Solid: Privacy on the Web
NaviServer Integration Proposal

Petar Petrov
WU - Vienna, Austria

Abstract

The strive for online privacy has peaked during the recent years, most notably with the new EU regulations for GDPR. The regulations that have been imposed by the governments are a step in the right direction. Nevertheless something more needs to be done in terms of technological stack and patterns in order to leave the users greater freedom to choose the way their data is treated. There are multiple ideas and methodologies circulating in the internet and trying to make their way and become a standard, but very few of them have the actual potential of achieving it. However there is one framework, which is gaining pace – Solid (SOcial LInked Data). The main idea behind it is of Tim Berners-Lee, the creator of the World Wide Web. He along with a team from MIT and many other online contributors are working on this new approach, aiming to build decentralized applications with the help of Linked Data. They are building on the concept of personal data stores, where the information is kept only by the user. In contrast, this framework intends to make a division between personal files and files intended for public use, and leave the final decision to the user's preferences where to keep the data. In this paper I will give some background information, look into the different functionalities provided by Solid and will provide a solution to make it work with NaviServer.

Keywords: Solid, Privacy, Personal Data Store, Linked Data, Reverse Proxy, NaviServer

---

Contents

1. Motivation and Research Question

In order to have a competitive company nowadays, achieving data protection and privacy is one of the most important factors. The users as well as the regulatory organs of the governments are becoming more and more demanding and wish to restrict the access to private information by third parties. This leads to applying pressure on the websites and online services, and forces them to change the way they operate. Therefore we can clearly see, that our experience when entering a website, which wants to use our private data (i.e. cookies) is changing. We are now asked to give consent for our data to be used and have the option to change our preferences and give fewer information than we did before. There is an improvement on this aspect of the problem, but there is one fundamental thing, that is still left unchanged - the way the data stored and processed. Basically all of the social networks nowadays store all of their users' data on their own servers. Users are never asked or given any other option - if they want to be part of the social network, they agree to upload their photos and other personal information on the social networks' servers. This has led to many concerns and complaints in the recent days and needs to be changed. Indeed, as technologies evolve and now we are living in the world of microservices, it is necessary to adapt the way the data is treated as well. Furthermore, the users always like to have an option to choose and this will make them feel safer when surfing the web. There seems to be a development in terms of the personal data stores, but this is a rather restricted way and it will be hard to implement it with broad adoption.

Nevertheless, the entire idea of personal data store is not bad. It encrypts the data of the users and gives them protection and privacy. And that is how we arrive to the idea of combining the both approaches - keeping the data on the server and keeping it encrypted somewhere else. This way, we can have the functionality and speed of the data, kept in a server and also security for the more sensitive data, that we are not willing to upload to the server.

1.1. Research Scope and Question

In this paper, I will provide information about the framework called Solid, which emerged in the recent years. It is aiming to achieve the right mixture between private and public data usage and also help the website developers manage the data. The paper will provide mostly technical details about the functionality and way of operation of Solid. In order to understand the framework from inside, I had the task to incorporate it with NaviServer. Therefore, in the scope is also the specific incorporation of Solid with NaviServer, with configuration details and requirements.
The final aim is to have a working prototype of NaviServer using Solid for the data storage and data-related processes.

1.2. Paper Outline

The paper will start with an Introduction. Afterwards, the background information will be provided with all the relevant technologies needed to understand the topic. We will continue with the research methodology and description of the steps needed to reproduce the incorporation. Next will come the analysis of the results and discussion on whether the technologies are suitable and what problems might emerge from using them together. Finally there will be a conclusion, which will wrap up the paper. At the end of the document, you can find an appendix, where longer snippets of code will be written along with some relevant pictures and also known bugs and fixes for Solid.

2. Background

In this section, I will give explanation about the relevant technologies needed to understand the way Solid operates and how my approach has been developed. I will start with some background information, which is a prerequisite to understand Solid - Linked Data. Then the way Solid operates will be explained. This section will conclude with a brief overview of NaviServer and explanation about reverse proxies.

2.1. Linked Data

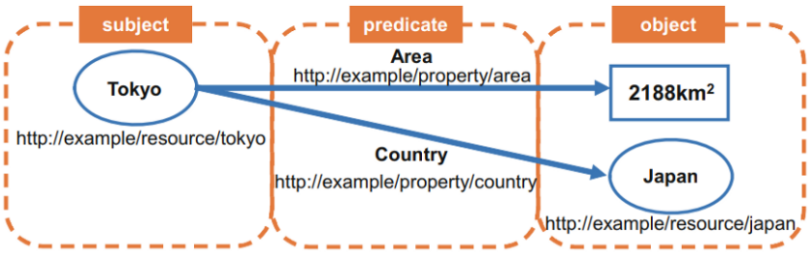
The Web initially started as a scattered collection of documents published online [1]. They were all accessible via a URL, but were not linked together and therefore it was hard to find resources one was looking for by just using a given web page. This problem was fixed by introducing the hypertext links and search engines, which index the documents and analyze the structure of the links between them to present the users with the best results for their query [2]. However the issue with the data remained - there was no data format or a standard that was good enough to store the data and keep it structured. There existed raw dumps in CSV or XML formats, which are good enough to store the data, but are not expressive enough to enable the relationships between different linked documents [2]. The problem is that these documents are mainly human-readable, but it is hard for a machine to understand them and therefore process them in a uniform way [3]. This means that it is very hard to automate anything on the web, also because of the volume of information it contains. For example, if we wanted to get the data about a person from their random social network profile and use it to make a reservation in another website, the requested data structure won't match the provided and therefore the transfer will be impossible. There is of course the option to create ad hoc APIs in order to query the selected resource, but it is impossible to have the same queries work on all of the websites [3]. The solution for this is to use metadata to describe all of the data on the web [3]. Metadata is data about data and in this context it stands for "data describing web resources".

And that is the what the main goal of Linked Data is - to publish structured data using vocabularies (like schema.org) that can be connected together and interpreted by machines [4]. It represents an effort to unify the data that exists in the web. One of the main contributors and supporters of the idea is Tim Berners-Lee, who in 2006 described linked data as follows: *"The Semantic Web isn't just about putting data on the web. It is about making links, so that a person or machine can explore the web of data. With linked data, when you have some of it, you can find other, related, data."* He also outlined 4 rules,

which are now known as Linked Data principles [2] [5]:

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

This is the basic recipe and as we can see, the main idea is to structure the data according to the standard and most importantly - to include links to other resources, so the network of linked documents can grow. As we can see from the rules above, while the primary units of the hypertext Web are HTML documents connected by untyped hyperlinks, Linked Data relies on documents containing data in RDF (Resource Description Framework) format [2]. It serves as a foundation for processing of metadata and provides interoperability between applications, which exchange information on the Web. RDF aims to enable automated processing of Web resources and can be used in many different domains such as search engine optimization, cataloging and etc. [3]. A RDF statement consists of 3 parts: subject, predicate and object [6].



