Building Your First MCP Server and Client

Project Setup

Prerequisites

Install UV (Python package manager):

- **Linux/macOS**: `curl -LsSf https://astral.sh/uv/install.sh | sh`
- **Windows**: Run in PowerShell terminal

Initial Setup

```
mkdir basic-mcp-server
cd basic-mcp-server
uv init
uv venv
source .venv/bin/activate
uv add anthropic httpx mcp
```

Project Structure

Create two essential files:

- `mcp_server.py` - MCP server implementation
- `host_client_test.py` - Host client test implementation

Building the MCP Server

Basic Server Implementation

```
from mcp.server.fastmcp import FastMCP

# Initialize MCP server
mcp = FastMCP("simple-mcp-server")

@mcp.tool
def create_file(file_path: str, content: str) -> dict[str, str]:
    """Creates a file with specified content at given path"""
    # Implementation creates file with content
    return {"file_path": file_path, "content": content}

if __name__ == "__main__":
    mcp.run(transport="stdio")
```

Key Components

- **FastMCP**: Simplified server setup

- **@mcp.tool decorator**: Exposes functions as MCP tools
- **stdio transport**: Local communication protocol
- **Tool function**: Returns structured data (file path and content)

Testing with MCP Inspector

Launch Inspector

```
mcp uv run mcp_dev mcp_server.py
```

Inspector Features

- **Connection management**: Connect/disconnect to server
- **Tool exploration**: List and test available tools
- **Real-time testing**: Execute tools with sample data
- **Debug capabilities**: Inspect server responses and errors

Test Example

- Tool: `create_file`
- Input: `file_path="test.txt"`, `content="testing our first mcp server"`
- Result: File created with specified content

Building the Host Client

Communication Flow Implementation

The client simulates the complete MCP communication flow:

1. **User Interaction**: Receives user queries
2. **Host Processing**: LLM processes requests
3. **Capability Discovery**: Lists available server tools
4. **Capability Invocation**: Executes requested tools
5. **Result Integration**: Combines results into final response

Client Class Structure

```
class MCPClient:
    async def connect_to_server(self, script_path):
        # Establishes connection to MCP server

    async def process_query(self, query):
        # Handles LLM processing and tool execution

    async def chat_loop(self):
        # Provides interactive chat interface
```

Key Functions

Server Connection:

- Sets up transport parameters
- Manages async communication
- Handles Python/JS server detection

Query Processing:

- Lists available tools from server
- Sends query to LLM (Claude 3.5 Sonnet)
- Processes text responses and tool calls
- Executes tool invocations asynchronously

Chat Interface:

- Continuous user interaction loop
- Processes multiple queries until "quit"
- Maintains conversation context

Running the Complete System

Execution Command

```
uv run python host_client_test.py mcp_server.py
```

Example Interaction

User Input: "Write a simple essay with two paragraphs explaining why pancakes are the best breakfast of all time"

System Process:

1. LLM receives query
2. Discovers `create_file` tool
3. Generates essay content
4. Invokes tool to create file
5. Returns success confirmation

Result: Essay file created with generated content

Architecture Summary

Complete MCP Flow Demonstrated:

- **User** → sends query
- **Host** → processes with LLM
- **Client** → discovers capabilities and invokes tools
- **Server** → executes functions and returns results
- **Response** → integrated results returned to user

This implementation provides a working foundation for understanding MCP's core communication patterns and can be extended with additional tools and capabilities.