hosting service in a following public repository:

```
https://github.com/carloss8/pixolid/
```

The repository also mentions information about the process of installing, testing, building, and deploying the application.

### 6.2 Deployment

For detailed information about how to build and deploy the application, please refer to Appendix C.

The process of deploying the application had several complications in its way. At first, the application was meant to be deployed to GitHub Pages<sup>59</sup>, a service provided by GitHub to host Web pages which are released in a GitHub repository. The deployment to GitHub Pages was attempted by following the instructions mentioned in [37] and [38]. After doing so, the application was deployed, but the application's routing was non-functioning. That was due to the use of client-side routing by the application's *BrowserRouter* React

---

<sup>59</sup><https://pages.github.com/>

component, which is problematic while deploying to GitHub Pages. The proposed solutions mentioned in [38] did not solve the problem. After further research, there were other proposed solutions to set a base domain directly to the *BrowserRouter* as a component *prop*, but none of these solutions worked either.

Despite many efforts to fix this issue, in the end, the application was deployed to Netlify<sup>60</sup> hosting service. Netlify is hooked to the repository mentioned in section 6.1. Every push to the *master* branch of the repository causes the application to build and deploy to Netlify. The client-side routing problem is handled here by setting the following redirect in a *netlify.toml* configuration file:

```
1  [[redirects]]
2    from = "/*"
3    to = "/index.html"
4    status = 200
```

This behavior essentially redirects all routes to the *index.html* file, from which then the application correctly handles the client-side routing itself.

As of writing this thesis, the application is deployed and available live at:

<https://pixolid.netlify.com/>

For further information about using the application, please refer to Appendix D.

### 6.3 Feedback

The application was introduced and brought up to the Solid community's attention by posting about it on the Solid forum<sup>61</sup>.

#### 6.3.1 Thread and Repository Statistics

As of writing this thesis, the thread itself was viewed 582 times and liked 51 times. There were 36 replies in total, out of which 24 were from other people than the creator of the thread.

Also, eight different people commented on the application and welcomed the efforts it brings to the community. In addition to that, five people stated that they tried the application either locally or remotely (deployed on the server) with their Solid PODs, and tested uploading new images, comments, likes, as well as viewing other people's images, comments, and likes.

---

<sup>60</sup><https://www.netlify.com/>

<sup>61</sup><https://forum.solidproject.org/>

There were also 70 registered clicks on the link to the live deployed version of the application. The link to the GitHub repository containing the source code of the application registered 31 clicks (15 before, and 16 after editing the post). The repository itself was favorited by seven other developers and was forked two times.

### 6.3.2 Comments and Suggestions

Several community members have warmly welcomed the idea and the software itself. They started to try out and explore the application by using it with their Solid PODs. They started to upload pictures, comments, and likes via the application while providing their valuable feedback and opinions on the application.

Most of the people who commented on the application had positive reactions. They appreciated the idea and thought that it is a good work. For example, among some of the replies was a comment from Justin Bingham: *"Looking good! Nice use of the solid-react-sdk."* [39]. There was another one from James Martin: *"This is really cool! Fired it up locally and uploaded a few pictures. I like how cleanly it is to follow the data in the pod. Time to test "friends" photos! Thanks for sharing this."* [40]. Then from Stain Vr ale: *"Hi, just tested your app and it is really nice! Keep up the good work, we need more apps like this."* [41]. And also Jules wrote: *"Great first app! and a great learning resource to help others like me trying to learn more about Solid."* [42]. Also, it was proposed by Margaret Warren, if it would be possible in some way to utilize the metadata created with ImageSnippets<sup>62</sup> within Pixolid [43]. The potential utilizing and connecting the said metadata within Pixolid could be a part of future work.

Before the application was deployed on a server, it was asked if it would be possible to release it live and running on a server [44]. In the end, the application was deployed to *Netlify*.

It was also discovered that the application's origin URL might need to be manually added to the user's profile document as a *trustedApp* for certain Solid POD providers [44]. As of writing this thesis, this behavior was disabled altogether on Inrupt and Solid Community PODs, and there are further ongoing works in this regard for users to somehow authorize applications directly from the POD itself [44].

It was suggested that it would be great if it was possible to control whether or not the user wants to link back to comments or likes from other people [44]. This behavior could be added either as an additional step in the data distribution or as a retrospective measure through a moderating tool to disable showing certain comments or likes under user's images. A minor clean up of the data model was also suggested, as it would be better not to use file

---

<sup>62</sup><http://www.imagesnippets.com/>

extension suffixes in URIs and was proposed to use nodes with hashes (e.g., #like) to denote the like statement [44]. That could be achieved by changing the naming of created files.

It was also proposed that it would be a good practice to use a preferred storage location from the user's profile document as a base for specifying the application folder [44]. That could be achieved by retrieving and using this information from the user's profile. Along with that, on the application folder selection screen, it would be better to make it clear from the UI that a subdirectory structure is allowed in the name of the application folder [44]. This could be for example achieved by adding an additional tooltip stating this fact. It was also suggested that it could be more clear as to where the images are being stored after the upload [44]. That could be done by stating the location of the image in a notification after the upload, or by adding additional info about the directories on the *profile* page.

### 6.3.3 Feedback Summary

Overall, the application was warmly welcomed and appreciated by the community, as well as provided with the constructive suggestions to how it could be improved. The suggested improvements and possible extensions could be further worked on and are taken into consideration for future work.

---

# Conclusion

The goal of this thesis was to design, implement, and test a Solid based application supporting uploading and viewing images stored on the Solid POD, which would include user authentication using WebID-TLS and WebID-OIDC.

After the research of state-of-the-art technology, we discussed the analysis of the photo management application, along with its requirements and functionality. With the usage of the defined requirements, we constructed individual use cases with their use case scenarios. The important application's entities were identified and captured in the domain model. Based on the analysis, we continued with designing the architecture, user interface, along with wireframes, and branding of the application. Application's key classes were defined and visualized in the class diagram. Afterwards, we covered the application's development process, along with used technologies, and implementation focuses. We also showcased the look of the final version of the application. Later, we described the process of testing the application, covering the automated testing, along with the alpha testing scenarios. Finally, we discussed the release and deploy of the application. We also addressed the positive feedback and suggestions received by the Solid developer community.