From the picture we see, that Tokyo is located in Japan. They are both objects (circles) and have their own URI in order to be identified in a machine-readable format. In contrast, we can see that the object 'area' in the rectangular box doesn't have an address - it is a literaI(constant). In order to access the specific resource we use the predicate. There exist four main RDF formats to store the data, which all adhere to the structure mentioned above: -Triples (.nt), Turtle

(.ttl), JSON-LD (.json) or RDF/XML (.rdf) [6]. They all have their benefits and advantages and have slightly different syntax, but the main concept is the same - to describe the data and links between resources. In this paper we will focus on the Turtle (.ttl) format, because it is the one, that Solid uses.
Turtle is perhaps the easiest format to read by a person. In contrast to the others, it is more concise and the statements, that are generated for it are shorter, because we can assign prefixes at the beginning of the .ttl file. Furthermore, instead of having repeated lines referencing the same resource, Turtle groups them into blocks, with all indented lines after initialization referring to the same resource [6]. Related information to the subject is separated with a semi-colon and finished with a full-stop and a newline to indicate a new subject. If we take a look into the code in the Appendix [1], we will see the typical schema used for the person Bob Marley. We see the assigned prefixes in the beginning for example the line

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

assigns to the variable foaf the URI "http://xmlns.com/foaf/0.1/". This leads to shortening of the statement foaf:Person afterwards. It refers to the longer <http://xmlns.com/foaf/0.1/Person> and if we have multiple persons, this will save us a lot of writing in contrast to the other RDF syntaxes. There is also one very specific shortening, which is represented with the letter 'a' and refers to http://www.w3.org/1999/02/22-rdf-syntax-ns#type. It is so common, that it is included in the Turtle syntax by default and doesn't have to be defined when initializing a file. All of the different shortenings lead to really clear and concise triples in Turtle in contrast to N-Triples. For example the N-Triple

```
<http://dbpedia.org/resource/Bob_Marley>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Person> .
```

can be represented in Turtle as

```
dbr:Bob_Marley a foaf:Person.
```

The first part represents the RDF subject (dbr:Bob_Marley), the second (a) - the predicate and the 3rd - the object - Person. Now, after we have connection to the Person model type, we can reuse it in all our applications and therefore make the exchange of data between them easier. The entire syntax of Turtle and RDF in general is difficult to read in the beginning, but after we know the foundations, we are able to understand the structure of the files and the meaning of the connections.

After all, that was being said about the way the structure and the linkage of the resources, one could ask himself: what is the point. And the answer is - to have higher availability and access to this information. The way to achieve this is by using queries, which need to be built to work with the RDF standard. That is why SPARQL (SPARQL Protocol and RDF Query Language) [7] was developed. SPARQL queries contain triple patterns, much like the data itself, which utilize the relationships to quickly navigate any linked data [8]. It is common for all linked data, and enables the queries to access multiple RDF databases at once. I won't go in-depth with SPARQL in this topic, as it is a broad area to discuss, but will provide a simple example, which will help us understand what we will later see in Solid. This query lists the names of every gold medalist (excluding duplicates) [8]:

```
PREFIX walls: <http://wallscope.co.uk/ontology/olympics/>
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?name
WHERE {
 ?instance walls:athlete ?athlete ;
 walls:medal <http://wallscope.co.uk/resource/olympics/medal/Gold> .
 ?athlete rdfs:label ?name .
}
```

We begin by listing the prefixes used in the query to make it more readable, as we did with Turtle syntax. Variables are created with a "?" prefix. The selected clause is ?name, which will display the names of the athletes. The first triple

we match is:

```
?instance walls:athlete ?athlete ;
```

This grabs every entity in the wallscope.co.uk database, that is of type athlete. We can see, that this statement ends with a semicolon, instead of a dot. This is because the next statement is referring to the same subject - ?instance and is adding another filter to the where clause, filtering by gold medal winners only. Afterwards, we see the dot and the next statement begins, which returns the name of the matched athletes in human-readable format. Notable examples using Linked Data and have SPARQL endpoints for querying are DBpedia [9] - the Wikipedia DB and many government websites, allowing for open data browsing.

2.2. Solid

The reason for all the information provided for Linked Data is, that it is the main driving force behind Solid and is the way the data is organized in the framework. The term Linked Data is integral part of Solid and is actually embedded in its name (Social Linked Data). In this subsection, I will start by giving an overview of Solid and then will dive into the specifics and provide several examples in the process in order for the reader to understand the functionalities and way of operation.

An idea that was borrowed by Solid, but further developed and mixed with others, was the concept of Personal Data Store. The PDS's goal is to separate the data from the applications and give the users more freedom and further options in deciding where to place their data. This will result in a better online privacy and will reduce the companies' power and influence over their users' decision making (personalized ads). It is important to note the difference between the Personal Data Stores and Personal Clouds. The former allow different organizations collecting data for their own use after authorization, in contrast to the clouds, which keep the encrypted data only for the owners' use [10]. The authors of this article published a research at the end of 2018 suggested, that at this

stage the adoption figures are low and some of the reasons for that might be, that they are too technical and are a challenge to use for non-technical users and the only thing they offer is the improved privacy, which is not appreciated so much by most of the regular users. Furthermore in this article, we can see, that Solid is not the only effort in this area and along it, few other PDS providers are mentioned: MyDex, Digi.me, Hub of all Things and etc. Each one providing pretty much the same functionality, but using different software architecture and having different business model.

So, how does a personal data store work? Instead of data about us belonging to internet monopolies, PDSs promise to give back control to users, enabling them to 'own' their data and control access through granular permissions [10]. In the best case scenario of this model, app developers simply provide the interface and functionality of, for example, a calendar or journal app. The data always lives in your datastore. When you browse your journal or calendar in a web or desktop/phone app, the data from your datastore is displayed in the interface, but it's securely transmitted between you and your datastore. No other parties are able to access it [11]. We can clearly see what the benefit from using a PDS is for the end user - the improved privacy, but there are also huge benefits for the app developers. They will be able to write software, which will adhere only to one data model and will therefore make the development faster and more app and platform-independent. However the author spotted a real issue with the most important party in the software world - the corporations [11]. If the PDSs become a standard in the future, they will lose one of their greatest advantages these days - the access to personal data and will therefore have reduced selling potential. That is why the market leaders in the software industry are not interested in the idea and are not investing in it.

