Building a Rust Language Server

A Complete Guide to Implementing LSP in Rust

Build a production-ready language server with autocomplete, go-to-definition, diagnostics, refactoring and more.

Autocomplete
Smarter code, faster development

Go to Definition
Jump to what matters

Hover Docs
Context at your fingertips

Diagnostics
Catch issues early

Refactoring
Better code, less effort

ALEXANDER RHODES

Building a Rust Language Server

A Complete Guide to Implementing LSP in Rust

Steve Publications

This book is available at https://leanpub.com/buildingarustlanguageserver

This version was published on 2026-09-18

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Complete Guide to Implementing LSP in Rust

Adding features like autocomplete, go-to-definition, hover documentation, real-time diagnostics and refactoring tools to a programming language is one of the most rewarding challenges in developer tooling. The Language Server Protocol turned this into a shared infrastructure problem, letting one language-aware server power dozens of editors. This book takes you from zero to a fully functional, production-ready language server written in Rust. You will learn the protocol inside out, build parsing and analysis infrastructure, implement every major IDE feature and ship a server that integrates with Visual Studio Code, Neovim and any LSP-compatible editor. Along the way you will gain deep practical knowledge of async Rust, compiler design basics, text-processing algorithms and the architecture behind modern development environments. No prior LSP experience is required; only comfort with Rust fundamentals and a willingness to build something real.

Chapter 1: What Is a Language Server?

You type fn, and before you finish the n your editor suggests fn main() with a signature you recognize. You move your cursor over a function call and a tooltip appears explaining its parameters. You press Ctrl+Click on an identifier and your view jumps to its definition. As you write, squiggly red underlines appear where you have made a mistake, and a quick menu offers fixes. You rename a variable and every reference across your project updates automatically. This is the baseline experience for modern software development, and behind every one of these behaviors is a language server quietly analyzing your code.

Language servers are not magic. They are programs that implement a well-defined protocol, exchanging structured messages with editors to provide intelligent, context-aware features. Building one is neither trivial nor impossible. This book is your guide from first principles to a working implementation.

The Problem Every IDE Solver Must Solve

Before language servers existed, every editor that wanted rich language support had to integrate directly with language-specific tooling. Visual Studio had deep integration with C#, Eclipse with Java, Sublime Text with custom plugins for each language. If you maintained a compiler or type checker for your language and wanted it to power an IDE experience, you had to build a custom plugin for every editor your users cared about. Multiply the number of languages by the number of editors, and you get the infamous M times N integration matrix that made broad IDE support prohibitively expensive.

Consider the situation for a language with no major corporate backer. You have a compiler that can type-check your code and a few basic linting rules. You want developers using Neovim, VS Code and Emacs to get autocompletion and go-to-definition. Before LSP, you needed to understand VimL, TypeScript extension APIs and Elisp – each with its own event model, threading semantics and extension lifecycle. Each integration was a separate project, and keeping them in sync was a constant battle. Even for languages that did have corporate support, the integration effort duplicated the same core logic: parsing, symbol resolution, type inference – reimplemented once for each editor plugin.

The fundamental challenge is separation of concerns. Language analysis is expensive. It requires loading entire projects, building dependency graphs, resolving cross-file references and sometimes invoking external build tools. Doing this inside the editor process introduces latency and instability. Doing it as a separate process requires a well-defined communication channel. The ideal solution factors the problem into two parts: a language-specific analysis engine that understands one language deeply, and editor-specific client code that understands how to present results to the user. The two communicate over a stable, language-agnostic protocol.

From Proprietary Plugins to a Universal Protocol

Microsoft began this journey internally. Visual Studio Code launched in 2015 with language support for JavaScript and TypeScript. These languages had sophisticated tooling: the TypeScript compiler could provide completions, type information and refactoring operations. But integrating that tooling into the lightweight Electron-based editor required a clean boundary between the analysis engine and the UI. Microsoft extracted the TypeScript language service into a standalone process and defined an internal protocol for VS Code to talk to it.

