Language Server Protocol for ExtendJ

Fredrik Siemund
D15, Lund University, Sweden
htx12fsi@student.lu.se

Daniel Tovesson
C15, Lund University, Sweden
dic15dto@student.lu.se

Abstract

Microsoft's Language Server Protocol (LSP) has been implemented with ExtendJ in two widely popular Integrated Development Environments (IDE), Eclipse and Sublime Text. LSP is a way of supplying high level support — such as code completion, hover tooltips, jump-to-definition and find-references — for a programming language in a wide variety of editors. ExtendJ is an extensible compiler for Java, specified using reference attribute grammars. Because of the time limit of the project, only error checking was implemented. As the user writes Java code, the IDEs will highlight erroneous code and thereby simplify the process of finding errors. On the client side, this was done by configuring two extensions, one for each IDE. On the server side, a language server was created that compiles the file with ExtendJ every time it is changed by the user to search for compilation errors. To make LSP work flawlessly with ExtendJ, an extension to ExtendJ was created.

1 Introduction

The traditional way of supplying high-level support, such as code completion, hover tooltips and jump-to-definition for a programming language in an editor, is to develop a separate plugin for each editor. *Language Server Protocol* (LSP) was created by Microsoft to make this process easier. Instead of developing and maintaining many different plugins, it is with the help of LSP possible to implement one language server which provides high-level support for many different editors.

The goal of this project was to implement LSP for *ExtendJ* [2], which is an extensible compiler for Java specified using *reference attribute grammars*. Reference attribute grammars makes it easy to extend the compiler with new language constructs and analyses [5]. This was useful because an extension to ExtendJ had to be implemented which is described later.

Focus was on implementing error checking, which is called `Diagnostics` in LSP. The implementation was done by creating a language server using LSP4J, a reusable Java implementation of LSP [10]. The language server in turn uses ExtendJ to compile the file that the user is editing to search for errors. To make ExtendJ work with the language server, an extension to ExtendJ had to be developed. LSP communicates with the language protocol over JSON-RPC [4], so the ExtendJ extension outputs the errors in JSON format instead of a sequence of strings. Finally, the IDEs were configured to make use of the language server created. This improved the process of developing in Java by providing the developer with better feedback in the editor. The work was evaluated by demonstrating the validation features implemented in the language server. The language server was tested with Eclipse and Sublime Text, which both worked well.

2 Background

2.1 Language Server Protocol

LSP is used to provide syntax highlighting and/or validation for a specific language. As mentioned in [8], you need a client and a server. The client, in this case an IDE, sends notifications to the server whenever some predefined events happen, for example if something in a file is changed. One goal of LSP is to standardize the exchange of these messages between the client and the server [7]. When the server receives a notification from the client, it interpreters it and eventually sends a notification to the client containing information about the change made, e.g. if it is valid or not.

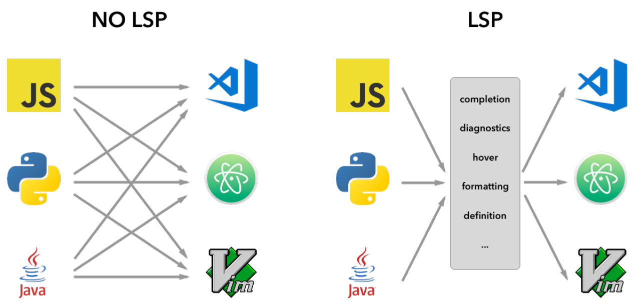


Figure 1. How Microsoft's Language Server Protocol works [8].

The big advantage of using LSP is that you just need to create one language server that handles all the logic, e.g. what text should be highlighted and what is considered an error. Earlier, each client needed to write an extension as well as a server for each specific language to add support for it. But with the introduction of LSP, just one language server needs to be implemented for every editor supporting LSP. On the client side only a small configuration has to be made that tells the IDE to use that specific language server when editing a text document. The benefits of LSP is visualized in Figure 1.

2.1.1 LSP Messages

Since the client and server are decoupled, the communication between them is done using the language protocol over JSON-RPC [4], which is a light-weight remote procedure call protocol. For its communication, the LSP uses three different types of messages: notifications, requests and responses [9]. Notifications don't require a response while requests always must be answered with a response.