The output of the thesis is a working Web application supporting re-decentralization of the Web. The application is capable of authenticating via WebID-OIDC and WebID-TLS, uploading and viewing user's and user's friends' images, comments, and likes stored on multiple Solid PODs. The thesis' output fulfills all of the requirements defined in the assignment of the thesis and its analysis. The application is published as open-source, hosted on a public repository.

The work was welcomed and appreciated by the community, as it supports the development of new Solid based applications. In the future, the application can be further extended to support additional functionality. There is for example room for editing and moderating content, further expanding access rights to comments and likes, or integrating with other Solid based applications.





---

# Bibliography

- [1] INTERNET USAGE STATISTICS. [online], December 2018, [cit. 2018-12-11]. Available from: <https://www.internetworldstats.com/stats.htm>
- [2] Bieg, M. *Server-based-network*. Wikimedia Foundation, Jan 2007, [cit. 2019-2-6]. Available from: <https://en.wikipedia.org/wiki/File:Server-based-network.svg>
- [3] Bieg, M. *P2P-network*. Wikimedia Foundation, Aug 2007, [cit. 2019-2-6]. Available from: <https://en.wikipedia.org/wiki/File:P2P-network.svg>
- [4] Cohen, B. Incentives Build Robustness in BitTorrent. [online], May 2003, [cit. 2019-2-6]. Available from: <http://bittorrent.org/bittorrentecon.pdf>
- [5] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. [online], Oct 2008, [cit. 2019-2-6]. Available from: <https://bitcoin.org/bitcoin.pdf>
- [6] McCoy, D.; Bauer, K.; et al. Shining Light in Dark Places: Understanding the Tor Network. [online], Jul 2008, [cit. 2019-2-6]. Available from: <http://www.cs.umd.edu/class/spring2017/cmsc8180/papers/understanding-tor.pdf>
- [7] Farmer, C. Five projects that are decentralizing the web in slightly different ways. [online], September 2018, [cit. 2019-1-19]. Available from: <https://medium.com/textileio/five-projects-that-are-decentralizing-the-web-in-slightly-different-ways-debf0fda286a>
- [8] How does diaspora\* work? [online], 2018, [cit. 2019-1-19]. Available from: <https://diasporafoundation.org/about>

## BIBLIOGRAPHY

---

- [9] Web inventor Tim Berners-Lee's next project: a platform that gives users control of their data. [online], November 2015, [cit. 2018-11-26]. Available from: <https://www.csail.mit.edu/news/web-inventor-tim-berners-lees-next-project-platform-gives-users-control-their-data>
- [10] Berners-Lee, S. T. One Small Step for the Web... [online], September 2018, [cit. 2018-11-26]. Available from: <https://www.inrupt.com/blog/one-small-step-for-the-web>
- [11] Solid. [online], October 2018, [cit. 2018-11-26]. Available from: <https://solid.mit.edu/>
- [12] How it works. [online], September 2018, [cit. 2018-11-26]. Available from: <https://solid.inrupt.com/how-it-works>
- [13] nicola; dmitrizagidulin; et al. Solid. [online], November 2018, [cit. 2018-11-26]. Available from: <https://github.com/solid/solid>
- [14] Mansour, E.; Sambra, A. V.; et al. A Demonstration of the Solid Platform for Social Web Applications. [online], May 2016, [cit. 2018-11-26]. Available from: <http://ds.qcri.org/publications/2016-mansour-www.pdf>
- [15] Capadislis, S.; Guy, A. Linked Data Notifications. [online], May 2017, [cit. 2018-11-26]. Available from: <https://www.w3.org/TR/ldn/>
- [16] nicola; deiu; et al. solid-apps. [online], September 2015, [cit. 2018-11-26]. Available from: <https://github.com/solid/solid-apps/>
- [17] Semantic Web. [online], 2015, [cit. 2018-12-11]. Available from: <https://www.w3.org/standards/semanticweb/>
- [18] Berners-Lee, T. Linked Data. [online], July 2006, [cit. 2018-12-11]. Available from: <https://www.w3.org/DesignIssues/LinkedData.html>
- [19] McCrae, J. P.; Abele, A.; et al. The Linked Open Data Cloud. [online], November 2018, [cit. 2018-12-11]. Available from: <https://lod-cloud.net/>
- [20] Berners-Lee, T.; Fielding, R. T.; et al. Uniform Resource Identifier (URI): Generic Syntax. [online], January 2005, [cit. 2018-12-11]. Available from: <https://tools.ietf.org/html/rfc3986>
- [21] Keil, J. M. IRI, URI, URL, URN and their differences. [online], November 2016, [cit. 2019-1-26]. Available from: <http://fusion.cs.uni-jena.de/fusion/blog/2016/11/18/iri-uri-url-urn-and-their-differences/>

