

How to use MCP Inspector

6 min read · Jun 18, 2025



Laurent Kubaski

Follow



Listen



Share



More

This article is part of my [MCP Explained](#) series.



It took me a while to get familiar with the [MCP Inspector](#) so I'm going to share my findings here since those are things that were not clearly documented elsewhere. I'll

also assume 2 things:

- You want to test a MCP Server written in Python using FastMCP 2.0.
- You are using pip and **not** the uv package manager (all the Python examples on modelcontextprotocol.io assume that you're using uv and I'm not the only one to find this a bit surprising).

1. FastMCP 1.0 vs FastMCP 2.0

First, one important point: the MCP Python SDK that is used in all the code examples on https://modelcontextprotocol.io is an *outdated* one that is known as “FastMCP 1.0”.

Instead, you should use “FastMCP 2.0” located here: https://gofastmcp.com/getting-started/welcome

However, FastMCP 2.0 depends on FastMCP 1.0 meaning that **there are 2 FastMCP classes with the exact same name so make sure that you use the right one:**

```
#from mcp.server import FastMCP # this is FastMCP 1.0 -> DO NOT USE IT
from fastmcp import FastMCP # this is FastMCP 2.0
```

Why am I calling this out? Because I've lost 60 minutes of my life trying to understand a cryptic error message that was due to the fact that I was using a FastMCP 2.0 code sample with a FastMCP 1.0 instance.

2. Running MCP Inspector

To run MCP Inspector, simply execute:

```
npx @modelcontextprotocol/inspector
```

In case you're wondering:

- MCP Inspector is a Node.js app and the npx command allows running MCP Inspector without having to permanently install it as a Node.js package.

- More precisely, the MCP Inspector package is installed in a *temporary* location and then executed.

If you get an error when running this command, it's probably because you have not installed Node.js yet: <https://nodejs.org/en/download>

Anyways, after running the command above you'll get something like this:

```
lkubaski@lkubask-ltmbrvd mcp % npx @modelcontextprotocol/inspector

Starting MCP inspector...
🔧 Proxy server listening on 127.0.0.1:6277
🔑 Session token: d57f209535c4de0e0731af108e1f3f5745aad7ea6d48b8bf932e0b12e89a4
Use this token to authenticate requests or set DANGEROUSLY_OMIT_AUTH=true to di

🔗 Open inspector with token pre-filled:
http://localhost:6274/?MCP_PROXY_AUTH_TOKEN=d57f209535c4de0e0731af108e1f3f57
(Auto-open is disabled when authentication is enabled)

🔍 MCP Inspector is up and running at http://127.0.0.1:6274 🚀
```

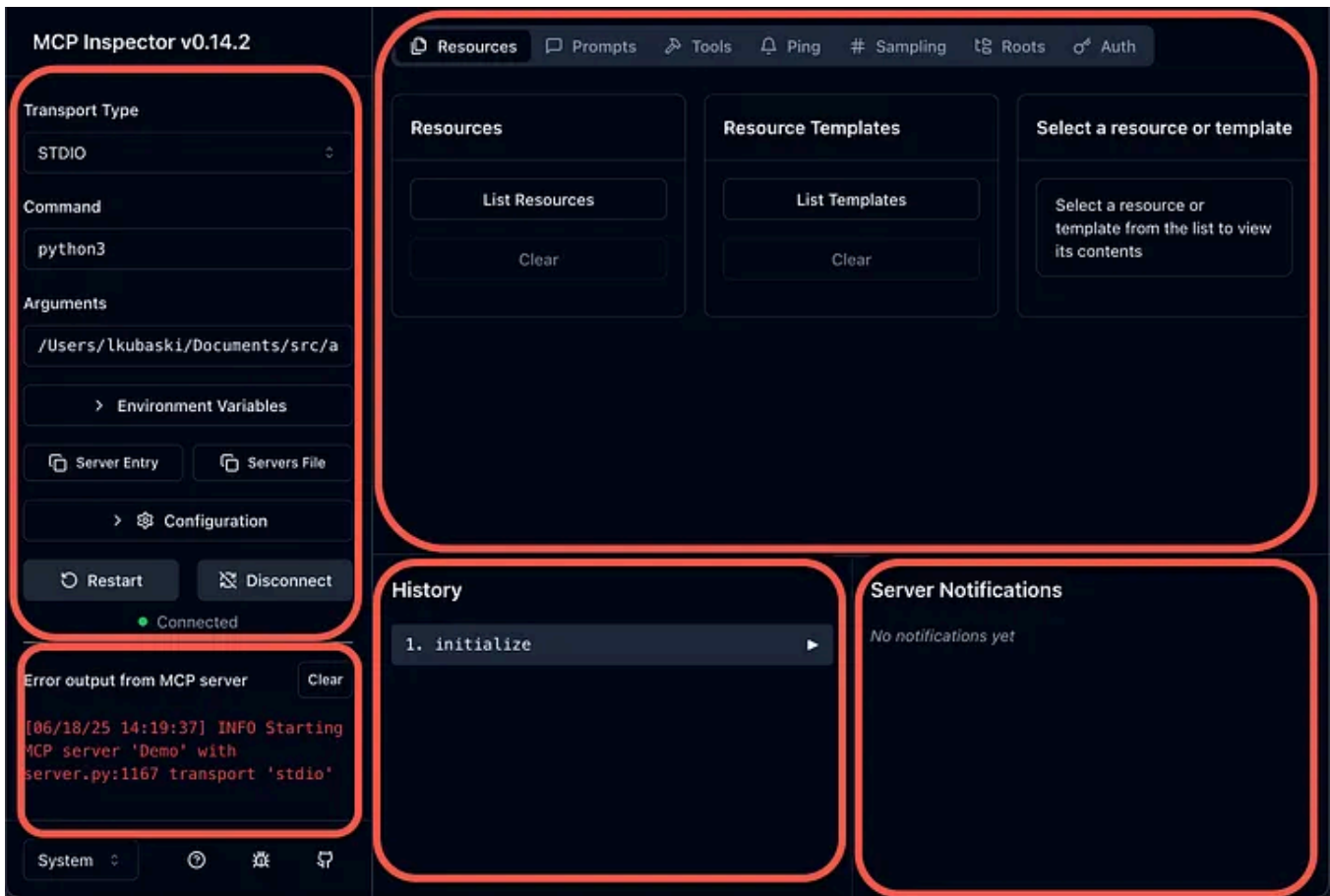
MCP Inspector is now available at <http://127.0.0.1:6274>. Notice that the command

