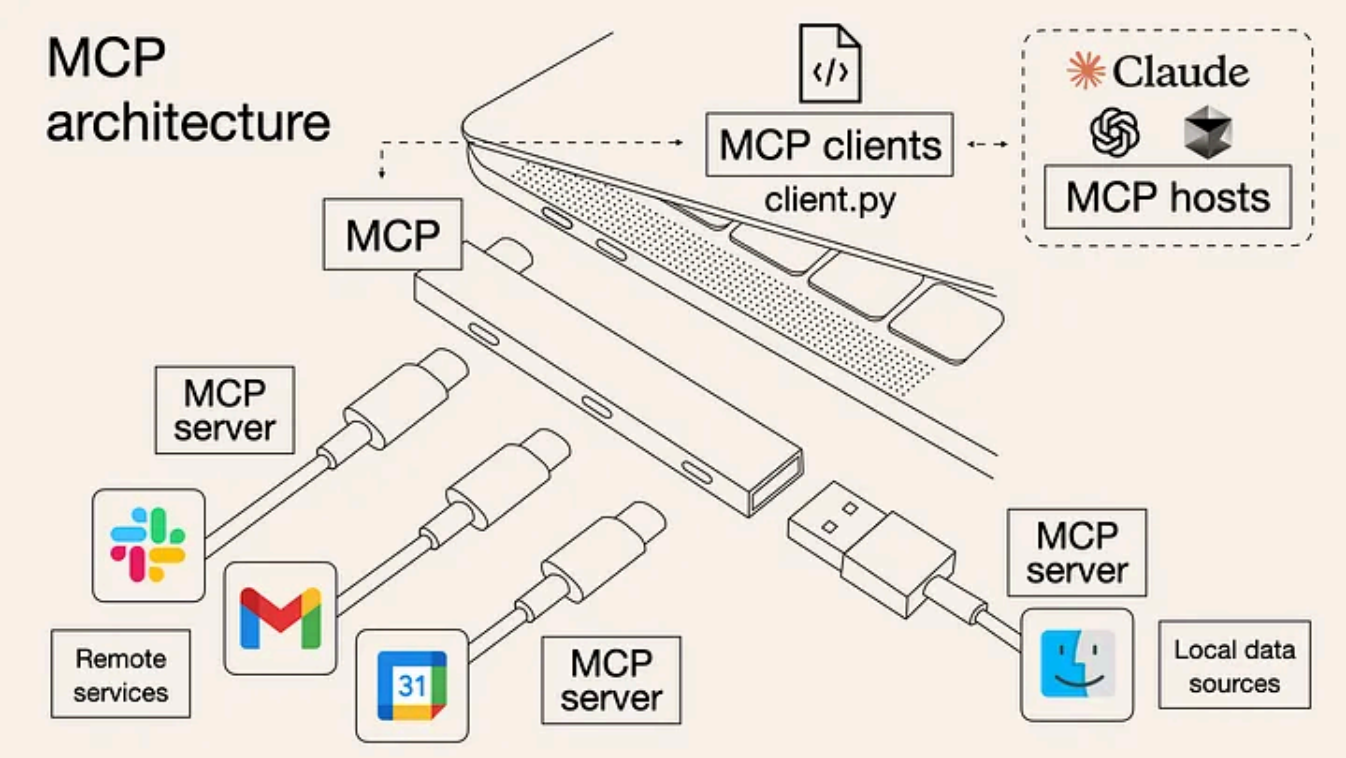
Understanding MCP: Part 1 Laying the Foundation for a Smart To-Do App

9 min read · Jul 4, 2025

Matthew MacFarquhar Follow



Introduction

In this tutorial series, we'll build a classic Todo list app — with a twist. Instead of using traditional UI elements like buttons, text inputs, or filters, all interactions will be handled through a conversational AI assistant. Powered by Model Context Protocol (MCP), this assistant will have access to tools and context, enabling it to manage your tasks entirely through natural language.

What is MCP?

MCP (Model Context Protocol) is a framework that enables AI models to interact with external tools, services, or data through standardized interfaces. It structures these interactions as callable “tools” (or “servers”) that expose capabilities via a schema, making them discoverable and executable by LLMs. This allows developers to build modular, composable systems where models can reason, call tools, and respond intelligently.