In another article published in Wired [12], the author provides a brief overview of Solid and gives further examples of how the power of data can be unlocked by using Solid. He suggests, that this architecture will address the data scaling issues and the AI algorithms may become more efficient. For example, an insurance firm may send you a program, that can be run on your own computer,

using the health data gathered by your fitness tracker instead of you sending the company the data and sharing it. This way, the company can offer a better quote, based on your current condition and won't receive sensitive information. *"This isn't somebody else running an AI on you; that's you running an AI on the whole planet - everything you can see, everything to do with you,"* Berners-Lee says [12]. Furthermore, the worry of hacking into centralized databases will be decreased, because the data will be scattered and held privately by its owner. After this overview of the personal data stores, it is now time to look into the Solid specifics. By reading the home page of the Solid project [13], the first thing to note is that it started as a university research project, led by Prof. Tim Berners-Lee, inventor of the World Wide Web, taking place at MIT. Then we can find an overview of the project goals and what it offers to its users. The project statement is: *Solid is a proposed set of conventions and tools for building decentralized social applications based on Linked Data principles. Solid is modular and extensible and it relies as much as possible on existing W3C standards and protocols* [13]. The authors have identified three main points, from which a user can benefit, when using Solid:

1. True data ownership - Users should have the freedom to choose where their data resides and who is allowed to access it. By decoupling content from the application itself, users are now able to do so.
2. Modular Design - Because applications are decoupled from the data they produce, users will be able to avoid vendor lock-in, seamlessly switching between apps and personal data storage servers, without losing any data or social connections.
3. Reusing existing data - Developers will be able to easily innovate by creating new apps or improving current apps, all while reusing existing data that was created by other apps.

Currently, the project has moved outside of the university scope and is receiving huge attention from web developers all around the world. In the official MIT website however, one company name is mentioned - the start-up Inrupt,

Inc. The company's Solid Community Site [14] is recommended by the founders as thee best resource for all things Solid, including documentation, examples, community and POD hosting.

POD in the context of Solid stands for personal online data store [15]. It is Web-accessible and is the place where user information is stored. Users are allowed to have multiple pods from different providers and switch between providers easily.

Furthermore it is also possible to setup and maintain your own Solid Server [16], which will be discussed in depth in the methodology section. Within the Solid ecosystem, each user has the right to chose where to store his data. One can have a workplace POD, POD at home or it can be hosted by a third party provider. Because the standard is the same in all different providers, the transfer between them can be done easily and without interruption. This helps not only the users by giving them the option to 'Sign in with Solid POD' instead of 'Login with X' or 'Login with Y'. It is also helpful for the software developers, by enabling them to use the same data aggregation patterns for all the applications. Currently there are many projects trying to develop applications conforming to Solid standards and some of them are already included in the Solid PODs i.e. Notes, Chat, Dokieli and etc.

Users are allowed to securely store anything they want in their PODs, furthermore they are capable of accessing pods of other users. In order to achieve consistency and to store the access configurations, Solid uses Web Access Control [17]. Notable benefits from using it are that it again uses RDF Turtle as the main syntax and supports inheritance, so we can describe access control on folder level and all the files in the folder will adhere to it. The different access levels are similar to the ones used in Linux file systems - Read, Write, Append, Control. Solid gives the option to store the files in a Private folder and a Public one. By default the Private folder is not accessible by anyone except the POD owner and the user is not allowed to grant privileges to external users from the Solid interface (this can be changed from the .acl file). In contrast, the Public folder is accessible in Read-only mode by default and the POD owner can grant

write, delete and control privileges to external users from the Solid interface [insert pic here]. Furthermore all of these access rights can be changed by editing the .acl file in the specific folder and defining new rules. As mentioned, we can change the access control on folder level, but also on file level, so we might want to create one file, that is editable by third parties in a folder, that doesn't permit it. The ACL files can be found in the folder, that Solid creates for the specific resource. The file is named $../resource_name/.acl$. Here is an example .acl - the default public folder access control file:

```
# ACL resource for the public folder
@prefix acl: <http://www.w3.org/ns/auth/acl#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

# The owner has all permissions
<#owner>
    a acl:Authorization;
    acl:agent <https://localhost:8443/profile/card#me>;
    acl:accessTo <./>;
    acl:default <./>;
    acl:mode acl:Read, acl:Write, acl:Control.

# The public has read permissions
<#public>
    a acl:Authorization;
    acl:agentClass foaf:Agent;
    acl:accessTo <./>;
    acl:default <./>;
    acl:mode acl:Read.
```

After having the access permissions in place, we are going to need some sort of authentication in order to identify ourselves and be able to access the different resources, based on the permissions we have. In Solid, the authentication is

achieved in two ways: using WebID [18] or using WebID - TLS [19]. The WebID method is simpler to start with and is suggested by the developers to use for new users. On the official project description page we can read: A global and decentralized social Web requires that each person be able to control their identity and that this identity be linkable across sites, thus placing each person in a Web of relationships. The general idea behind WebID is that agents (e.g., a person, an organization, a group, etc.) create their own identities by linking a unique identifier in the form of an HTTP(S) URI to a profile document, a type of Web page that any Web user is familiar with, and which uses a standardized RDF serialization format. The profile document contains all the necessary information to create a Web of trust which allows people to link together their profiles in a public or private manner [15]. By registering to Solid a new WebID is created and it stays in the address: https://solid_domain/profile/card#me. We can also find it in the Solid server folder /profile/card.ttl. That is where our identity is stored and how Solid recognizes the user that is initiating a request. This file can be changed depending on preferences and this is where permissions for new Apps can be granted.

The other authentication method - WebID-TLS is a decentralized authentication protocol which enables secure and efficient authentication on the Web. It enables people to authenticate onto any site by simply choosing one of the client certificates proposed to them by their browser. These certificates can be created by any Web site for its users.

In Solid, data is managed in a RESTful way. New data items are created in a container by sending them to the container URL with an HTTP POST or issuing an HTTP PUT within its URL space. Items are updated with HTTP PUT or HTTP PATCH and removal is done with HTTP DELETE. In order to find items, we are using HTTP GET and following links. A GET on a container returns an enumeration of the items in the container [15]. In order to express more complex data retrieval or manipulation queries, Solid supports SPARQL. Application data is stored in documents, which are recognized by URIs.

Initially Solid started with different prototype servers running on different soft-

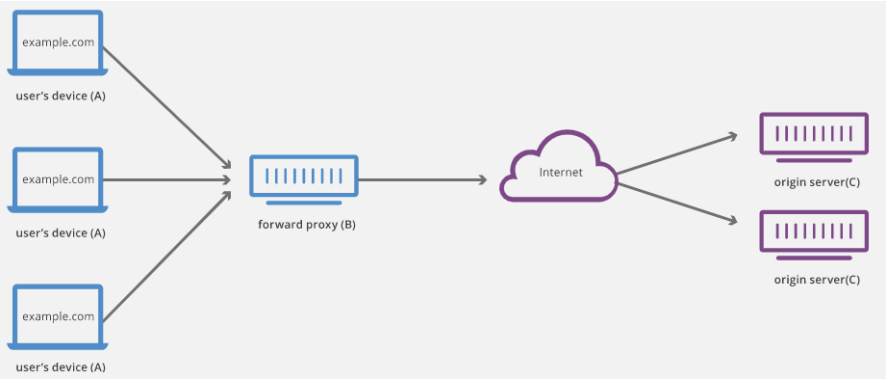
ware languages (i.e. PHP, JavaScript) and architectures (using the file system or using a RDBMS) [15]. Currently the most advanced one is the NodeJS version [16]. It is open source and is being updated and fixed every day by dozens of developers. In order to run our own Solid server and being able to host PODs, we must install the NodeJs version. This will allow us to run the service on top of the file-system and we will be able to configure it according to our preferences. The GitHub page gives us information on how to set it up, how to run it and also what the different flags are designed for. It provides variety of options - you can run it in a single or multiple user mode, with or without email server, via proxy, in debug mode and many others. I will provide information on how to set it up plus examples in the methodology section. The best place to find information and ask questions to the community are the Solid forums [20]. That is where we can really see, that there is constant activity on this project and progress is being made.

