ChatGPT

Building a Model-Context-Protocol Server with FastMCP, Python and Docker (Nov 22 2025)

Audience: experienced programmers preparing a 6-hour hackathon.
Scope: build a practical Model Context Protocol (MCP) server using Python and FastMCP, then package and publish it in Docker Hub.
Highlights: hands-on setup, project structure, secret handling, GraphQL integration, and theory comparing MCP to REST APIs.

Table of Contents

Background: Model Context Protocol & FastMCP

What is the Model Context Protocol (MCP)?

The Model Context Protocol is a standard for enabling **LLM agents** (e.g. Claude Desktop, Cursor, OpenAI's MCP client) to discover and call **tools** exposed by external services. MCP introduces several key elements:

Concept	Description
Tool	A function that an agent can call with structured input and get a structured output. Tools are described with a JSON schema. Apidog's FastMCP guide explains that a tool encapsulates a remote function, exposing input arguments and a description [1].
Resource	Static information that the agent can retrieve, such as documentation or configuration [1].
Prompt	Textual instructions that can help an agent use a tool properly [1].
Context	A summary of the conversation or environment that helps the agent reason; context is intrinsic in MCP servers while REST APIs leave context management to the client [2].

Unlike REST APIs, MCP servers maintain a **stateful connection** with agents (often over JSON-RPC) and expose a **runtime discoverable set of tools** [2]. Tools have structured input/output and are idempotent so that agents can plan, retry and chain calls. The protocol emphasises **predictability** and **composability**.

Why FastMCP?

FastMCP is a Python framework built on Starlette that simplifies building MCP servers:

- It allows you to define tools, resources and prompts via decorators, and automatically generates a tool manifest that clients can discover.
- FastMCP uses Pydantic for type validation; tool functions can declare typed parameters that get converted to JSON schema in the manifest.
- It can run via a CLI (`fastmcp run`) or directly with `uvicorn` [3].
- Configuration precedence: values passed as arguments override environment variables (prefixed `FASTMCP_SERVER_`), which override a `.env` file and defaults [4]. This is important for secrets.

FastMCP reduces boilerplate and encourages proper tool documentation, making it ideal for hackathons.

MCP vs REST APIs – Mental Models & Design Principles

Fundamental Differences

Axis	MCP	REST APIs
State management	**Stateful** – maintains context across interactions [2].	**Stateless** – each request stands alone.
Connection type	Persistent, bidirectional connections (often JSON-RPC) [2].	One-way request/response over HTTP [2].
Discovery	Tools are discovered at runtime by the agent; tool definitions include names, descriptions and JSON schemas [5].	Endpoints are discovered at design time via documentation or OpenAPI.

Axis	MCP	REST APIs
Context handling	The server can access conversation context; agents send and receive conversation context with calls [6] .	Context must be managed client-side.
Orientation	Action oriented – tools represent operations the agent can perform.	Resource oriented – verbs (GET/POST/etc.) operate on resources.
Audience	Tools are designed for LLM agents; output must be machine-parseable and semantically clear.	REST is designed for human developers; responses often include meta-fields for humans.

Design Principles for MCP Tools

1. **Small, composable tools.** The Docker MCP guidelines warn not to map every API endpoint to an individual tool; agents have a limited budget of tool calls and rely on combining smaller, general-purpose operations [7] . Provide a handful of clear tools that accomplish meaningful tasks.

2. **Rich outputs and no hidden work.** Matt Adams's MCP design notes emphasise that MCP servers are **information providers**. A tool should return rich, structured data instead of text summaries. Avoid withholding data because the agent can perform analysis; include all relevant fields so the agent can reason [8] .

3. **Agent-oriented errors.** Errors should instruct the agent on how to proceed (e.g., "invalid date format; please use YYYY-MM-DD") rather than generic messages [9] .

4. **Composition over monoliths.** Tools should be idempotent and safe to call repeatedly. Compose operations rather than hiding entire workflows inside one tool [7] .

5. **Thin wrappers & completeness.** MCP servers are wrappers around underlying data sources; they shouldn't do heavy analysis or filtering. Provide complete data sets; the agent can filter or summarise [10] .

6. **Connection & session management.** For resources requiring connections (databases, API sessions), open a connection, perform the operation and close it within the tool call rather than keeping a persistent connection [11] .

7. **Document for both humans & agents.** Each tool should have a human-readable description and examples. The overall server should include a README or resource describing how to run it. The Docker blog recommends using macros and prompts for repeat patterns [7] .

