How to build an MCP server

In this tutorial, you'll build a simple Model Context Protocol (MCP) server that exposes a single tool for searching IBM tutorials. By using the `fastmcp` framework and the requests library, the script downloads a JSON index of tutorials from a remote URL. It then searches for matches to a user's query and returns a neatly formatted list of results. You'll also add error handling for network issues, bad JSON and unexpected problems, making the tool robust and beginner-friendly. Finally, you'll run the MCP server so it can be connected to and tested with a client like Cursor.

What is MCP?

Enterprise and startup developers alike are increasingly developing generative artificial intelligence (AI) driven solutions. In order to make their solutions more useful, they need up-to-date information and context. Machine learning models need to interoperate with tools, application programming interfaces(APIs), software development kits (SDKs) and front-end systems for that to happen.

The MCP standardizes how context is passed between AI models and systems. It simplifies coordination across a large language model (LLM) and external data sources and tools. A common analogy is to think of MCP as a USB-C port for an LLM. They make an LLM much more useful because the model has access to capabilities and data that weren't part of the model's training. This ability is especially useful when building AI agents.

MCP was developed by Anthropic and was adopted by major AI providers including OpenAI, Google DeepMind and the wider industry. It provides a secure and standardized way for AI models to access and use external data, resources (such as prompt templates) and tools.

Furthermore, integrated development environments (IDEs) such as Cursor and Visual Studio Code have also adopted MCP, allowing their AI assistants to access MCP servers to make their use more context relevant and developer friendly. Built as an open standard, organizations use MCP to act as a bridge between the stochastic world of generative AI and the deterministic world of most enterprise systems that exist today. MCP provides an LLM with contextual information similarly to other design patterns that have started to emerge such as retrieval augmented generation (RAG), tool calling and AI agents.

Some advantages to using MCP in comparison with these other solutions include:

- Scale: MCP servers can be defined and hosted once and used by many AI systems. This capacity limits the need to define access to the same source data, resources and AI tools for multiple generative AI systems.

- Data retrieval: Unlike RAG where data retrieval requires preprocessing and vectorizing before query, MCP is dynamic and allows for fluctuations and updates from information sources in real-time.

- Complexity: MCP is fairly simple to setup and incorporate into AI applications, as we demonstrate here. You can use config files easily to make an MCP server portable across environments.

- Platform-independent: Beyond the fact that you can build MCP servers with Python, TypeScript or other languages, they are also not coupled to a specific LLM solution.

- Debugging through a client/server model: The MCP client sends requests to the MCP server, which then fetches the necessary data from various external systems and sources—be it APIs, databases or local files. This structured approach ensures that the AI model receives consistent and relevant context, leading to more accurate and reliable outputs. MCP uses JSON-RPC to encode messages and supports 2 transport

mechanisms, stdio and streamable HTTP. In previous iterations of the protocol, it also supported HTTP with server-sent events (SSE)

The need to constantly keep up to date with the latest information your enterprise needs can be daunting. MCP can help build context and incorporate new information for contracts as they are executed, legacy information that is being digitized but not necessarily made digestible and more. That information can be both internal and external, but adding context bypasses the time-consuming need to retrain an LLM in order to be useful.

There are many remote MCP servers available as well as plenty of reference implementations from github.com

Steps

This step-by-step guide can be found on our GitHub repository along with the server.py script you'll reference when creating your MCP server. In this tutorial we'll walk through building a basic custom MCP server that can:

- Connect to our GitHub repository of tutorials
- Perform a search for topics that the user might be interested in
- Return the results with links to the tutorial

In order to facilitate making this tutorial, we have created a mechanism by which the server you will build can easily consume our tutorial content with no required authentication.

Step 1. Set up your environment

- Python 3.11 or newer installed on your computer (check by running python3 --version in your terminal).

- The built-in venv module is available (it comes with Python on most systems; on some Linux® distributions you might need to install it separately with sudo apt install python3-venv).

- A command line terminal (CLI):

- macOS or Linux: use your Terminal app (these environments are Unix-like).

- Windows: use PowerShell or Command Prompt, with small syntax differences explained in the next step.

