Lesson 2 · Build an MCP Server

PDF: this lesson · **Install (Linux · macOS · Windows):** guide · PDF

> **Part of local-ai-lab** - a hands-on course for building local AI.

> **Interactive version (slides):** https://nikolareljin.github.io/local-ai-lab/lesson-2-mcp.html **Read it locally (no GitHub Pages):** `./run -l 2 lesson` **Course home:** https://nikolareljin.github.io/local-ai-lab/ **Source:** https://github.com/nikolareljin/local-ai-lab · the working server is `mcp_server.py` **Time:** ~30-45 min · **Prerequisites:** Lesson 1 · full objectives in SYLLABUS.md

> **Lessons:** 1 · RAG → **2 · MCP (you are here)** → 3 · Hybrid retrieval → 4 · RAG safety → 5 · RAG evaluation → 6 · Repo assistant → 7 · LangChain → ... → 15 · Docs from changes

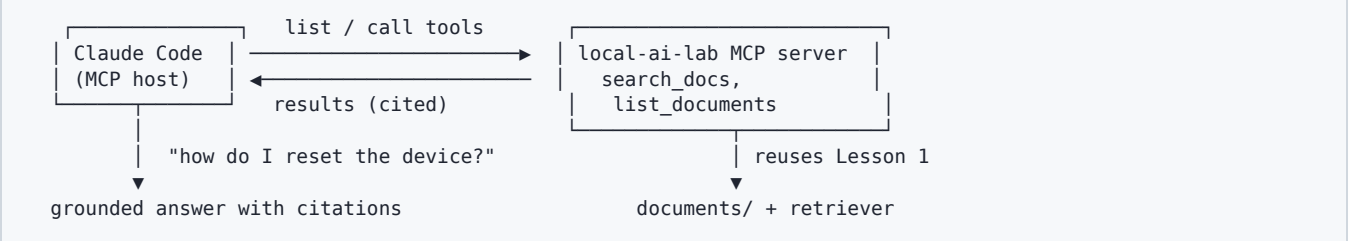
> **Status: complete & working.** Runnable code: `mcp_server.py`, tested by `tests/test_mcp.py`. Runs 100% locally.

> **Polyglot:** this lesson ships in **Python** (reference, below), **Node.js** (`node/lesson-2`) and **C# / .NET 8** (`dotnet/lesson-2`) - each on its official MCP SDK, each reusing its own Lesson 1 retriever. Run the demo of any of them with `./run -l 2 --lang python|node|csharp demo` (bare `./run -l 2` opens the interactive tool GUI).

What you'll learn

In Lesson 1 **your app** drove the pipeline and called the LLM. In Lesson 2 the relationship **flips**: **the LLM** drives, and your retriever becomes a **tool** it reaches for on demand. Same retrieval engine, new integration surface.

You'll build a real **MCP server** that exposes the Lesson 1 document search as tools, test it with the SDK's stdio client, and register it with Claude Code so you can ask questions about your files in plain chat.



Concept: what is MCP, and why?

The **Model Context Protocol (MCP)** is an open standard that lets an AI client discover and call external **tools** over JSON-RPC. A **server** advertises tools (name, description, input schema); the **host** (Claude Code) lists them and calls them when useful, feeding the results

back into the model's answer.

Three ideas to hold onto:

- **Tools** - functions the model can call (here: `search_docs`, `list_documents`).
- **Transport** - we use **stdio**: the host launches your server as a subprocess and talks over stdin/stdout. (HTTP/SSE transports also exist for remote servers.)
- **The handshake** - `initialize` → `list tools` → `call tool`.

Why it matters: RAG you can call from **any** MCP-aware client, with no bespoke UI. Your private documents become a first-class capability of the assistant itself.

Prerequisites

Finish Lesson 1 - the server is a thin wrapper over the retriever you built there. Then add the official Python SDK:

```
pip install mcp        # already in requirements.txt
```

The SDK ships **FastMCP** (a high-level server API) and an **stdio client** we'll use to test.

Step 1 · The server skeleton

Create `mcp_server.py`. `FastMCP` gives you a server object; tools are just decorated functions. Their **docstring becomes the description** the model reads, and the **type hints become the input schema** - so write them for the model.

```
from mcp.server.fastmcp import FastMCP

from localrag.config import load_config
from localrag.engine import get_retriever
from localrag.extract import discover_files

mcp = FastMCP("local-ai-lab-docs")

# ... tools go here ...

def main():
   mcp.run()        # default transport is stdio

if __name__ == "__main__":
   main()
```

> **Notice the imports.** `load_config`, `get_retriever`, and `discover_files` come straight from Lesson 1. MCP is a new doorway onto the same engine.

Step 2 · The `search_docs` tool

The star of the show. It runs the Lesson 1 retriever and returns passages tagged `[source:page]` so the model can cite them.

```
@mcp.tool()
def search_docs(query: str, k: int = 5) -> str:
   """Search the user's local documents and return the most relevant passages.

   Each passage is prefixed with its source as [filename:page] so the model
   can cite it. Call this to ground answers in the user's own files instead
   of relying on training data.
   """
   config = load_config()
   hits = get_retriever(config).search(query, max(1, int(k)))
   if not hits:
       return "No relevant passages found in the local documents."
   return "\n\n".join(
       f"[{h['source']}:{h['page_number']}] {h['text']}" for h in hits)
```

> **The docstring is a prompt.** The model reads it to decide **when** to call the tool, so it explicitly says "to ground answers... instead of relying on training data." Good tool descriptions are as important as good code.

