Understanding MCP (Model Context Protocol)

What is MCP?

Model Context Protocol (MCP) is a standardized protocol that allows AI models (like those running in Ollama) to interact with external tools, data sources, and services. Think of it as a bridge that connects AI models to the real world.

Why MCP?

- **Extends AI capabilities**: AI models can now read files, call APIs, interact with databases, etc.
- **Standardized**: One protocol works with multiple AI models
- **Modular**: Add new tools without changing the AI model
- **Secure**: Controlled access to external resources

MCP Components

MCP servers provide three main types of components:

1. Tools 🔧

Tools are **functions that the AI can call** to perform actions or get information.

Characteristics:

- Can take **parameters** (inputs)
- Return **results** (outputs)
- Execute **actions** (read files, make calculations, call APIs)
- AI decides **when and how** to use them

Example from your code:

```python
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two integers and return the result."""
    return a + b
```

How it works:

1. User asks: "What is 5 + 7?"
2. AI recognizes it needs the `add` tool

3. AI calls: `add(a=5, b=7)`
4. Tool returns: `12`
5. AI responds: "The answer is 12"

Types of Tools:

a) Synchronous Tools (regular functions)

```python
@mcp.tool()
def read_greeting() -> str:
    """Read and return the contents of greeting.txt file."""
    with open("greeting.txt", "r") as f:
        return f.read()
```

b) Asynchronous Tools (async functions)

```python
@mcp.tool()
async def echo(message: str) -> str:
    """Echo back the message."""
    return message
```

c) Tools with Multiple Parameters

```python
@mcp.tool()
def multiply(a: int, b: int, c: int = 1) -> int:
    """Multiply numbers together."""
    return a * b * c
```

d) Tools that Return Complex Data

```python
```

```python
@mcp.tool()
def get_user_info(user_id: int) -> dict:
    """Get user information."""
    return {
        "id": user_id,
        "name": "John Doe",
        "email": "john@example.com"
    }
```

---

2. Prompts 💬

Prompts are **pre-defined templates** that help users interact with the AI in specific ways.

Characteristics:

- **Template-based**: Reusable prompt patterns
- **Parameterized**: Can accept dynamic inputs
- **Contextual**: Provide specific instructions to the AI

Example from your code:

```python
@mcp.prompt()
async def greeting_prompt(name: str) -> str:
    """A simple greeting prompt."""
    return f"Greet {name} kindly lusu payalea."
```

How it works:

1. User invokes: `greeting_prompt(name="Alice")`
2. Returns prompt: "Greet Alice kindly lusu payalea."
3. This prompt is sent to the AI model
4. AI generates a kind greeting for Alice

More Prompt Examples:

a) Code Review Prompt

```python
```

```
@mcp.prompt()
def code_review_prompt(code: str, language: str) -> str:
    """Generate a code review prompt."""
    return f"""Review this {language} code:

{code}

Please check for:
- Code quality
- Best practices
- Potential bugs
- Performance issues"""
```

b) Translation Prompt

```
python

@mcp.prompt()
def translate_prompt(text: str, target_lang: str) -> str:
    """Generate a translation prompt."""
    return f"Translate the following to {target_lang}: {text}"
```

c) Summarization Prompt

```
python

@mcp.prompt()
def summarize_prompt(document: str, max_words: int = 100) -> str:
    """Generate a summarization prompt."""
    return f"Summarize in {max_words} words: {document}"
```

---

3. Resources 📁

Resources are **data sources** that the AI can access for information.

Characteristics:

- **URI-based**: Accessed using unique identifiers
- **Read-only**: Typically for fetching data
- **Static or Dynamic**: Can be files, database queries, API responses

Example from your code:

```
python
```

```python
@mcp.resource("file://./greeting.txt")
def greeting_file() -> str:
    """Read and return the contents of greeting.txt."""
    with open("greeting.txt", "r", encoding="utf-8") as f:
        return f.read()
```

How it works:

1. Resource is registered with URI: `file://./greeting.txt`
2. Client can request this resource by URI
3. Function executes and returns content
4. Content is available to the AI

More Resource Examples:

a) Multiple File Resources

```python
@mcp.resource("file://./config.json")
def config_file() -> str:
    """Return configuration file."""
    with open("config.json", "r") as f:
        return f.read()

@mcp.resource("file://./data.csv")
def data_file() -> str:
    """Return data file."""
    with open("data.csv", "r") as f:
        return f.read()
```

b) Dynamic Resources

```python
@mcp.resource("db://users/{user_id}")
def user_resource(user_id: str) -> str:
    """Fetch user from database."""
    # Simulated database query
    return f"User data for ID: {user_id}"
```

c) API Resources

```python
```

```python
@mcp.resource("api://weather/{city}")
def weather_resource(city: str) -> str:
    """Get weather for a city."""
    # Simulated API call
    return f"Weather data for {city}"
```

