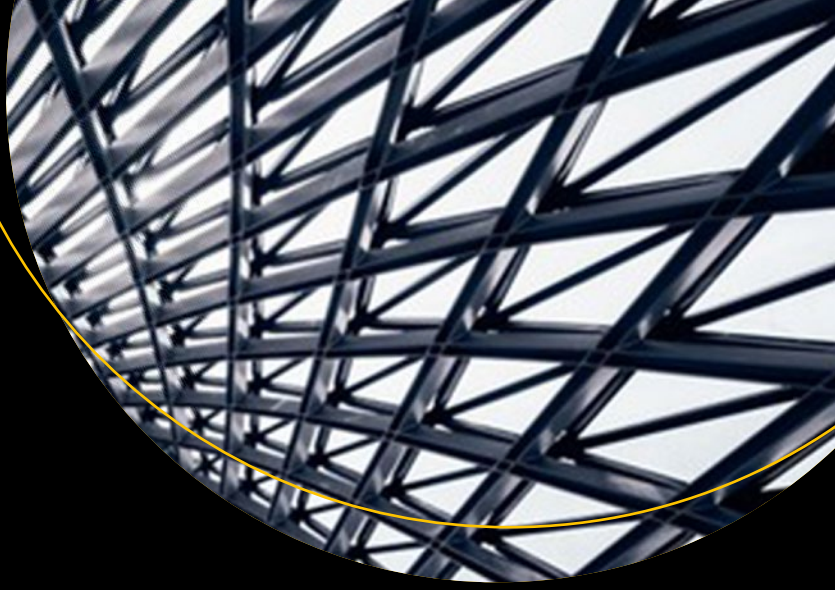

Language Server Protocol and Implementation

Supporting Language-Smart Editing and Programming Tools

Nadeeshaan Gunasinghe
Nipuna Marcus

Apress®

Language Server Protocol and Implementation

Supporting Language-Smart Editing and Programming Tools

Nadeeshaan Gunasinghe
Nipuna Marcus

Apress®

Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools

Nadeeshaan Gunasinghe
Walahanduwa, Sri Lanka

Nipuna Marcus
Mawathagama, Sri Lanka

ISBN-13 (pbk): 978-1-4842-7791-1
ISBN-13 (electronic): 978-1-4842-7792-8
https://doi.org/10.1007/978-1-4842-7792-8

Copyright © 2022 by Nadeeshaan Gunasinghe and Nipuna Marcus

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-7791-1. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To our beloved parents who always were behind us and
Kasun Indrasiri who always was an inspiration and
a role model

Table of Contents

About the Authors

Nadeeshaan Gunasinghe is Technical Lead at WSO2 and has more than five years of experience in enterprise integration, programming languages, and developer tooling. He leads the Ballerina Language Server team and is a key contributor to Ballerina, which is an open source programming language and platform for the cloud, and he is an active contributor to the WSO2 Enterprise Service Bus.

Nipuna Marcus is Technical Lead at WSO2 and has more than five years of experience in front-end development, programming languages, and developer tooling. He was a member of the Ballerina Language Server team and a key contributor to the Ballerina programming language.

About the Technical Reviewer

Andres Sacco has been working as a developer since 2007 in different languages including Java, PHP, Node.js, and Android. Most of his background is in Java and the libraries or frameworks associated with this language, for example, Spring, Hibernate, JSF, and Quarkus. In most of the companies that he worked for, he researched new technologies in order to improve the performance, stability, and quality of the applications of each company.

Acknowledgments

We would first like to thank Jonathan Gennick, Assistant Editorial Director at Apress, for evaluating and accepting our proposal for this book. We would also like to thank Laura Berendson, Development Editor at Apress, and Nirmal Selvaraj, Project Coordinator, for guiding us toward the end. Andres Sacco served as the Technical Reviewer. Thank you, Andres, for making sure we did our best.

Kasun Indrasiri, Software Architect and author of *Microservices for the Enterprise* and *GRPC: Up and Running*, inspired us to work on this book and mentored us throughout the process. We are eternally grateful to Kasun Indrasiri for the guidance and support.

Finally, we would like to thank our families and parents as, without them, none of our life achievements would be possible.

Introduction

The Language Server Protocol (LSP) has been one of the most talked about topics during the past few years when it comes to the tooling for programming languages. With the advancement of the developer tools and the programming languages, developers started to rely more and more on advanced tools and enhanced language services. When we consider one of the most focused branches of developer tools which is IDEs and text editors, there are many vendors who have released various editing tools in the past couple of decades. When we consider the number of programming languages along with the number of smart editors nowadays, in order to support language intelligence among the editors, these vendors have to repeat the same thing. The Language Server Protocol was introduced to solve this particular problem, and today it has become the norm of the development tools' language intelligence provider. By adopting the LSP, tools such as text editors and integrated development environments (IDEs) could expand the capabilities and avoid the users' burden of switching between the development tools for trying new programming languages and frameworks.