Source Code 1. JSON-RPC request for receiving symbol definition.

```
{
    "jsonrpc": "2.0",
    "id" : 1,
    "method": "textDocument/definition",
    "params": {
        "textDocument": {
            "uri": "file:///path/file.java"
        },
        "position": {
            "line": 3,
            "character": 12
        }
    }
}
```

A JSON-RPC message always contains the JSON-RPC version, which method is being called and a number of parameters. In addition to that, request and response messages also contains an unique id. See Source Code 1.

Below are some examples of how the editor and language server could communicate during an editing session [11]:

- User opens a file (called document in LSP): the client sends a notification to the language server with method `textDocument/didOpen` that has the file URI and file contents as a parameter.
- User edits the document: the client sends a notification to the language server with the method `textDocument/didChange`. The server answers with a new notification with the method `textDocument/publishDiagnostics`.
- User uses 'Go To Definition' on a symbol: client sends a request to the server with method `textDocument/definition` and two parameters: the file URI and the text position of the symbol (see Source Code 1). The server answers with a response containing a document URI and the symbol's definition inside the document.

2.1.2 LSP4J

The language server can be built in any programming language when developed from scratch, but there also exists frameworks to make this process a bit easier. In this project, LSP4J was used. LSP4J [3] is a reusable Java implementation of the language server protocol, developed and maintained by the Eclipse Foundation. The LSP4J framework provides several interfaces that need to be implemented to create a new language server. LSP4J handles all the communication to the client and makes it a lot easier to develop a new language server.

2.2 Motivating Example

When developing in Java using ExtendJ, no syntax highlighting or validation is provided when using a IDE. This makes the process of developing in Java using ExtendJ quite difficult when it comes to tracking down errors. There are several ways to add support for syntax highlighting and validation for a language in IDEs. You can create a client and a server for each editor or you can use LSP. The latter option was chosen because it would provide greater flexibility, be less time consuming to implement and make it easier to add support for ExtendJ in other editors in the future.

2.3 Process

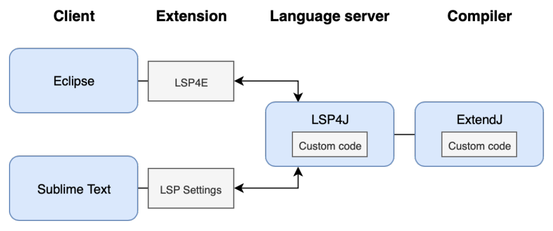


Figure 2. Project structure.

Two widely popular IDEs were chosen, Eclipse and Sublime Text, to evaluate that the language server was working. In the next sections it will be described in detail how the implementation of LSP was achieved with these two IDEs. The overall project structure can be seen in Figure 2.

2.3.1 Setup language server

A language server was setup following a template created by Lucas Bullen [1], which is based on the Java framework LSP4J. As mentioned earlier, LSP4J provides interfaces that can be implemented to create a language server that support LSP. It consist of three classes worth mentioning:

Source Code 2. Language server setup.

```
public CompletableFuture<InitializeResult>
initialize(InitializeParams params) {
    final InitializeResult res =
        new InitializeResult(
            new ServerCapabilities()
        );
    res.getCapabilities().
        setCodeActionProvider(Boolean.TRUE);
    res.getCapabilities().
        setCompletionProvider(
            new CompletionOptions()
        );
    res.getCapabilities().
        setDefinitionProvider(Boolean.TRUE);
    res.getCapabilities().
        setHoverProvider(Boolean.TRUE);
    res.getCapabilities().
        setReferencesProvider(Boolean.TRUE);
    res.getCapabilities().
        setTextDocumentSync(
            TextDocumentSyncKind.Full
        );
    res.getCapabilities().
        setDocumentSymbolProvider(Boolean.TRUE);

    return CompletableFuture.
        supplyAsync(() -> res);
}
```

ExtendJLanguageServer This class is where all the setup for the language server is made. LSP has a lot of functionality and neither editor nor client need to support all of it. What functionality (called capability in LSP) that is supported is negotiated as the language server starts and it is configured to support all features available in the IDEs, such as code completion and text hover. This is done in the `initialize` method of the language server which can be seen in Source Code 2.

