

Building an MCP Server

Building a real, runnable MCP server and client with the official TypeScript SDK: project setup, defining tools with `registerTool`, connecting over an in-memory transport, and real output.

Intermediate 28 min

Generated on 5 October 2026

Read online: <https://www.learnforgex.com/en/topics/building-an-mcp-server>

Building an MCP Server

Every lesson in this category so far has been conceptual: "Tools and Function Calling" described the loop, "Introduction to MCP" described the roles and primitives, and "MCP Architecture" described what actually travels over the wire. This lesson builds a real MCP server and a real client that talks to it, using the official TypeScript SDK (`@modelcontextprotocol/sdk`) -- the first runnable code in this course's AI category, and everything below was actually written, compiled, and run to confirm it behaves exactly as shown.

This lesson is written so it works even if you've never touched Node.js or TypeScript before. You don't need to know the language in depth -- we'll get the program running first, then walk through what each part of it does.

What We'll Do in This Lesson

1. Check that Node.js is installed on your computer.
2. Create an empty folder for the MCP project.
3. Create `package.json` and `tsconfig.json`.
4. Add the MCP server and client code (`GeoFactsServer.ts` , `RunServerWithClient.ts`) to that same folder.
5. Install the required packages with `npm install`.
6. Compile the TypeScript code to JavaScript.
7. Run the program with Node.js.
8. Watch the client discover and call the server's tools, using real output.

Prerequisites: Node.js and npm

The TypeScript SDK runs on **Node.js** -- a program that runs JavaScript (and, once compiled, TypeScript) outside a browser, playing a role similar to the JVM's for Java (not a one-to-one technical equivalent -- the JVM is a bytecode virtual machine, while Node.js is a JavaScript runtime built on V8, but the "platform that runs code outside a browser" analogy serves its purpose here). Node.js comes bundled with **npm** (Node Package Manager), the tool used, for a similar purpose to Maven in a Java project, to download library code -- like the MCP SDK itself -- into a project.

Open a terminal and check whether Node.js is already installed:

```
node --version
```

If you see a version number like `v22.14.0` (Node.js 18 or newer is needed for this lesson), you're set. Now check npm:

```
npm --version
```

npm ships with Node.js automatically, so if the first command worked, this one almost certainly will too.

If either command says something like `command not found`, install Node.js's LTS (long-term support) release first -- npm comes with it. Close and reopen your terminal after installing before continuing.

Create the Project Folder

Everything in this lesson happens inside one folder. Create one named `mcp-geo-facts-demo`.

On macOS or Linux:

```
mkdir mcp-geo-facts-demo  
cd mcp-geo-facts-demo
```

On Windows, creating the folder in File Explorer and opening a PowerShell window inside it works just as well.

Run every terminal command in the rest of this lesson from INSIDE this folder.

Project File Layout

To start, your project folder will contain exactly these four files:

```
mcp-geo-facts-demo/  
├─ package.json  
├─ tsconfig.json  
├─ GeoFactsServer.ts  
└─ RunServerWithClient.ts
```

After the compile step (in "Install, Compile, and Run" below), a `dist` folder appears too:

```
mcp-geo-facts-demo/  
├─ package.json  
├─ tsconfig.json  
├─ GeoFactsServer.ts  
├─ RunServerWithClient.ts  
└─ dist/  
    ├─ GeoFactsServer.js  
    └─ RunServerWithClient.js
```

You don't need to create `node_modules` (where dependencies get downloaded) yourself -- `npm install` does that automatically.

Create package.json

Create a file named `package.json` in the project folder with exactly this content:

```
{  
  "name": "mcp-geo-facts-demo",  
  "version": "1.0.0",  
  "type": "module",  
  "dependencies": {  
    "@modelcontextprotocol/sdk": "^1.30.0",  
    "zod": "^4.4.3"  
  },  
  "devDependencies": {  
    "typescript": "^5.7.0"  
  }  
}
```

`package.json` lists what the project depends on, similar to the `<dependencies>` block in a Java project's `pom.xml`:

- `@modelcontextprotocol/sdk` -- the official SDK for building MCP servers and clients.
- `zod` -- for describing a tool's parameter schema.
- `typescript` -- to compile the code.

Don't remove the `"type": "module"` line -- without it, Node treats compiled output as CommonJS, and the top-level `await` used throughout `RunServerWithClient.ts` fails to run.

Create tsconfig.json

In the same folder, next to `package.json`, create a second file named `tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "NodeNext",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "outDir": "dist"
  },
  "include": ["*.ts"]
}
```

This tells the TypeScript compiler how to turn the `.ts` files in this folder into runnable JavaScript (into an `outDir` named `dist`). The `"module": "NodeNext"` and `"moduleResolution": "NodeNext"` settings specifically matter because the SDK ships as native ESM: with them, imports from the SDK need an explicit `.js` extension even though the source files are `.ts` -- you'll see this in `GeoFactsServer.ts` below. It looks like a typo the first time you see it, but it's required.

Define the Server: GeoFactsServer.ts

Create a file named `GeoFactsServer.ts` in the project folder with the code below. A server is built with `McpServer` and exposes tools with `registerTool()`, giving each one exactly the three parts "Defining a Tool: Name, Description, and Schema" described -- a name, a description, and a `zod`-based parameter schema:

```

// A small, self-contained MCP server exposing two tools: one that looks up
// information the model would not otherwise know (get_capital_city), and one
// that performs a computation (calculate_sum). Kept in its own file/function
// so it can be reused both by a real stdio-connected server process and,
// as in RunServerWithClient.ts, by an in-process demo client.
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { z } from "zod";

const CAPITALS: Record<string, string> = {
  France: "Paris",
  Japan: "Tokyo",
  Turkey: "Ankara",
};

export function createGeoFactsServer(): McpServer {
  const server = new McpServer({ name: "geo-facts-server", version: "1.0.0" });

  server.registerTool(
    "get_capital_city",
    {
      title: "Get Capital City",
      description: "Looks up the capital city of a given country.",
      inputSchema: { country: z.string().describe("Country name, e.g. 'France'") },
    },
    async ({ country }) => {
      const capital = CAPITALS[country];
      if (!capital) {
        return {
          content: [{ type: "text", text: `No capital known for "${country}"` }],
          isError: true,
        };
      }
      return { content: [{ type: "text", text: `The capital of ${country} is ${capital}.` } ] };
    }
  );

  server.registerTool(
    "calculate_sum",
    {
      title: "Calculate Sum",
      description: "Adds a list of numbers together.",
      inputSchema: { numbers: z.array(z.number()).describe("Numbers to add") },
    },
    async ({ numbers }) => {
      const total = numbers.reduce((acc, n) => acc + n, 0);
      return { content: [{ type: "text", text: `Sum: ${total}` } ] };
    }
  );

  return server;
}

```

This server offers two tools:

- `get_capital_city` -- takes a country name and returns its capital (for example, `Japan` -> `Tokyo`). It demonstrates a tool supplying information the model wouldn't otherwise have, and returns `isError: true` when it doesn't know the answer, rather than guessing.
- `calculate_sum` -- takes a list of numbers and returns their sum (for example, `[4, 8, 15, 16, 23, 42]` -> `108`). It demonstrates a tool performing exact computation the model shouldn't attempt on its own.

Both directly match the motivation from "Why Does It Exist?" in "Tools and Function Calling". Wrapping the setup in a `createGeoFactsServer()` function (rather than running it as top-level code) is what lets the same server be reused by a real stdio-connected process or, as below, by an in-process demo client.

Connect a Client: `RunServerWithClient.ts`

Create a second file, `RunServerWithClient.ts`, in the same project folder, next to `GeoFactsServer.ts`. To actually exercise the server, this lesson connects a real `Client` using the **in-memory transport** introduced in "MCP Architecture":

```
Client -- InMemoryTransport -- MCP Server
```

Both sides run in this same file, inside the same Node.js process -- avoiding any need for a separate process or network connection, which keeps this first example as simple as possible. It still goes through the exact same initialize/discover/invoke lifecycle and JSON-RPC messages as a production stdio or Streamable HTTP connection:

```
// Demonstrates a real MCP client talking to the server from
// GeoFactsServer.ts over an in-memory transport (no network/stdio needed --
// ideal for a runnable, self-contained example). A real host application
// would instead connect over stdio or HTTP to a separately running server
// process, but the request/response behavior shown here -- discovering
// tools, then calling them -- is identical either way.
import { Client } from "@modelcontextprotocol/sdk/client/index.js";
import { InMemoryTransport } from "@modelcontextprotocol/sdk/inMemory.js";
import { createGeoFactsServer } from "./GeoFactsServer.js";

type ToolTextContent = Array<{ type: string; text: string }>;

async function main() {
  const server = createGeoFactsServer();
  const [clientTransport, serverTransport] = InMemoryTransport.createLinkedPair();
  const client = new Client({ name: "demo-client", version: "1.0.0" });

  await Promise.all([
    server.connect(serverTransport),
    client.connect(clientTransport),
  ]);

  const toolList = await client.listTools();
  console.log("Tools discovered by client:", toolList.tools.map((t) => t.name));
  // Tools discovered by client: [ 'get_capital_city', 'calculate_sum' ]
```

```

const capitalResult = await client.callTool({
  name: "get_capital_city",
  arguments: { country: "Japan" },
});
console.log("get_capital_city(Japan) ->", (capitalResult.content as ToolTextContent)[0].text);
// get_capital_city(Japan) -> The capital of Japan is Tokyo.

const sumResult = await client.callTool({
  name: "calculate_sum",
  arguments: { numbers: [4, 8, 15, 16, 23, 42] },
});
console.log("calculate_sum([4,8,15,16,23,42]) ->", (sumResult.content as ToolTextContent)[0].text);
// calculate_sum([4,8,15,16,23,42]) -> Sum: 108

const missingResult = await client.callTool({
  name: "get_capital_city",
  arguments: { country: "Wakanda" },
});
console.log(
  "get_capital_city(Wakanda) -> isError:",
  missingResult.isError,
  "text:",
  (missingResult.content as ToolTextContent)[0].text
);
// get_capital_city(Wakanda) -> isError: true text: No capital known for "Wakanda".

await client.close();
await server.close();
}

main();

```

`client.listTools()` performs the `tools/list` discovery step from "The Connection Lifecycle: Initialize, Discover, Invoke"; each `callTool()` performs a `tools/call` invocation, exactly as shown on the wire in "From Concepts to Wire Format: JSON-RPC 2.0". The final call deliberately requests a country that isn't in `CAPITALS` (`Wakanda`), to demonstrate the `isError: true` path from `GeoFactsServer.ts` actually reaching the client -- "Why the 'Wakanda' Example Exists" below explains that choice.

Install, Compile, and Run

With all four files (`package.json`, `tsconfig.json`, `GeoFactsServer.ts`, `RunServerWithClient.ts`) saved in the same project folder, run these three commands from a terminal **inside that folder**, in order:

1. `npm install` -- reads `package.json` and downloads the SDK, `zod`, and `typescript` into a new `node_modules` folder. This needs an internet connection and also creates a `package-lock.json` file. It only needs to run once (or again if you change the dependencies).

```
npm install
```

2. `npx tsc -p tsconfig.json` -- compiles both `.ts` files into plain JavaScript, following `tsconfig.json`'s settings. No output means it succeeded; you'll see a new `dist` folder afterward containing `GeoFactsServer.js` and `RunServerWithClient.js`.

```
npx tsc -p tsconfig.json
```

3. `node dist/RunServerWithClient.js` -- actually runs the compiled demo.

```
node dist/RunServerWithClient.js
```

That third command produces this, exactly:

```
Tools discovered by client: [ 'get_capital_city', 'calculate_sum' ]
get_capital_city(Japan) -> The capital of Japan is Tokyo.
calculate_sum([4,8,15,16,23,42]) -> Sum: 108
get_capital_city(Wakanda) -> isError: true text: No capital known for "Wakanda".
```

At this point you've built and run your first working MCP demo.

What Actually Happened

First, the server was created (`createGeoFactsServer()`), with its two tools: `get_capital_city` and `calculate_sum`. Then the client was created, and client and server were connected over an in-memory transport.

Next, `client.listTools()` was called -- this is MCP's **tool discovery** step. The client is asking the server "what tools do you have?", and the server answers with `get_capital_city` / `calculate_sum`. The first line of the output is real proof of this: the client never had those two names hardcoded anywhere -- it learned both from the server, over the protocol.

After that, the client calls tools with `client.callTool(...)` -- for example, the `get_capital_city("Japan")` call travels to the server, and the server sends the result back. The overall flow looks like this:

```

Client
  | tools/list
  v
MCP Server
  | tool list
  v
Client
  | tools/call
  v
MCP Server
  | result
  v
Client

```

This is a working example of the initialize/discover/invoke lifecycle from "MCP Architecture".

Why the 'Wakanda' Example Exists

The final call deliberately sends `Wakanda`. Since `Wakanda` isn't in `CAPITALS`, the server returns `isError: true` -- instead of crashing, or silently making up a wrong answer. This shows how an expected tool failure gets communicated back to the client cleanly; it's a direct application of the idea, from "Tools and Function Calling", that the tool-calling loop should hand the model something concrete to work with even when a tool fails, not an opaque crash.

Troubleshooting

- **node: command not found or npm: command not found**. Node.js isn't installed (or isn't on your terminal's PATH) -- recheck with the commands from "Prerequisites: Node.js and npm" before continuing.
- **An error mentioning Cannot find module '@modelcontextprotocol/sdk' (or zod)**. `npm install` either wasn't run, or was run in a different folder than the one containing `package.json` -- rerun it from inside the project folder.
- **An error like Cannot find module '../mcp'**. Don't remove the `.js` extension from SDK imports -- it should be `"@modelcontextprotocol/sdk/server/mcp.js"`, not `"../mcp"` or `"../mcp.ts"`. NodeNext module resolution requires that `.js` extension even from `.ts` source files.
- **Cannot use import statement outside a module**. Make sure `package.json` still has `"type": "module"` in it.
- **Error: Cannot find module '../dist/RunServerWithClient.js'**. The compile step (`npx tsc -p tsconfig.json`) either wasn't run, or exited with an error that needs fixing first -- scroll up in the terminal to see what it reported.

- **A TypeScript error pointing at a specific line.** Compare that line against "Define the Server: GeoFactsServer.ts" or "Connect a Client: RunServerWithClient.ts" character by character -- this is real code, not a fill-in-the-blank template, so a missing comma or bracket while typing it out is the most common cause.

From This Demo to a Real Deployment

Nothing about `GeoFactsServer.ts` is specific to the in-memory transport -- that's the point of keeping the server definition in its own function, separate from `RunServerWithClient.ts`'s demo wiring. In a real MCP setup, the server usually runs as its own process -- for example, a chat application or an IDE launches it as a subprocess and talks to it over

`StdioServerTransport` :

```
Host Application -- stdio -- MCP Server
```

To run this same server for a real host application instead of a demo client, only the transport changes: connecting `server` to a `StdioServerTransport` and running the file as its own process, exactly as "Transports: stdio and Streamable HTTP" described. The tool definitions (`get_capital_city` , `calculate_sum`), their schemas, and their behavior stay completely unchanged -- only which transport carries the same JSON-RPC messages is different.

Best Practices

- Keep server definition and transport wiring in separate files/functions, as `GeoFactsServer.ts` and `RunServerWithClient.ts` do here -- it's what makes a server usable both in a quick demo and a real stdio deployment without duplicating tool logic (see "From This Demo to a Real Deployment").
- Write tool descriptions with the same care "Tools and Function Calling" recommended -- the SDK enforces the parameter *schema*, but nothing stops a vague *description* from causing a model to pick the wrong tool.
- Returning `isError: true` with a clear message for expected, application-level failure cases (like `get_capital_city`'s unknown-country path), rather than throwing, is a good pattern -- it gives the model something concrete to work with instead of an opaque failure. (This isn't a universal rule every tool failure must follow -- which errors get handled this way depends on the tool's and application's own design.)

Common Mistakes

- **Forgetting "type": "module" in package.json** . As "Create package.json" showed, without it, top-level `await` (used throughout `RunServerWithClient.ts`) fails to compile under Node's default CommonJS treatment.

- **Importing SDK modules without the .js extension.** "Create tsconfig.json" explained that `NodeNext` module resolution requires it, even from `.ts` source files -- omitting it produces a module-not-found error at compile time.
- **Treating the in-memory transport as production-ready.** As "From This Demo to a Real Deployment" explained, it's a deliberate simplification for a self-contained lesson -- a real deployment connects the same server to `StdioServerTransport` or an HTTP-based transport instead.

Summary, Cheat Sheet, and Glossary

Summary

- This lesson's project is an ordinary Node.js/TypeScript project (npm plays a role similar to Maven's for dependency management, not a one-to-one technical equivalent) -- all four files (`package.json` , `tsconfig.json` , `GeoFactsServer.ts` , `RunServerWithClient.ts`) live in one folder.
- The TypeScript SDK (`@modelcontextprotocol/sdk`) needs `"type": "module"` in `package.json` and `NodeNext` module resolution in `tsconfig.json` to work correctly with its native ESM build.
- `McpServer` and `registerTool()` implement exactly the name + description + parameter schema pattern from "Tools and Function Calling", with `zod` describing each schema.
- A real `Client` , connected over an **in-memory transport**, performs genuine `tools/list` discovery and `tools/call` invocations against the server -- confirmed here by real, observed output, not a simulation.
- Separating server definition (`GeoFactsServer.ts`) from transport wiring (`RunServerWithClient.ts`) is what lets the exact same server logic run in-memory for a demo or over stdio for a real deployment.
- Returning `isError: true` (as in the `Wakanda` case) instead of crashing gives a model something concrete to work with when a tool fails.

Cheat Sheet

- Setup: one folder with `package.json` , `tsconfig.json` , `GeoFactsServer.ts` , `RunServerWithClient.ts` .
- `"type": "module"` in `package.json` + `module / moduleResolution: "NodeNext"` in `tsconfig.json`.
- SDK imports need explicit `.js` extensions, even in `.ts` files.
- Run order: `npm install -> npx tsc -p tsconfig.json -> node dist/RunServerWithClient.js` .
- Server: `new McpServer({...}) + server.registerTool(name, {title, description, inputSchema}, handler)` .
- Client: `new Client({...}) , client.listTools() , client.callTool({ name, arguments })` .

- Demo transport: `InMemoryTransport.createLinkedPair()` . Real deployment: `StdioServerTransport` (or an HTTP-based transport).

Glossary

- **Node.js:** a program that runs JavaScript/TypeScript (once compiled) outside a browser -- the runtime this lesson's server and client run on.
- **npm:** Node.js's bundled package manager, used here to download the SDK, `zod` , and `typescript` into the project.
- **package.json:** the file describing a Node.js project's identity and dependencies.
- **node_modules:** the folder where npm downloads a project's dependencies.
- **TypeScript compiler (`tsc`):** the tool that turns TypeScript code into JavaScript.
- **McpServer:** the TypeScript SDK class used to define and register a server's tools.
- **registerTool():** the current `McpServer` API for exposing a tool, taking its name, a config (title, description, input schema), and a handler function.
- **zod:** a TypeScript schema-validation library used here to describe a tool's expected parameters.
- **InMemoryTransport:** an SDK transport connecting a client and server within the same process, used in this lesson to keep the example self-contained.
- **StdioServerTransport:** the SDK transport used in a real deployment, connecting a server to a host application over standard input/output.