Complete Example: Student Management System

Here's a comprehensive MCP server for a student management system:

```python
```

```python
from fastmcp import FastMCP
import json

mcp = FastMCP("student-management")

# ========== TOOLS ==========

@mcp.tool()
def add_student(name: str, age: int, grade: str) -> str:
    """Add a new student to the database."""
    student = {"name": name, "age": age, "grade": grade}
    # In real app, save to database
    return f"Student {name} added successfully!"

@mcp.tool()
def calculate_average(scores: list[int]) -> float:
    """Calculate average of test scores."""
    if not scores:
        return 0.0
    return sum(scores) / len(scores)

@mcp.tool()
def grade_letter(percentage: float) -> str:
    """Convert percentage to letter grade."""
    if percentage >= 90:
        return "A"
    elif percentage >= 80:
        return "B"
    elif percentage >= 70:
        return "C"
    elif percentage >= 60:
        return "D"
    else:
        return "F"

@mcp.tool()
async def send_notification(student: str, message: str) -> str:
    """Send notification to student."""
    # Simulated async operation
    return f"Notification sent to {student}: {message}"

# ========== PROMPTS ==========

@mcp.prompt()
def report_card_prompt(student_name: str, scores: str) -> str:
    """Generate a report card prompt."""
```

```
    return f"""Create a detailed report card for {student_name}.

Test Scores: {scores}

Include:
1. Overall performance analysis
2. Strengths and weaknesses
3. Recommendations for improvement
4. Encouraging message"""

@mcp.prompt()
def parent_letter_prompt(student_name: str, issue: str) -> str:
    """Generate a letter to parents."""
    return f"""Write a professional letter to the parents of {student_name}
regarding: {issue}

Keep it respectful, clear, and constructive."""

# ========== RESOURCES ==========

@mcp.resource("file://./student_roster.json")
def student_roster() -> str:
    """Get the complete student roster."""
    with open("student_roster.json", "r") as f:
        return f.read()

@mcp.resource("file://./attendance.csv")
def attendance_data() -> str:
    """Get attendance records."""
    with open("attendance.csv", "r") as f:
        return f.read()

@mcp.resource("file://./curriculum.txt")
def curriculum() -> str:
    """Get curriculum information."""
    with open("curriculum.txt", "r") as f:
        return f.read()

# ========== RUN SERVER ==========

def main():
    mcp.run(transport="sse", host="127.0.0.1", port=8000)

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

Comparison Table

Feature	Tools	Prompts	Resources
Purpose	Execute actions	Guide AI behavior	Provide data
AI Control	AI decides when to call	User invokes	AI/User requests
Parameters	Yes	Yes	Optional
Returns	Any data type	String (prompt text)	String (content)
Side Effects	Can modify data	No	No
Examples	Calculate, Write files	Templates, Instructions	Files, Databases

---

Best Practices

For Tools:

1. **Clear descriptions**: Help AI understand when to use the tool
2. **Type hints**: Use proper Python type annotations
3. **Error handling**: Return useful error messages
4. **Single responsibility**: Each tool does one thing well

For Prompts:

1. **Specific instructions**: Be clear about expected output
2. **Parameterization**: Make prompts flexible with parameters
3. **Context**: Provide enough information for good results

For Resources:

1. **Unique URIs**: Use descriptive, unique identifiers
2. **Fast access**: Resources should load quickly
3. **Error handling**: Handle missing files/data gracefully
4. **Documentation**: Explain what data the resource provides

---

Transport Types

Your MCP server can use different transport methods:

1. SSE (Server-Sent Events) - Currently using

```
python
```

```
mcp.run(transport="sse", host="127.0.0.1", port=8000)
```

- **Use case**: Real-time updates, web-based clients
- **Connection**: HTTP-based, one-way server-to-client streaming

2. STDIO (Standard Input/Output)

```
python

mcp.run(transport="stdio")
```

- **Use case**: Command-line tools, local processes
- **Connection**: Direct process communication

3. HTTP

```
python

mcp.run(transport="http", host="127.0.0.1", port=8000)
```

- **Use case**: Traditional request-response patterns
- **Connection**: Standard HTTP requests

Practice Exercises for Students

Exercise 1: Create a Calculator Tool

Add these tools to your MCP server:

- `subtract(a, b)`
- `multiply(a, b)`
- `divide(a, b)`

Exercise 2: Create File Management Tools

Add tools to:

- List files in a directory
- Create a new file
- Delete a file

Exercise 3: Create Custom Prompts

Create prompts for:

- Writing an essay on a topic
- Creating a study schedule
- Generating quiz questions

Exercise 4: Add Resources

Add resources for:

- A todo list file
- A notes file
- A schedule file

Testing Your MCP Server

```python
# Test client code
from mcp_client_for_ollama.client import MCPClient
import asyncio

async def main():
    client = MCPClient(
        model="qwen2.5:1.5b",
        host="http://localhost:11434"
    )

    await client.connect_to_servers(
        server_urls=["http://127.0.0.1:8000/sse"]
    )

    await client.chat_loop()

if __name__ == "__main__":
    asyncio.run(main())
```

Test with these prompts:

- "Add 10 and 20"
- "Read the greeting file"
- "Echo hello world"

Summary

MCP enables AI models to:

- 🔧 **Execute actions** via Tools
- 💬 **Follow templates** via Prompts
- 📁 **Access data** via Resources

This creates powerful AI applications that can interact with the real world while maintaining security and control!