- 
- [22] Richard Cyganiak, M. L., David Wood. RDF 1.1 Concepts and Abstract Syntax. [online], February 2014, [cit. 2019-1-10]. Available from: <https://www.w3.org/TR/rdf11-concepts/>
- [23] About. [online], 2018, [cit. 2018-12-20]. Available from: <https://wiki.dbpedia.org/about>
- [24] Nlany; LionKimbrow; et al. RdfSyntax. [online], January 2011, [cit. 2019-1-30]. Available from: <https://www.w3.org/wiki/RdfSyntax>
- [25] Schreiber, G.; Raimond, Y. RDF 1.1 Primer. [online], June 2014, [cit. 2019-2-3]. Available from: <https://www.w3.org/TR/rdf11-primer/>
- [26] Beckett, D.; Berners-Lee, T.; et al. RDF 1.1 Turtle. [online], February 2014, [cit. 2019-1-10]. Available from: <https://www.w3.org/TR/turtle/>
- [27] Sambra, A.; Story, H.; et al. WebID 1.0. [online], March 2014, [cit. 2019-2-11]. Available from: <https://www.w3.org/2005/Incubator/webid/spec/identity/>
- [28] Inkster, T.; Story, H.; et al. WebID-TLS. [online], March 2014, [cit. 2019-2-14]. Available from: <https://www.w3.org/2005/Incubator/webid/spec/tls/>
- [29] Zagidulin, D.; Kjernsmo, K.; et al. WebID-OIDC Authentication Spec. [online], April 2017, [cit. 2019-2-16]. Available from: <https://github.com/solid/webid-oidc-spec>
- [30] Inrupt Design. [online], [cit. 2019-3-15]. Available from: <https://design.inrupt.com/>
- [31] Team, L. D. rdflib.js. [online], February 2019, [cit. 2019-3-7]. Available from: <https://linkeddata.github.io/rdflib.js/doc/>
- [32] Verborgh, R. solid-file-client. [online], September 2018, [cit. 2019-3-7]. Available from: <https://github.com/solid/query-ldflex>
- [33] Zucker, J. solid-file-client. [online], November 2018, [cit. 2019-3-7]. Available from: <https://github.com/jeff-zucker/solid-file-client>
- [34] WebAccessControl. [online], June 2018, [cit. 2019-3-15]. Available from: <https://www.w3.org/wiki/WebAccessControl>
- [35] Zagidulin, D.; Kjernsmo, K.; et al. Web Access Control (WAC). [online], April 2016, [cit. 2019-3-15]. Available from: <https://github.com/solid/web-access-control-spec>

## BIBLIOGRAPHY

---

- [36] Bingham, J.; Martin, J.; et al. Solid React SDK. [online], January 2019, [cit. 2019-3-23]. Available from: <https://github.com/inrupt-inc/solid-react-sdk/>
- [37] gitname. Deploying a React App\* to GitHub Pages. [online], June 2017, [cit. 2019-4-12]. Available from: <https://github.com/gitname/react-gh-pages/>
- [38] Cherouvim, I. Deployment. [online], February 2019, [cit. 2019-4-12]. Available from: <https://facebook.github.io/create-react-app/docs/deployment>
- [39] Bingham, J. Pixolid - Solid photo manager. [online forum comment], April 2019, [cit. 2019-4-30]. Available from: <https://forum.solidproject.org/t/pixolid-solid-photo-manager/1649/10>
- [40] Martin, J. Pixolid - Solid photo manager. [online forum comment], April 2019, [cit. 2019-4-30]. Available from: <https://forum.solidproject.org/t/pixolid-solid-photo-manager/1649/11>
- [41] Vråle, S. Pixolid - Solid photo manager. [online forum comment], April 2019, [cit. 2019-4-30]. Available from: <https://forum.solidproject.org/t/pixolid-solid-photo-manager/1649/35>
- [42] Jules. Pixolid - Solid photo manager. [online forum comment], April 2019, [cit. 2019-4-30]. Available from: <https://forum.solidproject.org/t/pixolid-solid-photo-manager/1649/36>
- [43] Warren, M. Pixolid - Solid photo manager. [online forum comment], April 2019, [cit. 2019-4-30]. Available from: <https://forum.solidproject.org/t/pixolid-solid-photo-manager/1649/29>
- [44] Veltens, A.; Bingham, J.; et al. Pixolid - Solid photo manager. [online forum thread], April 2019, [cit. 2019-4-30]. Available from: <https://forum.solidproject.org/t/pixolid-solid-photo-manager/1649>

---

# Acronyms

**API** Application Programming Interface.

**ASCII** American Standard Code for Information Interchange.

**CSAIL** Computer Science and Artificial Intelligence Lab.

**CSS** Cascading Style Sheets.

**DIG** Decentralized Information Group.

**FOAF** Friend of a Friend.

**HOC** Higher Order Component.

**HTML** Hypertext Markup Language.

**HTML5** Hypertext Markup Language 5.

**HTTP** Hypertext Transfer Protocol.

**ID Token** Identity Token.

**IP** Internet Protocol.

**IRI** Internationalized Resource Identifier.

**JPEG** Joint Photographic Experts Group.

**JSON** Javascript Object Notation.

**JSON-LD** A JSON Serialization for Linked Data.

**LDflex** Linked Data flex.

**LDP** Linked Data Platform.

**LOD** Linked Open Data.

**MIT** Massachusetts Institute of Technology.

**N3** Notation3.

**OAuth** Open Authentication.

**OWL** Web Ontology Language.

**Pixolid** Pictures on Solid.

**PNG** Portable Network Graphics.

**POD** Personal Online Datastore.

**RDF** Resource Description Framework.

**RDFa** Rich Structured Data Markup for Web Documents.

**rdflib.js** Javascript RDF library for browsers and Node.js.

**REST** Representation State Transfer.

**SDK** Software Development Kit.

**SKOS** Simple Knowledge Organization System.

**Solid** Social Linked Data.

**SPARQL** SPARQL Protocol and RDF Query Language.

**TCP** Transmission Control Protocol.

**TLS** Transport Layer Security.

**Turtle** Terse RDF Triple Language.

**UI** User Interface.

**URI** Uniform Resource Identifier.

**URL** Uniform Resource Locator.

**URN** Uniform Resource Name.

**W3C** World Wide Web Consortium.

**WebID** Web Identity and Discovery.

**WebID-OIDC** WebID – Open ID Connect.

**WebID-TLS** WebID – Transport Layer Security.

**XML** Extensible Markup Language.





---

## Developer Guide

This guide contains instructions for developers regarding the application's installation, running the application in the development mode, and running the tests. The application's structure was discussed in 4.2. Any changes to the code can be done in the corresponding files.

