Level Up Coding · Follow publication

How I Built a Local MCP Server to Connect Obsidian with AI

11 min read · Apr 28, 2025


Hyunjong Lee Follow

Photo by Maxence Pira on Unsplash

1. Introduction

In this article, I'll walk through how I built my own MCP(Model Context Protocol) server to enable AI models to access and reason over my personal knowledge base.

Why I started this project

Whenever rganize concepts a ough, I've accumulated a lot of empty pages (just titles with no content. 😅)

As LLM and AI agents become more capable, I've been thinking about how to better exploit them into my personal workflows. I've been using ChatGPT or Claude for brainstorming, writing, and learning new AI concepts. But when Claude Desktop introduced support for MCP, a protocol that allows AI models to extend their context with external tools and data sources, it opened up a new possibility: enabling the AI to understand and utilize my own knowledge base.

Why I built my own MCP server insted of using the official one

My personal knowledge base is just a directory full of markdown files. The official MCP FileSystem server could have technically worked, but I didn't feel comfortable giving a third-party tool access to my full file system or even revealing the directory structure. Also, Some of the content in my notes is confidential.

More importantly, I wanted full control. I didn't want the AI to modify anything. I just wanted to give it read-only access to selected content. And of course, I was curious to build one myself.

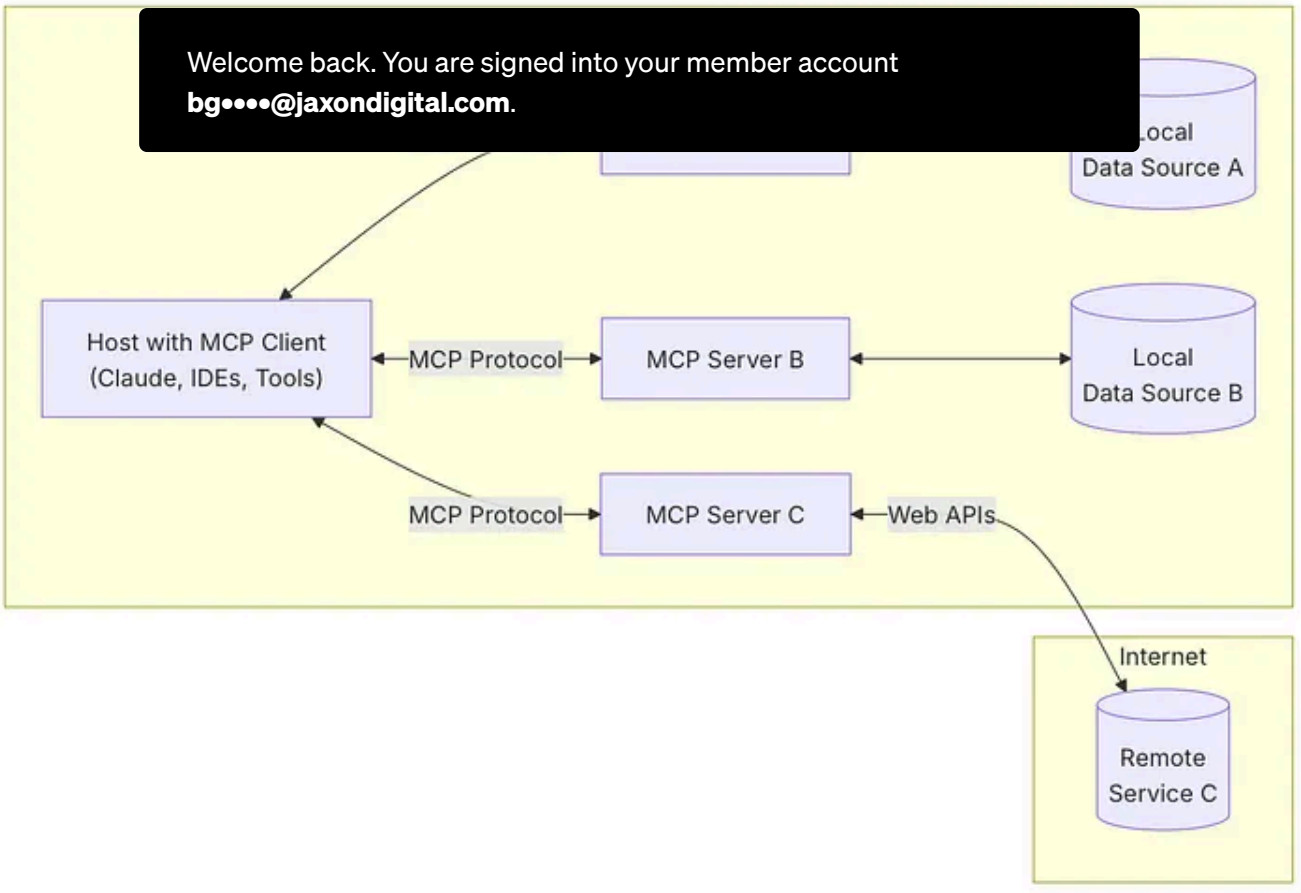
What I aim to do with this AI + knowledge base integration:

