

Quickstart

- [`<!-- page slug: mcp-docs-quickstart-client | order: 13 -->`](#)
- [`<!-- page slug: mcp-docs-quickstart-server | order: 14 -->`](#)
- [`<!-- page slug: mcp-docs-quickstart-user | order: 15 -->`](#)

<!-- page slug: mcp-docs-quickstart-client | order: 13 -->

title: "For Client Developers" description: "Get started building your own client that can integrate with all MCP servers." —

In this tutorial, you'll learn how to build a LLM-powered chatbot client that connects to MCP servers. It helps to have gone through the [Server quickstart](#) that guides you through the basic of building your first server.

[You can find the complete code for this tutorial here.](#)

System Requirements

Before starting, ensure your system meets these requirements: - Mac or Windows computer - Latest Python version installed - Latest version of `uv` installed

Setting Up Your Environment

First, create a new Python project with `uv`:

```
# Create project directory
uv init mcp-client
cd mcp-client

# Create virtual environment
uv venv
```

```
# Activate virtual environment
# On Windows:
.venv\Scripts\activate
# On Unix or MacOS:
source .venv/bin/activate

# Install required packages
uv add mcp anthropic python-dotenv

# Remove boilerplate files
rm main.py

# Create our main file
touch client.py
```

Setting Up Your API Key

You'll need an Anthropic API key from the [Anthropic Console](#).

Create a `.env` file to store it:

```
# Create .env file
touch .env
```

Add your key to the `.env` file:

```
ANTHROPIC_API_KEY=<your key here>
```

Add `.env` to your `.gitignore`:

```
echo ".env" >> .gitignore
```

Make sure you keep your `ANTHROPIC_API_KEY` secure!

Creating the Client

Basic Client Structure

First, let's set up our imports and create the basic client class:

```
import asyncio
from typing import Optional
from contextlib import AsyncExitStack

from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client

from anthropic import Anthropic
from dotenv import load_dotenv

load_dotenv() # load environment variables from .env

class MCPClient:
    def __init__(self):
        # Initialize session and client objects
        self.session: Optional[ClientSession] = None
        self.exit_stack = AsyncExitStack()
        self.anthropic = Anthropic()

        # methods will go here
```

Server Connection Management

Next, we'll implement the method to connect to an MCP server:

```
async def connect_to_server(self, server_script_path: str):
    """Connect to an MCP server

    Args:
        server_script_path: Path to the server script (.py or .js)
    """
    is_python = server_script_path.endswith('.py')
    is_js = server_script_path.endswith('.js')
    if not (is_python or is_js):
        raise ValueError("Server script must be a .py or .js file")
```

```

command = "python" if is_python else "node"
server_params = StdioServerParameters(
    command=command,
    args=[server_script_path],
    env=None
)

stdio_transport = await self.exit_stack.enter_async_context(stdio_client(server_params))
self.stdio, self.write = stdio_transport
self.session = await self.exit_stack.enter_async_context(ClientSession(self.stdio,
self.write))

await self.session.initialize()

# List available tools
response = await self.session.list_tools()
tools = response.tools
print("\nConnected to server with tools:", [tool.name for tool in tools])

```

Query Processing Logic

Now let's add the core functionality for processing queries and handling tool calls:

```

async def process_query(self, query: str) -> str:
    """Process a query using Claude and available tools"""
    messages = [
        {
            "role": "user",
            "content": query
        }
    ]

    response = await self.session.list_tools()
    available_tools = [{
        "name": tool.name,
        "description": tool.description,
        "input_schema": tool.inputSchema
    } for tool in response.tools]

```

```

# Initial Claude API call
response = self.anthropic.messages.create(
    model="claude-3-5-sonnet-20241022",
    max_tokens=1000,
    messages=messages,
    tools=available_tools
)

# Process response and handle tool calls
final_text = []

assistant_message_content = []
for content in response.content:
    if content.type == 'text':
        final_text.append(content.text)
        assistant_message_content.append(content)
    elif content.type == 'tool_use':
        tool_name = content.name
        tool_args = content.input

        # Execute tool call
        result = await self.session.call_tool(tool_name, tool_args)
        final_text.append(f"[Calling tool {tool_name} with args {tool_args}]")

        assistant_message_content.append(content)
        messages.append({
            "role": "assistant",
            "content": assistant_message_content
        })
        messages.append({
            "role": "user",
            "content": [
                {
                    "type": "tool_result",
                    "tool_use_id": content.id,
                    "content": result.content
                }
            ]
        })