All code corresponding to the current state of our project can be found here.

Todo MCP Server

The first step in our journey is setting up the MCP server. We’ll connect it directly to our “database” — a simple local text file. This will allow our AI assistant to retrieve todos, add new ones, and mark existing items as complete.

TodoDB

We will create a plain TS interface to define the structure of our todos.

```
interface Todo {
    title: string;
    description: string;
    completed: boolean;
}
```

We will encapsulate our interactions with the DB txt file behind a class called TodoLocalDB.

```
class TodoLocalDB {
    private filePath: string;

    constructor() {
        this.filePath = 'todo.txt';
        if (!fs.existsSync(this.filePath)) {
            fs.writeFileSync(this.filePath, '[]');
        }
    }

    public addTodo(title: string, description: string) {
        const todos = this._loadTodos();
        todos.push({ title, description, completed: false });
        this._saveTodos(todos);
    }
```

```
    public completeTodo(title: string): boolean {
        const todos = this._loadTodos();
        const todoToCompleteIndex = todos.findIndex(t => t.title === title);
        if (todoToCompleteIndex === -1) {
            return false;
        }

        todos[todoToCompleteIndex].completed = true
        this._saveTodos(todos);
        return true;
    }

    public getTodos(): Todo[] {
        return this._loadTodos();
    }

    private _loadTodos(): Todo[] {
        const fileContent = fs.readFileSync(this.filePath, 'utf8');
        return JSON.parse(fileContent);
    }

    private _saveTodos(todos: Todo[]) {
        fs.writeFileSync(this.filePath, JSON.stringify(todos, null, 2));
    }
}
```

This class will handle creating or opening the `todo.txt` file during initialization. It will expose public functions to add todos, mark them as complete, and retrieve all existing todos. Each of these functions maps directly to a corresponding tool exposed by our MCP server.

Server

Now we can set up our Todo MCP server. A lot of the boilerplate is handled by the npm library @modelcontextprotocol/sdk, making our setup quite simple.

```
export const server = new McpServer({
    name: "todo-server",
    version: "1.0.0",
    capabilities: {
        tools: {},
    },
});

server.tool(
```

```
    "add-todo",
    "Adds a todo to the user's todo list",
    {
        title: z.string().describe("The title of the todo"),
        description: z.string().describe("The description of the todo"),
    },
    async ({ title, description }) => {
        const db = new TodoLocalDB();
        db.addTodo(title, description);

        return {
            content: [{ type: "text", text: `Todo ${title} added` }],
        };
    }
)

server.tool(
    "complete-todo",
    "Completes the first todo in the todo list which matches the given title",
    {
        title: z.string().describe("The title of the todo"),
    },
    async ({ title }) => {
        const db = new TodoLocalDB();
        const success = db.completeTodo(title);

        if (success) {
            return {
                content: [{ type: "text", text: `Todo ${title} completed` }],
            };
        } else {
            return {
                content: [{ type: "text", text: `No Todo with title ${title} ex
            };
        }
    }
)

server.tool(
    "get-todos",
    "Gets the user's todo list",
    {},
    async () => {
        const db = new TodoLocalDB();
        const todos = db.getTodos();

        return {
            content: [{type: "text", text: JSON.stringify(todos)}]
        }
    }
)
```

The basic pattern for creating an MCP server involves initializing it with the capabilities you want to provide, then registering tools. It's important to write clear, descriptive definitions for each tool and its parameters — this helps the LLM decide which tool to call and what inputs to pass when handling user requests.

Each tool also includes a handler function that performs the associated task. After execution, the tool should return a message containing either the retrieved information or a confirmation of whether the action was successful. As mentioned earlier, our three tools align directly with the three public functions exposed by the `TodoLocalDB` class.

Endpoint Setup

The final step to make our MCP server usable is to expose it through an actual endpoint. MCP provides SDKs for two server protocols — over http and over stdio. Most tutorials use the stdio protocol, which is designed for locally running MCP servers communicating over the process's standard input and output. Since we want our server accessible over the network, we'll use the `StreamableHTTPServerTransport` protocol instead.

