Hand-on Introduction to MCP

Workshop plan:

* **Total Duration**: 1 hour
* **Audience**: Backend developers with technical experience
* **Format**: Hybrid — theoretical walkthrough followed by live coding demos
* **Size**: ~20 participants with laptops

## **Introduction:**

In this first part of the workshop, we’ll introduce the structure and goals of today’s session.

The aim is to give you a clear understanding of **how the workshop will unfold**, what we’ll build, and how you can follow along step by step.

### **Workshop Objective**

In this workshop, you will:

* Understand the **structure of a tool server** that follows the MCP format
* Learn how to **receive a tool request** and return a valid response
* Build and run a **minimal MCP server** in one of three languages: **Python**, **TypeScript**, or **Go**
* Be able to **test the interaction locally** using a simple curl command or HTTP client

This is a practical, hands-on session — the goal is not to cover everything about MCP but to give you the minimum you need to start working with it in your own systems.

### **What We Will Build**

We will implement a very simple tool server:

A tool called **addition** takes two number as input and returns the result as output.

This tool will:

* Accept a tool\_use request
* Return a valid tool\_response
* Be implemented in a fully working MCP-compatible server

You’ll see this implemented live in:

* **Python** using mcp-server
* **TypeScript** using @microsoft/MCP
* **Go** using mcp-go

All code and examples will be shared at the end of the session.

### **Prerequisites**

Make sure your machine has at least one of the following:

* Python 3.x + pip
* Node.js + npm
* Go 1.20+

Any basic HTTP client (like curl or Postman) will also help for testing.

## First part

### 1.1 What is MCP (Model Context Protocol)?

**Model Context Protocol**, or **MCP**, is a communication standard introduced by **Anthropic in November 2024**.

It was created in response to a growing challenge in AI development: as language models became more capable — able to reason, plan, and use external tools — there was no standard way to structure those interactions. Developers were building ad hoc systems with custom formats, making it hard to reuse tools, debug problems, or scale reliably.

Anthropic introduced MCP to solve that. The idea was simple: provide a clear and shared way for models, tools, and clients to exchange information — no matter the programming language, infrastructure, or specific AI model being used.

Since its release, MCP has been adopted across the AI ecosystem — from researchers and open-source contributors to companies building real-world applications. It’s now recognized as a foundational layer for building AI systems that need structure, memory, and the ability to take action through external tools.

In short: **Anthropic created MCP to bring order to a messy, fast-growing space — and it’s quickly become the common ground for modern AI tools to talk to each other.**

### 1.2 Tool definition in MCP

Before everything MCP is a definition for the LLM of **which tools are available** and **how to use them**.

In MCP, each tool is described using a standard JSON schema that includes:

* The tool’s name
* A description of what it does
* The parameters it expects (names, types, and descriptions)

These definitions are made available to the model before a conversation begins — either embedded during training, loaded dynamically at runtime (e.g., via system prompts or API metadata), or registered as part of the MCP tool configuration.

For example, you might define a calculator tool with:

* A calculate name
* A description like: “Performs basic math operations”
* A schema with parameters like operation, x, and y

Once the model has this definition, it can decide — on its own — when to call the tool and how to format the input.

This is what makes MCP different from raw prompting or function calling. The model isn’t blindly calling code — it understands what each tool does, when to use it, and how to shape its request.

This division of concern is a key idea in MCP:

* **The model reasons and decides**
* **The tool executes**
* **The message format connects them cleanly**

Once the model knows what tools are available — and how to use them — it’s ready to interact with them in real time. That’s where MCP’s message structure comes in.

### **1.3 MCP Message Types**

At runtime, the model interacts with tools using a structured set of messages. These 4 type of messages define how tool calls are issued, how results are returned, and how the overall flow is managed.

You can think of it like a conversation between an AI assistant and a set of utilities — for example, asking a calculator for a result, then using it to answer a user’s question.

* **Tool\_use :**  Sent by the model when it wants to call a tool. It’s essentially the model saying, “I need this tool to perform a task,” along with the tool name and input parameters. Like asking a calculator: “What’s 3 + 5?”
* **Tool\_response:**  Sent by the tool after completing its task. It includes the result of the operation and identifies which tool produced it. In our example, the calculator responds with “8”.
* **Observation:** Optionally sent to the model to inform it that a tool’s response has been received. This helps the model continue reasoning with up-to-date context. The model now knows: “Okay, the result is 8 — I can use that.”
* **Final\_response:** Sent by the model when it has completed its reasoning and is ready to provide an answer to the client or end user. That’s when it says to the user: “The answer to 3 + 5 is 8.”

These four messages — tool\_use, tool\_response, observation, and final\_response — only work because the model already understands the available tools through their definitions. Without that shared understanding, the message exchange wouldn’t mean anything.

### **1.4 Roles and Responsibilities**

Behind the message exchange, each part of the system plays a specific role. These roles define **who sends what**, **who reacts**, and **how tool use is coordinated**. Keeping these responsibilities separate makes the system more modular, testable, and easier to reason about.

**Client**

The client is the entry point of the system. It sends the user’s message — for example, a question or task — and eventually receives the model’s final response. It doesn’t make decisions or run tools itself. It just passes information in and out of the system.

**Model**

The model is the core reasoning engine. It interprets the user’s input, decides whether a tool is needed, and if so, sends a tool\_use message to request it. Once it receives the result, it processes that information and continues its reasoning, until it’s ready to return a final\_response.