Project Structure & Setup

Our project aims to provide a **minimal but realistic layout** for an MCP server. Below is an example tree:

```
mcp_demo/
├── app/
│   ├── __init__.py
│   └── services.py          # domain logic / GraphQL wrappers
├── mcp_server.py            # FastMCP server definition and tools
├── requirements.txt         # Python dependencies (or pyproject.toml)
├── Dockerfile               # Container build instructions
├── .env.example             # Template for env vars (no secrets committed)
└── README.md                # Basic instructions
```

1. Install Dependencies

During your hackathon, you can quickly install FastMCP using pip or the `uv` installer:

```
pip install fastmcp fastapi pydantic httpx python-dotenv
```

Alternatively, `uv` (recommended by FastMCP docs) can provide faster dependency resolution:

```
curl -Ls https://github.com/astral-sh/uv/releases/latest/download/uv-
installer.sh | bash
uv venv .venv
uv pip install fastmcp httpx python-dotenv
```

We also install `httpx` for asynchronous HTTP requests (useful for GraphQL calls) and `python-dotenv` to load `.env` files for local development. Adjust `requirements.txt` accordingly:

```
# requirements.txt
fastmcp==0.8.4    # Check for the latest version on release
httpx==0.26.0
python-dotenv==1.0.1
```

Note: always pin versions for reproducibility in hackathons.

Implementing the MCP Server with FastMCP

This section walks through writing `mcp_server.py`, defining tools, loading secrets and connecting to a GraphQL backend.

Tools & Resources in FastMCP

FastMCP servers are created by instantiating a `Server` object and decorating functions with `@server.tool`, `@server.resource` or `@server.prompt`. A tool's docstring becomes its description, and function type annotations define the input and output schema. FastMCP's tutorial shows a simple example where a tool receives two integers and returns their sum [12]. We expand on this pattern.

Example Endpoints/Tools

Our hackathon server will expose two tools:

1. `get_weather` – fetch the current weather for a city using a GraphQL endpoint (or any external API). It demonstrates reading an API key from the environment, constructing a GraphQL query and returning structured results.
2. `convert_currency` – convert an amount from one currency to another using a simple REST or GraphQL call. It shows input validation and error handling.

Create `app/services.py` to isolate the logic (especially GraphQL communication):

```
# app/services.py
import os
import httpx
from typing import Dict

API_KEY = os.getenv("WEATHER_API_KEY")
GRAPHQL_ENDPOINT = os.getenv("GRAPHQL_API_URL")
CURRENCY_API_URL = os.getenv("CURRENCY_API_URL")

async def fetch_weather(city: str) -> Dict:
    # Build a GraphQL query; adjust to your actual schema
    query = """
    query ($city: String!) {
      getWeather(city: $city) {
        temperature
        description
        humidity
      }
    }
    """
    variables = {"city": city}
    headers = {"Authorization": f"Bearer {API_KEY}"} if API_KEY else {}
    async with httpx.AsyncClient() as client:
        response = await client.post(
            GRAPHQL_ENDPOINT, json={"query": query, "variables": variables},
headers=headers
        )
        response.raise_for_status()
        data = response.json()
        return data["data"]["getWeather"]

async def convert_currency(amount: float, from_currency: str, to_currency: str)
-> Dict:
    params = {"amount": amount, "from": from_currency, "to": to_currency}
    async with httpx.AsyncClient() as client:
        response = await client.get(CURRENCY_API_URL, params=params)
        response.raise_for_status()
        return response.json()
```

⚠ **Secret handling:** never hard-code API keys. Load them via environment variables like `WEATHER_API_KEY` and `GRAPHQL_API_URL`. This aligns with best practice to eliminate hard-coding secrets [13].

