

Lab 020: Build an MCP Server in Python

Level: L200 Path: ? MCP Time: ~45 min ? Cost: Free ? Runs locally, no cloud account needed

What You'll Learn

How to build an MCP server from scratch using FastMCP (Python) How to define Tools with proper schemas and descriptions How to run the server and connect it to the MCP Inspector and GitHub Copilot (VS Code) How to add HTTP/SSE transport for cloud-agent compatibility

Introduction

An MCP server is a program that exposes tools (functions) to AI agents via the MCP protocol. When an agent needs to perform an action ? query a database, call an API, read a file ? it calls your MCP server's tools.

In this lab we build a product search MCP server with two tools:

1. `list_categories` ? returns product categories
2. `search_products` ? searches products by keyword

Prerequisites Setup

[diagram or code sample]

Make sure you have Python 3.10+: [diagram or code sample]

tip "Quick Start with GitHub Codespaces" Open in GitHub Codespaces

All dependencies are pre-installed in the devcontainer.

? Supporting Files

note "Download these files before starting the lab" Save all files to a `lab-020/` folder in your working directory.

File | Description | Download | `outdoorgear_mcp_server_starter.py` | Starter script with TODOs | ? Download |

Lab Exercise

Step 1: Create the project

[diagram or code sample]

Create `server.py`:

[diagram or code sample]

Step 2: Define your Tools

Add the tools after the `PRODUCTS` list:

[diagram or code sample]

Step 3: Run the server (stdio mode)

Add the entry point at the bottom of `server.py`:

[diagram or code sample]

Run it: [diagram or code sample]

The server is now listening on stdio. This is the default mode for local tools.

Step 4: Test with the MCP Inspector

Open a new terminal and run: [diagram or code sample]

The Inspector will open in your browser. Try:

1. Click "Tools" to see your three tools
2. Click `list_categories` ? "Run tool" ? see the categories
3. Click `search_products` ? fill in keyword: "tent" ? "Run tool" success "You should see the Camping Tent in the results"

Step 5: Run as HTTP/SSE server (for remote agents)

For cloud-hosted agents like Microsoft Foundry, we need HTTP/SSE transport. Add the startup option:

[diagram or code sample]

Run in HTTP mode: [diagram or code sample]

You'll see: [diagram or code sample]

Test with curl: [diagram or code sample]

Step 6: Connect to GitHub Copilot in VS Code

1. In VS Code, create .vscode/mcp.json in your workspace:

[diagram or code sample]

2. Open GitHub Copilot Chat in VS Code 3. Type: @copilot What product categories are available?

GitHub Copilot will call your list_categories tool and include the result in its response!

tip "VS Code MCP support" Make sure you have the GitHub Copilot extension version 1.99+ installed. You may need to enable MCP in VS Code settings: "chat.mcp.enabled": true

Adding a Resource (Bonus)

MCP also supports Resources ? data the agent can read. Add a resource that exposes the full product catalog:

[diagram or code sample]

? Starter File

This lab includes a starter file with TODO markers to guide you through building the server:

[diagram or code sample]

[diagram or code sample]

The starter contains the OutdoorGear product catalog (P001?P007) already populated. You implement: list_categories, search_products, get_product_details, and a challenge tool compare_products.

? Challenge: Add a compare_products Tool

Once you have the basic 3 tools working, add a fourth:

[diagram or code sample]

Test it in the MCP Inspector by asking: *"Compare the TrailBlazer Tent 2P and the TrailBlazer Solo. Which is lighter?"*

The agent should call compare_products(["P001", "P003"]) and return a structured comparison.

Summary

You've built a fully functional MCP server that:

? Defines 3 tools with proper descriptions (the LLM uses these to decide when to call) ?

Runs in stdio mode for local tools ? Runs in HTTP/SSE mode for remote agents ? Works with the MCP Inspector for testing ? Integrates with GitHub Copilot in VS Code

Next Steps

C# version: ? Lab 021 ? MCP Server in C# Connect to Microsoft Foundry Agent Service: ? Lab 030 ? Foundry Agent Service + MCP Add real database queries: ? Lab 031 ? pgvector Semantic Search

Phase 3 modernization

warning "Verified 2026-10-27" Target MCP specification 2026-07-28. Prefer stateless server design, capability discovery, current Tier 1 SDKs, and explicit multi-round-trip state. Add Registry and authorization only where the trust model requires them.

[diagram or code sample]

Model Context Protocol specification MCP authorization MCP Registry

Supporting kit

The site build generates stable PDF and ZIP downloads. The ZIP contains the lab guide and every supporting file declared for this lab.

Download generated PDF Download generated ZIP

Supporting file: outdoorgear_mcp_server_starter.py

"" outdoorgear_mcp_server_starter.py ? OutdoorGear MCP Server Starter

Lab 020: Build an MCP Server in Python

This starter file contains TODO markers where you need to write the code. Follow the instructions in each TODO to complete the lab.