The insight was simple but powerful: if the protocol is general enough, it does not have to be tied to TypeScript. Any language with a parser and analyzer could expose its capabilities through the same protocol. Any editor could implement the client side and talk to any language server. The result would be a composable ecosystem where language support and editor support were orthogonal.

On June 27, 2016, Microsoft announced the Language Server Protocol publicly in collaboration with Red Hat and Codenvy, and moved the specification to an open GitHub repository. The specification was built on JSON-RPC 2.0, a lightweight remote procedure call protocol that serializes calls and responses as JSON. This choice made the protocol easy to implement in any language and simple to debug with standard tooling.

The early adoption was rapid. clangd (the C/C++ language server built on LLVM tooling) adopted LSP in 2016. Pyright, the Python language server, shipped with LSP support from its inception. Java Language Server, based on Eclipse JDT, provided LSP bindings. By 2020, hundreds of language servers

existed, spanning everything from mainstream languages like Python, Go and Rust to domain-specific languages and markup formats. Today the ecosystem includes more than 400 language servers, with major editors like VS Code, Neovim, Emacs, Sublime Text, Helix and Zed all implementing LSP clients as their primary mechanism for language support.

What made the adoption stick was that LSP did not lock anyone in. It was a protocol, not a framework. Implementations could be minimal or feature-rich. Servers could run as separate processes, over TCP, or as embedded WebAssembly modules. Clients could cache results, parallelize requests, or batch updates. The protocol described what information was exchanged, not how either side implemented its responsibilities. This openness meant that language teams could focus on building analysis tools while editor teams focused on presentation, and the protocol connected them.

How Language Servers Fit Into Modern Development

A language server is a long-lived process that maintains a consistent view of a codebase while responding to requests from an editor client. Its responsibilities fall into several categories.

Document synchronization is the foundation. As the user types, the editor sends incremental changes to the server so it can maintain an up-to-date view of the file. The server must handle these updates efficiently, often parsing and analyzing code incrementally to keep latency below a few milliseconds.

Diagnostics provide real-time feedback. When the user types a line that violates the language rules, the server sends a diagnostic – a message indicating the position, severity and description of the error or warning. Modern language servers often run linters and type checkers continuously, updating diagnostics as the code changes.

Code navigation enables the user to understand the structure of the codebase. Go-to-definition jumps to where a symbol is declared. Find references locates every use of a symbol across the project. Document symbols provide an outline of the current file, showing the hierarchy of functions, classes and modules. Workspace symbols allow global search across all files.

Code completion is perhaps the most visible feature. As the user types, the server suggests completions based on the current context: available functions, variables, types and language keywords. Good completions are context-aware,

showing only the symbols visible at the current scope and they may include snippets that insert multi-line templates.

Hover information provides on-demand documentation. When the user places the cursor over a symbol, the server returns a tooltip with the symbol's type, signature and any available documentation.

Refactoring operations let the user restructure code safely. Rename symbol updates every reference to a variable, function or type. Code actions offer quick fixes for errors and automated refactoring like extracting a function or reorganizing imports.

Formatting ensures consistent code style. The server can format an entire document or a selected range according to language conventions or project configuration.

Beyond these core features, modern LSP includes semantic token highlighting (richer syntax coloring based on semantic analysis), code lenses (inline commands like Run Test or Go to Implementation), signature help (parameter hints as you type a function call) and inlay hints (inline type annotations and parameter names).

All of these features share a common pattern: the editor sends a request specifying a location in a document, and the server returns structured data describing what to display or how to modify the text. The protocol is methodical and predictable, which makes it ideal for implementation in a systems language like Rust.

What We Are Building: Scope and Roadmap

This book takes you through the complete construction of a language server in Rust, from initial project setup to editor integration and deployment. The server we build will be called MiniLSP and will target a simple language called MiniScript, which we will design and implement from scratch.