Next create `mcp_server.py` to register tools and resources:

```
# mcp_server.py
from fastmcp import Server
from pydantic import BaseModel, Field
from typing import Optional
from app.services import fetch_weather, convert_currency
import os
from dotenv import load_dotenv

# Load .env during local development (not in production containers)
load_dotenv()

server = Server(name="Demo MCP Server", description="Hackathon MCP server with
weather and currency tools")

class WeatherInput(BaseModel):
    """Input for the get_weather tool."""
    city: str = Field(..., description="Name of the city to look up (e.g.
'Tampa').")

class WeatherOutput(BaseModel):
    temperature: float = Field(..., description="Temperature in Celsius")
    description: str = Field(..., description="Weather conditions description")
    humidity: Optional[int] = Field(None, description="Humidity percentage")

class CurrencyInput(BaseModel):
    amount: float = Field(..., ge=0, description="Amount to convert")
    from_currency: str = Field(..., min_length=3, max_length=3,
description="3-letter ISO currency code to convert from (e.g. 'USD')")
    to_currency: str = Field(..., min_length=3, max_length=3,
description="3-letter ISO currency code to convert to (e.g. 'EUR')")

class CurrencyOutput(BaseModel):
    converted_amount: float = Field(..., description="Converted amount in the
target currency")
    rate: float = Field(..., description="Exchange rate used")

@server.tool(name="get_weather", description="Retrieve current weather for a
city.")
async def get_weather(input: WeatherInput) -> WeatherOutput:
    """
    Get the current weather for the provided city.

    This tool calls an external GraphQL API to fetch weather data.
    Returns temperature, description and optional humidity.
    """
    data = await fetch_weather(input.city)
    return WeatherOutput(**data)

@server.tool(name="convert_currency", description="Convert an amount from one
currency to another.")
async def convert_currency_tool(input: CurrencyInput) -> CurrencyOutput:
    """Convert currency using an external API."""
    result = await convert_currency(input.amount, input.from_currency,
input.to_currency)
    return CurrencyOutput(
        converted_amount=float(result["result"]),
        rate=float(result["rate"]),
    )

@server.resource(name="README", description="Human-readable instructions for
this server")
async def readme() -> str:
    return (
        "This server exposes two tools: get_weather and convert_currency.\n"
        "Use get_weather(city) to fetch current weather.\n"
        "Use convert_currency(amount, from_currency, to_currency) to convert
currencies.\n"
    )

if __name__ == "__main__":
    # For bare-metal running via `python mcp_server.py`
    import uvicorn
    uvicorn.run(server.app, host="0.0.0.0", port=int(os.environ.get("PORT",
8000)))
```

Key points:

- **Type annotations & Pydantic models** define the input/output schema that the MCP manifest uses; this helps the LLM agent reason.
- Tools return data classes (Pydantic models) rather than raw dicts to provide explicit JSON schema.
- Environment variables are loaded via `dotenv` only for local development. In production containers, they come from the runtime environment.

Running Bare-Metal with Uvicorn

During development you can run the server directly:

```
export WEATHER_API_KEY=<your-weather-api-key>
export GRAPHQL_API_URL=https://api.example.com/graphql
export CURRENCY_API_URL=https://api.currency.com/convert
python mcp_server.py  # runs on port 8000
```

Alternatively, use FastMCP's CLI to run the server automatically:

```
fastmcp run mcp_server:server
```

FastMCP's CLI reads environment variables with prefix `FASTMCP_SERVER_` and `.env` files; variables passed via CLI arguments override env variables [4]. However, for hackathons the plain `uvicorn` call is straightforward.

To test locally, open a new terminal and use an MCP client (e.g. VS Code, Claude Desktop, or a simple curl command) pointed at `http://localhost:8000`; the manifest will be available at `/manifest`.

Secrets and API-Key Handling

Build-time vs Runtime Secrets

- **Build-time secrets** are used during image building or CI. They include credentials for package registries or internal services. **Never bake user API keys into the image**; secrets at build time are general to the team, not per user.
- **Runtime secrets** are provided when you run the server. They can be passed via environment variables, CLI arguments or a client configuration file. For MCP, each user might have a different API token (e.g. a weather API key), so secrets must be runtime-configurable.

According to WorkOS's best practices for secret management, avoid hard-coding secrets and load them from environment variables or secret managers [13]. Prefer short-lived credentials and least-privilege access [14].

How MCP Clients Provide Secrets

MCP clients (LLM agents) allow users to configure environment variables when connecting to a server:

- **VS Code** uses a `servers` configuration in `.vscode/mcp_servers.json`. Each server entry can specify a `type` (`http` or `stdio`), a `command`/`args` for local processes, and an `env` object where you can inject `${input:my-key}` placeholders. VS Code will prompt the user to fill secret values [15].

- **Claude Desktop** and **Cursor** have similar MCP configuration sections. The Jentic documentation shows a JSON config where environment variables like `JENTIC_AGENT_API_KEY` and `OPENAI_API_KEY` are provided under `env` [16].

- **Docker Hub MCP integration** (alpha as of late 2025) allows you to connect an image to the MCP catalog. Users configure environment variables at runtime via the Docker Hub UI or their LLM client; the image should document which variables are required.

