

Basic LSP Implementation

Author: Nikita Vetluzhskikh

Date: July 7, 2025

[GitHub repository](#)

Introduction

This project demonstrates a simple LSP implementation using standard I/O. It includes dictionary-based word completion and basic diagnostics, develop within one week.

1 Project Structure

The project consists of the following files and directories:

- `CompilerHandler/` - Compiler processing
 - `Compiler.h` - main Class for compiler running
 - `CompilerOutput.h` - structures for JSON like output
- `external/` — Third-party dependencies.
 - `json.hpp` — Single-header JSON parser ([nlohmann/json](#)).
 - `test-VSCode-external-base` — VSCode client used for testing the server.
- `Logger/` — Logging module.
 - `Logger.cpp` — Logger implementation.
 - `Logger.h` — Logger interface.
- `Messages/` — Message types and protocol logic.
 - `methods/` — Protocol-specific methods.
 - * `InitializeResult.h` — Initialization result structure.
 - * `ServerInfo.h` — Server metadata structure.
 - `TextDocument/` — Text document handling logic.
 - * `Params/` — Request parameter structures (e.g., `CompletionParams.h`, `DidChangeTextParams.h`).
 - * `Result/` — Completion and diagnostic structures
 - * Other files: `TextDocument.cpp`, `TextDocument.h`, `TextDocumentItem.h`, `TextDocumentSync.h`, etc.
 - `DictionaryWords.*` — Dictionary lookup logic.
 - `Message.h` — Common message definitions.
 - `NotificationMessage.h`, `RequestMessage.h`, `ResponseMessage.h` — Message categories.
 - `ResponseResult.h` — Structure for result responses.
 - `Range.h` — Text range representation.
- `main.cpp` — Server application entry point.
- `CMakeLists.txt` — Build configuration.

2 Dependencies

- Language: C++
- OS: Ubuntu 24.04.2 LTS
- Libraries: STL, [nlohmann/json \(3.11.2\)](#)

3 How It Works

The client starts the Language Server Protocol and sends an `initialize` request. Upon receiving a response from the server with its capabilities, it sends an `initialized` notification, confirming the setup.

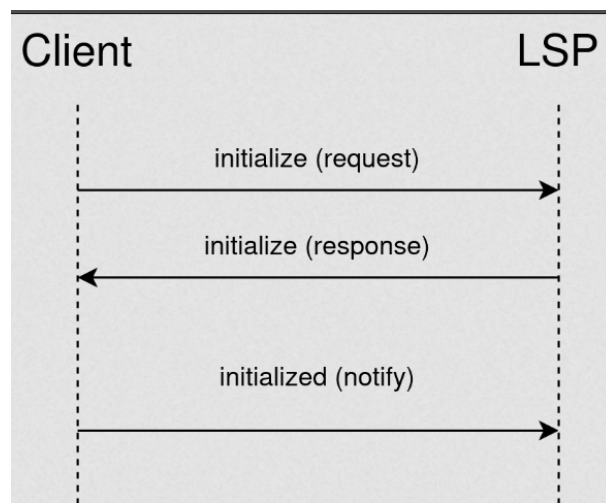


Figure 1: Initialization diagram

Communication uses standard input and output (`stdin` and `stdout`). `stderr` can be used for logging purposes.

When a file is edited, the client sends a `textDocument/completion` request (for word suggestions (At the moment work only with Keywords)) and a `textDocument/diagnostic` request (launch a compiler).

