Granular Access to Policy-Governed Linked Data via Partial Server-Side Query

Joachim Van Herwegen and Ruben Verborgh

IDLab, Ghent University – imec, Dept. of Electronics and Information Systems
firstname.lastname@ugent.be

Abstract. The Solid Protocol defines a document-oriented interface to access Linked Data subject to usage control policies that define who can read or write it. Common use cases with different read and write granularities cannot easily be supported with a one-to-one mapping from HTTP interface to a document-based storage system. We propose an interpretation of the Solid Protocol with Derived Data Resources, which are a server-side selection of data from different underlying resources. This demo covers our implementation in the Community Solid Server, detailing its more granular data control. Our implementation can guide other proposals, and the evolution of the specification to support the growing demand of Solid use cases with divergent read/write patterns.

Keywords: Solid · RDF · policies

1 Introduction

The Solid ecosystem is a standards-based initiative to extend the application range of RDF from Linked Open Data to the entire private–public data spectrum. The current Solid Protocol [1] presents the latest draft for an HTTP interface to RDF data under access control. At present, this interface—mirroring general Web architecture—defines a document-based view into the RDF-based knowledge graph hosted by a single server instance, also known as "pod". Most implementations conceptually implement this specification as a one-to-one mapping of HTTP resources to documents in an underlying storage. For example, the resources `/a` and `/b` would, under the hood, correspond to independent RDF documents in a database or filesystem.

However, earlier research revealed that such a straightforward correspondence incorrectly assumes use cases to exhibit a symmetry in read and write access granularity [2]. For example, just because one application has *write* access to a resource `/bank-statements/2024/03/15/`, this does not imply that the *read* access is also organized by date; one app might only be allowed to read `leisure` transactions across all bank statements. Such cases are highly common, but hard to support in an exclusive one-to-one document mapping.

To this end, we propose an extended interpretation of the existing Solid Protocol to support a virtual kind of resource, analogously to how the HTTP specification transparently supports the interpretation of resources as static files

on a server or as dynamically generated resources by server-side technologies such as PHP or Node.js. A *Derived Data Resource* can be defined as a selection of data from other underlying resources, thereby allowing more granular server-side permissioning beyond the document level.

2 Derived Data Resources

As support for use cases with read/write granularity mismatches [2], we extend a Solid server to support *Derived Data Resources*. These are resources whose contents are not written directly to the server by a client. Instead, their contents are based by performing a query on one or more other resources on the server, enabling an asymmetry in defining usage control policies. The query needs to happen server-side for confidentiality reasons, as the client might have read access to the query result, but not to the sources contributing to that result.

In this demo we showcase one way derived resources could be defined and interpreted on a Solid server. To do this we have extended the Community Solid Server [4], which is a Solid server designed specifically to be easily extensible to support research such as this. The new component can be found at `https://github.com/SolidLabResearch/derived-resources-component`.

2.1 Defining a derived resource

To define a derived resource, our implementation requires 3 parameters:

- A *template*, defining the URL template that needs to be matched.
- One or more *selectors*, which are the input resources.
- A *filter*, to pick the necessary parts from the input.

When a request is made to a derived resource, it goes through those three steps to generate a response. The component has been designed so that any of these parts can easily be replaced with a different implementation, so you could define a new way to filter without having to change how selectors work, for example.

Templates As template, the component expects a URL. When this URL is dereferenced, the result of the derived resource is returned.

Besides standard URLs, there is also support for URI templates as defined in RFC 6570 [3], such as `http://localhost:3000/foo/{bar}`, where a value will be assigned to `bar` based on the contents of the actual URL. These variable mappings are then passed to the selectors and filter for reuse there.

Selectors Selectors are the URLs of one or more RDF resources that already exist on the server. The contents of all these resources will then be merged into a single RDF store and passed to the filter.

Instead of an exact URL location of a selector, the selector can also contain a pattern defined using `*` and `**` wildcards, which follow a common path expansion pattern[1]. All resources on the server that match this pattern will be used.