```

```

    })

    # Get next response from Claude
    response = self.anthropic.messages.create(
        model="claude-3-5-sonnet-20241022",
        max_tokens=1000,
        messages=messages,
        tools=available_tools
    )

    final_text.append(response.content[0].text)

return "\n".join(final_text)

```

Interactive Chat Interface

Now we'll add the chat loop and cleanup functionality:

```

async def chat_loop(self):
    """Run an interactive chat loop"""
    print("\nMCP Client Started!")
    print("Type your queries or 'quit' to exit.")

    while True:
        try:
            query = input("\nQuery: ").strip()

            if query.lower() == 'quit':
                break

            response = await self.process_query(query)
            print("\n" + response)

        except Exception as e:
            print(f"\nError: {str(e)}")

    async def cleanup(self):
        """Clean up resources"""
        await self.exit_stack.aclose()

```

Main Entry Point

Finally, we'll add the main execution logic:

```
async def main():
    if len(sys.argv) < 2:
        print("Usage: python client.py <path_to_server_script>")
        sys.exit(1)

    client = MCPClient()
    try:
        await client.connect_to_server(sys.argv[1])
        await client.chat_loop()
    finally:
        await client.cleanup()

if __name__ == "__main__":
    import sys
    asyncio.run(main())
```

You can find the complete `client.py` file [here](#).

Key Components Explained

1. Client Initialization

- The `MCPClient` class initializes with session management and API clients
- Uses `AsyncExitStack` for proper resource management
- Configures the Anthropic client for Claude interactions

2. Server Connection

- Supports both Python and Node.js servers
- Validates server script type
- Sets up proper communication channels
- Initializes the session and lists available tools

3. Query Processing

- Maintains conversation context
- Handles Claude's responses and tool calls
- Manages the message flow between Claude and tools
- Combines results into a coherent response

4. Interactive Interface

- Provides a simple command-line interface
- Handles user input and displays responses
- Includes basic error handling
- Allows graceful exit

5. Resource Management

- Proper cleanup of resources
- Error handling for connection issues
- Graceful shutdown procedures

Common Customization Points

1. Tool Handling

- Modify `process_query()` to handle specific tool types
- Add custom error handling for tool calls
- Implement tool-specific response formatting

2. Response Processing

- Customize how tool results are formatted
- Add response filtering or transformation
- Implement custom logging

3. User Interface

- Add a GUI or web interface
- Implement rich console output
- Add command history or auto-completion

Running the Client

To run your client with any MCP server:

```
uv run client.py path/to/server.py # python server
uv run client.py path/to/build/index.js # node server
```

If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `python client.py ../quickstart-resources/weather-server-python/weather.py`

The client will: 1. Connect to the specified server 2. List available tools 3. Start an interactive chat session where you can: - Enter queries - See tool executions - Get responses from Claude

Here's an example of what it should look like if connected to the weather server from the server quickstart:

client-claude-cli-python.png

How It Works

When you submit a query:

1. The client gets the list of available tools from the server
2. Your query is sent to Claude along with tool descriptions
3. Claude decides which tools (if any) to use
4. The client executes any requested tool calls through the server
5. Results are sent back to Claude
6. Claude provides a natural language response
7. The response is displayed to you

Best practices

1. **Error Handling**
 - Always wrap tool calls in try-catch blocks
 - Provide meaningful error messages
 - Gracefully handle connection issues
2. **Resource Management**
 - Use `AsyncExitStack` for proper cleanup
 - Close connections when done
 - Handle server disconnections
3. **Security**
 - Store API keys securely in `.env`
 - Validate server responses
 - Be cautious with tool permissions

