CERN-Solid Code Investigation

Proof of Concept and Prospects

by

Jan Schill
schi@itu.dk

Supervisors:

Philippe Bonnet (ITU)
phbo@itu.dk

Maria Dimou (CERN)
maria.dimou@cern.ch

A thesis presented for the degree of
Master of Science

IT UNIVERSITY OF COPENHAGEN

Computer Science
IT University of Copenhagen
Denmark
01.06.2021

Table of Contents

Introduction

1 Context

Related Work

2 Background

3 Indico

4 Solid

Investigation

5 Proof of Concepts

One main part of the investigation into the CERN-Solid collaboration is the development of a proof of concept (POC). The POC contains the creation of two independent software modules in an existing system from European Organization for Nuclear Research (CERN). These software modules should show how it is to develop with the Solid principles in mind and to the Solid standard.

The goal of these modules is the symbiosis of decentralized stored data in a highly functional system without comprising its performance, security, or usability.

5.1 POC 1: Commenting Module for Events in Indico

The first POC is supposed to enrich the Indico system with some sort of Solid-based content. With the product owner and chief developer of Indico, the CERN-Solid project manager and a Solid developer it was decided a commenting module for Indico events is an adequate solution to include data from an external storage entity namely a data pod. The ability to allow users of Indico to leave a comment on an event, which then lives in a data pod completely controlled by the author of the comment was concluded to be an attractive feature for Indico.

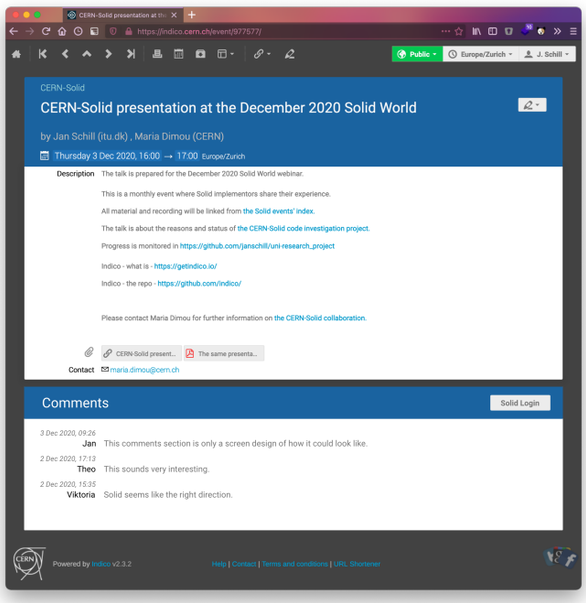


Fig. 1: User interface showing the comment module.

Design

For the implementation of this module several design decisions had to be made. From the fundamental choice of the module running on the client device or be computed on the server and then propagated to the client afterwards or even with a microservice proxying all traffic through it to enable Solid without changing Indico. Other design challenges were around how to protect the resources holding the comment information. These resources reside on the external data pod and need to be fetched from the application and read by other agents. Can access control lists (ACLs) be configured to allow the specific use-case?

Client- versus Server-Side versus Microservice

When an agent browses to a running instance of Indico most of the functionality is being prepared on the server hosting Indico. It retrieves the specific request, builds the Hypertext Markup Language (HTML), and sends it to the user. For Indico most of the functionality is built with Python and the web framework Flask. Sometimes functionality needs to be closer to the user, an example is dynamic rendering of Document Object Model (DOM) elements. This is useful when new data needs to be shown right away without getting the blank white screen on a page reload. Indico does send JavaScript, which is used for client-side features, but it focuses on keeping most its features on the server.

To make the right decision if the module should be primarily developed for the client- or server-side or even as a microservice, a list of requirements to the module had to be defined. With the defined requirements in place it had to be figured out how much functionality can be extracted from existing libraries and how much needed to be implemented with the new module. Implementing existing functionality for a new programming language would defeat the POC's purpose of showing how an existing software could work with the Solid principles.