ExtendJTextDocumentService After setting up the language server with the supported features, the code needs to be validated. This is done in this class. There are several methods in this class that can be implemented, but since error checking was the focus of this project only the methods `didOpen` and `didChange` were implemented. The method `didOpen` is called when a document is opened, `didChange` is called when a document is changed. These are valuable triggers as it allows real time feedback to the user as he or she is typing. Both of these methods make use of a method called `validate`. It was customary created for the sole purpose of validating the text in the document whenever it is opened or changed. The solution to this is quite straightforward. The file that the user currently is working in will be compiled with ExtendJ each time a change is made. This is done by using a version of ExtendJ that has been packaged into a `.jar` file. The output from the compiler is fetched through a input stream and is then parsed into a JSON object containing the error message, line number and column number. This JSON object is put into a `Diagnostic` object for each error that occur. The object is then used to provide information to the client which is the error and what line number and column number is affected.

ExtendJWorkspaceService This class was not implemented but can be used to trigger events when settings to the workspace is changed.

Source Code 3. ExtendJ Extension.

```
// Code in ExtensionMain.java
@Override
protected void
processErrors(Collection<Problem> errors,
CompilationUnit unit) {
    unit.process(errors);
}

// Code in ExtensionBase.jrag
aspect ExtensionBase {
    public void CompilationUnit.process(
    Collection<Problem> errors) {
        ArrayList<String> jsonStrings =
            new ArrayList<String>();
        for (Problem error : errors) {
            String fileName = "\"fileName\":\""
                + error.fileName()
                + "\"";
            String message = "\"message\":\""
                + error.message()
                + "\"";
            String line = "\"line\":"
                + error.line();
            String column = "\"column\":"
                + error.column();
            String endLine = "\"endLine\":"
                + error.endLine();
            String endColumn = "\"endColumn\":"
                + error.endColumn();
            jsonStrings.add("{"
                + String.join(",",
                new String[]{
                    fileName, message, line,
                    column, endLine, endColumn
                })
                + "}");
        }
        String json = "{\"errors\":["
            + String.join(",", jsonStrings)
            + "]}";
        System.err.println(json);
    }
}
```

2.3.2 Create ExtendJ extension

An ExtendJ extension was created so that the error output from the ExtendJ compiler is in a JSON format instead of a sequence of strings. The problem with having a sequence of strings was that a regular expression had to be made to interpret each string. The goal was to fetch the line number, column number and error message. If the format of the strings would be changed in the future, the regular expression would have to be updated as well. Writing an ExtendJ extension that changes the output to a JSON format made the solution more future proof. Each error was made to an object that was put into a array. Each error object has value-key pairs for line, column, end line, end column and error message that are fetched from the `Problem` objects. This was done by overriding the `processErrors` method as seen in Source Code 3.

2.3.3 Create IDE extension

When the language server is setup, the editors need to be configured so that they can make use of it. Currently, there exists no editor that has out of the box support for LSP, so to make the IDE talk with the server an extension has to be installed. This extension will handle the communication with the language server over JSON-RPC and among other things display errors in the editor. In this project the extension LSP4E was used for Eclipse and the extension LSP was used for Sublime Text.

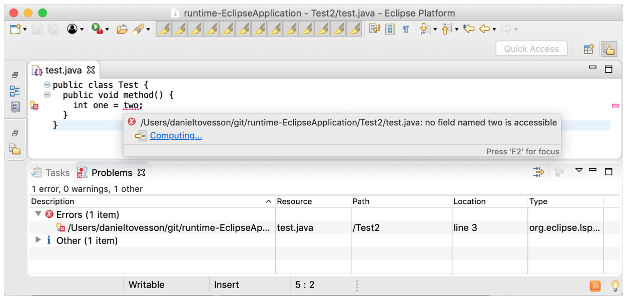
The Eclipse extension is a small plugin that runs a version of the language server packaged into a `.jar` file. Inspiration for this plugin was taken from Mickael Istria [6].

The Sublime Text extension is just a small build script which starts the server. Implementing this was straightforward as examples could be found by testing existing language server solutions and looking how they were setup to work with Sublime Text.

One major difference between the LSP extension for Eclipse and Sublime Text is that Sublime uses a generic plugin which only requires some configuration. Eclipse on the other hand requires a small plugin — based on LSP4E — for each new programming language.