Troubleshooting

Server Path Issues

- Double-check the path to your server script is correct
- Use the absolute path if the relative path isn't working
- For Windows users, make sure to use forward slashes (/) or escaped backslashes (\) in the path
- Verify the server file has the correct extension (.py for Python or .js for Node.js)

Example of correct path usage:

```
# Relative path
uv run client.py ./server/weather.py

# Absolute path
uv run client.py /Users/username/projects/mcp-server/weather.py

# Windows path (either format works)
uv run client.py C:/projects/mcp-server/weather.py
uv run client.py C:\\projects\\mcp-server\\weather.py
```

Response Timing

- The first response might take up to 30 seconds to return
- This is normal and happens while:
 - The server initializes
 - Claude processes the query
 - Tools are being executed
- Subsequent responses are typically faster
- Don't interrupt the process during this initial waiting period

Common Error Messages

If you see: - `FileNotFoundError`: Check your server path - `Connection refused`: Ensure the server is running and the path is correct - `Tool execution failed`: Verify the tool's required environment variables are set - `Timeout error`: Consider increasing the timeout in your client configuration

[You can find the complete code for this tutorial here.](#) ## System Requirements

Before starting, ensure your system meets these requirements: - Mac or Windows computer - Node.js 17 or higher installed - Latest version of `npm` installed - Anthropic API key (Claude)

Setting Up Your Environment

First, let's create and set up our project:

```
```bash MacOS/Linux # Create project directory mkdir mcp-client-typescript cd mcp-client-typescript
```

## Initialize npm project

```
npm init -y
```

## Install dependencies

```
npm install @anthropic-ai/sdk @modelcontextprotocol/sdk dotenv
```

## Install dev dependencies

```
npm install -D @types/node typescript
```

## Create source file

```
touch index.ts
```

```
```powershell Windows
# Create project directory
md mcp-client-typescript
cd mcp-client-typescript

# Initialize npm project
```

```
npm init -y

# Install dependencies
npm install @anthropic-ai/sdk @modelcontextprotocol/sdk dotenv

# Install dev dependencies
npm install -D @types/node typescript

# Create source file
new-item index.ts
```

Update your `package.json` to set `type: "module"` and a build script:

```
json package.json {  "type": "module",  "scripts": {    "build": "tsc && chmod 755 build/index.js"  } }
```

Create a `tsconfig.json` in the root of your project:

```
json tsconfig.json {  "compilerOptions": {    "target": "ES2022",    "module": "Node16",    "moduleResolution": "Node16",    "outDir": "./build",    "rootDir": "./",    "strict": true,    "esModuleInterop": true,    "skipLibCheck": true,    "forceConsistentCasingInFileNames": true  },  "include": ["index.ts"],  "exclude": ["node_modules"] }
```

Setting Up Your API Key

You'll need an Anthropic API key from the [Anthropic Console](#).

Create a `.env` file to store it:

```
echo "ANTHROPIC_API_KEY=<your key here>" > .env
```

Add `.env` to your `.gitignore`:

```
echo ".env" >> .gitignore
```

Make sure you keep your `ANTHROPIC_API_KEY` secure!

Creating the Client

Basic Client Structure

First, let's set up our imports and create the basic client class in `index.ts`:

```
import { Anthropic } from "@anthropic-ai/sdk";
import {
  MessageParam,
  Tool,
} from "@anthropic-ai/sdk/resources/messages/messages.mjs";
import { Client } from "@modelcontextprotocol/sdk/client/index.js";
import { StdioClientTransport } from "@modelcontextprotocol/sdk/client/stdio.js";
import readline from "readline/promises";
import dotenv from "dotenv";

dotenv.config();

const ANTHROPIC_API_KEY = process.env.ANTHROPIC_API_KEY;
if (!ANTHROPIC_API_KEY) {
  throw new Error("ANTHROPIC_API_KEY is not set");
}

class MCPClient {
  private mcp: Client;
  private anthropic: Anthropic;
  private transport: StdioClientTransport | null = null;
  private tools: Tool[] = [];

  constructor() {
    this.anthropic = new Anthropic({
      apiKey: ANTHROPIC_API_KEY,
    });
    this.mcp = new Client({ name: "mcp-client-cli", version: "1.0.0" });
  }

  // methods will go here
}
```