- Identify which notes are still marked as TBD and need completion
- Analyze inconsistencies or missing pieces in my documentation
- Generate short-answer questions based on existing notes to refresh knowledge

2. What is MCP?

Before diving into the implementation, let's quickly review what MCP is. If you've already gone through the official MCP docs, you can skip ahead to the next section.

MCP(Model Context Protocol) is an open protocol that standardizes how applications provide context to language models. It allows LLMs to interact with external tools and data sources in a modular way.



MCP Client-Server Architecture (Image by MCP Official Doc.)

MCP follows a client-server architecture, where the host application can connect to multiple MCP servers. Here's how its system is structured:

- **MCP Hosts:** Applications like Claude Desktop or other AI tools that access data through the MCP protocol.
- **MCP Clients:** Protocol clients that maintain a 1:1 connection with a server.
- **MCP Servers:** Lightweight programs that expose specific functionalities through the MCP interface.
- **Local Data Sources:** Files, databases, or service available on the host machine that the MCP server can access
- **Remote Services:** External systems the server can connect to.

Resources

Resources are the core primitives of MCP. They expose data or content on the server to the LLM, either for contextual understanding or for direct interaction. These resources are typically read by the client and used to enrich the model's responses.

Tools

Tools are a

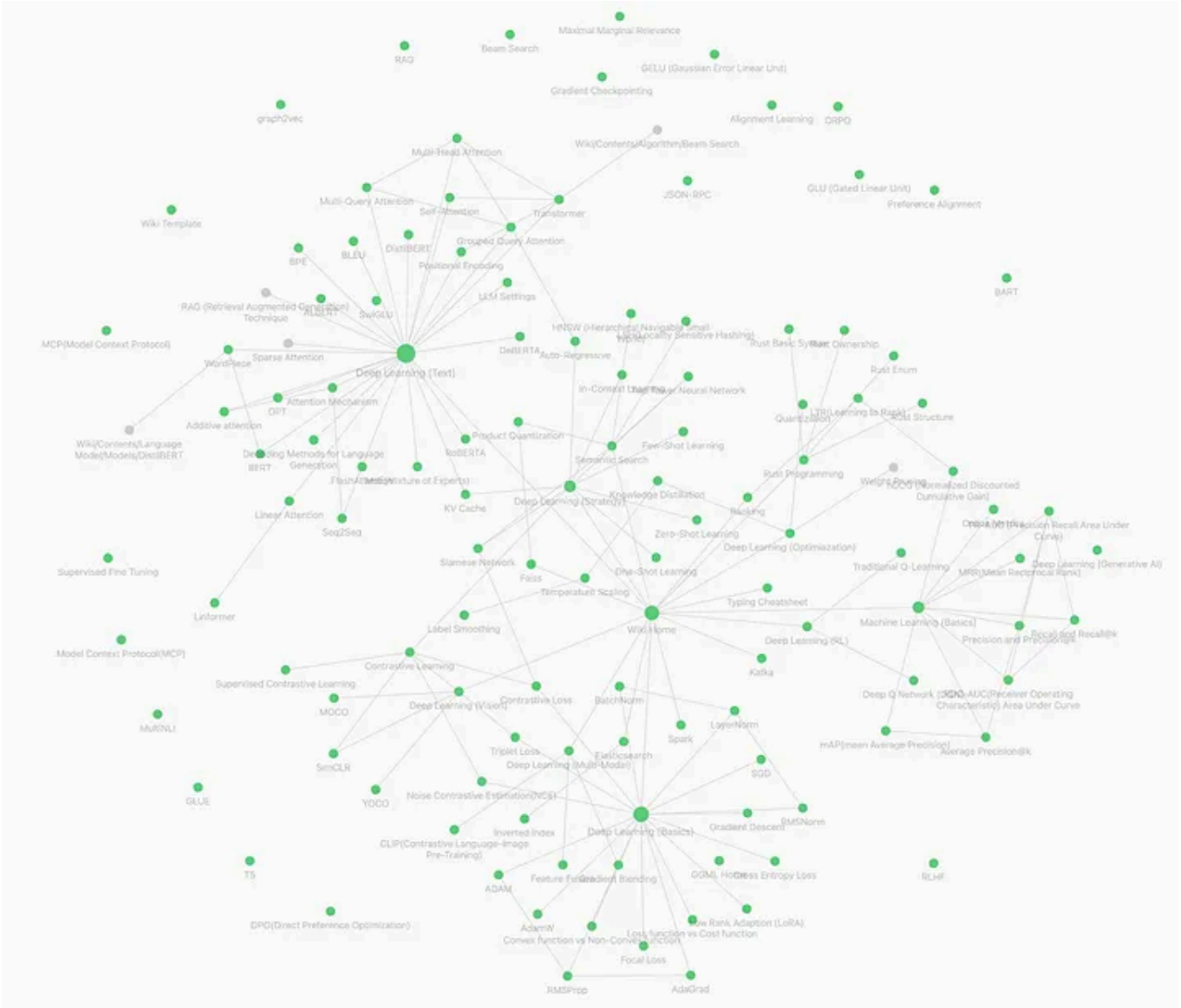
table functions t

ns, perfom computations, or even trigger real-world actions.

3. My Own Knowledge Base

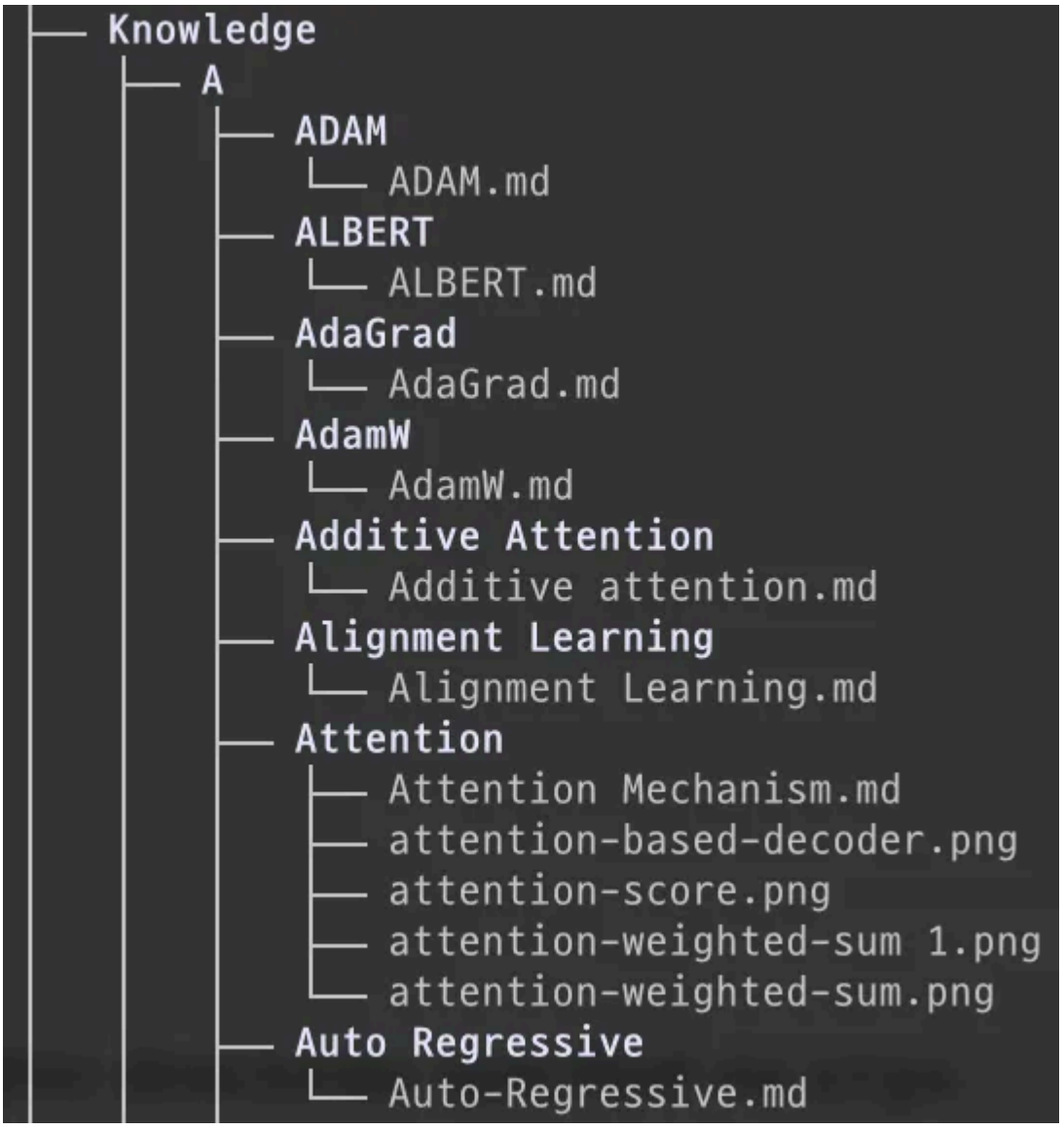
Whenever I come across a new concept while reading books or technical articles, I jot it down as a note in Obsidian. I then organize the content and link it to other related or dependent concepts, gradually building and managing a massive, interconnected knowledge base.

