SPARQL-parser: Query Rewriting for Authorization in the semantic.works Microservice Framework

Tom De Nies[1], Aad Versteden[1]

[1]*redpencil.io*

Abstract

In this paper, we demonstrate SPARQL-parser, the latest iteration of our authorization service in the semantic.works framework, which provides a developer-friendly way to create applications and microservices using Linked Data. SPARQL-parser allows developers to query the database transparently, without having to consider access rights for the current user. This service rewrites any SPARQL query based on its session headers, and makes use of a graph-based data division strategy for separate user groups. This accommodates the use of a combination of public and private data, without adding complexity to the application logic. In addition, the service provides delta notifications after executing INSERT/DELETE queries, which can be subscribed to by other microservices, allowing developers to build an reactive, data-driven application.

1. Introduction

The semantic.works platform [1] enables building applications using Linked Data at the core. This means that all data is Linked Data from the start. This has many advantages, the foremost of which is that it makes for easy, conversion-free dissemination and integration with other applications. It does, however, come with its own challenges. While Linked Data is often considered as open data, in realistic scenarios it is often interlaced with private as well as public data. With many of our projects dealing with governmental data, where public documents are mixed with private/sensitive information such as contracts and digitally signed legislation, the need for an authorization microservice in the architecture is clear.

In [2], we presented a previous iteration of such a microservice: *mu-authorization*, which faced a number of challenges: (timed) data distribution, data duplication, query execution overhead and search indexing speed. Additionally, configuration in the Elixir programming language was verbose and a bottleneck in terms of runtime, affecting query performance.

Additionally, one of its key features is the generation of so-called *deltas*, where any INSERT or DELETE on the database triggers a notification message, informing subscribed microservices of the changes made by the user (or service). This is a powerful concept, enabling reactive services to be built into an application. In the previous iteration, services were simply provided the URIs of resources that had changed, without listing the effective changes that were made. This required the reactive microservices to use additional querying, which affected performance.

Lastly, there was often a need to bypass the authorization component when executing queries without an active user session, e.g., in the context of a cronjob. This added code complexity, since these queries must include the correct graph statements, and introduced a potential security risk, since these services then have read/write access to the whole database, requiring additional development effort to properly shield their access points.

Now, we have rewritten the component as *SPARQL-parser*, with the following improvements:

1. Simplified configuration in LISP, allowing “scoped“ requests, where microservices can get their own access rights to specific graphs and resources, improving code clarity and security.
2. Delta notifications now include effective changes, instead of only all related resources as was the case in the previous version. This allows for more efficient handling of these deltas, especially in

SEMANTiCS'25: International Conference on Semantic Systems, September 3–5, 2025, Vienna, Austria

tom.de.nies@redpencil.io (T. De Nies); aad.versteden@redpencil.io (A. Versteden)

terms of caching and data distribution.
3. Improved performance, especially in read-only scenarios, which has significantly sped up search indexing.

2. Related Work

Authorization of SPARQL endpoints is by no means an unexplored problem. While a full overview of literature is impractical for this demonstration, surveys of related work exist in [3], [4], and [5]. Following the classification scheme of [3], our approach can be placed somewhere between *Role Based Access Control* and *Context Based Access Control*. While we do allow contextual access rules, in practice the complexity is often limited to the user role, in order to preserve query efficiency. When used as a standalone entry point, our approach is closely related in concept to an "authorization proxy", as described in [6]. However, the main difference is that SPARQL-parser is meant to exist in an ecosystem of closely related microservices, where it provides added value thanks to its emission of delta events and its transparency to other microservices when reading and writing data.

3. SPARQL-parser

In Figure 1, we provide a high-level overview of a typical semantic.works application stack. SPARQL-parser is part of the "Store & Sync" module, which handles all requests coming from the microservices (and indirectly: the frontend).

