# pygls Documentation

Open Law Library

Oct 24, 2022

# Contents

pygls (pronounced like “pie glass”) is a generic implementation of the Language Server Protocol written in the Python programming language. It allows you to write your own language server in just a few lines of code.

CHAPTER 1

# Features

- cross-platform support
- TCP/IP and STDIO communication
- runs in asyncio event loop
- register LSP features and custom commands as:
  - asynchronous functions (coroutines)
  - synchronous functions
  - functions that will be executed in separate thread
- thread management
- in-memory workspace with _full_ and _incremental_ document updates
- type-checking
- good test coverage

CHAPTER 2

# Python Versions

*pygls* works with Python 3.7+.

CHAPTER 3

# User Guide

## 3.1 Getting Started

This document explains how to install *pygls* and get started writing language servers that are based on it.

**Note:** Before going any further, if you are not familiar with *language servers* and *Language Server Protocol*, we recommend reading following articles:

- Language Server Protocol Overview
- Language Server Protocol Specification
- Language Server Protocol SDKs

### 3.1.1 Installation

To get the latest release from *PyPI*, simply run:

```
pip install pygls
```

Alternatively, *pygls* source code can be downloaded from our GitHub page and installed with following command:

```
python setup.py install
```

### 3.1.2 Quick Start

#### Spin the Server Up

*pygls* is a language server that can be started without writing any additional code:

```
from pygls.server import LanguageServer

server = LanguageServer()

server.start_tcp('127.0.0.1', 8080)
```

After running the code above, server will start listening for incoming `Json RPC` requests on `http://127.0.0.1:8080`.

#### Register Features and Commands

*pygls* comes with an API for registering additional features like `code completion`, `find all references`, `go to definition`, etc.

```
@server.feature(COMPLETION, CompletionOptions(trigger_characters=[',']))
def completions(params: CompletionParams):
    """Returns completion items."""
    return CompletionList(
        is_incomplete=False,
        item=[
            CompletionItem(label='Item1'),
            CompletionItem(label='Item2'),
            CompletionItem(label='Item3'),
        ]
    )
```

... as well as custom commands:

```
@server.command('myVerySpecialCommandName')
def cmd_return_hello_world(ls, *args):
    return 'Hello World!'
```

Features that are currently supported by the LSP specification can be found in pygls.lsp.methods module, while corresponding request/response classes can be found in pygls.lsp.types module.

### 3.1.3 Advanced usage

To reveal the full potential of *pygls* (`thread management`, `coroutines`, `multi-root workspace`, `TCP/STDIO communication`, etc.) keep reading.

### 3.1.4 Tutorial

We recommend completing the *tutorial*, especially if you haven't worked with language servers before.

## 3.2 Tutorial

In order to help you with *pygls*, we have created a simple json-extension example.

### 3.2.1 Prerequisites

In order to setup and run the example extension, you need following software installed:

- Visual Studio Code editor
- Python 3.7+
- vscode-python extension
- A clone of the pygls repository

---

**Note:** If you have created virtual environment, make sure that you have *pygls* installed and selected appropriate python interpreter for the *pygls* project.

---

### 3.2.2 Running the Example

For a step-by-step guide on how to setup and run the example follow README.

### 3.2.3 Hacking the Extension

When you have successfully setup and run the extension, open server.py and go through the code.

We have implemented following capabilities:

- `textDocument/completion` feature
- `countDownBlocking` command
- `countDownNonBlocking` command
- `textDocument/didChange` feature
- `textDocument/didClose` feature
- `textDocument/didOpen` feature
- `showConfigurationAsync` command
- `showConfigurationCallback` command
- `showConfigurationThread` command

When running the extension in *debug* mode, you can set breakpoints to see when each of above mentioned actions gets triggered.

Visual Studio Code supports *Language Server Protocol*, which means, that every action on the client-side, will result in sending request or notification to the server via JSON RPC.

#### Debug Code Completions

Set a breakpoint inside `completion` function and go back to opened *json* file in your editor. Now press `ctrl + space` (`control + space` on mac) to show completion list and you will hit the breakpoint. When you continue debugging, the completion list pop-up won't show up because it was closing when the editor lost focus.

Similarly, you can debug any feature or command.

Keep the breakpoint and continue to the next section.

### Blocking Command Test

In order to demonstrate you that blocking the language server will reject other requests, we have registered a custom command which counts down 10 seconds and sends notification messages to the client.

1. Press **F1**, find and run `Count down 10 seconds [Blocking]` command.
2. Try to show *code completions* while counter is still ticking.

Language server is **blocked**, because `time.sleep` is a **blocking** operation. This is why you didn't hit the breakpoint this time.