The image below shows a graph view of my Obsidian notes. it includes a mix of well-sturectured content, untouched stub notes with only titles, and orphan notes that are not yet connected to any other concept.



Obsidian Graph View for Knowledge Notes (Image by Author)

All of these notes are written in Markdown format. I went through several ways on how to org nd their related ima ed concepts semantically, I also created a dedicated “home” Markdown file that contains links to categorized note entries.



Directory Structure (Image by Author)

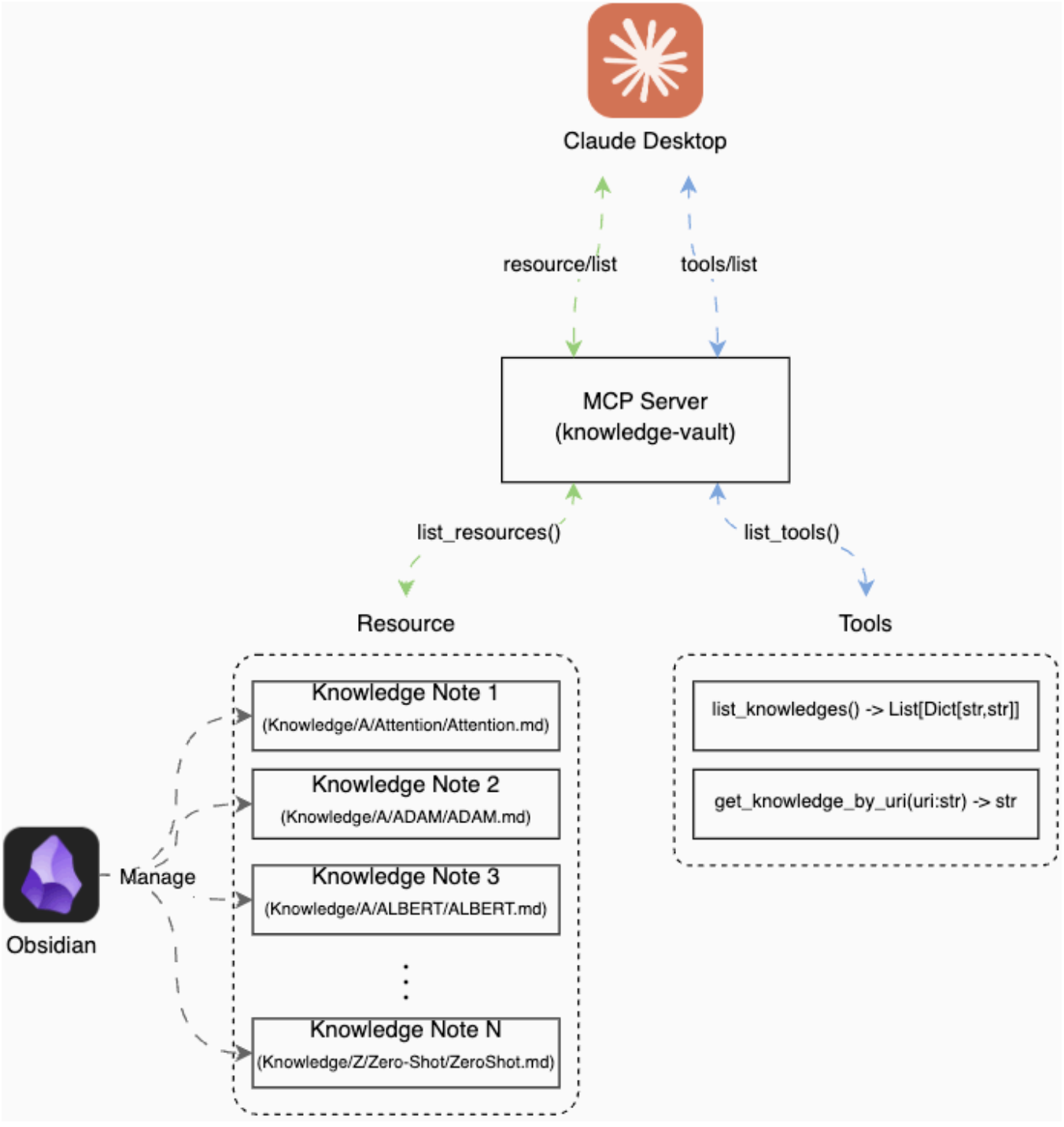
In this setup, the MCP server accesses the set of Markdown notes as its resources. Since these files are simple documents in a local file system — not a database — implementing the MCP server becomes significantly easier.

4. Building MCP Server

4.1. Architecture

The overall architecture of the MCP server I implemented is shown in the diagram below. I named the server “knowledge-vault”.

Each knowledge note is written using Obsidian editor. Once written, these Markdown [redacted] ed tools that allow [redacted] le resources or by reading the content of a specific file.



MCP Server Diagram (Image by Author)

When the MCP client in Claude Desktop sends a "*resource/list*" request, the server responds with metadata for all registered knowledge notes. Similarly, when it receives a "*tools/list*" request, the server returns metadata about the available tools.

4.2. MCP Server

The MCP server is implemented using the Python MCP SDK, and I used the FastMCP class. Whil r class, doing so re "*resource/read*", "*tools/list*", "*tools/read*", "*tools/call*" and so on.

With FastMCP, however, implementation becomes much easier. you simply register the resources and tools you want to expose.

```
class KnowledgeVault:
    def __init__(self):
        self.app = FastMCP("knowledge-vault")

        self.resource_map = {}

        self._init_resources()
        self._init_tools()
```

To organize the server logic, I defined a *KnowledgeVault* class that wraps the *FastMCP* instance along with the resources and tools to be registered.

Internally, the *FastMCP* class includes a *ResourceManager* object that manages all registered resources. However, I chose to maintain a separate resource map within my server to track metadata for each registered resources.

Both "*_init_resources()*" and "*_init_tools()*" methods are responsible for registering their respective components with the MCP server.

4.2. Resources

The *ResourceManager* inside the *FastMCP* object requires you to define each resource using a *Resource* class instance. This class specifies metadata such as the resource's URI, name, description, and MIME type, which are exchanged between the client and server.

