Building an MCP Server

Give Claude direct access to your servers, databases, and APIs

Building an MCP Server So Claude Can Access Your Systems

What This Is

MCP (Model Context Protocol) is how you give Claude direct access to your tools, APIs, and servers. Instead of copying data back and forth, Claude calls your systems directly - reading files, running commands, querying databases, managing your website, whatever you expose.

This guide gets you a working MCP server. You don't need to be a developer. You need a server (or a computer that stays on) and the willingness to follow instructions.

The Fast Way: Tell Claude to Build It

You don't need to write code. Claude can build the entire server for you. Copy this prompt into a Claude conversation, fill in the blanks, and Claude will generate everything you need - the server code, the configuration files, the deployment instructions, all of it.

The Prompt

> I need you to build me an MCP server in Python using the FastMCP SDK (`pip install "mcp[cli]"`).
> **What it should access:** [describe what you want - read and write files on my server, manage my WordPress site, query my database, run shell commands, access my practice management API, etc.]
> **Where it will run:** [your server details - e.g., Ubuntu 22.04 VPS, a Mac Mini in my office, a Windows desktop that stays on, etc.]
> **Server's IP or domain:** [e.g., 123.45.67.89, or mcp.mysite.com, or "local only for now"]
> **Reverse proxy:** [Nginx, Caddy, Traefik, or "I don't have one - set one up for me"]
> **What Claude should be able to do:** - [Read files in specific directories] - [Run shell commands] - [Query my PostgreSQL/MySQL/SQLite database] - [Manage WordPress posts and pages] - [Access my practice management system API] - [Whatever else you need]

> **What Claude should NOT be able to do:** - [Delete files, drop tables, modify system config - whatever you want restricted]
> Please generate: 1. The complete `server.py` file with all tools 2. A `requirements.txt` with dependencies 3. A systemd service file to run it automatically 4. The reverse proxy configuration 5. Step-by-step deployment instructions I can follow in order

Claude will generate all of it. You copy the files to your server, follow the steps, and connect.

After Claude Generates It

If you included shell access or file access in your tools list, Claude just built itself the ability to deploy its own server. Tell it:

> Now deploy this to my server. SSH details are [user@ip]. Walk me through anything you can't do directly.

Claude will create the directory, write the files, install dependencies, configure the reverse proxy, and start the service. You may need to provide your sudo password or run a couple of commands it can't execute remotely - it'll tell you which ones.

If you don't have an existing connection to your server from Claude (this is your first MCP server, so you probably don't), you'll need to do the initial deployment yourself. Claude will give you step-by-step instructions - copy-paste each command in order. If something fails, paste the error back into Claude. It'll fix it.

Once the MCP server is running and connected, Claude can update its own server code, add tools, and redeploy - through the very tools it just built. The first deployment is the only manual one.

What You Need Before You Start

- **A server or computer that's always on.** A $5/month VPS from any provider works. A desktop or laptop that stays on works too, but it needs to be reachable - either on your local network or exposed to the internet with a domain.
- **Python 3.10 or newer** installed on that server. Most Linux servers have this already. Check with `python3 --version`.
- **A domain or subdomain** pointing to your server (for remote access). If you're running locally and connecting from the same network, you can skip this.
- **Basic comfort with the command line.** You need to be able to SSH into a server and copy-paste commands. That's it.

How It Works

An MCP server is a small program that runs on your machine and exposes "tools" - specific functions Claude can call. When you connect Claude to your server, Claude sees those tools and can use them like any of its built-in capabilities.

```
Your Phone / Computer
    ↓ (HTTPS)
Claude.ai
    ↓ (MCP Protocol over HTTPS)
Your Reverse Proxy (handles SSL)
    ↓ (localhost)
Your MCP Server (Python program)
    ↓ (direct access)
Your Stuff (files, databases, APIs, WordPress, etc.)
```

Claude never touches your systems directly. It talks to your MCP server, which acts as a controlled gateway. You decide what tools to expose and what to restrict. If you don't want Claude deleting files, don't expose a delete tool. It's that simple.