The rudimentary set of features to enable commenting for users in Indico while saving the data in a data pod includes:

1. Authentication with a Solid identity provider (IDP)
2. (Authenticated) Requests to a data pod
3. Parsing of structured data (Linked Data)

Client Approach

The module runs in the browser and is therefore written in JavaScript. A programming language which compiles to JavaScript, such as TypeScript, is also possible. This means Indico remains mostly untouched, but would have to serve the needed JavaScript to the client on traffic to an event endpoint where the comment module is integrated.

Problem	Solution
Language	JavaScript or TypeScript
Framework	Native JavaScript
Client	solid-client-js [**solid-client-js**]
Authentication	solid-client-authn-browser [**solid-client-authn-browser**]
RDF	solid-common-vocab-js [**solid-common-vocab-js**], rdflib.js [**rdflib.js**]

Table 1: Existing solutions to problems for a client approach.

The communication flow with the data pod and the module would happen primarily from the browser.

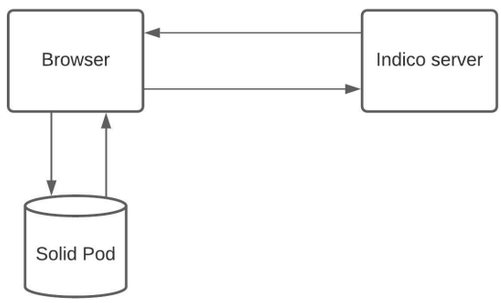


Fig. 2: Communication flow for a module developed on the client.

Microservice Approach

The microservice approach would allow developing the needed Solid logic on a separate service, which proxies all Solid related traffic through it an enable the Solid functionality. Most of the libraries from the client implementation can be used as well, as both developments would be written in JavaScript. Only the authentication flow would work a bit different.

The microservice module would handle take all requests aimed at the data pod and make it compliant with the Solid server. It would provide the client with the proper Solid OpenID Connect (Solid OIDC) flow to attach the access token to all authenticated requests.

Problem	Solution
Language	JavaScript or TypeScript
Framework	Node.js
Client	solid-client-js [**solid-client-js**]
Authentication	solid-client-authn-node [**solid-client-authn-node**]
RDF	solid-common-vocab-js [**solid-common-vocab-js**], rdflib.js [**rdflib.js**]

Table 2: Existing solutions to problems for a microservice approach.

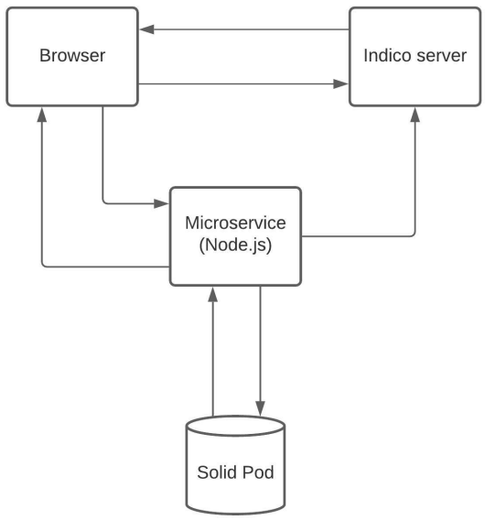


Fig. 3: Communication flow for a module developed as a microservice.

Server Approach

Goal of the server approach would be just like with the microservice approach to decouple the logic needed to work with Solid from the client and have it run on a server instance. The attractiveness for the server approach would be it could be fully integrated within Indico and be part of its Python code base. The major drawbacks are no direct Solid libraries written in Python exist to allow a seamless integration into the ecosystem.

Problem	Solution
Language	Python
Framework	Flask
Client	-
Authentication	pyoidc [**pyoidc**] missing DPoP
RDF	solid-common-vocab-js [**solid-common-vocab-js**], rdflib.js [**rdflib.js**]

Table 3: Existing solutions to problems for a server approach.

The authentication library pyoidc allows authenticating with OpenID Connect (OIDC) systems, but is missing a mandatory feature called Demonstrating Proof-of-Possession (DPoP), which is needed to make requests to protected resources on a data pod.