- A text editor or IDE of your choice

Make a new directory and cd into it

mkdir ibmtutorialmcpserver and cd ibmtutorialmcpserver

Ensuring that you are in the directoryibmtutorialmcpserver you can run the following command to create a virtual environment

python3 -m venv venv

Note: On Windows, you might be able to replacepython3 withpython

Once you have created the virtual environment, you need to activate it by using the following command

source venv/bin/activate

Once activated, your shell will likely show you(venv) in the prompt

Now you need to install the Python package forfastMCP . This open source framework provides all the features required for running an MCP server and is actively maintained. We are also going to install therequests package for making simple HTTP requests

Install thefastMCP andrequests package withpip by using the following command

pip install fastmcp requests

After this step completes, you can check that fastMCP is installed correctly by running the following command

fastmcp version

If you get output similar to below, you have fastMCP installed in your virtual environment.

```
FastMCP version:                                     2.10.1
MCP version:                                         1.10.1
Python version:                                     3.11.13
Platform:                           macOS-15.5-arm64-arm-64bit
FastMCP root path: /opt/homebrew/lib/python3.11/site-packages
```

Now you have fastMCP installed, let's get our MCP server created.

Step 2. Create an MCP server

Create a new file in the directory and let's give it the nameserver.py .

Once you have that file, open it and copy and paste the following code snippet into it
Simple MCP server that exposes a single tool to search IBM tutorials. # How to run: # 1) Install dependencies: pip install fastmcp requests # 2) Start the server using an MCP client with the command: fastmcp run <YOUR PATH>/server.py from fastmcp import FastMCPimport requests # Source of the tutorials index DOCS_INDEX_URL = "https://raw.githubusercontent.com/IBM/ibmdotcom-tutorials/refs/heads/main/docs_index.json" mcp = FastMCP("IBM Tutorials") @mcp.tool def search_ibmtutorials(query: str) -> str: """ Search for tutorials on GitHub by downloading a JSON file from a GitHub repo and searching the payload for any relevant results and the respective details Args: query: The search term to look for in tutorial titles and URLs Returns: A formatted list of relevant tutorial results """ try: # Download the JSON file from the GitHub repo response = requests.get(DOCS_INDEX_URL, timeout=10) response.raise_for_status() # Raise an exception for bad status codes # Parse the JSON data tutorials = response.json() # Search for relevant tutorials (case-insensitive) query_lower = query.lower() relevant_tutorials = [] for tutorial in tutorials: # Search in title and URL title = tutorial.get('title', '').lower() url_path = tutorial.get('url', '').lower() if query_lower in title or query_lower in url_path: relevant_tutorials.append(tutorial) # Format and return results if not relevant_tutorials: return f"No IBM tutorials found matching '{query}'" # Format the results result_lines = [f"Found {len(relevant_tutorials)} tutorial(s) matching '{query}':\n"] for i, tutorial in enumerate(relevant_tutorials, 1): title = tutorial.get('title', 'No title') tutorial_url = tutorial.get('url', 'No URL') date = tutorial.get('date', 'No date') author = tutorial.get('author', '') result_lines.append(f"{i}. **{title}**") result_lines.append(f" URL: {tutorial_url}") result_lines.append(f" Date: {date}") if author: result_lines.append(f" Author: {author}") result_lines.append("") # Empty line for spacing return "\n".join(result_lines) except requests.exceptions.RequestException as e: return f"Error fetching tutorials from GitHub: {str(e)}" except ValueError as e: return f"Error parsing JSON data: {str(e)}" except Exception as e: return f"Error searching IBM tutorials: {str(e)}" if __name__ == "__main__": mcp.run()

Let's go through the preceding code and explain what the key parts are doing.

Imports and setup

The script starts by importing FastMCP, which provides the framework for creating an MCP server, and requests, that are used to download data over HTTP. We've added a constantDOCS_INDEX_URL to hold the remote JSON URL for the tutorials index. This approach makes it easier to change the source location if you had another source of JSON data you wanted to reuse this tutorial for later.

Setting up the MCP server

We then create an MCP server instance withFastMCP("IBM Tutorials") . This step acts as the central controller for all the tools the server will expose to MCP clients, such as thesearch_ibmtutorials tool we define.