This book is for the developers who are passionate about developing programming language tools. In this book, we provide the readers a comprehensive understanding about the Language Server Protocol and how to develop a Language Server from scratch. The readers will be guided with code samples to provide a better understanding about the server implementation by adhering to the user experience best practices as well as the LSP best practices. The readers are expected to use the book along with the example implementation, in order to get a better understanding about the concepts described in the book. In the example implementation, the book refers to VS Code as the client; however, the readers can use any other client and integrate the server implementation as desired.

The chapters of the book have been ordered to capture various aspects of the developer experience when it comes to the programming language tooling, and the LSP operations and features are categorized under these aspects. The readers are not overwhelmed by including the code snippets of the data structures in the LSP and it is recommended to refer to the official documentation of the Language Server Protocol for the data structures.

CHAPTER 1

Developer Tools and Language Services

Today, software development has become an area where there are higher expectations when considering the rapid development, go-to market, deployment, distribution, and similar aspects. In this book, we are going to focus on a specific technical perspective related to source code editing or, in other words, writing the software.

Early Programmable Computers

Decades before the golden age of digital computers, it all began with mechanical computers. Even though today computers can carry out various tasks, early mechanical computers could carry out a specific task. Among them, one of the most important programmable machines developed was the Jacquard loom. Jacquard loom's idea of programming the machinery was later inspired by other programmable inventions as well.

Joseph-Marie Jacquard invented the Jacquard loom[1] in 1804. The Jacquard loom was programmed by using punched cards. Different weaving patterns could be programmed by using punched cards, and the loom was automated on top of the program. Therefore, the Jacquard loom is considered the first programmable machine, and the concept of punched cards was adopted later by both Babbage as well as digital computers.

In the history of mechanical computers, the Analytical Engine developed by Charles Babbage[2] can be considered as the earliest mechanical computer. Input data was fed to the machine by means of punched cards which were used in the Jacquard

[1] www.columbia.edu/cu/computinghistory/jacquard.html

[2] www.computerhistory.org/babbage/

© Nadeeshaan Gunasinghe and Nipuna Marcus 2022
N. Gunasinghe and N. Marcus, *Language Server Protocol and Implementation*,
https://doi.org/10.1007/978-1-4842-7792-8_1

programmable loom. One of the most important aspects of the Analytical Engine is the ability to program the engine by changing the instructions on punched cards.

In both of the aforementioned examples, the medium of programming the computers was using punched cards. Not only mechanical computers but also early digital age computers used punched cards to input programs to the computer and store data. For example, if we consider computers such as IBM 360, it was the punched cards that were used to write the programs.

Code Forms and Punched Cards

For instance, let's compare how we write our programs with a programming language today with the punched card era.[3] Have you ever wondered that it might have taken hours to write a program, run it, and see the output? Have you ever tried to write a program on a piece of paper? Imagine how tedious that would be to develop even a simple "Hello world!" in such a manner? Decades before today, this is how even professional programmers used to write their programs. As briefly described in the previous section, in early days punched cards were used to store data and programs, and those cards were fed into the computer for execution. Keypunching your program to the punched card in a single run is not an easy task. Therefore, another technique called code sheets, also known as coding forms, was used. The programmers had to write their programs' instructions on the code sheets at first. Then they had to convert these instructions to a form which can be identified by the computers.

Computers could identify and process the instructions fed in the form of punched cards. A punched card operator keypunches the instructions onto the punched cards to insert them into the computer to run. If there is an error or a bug, you have to follow the same routine again and again to fix it.

Now consider a program with a number of statements and the cycle of writing, running, and fixing issues. Each time, you will get a stack of punched cards where the program is embedded into. This is the amount of work that had to be done decades ago in order to write even a simple `"Hello World"` program.

[3] www.columbia.edu/cu/computinghistory/fisk.pdf

Text Editors vs. Source Code Editors

With the advancement of technology, programmers started using various tools for writing their codes with convenience. Among these tools, text-based editors have become more popular than other options. Even though text-based tools are more popular than the others, there are other tools such as visual programming editors which allow users to write their code with graphic components. Developers can drag and drop high-level graphical constructs/basic programming constructs to build the program, and the editor auto-generates the textual code on behalf of the user. Google's Blockly is a library which allows building visual programming editors. Also, programming languages such as Prograph and visual programming environments such as Cantata[4] can be considered as tools for visual programming.