Server Connection Management

Next, we'll implement the method to connect to an MCP server:

```

async connectToServer(serverScriptPath: string) {
  try {
    const isJs = serverScriptPath.endsWith(".js");
    const isPy = serverScriptPath.endsWith(".py");
    if (!isJs && !isPy) {
      throw new Error("Server script must be a .js or .py file");
    }
    const command = isPy
      ? process.platform === "win32"
        ? "python"
        : "python3"
      : process.execPath;

    this.transport = new StdioClientTransport({
      command,
      args: [serverScriptPath],
    });
    this.mcp.connect(this.transport);

    const toolsResult = await this.mcp.listTools();
    this.tools = toolsResult.tools.map((tool) => {
      return {
        name: tool.name,
        description: tool.description,
        input_schema: tool.inputSchema,
      };
    });
    console.log(
      "Connected to server with tools:",
      this.tools.map(({ name }) => name)
    );
  } catch (e) {
    console.log("Failed to connect to MCP server: ", e);
    throw e;
  }
}

```

Query Processing Logic

Now let's add the core functionality for processing queries and handling tool calls:

```
async processQuery(query: string) {
  const messages: MessageParam[] = [
    {
      role: "user",
      content: query,
    },
  ];

  const response = await this.anthropic.messages.create({
    model: "claude-3-5-sonnet-20241022",
    max_tokens: 1000,
    messages,
    tools: this.tools,
  });

  const finalText = [];
  const toolResults = [];

  for (const content of response.content) {
    if (content.type === "text") {
      finalText.push(content.text);
    } else if (content.type === "tool_use") {
      const toolName = content.name;
      const toolArgs = content.input as { [x: string]: unknown } | undefined;

      const result = await this.mcp.callTool({
        name: toolName,
        arguments: toolArgs,
      });
      toolResults.push(result);
      finalText.push(
        `[Calling tool ${toolName} with args ${JSON.stringify(toolArgs)}]`
      );
    }

    messages.push({
      role: "user",
      content: result.content as string,
    });
  }
}
```

```

const response = await this.anthropic.messages.create({
  model: "claude-3-5-sonnet-20241022",
  max_tokens: 1000,
  messages,
});

finalText.push(
  response.content[0].type === "text" ? response.content[0].text : ""
);
}
}

return finalText.join("\n");
}

```

Interactive Chat Interface

Now we'll add the chat loop and cleanup functionality:

```

async chatLoop() {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
  });

  try {
    console.log("\nMCP Client Started!");
    console.log("Type your queries or 'quit' to exit.");

    while (true) {
      const message = await rl.question("\nQuery: ");
      if (message.toLowerCase() === "quit") {
        break;
      }

      const response = await this.processQuery(message);
      console.log("\n" + response);
    }
  } finally {

```

```
        rl.close();
    }
}

async cleanup() {
    await this.mcp.close();
}
```

Main Entry Point

Finally, we'll add the main execution logic:

```
async function main() {
    if (process.argv.length < 3) {
        console.log("Usage: node index.ts <path_to_server_script>");
        return;
    }
    const mcpClient = new MCPClient();
    try {
        await mcpClient.connectToServer(process.argv[2]);
        await mcpClient.chatLoop();
    } finally {
        await mcpClient.cleanup();
        process.exit(0);
    }
}

main();
```

Running the Client

To run your client with any MCP server:

```
# Build TypeScript
npm run build

# Run the client
```

```
node build/index.js path/to/server.py # python server
node build/index.js path/to/build/index.js # node server
```

If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `node build/index.js ../quickstart-resources/weather-server-typescript/build/index.js`

The client will: 1. Connect to the specified server 2. List available tools 3. Start an interactive chat session where you can: - Enter queries - See tool executions - Get responses from Claude

How It Works

When you submit a query:

1. The client gets the list of available tools from the server
2. Your query is sent to Claude along with tool descriptions
3. Claude decides which tools (if any) to use
4. The client executes any requested tool calls through the server
5. Results are sent back to Claude
6. Claude provides a natural language response
7. The response is displayed to you