[Open in app](#) ↗

≡ **Medium**



The MCP Inspector UI consists of 5 different panels that I'm going to describe in the following chapters.



The 5 MCP Inspector panels

3. The Server Connection Panel

Official doc: <https://modelcontextprotocol.io/docs/tools/inspector#server-connection-pane>

This is where you connect to the MCP server. The procedure depends on which transport type you want to use:

- STDIO
- SSE (now deprecated so I won't talk about that)
- Streamable HTTP.

3.1. Connecting using STDIO

Assuming that your MCP Server looks like this:

```
from fastmcp import FastMCP

server = FastMCP("Demo")

@server.tool()
```

```
def say_hello(name:str) -> str:
    return f"hello {name}"

if __name__ == "__main__":
    server.run(transport="stdio")
```

Then the MCP Inspector configuration to connect to that server must look like this:

In the screenshot above, I've provided the **absolute path** of the MCP Server Python script:

- You can also just provide **the name of the Python script**, but then the npx command must first be executed from the correct folder.
 - For example, if the argument is just “test_mcp_server.py” then the npx command must be executed from the folder that contains “test_mcp_server.py”.
- You can also use the “-m <module-name>” syntax, but then the npx command must first be executed from the folder that contains the module.

- For example, if the argument is “-m server.test_mcp_server”, then the npx command must be executed from the folder that contains the “server” folder.

Important: when using STDIO, you do NOT need to start the MCP Server yourself. MCP Inspector will take care of this.

3.2. Connecting using Streamable HTTP

Assuming that your MCP Server looks like this:

```
from fastmcp import FastMCP

server = FastMCP("Demo")

@server.tool()
def say_hello(name:str) -> str:
    return f"hello {name}"

if __name__ == "__main__":
    server.run(transport="streamable-http")
```

Then you first need to start your server, either like this:

```
lkubaski@lkubask-ltmbrvd src % python3 ./server/test_mcp_server.py
INFO:      Started server process [87428]
INFO:      Waiting for application startup.
[06/15/25 22:57:46] INFO      StreamableHTTP session manager started
INFO:      Application startup complete.
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

OR by using the FastMCP CLI:

```
lkubaski@lkubask-ltmbrvd server % fastmcp run test_mcp_server.py --transport st
[06/16/25 18:49:23] INFO      Starting MCP server 'Demo' with transport 'streama
INFO:      Started server process [97770]
INFO:      Waiting for application startup.
```

```
INFO:      Application startup complete.  
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Note #1: even though the command window above states that the MCP server is running on <http://127.0.0.1:8000>, it is actually running on <http://127.0.0.1:8000/mcp>

Note #2: the FastMCP CLI completely ignores the “if __name__” section, meaning that the transport needs to be provided on the command line (it defaults to STDIO which is not what you want here).