### B.1 Installation

The application's source code is available in the following repository:

```
https://github.com/carloss8/pixolid/
```

Open the terminal and clone the repository:

```
$ git clone https://github.com/carloss8/pixolid.git
```

Switch to the *pixolid* folder:

```
$ cd ./pixolid/
```

Install application's dependencies by:

```
$ npm install
```

or

```
$ yarn install
```

You have successfully installed the development version of the application.

## B.2 Running the Development Mode

To start the application in the development mode, run the following command:

```
$ npm start
```

---

or

```
$ yarn start
```

---

The application is now available at `http://localhost:3000` which can be viewed in the Web browser. As you make edits to the source code, the page will automatically reload in real time.

## B.3 Running Tests

To start the test runner in the interactive watch mode, run the following command:

```
$ npm test
```

---

or

```
$ yarn test
```

---

You can then see the test runner options. You can:

- Press `a` to run all tests.
- Press `f` to run only failed tests.
- Press `p` to filter by a filename regex pattern.
- Press `t` to filter by a test name regex pattern.
- Press `q` to quit watch mode.
- Press `Enter` to trigger a test run.

---

# Administrator Guide

This guide contains instructions for administrators regarding the application's installation, building, and deploying.

## C.1 Installation

The application's source code is available in the following repository:

```
https://github.com/carloss8/pixolid/
```

Open the terminal and clone the repository:

```
$ git clone https://github.com/carloss8/pixolid.git
```

Switch to the *pixolid* folder:

```
$ cd ./pixolid/
```

Install application's dependencies by:

```
$ npm install
```

or

```
$ yarn install
```

You have successfully installed the development version of the application.

## C.2 Building the Application

To create a production build of the application, run the following command:

---

```
$ npm run build
```

---

or

---

```
$ yarn build
```

---

The built application ready for production is available in the *build* folder.

## C.3 Deploying the Application

The deployment process varies by concrete deployment solutions. To specify a URL where the application will be deployed and served, set the *homepage* property in the *package.json*. Open *package.json* in a text editor of your choice. If the application was deployed to the following URL: `http://example.org`, you would set the *homepage* property to:

---

```
"homepage": "http://example.org"
```

---

To run the application on a static server, you can do so, for example with *serve*. You can pass the *build* folder to it by:

---

```
$ serve -s ./build/ -l 3000
```

---

The application is then served at: `http://localhost:3000`. In Figure C.1, we can see a deployment diagram for the application.

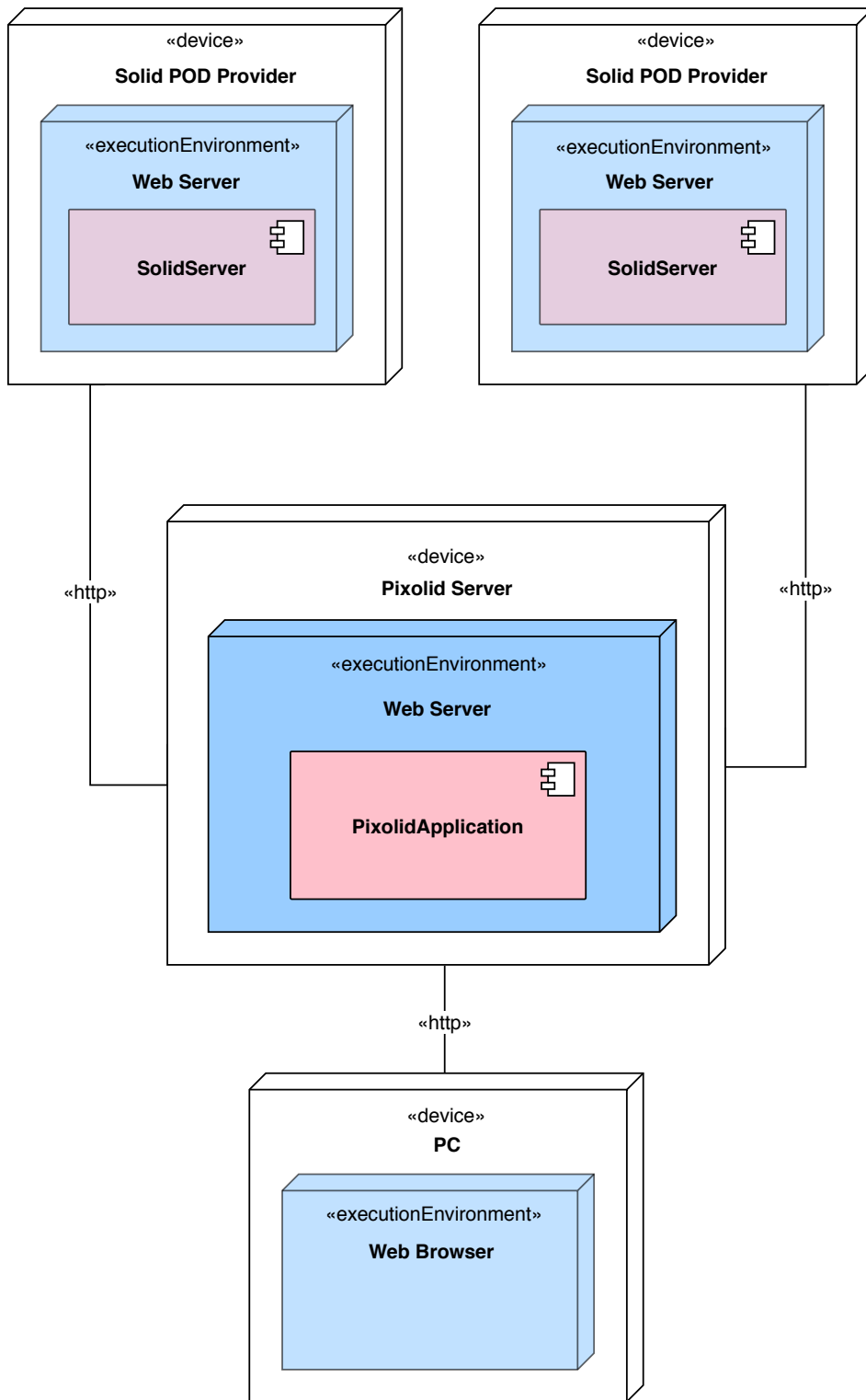


Figure C.1: Deployment Diagram: Relations among the devices.