2.3. NaviServer

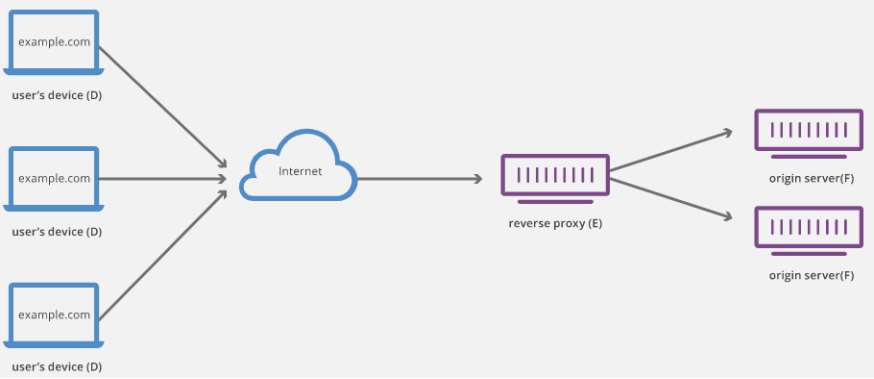
The web server, that is going to be used for the proof of concept is by requirements NaviServer. It is an extensible web server suited to create scalable web services and sites [21]. It was originally based on AOLServer [22], but has developed into an independent web server and is under Mozilla Public License. It supports multiple features, that a modern web server does, like [23]: Built-in cron-like scheduling, Pooled database connections, IPv4 and IPv6, HTTP/HTTPS client support, Reverse proxy, Pluigins, Extensions and many more. The pages, which are hosted on NaviServer are using the TCL language [24], and it furthermore supports NX and Next Scripting Framework (NSF) [25], which are allowing object-oriented software development with TCL. Currently the most notable use of NaviServer is with OpenACS [26]. It is a toolkit for building scalable community-oriented web applications and is the foundation for many products and websites including .LRN (e-learning platform) [27]. The forum of OpenACS [28] is where one can find support for NaviServer and is home to the NaviServer community.

2.4. Reverse Proxy

In order to understand the way the proposed solution will work, we need to have knowledge of what a Reverse proxy is. First, one must make a difference between a reverse proxy and a typical proxy server. A forward proxy (proxy server) is a server that sits in front of a group of client machines. When those computers make requests to sites and services on the Internet, the proxy server intercepts those requests and then communicates with web servers on behalf of those clients, like a middleman [29]:



A reverse proxy is a server that sits in front of web servers and forwards client (e.g. web browser) requests to those web servers. Reverse proxies are typically implemented to help increase security, performance, and reliability [29].



As we can see, in contrast to the way the forward proxy operates, the reverse proxy is situated in front of the web servers rather than in front of the client machines. The reverse proxy is intercepting requests, sent by the client to any of the web servers in front of which the proxy operates. The proxy then forwards the request towards the requested server. After a response to the request is available, it is sent by the server to the reverse proxy and then to the client. In simpler terms, we could say, that the reverse proxy acts as a intermediary between the client and the server. A reverse proxy server is a type of proxy server that typically sits behind the firewall in a private network and directs client requests to the appropriate backend server [30]. Furthermore, we could use a reverse proxy to act as a gateway to access different services, which we want to provide, running on different machines and using different programming languages. Forward proxies are used to bypass some state or institutional restrictions or to protect one's identity and location online. Reverse proxies are most commonly used for load balancing, CDNs, web acceleration and improved security.
To sum it up we could say that a forward proxy sits in front of a client and ensures that no origin server ever communicates directly with that specific client. A reverse proxy sits in front of a server and ensures that no client ever communicates directly to that server.

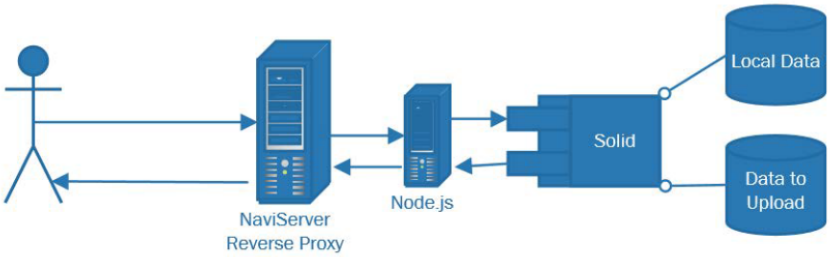
3. Research Method

In this section, I will explain the way my proposed solution works and will justify my approach selection. In order to do that, I will start with providing two alternate approaches, which I deem appropriate and will compare the chosen one with them. Afterwards I will provide a step-by-step guide to reproduce the solution along with clarifications for each step.

The first solution I came up with was the following: Create a NaviServer extension module, that will ask the user when uploading a file, whether he/she wants to store the data on the server or locally. In order to achieve the required

functionality I was going to need to extend NaviServer to operate with the file-system via some pre-defined tcl functions. There should be a folder with files marked for server storage and a user folder with all the private files. The user folder, will only be accessible with permission from the user. [Fig. 1 in the Appendix] There are multiple problems with this solution. The first one is the time it will take to implement such a solution. This is an entire module and hides many obstacles, because of the encryption and the interoperability with the Linux file-system. Another issue is the functionality - by using this solution, we will have just basic personal data store functionality and will miss out on most of the good aspects it provides.

The second applicable solution was to use the SpiderMonkey Tcl to JS extension [31]. It embeds a JavaScript interpreter to a Tcl application. The idea was to configure SpiderMonkey to act as a bridge between Solid and the server directly. Using it, the JavaScript code will be run directly on the server side (being translated to Tcl) and this will remove the need to write an extension NaviServer module [Fig.2 in the Appendix]. This solution looked more feasible and attractive, but I found the SpiderMonkey interpreter rather difficult to configure. Furthermore, after examination of some of the Solid files (i.e. mashlib.js), I assumed there will be many inconsistencies between the languages and the mapping would be difficult for some of the files. Therefore , after careful consideration, I decided to select a third approach:



Running Solid behind the NaviServer Reverse Proxy [32]. The idea is, if a specific URL is called, the NaviServer will redirect us and will call the Solid service, which is running simultaneously. This solution includes many advantages over

the other two:

1. It doesn't interfere with any of the other components of the software system. As the Solid service is running independently behind the proxy, only a specific call to it will let one access it.
2. From maintainability perspective, this is the best option - if we wish to update it, we are going to be able to use the latest version available directly from the software provider.
3. Extensibility - it provides the option to define special rules for the way the data is treated on our server.
4. The portability is also addressed with this solution. Solid is platform-independent and therefore the configuration will work for all platforms.