**Tool Server**

The tool server is responsible for executing the actual task. It waits for tool\_use messages, runs the appropriate logic, and replies with a tool\_response. Each tool is a standalone unit — it doesn’t need to know anything about the model or the user, only how to do its specific job when called.

Together, these three roles create a clean separation of concerns. The client handles the interface, the model handles the thinking, and the tool handles the doing.

### **1.5 Minimal Flow**

Let’s put it all together. With the roles defined and messages flowing, here’s what a minimal MCP interaction looks like from end to end. This will give you a mental model for what’s happening under the hood — and it’s exactly the flow you’ll build in the hands-on section.

It starts with the **client**, which sends a user message — something like a question or command — to the **model**.

The **model** interprets that input. If it needs to use a tool, it emits a tool\_use message that includes the name of the tool and the required parameters.

That message is received by the **tool server**, which processes it. The tool does its job — maybe it calculates a number, fetches data, or generates some text — and then returns a tool\_response with the result.

The **model** then receives this response, uses it as part of its reasoning process, and ultimately produces a final\_response.

That response is sent back to the **client**, which delivers the final output to the user.

So the full flow looks like this:

Client → Model → Tool Server → Model → Client

It’s a simple loop, but it enables powerful, structured reasoning — and it’s what we’ll implement together in just a moment.

## 2. Hands-On MCP Server

Now that we’ve covered the theory behind the Model Context Protocol (MCP), it’s time to get our hands dirty.

In this section, each participant will **build and run an MCP-compatible server** using one of three supported languages: **Python**, **TypeScript**, or **Go**.

The goal is to give you direct experience with:

* Defining tools and making them discoverable by a model
* Creating a minimal MCP-compliant HTTP server
* Testing interactions between the model and your tool

By the end of this exercise, you’ll have a functional server that a language model can call in a structured, predictable way — just like in real-world applications of MCP.

### 2.1 Step 1: Define the MCP Tool Endpoint

Here our tool definition:

| {  "name": "add",  "description": "Adds two numbers together.",  "parameters": {  "type": "object",  "properties": {  "a": { "type": "number" },  "b": { "type": "number" }  },  "required": ["a", "b"]  } } |
| --- |

### 2.2 Step 2: Define the MCP Tool Endpoint

Endpoint:

| GET /\_mcp/tools |
| --- |

Now here’s **starter code** for each language

Go :

| package main import (  "context"  "fmt"  "log"  "net/http"  "github.com/anthropics/mcp-go/mcp" ) var addTool = &mcp.ToolDefinition{  Name: "add",  Description: "Adds two numbers together.",  Parameters: map[string]any{  "type": "object",  "properties": map[string]any{  "a": map[string]string{"type": "number"},  "b": map[string]string{"type": "number"},  },  "required": []string{"a", "b"},  },  Handler: func(ctx context.Context, input map[string]any) (any, error) {  a, okA := input["a"].(float64)  b, okB := input["b"].(float64)  if !okA || !okB {  return nil, fmt.Errorf("invalid input: expected numbers for 'a' and 'b'")  }  return map[string]any{"result": a + b}, nil  }, } func main() {  server := mcp.NewServer()  // Health check  http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {  w.Write([]byte("OK"))  })  server.AddTool(addTool)  fmt.Println("Server listening on http://localhost:3000")  log.Fatal(http.ListenAndServe(":3000", server)) } |
| --- |

Python: (Express)

| from fastapi import FastAPI, HTTPException, Request from fastapi.responses import JSONResponse  app = FastAPI()  @app.get("/") def health():  return "OK"  @app.get("/\_mcp/tools") def get\_tools():  return [  {  "name": "add",  "description": "Adds two numbers together.",  "parameters": {  "type": "object",  "properties": {  "a": { "type": "number" },  "b": { "type": "number" }  },  "required": ["a", "b"]  }  }  ]  @app.post("/add") async def add\_handler(request: Request):  body = await request.json()  a = body.get("a")  b = body.get("b")  if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):  raise HTTPException(status\_code=400, detail="a and b must be numbers")  return {"result": a + b} |
| --- |

To run

| pip install fastapi uvicorn uvicorn main:app --reload |
| --- |

Typescript:

| import express from "express";  const app = express(); const port = 3000;  app.use(express.json());  app.get("/", (\_req, res) => {  res.send("OK"); });  app.get("/\_mcp/tools", (\_req, res) => {  res.json([  {  name: "add",  description: "Adds two numbers together.",  parameters: {  type: "object",  properties: {  a: { type: "number" },  b: { type: "number" }  },  required: ["a", "b"]  }  }  ]); });  app.post("/add", (req, res) => {  const { a, b } = req.body;  if (typeof a !== "number" || typeof b !== "number") {  return res.status(400).json({ error: "a and b must be numbers" });  }  res.json({ result: a + b }); });  app.listen(port, () => {  console.log(`Server running at http://localhost:${port}`); }); |
| --- |

Run:

| npm init -y npm install express @types/express ts-node typescript npx ts-node index.ts |
| --- |

### **Testing the Tool**

Participants will use the same request:

| curl -X POST http://localhost:3000 \  -H "Content-Type: application/json" \  -d '{  "type": "tool\_use",  "tool\_name": "echo",  "input": {  "message": "Hello MCP!"  }  }' Expected reply: {  "type": "tool\_response",  "output": {  "message": "Hello MCP!"  } } |
| --- |