---

## User Guide

This guide contains information about the application’s functionality, as well as instructions for users on how to start using the application.

### D.1 Functionality

Pictures on Solid (Pixolid) offers a new way to manage your images. All of your data is stored in your Solid POD. You can sign into the application with your existing WebID, or by creating a new one.

The application supports uploading images with descriptions to your Solid POD. You can also set access rights for newly uploaded images to either be public or private. In the private mode, it is possible to select multiple users (your friends gathered from your profile), which will be allowed to access and view the image via the application.

The application supports viewing your uploaded images. You can also view images posted by your friends (gathered from your profile) via the application. The application also supports showing the image detail. The detail shows the image with its description, author, and date of posting.

In the application, you can also post comments to the images, as well as like the images. Individual comments and likes are shown in the image detail.

Every image, comment, and like is stored in the Solid POD of the user who posted it. The application gathers all the content from the PODs of individual users.

### D.2 Start Using Pixolid

Assuming that the application is built, deployed, and running on the localhost, you can start using it by opening: `http://localhost:3000` in your Web browser. As of writing this thesis, you can also start using it by opening: `http://pixolid.netlify.com/` in your Web browser.

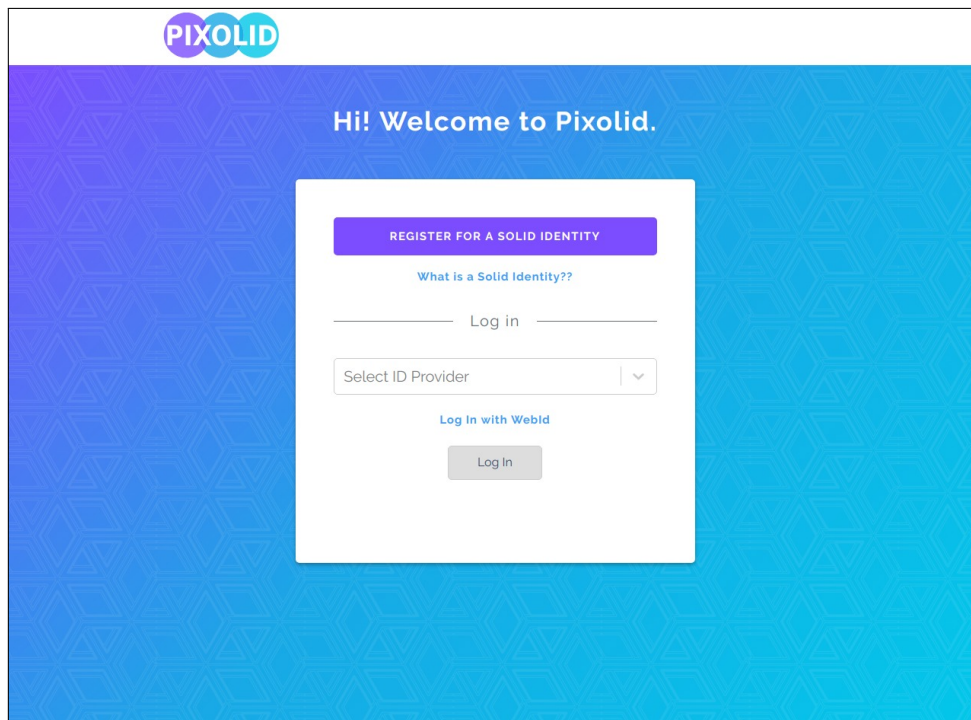


Figure D.1: Login screen.

## D.3 Individual Screens' Description

Let us cover what you can achieve on individual Pixolid's screens.

### D.3.1 Login Screen

On a *login* screen (see Figure D.1), you can log into Pixolid with a WebID. You can get the WebID from multiple providers, or you can directly start the registration process by clicking on the *register for a Solid identity* button. To use an existing WebID, you can either choose a provider from the *Select ID Provider* list menu, or click on the *Log In with WebID* link and paste your WebID into the *WebID* text area. Confirm logging in by clicking on the *Log In* button. You will get redirected towards the WebID provider login screen, where you shall enter your username and password. After authenticating with your provider you will get redirected back into Pixolid.

### D.3.2 Application Folder Selection Screen

After your first successful logging into Pixolid, you will see an *application folder selection* screen (see Figure D.2). Here you can specify the location, where Pixolid will store your images, comments, and likes on your Solid POD.