---

**Hint:** To make this command **non blocking**, add `@json_server.thread()` decorator, like in code below:

```
@json_server.thread()
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_BLOCKING)
def count_down_10_seconds_blocking(ls, *args):
    # Omitted
```

*pygls* uses a **thread pool** to execute functions that are marked with a `thread` decorator.

---

### Non-Blocking Command Test

Python 3.4 introduced *asyncio* module which allows us to use asynchronous functions (aka *coroutines*) and do cooperative multitasking. Using the *await* keyword inside your coroutine will give back control to the scheduler and won't block the main thread.

1. Press **F1** and run the `Count down 10 seconds [Non Blocking]` command.
2. Try to show *code completions* while counter is still ticking.

Bingo! We hit the breakpoint! What just happened?

The language server was **not blocked** because we used `asyncio.sleep` this time. The language server was executing *just* in the *main* thread.

### Text Document Operations

Opening and closing a JSON file will display appropriate notification message in the bottom right corner of the window and the file content will be validated. Validation will be performed on content changes, as well.

### Show Configuration Data

There are *three* ways for getting configuration section from the client settings.

---

**Note:** *pygls*' built-in coroutines are suffixed with *async* word, which means that you have to use the *await* keyword in order to get the result (instead of *asyncio.Future* object).

---

- **Get the configuration inside a coroutine**

```
config = await ls.get_configuration_async(ConfigurationParams([
    ConfigurationItem('', JsonLanguageServer.CONFIGURATION_SECTION)
]))
```

- **Get the configuration inside a normal function**

We already saw that we *don't* want to block the main thread. Sending the configuration request to the client will result with the response from it, but we don't know when. You have to pass *callback* function which will be triggered once response from the client is received.

```
def _config_callback(config):
    try:
        example_config = config[0].exampleConfiguration

        ls.show_message(
            f'jsonServer.exampleConfiguration value: {example_config}'
        )

    except Exception as e:
        ls.show_message_log(f'Error ocurred: {e}')

ls.get_configuration(ConfigurationParams([
    ConfigurationItem('', JsonLanguageServer.CONFIGURATION_SECTION)
]), _config_callback)
```

As you can see, the above code is hard to read.

- **Get the configuration inside a threaded function**

Blocking operations such as `future.result(1)` should not be used inside normal functions, but to increase the code readability, you can add the *thread* decorator to your function to use *pygls*' *thread pool*.

```
@json_server.thread()
@json_server.command(JsonLanguageServer.CMD_SHOW_CONFIGURATION_THREAD)
def show_configuration_thread(ls: JsonLanguageServer, *args):
    """Gets exampleConfiguration from the client settings using a thread pool."""
    try:
        config = ls.get_configuration(ConfigurationParams([
            ConfigurationItem('', JsonLanguageServer.CONFIGURATION_SECTION)
        ])).result(2)

        # ...
```

This way you won't block the main thread. *pygls* will start a new thread when executing the function.

#### Modify the Example

We encourage you to continue to *advanced section* and modify this example.

## 3.3 Advanced Usage

### 3.3.1 Language Server

The language server is responsible for receiving and sending messages over the Language Server Protocol which is based on the Json RPC protocol.

#### Connections

*pygls* supports *TCP* and socket *STDIO* connections.

### TCP

TCP connections are usually used while developing the language server. This way the server can be started in *debug* mode separately and wait for the client connection.

**Note:** Server should be started **before** the client.

The code snippet below shows how to start the server in *TCP* mode.

```
from pygls.server import LanguageServer

server = LanguageServer()

server.start_tcp('127.0.0.1', 8080)
```

### STDIO

STDIO connections are useful when client is starting the server as a child process. This is the way to go in production.

The code snippet below shows how to start the server in *STDIO* mode.

```
from pygls.server import LanguageServer

server = LanguageServer()

server.start_io()
```

### WEBSOCKET

WEBSOCKET connections are used when you want to expose language server to browser based editors.

The code snippet below shows how to start the server in *WEBSOCKET* mode.

```
from pygls.server import LanguageServer

server = LanguageServer()

server.start_websocket('0.0.0.0', 1234)
```

### Logging

Logs are useful for tracing client requests, finding out errors and measuring time needed to return results to the client.

*pygls* uses built-in python *logging* module which has to be configured before server is started.

Official documentation about logging in python can be found here. Below is the minimal setup to setup logging in *pygls*:

```
import logging

from pygls.server import LanguageServer

```

(continues on next page)

(continued from previous page)

```
logging.basicConfig(filename='pygls.log', filemode='w', level=logging.DEBUG)

server = LanguageServer()

server.start_io()
```

#### Overriding `LanguageServerProtocol`

If you have a reason to override the existing `LanguageServerProtocol` class, you can do that by inheriting the class and passing it to the `LanguageServer` constructor.