Filters To filter the selected data we make use of SPARQL CONSTRUCT queries, executed over the RDF store generated in the previous step. The result of this query is what is returned when a derived resource is being accessed.

To support the variables generated by matching a URI template, values can be injected into the SPARQL query. To do this, the variable names should be surrounded with two `$`. Using the example above, `$bar$` would be replaced with the actual value of the URL.

2.2 Where to define a derived resource

Solid resources can have a description resource, also commonly called the metadata of a resource. We make use of this metadata to store the necessary information that defines a derived resource, using a custom ontology.

3 Demonstration

The README[2] details the instructions on how to run the demonstration. It includes a script to start a server with several pre-defined example resources.

```
@prefix derived: <urn:npm:solid:derived-resources:> .
<> derived:derivedResource [
     derived:template "template/{var}";
     derived:selector </selectors/data>;
     derived:filter </filters/var>
   ].
```

Listing 1. Metadata defining a derived resource

Listing 1 shows a sample of the example metadata that defines such a derived resource, which can be found at `http://localhost:3000/.meta` after starting the demo server. There we can see all the fields that were discussed in Section 2.1. The template is a string as the full URL is determined by concatenating its value to the URL of the resource this metadata belongs to. The other fields reference existing resources, where the selector contains the input data, and the filter contains a SPARQL query to execute on that data. Combining all of the above, after starting the server, we can access a derived resource by going to `http://localhost:3000/foaf:knows`, which will result in an RDF dataset only returning the `foaf:knows` triple. This is just one of the examples found in the repository.

[1] `https://git-scm.com/docs/gitignore`

[2] `https://github.com/SolidLabResearch/derived-resources-component/blob/main/README.md`

4 Conclusion

Our new components can help Solid specification writers and implementers by showing how support for different read/write granularities could be implemented without changes to the protocol. By allowing users to create resources that depend on others, they can update their data in one place and still share different parts of it in multiple different places. This way, they no longer have to duplicate data to achieve the same goal.

Derived resources are defined through three different parts: the pattern determines where to find the derived resource the selector determines the input data, and the filter determines the query to perform on the input data.

While the component is an extension of the Community Solid Server, its design allows for this component itself to be further extended for purposes of research of specification design. For example, future work could examine the impact of using reasoning as a filter instead of the current SPARQL proposal.

This demo exemplifies a solution direction within the current document-based entry point of a Solid server. Other solutions include building different entry points into the Protocol, such as query-based mechanisms to make data selections of varying granularities [5]. Should the Solid Protocol evolve to have such functionality built-in, this solution can thus serve as a comparison point for evidence-based research into how the Solid protocol can best be modified to support this functionality.

Acknowledgements The research in this paper was supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10).

References

1. Capadisli, S., Berners-Lee, T., Verborgh, R., Kjernsmo, K.: Solid protocol. Editor's draft, W3C Solid Community Group (Dec 2022), `https://solidproject.org/TR/2022/protocol-20221231`
2. Dedecker, R., Slabbinck, W., Wright, J., Hochstenbach, P., Colpaert, P., Verborgh, R.: What's in a pod? – a knowledge graph interpretation for the Solid ecosystem. In: Proceedings of the 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs. CEUR Workshop Proceedings, vol. 3279, pp. 81–96 (Oct 2022), `https://solidlabresearch.github.io/WhatsInAPod/`
3. Fielding, R.T., Nottingham, M., Orchard, D., Gregorio, J., Hadley, M.: URI Template. RFC 6570 (Mar 2012). https://doi.org/10.17487/RFC6570, `https://www.rfc-editor.org/info/rfc6570`
4. Van Herwegen, J., et al.: Community Solid Server (Feb 2024). https://doi.org/10.5281/zenodo.7595116
5. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. Journal of Web Semantics **37–38**, 184–206 (Mar 2016). https://doi.org/10.1016/j.websem.2016.03.003