Today, when editing source codes, there are various editing tools to be chosen among. The most common choice would be a source editor, text editor, or an IDE. In one aspect, all the aforementioned choices are similar, in that all those options support text editing. But, those are completely three different tools when it comes to the editing experience. Most of the time, there is a misconception to consider that source code editors and text editors are the same even though they are not. Text editors, as the name implies, are used for editing textual content and can also be used for composing and editing source codes. Source code editors on the other hand have language sensitivity and context awareness. There are certain features incorporated into source code editors which represent language sensitivity and context awareness. Among them, auto-completion, syntax highlighting, and intelligent refactoring options such as source formatting can be shown as the most commonly used features.

Before the origin of graphical user interfaces (GUIs), command-line interface (CLI)–based text editors were popular among the developers. While CLI-based editors are command driven, GUI-based editors are menu driven. Even though GUI-based editors became popular, people continued to use CLI-based editors even today. Also, having certain expertise in using a CLI-based editor is a must when you get to work in an environment without a GUI such as a server. Today, we can install plugins and extensions for text editors, and we can convert them to behave similarly to source code editors. One such example is the Vim editor, where you can install various plugins for formatting and auto-completion for JavaScript development.[5]

[4] https://dl.acm.org/doi/abs/10.1145/204362.204367

[5] https://vimawesome.com/plugin/vim-javascript

Some examples of text-based editing tools are as follows:

- Vi
- Vim[6]
- Emacs[7]
- Notepad++[8]
- Visual Studio Code (VS Code[9])
- NetBeans[10]
- Eclipse[11]
- IntelliJ IDEA[12]

This is just a small sample of editors among numerous available options. These various editors can be categorized based on multiple factors, such as

- Language support (multi-language or specific language support)

  IDEs such as IntelliJ IDEA, Eclipse, and VS Code support multiple programming languages allowing the user to install plugins and extensions. When the Java ecosystem is considered, IntelliJ IDEA and Eclipse are the popular choices among the developers.

- Hosting environment (cloud hosted or locally hosted)

  IDEs such as Codenvy, JSFiddle, and CodePen are popular cloud-hosted IDEs. CodePen and JSFiddle have become the most widely used IDE solutions to share working code examples.

[6] www.vim.org/download.php
[7] www.gnu.org/software/emacs/
[8] https://notepad-plus-plus.org/
[9] https://code.visualstudio.com/
[10] https://netbeans.apache.org/
[11] www.eclipse.org/downloads/
[12] www.jetbrains.com/idea/download/

- Development type oriented (web development/mobile development/DevOps)

  Android Studio is the official IDE by Google for Android development which is built on IntelliJ IDEA. If we consider tools such as Vi and Vim, they are widely used when it comes to DevOps tasks such as server configurations.

- Extensibility and plugin support

  Users can install plugins to the VS Code editor and expand the default capabilities. For example, the VS Code marketplace has plugins for numerous programming languages.

Given the aforementioned categorization, if we consider the JVM ecosystem, the most widely used IDE is IntelliJ IDEA, while text editors such as VS Code and Vi/Vim have a lesser usage.[13] Even though we consider the JVM ecosystem here, according to the Stack Overflow Developer Survey (2021),[14] VS Code is the most commonly used text editor among the developers.

Why IDEs

Source code editing is not the only phase in the software development life cycle, which consists of a number of phases as well as associated supporting steps. In each of these steps, the stakeholders use various tools to facilitate these phases. Several such tools can be listed as follows:

- Diagramming tools (for use case identification, ER diagrams[15])
- Source code editors (Notepad++, Vim)
- Version control systems (Git,[16] SVN[17])

[13] https://res.cloudinary.com/snyk/image/upload/v1623860216/reports/jvm-ecosystem-report-2021.pdf

[14] https://insights.stackoverflow.com/survey/2021#integrated-development-environment

[15] https://staruml.io/

[16] https://docs.github.com/en/get-started

[17] https://subversion.apache.org/

- Debuggers (JSwat[18])
- Coverage tools (JaCoCo[19])
- Code search tools (Sourcegraph[20])
- Linters (Checkstyle[21])
- Issue trackers
- DevOps tools
- And so on

Among these tools, we can identify categories, and during the last couple of decades, various platforms and products were introduced focusing on specific categories.

When we consider the IDEs, they put numerous features together. Therefore, we can identify IDEs as an all-in-one package for a developer. This is one reason for IDEs becoming popular over time when compared to text editors. Among the incorporated features in IDEs, the following can be identified as the most widely used ones:

- Context-aware auto-completions (smart completions)
- Source refactoring options (rename references, formatting)
- Code search options (for Java, class and method search)
- Run and debug
- Code coverage
- Integrated version controlling (Git integration and SVN)

[18] https://github.com/nlfiedler/jswat
[19] www.eclemma.org/jacoco/
[20] https://about.sourcegraph.com/
[21] https://checkstyle.sourceforge.io/config_design.html

Language Intelligence

In the previous sections, we had a look at the various options available for source code editing. Whether it is a text editor, a source code editor, or an IDE, all of those tools have a common underlying competency to be aware of the language syntax and semantics; we call it language intelligence. Let's consider an example usage of an IDE for programming. There are numerous language-sensitive features a user uses during coding.