### 3.3.2 Features

What is a feature in *pygls*? In terms of language servers and the Language Server Protocol, a feature is one of the predefined methods from LSP specification, such as: *code completion*, *formatting*, *code lens*, etc. Features that are available can be found in pygls.lsp.methods module.

#### *Built-In* Features

*pygls* comes with following predefined set of Language Server Protocol (LSP) features:

- The initialize request is sent as a first request from client to the server to setup their communication. *pygls* automatically computes registered LSP capabilities and sends them as part of `InitializeResult` response.
- The shutdown request is sent from the client to the server to ask the server to shutdown.
- The exit notification is sent from client to the server to ask the server to exit the process. *pygls* automatically releases all resources and stops the process.
- The textDocument/didOpen notification will tell *pygls* to create a document in the in-memory workspace which will exist as long as document is opened in editor.
- The textDocument/didChange notification will tell *pygls* to update the document text. *pygls* supports _full_ and _incremental_ document changes.
- The textDocument/didClose notification will tell *pygls* to remove a document from the in-memory workspace.
- The workspace/didChangeWorkspaceFolders notification will tell *pygls* to update in-memory workspace folders.

### 3.3.3 Commands

Commands can be treated as a *custom features*, i.e. everything that is not covered by LSP specification, but needs to be implemented.

### 3.3.4 API

#### *Feature* and *Command* Advanced Registration

*pygls* is a language server which relies on *asyncio event loop*. It is *asynchronously* listening for incoming messages and, depending on the way method is registered, applying different execution strategies to respond to the client.

Depending on the use case, *features* and *commands* can be registered in three different ways.

To make sure that you fully understand what is happening under the hood, please take a look at the *tutorial*.

---

**Note:** *Built-in* features in most cases should *not* be overridden. Instead, register the feature with the same name and it will be called immediately after the corresponding built-in feature.

---

### *Asynchronous* Functions (*Coroutines*)

*pygls* supports `python 3.7+` which has a keyword `async` to specify coroutines.

The code snippet below shows how to register a command as a coroutine:

```
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_NON_BLOCKING)
async def count_down_10_seconds_non_blocking(ls, *args):
    # Omitted
```

Registering a *feature* as a coroutine is exactly the same.

Coroutines are functions that are executed as tasks in *pygls*'s *event loop*. They should contain at least one *await* expression (see awaitables for details) which tells event loop to switch to another task while waiting. This allows *pygls* to listen for client requests in a *non blocking* way, while still only running in the *main* thread.

Tasks can be canceled by the client if they didn't start executing (see Cancellation Support).

**Warning:** Using computation intensive operations will *block* the main thread and should be *avoided* inside coroutines. Take a look at *threaded functions* for more details.

### *Synchronous* Functions

Synchronous functions are regular functions which *blocks* the *main* thread until they are executed.

*Built-in features* are registered as regular functions to ensure correct state of language server initialization and workspace.

The code snippet below shows how to register a command as a regular function:

```
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_BLOCKING)
def count_down_10_seconds_blocking(ls, *args):
    # Omitted
```

Registering *feature* as a regular function is exactly the same.

**Warning:** Using computation intensive operations will *block* the main thread and should be *avoided* inside regular functions. Take a look at *threaded functions* for more details.

### *Threaded* Functions

*Threaded* functions are just regular functions, but marked with *pygls*'s `thread` decorator:

```
# Decorator order is not important in this case
@json_server.thread()
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_BLOCKING)
def count_down_10_seconds_blocking(ls, *args):
    # Omitted
```

*pygls* uses its own *thread pool* to execute above function in *daemon* thread and it is *lazy* initialized first time when function marked with `thread` decorator is fired.

*Threaded* functions can be used to run blocking operations. If it has been a while or you are new to threading in Python, check out Python's `multithreading` and GIL before messing with threads.

### Passing Language Server Instance

Using language server methods inside registered features and commands are quite common. We recommend adding language server as a **first parameter** of a registered function.

There are two ways of doing this:

- **ls** (**l**anguage **s**erver) naming convention

Add **ls** as first parameter of a function and *pygls* will automatically pass the language server instance.

```
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_BLOCKING)
def count_down_10_seconds_blocking(ls, *args):
    # Omitted
```

- add **type** to first parameter

Add the **LanguageServer** class or any class derived from it as a type to first parameter of a function

```
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_BLOCKING)
def count_down_10_seconds_blocking(ser: JsonLanguageServer, *args):
    # Omitted
```

Using outer `json_server` instance inside registered function will make writing unit *tests* more difficult.

