LSP or How to unlock builtin IDE features in your favorite text editor

Introduction

Text editors are often described as lackluster compared to `Integrated development environments` (long for **IDE**) and their builtin features that can considerably enhanced the productivity when writing code. A good example to illustrate this point is the auto-completion which is enabled by default on most IDEs out there. Even if some people do not consider this feature as mandatory, it can be a important criteria for the tools one may want to choose to write code. However few years ago, Microsoft managed to find a way that can help implementing such feature in your favorite text editor and help it ascend to the level of an IDE. Its name is `Language Server Protocol`, also known as **LSP**. This article aims to give some background knowledge about LSP, what it consists of, its main uses and finally its current place.

Context and background

Microsoft and VSCode

In 2015, Microsoft announced the development of **Visual Studio Code**, their homemade open sourced text editor oriented on the programming side. However, the competition was already important on this market, with the presence of the almighties IDEs and fossils still widely used such as Emacs, Vim or Atom. Nonetheless, the team leading the project aimed for high objectives and planned the whole to scale well to make sure it will get an important share of the programmers on the long run. As such, they encountered multiple problems when concepting builtin features such as the one that is interesting us here, the auto-completion.

The M * N problem

There was a very annoying problem for companies promoting a new language and the communities built around them. Given M different languages (C, C++, Java, Lisp, etc) and N different editors (Emacs, Vim, Atom, etc), then there are **M * N** different tools to make every languages available on every editors used by programmers to write code.

As you can imagine, it can be very costly and very hard to develop even a naive solution of one language for only one editor. However with the introduction of the LSP, the editor does not even need to care about the language being used since all the data is seamlessly treated

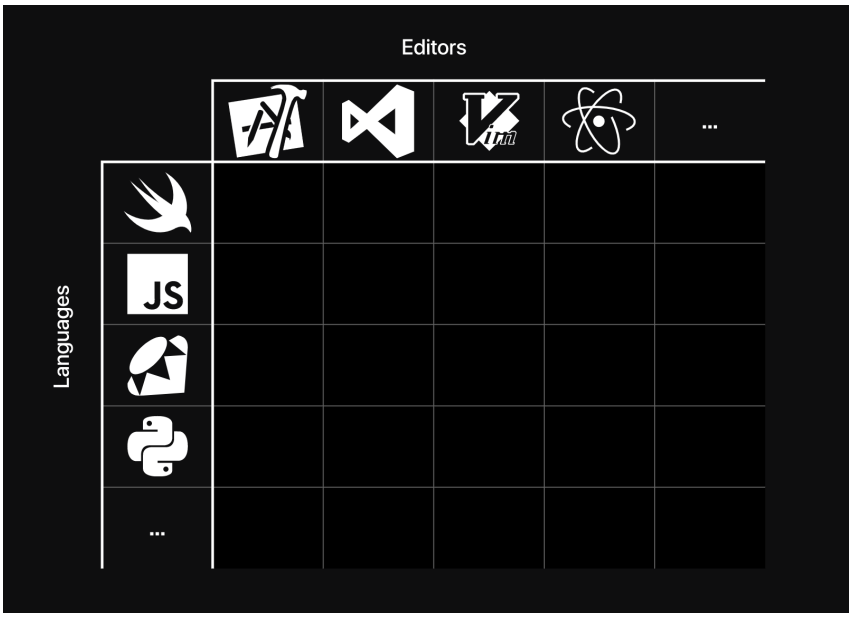

Figure 1: The infamous M * N problem (from freeCodeCamp)

in a different place. In fact, the editor does not think anymore, it only edits and sends some contexts to some servers. Of course, this solution only works if the given editor also offers an extension of the protocol.

A story that started with a compiler and finished with a communication protocol

Microsoft already tried to solve the M * N problem with their C# cross-platform compiler, codenamed **Roslyn**. At the opposite of traditional compilers, it was not a black box that would try to isolate itself as much as possible from the developpers. Indeed, the compiler was developed with the idea to be hacked by its users and let them extend its basic functionnalities. As a result, most of its APIs are available to work with and make it possible to develop tools such as source code analysis, which is the first basis for features like code completion.

Following Microsoft's efforts on their Roslyn compiler to bring cross-platform support of C# and .NET, a new project was born: the **Omnisharp project**. It is basically a simple server that leverages some integration toolings made on top of the Roslyn compiler. By communicating with it, it is possible for any editor that has integrates such server to access language-dependant features for any language without having to reimplement the complicated details every time.