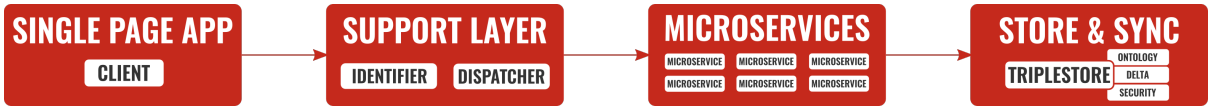

Figure 1: High-level overview of a semantic.works application. SPARQL-parser is part of the Store & Sync module.

The general principle of our authorization approach as described in [2] has not changed: data in the triplestore is organized into graphs, and the read access to these graphs is restricted to a certain set of criteria for each user, usually association with a certain group and/or role. Additionally, the write access is restricted even further to a specified set of resource types and predicates. SPARQL-parser then rewrites incoming queries by adding the correct graphs based on the session information of the user and the access rights on the data, and forwards the rewritten query to the triplestore. Paired with identification & authentication services, this allows the application frontend and other microservices to transparently query the triplestore, without having to implement any custom authorization logic themselves.

A major feature of SPARQL-parser is the emission of delta messages. When a user (or service) writes data to the triplestore, SPARQL-parser generates a delta message, which includes the inserted quads, deleted quads, effective changes, and the allowed groups of the original request. This effectively allows a microservice to react on behalf of the user based on the changes they made to the data.

The code for SPARQL-parser is open source, available at https://github.com/mu-semtech/sparql-parser. A basic configuration is structured as follows (omitting boilerplate code):

```
(define-graph public ("http://mu.semte.ch/graphs/public")
  (_ -> _)) ; public allows ANY TYPE -> ANY PREDICATE
            ; in the direction of the arrow

(define-graph privatebooks ("http://mu.semte.ch/graphs/privatebooks/")
  ("schema:Book"
   -> "schema:genre"
   -> "dct:creator"
   -> "dct:issued"))
```

```
(supply-allowed-group "public")

(grant (read)
       :to-graph public
       :for-allowed-group "public")

(supply-allowed-group "privatebooks"
  :query "PREFIX org: <http://www.w3.org/ns/org#>
        PREFIX ext: <http://mu.semte.ch/vocabularies/ext/>
          PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
          SELECT DISTINCT ?role_label WHERE {
            <SESSION_ID> ext:sessionMembership / org:role ?role .
            ?role skos:notation ?role_label .
              VALUES ?role { <https://example.org/privateBookReader> } .
          }"
  :parameters ("role_label"))

(grant (read)
        :to-graph privatebooks
        :for-allowed-group "privatebooks")

(with-scope "service:privatebook-service"
  (grant (write)
         :to-graph privatebooks
         :for-allowed-group "public"))
```

In this example, a *public* and *private* graph are defined, which will house publicly available books and private ones, respectively. The `(_ -> _)` statement means that the public graph could contain any resource and predicate. On the other hand, the privatebooks graph defined below is restricted to a specific set of resources with type `schema:Book` with specific predicates.

The statement `(supply-allowed-group "public")` supplies the group "public" to any incoming request, and the `grant (read)` statement provides read access to the public graph for this group. This means that any unauthenticated users will have access to the public graph of the triplestore, which is realistic in a scenario where public data is disseminated.

Below that, we see that the `supply-allowed-group "privatebooks"` statement is more complex, and includes a template query that checks whether the session is associated with a user that has a role that allows the user to read private books. Additionally, the `role_label` parameter will be appended to the `http://mu.semte.ch/graphs/privatebooks/` URL for every query, ensuring data separation for each user role.

Based on these allowed groups, many optimizations become possible, since we know that the same allowed groups will have access to the same data. For example, in our real-world implementations, we use this for caching, search indexing, and push-updates.

Finally, we see a service that receives scoped write access to the privatebooks graph, even when the only allowed group it has is "public". This means that this service will be able to write all resource types and predicates specified for the privatebooks graph, into this graph and this graph only. This allows the service to work without the need for an active session, enabling it to read & write data while reacting to deltas, or when executing cronjobs, for example.

4. Demonstration

Our demonstrator is hosted at https://authorization-demo.redpencil.io. This small web application showcases the two main features of SPARQL-parser: *access rights* and *reactive services*, as well as more generic features of the semantic.works framework. All code is open source, available at https://github.com/tdn/app-auth-demo/ (backend) and https://github.com/tdn/frontend-auth-demo (frontend).