Prerequisites: pip install fastmcp

Run with: python outdoorgear_mcp_server_starter.py ""

```
from fastmcp import FastMCP
```

```
===== Server initialization
```

```
=====
```

TODO 1: Initialize the FastMCP server with: name = "outdoorgear-catalog" description = "OutdoorGear Inc. product catalog MCP server" mcp = FastMCP(...) mcp = None # Replace this line with the FastMCP initialization

```
===== Data: OutdoorGear product catalog =====
```

```
PRODUCTS = [ { "id": "P001", "name": "TrailBlazer Tent 2P", "category": "Tents",
"description": "Lightweight 2-person backpacking tent, 3-season, aluminum poles",
"price_usd": 249.99, "weight_grams": 1800, "in_stock": True, "season": "3-season", }, {
"id": "P002", "name": "Summit Dome 4P", "category": "Tents", "description": "4-season
expedition tent for 4 people, steel poles, snow rated", "price_usd": 549.99,
"weight_grams": 3200, "in_stock": True, "season": "4-season", }, { "id": "P003", "name":
"TrailBlazer Solo", "category": "Tents", "description": "Ultra-light solo tent, 3-season,
850g, fast-and-light hiking", "price_usd": 299.99, "weight_grams": 850, "in_stock": True,
"season": "3-season", }, { "id": "P004", "name": "ArcticDown -20°C Sleeping Bag",
"category": "Sleeping Bags", "description": "Winter sleeping bag, 800-fill down, rated to
-20°C, mummy shape", "price_usd": 389.99, "weight_grams": 1400, "in_stock": True, "season":
"winter", }, { "id": "P005", "name": "SummerLight +5°C Sleeping Bag", "category": "Sleeping
Bags", "description": "Lightweight summer sleeping bag, synthetic insulation, +5°C",
"price_usd": 149.99, "weight_grams": 700, "in_stock": True, "season": "summer", }, { "id":
"P006", "name": "Osprey Atmos 65L Backpack", "category": "Backpacks", "description":
"65-liter backpacking pack, anti-gravity suspension, hipbelt pockets", "price_usd": 289.99,
"weight_grams": 1980, "in_stock": True, "season": "all", }, { "id": "P007", "name":
"DayHiker 22L Daypack", "category": "Backpacks", "description": "22-liter lightweight
daypack, laptop sleeve, hydration compatible", "price_usd": 89.99, "weight_grams": 580,
"in_stock": True, "season": "all", }, ]
```

```
===== TOOL 1: list_categories
```

```
=====
```

TODO 2: Define a tool called list_categories using the @mcp.tool() decorator. Requirements:

- Takes no parameters - Returns a sorted list of unique category names (strings) from PRODUCTS - Include a docstring: "List all available product categories." Hint: use set() + sorted() to get unique sorted categories @mcp.tool() def list_categories() -> list[str]: ...

```
===== TOOL 2: search_products
```

```
=====
```

TODO 3: Define a tool called search_products using the @mcp.tool() decorator. Parameters: -

keyword: str ? search term matched against product name and description - category: str |

None = None ? optional filter (use list_categories for valid values) - max_price: float |

None = None ? optional maximum price in USD - in_stock_only: bool = True ? if True, only

return in-stock items - max_results: int = 5 ? maximum number of results Returns:

list[dict] ? matching products (each dict has id, name, category, price_usd, in_stock)

Docstring should explain each parameter (the LLM reads this to decide when/how to use the

```
tool!) @mcp.tool() def search_products(keyword: str, ...) -> list[dict]: ...
```

```
===== TOOL 3: get_product_details
```

```
=====
```

TODO 4: Define a tool called `get_product_details` using the `@mcp.tool()` decorator.

Parameters: - `product_id: str` ? the product ID (e.g. "P001") Returns: - The full product dict if found (all fields including description, weight_grams, season) - {"error": "Product not found", "product_id": product_id} if not found

```
@mcp.tool() def get_product_details(product_id: str) -> dict: ...
```

```
===== TOOL 4: compare_products
```

```
(CHALLENGE ? more advanced) =====
```

TODO 5 (Challenge): Define a tool called `compare_products`. Parameters: - `product_ids:`

`list[str]` ? list of product IDs to compare (2-4 items) Returns: dict with structure: {

"products": [...], # list of found products "not_found": [...], # product_ids that weren't

found "lightest": "P003", # id of lightest product (by weight_grams) "cheapest": "P007", #

id of cheapest product (by price_usd) } @mcp.tool() def compare_products(product_ids:

```
list[str]) -> dict: ...
```

```
===== Entry point
```

```
=====
```

```
if __name__ == "__main__": # TODO 6: Start the server by calling mcp.run() # This runs the  
server in stdio mode (default for local development) pass # Replace with: mcp.run()
```