Understanding What Claude Built for You

If you want to understand the code Claude generated - or modify it yourself - here's what's inside.

The Basic Structure

Every MCP server follows the same pattern:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("my_server")

@mcp.tool(name="tool_name")
async def my_tool(some_input: str) -> str:
    """Description Claude reads to understand what this tool does."""
    # Do something
    return "result"

if __name__ == "__main__":
    mcp.run(transport="streamable-http", host="0.0.0.0", port=8200)
```

That's it. Each `@mcp.tool` block is one tool Claude can call. The docstring (the text in triple quotes) is what Claude reads to decide when to use it. The function does the work and returns a result.

A Complete Minimal Server

This gives Claude four basic capabilities - enough to operate on any server:

```
import os
import subprocess
import json
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("my_server")

@mcp.tool(name="list_files")
async def list_files(path: str = ".") -> str:
    """List files and directories at the given path."""
    try:
        entries = os.listdir(path)
        return "\n".join(entries)
    except Exception as e:
        return f"Error: {e}"

@mcp.tool(name="read_file")
async def read_file(path: str) -> str:
    """Read the contents of a file."""
    try:
        with open(path, "r") as f:
            return f.read()
    except Exception as e:
        return f"Error: {e}"

@mcp.tool(name="write_file")
async def write_file(path: str, content: str) -> str:
    """Write content to a file. Creates parent directories if needed."""
    try:
        os.makedirs(os.path.dirname(path), exist_ok=True)
        with open(path, "w") as f:
            f.write(content)
        return f"Wrote {len(content)} bytes to {path}"
    except Exception as e:
        return f"Error: {e}"

@mcp.tool(name="run_command")
async def run_command(command: str) -> str:
    """Execute a shell command and return the output."""
    try:
        result = subprocess.run(
            command, shell=True, capture_output=True,
            text=True, timeout=30
        )
        output = result.stdout
        if result.stderr:
            output += f"\nSTDERR: {result.stderr}"
        return output + f"\nExit code: {result.returncode}"
```

```
    except subprocess.TimeoutExpired:
        return "Error: Command timed out after 30 seconds"
    except Exception as e:
        return f"Error: {e}"

if __name__ == "__main__":
    mcp.run(transport="streamable-http", host="127.0.0.1", port=8200)
```

Four tools. List files, read files, write files, run commands. With just these, Claude can navigate your file system, read configuration files, edit documents, check system status, restart services, and more. Everything else is building on this pattern.

Adding Specialized Tools

The same pattern scales to anything you can access from Python.

Database queries:

```
import sqlite3

@mcp.tool(name="query_db")
async def query_db(sql: str) -> str:
    """Run a read-only SQL query against the database."""
    conn = sqlite3.connect("/path/to/database.db")
    try:
        cursor = conn.execute(sql)
        rows = cursor.fetchall()
        columns = [desc[0] for desc in cursor.description]
        return json.dumps(
            [dict(zip(columns, row)) for row in rows],
            indent=2
        )
    finally:
        conn.close()
```

API wrappers:

```
import httpx

@mcp.tool(name="get_clients")
async def get_clients(search: str = "") -> str:
    """Search for clients in the practice management system."""
    async with httpx.AsyncClient() as client:
        resp = await client.get(
            "https://your-api.com/clients",
            params={"search": search},
            headers={"Authorization": f"Bearer {os.environ['API_TOKEN']}"}
        )
        return resp.text
```

WordPress management:

```
@mcp.tool(name="wp_list_posts")
async def wp_list_posts(per_page: int = 10) -> str:
    """List recent WordPress posts."""
    async with httpx.AsyncClient() as client:
        resp = await client.get(
            "https://yoursite.com/wp-json/wp/v2/posts",
            params={"per_page": per_page},
            headers={"Authorization": f"Basic {your_auth_token}"}
        )
        return resp.text
```

Each tool is a function. If you can do it in Python, you can expose it to Claude.

Security

This matters. Your MCP server gives Claude access to your systems. Think about what you're exposing.

The Basics

- **Bind to localhost, not the internet.** Your server should listen on `127.0.0.1`, not `0.0.0.0`. A reverse proxy handles the internet-facing side with SSL.
- **Use SSL.** Always. Caddy does this automatically. Nginx with certbot works too.
- **Don't run as root.** Create a regular user for the MCP server.
- **Don't hardcode credentials.** Use environment variables and a `.env` file.
- **Restrict what you expose.** If you don't want Claude deleting databases, don't build a delete tool.

Reverse Proxy Setup

Your MCP server runs on localhost. A reverse proxy sits in front of it, handles SSL, and makes it reachable from the internet.

Caddy (recommended - automatic SSL, zero config):

```
mcp.yourdomain.com {
    reverse_proxy 127.0.0.1:8200
}
```

Nginx (with Let's Encrypt):

```
server {
    server_name mcp.yourdomain.com;

    location / {
        proxy_pass http://127.0.0.1:8200;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_read_timeout 300s;
    }

    listen 443 ssl;
    # SSL certificates managed by certbot
}
```

Running as a Service

You want the server to start automatically and restart if it crashes:

```
# /etc/systemd/system/my-mcp.service
[Unit]
Description=My MCP Server
After=network.target

[Service]
Type=simple
User=yourusername
WorkingDirectory=/path/to/server
EnvironmentFile=/path/to/server/.env
ExecStart=/usr/bin/python3 /path/to/server/server.py
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Enable and start:

```
sudo systemctl enable my-mcp
sudo systemctl start my-mcp
```

Connecting Claude to Your Server

Claude.ai (Web/Mobile)

Settings → MCP Servers → Add your server URL: `https://mcp.yourdomain.com/mcp`

Claude Code (Terminal)

Add to your config:

```
{
  "mcpServers": {
    "my-server": {
      "url": "https://mcp.yourdomain.com/mcp"
    }
  }
}
```

Claude Desktop

Add to `~/.config/claude/claude_desktop_config.json`:

```
{
  "mcpServers": {
    "my-server": {
      "url": "https://mcp.yourdomain.com/mcp"
    }
  }
}
```

After connecting, ask Claude: *"What tools do you have from my MCP server?"* It should list them.

Gotchas and Lessons Learned

Timeouts. If a tool takes more than 30 seconds, the connection may drop. Set timeouts on your operations.

Too much data. If a tool returns 50,000 lines, Claude's context window fills up. Paginate or limit output.

Test before connecting Claude. Test each tool with `curl` first. A broken tool is easier to debug directly than through Claude.

Restart after code changes. Modified `server.py`? Restart the service: `sudo systemctl restart my-mcp`

Put it in git. Version your server code. When something breaks, you can roll back.

Log everything. Add logging. When something breaks at 11 PM, you'll want to see what happened.

What This Enables

Once your MCP server is running, Claude stops being just an advisor and becomes an operator. It can read your files, check your systems, update your content, query your data, and execute workflows - all through a controlled interface you built.

A law firm might expose case management, document assembly, and client lookup. A publisher might expose content management and analytics. A small business might expose inventory and invoicing.

Start small - file access and a command runner. Add tools for the specific things you need. You'll be surprised how fast it becomes essential.

This system was developed through daily production use in a law practice, publishing company, and multi-server infrastructure.

About the Author

Patrick Nolan is a solo estate planning attorney at Nolan Law Firm in Kirksville, Missouri. He is a U.S. Army veteran, former reporter, and the author of *Dead Man's Guide to Estate Planning*. His practice runs on flat fees, aggressive automation, and the belief that a 70% solution shipped today beats a perfect one shipped never.

Find him at @PatTalksLaw and nemolegal.com.

See our public-facing AI Use Policy.

This guide was written by Patrick Nolan and Claude.