The backend code is relevant to this paper in particular, as it includes a more complete authorization configuration that the example listed in Section 3.

When accessing the demonstrator webpage, the user is provided with a YASGUI[7] query editor, through which a portion of publicly available data can be queried right away. The public graph was seeded with a number of RDF resources with the type ‘schema:Book‘, and a number of properties such as author, genre, issue date, wikidata reference, etc.

However, when using the `mock-login` route, we can simulate an authenticated user, for whom private data becomes available. As the demonstrator explains, the same SPARQL query will yield different results, depending on whether the user is authenticated or not.

To demonstrate write access, we allow authenticated users to store “favorites“, either through the user interface or by writing a SPARQL INSERT query. Note that when the user attempts to write other data, this will be stopped by SPARQL-parser, and an error will be returned by the SPARQL endpoint. In other words, users only see what they are allowed to see, and can only write what they are allowed to write.

Finally, to showcase scoped write access, we created a QR-code generating service, which reacts on any “favorites“ write actions being added to the system by generating a QR code for the URI of the favorited book, and storing it into the database. This demonstrates that even though the user has no writing rights for these types of resources, a scoped service can still get the necessary rights to do so.

5. Discussion & Future Work

Since its first stable release in 2024, we have deployed SPARQL-parser in multiple production environments such as Rollvolet’s CRM[1], the Local Mandatee management app of LBLOD[2], the Veeakker webshop[3], and the Flemish Governments decisionmaking support platform Kaleidos[4].

While we have not noticed significant *write* speed improvements (or deteriorations) in these apps compared to *mu-authorization*, we did measure a *read* speed increase in Kaleidos. Its elasticsearch indexer, which performs thousands of read queries in rapid succession, has sped up by a factor of 3 (from approx. 36 hours to approx. 12 hours for a full reindex).

In future work, we aim to improve SPARQL-parser even further by working on read-only resource constraints for graphs, since read access is now generalized per graph, regardless of resource type. This will create less load on the triplestore and improve caching. Additionally, we want to extend graph specifications and scopes for enhanced constraints, allowing enhanced model validation and basic forward reasoning. Lastly, while the current implementation supports read/write to multiple triplestores, we are looking into the possibility to have SPARQL-parser act as a Linked Data proxy to other types of data stores, that are not necessarily triplestores.

Declaration on Generative AI

The authors have not employed any Generative AI tools in this paper.

References

[1] A. Versteden, E. Pauwels, A. Papantoniou, An ecosystem of user-facing microservices supported by semantic models., USEWOD-PROFILES@ ESWC 1362 (2015) 12–21.

[2] T. De Nies, A. Versteden, E. Pauwels, J. Delaure, Combining public and private linked data through graph-based authorization profiles in the semantic. works framework., in: QuWeDa/MEPDaW@ ISWC, 2023, pp. 22–25.

[1] https://github.com/rollvolet/app-crm/

[2] https://github.com/lblod/app-lokaal-mandatenbeheer/

[3] https://github.com/veeakker/app-veeakker-webshop/

[4] https://github.com/kanselarij-vlaanderen/app-kaleidos/

[3] S. Kirrane, A. Mileo, S. Decker, Access control and the resource description framework: A survey, Semantic Web 8 (2016) 1–42. doi:10.3233/SW-160236.
[4] V. Zdraveski, D. Trajanov, R. Stojanov, S. Stojanova, M. Jovanovik, Ranking semantic web authorization systems, Semantic Web (2017).
[5] T. G. da Silva, Access control in linked data archives (2023).
[6] R. Stojanov, M. Jovanovik, Authorization proxy for sparql endpoints, in: ICT Innovations 2017, Springer International Publishing, Cham, 2017, pp. 205–218.
[7] L. Rietveld, R. Hoekstra, Yasgui: not just another sparql client, in: Extended Semantic Web Conference, Springer, 2013, pp. 78–86.