I am using Ubuntu 16.04 operating system and this guide will include information and commands to be run on Ubuntu. Nevertheless this proposal should be able to be used on different systems after a few tweaks. The first thing to do in order to reproduce the proposed setup is to install NaviServer [23]. This can be easily done by following the instructions on the OpenACS "How to install" page [33]. The guide on this page is designed for Linux users and the entire process requires to clone a git repository and run a script from it. It will install all the dependencies and after a few minutes the process should finish with the message:
Congratulations, you have installed OpenACS with NaviServer on your machine. You might start the server manually with sudo /usr/local/ns/bin/nsd -t /usr/local/ns/config-oacs-5-9-1.tcl -u nsadmin -g nsadmin
This is the default command to run OpenACS. We can see from it where the configuration file is located: /usr/local/ns/config-oacs-5-9-1.tcl. This will be used later in order to configure the reverse proxy. In order to run the server more efficiently and with the log appearing on the console, I suggest to run it with the -f flag (in the foreground). After running the suggested command, oacs-5-9-1 service will be started on port 8000 and we will be able to access it and see the service log in the shell. In the instructions page you can find also

other ways to run the service and read from the log file in parallel in order to debug, so there are options to chose from.

Next, after already having NaviServer with OpenACS up and running, we are going to need to install the Reverse Proxy extension module [32], as it doesn't come with the basic installation. In the ReadMe file on the Bitbucket page the process to install the reverse proxy is explained along with a sample configuration. We need to clone the repository again, however this time instead of Git, Mercurial is required. Afterwards we go into the cloned folder and write "make install" in the shell. A folder will appear in the NaviServer modules and we will be able to use the reverse proxy functionality.

The final piece of the puzzle is Solid. As mentioned above, Solid runs on Node.JS, therefore Node is a prerequisite. Then the instructions for installation can be found in the GitHub page of the server, but in order to reproduce my setup entirely the commands can be found in the Appendix [2]. Here again we start by cloning a git repository. Afterwards in order to run Solid, we are going to need an SSL Certificate. For testing purposes it doesn't need to be signed by a certificate authority and can be self-signed instead. That is what the **"openssl req..."** command serves for - it creates a private and public keys, which can be presented to Solid, so it can run. Next, we need to initiate Solid and create the configuration file. This is done with the command **"solid init"** (you may be prompted to install the commander package).

When we are initiating Solid, we can choose where the data is to be stored, which port Solid is allowed to run on (default is the secure port 8443), the URI, the configuration and metadata locations. Here is also where we need to point to those certificates we created earlier. In this guide, we will be running Solid for single user only, as this is only a test environment, but the multi-user version should be able to be configured in a similar manner. The only difference would be, that instead of the default URL localhost:8443, we will have the URL: username.localhost:8443 for each user. The last few lines of the configuration are mostly informational and we can write whatever we like there as description.

After having the Solid server initiated, we are now able to start it with the

command **"solid start"**. However, in order to use self-signed certificates in our test environment, we must use the flag **"–no-reject-unauthorized"**. Furthermore, a useful way to find errors in Solid and track all the requests is to start it in Debug mode. This can be done using the following command: **sudo DEBUG=solid* ../bin/solid start –no-reject-unauthorized**. After running it, the access and request logs will appear in the terminal and it is easy to check what is not working and where it comes from. For example, after running it for the first time, you may find that a file for the storage quota is missing. This is not a required file to run the Solid service, but can be useful if we want to place limits on the folders of the different users.

When we are at this point, all the prerequisites have been met and we are able to start configuring them to work together. As already mentioned, the main place for this to be done, is the OpenACS configuration file, which is located in the **/usr/local/ns** folder. There is also the option to run it with plain NaviServer with the shorter configuration file, but this guide focuses on OpenACS setup. The first thing to be done is to enable the NaviServer https port. By default the configuration comes with disabled secure connections, but for this project, we are going to need to enable them. This is done by simply uncommenting the **set httpsport 0** line. This will enable all ports to be used for secure connections.

Next comes the configuration of the reverse proxy. This is where the redirection is being made and is therefore the most important part to configure. If installed correctly, the revproxy module must have added an entire section of code, which is pointing to the /revproxy/ folder. This is the way NaviServer is organizing its modules and in this code section is where we need to make our adjustments in order to make Solid work appropriately in parallel. The way the reverse proxy works, is that it checks if a URL contains a specific string. If it contains it, the request is redirected to another pre-defined URL, possibly doing string substitutions and parsing. If the called URL doesn't contain the required string, it is processed as normal. There are three different points in time this redirection can be made: preauth, postauth or trace. The preauth redirects us before the authorization is made from the called service. Postauth - after the

authorization was made. The trace command makes the redirection after the first request is finished. In this context, we are using the **postauth** option. In order to register a new filter we use the command:

```
ns_register_filter postauth HTTP_METHOD STRING_TO_MATCH
::revproxy::upstream -target TARGET -regsubs STRING_TO_SUBSTITUTE
```

For HTTP method, we can substitute with whichever method we like to e.g. GET, POST, PUT, PATCH, DELETE, as long as the target service supports the method and understands what it is meant to do. After the HTTP method comes the string, by which the filtering must be made. In our case, the first string, which we would want to filter for is **/solid/***. This means, that we will be redirected each time we call url with the following pattern: **http://localhost:8000/solid/***. The redirection will be processed in the ::revproxy::upstream function, which we can further examine in the revproxy module. It is beneficial to use the command ns_log in different places in the reverse proxy file in order to understand what happens in the lifetime of a request, along with all the headers and string substitutions. The -target parameter is used to point to the target location, where we would like to redirect us if the function is called. And finally, the regsubs method substitutes parts of the captured URL with a desired pattern. For example we can substitute everything after the /solid/ part with the empty string. This is useful if we want to use the upstream URL for advanced redirection. It is possible to register only one filter and then redirect all the requests happening from inside this address to the target URL after doing the string substitution and appending the target. However this is not the case with Solid. There is a problem with it and we are not able to keep track of the URL from which a request is being called. Therefore we are not going to use a single filter here, but instead we will have multiple filters for the different pages in Solid.

As mentioned, the first filter, that was registered is serving the purpose to redirect us to the Solid home page:

```
ns_register_filter postauth GET  /solid/* ::revproxy::upstream
```

```
-target https://localhost:8443/ -regsubs {{/solid/ ""}}
```

However this is not enough, we need to redirect to all the CSS files, which are responsible for the appearance of the site. They are found in the Solid folder /common/ and this is where it looks for them. That is why, we need to filter for **/common/*** and redirect. Furthermore the login page, login popup window and profile page need to be adjusted. For them, it is enough to have only GET requests. However if we want to be able to properly access, create, edit and delete files, we need to make the proper redirection to the private and public folders. We have to register a separate filter for each action with the specific HTTP method. All the filters required to process the requests can be found in the Appendix [3].