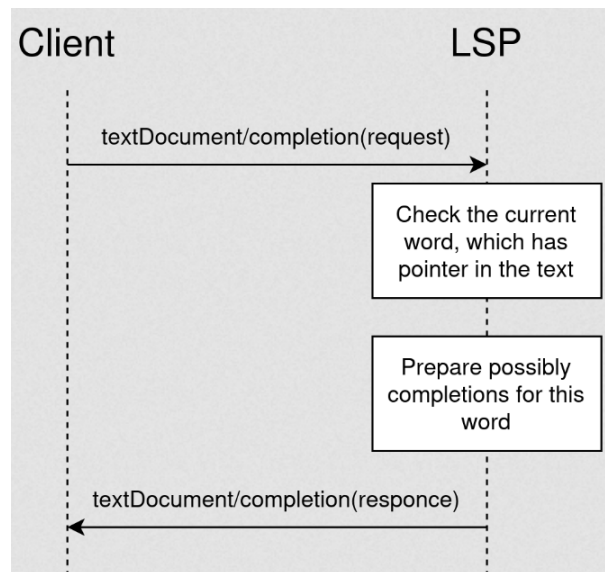


Figure 2: Completion workflow diagram

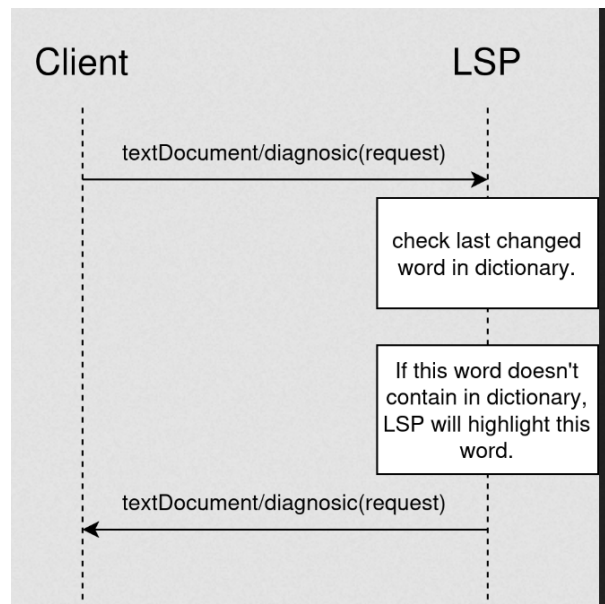


Figure 3: Diagnostic workflow diagram

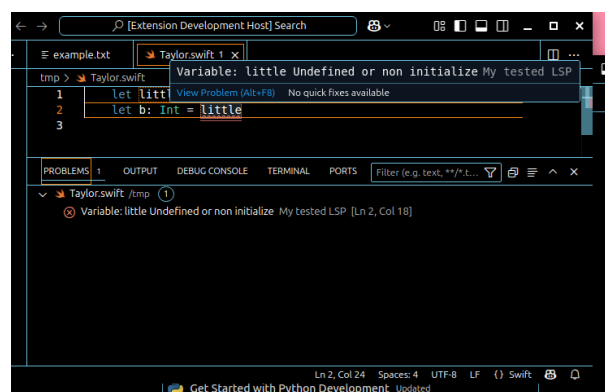


Figure 4: Highlighting showcase

When user opens a new file, VSCode sends `textDocument/didOpen` notification, and the server saves this document filepath to map – filepath is key, last changes is value.

Notification `textDocument/didChange` updates the value in the map.

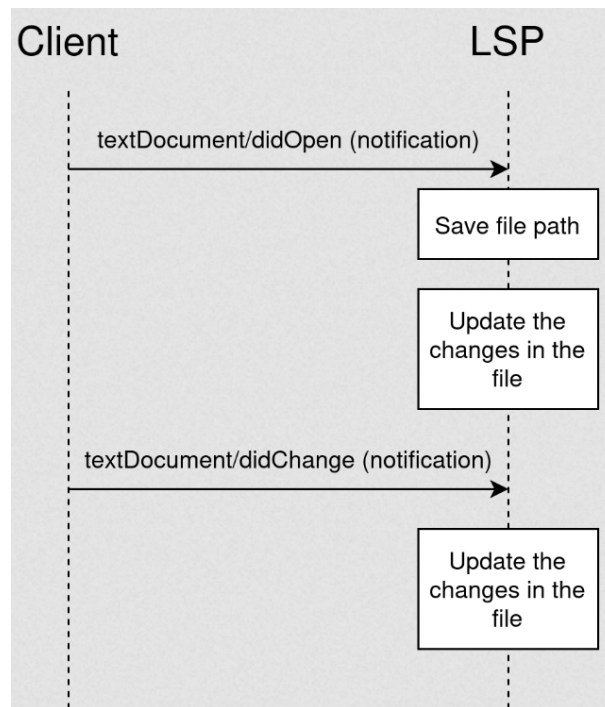


Figure 5: `textDocument/didOpen` and `didChanges`

4 Language syntax

Language for my compiler, it's a sublanguage from swift. This language has a basic language structures.

```

1  // comments
2  // variable declaration
3  var a
4  var b: Int
5  var c: Double = 2.0
6  // variable declaration with expression
7  var d = (2 + 4) 3.5
8  // const declaration
9  let a
10
11 // while cycle
12 while ( d > c) {
13   write("d more than c")
14 }
15
16 // conditions
17 if ( 5 <= 10 ){
18     write("5 less than 10")
19 }
20 else {
21     write("Is 5 less than 10? Something go wrong...")
  