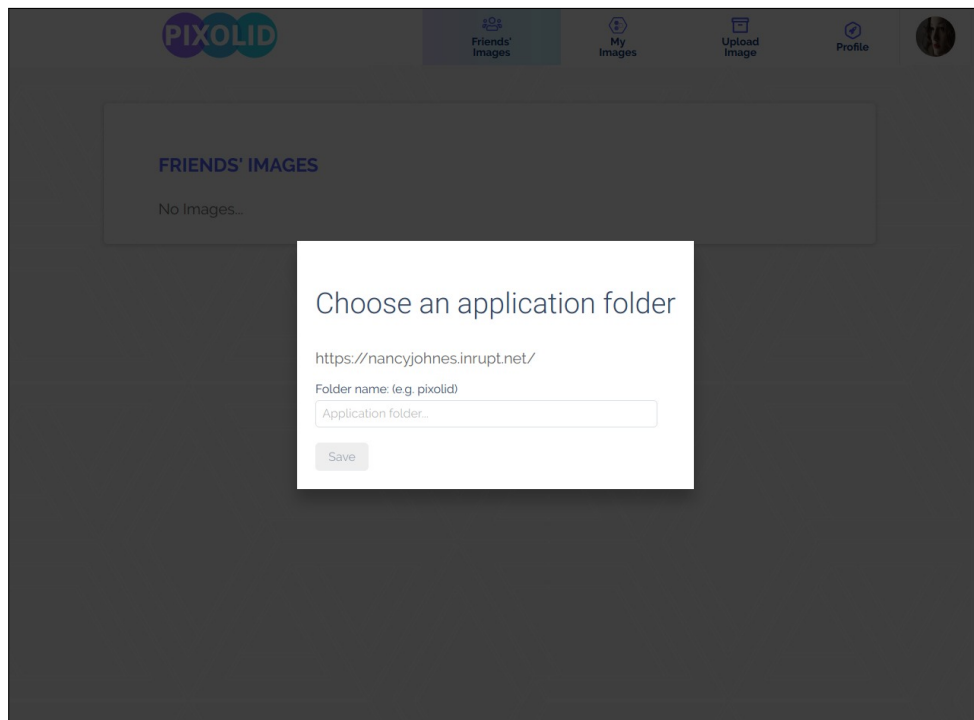


Figure D.2: Application folder selection screen.

You can do so by typing into the *application folder* text area. Folder names including subdirectories are supported, e.g., *public/pixolid* will create a *pixolid* folder inside the *public* folder. Confirm the folder selection by clicking on the *Save* button.

### D.3.3 Application Navigation

To navigate through the application, you can use the *tab bar*, which is visible on every screen next to the Pixolid logo. The logo itself and *friends' images* tab button take you to the *friends' images* page (see Figure D.4). *My images*, *upload image*, and *profile* tab buttons take you to *my images* (see Figure D.5), *upload image* (see Figure D.3), and *profile* (see Figure D.7) pages respectively. Click on any *tab button* to show the desired page. You can also hover your mouse over a *profile image* button (see Figure D.4), and log out of the application by clicking on the *Log Out*. Clicking on the *profile image* button itself will take you to the *profile* page.

### D.3.4 Upload Image Screen

On a *upload image* screen (see Figure D.3) you can upload a new image. Type its description into the *image description* text area. Select an image by

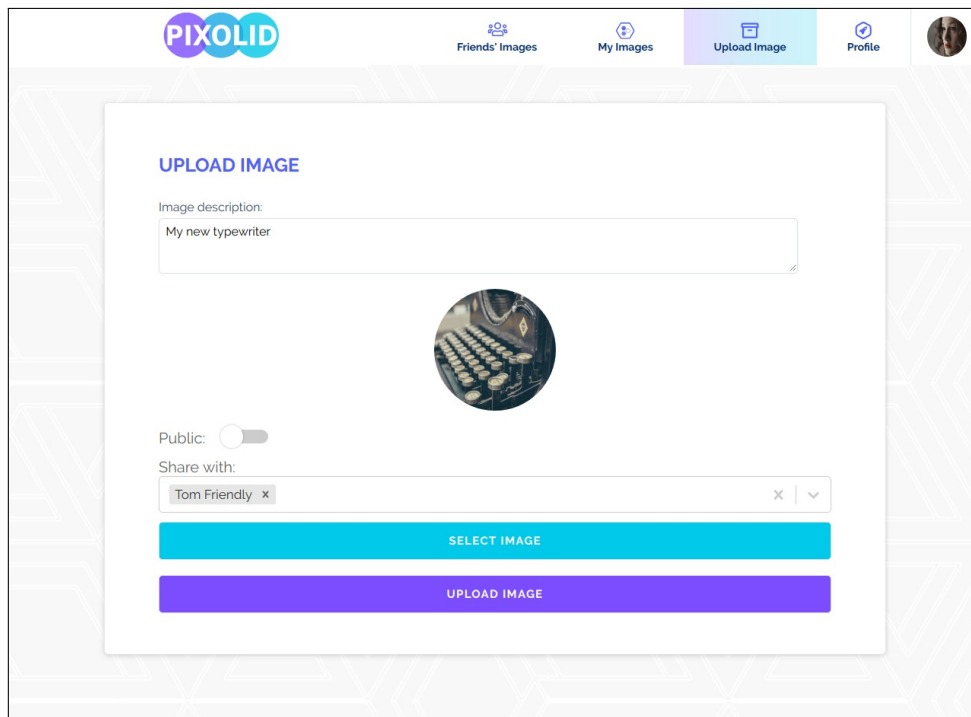


Figure D.3: Upload image screen.

clicking on the *select image* button. The image preview is shown on under the *image description* text area. Select sharing to public or private by clicking on the *Public* toggle. When the public sharing is toggled off, you can specify individual users who can access the image by selecting them from the *Share with* list menu. You can choose multiple users or none, which denotes that you want the image to be accessible only to you. Users can be removed from the list by clicking on the *X* button next to their names.

### D.3.5 Friends' Images and My Images Screens

On a *friends' images* screen (see Figure D.4) you can see images posted by your friends. Friends are discovered from your profile document tied to the WebID. On a *my images* screen (see Figure D.5) you can see your posted images. Images on both *friends' images* and *my images* screens are sorted from newest to oldest, and you can click on any image to see its detail.

### D.3.6 Image Detail Screen

On an *image detail* screen (see Figure D.6) you can see the image detail along with its description, author's name and profile picture, date of posting, number of likes, and comments with their authors. You can like the image as well by

