Bringing IDE Support to JSON-LD with the Language Server Protocol

Arthur Vercruysse[1], Julián Andrés Rojas[1] and Pieter Colpaert[1]

[1]*IDLab, Department of Electronics and Information Systems, Ghent University – imec*

Abstract

JSON-LD is a popular data format used to describe and share semantic data on the web. However, creating and editing JSON-LD documents can be a challenging task, especially when dealing with complex contexts that include many properties. The existing JSON editing functionality may not suffice for developers, and a JSON-LD editor could greatly enhance their experience. In this paper, we introduce a JSON-LD Language Server based on the Language Server Protocol (LSP) that empowers text editors compatible with the LSP (e.g., Visual Studio Code and NeoVim) with IDE functionality, including autocompletion suggestions based on the defined context, semantic highlighting and renaming identifiers inside the document. We believe that a JSON-LD LSP will enhance developer ergonomics and promote its adoption. Moreover, we see high potential for additional features that can be added such as hovering, go-to-definition and code actions like flattening or structuring of JSON-LD documents.

Keywords

Linked Data, JSON-LD, Developer UX, Tool, Language Server

1. Introduction

JSON-LD is a data serialization format that is used to describe semi-structured data on the web[1]. It adds semantic information to JSON, among others with the *@context* property, pointing to a JSON-LD context. This context specifies how to map properties to predicates, called aliases.

With its increasing popularity, contexts in JSON-LD also increase in complexity. For example, the Flemish (Belgium) OSLO initiative is working to achieve interoperability by building semantic standards for different stakeholders such as government, industry and academia[1]. They focus on standardizing JSON-LD context files for their vocabularies and application profiles. An example is the Flemish codex application profile, the context defined for this profile enables JSON objects with Dutch properties to be interpreted as linked data.

Another example where JSON-LD contexts are extensively used is the semantic dependency injection framework ComponentsJs[2]. ComponentsJs allows Javascript packages to define JSON-LD contexts that describe the package, these contexts are then used to configure and instantiate an application.

[1]OSLO - Open Standaarden voor Linkende Organisaties

Creating new JSON-LD documents is a difficult task, before the developer knows the defined aliases, he has to scrupulously explore the context and remember the defined names. Accidental typos can result in broken linked data and to the best of our knowledge, there are no tools in existence that could assist in preventing this issue. We show that using a language server for JSON-LD helps alleviate this problem and can also bring better developer ergonomics for experienced and new semantic web developers.

Similar efforts have been made before, but never for JSON-LD. For example, the Yasgui editor[2], a popular SPARQL human query interface, provides autocompletion based on the LOV API[3]. Plugins can also add autocompletion based on domain knowledge. These capabilities are built into the deployed editor and can only be used with Yasgui. Stardog created open-source LSPs for turtle, TRIG, SPARQL, and more, however, they focus only on correct syntax and keyword auto-completion[4].

In this paper, we introduce a JSON-LD language server implementation following the Language Server Protocol (LSP), a JSON RPC protocol designed by Microsoft[3]. LSP is designed to simplify the process of integrating language-specific logic into an editor, removing the need to implement language-specific logic into each editor, and reducing the $O(n * m)$ complexity to a much more manageable $O(n + m)$ complexity[5].

A full LSP implementation supplies the editor with IDE features, including autocompletion, diagnostics, formatting, identifier renaming, hovering, go-to-definition and generic code actions like *extract highlighted code to function*. Not all features have to be implemented, the Language Server and the editor communicate their capabilities and only use what is supported by both parties.

LSP source code and installation instructions are available on GitHub (MIT)[4].

2. Features of the JSON-LD language server

This section covers the implemented features, explaining what they are and how they benefit the developer.

- **Autocompletion** enables the developer to type less and thus make fewer mistakes. The created suggestions are derived from the defined context[5] and defined entity ids inside the document. This way developers don't have to remember aliases by heart and can more easily explore the context. An example can be seen in Figure 1. It should be possible to write *foaf:* and get autocompletion suggestions for all defined predicates in the *foaf* namespace, but this is not yet supported.
- **Diagnostics** represent errors and warnings in the file. Our language server only supports syntax errors but emitting a warning when an undefined property is encountered, can be an easy extension.
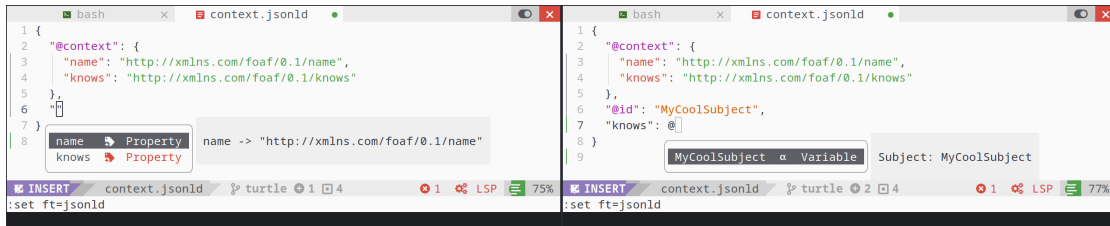
---

[2]Yasgui - SPARQL editor
[3]Microsoft: Language Server Protocol
[4]Github: https://github.com/ajuvercr/jsonld-lsp
[5]Only simple and explicit aliases are extracted from the context, context overloading or scoped contexts are not supported.

Figure 1: Screenshots showing completion functionality in NeoVim: the left side shows completion suggestions, and the right side also shows defined identifiers.
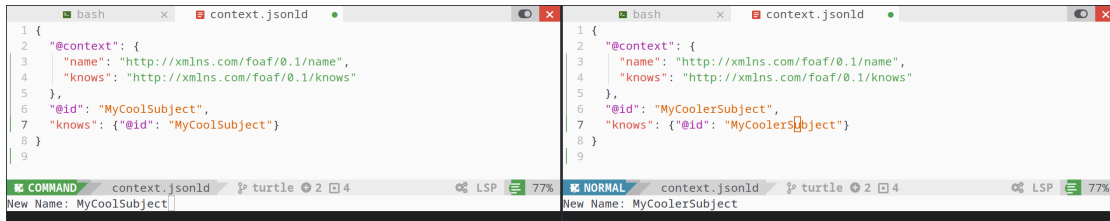
Figure 2: Screenshots showing renaming functionality in NeoVim: left NeoVim asks for the new name, right the subject is renamed.

- **Semantic Highlighting** differs from syntax highlighting. The syntax of JSON-LD is very easy, but semantic highlighting highlights keywords, identifiers and tokens differently. Other semantic highlighting functionality includes highlighting defined/undefined properties and highlighting entity ids and types, these are however not implemented.
- **Renaming**, in our LSP, allows the developer to rename defined identifiers and their references. An example of renaming an entity can be seen in Figure 2, first the editor will ask for a new name, then all occurrences are renamed.

3. Demo

Our demo presents an implementation of a JSON-LD Language Server according to the Language Server Protocol (LSP), developed using the Rust programming language based on a basic LSP implementation[6]. The full source code and installation instructions can be found on GitHub (MIT). A demo version of the LSP is available on the web. The full LSP is available as a VS Code extension and as a NeoVim LSP.

The Language server is compiled to WASM to be integrated inside the VSCode extension. This enables the extension to be a Web Extension[7], which are available in online Visual Studio Code instances like *github.dev*.

Extracting predicate mappings from the context is handled by the JSON-LD crate[8]. Some

---

[6]Tower LSP - Crate by Eyal Kalderon

[7]https://code.visualstudio.com/api/extension-guides/web-extensions

[8]JSON-LD - Crate by Timothée Haudebourg

parts of the JSON-LD crate can't compile to WASM, because of internal Rust requirements[9] and required some rewriting of the crate.

Parsing the JSON structure is done with the Chumsky crate[10]. This was no easy task because the moment the Language Server should suggest completion options, the document is no valid JSON anymore. This required a special internal JSON structure allowing parts of a dictionary to be invalid.

4. Conclusion

This JSON-LD LSP is an example of what impact developer tools can have on productivity and therefore also on adoption. The same ideas can be applied to other semantic data formats to help write but also, with hover functionality, to help understand the data more easily.

This demonstration merely scratches the surface of the vast capabilities of a JSON-LD LSP. By leveraging fully-interpreted contexts, the LSP can provide more contextually-relevant suggestions, taking into account context overloading and scoped contexts. Additionally, the LSP's functionality can be expanded to include the interpretation of referenced vocabularies, allowing completion for compacted predicates, like *foaf:knows*. Despite its current limitations, the LSP already enhances the developer experience by facilitating the creation and editing of JSON-LD documents.

On the development side, this demo contributes to the open-source community. There are not many examples of Language Servers written in Rust and compiled to WASM to be included in a Visual Studio Code extension. Hopefully, this implementation starts a cascade of developer tools for the semantic web community.

References

[1] M. Sporny, G. Kellogg, M. Lanthaler, JSON-LD 1.1", W3C Recommendation, W3C, 2020. Https://www.w3.org/TR/json-ld/.

[2] Taelman, Ruben and Van Herwegen, Joachim and Vander Sande, Miel and Verborgh, Ruben, Components.js : semantic dependency injection, SEMANTIC WEB 14 (2023) 135–153. URL: http://dx.doi.org/10.3233/SW-222945.

[3] P.-Y. Vandenbussche, G. Atemezing, M. Poveda-Villalón, B. Vatant, Linked open vocabularies (lov): A gateway to reusable semantic vocabularies on the web, Semantic Web 8 (2017) 437–452. doi:10.3233/SW-160213.

[4] Stardog - language servers github, 2008. URL: https://github.com/stardog-union/stardog-language-servers.

[5] H. Bünder, H. Kuchen, Towards multi-editor support for domain-specific languages utilizing the language server protocol, in: S. Hammoudi, L. F. Pires, B. Selić (Eds.), Model-Driven Engineering and Software Development, Springer International Publishing, Cham, 2020, pp. 225–245.

---

[9]Fetching a resource in WASM is not *Send* or *Sync*, which is required in *async* environments

[10]Chumsky - Crate by Joshua Barretto