Reading Secrets in Python

Use `os.getenv` to read secrets. Do not print them or include them in error messages.

```python
import os
api_key = os.getenv("WEATHER_API_KEY")
if not api_key:
    raise RuntimeError("WEATHER_API_KEY must be set in the environment")
```

When running locally, you can set variables on the command line or store them in a `.env` file (excluded from version control) and load with `python-dotenv`. In Docker, secrets are passed via `-e` flags or via the client's configuration.

Note: Each user will run their own container or process with their own API key; the code does not need to change between users. Only the runtime environment differs.

Dockerizing the MCP

Dockerfile & Building Locally

Here is a simple `Dockerfile` for our MCP server:

```dockerfile
# Dockerfile
FROM python:3.11-slim

# Install runtime dependencies
```

```
RUN pip install --no-cache-dir fastmcp==0.8.4 httpx==0.26.0 python-dotenv==1.0.1

# Copy application files
WORKDIR /app
COPY app app
COPY mcp_server.py mcp_server.py

# Expose port and run the server
ENV PORT=8000
CMD ["python", "mcp_server.py"]
```

This container uses the official Python slim image, installs dependencies and runs `mcp_server.py`. We avoid storing secrets in the image.

To build and run locally:

```
# Build image tagged locally
docker build -t myusername/mcp-demo:latest .

# Run with environment variables (replace with your keys)
docker run -p 8000:8000 \
  -e WEATHER_API_KEY=<your-weather-api-key> \
  -e GRAPHQL_API_URL=https://api.example.com/graphql \
  -e CURRENCY_API_URL=https://api.currency.com/convert \
  myusername/mcp-demo:latest
```

The `-p` flag maps the container port 8000 to a host port. Secrets (`WEATHER_API_KEY`, etc.) are passed as environment variables.

Running the Container Locally

After running the container, your MCP manifest is available at `http://localhost:8000/manifest`. You can configure your MCP client to connect to `http://localhost:8000`. Ensure the client can reach your machine; if running on a remote host you may need to expose the port.

Publishing to Docker Hub & MCP Integration

1. **Tag and push the image:**

```
docker login
docker tag myusername/mcp-demo:latest docker.io/myusername/mcp-demo:0.1.0
docker push docker.io/myusername/mcp-demo:0.1.0
```

1. **Document required environment variables** in your `README.md` and Docker Hub description. Include variables like `WEATHER_API_KEY`, `GRAPHQL_API_URL` and `CURRENCY_API_URL`. Users

will supply these via the MCP client configuration; the Docker Hub UI may also allow specifying default values.

2. **Register in MCP client**: LLM clients (Claude Desktop, Cursor, OpenAI's MCP) allow connecting to a Docker Hub MCP. Typically you provide the image name and version; the client spins up the container and prompts the user for required env vars. The code inside the container remains unchanged.

3. **Stdio vs HTTP:** Some clients (Docker MCP Catalog) require your server to speak via **stdio** instead of HTTP. In that case, your `Dockerfile` would run a different entrypoint (e.g. `fastmcp run --transport stdio mcp_server:server`). The Docker best-practices article states that servers submitted to the catalog must use stdio transport [17]. Adjust accordingly if you plan to submit to the catalog.

GraphQL Integration Inside the MCP

Our example calls a GraphQL API in `fetch_weather`. Designing the GraphQL integration inside an MCP server requires careful consideration.

Architecture

- **MCP server** acts as an outer layer. It exposes high-level tools to the agent and hides the underlying GraphQL complexity.
- **GraphQL API** is an internal service. The MCP server constructs queries/mutations and sends them via HTTP.
- **Tools** should map to specific GraphQL operations rather than exposing raw GraphQL queries to the agent.

Design Options

1. **Generic GraphQL tool**: Accepts a GraphQL query string and variables. This gives the agent full flexibility but imposes a heavy burden: the agent must know the GraphQL schema and craft queries. It also raises security concerns (arbitrary queries) and increases the risk of prompt injection. Use this only if the GraphQL API is simple and safe.

2. **Specific tools per GraphQL operation** (recommended). Each tool corresponds to a GraphQL query/mutation. The tool's input schema maps to GraphQL variables, and the tool returns a structured response. This pattern is hackathon-friendly and encourages type safety.

3. **REST wrapper**: Build a REST API around the GraphQL API and call it from the MCP. This adds an extra layer and is generally unnecessary for hackathons.

Implementation Pattern

The Mirumee FastMCP + Ariadne blog demonstrates how to use Pydantic models and `@server.tool` to wrap GraphQL queries [18]. The blog emphasises that GraphQL's schema-first nature suits tool definitions and codegen [19]. Our `fetch_weather` function above follows the same pattern: build a GraphQL query string, set variables, call `httpx.AsyncClient.post` with the query and parse the response.