Defining the MCP tool

The@mcp.tool decorator markssearch_ibmtutorials as an MCP tool. This function takes a search term, downloads the tutorial index from theDOCS_INDEX_URL by usingrequests.get() (with a 10-second timeout for network safety) and raises an exception if the HTTP response status indicates an error. Once the data is retrieved, it's parsed from JSON into Python objects.

The search is case-insensitive: the query is converted to lowercase, and each tutorial's title and URL are also lowered for matching. If a tutorial contains the search term in either the title or the URL, it's added to a list of relevant results.

Formatting and returning results

If no tutorials match the search, the function returns a friendly message indicating that nothing was found. If there are matches, the function builds a formatted, numbered list showing each tutorial's title, URL, date, and—if available—the author. The formatting uses Markdown-style bold for titles so they stand out in clients that support it. The final formatted text is returned as a single string.

Error handling

The function includes targeted exception handling:

requests.exceptions.RequestException catches network issues like timeouts or bad connections.

ValueError (which can be raised by .json()) catches cases where the response isn't valid JSON.

A generalException handler catches anything else unexpected.

Each error returns a descriptive message to the caller instead of stopping the program.

Starting the server

At the bottom, theif name == "main" : block ensures thatmcp.run() executes only when the script is run directly. This step starts the MCP server, making thesearch_ibmtutorials tool available to any MCP client, such as the MCP Inspector. This tool is useful for troubleshooting and debugging your MCP server. The tool provides a UI that you can use to debug the MCP server and validation of expected behavior.

Step 3. Add your MCP server to your IDE

Now that you have built your server, you need to enable it in your IDE before you can use it. There are many clients that support MCP with various levels of integration with the protocol. The official MCP website provides an exhaustive list of example clients.
If you have Cursor installed, you can add the MCP server within Cursor by following these instructions.

Open Cursor settings and navigate to Tools & Integrations. Select New MCP Server and paste into the mcp.json file that Cursor opens in a new tab. Make sure that you replace <YOUR PATH> with the directory you're in. You can run pwd in your terminal to get the full path. For further information about Cursor and MCP, see the **Cursor docs**.

```
{
  "mcpServers": {
    "tutorials": {
      "command": "fastmcp",
      "args": ["run <YOUR PATH>/ibmtutorialmcpserver/server.py"],
      "env": {
      }
    }
  }
}
```

If you are a user of Microsoft VS Code, you can add the MCP server by using the instructions linked here. Make sure that Copilot is set up in VS Code before proceeding.
If you'd like to enable the MCP server by using a file, create a .vscode/mcp.json file in this project's directory and copy and paste this code into the file. Make sure that you replace <YOUR PATH> with the directory you're in. You can run pwd in your terminal to get the full path.

```
"servers": {
                "IBM Tutorials": {
                        "type": "stdio",
                        "command": "fastmcp",
                        "args": [
                                "run",
                                "<YOUR PATH>/ibmtutorialmcpserver/server.py"
                        ]
                },
        },
```

Step 4. Use your MCP server

Now that you've enabled your MCP server, let's get the server running so you can use the tool created in server.py. If you're using VS Code, check out these docs.

In the IDE chat, I'll ask "what are some IBM time series tutorials?" The following shows that the output received, but your response can vary depending on the model used and your IDE.

Output:

```
Here are some IBM time series tutorials:

Time series forecasting with Lag-Llama (zero-shot learning)
Tutorial link
Predict overnight low temperatures using the Lag-Llama model in a zero-shot learning scenario.

Using the watsonx.ai Time Series Forecasting API to predict energy demand
Tutorial link
Predict energy demand with the watsonx.ai Time Series Forecasting API.
Authors: Aleksandra Kłeczek and Meredith Syed

Let me know if you want details or help with a specific tutorial.
```

Great! The agent was able to use the search_ibmtutorials tool we made to search for tutorials related to time series.

Summary

In this tutorial, you learned to build an MCP server to search through all our tutorials by using any MCP client you prefer. You created an MCP server with a single search tool that retrieves a remote JSON index of tutorials, filtered results based on a search term and returned them in a readable format. It uses fastmcp to register and run the tool, requests to fetch the data and includes error handling for network, parsing and unexpected issues. When run, the server can be connected to by MCP clients for live querying of all our tutorials.