The first thing we will do is just set up a basic express server

```
const app = express();
app.use(express.json());
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS")
  res.header("Access-Control-Allow-Headers", "*");
  next();
});

// ROUTES HERE

app.listen(5000, async() => {
  console.log("Server is running on port 5000");
});
```

We will need three routes for GET, POST and DELETE on /mcp. For the time being, we will not support session memory or server side notifications, so our GET and DELETE endpoints are pretty boring.

```
// SSE notifications not supported in stateless mode
app.get('/mcp', async (req, res) => {
  console.log('Received GET MCP request');
  res.writeHead(405).end(JSON.stringify({
    jsonrpc: "2.0",
    error: {
      code: -32000,
      message: "Method not allowed."
    },
    id: null
  }));
});

// Session termination not needed in stateless mode
app.delete('/mcp', async (req, res) => {
  console.log('Received DELETE MCP request');
  res.writeHead(405).end(JSON.stringify({
    jsonrpc: "2.0",
    error: {
      code: -32000,
      message: "Method not allowed."
    },
    id: null
  }));
});
```

The POST endpoint will create the transport and connect it to our `TodoMCPServer`. From there, we simply delegate incoming requests to the transport object, which handles invoking the corresponding MCP tools.

```
app.post('/mcp', async (req, res) => {
    console.log("MCP request received");

    try {
      const transport: StreamableHTTPServerTransport = new StreamableHTTPServer
        sessionIdGenerator: undefined,
      });
      res.on('close', () => {
        console.log('Request closed');
        transport.close();
```

```
        server.close();
      });
      await server.connect(transport);
      await transport.handleRequest(req, res, req.body);
    } catch (error) {
      console.error('Error handling MCP request:', error);
      if (!res.headersSent) {
        res.status(500).json({
          jsonrpc: '2.0',
          error: {
            code: -32603,
            message: 'Internal server error',
          },
          id: null,
        });
      }
    }
  });
```

At this point we have a fully functioning MCP server that can communicate to clients over HTTP.

Todo MCP Client

Next, we need to create a client that will act as a bridge between the LLM “brain” and the tools provided by our MCP servers.

OpenAI Host

First, let’s set up the OpenAI integrations. We will need an endpoint to hit OpenAI where we can safely expose our api key. I am using Next js for this project and will host the OpenAI endpoint on the server side APIs in /api/llm.

```
import { OpenAI } from "openai";
const openai = new OpenAI();

export async function POST(request: Request) {
    const body = await request.json();
    const response = await openai.chat.completions.create({
        model: "gpt-4.1-nano",
        messages: body.messages,
        tools: body.tools,
    });
    return new Response(JSON.stringify({ message: response.choices[0].message.c
      headers: { 'Content-Type': 'application/json' },
```

```
    });
}
```

This route will accept a request containing `messages` and `tools`, call the OpenAI model of our choice, and return the resulting message to the caller.

We'll also create an `OpenAIHost` class to handle making this request. It's a simple wrapper but useful for keeping our MCP client clean, so we don't have fetch calls scattered throughout the code.

```
export class OpenAIHost {
    public async sendMessage(messages: any[], tools: any[]): Promise<any> {
        const response = await fetch("/api/llm", {
            method: "POST",
            body: JSON.stringify({messages, tools}),
        });
        const data = await response.json();
        return data;
    }
}
```

Through some trial and error, I discovered that the tools in the MCP protocol don't exactly match the format OpenAI expects. To address this, I created a helper function that converts them into the proper format for OpenAI's use.

```
export interface OpenAIMCPTool {
    type: 'function';
    function: {
        name: string;
        description: string;
        parameters: {
            type: 'object',
            properties: {
                [key: string]: {
                    type: string;
                    description: string;
                }
            },
            required: string[];
        }
```

```
    }
}

export const createOpenAIMCPTool = (name: string, description: string, inputSch
  return {
      type: 'function',
      function: {
          name,
          description,
          parameters: inputSchema
      }
  };
}
```

Client

Now we're ready to build our final piece: the MCP client. This class will receive user requests, enrich them with tools from our connected MCP servers, forward the requests to the LLM host, and manage any follow-up tool calls initiated by the LLM.