Best practices

1. **Error Handling**
 - Use TypeScript's type system for better error detection
 - Wrap tool calls in try-catch blocks
 - Provide meaningful error messages
 - Gracefully handle connection issues
2. **Security**
 - Store API keys securely in `.env`
 - Validate server responses
 - Be cautious with tool permissions

Troubleshooting

Server Path Issues

- Double-check the path to your server script is correct
- Use the absolute path if the relative path isn't working
- For Windows users, make sure to use forward slashes (/) or escaped backslashes (\) in the path
- Verify the server file has the correct extension (.js for Node.js or .py for Python)

Example of correct path usage:

```
# Relative path
node build/index.js ./server/build/index.js

# Absolute path
node build/index.js /Users/username/projects/mcp-server/build/index.js

# Windows path (either format works)
node build/index.js C:/projects/mcp-server/build/index.js
node build/index.js C:\\projects\\mcp-server\\build\\index.js
```

Response Timing

- The first response might take up to 30 seconds to return
- This is normal and happens while:
 - The server initializes
 - Claude processes the query
 - Tools are being executed
- Subsequent responses are typically faster
- Don't interrupt the process during this initial waiting period

Common Error Messages

If you see: - `Error: Cannot find module`: Check your build folder and ensure TypeScript compilation succeeded - `Connection refused`: Ensure the server is running and the path is correct - `Tool execution failed`: Verify the tool's required environment variables are set - `ANTHROPIC_API_KEY is not set`: Check your .env file and environment variables - `TypeError`: Ensure you're using the correct types for tool arguments

This is a quickstart demo based on Spring AI MCP auto-configuration and boot starters. To learn how to create sync and async MCP Clients manually, consult the [Java SDK Client](#) documentation

This example demonstrates how to build an interactive chatbot that combines Spring AI's Model Context Protocol (MCP) with the [Brave Search MCP Server](#). The application creates a conversational

interface powered by Anthropic's Claude AI model that can perform internet searches through Brave Search, enabling natural language interactions with real-time web data. [You can find the complete code for this tutorial here.](#)

System Requirements

Before starting, ensure your system meets these requirements: - Java 17 or higher - Maven 3.6+ - npx package manager - Anthropic API key (Claude) - Brave Search API key

Setting Up Your Environment

1. Install npx (Node Package eXecute): First, make sure to install [npm](#) and then run:

```
npm install -g npx
```

2. Clone the repository:

```
git clone https://github.com/spring-projects/spring-ai-examples.git  
cd model-context-protocol/brave-chatbot
```

3. Set up your API keys:

```
export ANTHROPIC_API_KEY='your-anthropic-api-key-here'  
export BRAVE_API_KEY='your-brave-api-key-here'
```

4. Build the application:

```
./mvnw clean install
```

5. Run the application using Maven:

```
./mvnw spring-boot:run
```

Make sure you keep your `ANTHROPIC_API_KEY` and `BRAVE_API_KEY` keys secure!

How it Works

The application integrates Spring AI with the Brave Search MCP server through several components:

MCP Client Configuration

1. Required dependencies in pom.xml:

```
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-mcp-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-model-anthropic</artifactId>
</dependency>
```

2. Application properties (application.yml):

```
spring:
  ai:
    mcp:
      client:
        enabled: true
        name: brave-search-client
        version: 1.0.0
        type: SYNC
        request-timeout: 20s
        stdio:
          root-change-notification: true
          servers-configuration: classpath:/mcp-servers-config.json
      anthropic:
        api-key: ${ANTHROPIC_API_KEY}
```

This activates the `spring-ai-starter-mcp-client` to create one or more `McpClient`s based on the provided server configuration.

