Lesson 6: Building MCP Server-Client with Streamable HTTP Transport

Overview

In this lesson, we'll build an MCP (Model Context Protocol) server-client example using Streamable HTTP transport. This approach is essential when you want to deploy your MCP servers to be accessed across the internet, potentially sell them in an MCP marketplace, or enable remote MCP usage.

Learning Objectives

By the end of this lesson, you will be able to:

1. Set up an MCP server with Streamable HTTP transport
2. Create a personal assistant MCP server with file management capabilities
3. Build a client that connects to the server using Streamable HTTP
4. Test the server using MCP Inspector
5. Understand the differences between STDIO and Streamable HTTP transports

Prerequisites

- Basic understanding of Python
- Familiarity with MCP concepts from previous lessons
- UV package manager installed
- API key for Anthropic (for client testing)

Setting Up the Project

Step 1: Initialize the Project

Create a new project folder and initialize it with UV:

```
uv init
```

This creates a `pyproject.toml` file for your project configuration.

Step 2: Create Virtual Environment

Set up and activate a virtual environment:

```
uv venv
source bin/activate  # On macOS/Linux
```

Step 3: Install Dependencies

Add the required dependencies:

```
uv add mcp httpx anthropic
```

Step 4: Create Project Files

Create the server and client files:

```
touch server.py
touch client.py  # We'll call it host_client.py to emphasize it's a host app
```

Building the MCP Server

Server Overview

Our personal assistant MCP server will provide four essential file management tools:

1. **Create File** - Create new files with specified content
2. **Read File** - Read contents from existing files
3. **Update File** - Modify existing file contents
4. **List Files** - List all files in a directory

Server Implementation

Create `server.py` with the following structure:

```
"""
A simple personal assistant MCP server with file management capabilities.
Supports creating, reading, updating, and listing files locally.
"""

from pathlib import Path
from mcp import FastMCP
import json

# Initialize MCP instance
mcp = FastMCP("simple-mcp-assistant")

@mcp.tool
def create_file(file_path: str, content: str) -> dict:
    """Create a new file with the specified content."""
    try:
        path = Path(file_path)
        # Create parent directories if they don't exist
        path.parent.mkdir(parents=True, exist_ok=True)
        # Write content to file
        path.write_text(content)
        return {
            "success": True,
            "message": f"File created at {file_path}",
            "file_path": str(path.absolute())
        }
    except Exception as e:
        return {
            "success": False,
            "error": str(e)
        }
```

```python
@mcp.tool
def read_file(file_path: str) -> dict:
    """Read contents from a file."""
    try:
        path = Path(file_path)
        if not path.exists():
            return {
                "success": False,
                "error": f"File not found: {file_path}"
            }
        content = path.read_text()
        return {
            "success": True,
            "content": content,
            "file_path": str(path.absolute())
        }
    except Exception as e:
        return {
            "success": False,
            "error": str(e)
        }

@mcp.tool
def update_file(file_path: str, content: str) -> dict:
    """Update an existing file with new content."""
    try:
        path = Path(file_path)
        if not path.exists():
            return {
                "success": False,
                "error": f"File not found: {file_path}"
            }
        path.write_text(content)
        return {
            "success": True,
            "message": f"File updated: {file_path}",
            "file_path": str(path.absolute())
        }
    except Exception as e:
        return {
            "success": False,
            "error": str(e)
        }

@mcp.tool
def list_files(directory_path: str = ".") -> dict:
    """List all files in a directory."""
```

```
    try:
        path = Path(directory_path)
        if not path.exists():
            return {
                "success": False,
                "error": f"Directory not found: {directory_path}"
            }

        files = []
        for item in path.iterdir():
            files.append({
                "name": item.name,
                "type": "directory" if item.is_dir() else "file",
                "path": str(item.absolute())
            })

        return {
            "success": True,
            "files": files,
            "count": len(files),
            "directory": str(path.absolute())
        }
    except Exception as e:
        return {
            "success": False,
            "error": str(e)
        }

# Run the server with Streamable HTTP transport
if __name__ == "__main__":
    mcp.run(transport="streamable-http")
```

Key Points About the Server

1. **FastMCP Instance**: We create an MCP instance with a descriptive name
2. **Tool Decorators**: Each function is decorated with `@mcp.tool` to expose it as an MCP tool
3. **Error Handling**: Each tool includes try-except blocks for robust error handling
4. **Return Format**: All tools return dictionaries with success status and relevant data
5. **Transport Type**: The server runs with `transport="streamable-http"` for remote access

Building the Client

Client Overview

The client will: - Connect to the MCP server via Streamable HTTP - Provide a chat loop for continuous interaction - Process queries using an AI model (Claude) - Handle tool calls and display results

Client Implementation

Create `client_streamable_http.py`:

```
"""
MCP Client with Streamable HTTP transport.
Connects to an MCP server and provides a chat interface for interaction.
"""

import asyncio
import os
from typing import Optional
import click
from mcp import ClientSession
from mcp.client.session import StreamClient
from mcp.types import TextContent, Tool
import anthropic
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

class MCPClient:
    def __init__(self):
        self.session: Optional[ClientSession] = None
        self.anthropic_client = anthropic.Anthropic(
            api_key=os.getenv("ANTHROPIC_API_KEY")
        )

    async def connect_to_server(self, server_url: str, headers: dict = None):
        """Connect to an MCP server via Streamable HTTP."""
        if headers is None:
            headers = {}

        # Create StreamClient for HTTP connection
        stream_client = StreamClient()

        # Set up connection
        self.session_context = await ClientSession(
            stream_client,
            server_url,
            headers=headers
        ).__aenter__()

        self.session = self.session_context

        # Initialize the connection
        await self.session.initialize()
```

```
        print(f"  Connected to MCP server at {server_url}")

        # List available tools
        tools = await self.session.list_tools()
        print(f"\nAvailable tools ({len(tools)}):")
        for tool in tools:
            print(f"  - {tool.name}: {tool.description}")

    async def process_query(self, query: str):
        """Process a user query using the connected MCP server."""
        if not self.session:
            print("  Not connected to any MCP server")
            return

        # Get available tools
        tools = await self.session.list_tools()

        # Prepare tools for Claude
        tool_schemas = []
        for tool in tools:
            tool_schemas.append({
                "name": tool.name,
                "description": tool.description,
                "input_schema": tool.inputSchema
            })

        # Send query to Claude
        response = self.anthropic_client.messages.create(
            model="claude-3-5-sonnet-20241022",
            max_tokens=1000,
            messages=[{
                "role": "user",
                "content": query
            }],
            tools=tool_schemas
        )

        # Process Claude's response
        for content in response.content:
            if content.type == "text":
                print(f"\nAssistant: {content.text}")
            elif content.type == "tool_use":
                print(f"\n  Using tool: {content.name}")
                print(f"    Parameters: {content.input}")

                # Execute the tool
                result = await self.session.call_tool(
```

```
                    content.name,
                    content.input
                )

                print(f"   Result: {result.content}")

    async def chat_loop(self):
        """Run an interactive chat loop."""
        print("\n Chat with MCP Assistant (type 'quit' to exit)")

        while True:
            try:
                query = input("\nYou: ")
                if query.lower() in ['quit', 'exit', 'q']:
                    break

                await self.process_query(query)

            except KeyboardInterrupt:
                break
            except Exception as e:
                print(f" Error: {e}")

    async def cleanup(self):
        """Clean up resources."""
        if self.session_context:
            await self.session_context.__aexit__(None, None, None)

async def main(port: int):
    """Main function to run the client."""
    client = MCPClient()

    try:
        # Connect to server
        server_url = f"http://localhost:{port}/mcp"
        await client.connect_to_server(server_url)

        # Run chat loop
        await client.chat_loop()

    finally:
        await client.cleanup()
        print("\n Goodbye!")

@click.command()
@click.option('--port', default=8000, help='MCP server port')
def cli(port):
```

```
    """Run the MCP client."""
    asyncio.run(main(port))

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

Key Differences: STDIO vs Streamable HTTP

Feature	STDIO Transport	Streamable HTTP Transport
Connection Method	Local process communication	HTTP-based communication
Server Parameters	Requires command to run server	Requires server URL
Deployment	Local only	Can be deployed remotely
Use Case	Local tools, development	Production, distributed systems
Client Setup	`StdioClient` with server script path	`StreamClient` with server URL

Testing the Implementation

Step 1: Start the Server

In one terminal window:

```
uv run server.py
```

The server will start on port 8000 by default.

Step 2: Test with MCP Inspector

In another terminal:

```
uv run mcp dev server.py
```

Then: 1. Open the browser at the provided localhost URL 2. Change transport type to "streamable-http" 3. Set URL to `http://localhost:8000/mcp` 4. Click "Connect" 5. Test the tools using the inspector interface

Step 3: Run the Client

Create a `.env` file with your Anthropic API key:

```
ANTHROPIC_API_KEY=your_api_key_here
```

Then run the client:

```
uv run client_streamable_http.py
```

Example Interaction

"‘ You: Create a simple Python script in the current folder to print a message about MCP servers

Using tool: create_file Parameters: {‘file_path’: ‘mcp_message.py’, ‘content’: ‘print(“MCP servers enable powerful tool integration!”)’} Result: {‘success’: True, ‘message’: ‘File created at mcp_message.py’, ‘file_path’: ‘/path/to/mcp_message.py’}