And finally, the MCP Inspector configuration to connect to that server must look like this:

4. The Error Output Panel

Official doc: none.

Error output panel

The name of that panel is actually incorrect: this actually displays **all** the MCP Server logs, not just the error logs.

For example to display the logs you see in the screenshot above, I've used the following MCP Server:

```
from fastmcp import FastMCP
from fastmcp.utilities.logging import get_logger

logger = get_logger(__name__)

server = FastMCP("Demo")

@server.tool()
async def say_hello(name:str) -> str:
    logger.info(f"logger.info: say_hello(): name={name}")
    logger.warning(f"logger.warning: say_hello(): name={name}")
    logger.debug(f"logger.debug: say_hello(): name={name}")
    logger.error(f"logger.error: say_hello(): name={name}")
    return f"hello {name}"
```

Notice that by default, debug logs are not displayed.

5. The Resources, Prompts, Tools etc... Panel

Official doc: <https://modelcontextprotocol.io/docs/tools/inspector#resources-tab>

Using the Tools tab Calling a tool

This is where you interact with your MCP Server. That panel is self-explanatory, for example to call a tool:

- Click the “Tools” button in the header.
- Click “List tools”.
- Click the tool of your choice: a form appears on the right side.
- Fill in all the input fields.
- Click “Run Tool”.

6. The History Panel

Official doc: none

The messages exchanged between MCP Inspector and the MCP Server are displayed here. For example, here are the messages exchanged during a tool call:

Message Exchanged during a tool call.

The format of those messages are described in the [MCP Specification](#). If you pay close attention, you'll notice that MCP Inspector does **not** include a message id even though this is a [mandatory field](#).

By comparison, here is the message that Claude Desktop sends when calling the same tool (notice the message id).

```
{
  "method": "tools/call",
  "params": {
    "name": "say_hello",
    "arguments": {
      "name": "foo"
    }
  },
  "jsonrpc": "2.0",
  "id": 9
}
```

7. The Server Notifications Panel

Official Documentation: none

This is where MCP Inspector displays the notifications sent by the Server. Here is how to send a notification during a tool call when using FastMCP 2.0:

```
from fastmcp import FastMCP
from fastmcp.server.dependencies import get_context

server = FastMCP("Demo")

@server.tool()
async def say_hello(name:str) -> str:
    ctx = get_context()
    await ctx.info(f"context: say_hello(): name={name}")
    return f"hello {name}"
```

8. Further Readings

- [Official Documentation \(github\)](#)
- [MCP Inspector: Debugging Your MCP Servers Made Easy](#)

Python

Mcp Server

Mcp Inspector

Artificial Intelligence



Follow

Written by Laurent Kubaski

102 followers · 3 following

I'm a product manager at Salesforce, more info here: <https://kubaski.com/>

No responses yet



Bgerby

What are your thoughts?

More from Laurent Kubaski




Laurent Kubaski

MCP Resources explained (and how they differ from MCP Tools)

This article is part of my MCP Explained series and a follow up to my MCP Prompts Explained article.

Aug 1 🖱️ 69 💬 2




 Laurent Kubaski

Cursor AI Pricing Explained

July 2025 update: the Cursor pricing model changes a lot and as I was expecting, what I wrote back in this article back in May 2025 is not...



May 21 🖱️ 11 💬 2




 Laurent Kubaski

Ollama prompt templates

This article is part my “Ollama Explained” series.

Feb 9  17  1



 Laurent Kubaski

Connecting your Python MCP Server to Copilot Chat in Visual Studio Code


This article is part of my MCP Explained series.

Jun 20  5



[See all from Laurent Kubaski](#)


Recommended from Medium

 AI Rabbit

Test and Debug Any MCP Server with Inspector

 Apr 23  65




 In The AI Language by Kartik Marwah

Streamable HTTP MCP Server—Full Code Implementation

Build a Streamable HTTP MCP Server and test with Claude Desktop

★ Jul 5 🖱️ 5




 Laurent Kubaski

Tool (aka Function Calling) Best Practices

This article is part of my MCP Explained and part of my Ollama Explained article series.

Jul 2  21



 Sanath Shetty

Exposing MCP Servers as APIs: Building Bridges Between AI Models and Applications

In the rapidly evolving landscape of AI integration, developers face a common challenge: how to efficiently connect AI models with...

May 19 🖱️ 3




 Paul Fruitful

Building A Simple MCP Server For Operating Systems With Python

 Day 49 of #100daysOfAIEngineering

Apr 23



 Andre Botta

Running an MCP Server and Client locally with Ollama

If you're into AI, you've probably heard of MCP, the Model Context Protocol. It's an open standard that lets large language models connect...

Aug 11



See more recommendations