3 Evaluation

The implementation of the LSP server was evaluated by writing Java code in the two chosen IDEs, Sublime Text and Eclipse. Many different Java files were written, both with correct and erroneous code, to see if the language server behaved as expected. The output was compared with built-in compilers, like javac, to see if they matched. It was crucial to make the LSP implementation work seamlessly with the two chosen IDEs. The response from the language server when writing code containing errors should be quick and by listening to changes in the text document the response was immediate. When the user wrote code containing errors,



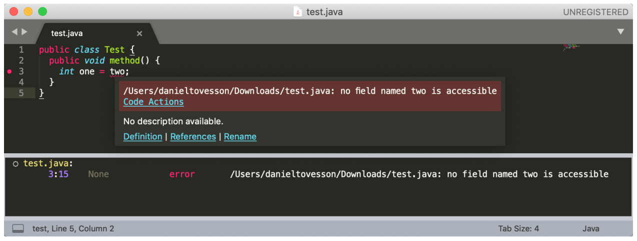
```
public class Test {
  public void method() {
    int one = two;
  }
}
```

/Users/danieltovesson/git/runtime-EclipseApplication/Test2/test.java: no field named two is accessible

1 error, 0 warnings, 1 other

Description	Resource	Path	Location	Type
Errors (1 item)				
/Users/danieltovesson/git/runtime-EclipseAp...	test.java	/Test2	line 3	org.eclipse.lsp...
Other (1 item)				

Figure 3. Errors in Eclipse



```
public class Test {
  public void method() {
    int one = two;
  }
}
```

/Users/danieltovesson/Downloads/test.java: no field named two is accessible

No description available.

```
test.java:
    3:15    None        error       /Users/danieltovesson/Downloads/test.java: no field named two is accessible
```

Figure 4. Errors in Sublime Text

they were highlighted as soon as they were made, as can be seen in Figure 3 and 4. Since this worked so well, the IDE warned before a line of code was completed were the part of the code written could be considered erroneous. This is the way all modern IDEs work today so it was not considered a problem.

4 Related work

This project has taken inspiration from Lucas Bullen's project [1]. The aim with Bullen's project was to give people not familiar with LSP a tutorial on how to set one up. The goal of the tutorial was to create an itinerary for what sessions to attend at EclipseCon 2017, using a `.txt` file. This project was extended to fit the needs of ExtendJ. Instead of `.txt` files, `.java` files were used and instead of static suggestions the text document was compiled with ExtendJ and the errors were sent back to the client.

5 Conclusion and Future Work

We have implemented LSP support for the extensible Java compiler ExtendJ. It has been evaluated by testing LSP functionality in different IDEs. Because of the time limit of this project, the only functionality implemented was error handling. The limit was also set to two IDEs, Eclipse and Sublime Text. The goals of the project were thereby achieved and the result was satisfactory.

To further extend the project things like code completion, hover tooltips, jump-to-definition, find-references, and more could be implemented. Some of these would require additional extensions to ExtendJ but there should be no limits regarding how many features that can be implemented.

Acknowledgments

We would like to thank Niklas Fors for mentoring us through this project.

References

[1] Lucas Bullen. 2018. Build Language Servers in Java. (2018). https://github.com/LucasBullen/LSP4J_Tutorial

[2] Torbjörn Ekman. 2018. ExtendJ - The JastAdd Extensible Java Compiler. (2018). https://extendj.org

[3] The Eclipse Foundation. 2016. Eclipse LSP4J Project. (2016). https://projects.eclipse.org/proposals/eclipse-lsp4j

[4] JSON-RPC Working Group. 2013. JSON-RPC 2.0 Specification. (2013). https://www.jsonrpc.org/specification

[5] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24 (2000), 301–317.

[6] Mickael Istria. 2018. Language Server Protocol demo. (2018). https://github.com/mickaelistria/eclipse-languageserver-demo

[7] R. Rodriguez-Echeverria M. Wimmer J. Cabot, J. Luis Cánovas Izquierdo. 2018. Towards a Language Server Protocol Infrastructure for Graphical Modeling. *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* 1 (2018), 370–380. https://dl.acm.org/ft_gateway.cfm?id=3239383&ftid=2007177&dwn=1&CFID=6159321&CFTOKEN=bc04a47d03daf50a-8DC8CD34-E02D-4C30-3BC2482066F596F0

[8] Microsoft. 2018. Example - Language Server. (2018). https://code.visualstudio.com/docs/extensions/example-language-server

[9] Microsoft. 2018. What is the Language Server Protocol? (2018). https://microsoft.github.io/language-server-protocol/

[10] Miro Spönemann. 2018. Eclipse LSP4J. (2018). https://github.com/eclipse/lsp4j

[11] The VS Code Team. 2016. A Common Protocol for Languages. (2016). https://code.visualstudio.com/blogs/2016/06/27/common-language-protocol