After the configuration file has been updated, we can now start Solid and OpenACS and explore what has been achieved. We type the request to /localhost:8000/solid/. The home page appears, but we are not logged in yet. We can see, that the url is showing localhost:8000 instead of localhost:8443, which was the purpose of using the reverse proxy. Next, we press the Login button and a pop-up window shows. There, if we are running Solid for the first time behind the proxy we need to write manually the address: https://localhost:8443/ and then click go. This will lead us to the login page of Solid, assuming we have an account already, if not - we are able to register as well. Afterwards we log in and we are redirected to the home page again, but this time we are authenticated and our unique WebID appears on screen, along with a link to the profile. We are now able to go there and change our profile settings as well as add URLs, which are able to access our private and edit our public files. We have to add the URL http://localhost:8000/ to our trusted sites list in order to make it possible to change files from the reverse proxy. Even if we forget to do it manually, we are going to be prompted to on our first visit of the public folder [Fig. 3 in the Appendix].

Now, after we have access, we are able to navigate to all folders without obstruction - all the links are working fine and we are able to read from the public

folder. However, after we try to access the Inbox or the Private folders, we get an error message telling us, that we are not logged in (Error 401 - Unauthorized). I have found a way to bypass this manually. Nevertheless, this is one of the limitations of the current approach and will be further discussed in the Limitations section.

4. Analysis and Limitations

4.1. Analysis

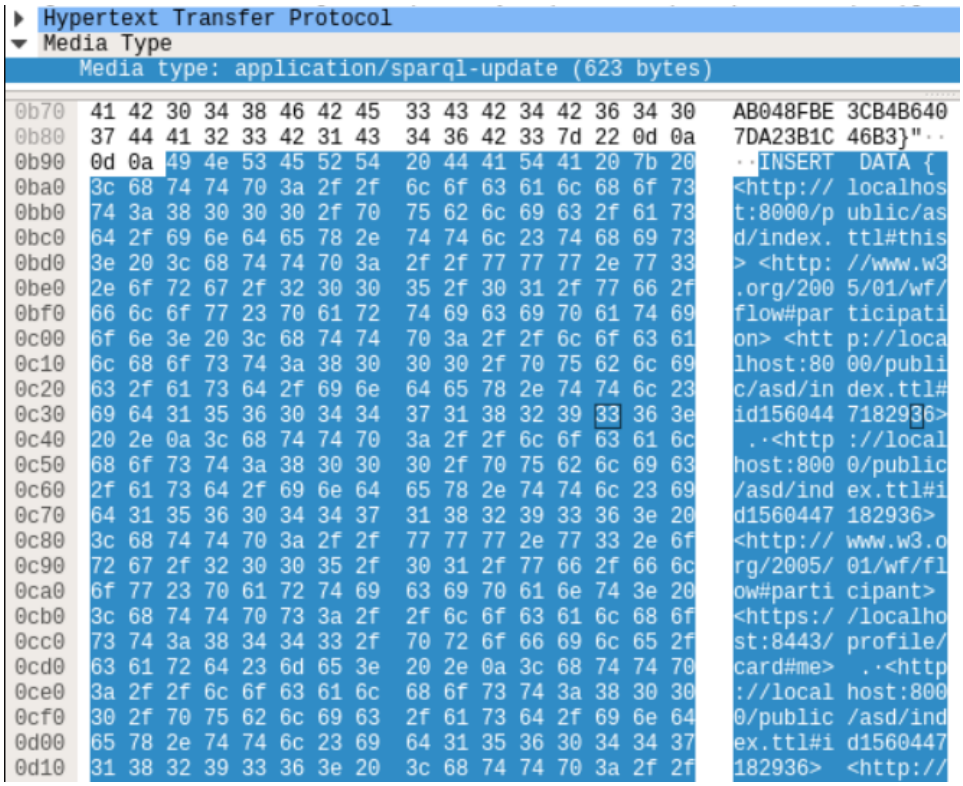
The current solution is sufficient to serve as a proof of concept for the idea. It shows that the approach, that was chosen is possible and the configuration made has the potential to become robust and error-proof after more testing. The requests are being called from the NaviServer and processed by Solid. For the end-user it would seem, that everything is being served by NaviServer, which was the initial idea. The main advantages of this solution were, that we would have higher tuning potential of the server and easier maintainability than with the other approaches. As we can see, by adopting this approach, the goals can be met successfully. The Solid server can be configured independently according to our preferences and it won't interfere with the NaviServer. Furthermore, we are able to use the latest updates and bug fixes, because we are using the official Solid version, instead of building less efficient and functional one.

4.2. Limitations

However, there are not only advantages by using this solution, but also limitations. The most obvious one is that when we are using the filtering, we are actually eliminating some strings to be used from NaviServer. For example we won't be able to have the URL http://localhost:8000/public/ serving a different purpose, than accessing the Solid public page. This may prove to be an issue in bigger websites as some conflicts may arise and must be paid attention to.
Another obstacle, which I couldn't fix is the authentication. Currently, when we log in, a session is opened and we can see, that the user appears to be logged in.

This is partially correct, as we can access the profile page and make adjustments there. However, we are not allowed to edit new files, or access the private folder (assuming we are using the default Access Control Levels). By digging into the problem, I found out, that Solid uses two separate cookies for authentication - one for session id and one for the connected user id. The session id is passed from Solid to NaviServer via the reverse proxy, but the one called connect.sid is not. This can be fixed manually by editing the reverse proxy module and appending the connect.sid cookie to the already existing cookie header. The cookie can be found by accessing the Solid service on its default address and loading the private page with F12 key (developer tools) turned on. From there we can copy the missing cookie from the header and append it to the existing one.

Furthermore, by testing all the HTTP methods, they all seem to work properly except for the PATCH method. This means, that we are able to create, view and delete files from NaviServer, but we are not able to edit them. The reason for this is, that Solid is using SPARQL queries to edit the data in the files. I have used WireShark [34] to capture the packets and check what causes the error. Usually when we try to update the file /public/filename/index.ttl, Solid sends a SPARQL query to first delete the old data and write it again, but this time including the appended data. However, as we can see, the INSERT request from SPARQL is made to http://localhost:8000/ instead of https://localhost:8443/. There is no such file on the location, where SPARQL is looking for it and we get Error 409 - Conflict. I have tried fixing this by moving the Solid data folders to the locations, where NaviServer is looking for its data, but this didn't seem to have any effect.



5. Conclusion

In general the entire concept of Solid looks great and if used correctly can lead to improvement of the software systems and the privacy of their users. However currently the state of it is not at the sufficient level to be used by a non-technical person and will require few more years of development to have a greater impact. Furthermore the idea of incorporating Solid with NaviServer seems feasible and the approach presented can be the first step in the right direction.