Comparison of the Different Approaches

Benefits from developing the module for the client:

- Necessary libraries exist (Major release for all basic Solid flows exist)
- Community support
- Programming effort for an minimum viable product (MVP) lowest
- Documentation on developing Solid apps in JavaScript exist

Single versus Multiple Resource(s) for Comments

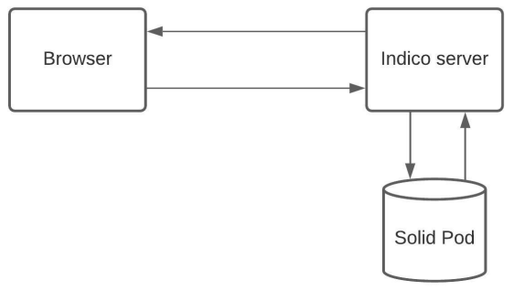


Fig. 4: Communication flow for a module developed on the server.

Library	Description
solid-client	A client library for accessing data stored in Solid Pods.
solid-client-authn	A set of libraries for authenticating to Solid identity servers:solid-client-authn-browser for use in a browser.solid-client-authn-node for use in Node.js.
vocab-common-rdf	A library providing convenience objects for many RDF-related identifiers, such as the Person and familyName identifiers from the Schema.org vocabulary from Google, Microsoft and Yahoo!
vocab-solid-common	A library providing convenience objects for many Solid-related identifiers.
vocab-inrupt-common	A library providing convenience objects for Inrupt-related identifiers.

Table 4: Existing solutions to problems for a server approach.

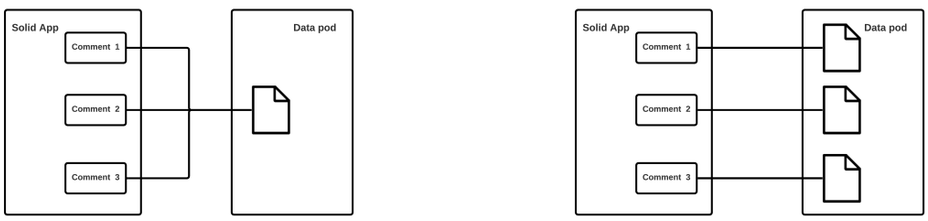
Storing the comments in Resource Description Framework (RDF) can be done in two ways: storing it in one file as a graph with a list of comments, or creating a file for every comment.

When fetching the container with all the comment resources the request returns a Turtle file describing the container, but not the actual content of the contained resources. The misconception of receiving the content as well, was thought to be a problem, as it was assumed an initial request to the container reading its child resources had to be made and then for each resource a request needed to be built to retrieve the resource. This overhead was later disproved as the application where this module is embedded maintains the list of resources to be fetched. Therefore, no manual building of requests to those resources had to be done. The next paragraph also lays out how this design would not work with the protection of the resources.

Protection on Resource

Every container and resource in Solid is protected with Web Access Control (WAC), which determines if specific agents, groups, or the world can have read, write, append, or control access. These control access modes are defined in ACL files. The Solid ACL inheritance algorithm looks for an ACL file attached to a specific resource, if it cannot find one it goes recursively up the file hierarchy and looks for ACLs on the containers. Indico allows two general types of protection *private* and *public* on its events. Public means open to everyone, no Indico account or any type of authorization is needed to see the event. Whereas private can be as fine-grained as only to specific agents or groups. A comment module is only valuable if the comments can be read by anyone and be written by authorized users.

In order for visitors of a private or public event in Indico to be able to see the comment, the comment's ACL needs to allow the public to read the resource. This can be achieved by using the *public* container, which comes with public-read by default on the Node Solid Server (NSS) or by creating a new container and setting the ACL with:



(a) TODO: caption (b) TODO: caption

Fig. 5: TODO: two approaches

Listing 3.1: TODO: Label caption

```
@prefix acl: <http://www.w3.org/ns/auth/acl#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

# ... Definition for owner

<#example-container-name>
    a acl:Authorization;
    acl:agentClass foaf:Agent;
    acl:accessTo <./>;
    acl:mode acl:Read.
```

Every resource in this container is by definition readable by the public – if not otherwise stated in a more detailed resource ACL. The above definition even allows the reading of the container's content, meaning a request to the container would yield a list of resources in the container. This becomes unpleasant if the Indico event is private and the comments for this Indico event should not be read by the world, which is entirely possible, when browsing to the location of a specific data pod and then looking into the public container.