```

```

22 }
23
24 // function creating
25 func foo(){
26     return
27 }
28 // function with arguments
29 func foo(var_name some_id: Int){
30     return
31 }
32 //function with returning type
33 func boo() -> Int {
34     return 2
35 }
36 // function call
37 write("Hello world)
38
39 // internal functions
40 // more in the compiler's documentation

```

5 Compiler processing

The server runs the compiler when the textDocument/diagnostic function is called. If an error occurs, the compiler returns a message in JSON format. The compilation is terminated when the first error is detected.

Details about the compiler implementation can be found in its documentation prepared for the project defence.

The following error types are provided in my implementation of the compiler:

```

1  enum ErrorCode{
2      ER_NONE          = 0,
3      ER_LEX           = 1, // Lexical error when the token cannot be identified
4      ER_SYNTAX        = 2, // Syntax error when token order is violated
5      ER_UNDEF_FUNC_OR_REDEF_VAR = 3,
6      ER_PARAMS        = 4, // Semantic error in the program - wrong number
7                          // type of parameters in the function call
8                          // or wrong type of return value from
9                          // the function
10     ER_UNDEF_VAR_OR_NOTINIT_VAR = 5, // Semantic error in the program - using an
11                                     // undefined or uninitialized variable
12     ER_FUNC_RETURN     = 6, // Semantic error in the program - missing/absent
13                             // expression in the function return statement
14     ER_TYPE_COMP       = 7, // Semantic type compatibility error in arithmetic string
15                             // and relational expressions
16     ER_INFERENCE       = 8, // Semantic type derivation error - the type of the
17                             // variable or parameter is not specified and cannot be
18                             // derived from the expression used
19     ER_OTHER_SEM       = 9, // Other semantic errors
20     ER_PARAMS_ARGS_MISMATCH = 40, // Concretisation of ER_PARAMS error for the server
21     ER_PARAMS_TYPE_MISMATCH = 41, // Concretisation of ER_PARAMS error for the server
22     ER_INTERNAL        = 99 // Internal compiler error i.e. not influenced by the
23                             //input program
24 };

```

JSON variables containing additional information are provided for all errors. At the moment, not all errors contain exact data - such errors are marked with the message "**Unresolved error**".

The server can handle errors of types `ER_UNDEF_OR_NON_INIT_VAR` and `ER_SYNTAX`.

5.1 `ER_UNDEF_OR_NON_INIT_VAR`

The compiler returns the following data: ...

```

1 { // json message for ER_SYNTAX
2     "error_code": 2,
3     "message": "Syntax error: Waiting keyword or variable doesnt have nil_flag",
4     "line": 2,
5     "char_pos": 10,
6     "token_type": 33, // enum with type of token
7     "token_string": "null" // if the compiler can find out the token identifier
8 }

1 // the code which throws error ER_SYNTAX
2 var abc:

```

Thanks to the already detected variable name, it is sufficient to locate this variable in the specified line and indicate its position to the text editor.

Thanks to the already found variable name, you only need to find this variable in the specified line and pass the text editor the exact position of the variable in the text.

5.2 `ER_SYNTAX`

If the compiler fails to match the input text with LL rules, it raises a syntax error. In this case, it reports which token was received and which was expected (sometimes only the text of the error message is returned).

```

1 { // json message for ER_SYNTAX
2     "error_code": 2,
3     "message": "Syntax error: Waiting keyword or variable doesnt have nil_flag",
4     "line": 2,
5     "char_pos": 10,
6     "token_type": 33, // enum with type of token
7     "token_string": "null" // if the compiler can find out the token identifier
8 }

1 // the code which throws error ER_UNDEF_VAR_OR_NOT_INIT_VAR
2 let a: Int
3 let b: Int = a