First, we will take a look at the initialization and server connection process. Our client will have state for the llm we will call, an MCP Client class, our running list of messages and the tools we have access to.

```
export class MCPClient {
    private mcp: Client;
    private llm: OpenAIHost;
    private messages: any[];
    private tools: OpenAIMCPTool[];

    constructor() {
      this.llm = new OpenAIHost();
      this.mcp = new Client({ name: "mcp-client-cli", version: "1.0.0" });
      this.messages = [];
      this.tools = [];
    }

    public async connectRemoteServer(name: string, url: string) {
        console.log("Adding remote server", name, url);
        const baseUrl = new URL(url);

        const transport = new StreamableHTTPClientTransport(baseUrl);
        await this.mcp.connect(transport);
        console.log("Connected to remote server streamable", name);

        let tools: OpenAIMCPTool[] = [];
```

```
        try {
          const toolsRes = await this.mcp.listTools();
          console.log("Available tools", toolsRes.tools);
          tools = toolsRes.tools.map((tool) => {
              return createOpenAIMCPTool(tool.name, tool.description || `Tool:
          });
          this.tools = tools;
        } catch(error: any) {
          console.error(`Error listing tools for server ${name}:`, error)
        }
    }

  // MORE STUFF BELOW
}
```

When we connect to our MCP server, we create a `StreamableHTTPClientTransport`—the counterpart to the `StreamableHTTPServerTransport` used on the server side. After establishing the connection, we retrieve the tools exposed by the MCP server and map them into our `OpenAIMCP` tool format, which OpenAI can understand and use.

To handle making a message call, we'll need two functions: one to take a user query and call the LLM with our tools, and another to process the LLM's response, perform any necessary tool calls, and recurse until we receive a final message to return to the user.

```
public async queryLLMWithTools(userQuery: string) {
      this.messages.push({role: "user", content: userQuery})
      const response = await this.llm.sendMessage(this.messages, this.tools);
      return response;
  }

public async processLLMResponse(llmResponse: any): Promise<any> {
  console.log("Processing LLM response", llmResponse);
  if (llmResponse.tool_calls && llmResponse.tool_calls.length > 0) {
    for (const toolCall of llmResponse.tool_calls) {
        const {name, arguments: args} = toolCall.function;
        const id = toolCall.id;

        try {
            const toolResult = await this.mcp.callTool({name: name, arguments:

            this.messages.push({
                role: 'assistant',
                content: "",
```

```
                tool_calls: [{id, function: {name, arguments: JSON.stringify(ar
            });

            this.messages.push({
                role: "tool",
                tool_call_id: id,
                content: toolResult.isError ? `Error: ${(toolResult.content as
            })
        } catch(error: any) {
            console.error(`Error calling tool ${name}:`, error)

            this.messages.push({
                role: "tool",
                content: `Error executing tool ${name}: ${error.message}`
            })
        }

    }

    // Recursively process the LLM response if there are more tool calls
    const newLLMResponse = await this.llm.sendMessage(this.messages, this.tools
    return this.processLLMResponse(newLLMResponse);
  }

  this.messages.push({role: "assistant", content: llmResponse.message});
  return llmResponse;
}
```

In processLLMResponse, if we have no tool calls, we simply add the response message to our messages list and return the response to the caller.

If tool calls are required, we use our MCP client to execute them and add the results to our message list. We then recursively call the LLM and process its response, repeating this cycle until the LLM decides to stop and returns a final message. Some applications limit the number of recursive calls to prevent infinite loops.

Front End

We've now completed the full MCP client and MCP server interface. The final step is to integrate our client into the app.