MiniScript is a minimal C-like language with typed variables, functions, conditionals and loops. It is not intended to be a practical language but rather a non-trivial example that demonstrates every aspect of language server development: lexical analysis, parsing, symbol resolution, type checking and all the LSP features listed above. By building a complete language alongside the server, we can show how each feature depends on the underlying analysis infrastructure.

The roadmap follows a deliberate progression. We begin by understanding the LSP specification and choosing the right Rust crates. We set up the project and implement the initialization handshake, where the server advertises its capabilities to the client. We then implement document synchronization, the mechanism that keeps the server informed about file changes. With documents in hand, we build the lexer and parser for MiniScript, producing an abstract syntax tree that represents the structure of the code.

From the AST we can generate diagnostics, report symbols and enable code navigation. We implement hover, go-to-definition and find references by walking the AST and maintaining a symbol table. We add completions, signature help and code actions by extending our analysis with type inference and scope resolution. We implement semantic tokens for highlighting, code lenses for inline commands and a formatter for consistent code style.

In parallel, we address the engineering concerns that separate a prototype from a production tool: asynchronous request processing, caching and incrementality, testing at multiple levels, logging and debugging and packaging for distribution. We conclude by integrating the server with Visual Studio Code and Neovim, showing the exact configuration files needed to make it work in real editors.

By the end of this book you will have a working language server that demonstrates all major LSP features and more importantly you will understand the architecture and tradeoffs that underlie every language server you will encounter in your development career. Whether you ultimately build a server for a real language, contribute to rust-analyzer, or simply want to understand how your IDE works, the knowledge you gain here will be directly applicable.

The next chapter dives into the LSP specification itself, examining the protocol's architecture, the JSON-RPC message format and the lifecycle of a client-server session. With that foundation, we will be ready to choose our tools and start writing Rust code.

Chapter 2: The Language Server Protocol

The Language Server Protocol defines a shared vocabulary for the interactions between an editor and a language-aware service. It does not mandate how a server parses code or how a client renders results. Instead, it specifies a set of requests, responses and notifications that every implementation must understand, along with the capability flags that let each side discover what the other supports. Mastering LSP means understanding three layers: the transport mechanism, the JSON-RPC message format and the lifecycle and feature methods that make up the protocol proper.

LSP Architecture: Client and Server Roles

At the highest level, LSP follows a client-server model with clearly separated responsibilities.

The client is the editor or IDE. It owns the user interface, the file system interaction and the selection of which documents to send to the server. When the user opens a file, the client notifies the server. When the user types, the client sends incremental changes. When the user asks for completions or navigates to a definition, the client sends a request and renders the response. The client may also push information to the server, such as workspace configuration changes or the current selection.

The server is a dedicated analysis process for one language. It receives document contents and editor requests, runs analysis (parsing, type checking, symbol indexing) and returns structured results. The server maintains its own internal state: the current contents of every open document, the parse trees, the symbol tables and any cached analysis results. It never directly touches the file system; all file contents arrive through the client and any file system operations (creating, renaming or deleting files) are delegated back to the client through workspace edit requests.

This separation yields three critical benefits. First the heavy computation of language analysis runs in a separate process, so a parsing error or runaway analysis cannot crash the editor. Second the same server can serve multiple clients (for example a terminal editor and a graphical IDE) and the same client can talk to multiple servers (for example a TypeScript server and a Markdown server running concurrently). Third the language logic and the UI logic can evolve independently; new LSP features are added as optional capabilities that older implementations can safely ignore.

Communication between client and server typically occurs over standard input and standard output (stdio), although TCP sockets are also supported. The default is stdio because it is simple: the editor spawns the server process, and all messages flow through pipes. This keeps the architecture minimal and avoids port allocation issues. The protocol defines a line-delimited framing format that wraps JSON-RPC messages, which we will examine next.

JSON-RPC 2.0: The Transport Layer

JSON-RPC 2.0 is the message protocol underlying LSP. It is a lightweight, language-neutral RPC protocol defined by the JSON-RPC Working Group. LSP builds on top of it by defining specific method names, parameters and response structures.