### D.3. Individual Screens' Description

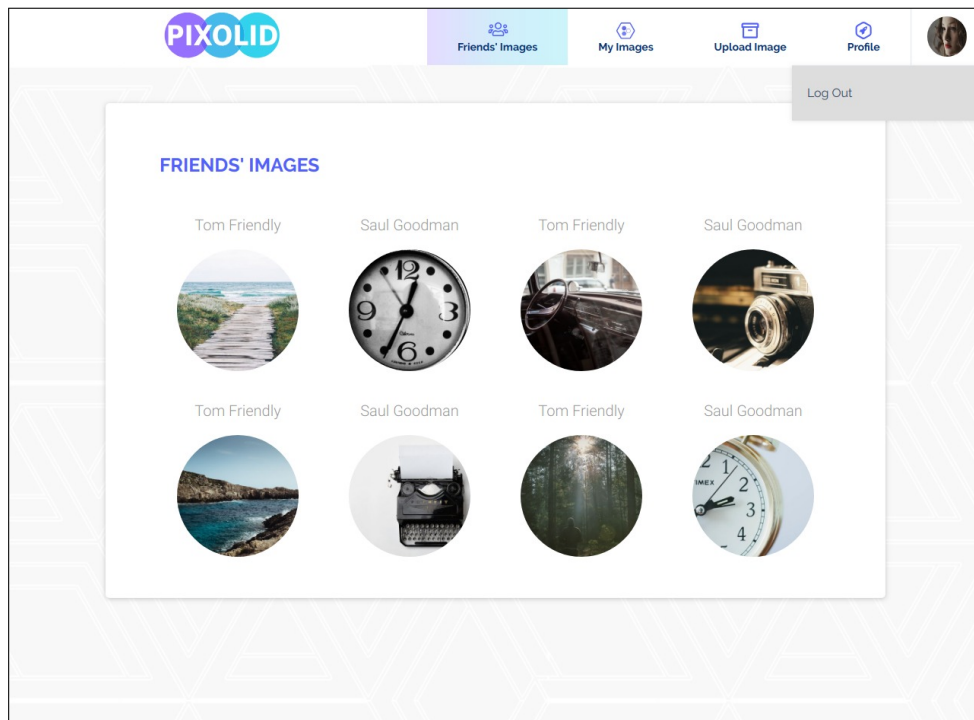


Figure D.4: Friend's images screen.

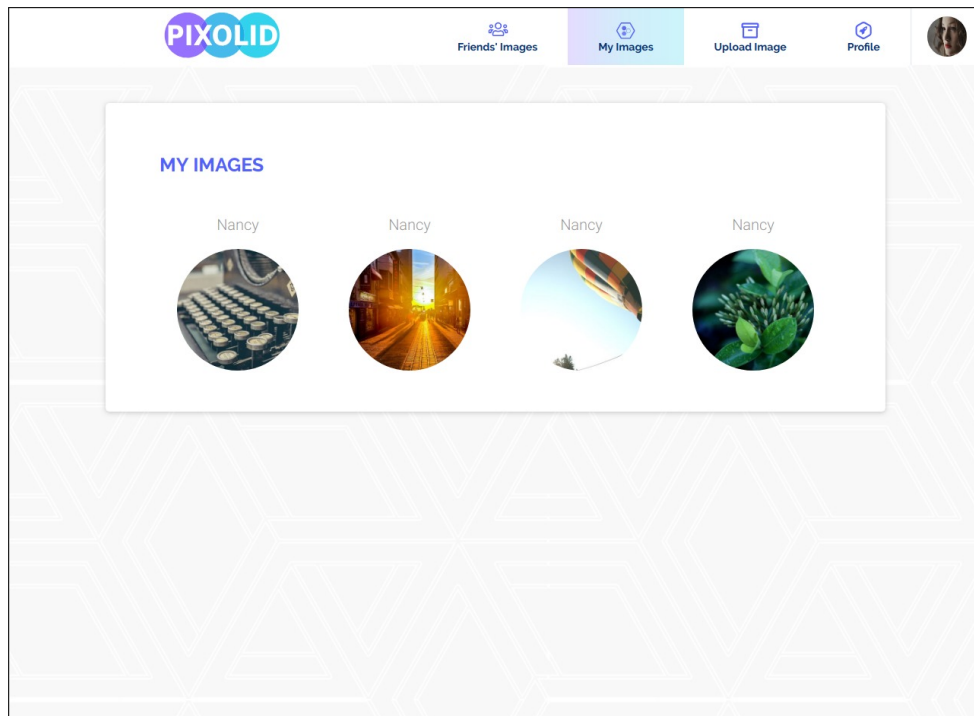


Figure D.5: User's images screen.

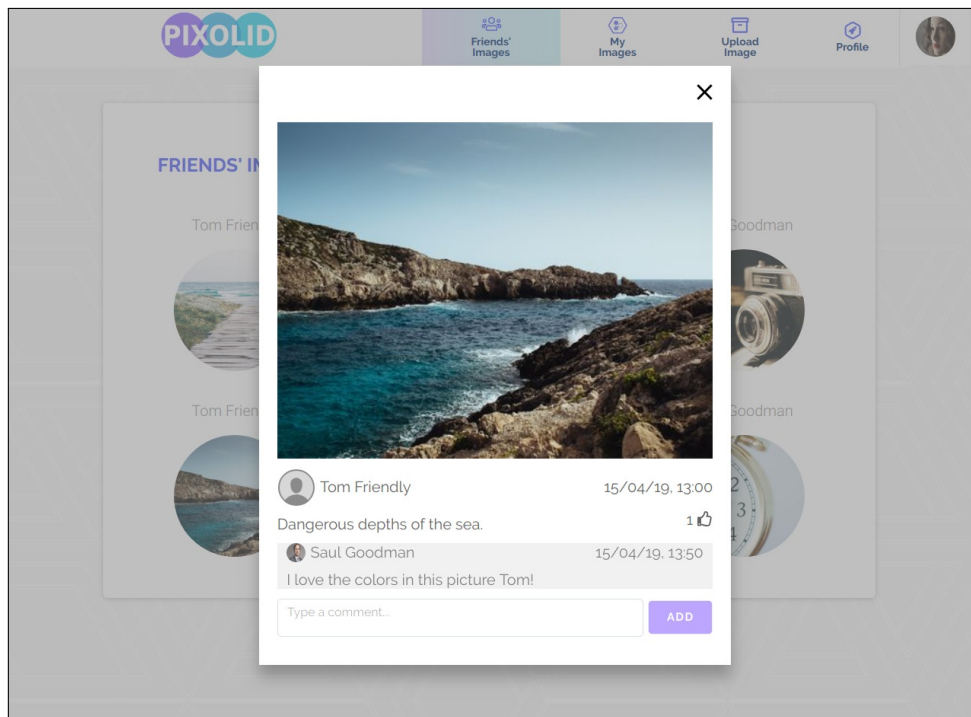


Figure D.6: Image detail screen.

clicking on the *Like* button with a thumb icon. You can type a comment into the *type a comment* text area, and add it by clicking on the *Add* button. You can close the image detail screen by clicking either on the *X* button or the gray area around the window.