### Notifications

A *notification* is a request message without the `id` field and server *must not* reply to it. This means that, if your language server received the notification, even if you return the result inside your handler function, the result won't be passed to the client.

The `Language Server Protocol`, unlike `Json RPC`, allows bidirectional communication between the server and the client.

### Configuration

The configuration request is sent from the server to the client in order to fetch configuration settings from the client. When the requested configuration is collected, the client sends data as a notification to the server.

---

**Note:** Although `configuration` is a `request`, it is explained in this section because the client sends back the `notification` object.

---

The code snippet below shows how to send configuration to the client:

```
def get_configuration(self,
                      params: ConfigurationParams,
                      callback: Optional[Callable[[List[Any]], None]] = None
                      ) -> asyncio.Future:
    # Omitted
```

*pygls* has three ways for handling configuration notification from the client, depending on way how the function is registered (described *here*):

- *asynchronous* functions (*coroutines*)

```
# await keyword tells event loop to switch to another task until notification is␣
↪received
config = await ls.get_
↪configuration(ConfigurationParams(items=[ConfigurationItem(scope_uri='doc_uri_here',
↪ section='section')]))
```

- *synchronous* functions

```
# callback is called when notification is received
def callback(config):
    # Omitted

config = ls.get_configuration(ConfigurationParams(items=[ConfigurationItem(scope_uri=
↪'doc_uri_here', section='section')]), callback)
```

- *threaded* functions

```
# .result() will block the thread
config = ls.get_configuration(ConfigurationParams(items=[ConfigurationItem(scope_uri=
↪'doc_uri_here', section='section')])).result()
```

### Show Message

Show message is notification that is sent from the server to the client to display text message.

The code snippet below shows how to send show message notification:

```
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_NON_BLOCKING)
async def count_down_10_seconds_non_blocking(ls, *args):
    for i in range(10):
        # Sends message notification to the client
        ls.show_message(f"Counting down... {10 - i}")
        await asyncio.sleep(1)
```

### Show Message Log

Show message log is notification that is sent from the server to the client to display text message in the output channel.

The code snippet below shows how to send show message log notification:

```
@json_server.command(JsonLanguageServer.CMD_COUNT_DOWN_NON_BLOCKING)
async def count_down_10_seconds_non_blocking(ls, *args):
```

(continues on next page)

(continued from previous page)

```
    for i in range(10):
        # Sends message log notification to the client's output channel
        ls.show_message_log(f"Counting down... {10 - i}")
        await asyncio.sleep(1)
```

### Publish Diagnostics

Publish diagnostics notifications are sent from the server to the client to signal results of validation runs.

Usually this notification is sent after document is opened, or on document content change, e.g.:

```
@json_server.feature(TEXT_DOCUMENT_DID_OPEN)
async def did_open(ls, params: DidOpenTextDocumentParams):
    """Text document did open notification."""
    ls.show_message("Text Document Did Open")
    ls.show_message_log("Validating json...")

    # Get document from workspace
    text_doc = ls.workspace.get_document(params.text_document.uri)

    diagnostic = Diagnostic(
                    range=Range(
                        start=Position(line-1, col-1),
                        end=Position(line-1, col)
                    ),
                    message="Custom validation message",
                    source="Json Server"
                )

    # Send diagnostics
    ls.publish_diagnostics(text_doc.uri, [diagnostic])
```

### Custom Notifications

*pygls* supports sending custom notifications to the client and below is method declaration for this functionality:

```
def send_notification(self, method: str, params: object = None) -> None:
    # Omitted
```

And method invocation example:

```
server.send_notification('myCustomNotification', 'test data')
```

### Workspace

Workspace is a python object that holds information about workspace folders, opened documents and has the logic for updating document content.

*pygls* automatically take care about mentioned features of the workspace.

Workspace methods that can be used for user defined features are:

- Get document from the workspace

```
def get_document(self, doc_uri: str) -> Document:
    # Omitted
```

- Apply edit request

```
def apply_edit(self, edit: WorkspaceEdit, label: str = None) ->␣
↪ApplyWorkspaceEditResponse:
    # Omitted
```

## 3.4 Testing

### 3.4.1 Unit Tests

Writing unit tests for registered features and commands are easy and you don't have to mock the whole language server. If you skipped the advanced usage page, take a look at *passing language server instance* section for more details.

Json Extension example's unit tests might be helpful, too.

### 3.4.2 Integration Tests

Integration tests coverage includes the whole workflow, from sending the client request, to getting the result from the server. Since the *Language Server Protocol* defines bidirectional communication between the client and the server, we used *pygls* to simulate the client and send desired requests to the server. To get better understanding of how setup it, take a look at our test fixtures.