Every JSON-RPC message is a JSON object that begins with a jsonrpc field set to the string 2.0. Messages come in three flavors: requests, responses and notifications.

A request is sent by either the client or the server to invoke an operation. It contains a method name and optional parameters, and it always includes an identifier (id) so the recipient can match the response to the request. The identifier can be any type: an integer, a string, or even an object. The critical rule is that it must be unique within a single session. Here is a canonical example of an LSP initialize request as a JSON-RPC message:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "processId": 12345,
    "rootUri": "file:///home/user/project",
    "capabilities": {
      "textDocument": {
        "completion": {
          "completionItem": {
            "snippetSupport": true
          }
        }
      }
    }
  }
}
```

A response is sent back to the sender of a request. It contains the same id as the request and either a result field (on success) or an error field (on failure), never both. A successful response to the initialize request might look like:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "capabilities": {
      "textDocumentSync": {
        "change": 2
      },
      "completionProvider": {
        "triggerCharacters": [".", ":"]
      }
    },
    "serverInfo": {
      "name": "minilsp",
      "version": "0.1.0"
    }
  }
}
```

A notification is a one-way message. It looks like a request but has no id field, and the receiver must not send a response. Notifications are used for events that do not require confirmation, such as document changes or log messages. An example of a document change notification:

```
{
  "jsonrpc": "2.0",
  "method": "textDocument/didChange",
  "params": {
    "textDocument": {
      "uri": "file:///home/user/project/main.ms",
      "version": 3
    },
    "contentChanges": [
      {
        "range": {
          "start": {"line": 5, "character": 10},
          "end": {"line": 5, "character": 15}
        },
        "text": "42"
      }
    ]
  }
}
```

JSON-RPC defines standard error codes for protocol-level failures: -32700 for parse errors, -32600 for invalid requests, -32601 for methods not found, -32602 for invalid parameters, and -32603 for internal errors. The range -32000 to -32099 is reserved for server-defined errors. LSP defines additional error codes specific to language server operations, such as -32803 for content modifications on readonly documents.

The transport layer (how JSON-RPC messages are physically sent over stdio or TCP) is defined by LSP itself, not by JSON-RPC. Over stdio, each message is framed as a header followed by the JSON content. The header contains a Content-Length field that specifies the length of the JSON payload in bytes, followed by an optional Content-Type field. Header fields are separated by newlines, and the header itself is terminated by an extra newline. This framing allows the receiver to read complete messages even when they arrive in partial chunks over the pipe.

The Initialization Handshake

Every LSP session begins with an initialization handshake. This is a two-step process that establishes the protocol version, negotiates capabilities, and configures both sides for the remainder of the session.

Step one is the initialize request, sent by the client immediately after starting the server. The client includes InitializeParams, which contains the client's

capabilities, the root URI of the workspace (if applicable), the process ID of the client, and locale information. The server must respond with InitializeResult, which declares its capabilities and optionally its identity (name and version).

The initialize request is special. If the server receives any other request or notification before initialize, it must respond with a JSON-RPC error of code -32002 (server not initialized). This ensures that capability negotiation always happens first.

After the server responds to initialize, the client sends an initialized notification (with no id). This tells the server that the client has processed the capabilities and is ready to send regular requests. The server may use the initialized handler to perform any startup work: loading workspace configuration, indexing files, or setting up background workers.

The initialization handshake is critical because it is where both sides learn what they can ask of each other. The client might support snippets in completions but not semantic tokens; the server might support go-to-definition but not code actions. The capability objects exchanged during initialization record these differences, and both sides must respect them throughout the session.

Capability Negotiation and Feature Discovery

Capabilities are the heart of LSP's extensibility. They are nested, self-describing objects that declare what features each side supports. The client sends ClientCapabilities in the initialize params, and the server responds with ServerCapabilities in the initialize result.