One of the most frequently used language intelligence features is diagnostics. For example, consider writing Java code with a missing semicolon (syntax error) and compile it. Then the compiler will stop in the middle of compilation, and the console will show the problems in the source. Listing 1-1 is an example of an erroneous Java source where there is a missing semicolon. You can copy and paste the sample code snippet in your favorite source code editor and observe how the editor represents these errors to the developer.

Listing 1-1. Invalid Java Source with a Missing Semicolon

```
public class Greeting {
    public static void main(String[] args) {
        // Semicolon is missing in print statement
        System.out.println("Hello World")
    }
}
```

As an exercise, you can select more than one IDE/source editor and observe the diagnostic representations in those. Depending on the tool, the representation of the diagnostics can be different. Figure 1-1 is an example of the diagnostic representation for the preceding erroneous source snippet.

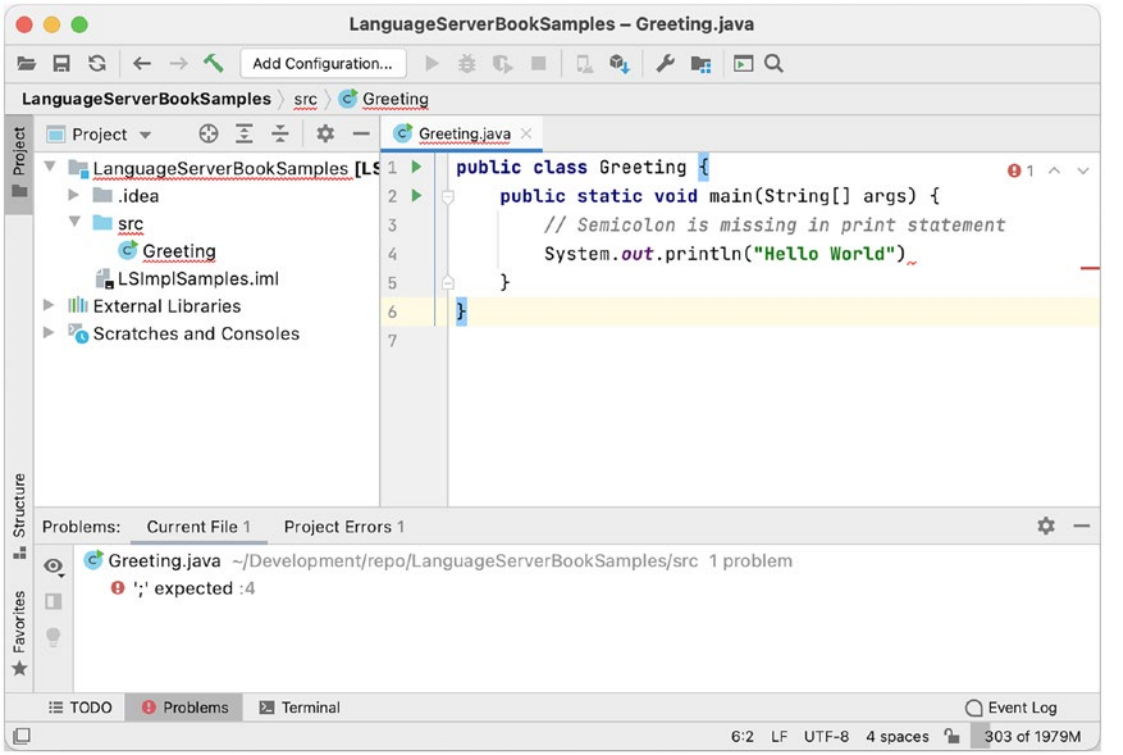

Figure 1-1. *Showing diagnostics for Java in IntelliJ IDEA*

When it comes to the IDEs/editors for source code editing, it is important to show the syntax or semantic errors on the fly. In order to implement the diagnostics for a particular programming language or even a configuration language such as Swagger,[22] the IDE or tooling developer needs to map the compiler's knowledge to the tooling APIs preserving the user experience.

We had a look at the diagnostics since it is one of the basic language intelligence features. Similar to the diagnostics, other features such as smart completions, refactoring (rename and formatting), code navigation, etc., play a major role in the development experience. Therefore, every developer tooling vendor pays considerable attention to language intelligence features integrated with the tools. With time, these language features were enriched with capabilities such as incorporating machine learning, artificial intelligence, and smart decompilers.[23]

[22] https://swagger.io/solutions/api-documentation/

[23] https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler