```
/**
 * Connect model with mcp tools in Node.js
 *
 * npm install mcp openai
 * node <this-script-path>.js
 */
const { spawn } = require('child_process');
const { Client } = require('@modelcontextprotocol/sdk/client/index.js');
const { StdioClientTransport } = require('@modelcontextprotocol/sdk/client/stdio.js');
const { SSEClientTransport } = require('@modelcontextprotocol/sdk/client/sse.js');
const { StreamableHTTPClientTransport } = require('@modelcontextprotocol/sdk/client/streamableHttp.js');
const OpenAI = require('openai');

class MCPClient {
    constructor() {
        this.servers = new Map();
        this.toolToServerMap = new Map();
        // To authenticate with the model you will need to generate a github gho token in your GitHub settings.
        // Create your github gho token by following instructions here:
https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens
        this.openai = new OpenAI({
            baseURL: "https://models.github.ai/inference",
            apiKey: process.env.GITHUB_TOKEN,
            defaultQuery: {
                "api-version": "2024-08-01-preview"
            }
        });
    }

    async connectStdioServer(serverId, command, args, env = {}) {
        const transport = new StdioClientTransport({
            command: command,
            args: args,
            env: { ...process.env, ...env }
        });

        const client = new Client({
            name: "openai-client",
            version: "1.0.0"
        }, {
            capabilities: {}
        });

        await client.connect(transport);
        await this.registerServer(serverId, client);
    }

    async connectSseServer(serverId, url, headers = {}) {
        const transport = new SSEClientTransport(new URL(url), { headers });

        const client = new Client({
            name: "openai-client",
            version: "1.0.0"
        }, {
            capabilities: {}
        });

        await client.connect(transport);
        await this.registerServer(serverId, client);
    }

    async connectHttpServer(serverId, url, headers = {}) {
        const transport = new StreamableHTTPClientTransport(new URL(url), { headers });

        const client = new Client({
            name: "openai-client",
            version: "1.0.0"
        }, {
            capabilities: {}
        });

        await client.connect(transport);
        await this.registerServer(serverId, client);
    }

    async registerServer(serverId, client) {
        const response = await client.listTools();
        const tools = response.tools;

        this.servers.set(serverId, {
            client: client,
            tools: tools
        });

        for (const tool of tools) {
            this.toolToServerMap.set(tool.name, serverId);
        }
```

```
    console.log(`¥nConnected to server '${serverId}' with tools:`, tools.map(tool => tool.name));
}

async chatWithTools(messages) {
    if (this.servers.size === 0) {
        throw new Error("No MCP servers connected. Connect to at least one server first.");
    }

    const availableTools = [];
    for (const [serverId, serverInfo] of this.servers) {
        for (const tool of serverInfo.tools) {
            availableTools.push({
                type: "function",
                function: {
                    name: tool.name,
                    description: tool.description,
                    parameters: tool.inputSchema
                }
            });
        }
    }

    while (true) {
        const response = await this.openai.chat.completions.create({
            messages: messages,
            model: "openai/gpt-4.1",
            tools: availableTools,
            response_format: {
                "type": "text"
            },
            temperature: 1,
            top_p: 1,
        });

        const choice = response.choices[0];
        let hasToolCall = false;

        if (choice.message.tool_calls) {
            for (const tool of choice.message.tool_calls) {
                hasToolCall = true;
                const toolName = tool.function.name;
                const toolArgs = JSON.parse(tool.function.arguments);

                messages.push({
                    role: "assistant",
                    tool_calls: [{
                        id: tool.id,
                        type: "function",
                        function: {
                            name: tool.function.name,
                            arguments: tool.function.arguments
                        }
                    }]
                });

                if (this.toolToServerMap.has(toolName)) {
                    const serverId = this.toolToServerMap.get(toolName);
                    const serverClient = this.servers.get(serverId).client;

                    const callResult = await serverClient.callTool({
                        name: toolName,
                        arguments: toolArgs
                    });

                    console.log(`[Server '${serverId}' call tool '${toolName}' with args ${JSON.stringify(toolArgs)}]:
${JSON.stringify(callResult.content)}`);

                    messages.push({
                        role: "tool",
                        tool_call_id: tool.id,
                        content: [
                            {
                                type: "text",
                                text: JSON.stringify(callResult.content)
                            }
                        ]
                    });
                }
            }
        } else {
            messages.push({
                role: "assistant",
                content: choice.message.content
            });
            console.log(`[Model Response]: ${choice.message.content}`);
        }

        if (!hasToolCall) {
```

```
                break;
            }
        }
    }

    async cleanup() {
        for (const [serverId, serverInfo] of this.servers) {
            await serverInfo.client.close();
        }
        await new Promise(resolve => setTimeout(resolve, 1000));
    }
}

async function main() {
    const client = new MCPClient();
    const messages = [
        {
            role: "system",
            content: "Xから話題の観光地を1つ選び、Google Mapsの情報に基づき、食べ歩きをテーマとした日帰り旅行プランを提案してください。¥n¥n以下のステップに従い、論理的な検討や調査結果（理由付け）を行ってから、最終的な旅行プラン（結論）を記載してください。必ず、観光地の選定理由・プラン作成の根拠から始め、結論（旅行プラン）を最後に記載してください。¥n¥n# Steps¥n¥n1. 観光地の候補を検討し、話題性やアクセス性、食べ歩きに適した理由などを調査・記載¥n2. Google Mapsで現地の飲食店やスポットをリサーチし、魅力的な食べ歩きルート案を検討・記載¥n3. 独自の工夫（時間配分、店の順番、立ち寄りスポット等）を考慮し、日帰りで回れるプランにまとめる¥n¥n# Output Format¥n¥n下記の構造で日本語で記載してください。¥n¥n{¥n  ¥"理由・調査結果¥": [観光地選定の理由、Google Mapsでの調査内容や工夫した点などを具体的に記載],¥n  ¥"旅行プラン¥": {¥n    ¥"観光地名¥": [選定した観光地名],¥n  ¥"日帰り食べ歩きプラン¥": [朝・昼・夕に分けた食べ歩きルート案（1日の流れ、店名や場所、各立ち寄りポイント、交通手段）を箇条書きまたは時系列で具体的に記載]¥n  }¥n}¥n¥n# Examples¥n¥n例1:¥n{¥n  ¥"理由・調査結果¥": ¥"都内からアクセスが良く、SNSで話題の『鎌倉』を選定。Google Mapsで食べ歩きスポットを事前調査し、小町通り周辺は多様なグルメ店や観光名所が集まっていたため、食べ歩きに最適と判断。¥",¥n  ¥"旅行プラン¥": {¥n    ¥"観光地名¥": ¥"鎌倉¥",¥n    ¥"日帰り食べ歩きプラン¥": [¥n      ¥"09:30 鎌倉駅到着。小町通りを散策¥",¥n      ¥"10:00 『アイスキャンディー本舗』で朝のスイーツ¥",¥n      ¥"10:30 『若宮大路』でたい焼き＆お団子を食べ歩き¥",¥n      ¥"12:00 『鶴岡八幡宮』参拝¥",¥n      ¥"13:00 『しらす丼の店』で昼食¥",¥n      ¥"14:30 『カフェ鎌倉』で休憩＆スイーツ¥",¥n      ¥"16:00 江ノ電で長谷エリアへ移動、『大仏』見学¥",¥n      ¥"17:00 鎌倉駅発、帰路へ¥"¥n  ]¥n  }¥n}¥n¥n（実際の出力では、理由・プラン部分がさらに詳細になるよう、Google Mapsの情報や具体的な店名、工夫点なども明記してください）¥n¥n# Notes¥n¥n- 必ず観光地選定理由・Google Maps調査結果・プラン作成時の工夫などを「理由・調査結果」欄に記載してください¥n- 「旅行プラン」欄は朝から夕方までの時系列で、店名・スポット名を含めて具体的に記載してください¥n- 出力はJSON形式で、コードブロックは使用しないでください¥n- 結論（旅行プラン）は必ず論理的な検討・根拠（理由・調査結果）の後に記載してください"
        },
    ];

    try {
        await client.connectStdioServer(
            "google-maps",
            "npx",
            [
                "-y",
                "@modelcontextprotocol/server-google-maps",
            ],
            {
                "GOOGLE_MAPS_API_KEY": process.env["GOOGLE_MAPS_API_KEY"],
            }
        );
        await client.connectStdioServer(
            "filesystem",
            "npx",
            [
                "-y",
                "@modelcontextprotocol/server-filesystem",
                "C:¥¥Users¥¥[ユーザ名]¥¥Desktop¥¥mcp-data",
            ],
            {
            }
        );
        await client.connectStdioServer(
            "twitter-mcp",
            "npx",
            [
                "-y",
                "@enescinar/twitter-mcp",
            ],
            {
                "API_KEY": process.env["API_KEY"],
                "API_SECRET_KEY": process.env["API_SECRET_KEY"],
                "ACCESS_TOKEN": process.env["ACCESS_TOKEN"],
                "ACCESS_TOKEN_SECRET": process.env["ACCESS_TOKEN_SECRET"],
            }
        );
        await client.chatWithTools(messages);
    } catch (error) {
        console.error(`¥nError: ${error.message}`);
    } finally {
        await client.cleanup();
    }
}

main().catch(console.error);
```