On init, we create an instance of our MCPClient and then connect it to our remotely run todo-server.

```tsx
export default function Home() {
  const mcpClientRef = useRef<MCPClient | null>(null);
  const [messages, setMessages] = useState<any[]>([]);
  const [input, setInput] = useState("");

  useEffect(() => {
    mcpClientRef.current = new MCPClient();
    mcpClientRef.current.connectRemoteServer("todo-server", "http://localhost:5
  }, []);

  const callAI = async (userMessage: string) => {
    const response = await mcpClientRef.current!.queryLLMWithTools(userMessage)
    const processedResponse = await mcpClientRef.current!.processLLMResponse(re
    return processedResponse.message;
  }

  const handleSendMessage = async (e: any) => {
    e.preventDefault();
    if (!mcpClientRef.current) return;
    const response = await callAI(input);
    console.log("RESPONSE", response);
    setMessages([...messages, { role: "user", content: input }, { role: "assist
    setInput("");
  };

  return (
    <div className="flex flex-col h-screen">
      <div className="flex-1 overflow-y-auto p-4">
        {messages.map((message, index) => (
          <div key={index} className="mb-4">
            <div className="font-bold">{message.role}</div>
            <div>{message.content}</div>
          </div>
        ))}
      </div>
      <div className="border-t p-4">
        <form className="flex gap-2">
          <input
            type="text"
            value={input}
            onChange={(e) => setInput(e.target.value)}
            placeholder="Type your message..."
            className="flex-1 rounded border p-2"
          />
          <button
            onClick={(e: any) => handleSendMessage(e)}
            type="submit"
            className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-6
          >
            Send
          </button>
        </form>
```

```
        </div>
      </div>
    );
  }
```

Our main interaction happens in the `callAI` function. This function takes a `userMessage`, calls `queryLLMWithTools`, and then passes the result to `processLLMResponse` to produce a final response after all tool calls are completed.

When we run the server and our Next.js app, we'll see a chat interface like this — allowing us to interact with our AI assistant, which can invoke tools to update our `todo.txt` file seamlessly.

Conclusion

In this article, we successfully built an MCP server that handles HTTP traffic and uses a `todo.txt` file as its database. We then created an MCP client to connect to the server and expose its tools to our OpenAI integration. Finally, we developed a simple chat interface that uses the MCP client to send user messages through the tool-augmented LLM, enabling informed actions and live, data-driven responses.

Mcp Server | Nextjs | Typescript | Programming | AI

Follow

Written by Matthew MacFarquhar

326 followers · 144 following

I am a software engineer working for Amazon living in SF/NYC.

No responses yet

Bgerby

What are your thoughts?

More from Matthew MacFarquhar

Matthew MacFarquhar

Hello Docker! Creating a super simple local Docker Image

In this article I will show you how to create and launch a Docker Image on your local machine in under 10 minutes. I will explain the...

Dec 12, 2022 103 1

Matthew MacFarquhar

Mastering MapReduce: A Step-by-Step Java Tutorial for Big Data Processing

Introduction

Jul 8, 2023 61

Matthew MacFarquhar

Learning Svelte: Part 1 Basic State

Introduction

Aug 26 13

Matthew MacFarquhar

Learning Svelte: Part 3 Working with Shared State

Introduction

Sep 4 13

See all from Matthew MacFarquhar

Recommended from Medium

Aparna Prasad

🚀 Build Your First MCP Server in TypeScript — Step-by-Step Guide (with Code) 💻

What is MCP (Modal Context Protocol)?

Jul 30 1

Dmitrii Pashkevich

TanStack DB Query: Step-by-step guide to combining parameter-based loading and normalized storage

In this guide, I will demonstrate how to use TanStack Query v5 and TanStack DB to achieve a seamless user interface with incremental...

Sep 27

Yanxing Yang

Getting Started with uv: A Modern Python Environment and Package Manager

Getting Started with uv

May 30 11

In Python in Plain English by Rohan Mistry

🧩 Dependency Injection in Python: A Complete Guide to Cleaner, Scalable Code

When you hear “Dependency Injection” (DI), your mind might jump to complex enterprise frameworks in Java or C#. But Python, with its...

May 11 121 1

Frontend Highlights

Understanding tRPC: Building Type-Safe APIs in TypeScript

Introduction

May 4 5

Vignesh

🤖 LangGraph vs MCP—You're Asking the Wrong Question

As someone exploring AI development, you've probably come across LangGraph and Model Context Protocol (MCP). And maybe you've wondered:

Jun 14 201 1

See more recommendations