Connect an MCP Client to an MCP Server locally built

In this lesson, we explored the exciting world of **MCP clients** and how they communicate with **MCP servers**. Here's a breakdown of what we covered:

Overview

- We learned how to set up an MCP client to connect with a locally built MCP server.
- Our focus was on a simple "hello world" example to understand the basics of the connection process.
- The mission was to create a client that communicates with the server, lists and calls tools, then brings back the results.

Environment Setup

- We started by creating a new workspace folder, which we set up in **VS Code** as `client`.
- Initialization of the virtual environment was done using `uvinit` and `uvvenv`.
- We ensured activation of the environment with commands like `.venv scripts activate` or `source venv scripts activate` for Mac users.
- Essential libraries were added, particularly `MCCLI`.

Server Creation

- The creation of a server file named `weather.py` was discussed.
- It’s a simple server containing one tool, `getWeather`, returning static weather data (hot and dry).

Client Code Creation

- We moved on to create the client file, `client.py`.
- The `MCP` library provides us with features to effortlessly connect to servers.

Key Concepts

- **Asynchronous Tasks**: We used `asyncio` for handling async operations since client-server communications are inherently async.
- **Error Handling**: Implemented using import `traceback`.

Connecting to the Server

- **Server Parameters**: Defined a `server_params` object using standard input-output to provide necessary commands and arguments to run the server.
- Command `UV run weather.py` was used as an example to start the server.

Creating Client Sessions

- We created a new client using the defined server parameters.
- Using `async with` ensured the sessions or connections don’t linger after script execution.
- Important operations within the session:

    - **Initialization** of the server connection.
    - **Listing Tools**: Retrieved tools available on the server using `await session.list tools`.
    - **Calling Tools**: Demonstrated with `session.call tool` to execute tools and get results.

Execution

- Ensured the script runs by wrapping it within `asyncio.run` to maintain async function handling.
- Execution was demonstrated using `uv run` or `python` with `client.py`.

Conclusion and Results

- We saw how creating a session, initializing it, listing, and calling tools worked.
- The process highlighted how **MCP libraries** streamline server interactions.
- The results validated that the client could list, call, execute tools, and provide outputs.

This lesson set the groundwork for more complex integrations in future modules. We achieved a solid understanding of making an MCP client talk to an MCP server. Ready for more advanced features in upcoming deep dives!