### D.3.7 Profile Screen

On a *profile* screen (see Figure D.7) you can see information about the current application folder and your profile. You can change the application folder by clicking on the *Change Folder* button. The *application folder selection* screen (see Figure D.8) which is similar to the previous one. You can cancel changing the *application folder selection* by clicking either on the *X* button or the gray area around the window. You can also log out of the application by clicking on the *Log Out* button.

### D.3. Individual Screens' Description

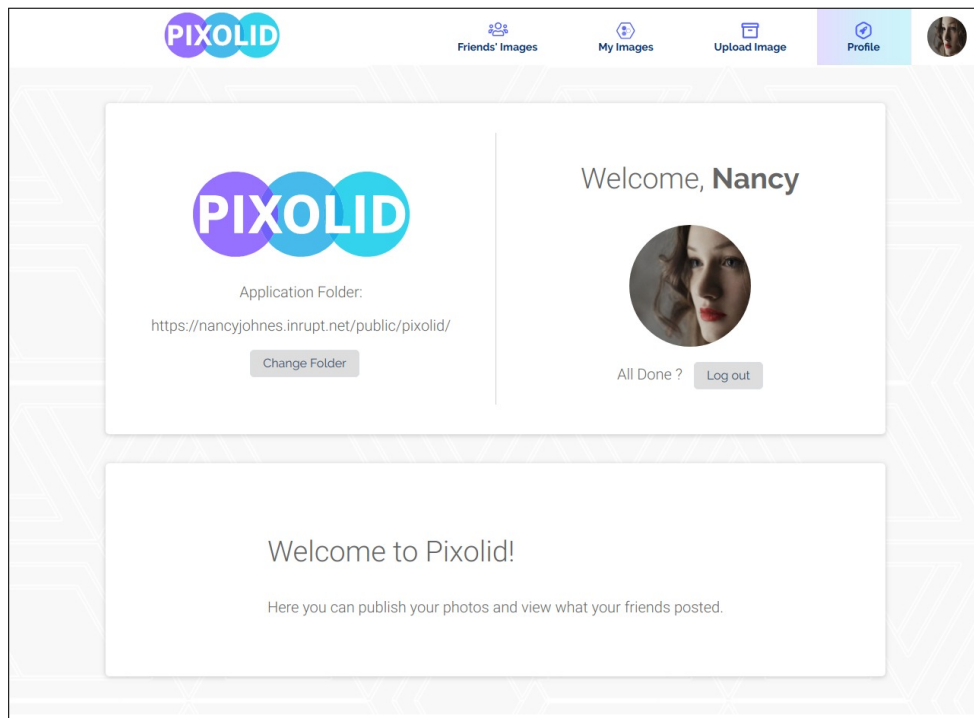


Figure D.7: User profile screen.

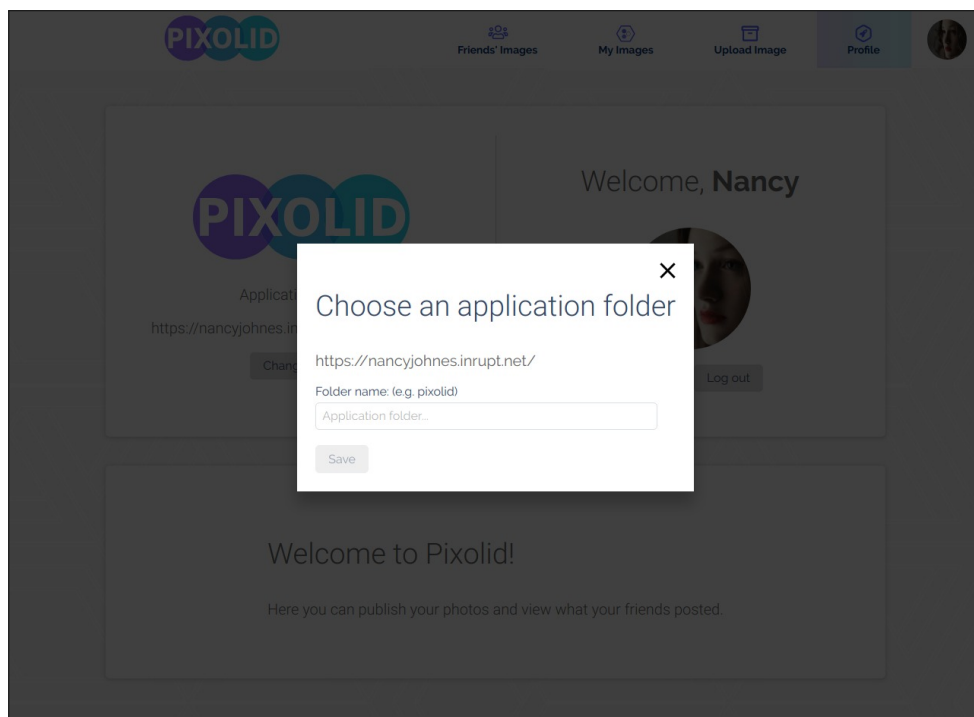


Figure D.8: Application folder change screen.



---

## Contents of enclosed CD

	readme.txt.....	the file with CD contents description
	exe .....	the directory with executables
	build.....	the directory with the build of the application
	src .....	the directory of source codes
	impl .....	the directory of source codes of the implementation
	thesis.....	the directory of $\text{\LaTeX}$ source codes of the thesis
	text .....	the thesis text directory
	thesis.pdf.....	the thesis text in PDF format