```

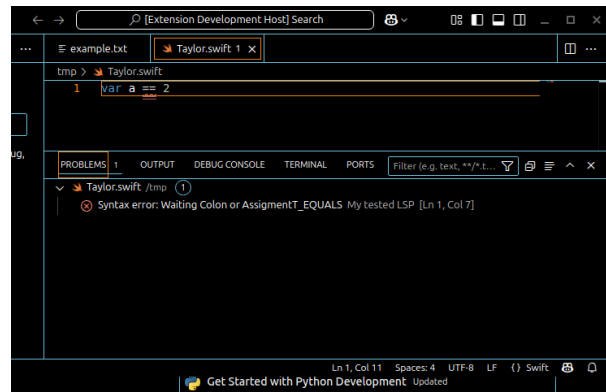


Figure 6: ER_SYNTAX showcase

5.3 Runtime error

In some cases, the compiler would go into infinite recursion. If this happens, the server waits 10 seconds, resets the compiler and sends an error.

```

1  // code with infinite recursion for example
2  Double
3
4  var a :Int

```

6 Implemented Features

6.1 initialize

Establishes initial communication. Returns `ServerCapabilities` and `ServerInfo`.

```

1  { // response from server
2    "id": 0,
3    "jsonrpc": "2.0",
4    "result": {
5      "capabilities": {
6        "completionProvider": {},
7        "diagnosticProvider": {
8          "interFileDependencies": false,
9          "workspaceDiagnostics": false
10       },
11       "textDocumentSync": 1
12     },
13     "serverInfo": {
14       "name": "my-lsp-server",
15       "version": "0.0.1"
16     }
17   }
18 }

```

6.2 textDocument/completion

Returns word suggestions. If results exceed 1000 items, the server sets `isIncomplete` to `true`.

```
1 { //request from client
2   "id":14,
3   "jsonrpc":"2.0",
4   "method":"textDocument/completion",
5   "params":{
6     "context":{
7       "triggerKind":3
8     },
9     "position":{
10      "character":8,
11      "line":2
12    },
13    "textDocument":{
14      "uri":"file:///tmp/example.txt"
15    }
16  }}

1 { //responce from client
2   "id":14,
3   "jsonrpc":"2.0",
4   "result":{
5     "isIncomplete":false, //count of items less than 1000 -> false
6     "items":[
7       {"label":"truths"},
8       {"label":"truth's"},
9       {"label":"truthiness"},
10      // many words
11      {"label":"truckled"},
12      {"label":"truncating"}
13    ]
14  }
15 }
```

6.3 textDocument/didOpen

Notifies the server that a file was opened. The server stores the file path.

```
1 { // notification from client
2   "jsonrpc":"2.0",
3   "method":"textDocument/didOpen",
4   "params":{
5     "textDocument":{
6       "languageId":"plaintext",
7       "text":"hello\n\ntest \n\nthis the first file\n\nthis the fisst file",
8       "uri":"file:///tmp/example.txt",
9       "version":1
10    }
11  }
12 }
```


6.4 textDocument/didChange

Notifies the server of changes on an opened file.

```
1 { // notification from client
2   "jsonrpc": "2.0",
3   "method": "textDocument/didChange",
4   "params": {
5     "contentChanges": [
6       {
7         "text": "hello\n\ntest trust\n\nthis the first file\n\nthis the fisst file"
8       }
9     ],
10    "textDocument": {
11      "uri": "file:///tmp/example.txt",
12      "version": 12
13    }
14  }}
```

6.5 textDocument/diagnostic

Client requests file diagnostics. The server analyzes text and returns any found issues.

```
1 { // responce from server
2   "id": 1,
3   "jsonrpc": "2.0",
4   "result": {
5     "items":
6     [{
7       "message": "fisst is not in our dictionary",
8       "range": {
9         "end": {
10          "character": 14,
11          "line": 6
12        },
13        "start": {
14          "character": 9,
15          "line": 6
16        }
17      },
18      "severity": 1,
19      "source": "My tested LSP"
20    }],
21    "kind": "full"
22  }
23 }
```

6.6 shutdown

Client send shutdown, and the server stops responding to any messages from the client except initialise or exit

6.7 exit

The server finish the process

7 Install

Run

```
1 git clone --recurse-submodule https://github.com/SyruSTR/test_lsp.git {DIRECTORY_PATH}
```

8 Running the Project

Run `start.sh` to launch VSCode and start the client. Press F5 to run the LSP client.

9 Debugging

Use `./start.sh -d` to start in debug mode. Then use `Attach to Process...` in Clion to begin debugging.