The idea proved itself simple and successful among the communities of the .NET environment. Integrations for some well-known editors such as Emacs, Vim, Sublime Text or Atom can be found on their respective package distributions. However, the generic integration was still missing. The Omnisharp project was only a proof that such solution was possible. There was a need to standardized the protocol being used as well as the integration of the server

inside the editor, which Microsoft did with the LSP.

The protocol in itself

Overview

LSP is really simple to understand and require only some basic knowledge about networking. Consider you have two actors in process: a client, which can be programmer asking for auto-completion, and a server, which can provide auto-completion if a context is given to him. If this condition is satisfied, then only the communication between the two is missing, and that is where Microsoft and the LSP come into action. Here, the server designates either a local server exchanging `HTTP` messages using `TCP/IP` or a process that communicates though its `standard input and output`. Using custom version of the `JSON-RPC` protocol to send the data, the information is easily exchanged and the process is fast even compared to the builtin implementation that can be found in some IDEs.

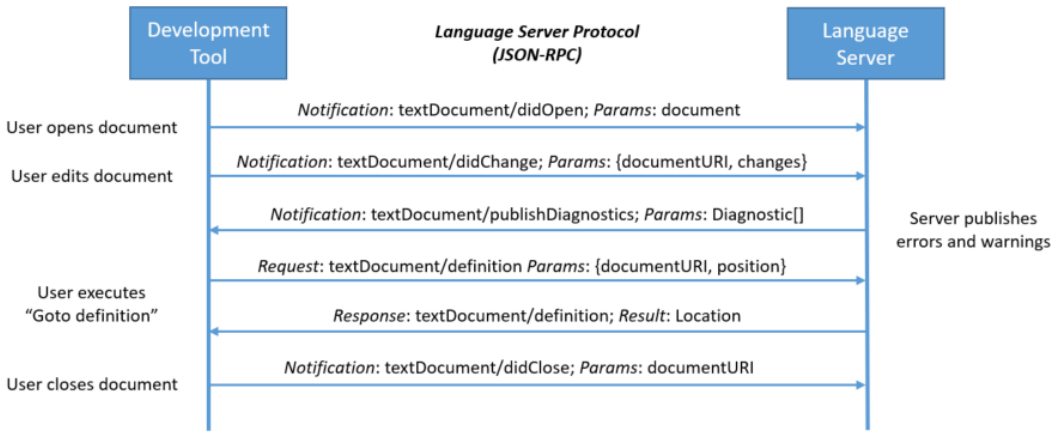

Figure 2: Explanation on what happens during the exchange between the client and the server (from the official documentation of microsoft)

As illustrated on the above image, the client sends lots of different notifications about various changes by contacting what could be compared to the endpoints of a REST API. Even if it's a huge shortcut to put it this way, the logic is still the same. As an example, the client will send a request to contact the method `didChange` each time it will save the file he is editing. That way, the server will be able to update the state of the same file he was watching to make sure he will not publish outdated reports after a request from the client.

For more precisions on the available endpoints, Microsoft offers in its specifications of the protocol a list of the classic ones that should be present for any implementation of the protocol.

The meaning of the messages exchanged during the process

Following the specifications of the protocol, the base protocol relies on two headers:

- `Content-Length`: the length of the content part in bytes.
- `Content-Type`: the type of the content in the body. This header is optional and by default set to 'application/vscode-jsonrpc; charset=utf-8'

The body of the request is a JSON file that contains the informations required by the server. The following request is a generic example that asks to the server to apply the "completion" feature:

```
HTTP / 2.0
Content-Length: 83

{
    "jsonrpc" : "2.0",
    "method": "completion",
    "params": { "file": "foo.txt", "line": 10, "begin": 6, "end": 8},
    "id": 1
}
```

As you may have guessed, the server will try to provide code completion for the word of length 4 at line 10 of the file foo.txt. If the prefix were to be "is", The answer from the server could be:

```
200 / OK
Content-Length:

{
"jsonrpc": "2.0",
"result": {
    "completions":[
        {
            "value": "isBoolean",
            "type": "variable"
        },
        {
            "value": "isDigit",
            "type": "function"
        }
    ]},
    "id": 1
}
```

After processing the file `foo.txt`, the language server found two possible completions: a **variable** `isBoolean` and a **function** `isDigit`. If it has no method named `completion`, the backend would have send a similar response but with an error field instead of the `result` one:

```
"error": { "code": -32601, "message": "no such method 'completion'" }
```

Pros and Cons of the LSP

As illustrated with the previous example, what is exchanged between the client and the server is fairly simple to understand and easy to deal with. Integrating a server supporting LSP seems way more easier than developping several times the same extension for each language. Moreover, there's no need to stick anymore to PyChar for developping python and Intellij to write java code. Both can be done with LSP in your favorite text editor if it has an integration with LSP.

There are still some downsides with this solution. The main one is that there will always be the same number of servers running in background as they are languages using LSP. In a same way, a server is bound to a tool which means that if a programmer is using both Emacs and VSCode to write some C++ code, then this person will need to run two LSP servers for C++ **at the same time**. This last con also means that it is not possible for now to use the LSP for tools integrated in the cloud. Finally, some editors do not support multiple servers for the same language. This can be troublesome since a custom implementation would not be able to be used at the same time at the major one. As a result, LSP possesses some negative points. Even though, this trade-off is not that bad considering the kind of a mess this problem was for both the companies and the users. Finally, this solution manages to save considerable ammount of time and costs and help developping new languages through the growth of the communities built around them.

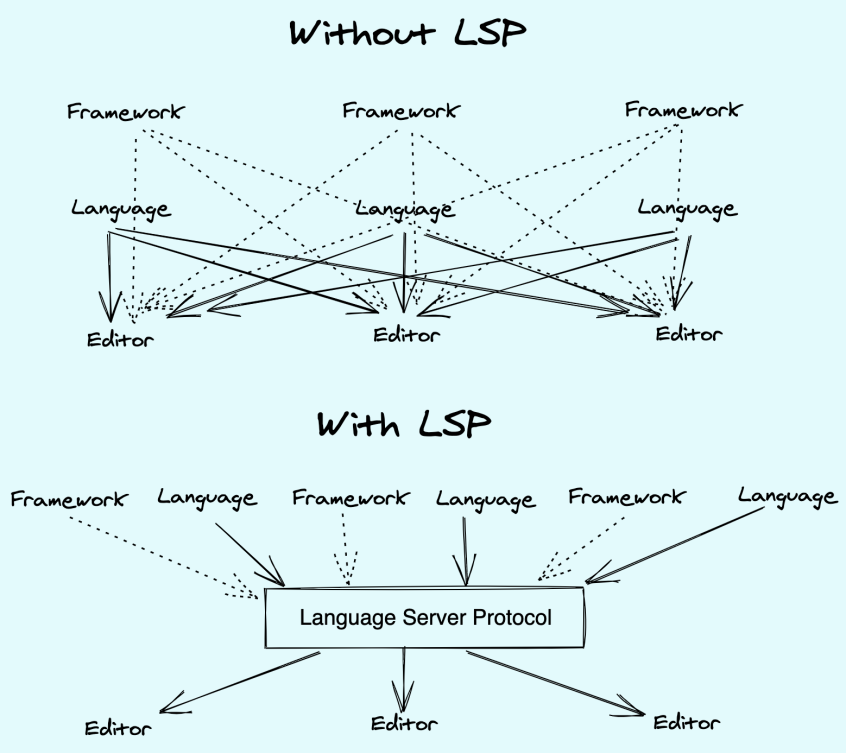

Figure 3: What the LSP manages to clean (from swyx.io)

What it can offer

- Amazing IDEish features

What's amazing with the LSP is that it has many possible use cases other than code completion. Among the listed implementations on the official website, most of them have five main uses of the LSP:

- `Hovering`: complementary information such as documentation, uses or signature function appearing when placing the cursor on a given word,
- `Goto definitions`: find the definitions of a symbol, for example a variable, a function or a class,
- `Workspace Symbols`: offers a list of all the matches within the workspace of a given query string.
- `Find references`: search in the workspace for all the uses of a given symbol.
- `Diagnostics`: the backend language server handles diagnostics on either a whole project or a specific file. A diagnostic can be for example checking that no variable is written in uppercase. As a result, this feature can be a good support on developping tools for spell checking or coding style reports.

The above features are not the only ones described in the specifications. More advanced ones code lens (somewhat hidden source code) or renaming functions, can still sometimes be implemented in the language servers even if it is less likely.

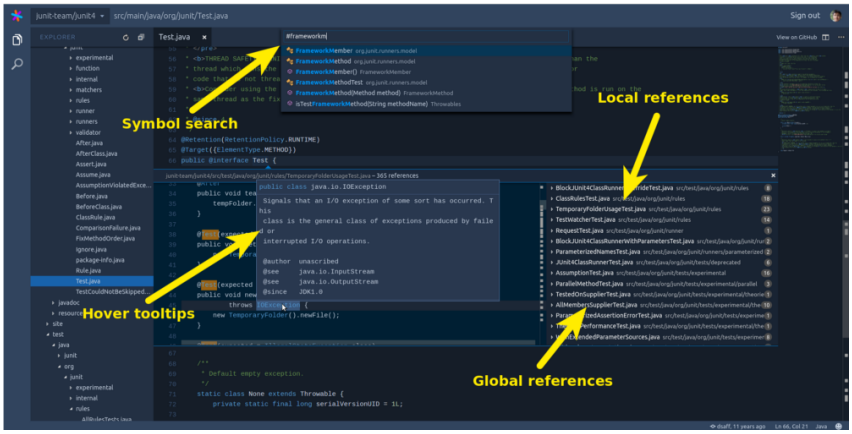

Figure 4: Some features offered by the LSP implementation of VSCode (from Sourcegraph)

- The possibility to write your own language server

Since the LSP is very easy to work with, writing your own language server with customized validations is also. There are many APIs available to getting started with the language of your choice, may it be Python, C#, Java, Lisp or many others. However, most of the articles I found to write this article were using Typescript because this is both the language used to develop VSCode extensions, and the one used in the tutorial proposed by Microsoft. If you are interested in trying to write one, I would recommand these readings:

- VSCode official language server extension guide: a simple tutorial to write a language server built on a VSCode extension in Typescript.
- Extending a client with the language server protocol by Florian Rappl: a detailed explanation on the calls made on the Typescript API. It is followed with a detailed demo on how to implement some simple functionalities of a language server in Typescript.
- Language Server Protocol tutorial: From VSCode to Vim by Jeremy Greer: an article about the implementation of a language server that blacklist some words, and how its author made it works for several editors without modifying the server.
- Java implementation of a language server maintained by Eclipse: a github repository that proposes an implementation of an LSP API in Java.

The LSP since then

Quickly after the first integration on VSCode, many language servers as well as extensions to editors to integrate LSP support were developped. As of now, there are more than 140 maintained language servers listed on Microsoft's official page on LSP. Even old languages such as COBOL found people among their communities to develop a version of LSP and bring support of modern features to them. As a result, it becomes way easier to learn them and give them a relative second youth by making them more accessible to newcommers.

Concerning the editors, some have builtin integration such as VSCode and NeoVim, others need complementary extension like Emacs or Atom which and others do not support it at all like Notepad++.

A list of all the implementations and the available clients driven by the community built around the LSP can be found by clicking here. As you may have seen, this list also includes in the clients section IDEs like the Jetbrains Product or Eclipse, which were not at all the target of the LSP but in the end found some good use of the LSP. The main reason is because it becomes easier to develop new extensions and it may offer approach that were not available if the devs were to stick only on the builtin functionalities of the IDEs.

Conclusion

The Language Server Protocol is one of these tools that fixes very annoying issues in a very simple way. It's a blessing considering all the features it can bring to many different tools, may it be a text editor or an IDE. However, it is still far from being perfect and there are many possible upgrades that are very anticipated by the community it has build over the past few years.

Thank you for reading this small article, I hope you learned something new today through it!

Sources

- The Impact of the Language ServerProtocol on Textual Domain-Specific Languages: https://www.scitepress.org/Papers/2019/75563/75563.pdf

- Microsoft's offical webpage on the LPS: https://microsoft.github.io/language-server-protocol/
- A bird's view on Language Servers: https://blogs.itemis.com/en/a-birds-view-on-language-servers
- VSCode language server extension guide: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide
- Emacs integration for LSP: https://emacs-lsp.github.io/lsp-mode/
- How the Language Server Protocol Affects the Future of IDEs: https://www.freecodecamp.org/news/language-server-protocol-and-the-future-of-ide/