Client capabilities describe what the client can handle. For example:

- textDocument.completion.completionItem.snippetSupport indicates that the client can expand snippet syntax in completions.
- textDocument.semanticTokens indicates support for semantic token highlighting.
- workspace.applyEdit indicates that the workspace supports applyEdit requests (the server can ask the client to perform edits).
- workspace.workspaceEdit.documentChanges indicates support for atomic document changes in workspace edits.

A client may also advertise experimental capabilities that are not part of the official specification, under a top-level experimental field. Servers should treat experimental capabilities as optional and undefined.

Server capabilities describe what the server provides. The structure is extensive, with fields for each LSP feature:

- textDocumentSync configures document synchronization behavior. It can be a single integer (TextDocumentSyncKind.Full or Incremental) or a detailed object specifying open/close, change and save behavior.
- completionProvider declares that the server provides completions and may specify trigger characters.
- hoverProvider, definitionProvider, referencesProvider, documentSymbolProvider, workspaceSymbolProvider, renameProvider, codeActionProvider, documentFormattingProvider, signatureHelpProvider, semanticTokensProvider, codeLensProvider each indicate support for the corresponding feature.

Each capability field follows a strict pattern: if it is absent or null, the server does not support that feature. If it is true or an options object, the server supports it. The options object may include additional configuration, such as the list of trigger characters for completions.

Capability negotiation is one-directional for initialization but two-directional in spirit. The server reads the client's capabilities to decide what to advertise back. For example, a server that supports snippets will only enable snippet syntax in completions if the client's capabilities indicate snippetSupport is true. Similarly, a server should not use dynamic registration (registerCapability requests) if the client does not indicate support for it.

Here is a minimal but realistic server capabilities object that MiniLSP will eventually return:

```
{
  "textDocumentSync": {
    "openClose": true,
    "change": 2,
    "save": true
  },
  "completionProvider": {
    "resolveProvider": false,
    "triggerCharacters": [".", ":", " "]
  },
  "hoverProvider": true,
  "definitionProvider": true,
  "referencesProvider": true,
  "documentSymbolProvider": true,
  "workspaceSymbolProvider": true,
  "renameProvider": {
    "prepareProvider": true
  },
  "codeActionProvider": {
    "codeActionKinds": ["quickfix", "refactor"]
  },
  "documentFormattingProvider": true,
  "semanticTokensProvider": {
    "legend": {
      "tokenTypes": ["variable", "function", "type", "keyword"],
      "tokenModifiers": ["declaration", "readonly"]
    },
    "range": true,
    "full": true
  },
  "codeLensProvider": {
    "resolveProvider": false
  },
  "signatureHelpProvider": {
    "triggerCharacters": ["(", ","]
  },
  "positionEncoding": "utf-8"
}
```

This object says: we support incremental document sync with open/close and save events; we provide completions triggered by dots, colons and spaces; we support hover, definition, references, document symbols, workspace symbols, rename, code actions, formatting, semantic tokens, code lenses and signature help; and we prefer UTF-8 position encoding. Every feature we will implement in this book corresponds to one of these capability fields.

The last field, positionEncoding, deserves special attention. It was added in LSP 3.17 to allow servers and clients to negotiate the character encoding used

for positions and ranges. Before 3.17, all positions used UTF-16 code units, which is problematic for languages or servers that work natively with UTF-8 byte offsets (like Rust). With LSP 3.17+, the server can advertise positionEncoding: "utf-8" if it prefers UTF-8 byte offsets, and clients that support it will respect that preference. MiniLSP will use UTF-8 for simplicity and correctness.

Understanding capabilities is essential for building a correct language server. You cannot enable a feature without advertising it, and you cannot use a feature without checking that the other side supports it. The remainder of this book will show how to wire each capability to concrete Rust code, starting with project setup in the next chapter.