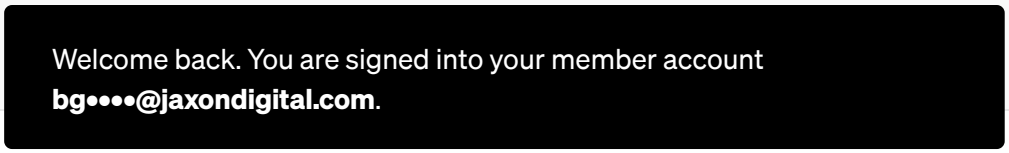
```
class Resource(BaseModel, abc.ABC):
    """Base class for all resources."""

    model_config = ConfigDict(validate_default=True)

    uri: Annotated[AnyUrl, UrlConstraints(host_required=False)] = Field(
        default=..., description="URI of the resource"
```

```
    )
    nam                                                    ault=None)
    des
    )
    mime_type: str = Field(
        default="text/plain",
        description="MIME type of the resource content",
        pattern=r"^[a-zA-Z0-9]+/[a-zA-Z0-9\-+.]+$",
    )
...

    @abc.abstractmethod
    async def read(self) -> str | bytes:
        """Read the resource content."""
        pass
```


When a client requests to read a resource, the server invokes the *read()* method. This method is abstract, so it must be overridden to return the content of the specific resource.

In this MCP server, I created a custom *MarkdownResource* class by inheriting from the *Resource* class. I added a custom field for file size and implemented the *read()* method to handle reading the file content.

```
class MarkdownResource(Resource):
    size: int = -1

    async def read(self) -> str | bytes:
        uri = uri2path(str(self.uri))

        if not uri.startswith("file://"):
            raise ValueError(f"Only file resource permits. got uri({uri})")

        path = uri[len('file://'):]
        file_path = os.path.join(VAULT_PATH, path).lower()

        if not os.path.exists(file_path):
            raise ValueError(f"file_path({file_path}) not exists")

        async with aiofiles.open(file_path, 'r') as fd:
            contents = await fd.read()
```



Importantly, the uri field stores relative paths rather than absolute ones. This prevents exposing my full file system paths to the AI model.

When the *read()* method is called, the server converts the relative URI into an absolute path by prepending a hidden base path (*VAULT_PATH*). This ensures secure and consistent file access.

```
class KnowledgeVault:
    ...
    def _init_resources(self):
        file_list = glob.glob(os.path.join(VAULT_PATH, '*/*/*.md'))

        self.resource_map = {}
        for path in file_list:
            path = path.lower()
            uri = f"file://{os.path.relpath(path, start=VAULT_PATH)}".lower()

            rsrc = MarkdownResource(
                uri=uri,
                name=os.path.basename(path).split('.')[0],
                mime_type="text/markdown",
                size=os.path.getsize(path)
            )
            self.resource_map[uri] = rsrc
            self.app.add_resource(rsrc)
```

Inside the *KnowledgeVault* class, Markdown files are scanned, and each file is wrapped into a *MarkdownResource* instance using a relative *"file://"* URI. These resources are then registered with the MCP server via the "*add_resource()*" method.

Once resources are successfully registered, you can see the full list of resources that have been exposed by the MCP server.

Registered Resources in Claude Desktop (Image by Author)

According to the official MCP documentation, resources are meant to be controlled by the client application. For instance, Claude Desktop does not automatically consume registered resources. Instead, it requires the user to explicitly select which resources they want the AI to use.

This approach provides user control and transparency — but it wasn't quite what I needed. My goal was for Claude AI to automatically access relevant notes and retrieve their content when needed. So I defined a custom tool that allows the AI to directly read the contents of registered resources, without requiring user interaction.

4.3. Tools

When I started this project, I thought I would need to implement a wide range of tools. But it turned out that the AI model was much more capable than I expected. With just a few basic tools, the model was able to accomplish most of the task I needed on its own.

Here are the tools I implemented. I used decorators provided by the *FastMCP* object to register them directly.

```
class KnowledgeVault:
 ...
```

```
    def _
        @s
        async def list_knowledges() -> list[dict[str, str]]:
            '''List the names and URIs of all knowledges written in the the vault
            '''
            return [{'name':rsrc.name, 'uri':rsrc.uri, 'size':rsrc.size} for rsrc

        @self.app.tool()
        async def get_knowledge_by_uri(uri:str) -> str:
            '''get contents of the knowledge resource by uri
            '''

            uri = uri2path(uri)
            rsrc = self.resource_map.get(uri, None)
            if not rsrc:
                raise ValueError(f"Not registered resource URI")

            return await rsrc.read()
```


As you can see from the code, I created two tools:

- **list_knowledges()** : returns a list of all registered knowledge notes.
- **get_knowledge_by_uri(uri)**: reads and returns the content of a specific note, given its URI.

Once these tools are successfully registered, Claude Desktop displays them like this.

Registered Tools in Claude Desktop (Image by Author)

4.4. Integration

To run the MCP server, simply call the *run()* method of the *FastMCP* instance. Since this setup i ver-sent events(SSE tput) for communication.

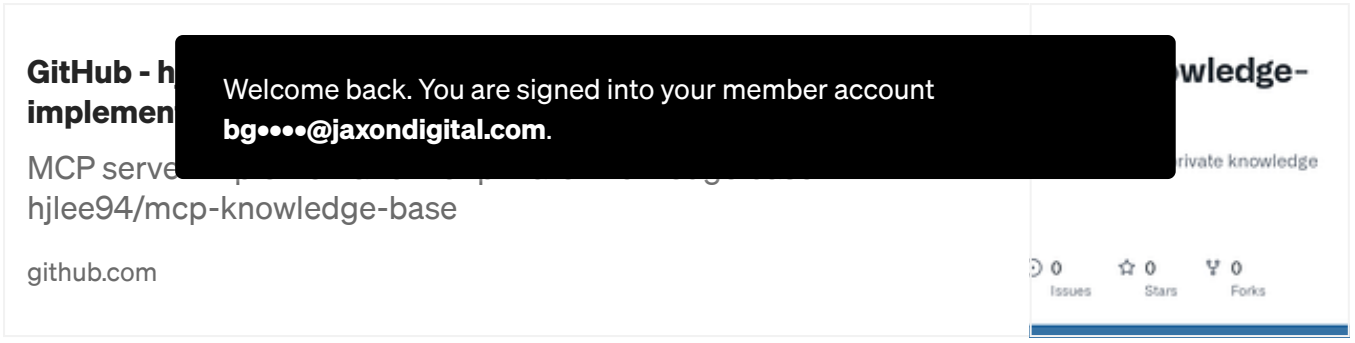

```
class KnowledgeVault:
 ...
    def run(self):
        self.app.run(transport='stdio')

if __name__ == '__main__':
    app = KnowledgeVault()
    app.run()
```

To make Claude Desktop communicate with this MCP server over standard I/O, you'll need to update the configuration.

For macOS users, this involves modifying the config json file of Claude Desktop. This file is located inside the application's configuration directory(*~/Library/Application\ Support/Claude/claude_desktop_config.json*). In that file, you can specify the executable path for your local MCP server script so that Claude Desktop knows how to launch and connect to it.

```
{
    "mcpServers": {
        "knowledge-vault": {
            "command": "uv",
            "args": [
                "--directory",
                "<package directory>",
                "run",
                "main.py"
            ]
        }
    }
}
```

The implementation of the MCP server is complete. You can find the full source code in the GitHub repository linked below.



5. Results

Now, let's see whether the AI model can perform the tasks on my behalf.

When I prompted Claude with a query like the one below, the model correctly invoked the tool and retrieved the list of notes stored in the vault.

Query for invoking the "list_knowledges()" tool (Image by Author)

The MCP server responded with a list of resources. As designed, the response hides certain abs

Response by MCP Server (Image by Author)

B. get_knowledge_by_uri()

Using the URI from the previous response, the model then invoked the tool to read a specific note and successfully returned the full content of the Markdown file.

Query for invoking the “get_knowledge_by_uri(uri)” tool and its response (Image by Author)

5.2. Identifying Incomplete Notes

Next, I asked the AI to find notes that were merely stubs — titles with no content. The model … esented them as em…

Query for Identifying Incomplete Notes (Image by Author)

Then I asked the model to go a step further and identify notes with weak or minimal content, based on their actual text.

Query for Searching Notes have Insufficient contents (Image by Author)

Impressively, the AI scanned each resource, read its content, and returned a list of knowledge notes with insufficient information.

Search Results (Image by Author)

Even more powerful, the model aggregated this information into a table showing the completion status of all knowledge notes.

Query for Creating Table about Completion Status of Each Notes (Image by Author)

Using this table, I can now systematically track and improve the weakest parts of my knowledge base.

Created Table based on Knowledge Base (Image by Author)

In addition, the model was able to flag notes that were poorly structured or contained inconsistent formatting — making it much easier to maintain a clean and useful knowledge base.

5.3. Generating Short-Answer Questions from Existing Notes

To reinforce my memory of what I had documented in the past, I asked the AI to generate descriptive, short-answer questions based on one of my previously written knowledge notes.

For example, I prompt the model with a request like this.

Query for Generating Short-Answer Questions (Image by Author)

The result was exactly what I had hoped for: the model selected a note, analyzed its content, and generated a relevant open-ended question that could help me recall and internalize the key ideas.

This technique has proven to be incredibly helpful for revisiting older notes and identifying which topics I've truly understood.

6. Conclusion

Here are the takeaways from this article:

- MCP enables AI models to securely access external data sources.
- By building a custom MCP server, you can give AI access to data under your specific requirements and control.
- Even with just a few simple tools, LLMs can perform surprisingly powerful tasks — reasoning about which tools to invoke and which resources to access, entirely on their own.

While Claude Desktop provided a convenient way to validate the MCP integration, this setup can be made fully local and independent. My next goal is to:

- Build a custom MCP client
- Connec
- Eliminate the dependency on external AI providers

This will give me full control over the entire pipeline — from the model to the knowledge base — and enable a truly private, offline AI assistant that knows everything I've written.

. . .

Any feedback about this article or the source code is welcome. If you are interested in future articles, just follow me. If you want to discuss further topics, feel free to connect with me on LinkedIn.

Mcp Server | AI | Obsidian | Llm | Python

Responses (6)

Bgerby

What are your thoughts?

See all responses

More from Hyunjong Lee and Level Up Coding

In Level Up Coding by Hyunjong Lee

How I Built a Tool-Calling Llama Agent with a Custom MCP Server

Integrating sLLMs, tool-calling, and custom context retrieval via a private MCP server

May 18 255 4

In Level Up Coding by Fareed Khan

Building 17 Agentic AI Patterns and Their Role in Large-Scale AI Systems

Ensembling, Meta-Control, ToT, Reflexive, PEV and more

Sep 25 2K 43

In Level Up Coding by Vivedha Elango

How to choose the right embedding model for your RAG application?

A Comprehensive Guide to Selecting the Ideal Embedding Model for Your RAG Application

Sep 16 625 13

Hyunjong Lee

Understanding the Seq2Seq Model—What You Should Know Before Understanding Transformers

This is a good starting point or review to understand the inner concepts of the Transformer and how the DL models generate text.

Jul 16, 2024 1

See all from Hyunjong Lee

See all from Level Up Coding

Recommended from Medium

Tosny

7 Websites I Visit Every Day in 2025

If there is one thing I am addicted to, besides coffee, it is the internet.

Sep 23 3.8K 153

In Heptabase by 詹雨安 Alan Chan

The Best Way to Use AI for Learning

An effective method of learning complex knowledge with AI.

Sep 26 1.8K 22

Mihailo Zoin

The Obsidian-NotebookLM Integration Hack: Building Your External Brain

Strategy: Cross-System Cognitive Amplification Loop

Oct 7

In Data Science Collective by Amanda Iglesias Moreno

NotebookLM Just Got a Serious Upgrade—Exploring NotebookLM's Newest Features and Updates

What's New and Why It Matters

Sep 29 823 23

Berker Ceylan

What Is The Best Diagramming Software in 2025

I wasted thre[illegible] p doubled my team's ou[illegible]

Welcome back. You are signed into your member account **bg••••@jaxondigital.com**.

Sep 11 [illegible]

Chris Evans

Choosing a Terminal on macOS (2025): iTerm2 vs Ghostty vs WezTerm vs kitty vs Alacritty

Developer-first guide with practical setups.

Sep 20 37 7

See more recommendations