Bibliography

[1] European Data Portal. Introduction to linked data. URL https://www.europeandataportal.eu/sites/default/files/d2.1.2_training_module_1.2_introduction_to_linked_data_en_edp.pdf.

[2] Tim Berners-Lee Christian Bizer, Tom Heath. Linked data - the story so far.

[3] W3C. Resource description framework (rdf) model and syntax specification, . URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.6030&rep=rep1&type=pdf.

[4] Wordlift.io. What is linked data. URL https://wordlift.io/blog/en/entity/linked-data/.

[5] Tim Berners-Lee. Linked data principles. URL https://www.w3.org/DesignIssues/LinkedData.html.

[6] Angus Addlesee. Understanding linked data formats, . URL https://medium.com/wallscope/understanding-linked-data-formats-rdf-xml-vs-turtle-vs-n-triples-eb931dbe9827.

[7] W3C. Sparql query language for rdf, . URL https://www.w3.org/TR/rdf-sparql-query/.

[8] Angus Addlesee. Constructing sparql queries, . URL https://medium.com/wallscope/constructing-sparql-queries-ca63b8b9ac02.

[9] Dbpedia. URL http://dbpedia.org/sparql.

[10] Simon Worthington Irina Bolychevsky. Are personal data stores about to become the next big thing? URL https://medium.com/@shevski/are-personal-data-stores-about-to-become-the-next-big-thing-b767295ed842.

[11] Irina Bolychevsky. How solid is tim's plan to redecentralize the web? URL https://medium.com/@shevski/how-solid-is-tims-plan-to-redecentralize-the-web-b163ba78e835.

[12] Nadav Kander. The web's greatest minds explain how we can fix the internet. URL https://www.wired.co.uk/article/the-webs-greatest-minds-on-how-to-fix-it.

[13] Solid home. URL https://solid.mit.edu/.

[14] Inrupt. URL https://solid.inrupt.com/.

[15] Essam Mansour et.al. Solid: A platform for decentralized social applications based on linked data.

[16] Solid server. URL https://github.com/solid/node-solid-server#solid-server-in-node.

[17] Web access control. URL https://github.com/solid/web-access-control-spec.

[18] Webid specification. URL https://www.w3.org/2005/Incubator/webid/spec/identity/.

[19] Webid-tls specification. URL https://www.w3.org/2005/Incubator/webid/spec/tls/.

[20] Solid forum. URL https://forum.solidproject.org/.

[21] Naviserver. URL https://sourceforge.net/projects/naviserver/.

[22] Aol. URL http://aolserver.sourceforge.net/.

[23] Naviserver home. URL https://wiki.tcl-lang.org/page/NaviServer.

[24] About tcl. URL https://www.tcl.tk/about/language.html.

[25] Next scripting framework. URL https://next-scripting.org/xowiki/about.

[26] Openacs. URL https://openacs.org/.

[27] Learn. URL http://www.dotlrn.org/.

[28] Openacs forum, . URL https://openacs.org/forums/.

[29] What is a reverse proxy? URL https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/.

[30] What is a reverse proxy server? URL https://www.nginx.com/resources/glossary/reverse-proxy-server/.

[31] Spidermonkey tcl to js interpreter. URL https://github.com/flightaware/tcljs.

[32] Gustaf Neumann. Naviserver reverse proxy. URL https://bitbucket.org/naviserver/revproxy/src/default/.

[33] Openacs - how to install, . URL https://openacs.org/xowiki/naviserver-openacs.

[34] Wireshark. URL https://www.wireshark.org/.

6. Appendix

[1] Linked Data Example

```
@prefix dbr:    <http://dbpedia.org/resource/> .
@prefix dbo:    <http://dbpedia.org/ontology/> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix geo:    <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <http://schema.org/> .

dbr:Bob_Marley
a foaf:Person ;
rdfs:label "Bob Marley"@en ;
rdfs:label "Bob Marley"@fr ;
rdfs:seeAlso dbr:Rastafari ;
dbo:birthPlace dbr:Jamaica .

dbr:Jamaica
a schema:Country ;
rdfs:label "Jamaica"@en ;
rdfs:label "Giamaica"@it ;
geo:lat "17.9833"^^xsd:float ;
geo:long "-76.8"^^xsd:float ;
foaf:homepage <http://jis.gov.jm/> .
```

[2] Solid Server Setup

```
 999  mkdir Solid
 1000  cd Solid/
 1001  git clone https://github.com/solid/node-solid-server.git
 1002  cd node-solid-server/
```

```
 1003  mkdir build
 1004  pushd build
 1005  openssl req -outform PEM -keyform PEM -new -x509 -sha256
 -newkey rsa:2048 -nodes -keyout ../privkey.pem -days 365
 -out ../fullchain.pem

 1007  ../bin/solid init
 1009  npm install commander --save
 1013  ../bin/solid init

? Path to the folder you want to serve. Default is
/home/pmpetrov/Solid/node-solid-server/build/data
? SSL port to run on. Default is 8443
? Solid server uri (with protocol, hostname and port) https://localhost:8443
? Enable WebID authentication Yes
? Serve Solid on URL path /
? Path to the config directory (for example: /etc/solid-server) ./config
? Path to the config file (for example: ./config.json) ./config.json
? Path to the server metadata db directory (for users/apps etc) ./.db
? Path to the SSL private key in PEM format
/home/pmpetrov/Solid/node-solid-server/privkey.pem
? Path to the SSL certificate key in PEM format
/home/pmpetrov/Solid/node-solid-server/fullchain.pem
? Enable multi-user mode No
? Do you want to set up an email service? No
? A name for your server (not required, but will be presented
on your server's frontpage) localhost
? A description of your server (not required)
? A logo that represents you, your brand, or your server (not required)

 1015  ../bin/solid start --no-reject-unauthorized
```