To prevent a random agent to see the contents of a container, the container can be set to private, with the container's resources to still be public. This would allow everyone provided they have the Uniform Resource Locator (URL) to browse to the public resource and read it, but not look into the resource's parent container. To achieve this behavior with ACL, the container needs to just define its owner and no specific rules for the public, as WAC comes with a default private access control. Each child resource needs to define an ACL now, allowing public read. The container's ACL would look like the following with just a owner defined:

Listing 3.2: TODO: Label caption

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

<#owner>
    a acl:Authorization;
    acl:agent <https://janschill.net/profile/card#me>;
    acl:accessTo <./>;
    acl:default <./>;
    acl:mode acl:Read, acl:Write, acl:Control.
```

A child resource would allow public read with:

Listing 3.3: TODO: Label caption

```
@prefix : <#>.
@prefix acl: <http://www.w3.org/ns/auth/acl#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

# ... Definition for owner

:Read
    a acl:Authorization;
    acl:accessTo <test.txt>;
    acl:agentClass foaf:Agent;
    acl:mode n0:Read.
```

Another approach and the one implemented after iterating through the previous ones is to have the container's ACL resource define a default access mode for its child resources. This way one ACL only needs to be created on the container and all resources have proper access modes for public read and are not listed publicly in the container's description.

Listing 3.4: TODO: Label caption

```
@prefix acl: <http://www.w3.org/ns/auth/acl#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix target: <./>.

:ReadDefault
    a acl:Authorization;
    acl:default target:;
    acl:agentClass foaf:Agent;
    acl:mode acl:Read.
```

Preventing Resources From Unwanted Discovery

With the resources having proper access modes but being publicly readable a simple naming convention of taking the International Organization for Standardization (ISO) 8601 string and using it as a filename for the resources created on the data pod does not suffice – even though it is a good strategy when looking for a reliable naming convention to prevent duplication. Considering performance improvements such as pagination for future iterations of the module, which would require some sort of iterative indication, a combination of randomness, but also a order indicator it was settled for using universally unique identifier (UUID) plus the ISO 8601 string to form a filename.

Other ideas included hashing a random string with the timestamp to generate non-guessable filenames. The filename would need to use the same hash function to decipher the filename to figure out when the comment was generated. UUID is a reliable and easy to use system to generate *truly* globally unique strings.

Modification of Resource from Data Pod

Mitigation of Spam

Enabling user input in form of comment module without application authentication is a gateway to spam. Even though authentication with a Solid IDP is necessary, it does not hinder a malicious actor to create a multitude of Solid accounts and spam into the application. In the first iteration of the module only Solid authentication was integrated into the module and would therefore allow anyone with a Solid account to post comments and thus also spam the Indico event theoretically. In a second iteration of the module another authentication layer was added to mitigate spam from outside of Indico. For CERN's use-case an authenticated Indico session was enough. This adds an extra step to the comment process and it is ensured only registered Indico users can comment.

Giving Application Full Control of Data Pod

Integration with Indico

Storing Reference to Comments in Indico

Enforce Authenticated Session for Posting Comments

Evaluation

System Description

Context Diagram

Stakeholders

Drivers

Metrics

Levels

Components

Analysis

5.2 POC 2: Auto-Complete for Conference Registration in Indico

Design TODO: 1st iteration, save data in pod 2nd iteration, only pull data from pod

Modification of Resource from Data Pod

Payment on Input Fields

Performance of Large Conference

Availability of Crucial User Data

Integration with Indico

Bind to Dynamically Created Form

Evaluation

System Description

Context Diagram

Stakeholders

Drivers

Metrics

Levels

Components

Analysis

5.3 Deployment of Indico Instance

6 Comparison of Solid and Indico Design Principles

7 Challenges, Advantages, and Gaps of Existing Solid Solutions versus CERN Ones

8 Proceedings in the CERN-Solid Collaboration

8.1 Servers

Solution

Hosting

8.2 Applications

Conclusion