Step 3 · A second tool: `list_documents`

Servers usually expose more than one tool. This one lets the model see what's in the corpus before searching.

```
@mcp.tool()
def list_documents() -> str:
   """List the documents currently available to search in the local corpus."""
   config = load_config()
   names = [p.name for p in discover_files(config.docs_dir)]
   return "\n".join(names) if names else "(no documents indexed yet)"
```

> **Tool design tip:** keep each tool small and single-purpose with a clear name. The model composes them - it might call `list_documents`, then `search_docs` - just like you'd chain functions.

The complete file is `mcp_server.py`.

Step 4 · Run it over stdio

`mcp.run()` defaults to the **stdio** transport: it reads JSON-RPC from stdin and writes to stdout. That's exactly how a host like Claude Code launches a local server.

```
python mcp_server.py        # waits silently for an MCP client to connect
```

> It looks like it's hanging - that's correct. The server is waiting for a client to speak the protocol over stdin. You won't talk to it by hand; the next step drives it with a client.

Step 5 · Test it with an stdio client

The SDK includes a client. This spawns the server, does the handshake, lists tools, and calls `search_docs` - a real integration test, **no LLM needed**.

```
# tests/test_mcp.py (essence)
from mcp import ClientSession
from mcp.client.stdio import StdioServerParameters, stdio_client

async def run():
   params = StdioServerParameters(command=sys.executable,
                                  args=["mcp_server.py"], cwd=str(ROOT))
   async with stdio_client(params) as (read, write):
       async with ClientSession(read, write) as session:
           await session.initialize()
           tools = await session.list_tools()
           result = await session.call_tool(
               "search_docs", {"query": "how do I reset the device", "k": 3})
           text = "".join(c.text for c in result.content)
           return [t.name for t in tools.tools], text
```

```
pytest -q tests/test_mcp.py     # passes: tools listed, cited passage returned
```

> **The handshake in code:** `initialize()` → `list_tools()` → `call_tool()`. That's the entire MCP lifecycle. The test asserts the result contains `power button` and `sample_manual.md` - grounded, cited, verified.

Step 6 · Register it with Claude Code

Now hand the server to a real host. From the repo directory:

```
claude mcp add local-ai-lab-docs -- python mcp_server.py
claude mcp list                          # confirm it's registered
```

This tells Claude Code: "when you start, launch `python mcp_server.py` and treat its tools as your own." Prefer an **absolute path** to `mcp_server.py` and your project's **venv Python** if you run Claude Code from elsewhere. (Equivalent to editing your MCP config JSON by hand.)

Step 7 · See it work - RAG, native to the assistant

Open Claude Code in the repo and just ask:

```
You:    How do I reset the device?

Claude: (calls search_docs "reset device")
       Hold the power button for 10 seconds until the LED blinks blue
       three times. [sample_manual.md:1]
```

Your local, private documents are now a tool the assistant uses on its own initiative - the same retriever, reachable from any MCP host.

Recap

Piece	What it does
`FastMCP("...")`	the server; handles all protocol plumbing
`@mcp.tool()`	turns a function into a callable tool (docstring = description, hints = schema)
`search_docs` / `list_documents`	your tools, reusing the Lesson 1 engine
`mcp.run()`	serves over stdio for the host to launch
`claude mcp add`	registers it so Claude Code can call it

> **The through-line:** `search_docs` is the same capability you'll rebuild in every later lesson - as an Ollama function call, a Semantic Kernel plugin, a Bedrock action group, and a Google ADK tool. Master the primitive once; the frameworks are just wrappers.

Exercises

- Add a `get_document(name)` tool that returns a whole file's text.
- Expose the corpus as an MCP **resource** (not just a tool) so hosts can browse it.
- Return structured JSON (source, page, score) instead of a flat string and let the host format it.

Other languages (polyglot)

MCP is multi-language, and so is this lesson. The same two tools - `search_docs` and `list_documents`, same `[filename:page]` citation format - are implemented on each language's **official** MCP SDK, reusing that language's Lesson 1 retriever. No LLM is needed to test any of them; the demo client drives the server over stdio and prints cited passages.

```
./run -l 2 demo                 # Python (reference) · FastMCP + mcp stdio client
./run -l 2 --lang node demo     # Node.js     · @modelcontextprotocol/sdk (McpServer)
./run -l 2 --lang csharp demo   # C# / .NET 8 · ModelContextProtocol ([McpServerTool])
```

(Bare `./run -l 2` opens the interactive tool GUI instead; `demo` is the one-shot stdio client.)

Language	Server entry	SDK	Details
Python	`mcp_server.py`	`mcp` (FastMCP)	the reference above
Node.js	`node/lesson-2/src/server.js`	`@modelcontextprotocol/sdk`	README
C# / .NET	`dotnet/lesson-2/Program.cs`	`ModelContextProtocol`	README

Each server registers under a distinct name (`local-ai-lab-docs`, `local-ai-lab-docs-node`, `local-ai-lab-docs-dotnet`), so all three can coexist in Claude Code at once. Pick a language with the **Follow along in:** selector on the interactive slides.

Next lesson

Lesson 3 · Hybrid Retrieval & Reranking → - fuse BM25 with a semantic arm using Reciprocal Rank Fusion, then rerank the merged list.

Course: nikolareljin.github.io/local-ai-lab · **Source:** github.com/nikolareljin/local-ai-lab · **Author:** Nik Reljin