[3] OpenACS Reverse Proxy Configuration for Solid

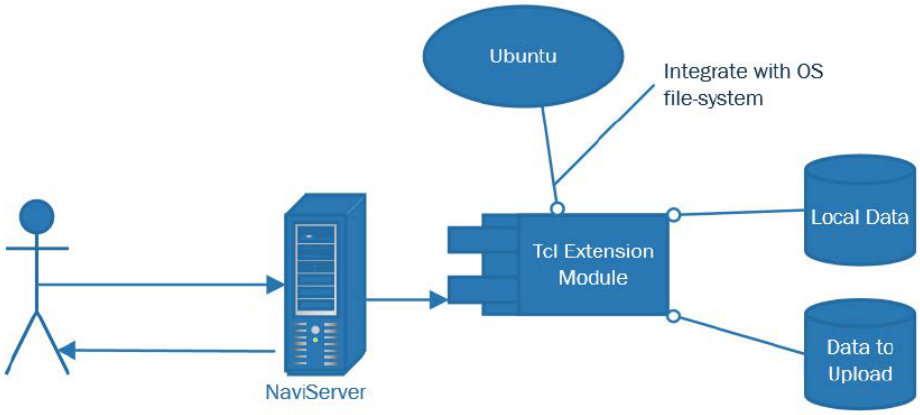
```
ns_section ns/server/${server}/module/revproxy {
    ns_param filters {
#ns_register_proc GET /* ::revproxy::check_for_solid

#Index page of Solid
        ns_register_filter postauth GET  /solid/* ::revproxy::upstream
        -target https://localhost:8443/
        -regsubs {{/solid/ ""}}
        ns_register_filter postauth POST /solid/* ::revproxy::upstream
        -target https://localhost:8443/ -regsubs {{/solid/ ""}}
#Inbox
ns_register_filter postauth GET  /inbox/ ::revproxy::upstream
 -target https://localhost:8443/inbox/ -regsubs {{/inbox/ ""}}

#profile and settings
ns_register_filter postauth GET  /profile/* ::revproxy::upstream
-target https://localhost:8443
ns_register_filter postauth GET  /settings/prefs.ttl ::revproxy::upstream
-target https://localhost:8443

#Login Popup window
ns_register_filter postauth GET /.well-known/solid/login ::revproxy::upstream
-target https://localhost:8443/.well-known/solid/login
-regsubs {{/.well-known/solid/login ""}}
ns_register_filter postauth GET /common/popup.html ::revproxy::upstream
-target https://localhost:8443/common/popup.html
-regsubs {{/common/popup.html ""}}
```

```
#Requests for private page
ns_register_filter postauth GET /private/* ::revproxy::upstream
 -target https://localhost:8443/
ns_register_filter postauth PUT /private/* ::revproxy::upstream
 -target https://localhost:8443/
ns_register_filter postauth PATCH /private/* ::revproxy::upstream
-target https://localhost:8443/
ns_register_filter postauth DELETE /private/* ::revproxy::upstream
-target https://localhost:8443/

#The requests for Public page
ns_register_filter postauth GET /public/* ::revproxy::upstream
 -target https://localhost:8443
ns_register_filter postauth PATCH /public/* ::revproxy::upstream
-target https://localhost:8443
ns_register_filter postauth PUT /public/* ::revproxy::upstream
 -target https://localhost:8443
ns_register_filter postauth POST /public/* ::revproxy::upstream
 -target https://localhost:8443
ns_register_filter postauth DELETE /public/* ::revproxy::upstream
-target https://localhost:8443

  #All the view elements and JS files required to process the
   requests
ns_register_filter postauth GET  /common/* ::revproxy::upstream
 -target https://127.0.0.1:8443
ns_register_filter postauth POST  /common/* ::revproxy::upstream
-target https://127.0.0.1:8443
}
}
```

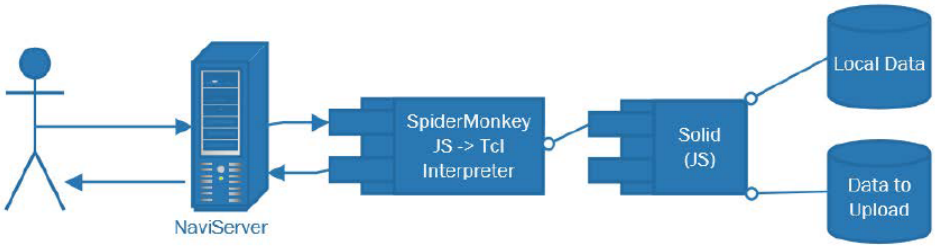
[KNOWN SOLID BUG]

If after running solid with the flag –no-reject-unauthorized you still get an error Unauthorized - delete the cache of the page.

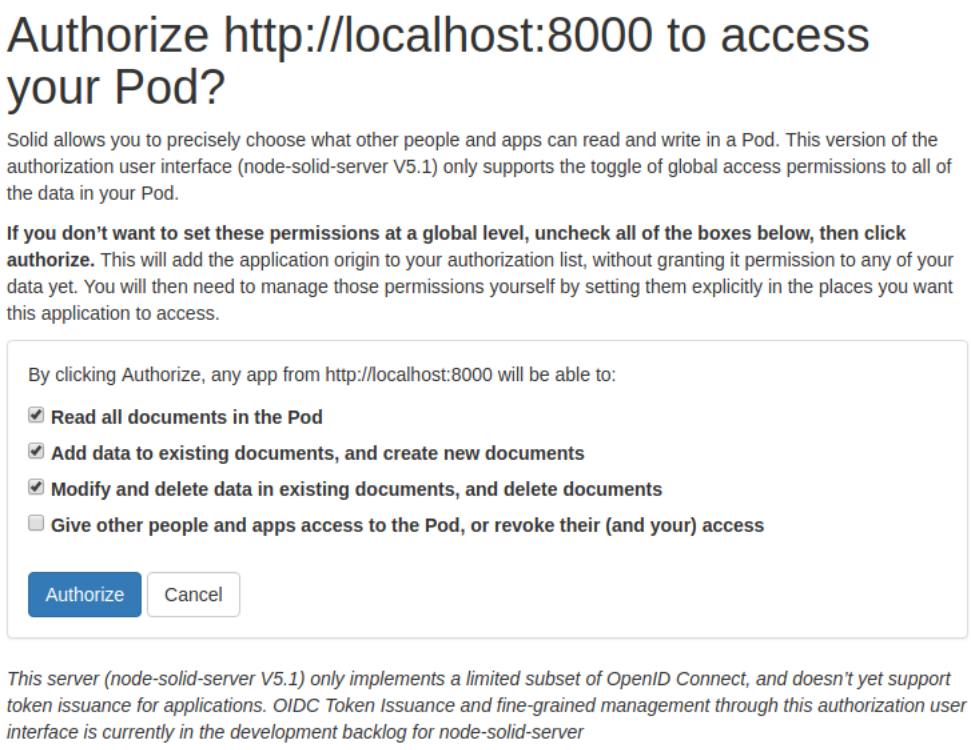
[Fig.1]



[Fig.2]



[Fig.3]


Authorize http://localhost:8000 to access your Pod?
Solid allows you to precisely choose what other people and apps can read and write in a Pod. This version of the authorization user interface (node-solid-server V5.1) only supports the toggle of global access permissions to all of the data in your Pod.
If you don't want to set these permissions at a global level, uncheck all of the boxes below, then click authorize. This will add the application origin to your authorization list, without granting it permission to any of your data yet. You will then need to manage those permissions yourself by setting them explicitly in the places you want this application to access.
By clicking Authorize, any app from http://localhost:8000 will be able to:
Read all documents in the Pod
Add data to existing documents, and create new documents
Modify and delete data in existing documents, and delete documents
Give other people and apps access to the Pod, or revoke their (and your) access
Authorize
Cancel
This server (node-solid-server V5.1) only implements a limited subset of OpenID Connect, and doesn't yet support token issuance for applications. OIDC Token Issuance and fine-grained management through this authorization user interface is currently in the development backlog for node-solid-server