Chapter 3: Rust Tooling and Project Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Choosing Your LSP Crate: lsp-server and Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Project Structure and Cargo Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Async Runtime Selection: Tokio vs. Others

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

First Compile: Hello, Language Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 4: LSP Messages and Transport

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Requests, Responses and Notifications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Message Routing and Handler Registration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Error Handling and JSON-RPC Errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Building a Minimal Request Handler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 5: Initialization and Capabilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The initialize Request and Its Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Declaring Server Capabilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Handling initialized Notifications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Exposing Workspace and Document Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 6: Document Synchronization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Opening, Changing, Saving and Closing Documents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Incremental vs. Full Document Updates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

URI Handling and Encoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Building an In-Memory Document Store

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 7: Positions, Ranges and Text Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

LSP Position, Range and Location Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Converting Between Byte Offsets and Positions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

TextEdit and OptionalVersionedTextDocumentContentChangeEvent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Building a Text Document Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 8: Parsing and Syntax Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Designing a Minimal Language for Our Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Writing a Lexer: Tokenization in Rust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 9: Diagnostics and Error Reporting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Diagnostic Severity, Codes and Ranges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Generating Diagnostics from Parse Errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Semantic Diagnostics: Type Checking Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Publishing Diagnostics with publishDiagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 10: Symbol Indexing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Collecting Symbols from the AST

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Symbol Kinds and Hierarchical Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Building a Fast Symbol Index

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Scope Resolution and Name Lookup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 11: Document Symbols and Workspace Symbols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The documentSymbol Request: Generating Outlines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The workspaceSymbol Request: Global Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Symbol Filters and Result Scoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Performance Considerations for Large Workspaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 12: Go to Definition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The definition Request and Its Protocol

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Resolving References to Definitions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Handling Multiple Definitions and Ambiguities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Cross-File Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 13: Find All References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The references Request and ReferenceContext

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Walking the AST for Reference Collection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Excluding the Definition Itself

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Scaling Reference Search Across Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 14: Hover Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The hover Request and MarkupKinds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Resolving Symbols Under the Cursor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Formatting Hover Content with Markdown

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Including Type Information and Signatures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 15: Completions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The completion Request and CompletionItem Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Trigger Characters and Context-Aware Suggestions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

CompletionItemKinds and Detail Formatting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Completion Item Resolve and Snippets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 16: Signature Help

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The signatureHelp Request and Trigger Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Identifying the Active Parameter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

SignatureInformation and ParameterInformation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Handling Nested Calls and Overloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 17: Code Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The codeAction Request and Diagnostic Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Quick Fixes: Converting Diagnostics to Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Refactoring Code Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Command Execution and CodeActionResolve

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 18: Rename

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The rename and prepareRename Requests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Collecting All References Before Renaming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Preparing the WorkspaceEdit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Validation and Conflict Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 19: Semantic Tokens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Semantic Tokens Overview and Encoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Token Types and Modifiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Generating Tokens from the AST

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Incremental Token Updates and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 20: Code Lenses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The codeLens Request and Its Use Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Generating Lenses from AST Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

CodeLens Resolve and Dynamic Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Real-World Examples: Run, Test, References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 21: Code Formatting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The formatting Requests and FormattingOptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Building a Formatter from the AST

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Preserving Source Formatting Where Possible

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Formatter Configuration and Style Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 22: Concurrency and Async Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The Event Loop and Request Processing Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Async/Await Patterns in Language Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Thread Pools and Parallel Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Avoiding Deadlocks and Priority Inversion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 23: Performance and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Profiling a Language Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Incremental Parsing and AST Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Efficient Data Structures for Indexing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Debouncing and Request Throttling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 24: Testing the Language Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Unit Testing Parsers and Analyzers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Integration Testing LSP Requests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Protocol-Level Testing with Mock Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

End-to-End Testing with Real Editors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 25: Logging, Debugging and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

LSP Logging Channels and Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Structured Logging with tracing and tracing-subscriber

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Debugging Common Language Server Bugs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Remote Debugging and Attach Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Chapter 26: Packaging, Distribution and Editor Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Building and Cross-Compiling for Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Publishing as a GitHub Release

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Visual Studio Code Extension Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Neovim, Emacs and Other LSP Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Conclusion: The Road Ahead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

From Educational to Production: What Is Missing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Advanced Topics: Inlay Hints, Inline Values, Type Hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

Contributing to rust-analyzer and Other Projects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

The Future of Language Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingarustlanguageserver.