Language Server Protocol for ExtendJ

Joakim Ericson
D09, Lund University, Sweden
ada09jer@student.lu.se

Abstract

Integrated Development Environments (IDEs) are popular tools and while they have their benefits, such as diagnostic messages in the editor, code completion, or debugging. However a drawback of most modern IDEs is the fact that they only work with a handful of languages, that they are developed for. The Language Server Protocol (LSP) was developed to extract some of the language related functionality of an IDE [1]and make it available for any editing software that would implement a client for the LSP. Currently there are two major language servers for Java. One using the official Javac compiler, and the other using Eclipse's jdt compiler. While both of these solutions work fine for working with the standard features of Java, if there were new features of Java that one wanted to test and still have full support by a language server, that would not work currently. Thus we propose to create one using the compiler ExtendJ[2], that is developed at Lund University. We decided that we wanted to implement a language server using ExtendJ due to the fact that it is designed to be easy to extend Java with new features. Since this proposal combines ExtendJ with the LSP any existing editor, such as vim or sublime, both of which has a working client would be usable. And can provide helpful features such as code completion and other such features that an IDE would traditionally provide. While adding the extensibility of ExtendJ to the mix.

1 Introduction

Integrated Development Environments (IDEs) provides useful features for their users, such as code completion or type checking. To bring this type of functionality to more tools Microsoft, Codenvy and Red Hat collaborated on a protocol, the Language Server Protocol (LSP)[3]. In this blog the CEO of Codenvy is cited stating that a drawback of traditional IDEs is the fact that they usually bring good features for a specific language, and if a developer wanted this support they were forced to use that specific IDE. With the LSP Microsoft, Codenvy and Red Hat set out to standardize a protocol that would bring core functionality from an IDE and making it available to any editor that has a client for LSP. This would give developers a choice regarding what tools they want to use when working in different languages. The language server interface was implemented so that developers would only have to create one language server per language, and one client per tool instead. This client would be language agnostic, and as such the ones developing editing tools could focus on the editing part, and the ones working on language features could focus on the actual language part. This means that developers could get the possibility of choosing their favourite tool instead of being forced into having more tools that they might not actually need. In an experience report on implementing a Language Server for OCaml[1] the authors report that one of the benefits with a working language server for less popular languages is the fact that the maintainers of that language would not have to implement an entire IDE of their own. But rather just supply a working language server, and developers that want to try out that language would already have tool support in their favourite editor. At the time of this report, there was a webpage listing currently working implementations of the protocol. For convenience they have listed both what languages have a working server, and what editors currently have a working client.[4].

The current implementations of Java language servers lack support for extended versions of Java, such as the ones provided by ExtendJ. To solve this we set out to implement a Language Server using ExtendJ for its back end. This would mean that any extensions to the compiler would automatically be reflected in the language server as well. The evaluation for this implementation will mostly consist of trying the features by hand in two editors, Vim and Sublime text. It would also be interesting to compare our language server with the one that the eclipse foundation have implemented. However due to the scope of this project we will probably have a slower system since it is probable that the compiler itself would need work, to make it faster in this application. The reasoning behind this as mentioned in section 3 in Merlin[1], is due to the fact that a compiler that is developed for running once every time the programmer deems the file complete enough for compilation might not have support for unfinished lines and might give errors that might not be interesting in the middle of writing program code. Also, it was mentioned that a good optimization is to not compile all source code at all times for diagnostics, but rather removing most code in method calls, since the signature of

[1]https://code.visualstudio.com/blogs/2016/06/27/common-language-protocol
[2]https://extendj.org/
[3]https://blog.codenvy.com/press-release-red-hat-codenvy-and-microsoft-collaborate-on-language-server-protocol-8f7c27f2d2ab

[4]https://langserver.org/

Course paper, EDAN70, Lund University, Sweden
February 4, 2019.

the method call would still be the same. And thus incremental compilation would be faster. That would then lead to the time between typing and getting good suggestions for code completion being shorter than if the compiler would have to compile everything all the time. This would be more noticeable the bigger the project that was worked on.

2 Background on the Language Server Protocol

Language servers are supposed to handle language smartness[5] such as Code completion, hover provider, jump to definition, workspace symbols, find references, and diagnostics. The most important message in the communication has to be the capabilities message that is the first message the server will send to the client. This message is a list of features that this server will provide, and as such gives the client an idea of what it can expect to get valid responses to.

2.1 Features

If code completion is implemented, the language server will be able to send hints to the client on what the programmer might want to type. This can be based on language features and if the server has a way of keeping track of what variables are currently defined, it could suggest variable names as well.

Hover provider means that the language server will be able to send some information to the client, if the user is hovering over a word in their editor. From our experience this message can be anything, but a good language server should provide useful information about what is being hovered over. An example would be that Eclipse can put Javadoc for the item you are hovering over, if it can be found.

Jump to definition means that the server has the capability of finding where a symbol is defined, and provide a link for the editor to open and mark. If the server supports workspace symbols, then it can figure out symbols not only locally in the text document that is currently open, but all symbols defined in a predefined workspace. Find references means that the server should be able to find where a symbol is being used. Lastly the diagnostics feature means that the server can send messages to the client and note if there are potential errors or warnings in the current code base. These messages can then be shown to the user in useful ways, for example sublime underlines errors with a red line.

2.2 Communication

Since the communication of the protocol is based on JSON it means that the client and server are not locked into a specific way of communication. There are both solutions using TCP sockets and solutions using standard in and standard out, meaning direct communication between the programs. Most examples we found while working on this project seemed to be biased towards the direct communication. This was also fairly noticeable in how the editors were set up to work with the clients, since they all had a way of running an external program, in this case the expected external program should a language server.

The API we used for this project, LSP4J has 3 different modes for synchronizing documents between the client and the server. None, Full and Incremental. We were using the mode Full, which meant that the client would send all the contents of the file that is worked on every time it would send an update message. Incremental means that when the client first starts up, it should send all of the text that is in the file that was opened, and then just send what is changed between each time. As for none, this is when the server does not want any synchronization between the server and the client. This would still make sense, considering that the client will tell the server where the file is currently located, and as such any time the file would be written, the server could check what is in the file. While the full mode would make it easier for the language server to always be sure that it has the most current version of a file, in the case of a large source a lot of data would have to be communicated. Thus, incremental makes sense if the language server is confident that it has a good way to keep track of the changes and keep an updated model by itself. None would be the mode that transfers the least amount of data between the client and the server, and would mostly be useful if the server does not rely on being in sync with the client.

In figure 1 there is a diagram showing example communication between a client, labeled Development tool, and a Language server, in our case the client would be either sublime or vim. And the server would be our implementation of the LSP. This figure shows what messages might be sent between the server and a client during a routine editing session.

2.3 LSP4J

LSP4J is a framework that is developed to assist developers interested in creating their own language server in Java[7]. It is designed to manage the JSON communication between the LSP server and client, and to make this communication available for a Java developer. We use LSP4J in this project to communicate between our server and the clients we have tested it with. From the description on the project site on Eclipse's webpage they state that Microsoft, Red Hat, and TypeFox are helping with the implementation of LSP4J. Thus, it is reasonable that this project would be useful even if the LSP specification were to change, as both Microsoft and Red Hat are active in developing both the LSP and LSP4J.

[5]https://langserver.org/

[6]https://microsoft.github.io/language-server-protocol/overview

[7]https://projects.eclipse.org/projects/technology.lsp4j

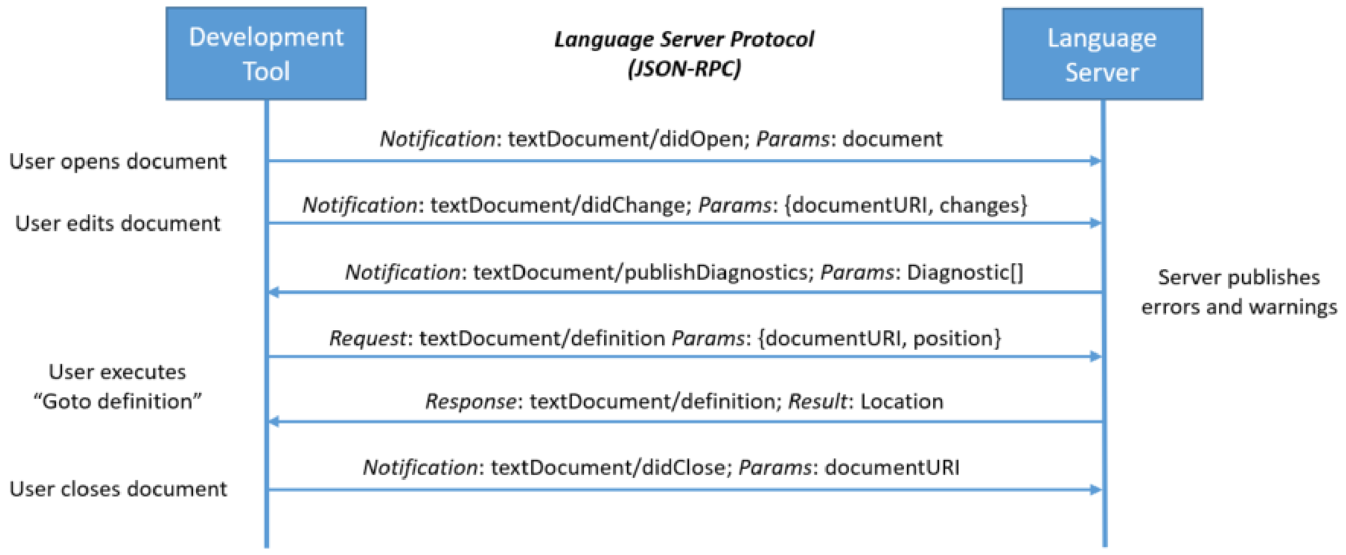

Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
Request: textDocument/definition Params: {documentURI, position}
User executes "Goto definition"
Response: textDocument/definition; Result: Location
User closes document
Notification: textDocument/didClose; Params: documentURI

Figure 1. This is a diagram provided by Microsoft[6]that shows example communication during a routine editing session. These messages are examples of the communication that would occur between a LSP server and client, such as the communication between our server and any chosen client.

3 Implementation

Currently the system only sends basic diagnostics from the compiler to the editor. Meaning that our language server is able to tell the editor to mark lines that contain warnings or errors from the compiler. Naturally, this means that we need to start adapting the compiler to be able to send these kinds of messages.

3.1 ExtendJ as a backend

ExtendJ is an extensible Java compiler that is being developed at Lund University. The usefulness of this compiler compared to the standard Java compiler comes from the fact that ExtendJ is designed to be simple to add support for new features for the Java language. This and the fact that the compiler is built using reference attribute grammars, that can expose some of the inner workings of the compiler, made it a good candidate for use with a language server. The fact that it is easy to add and test new language features to the compiler, would mean that if a language server was developed using it would also be able to recognize the new functionality that was added to the compiler.

If the compiler can expose some of its inner workings to an outside source, which in this case would be a language server. Then the language server would benefit from having code analysis done by the compiler, instead of having to implement that in the language server as well.

We wanted to send diagnostic messages from ExtendJ to our language server, instead of having these kind of messages output as text as this would be the first step of having some sort of language smartness in our language server. Usually when working with compilers the compiler would print status reports, including warnings and errors in a terminal, either inside an IDE or wherever the compiler was run from. These messages are however not the most ideal way of presenting certain information, such as if a class name was miss typed and the only message states that the error was on a certain line. The goal of the diagnostic messages were that they should be underlined in the editor by sending the correct type of message.

3.2 LSP4J

Since the LSP works using JSON for its communication, we needed a way to convert our messages from the server into valid JSON, for this we used a framework called LSP4J[8]. This framework handles communication between a language server and a client. It has functions that fetches the JSON requests from a client and simplify the implementation, so that when the framework is used the details of the communication is handled by this framework, giving us more time to work on the actual functionality of the server.

To make this work together with our compiler we started working out what requests we should run the compiler with. And for testing purposes we only compile the file that is currently being saved. This current implementation will then push errors, if there are any, to the client.

4 Evaluation

For this kind of project it is hard to write useful test suites. In an experience report[1] the authors reasoned that the fact that the functionality that we are trying to implement is

[8]https://projects.eclipse.org/projects/technology.lsp4j

something a user will have to experience to see if it works. This led us to decide that manual testing would be more reasonable, and set up the editors sublime text and vim to work with our language server. Since a language server by design should not care what program it is communicating with, as long as it follows the protocol, the setup was mostly based on getting the editors to work with their respective language server clients. For testing, we made a very small Java program that was opened in the editor when it was communicating with the language server. Since it was a small program, we could very simply introduce compiler errors and make sure that the diagnostics would work. In figures 2 and 3 we demonstrate how the two editors show the diagnostic message we send.

5 Related work

5.1 ExtendJ

ExtendJ an extensible Java compiler built on JastAdd[9]. The goal of this compiler is to have a fully functional Java compiler that is easy to extend with new language functionality. The compiler is built using reference attribute grammars and this is one of the reasons the compiler is easy to extend. [2]

5.2 Existing Language servers for Java

We noticed when we wanted to test the current implementations of language servers for Java that both the major implementations were not that straight forward to use. Since they seemed to have been developed for use with particular systems in mind. One of the language servers only mentioned that it should be installed from Visual Studio code[10], and we were not able to get it running without Visual Studio code. The other major implementation seems to be focused around using with Eclipse[11], and we were not able to find a good guide for getting this to run, either in eclipse or as a stand alone program. Thus, it would seem that not all the current language servers are as editor agnostic as it sounded when reading about the project. Our experience when trying to get our language server to work, was that Eclipse treated their language servers as Eclipse plugins, and we are not sure at this moment how much work would have to be done to make our language server to work in Eclipse. This seems to be the case for Visual Studio Code as well, but since we have never used Visual Studio Code we are not able to say if our implementation would work there either.

6 Future work

Since this project barely scratched the surface of what a language server can provide, there is definitely room for more work. However, when reading an experience report [1] the authors also state that during their work they noticed that the if this protocol is implemented with a compiler that is not already made for an IDE the compiler will have to be adapted to handle compiling partial programs. Other optimizations of the compiler might also be useful if a faster language server is desired. A good start to improve our implementation would be to make the diagnostic messages show up in real-time rather than only when the file is saved. This is more in line with how a traditional IDE work.

7 Conclusion

The functionality provided by the LSP provides the potential for adding IDE features for new languages easier than if the language developers had to develop an entire tool chain. As would be the case if they wanted language support in an editor, since the language developers might not have access to work on existing IDEs. However, if they were to implement a working language server, then anyone interested in trying out their new language would get all the helpful features the protocol can provide. While not being forced to install and use a completely new tool. Of course from our experience an IDE might have more features than the LSP provides, such as built in debugger, or the ability to compile and run projects with a simple click of a button in the interface.

It was also interesting to see that the IDE developers also seem interested in this protocol, as Eclipse is developing a new client that has support for the LSP[12]. The fact that Microsoft also supports this protocol in their Visual Studio Code editor shows the potential of the protocol, as Microsoft is a large company that already has developed IDEs for C/C++ in the form of Visual Studio, and they could have simply kept iterating on this IDE instead being part of the development of a new way of implementing this type of functionality.

It is worth noting that most of our time on this project was spent understanding how the LSP works and getting a simple server to run. We would have liked to get more time to dig deeper into the inner workings of the compiler to see what features that might be easy to add to the language server. That in itself could be an entire project from what we gathered from working from the perspective of the LSP. We also came to appreciate the hard work that each IDE have had to do in the past to get all of the features commonly associated with an IDE working as good as they have. We also came to appreciate the fact that there already was a framework for the JSON messages, this meant that we did not have to put too much time into implementing that, even if getting it working from the beginning took longer than we had hoped.

Acknowledgments

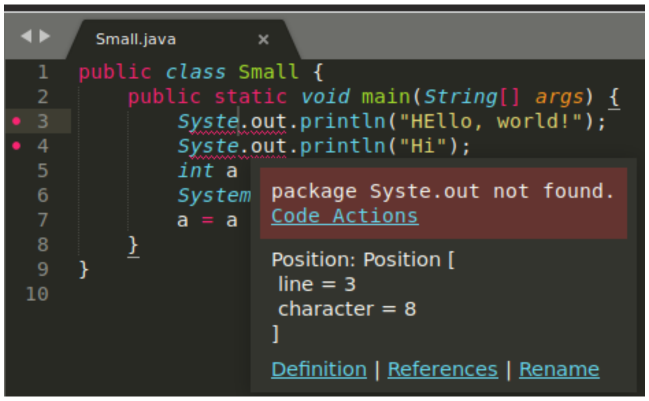
We would like to thank our supervisor Christoph Reichenbach for all the good feedback and help we got during this project.

[9]http://jastadd.org/

[10]https://github.com/georgewfraser/java-language-server

[11]https://github.com/eclipse/eclipse.jdt.ls

[12]https://www.eclipse.org/che/features/



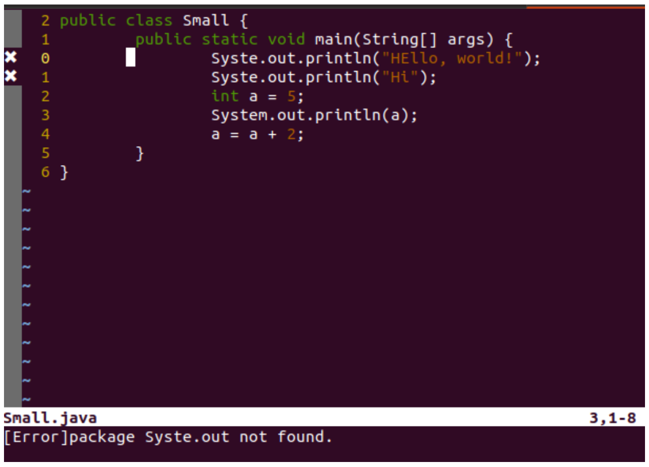
```
public class Small {
    public static void main(String[] args) {
        Syste.out.println("HEllo, world!");
        Syste.out.println("Hi");
        int a
        System
        a = a
    }
}
```

```
package Syste.out not found.
Code Actions

Position: Position [
 line = 3
 character = 8
]

Definition | References | Rename
```

Figure 2. An example of the diagnostics message being displayed in sublime.



```
public class Small {
        public static void main(String[] args) {
                Syste.out.println("HEllo, world!");
                Syste.out.println("Hi");
                int a = 5;
                System.out.println(a);
                a = a + 2;
        }
}
Small.java                                          3,1-8
[Error]package Syste.out not found.
```

Figure 3. An example of the diagnostics message being displayed in vim.

References

[1] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: A Language Server for OCaml (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 103 (July 2018), 15 pages. https://doi.org/10.1145/3236798

[2] Jesper Öqvist. 2018. ExtendJ : Extensible Java compiler, Jennifer B. Sartor and Stefan Marr (Eds.), Vol. Part F137691. Association for Computing Machinery (ACM), 234–235. http://dx.doi.org/10.1145/3191697.3213798