When designing your own tools:

- Use **async functions** and `httpx.AsyncClient` for concurrency and performance.
- Define separate functions in `app/services.py` to encapsulate GraphQL calls; keep your tool functions thin.
- Read the GraphQL endpoint and API key from environment variables; this makes the same container usable by different users.
- Return only the fields needed by the agent, with clear descriptions. Do not rely on the agent to filter nested GraphQL responses.

Example GraphQL integration in `fetch_weather`: See `app/services.py` above. The GraphQL query and variables are defined inline. You could also store queries in separate `.graphql` files.

Quick Start Checklist

Follow these steps to get a working MCP server quickly:

1. **Clone the template** or create the file structure described above (`app/`, `mcp_server.py`, `requirements.txt`, `Dockerfile`, `.env.example`).
2. **Install dependencies:** `pip install -r requirements.txt` (or use `uv`).
3. **Fill** `.env` with your API keys:

```
WEATHER_API_KEY=your-weather-api-key
GRAPHQL_API_URL=https://api.example.com/graphql
CURRENCY_API_URL=https://api.currency.com/convert
```

4. **Run bare-metal:**

```
source .env
python mcp_server.py
```

   Or run `fastmcp run mcp_server:server` if you prefer CLI.
5. **Test locally:** Use a tool like `curl` or an MCP client to fetch `http://localhost:8000/manifest` and call the tools.
6. **Build the Docker image:** `bash
   docker build -t myusername/mcp-demo:latest .`

7. **Run the container:** `bash`
```
docker run -p 8000:8000 -e WEATHER_API_KEY=... -e GRAPHQL_API_URL=... -e
CURRENCY_API_URL=... myusername/mcp-demo:latest
```
8. **Push to Docker Hub:** Tag and push your image. Update `README.md` and the Docker Hub description with required env vars.
9. **Configure MCP client:** In your agent tool (Claude Desktop, Cursor, VS Code), add a new MCP server pointing to your container or image. Provide the required environment variables via the client's UI/ config.

Design Checklist

Use this checklist to ensure your MCP server is agent-friendly and maintainable:

- **[ ] Clear, small tools:** Each tool performs one action and has a clear name & description.
- **[ ] Rich, structured outputs:** Tools return all relevant data for the agent; avoid summarising or hiding details [8] .
- **[ ] Typed input/output:** Use Pydantic models and explicit types to generate precise schemas.
- **[ ] Proper error messages:** Errors guide the agent on how to correct the input [9] .
- **[ ] Idempotent and retryable:** Tools should be safe to call repeatedly without side effects.
- **[ ] No persistent connections:** Open and close connections within each tool call [11] .
- **[ ] Secrets loaded from environment:** Do not hard-code API keys; read from environment variables [13] .
- **[ ] Documented configuration:** Provide a resource or README that lists required environment variables and usage instructions.
- **[ ] GraphQL calls isolated:** Keep GraphQL queries in service functions; map tool inputs to variables.
- **[ ] Ready for Docker Hub:** Provide a minimal `Dockerfile`, avoid installing dev dependencies, and expose the correct transport (HTTP or stdio) [17] .

By following this guide you can spin up a functional Model Context Protocol server with FastMCP, wrap a GraphQL API, handle per-user secrets cleanly, and package everything for local use or Docker Hub. Keep the tools small and composable, provide rich output, and let the agent do the reasoning.

---

[1] [3] [4] [12] A Beginner's Guide to Use FastMCP
https://apidog.com/blog/fastmcp/

[2] [5] [6] MCP vs REST APIs: A Fundamental Distinction | Roo Code Documentation
https://docs.roocode.com/features/mcp/mcp-vs-api

[7] [9] [11] [17] Top 5 MCP Server Best Practices | Docker
https://www.docker.com/blog/mcp-server-best-practices/

[8] [10] MCP Server Design Principles: Building Effective Information Providers for LLMs | Matt Adams
https://www.matt-adams.co.uk/2025/08/30/mcp-design-principles.html

[13] [14] Best practices for MCP secrets management — WorkOS Guides
https://workos.com/guide/best-practices-for-mcp-secrets-management

[15] Use MCP servers in VS Code
https://code.visualstudio.com/docs/copilot/customization/mcp-servers

[16] Claude Desktop - Jentic Documentation
https://docs.jentic.com/guides/mcp/claude-desktop/

[18] [19] From GraphQL Schema to AI-Ready MCP Server in Minutes
https://mirumee.com/blog/from-graphql-schema-to-mcp-server