3. MCP Server Configuration (`mcp-servers-config.json`):

```
{
  "mcpServers": {
    "brave-search": {
      "command": "npx",
      "args": [
```

```
        "-y",
        "@modelcontextprotocol/server-brave-search"
    ],
    "env": {
        "BRAVE_API_KEY": "<PUT YOUR BRAVE API KEY>"
    }
}
}
```

Chat Implementation

The chatbot is implemented using Spring AI's ChatClient with MCP tool integration:

```
var chatClient = chatClientBuilder
    .defaultSystem("You are useful assistant, expert in AI and Java.")
    .defaultTools((Object[]) mcpToolAdapter.toolCallbacks())
    .defaultAdvisors(new MessageChatMemoryAdvisor(new InMemoryChatMemory()))
    .build();
```

Key features: - Uses Claude AI model for natural language understanding - Integrates Brave Search through MCP for real-time web search capabilities - Maintains conversation memory using InMemoryChatMemory - Runs as an interactive command-line application

Build and run

```
./mvnw clean install
java -jar ./target/ai-mcp-brave-chatbot-0.0.1-SNAPSHOT.jar
```

or

```
./mvnw spring-boot:run
```

The application will start an interactive chat session where you can ask questions. The chatbot will use Brave Search when it needs to find information from the internet to answer your queries.

The chatbot can: - Answer questions using its built-in knowledge - Perform web searches when needed using Brave Search - Remember context from previous messages in the conversation - Combine information from multiple sources to provide comprehensive answers

Advanced Configuration

The MCP client supports additional configuration options:

- Client customization through `McpSyncClientCustomizer` or `McpAsyncClientCustomizer`
- Multiple clients with multiple transport types: `STDIO` and `SSE` (Server-Sent Events)
- Integration with Spring AI's tool execution framework
- Automatic client initialization and lifecycle management

For WebFlux-based applications, you can use the WebFlux starter instead:

```
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-mcp-client-webflux-spring-boot-starter</artifactId>
</dependency>
```

This provides similar functionality but uses a WebFlux-based SSE transport implementation, recommended for production deployments.

[You can find the complete code for this tutorial here.](#)

System Requirements

Before starting, ensure your system meets these requirements: - Java 17 or higher - Anthropic API key (Claude)

Setting up your environment

First, let's install `java` and `gradle` if you haven't already. You can download `java` from [official Oracle JDK website](#). Verify your `java` installation:

```
java --version
```

Now, let's create and set up your project:

```
```bash MacOS/Linux # Create a new directory for our project mkdir kotlin-mcp-client cd kotlin-mcp-client
```

# Initialize a new kotlin project

gradle init

```
```powershell Windows
# Create a new directory for our project
md kotlin-mcp-client
cd kotlin-mcp-client
# Initialize a new kotlin project
gradle init
```

After running `gradle init`, you will be presented with options for creating your project. Select **Application** as the project type, **Kotlin** as the programming language, and **Java 17** as the Java version.

Alternatively, you can create a Kotlin application using the [IntelliJ IDEA project wizard](#).

After creating the project, add the following dependencies:

```
```kotlin build.gradle.kts val mcpVersion = "0.4.0" val slf4jVersion = "2.0.9" val anthropicVersion = "0.8.0"
```

```
dependencies { implementation("io.modelcontextprotocol:kotlin-sdk:$mcpVersion")
implementation("org.slf4j:slf4j-nop:$slf4jVersion") implementation("com.anthropic:anthropic-
java:$anthropicVersion") }
```

```
```groovy build.gradle
def mcpVersion = '0.3.0'
def slf4jVersion = '2.0.9'
def anthropicVersion = '0.8.0'
dependencies {
    implementation "io.modelcontextprotocol:kotlin-sdk:$mcpVersion"
    implementation "org.slf4j:slf4j-nop:$slf4jVersion"
    implementation "com.anthropic:anthropic-java:$anthropicVersion"
}
```

Also, add the following plugins to your build script: `kotlin build.gradle.kts plugins { id("com.github.johnrengelman.shadow") version "8.1.1" }`

```
groovy build.gradle plugins { id 'com.github.johnrengelman.shadow' version '8.1.1' }
```

Setting up your API key

You'll need an Anthropic API key from the [Anthropic Console](#).

Set up your API key:

```
export ANTHROPIC_API_KEY='your-anthropic-api-key-here'
```

Make sure you keep your `ANTHROPIC_API_KEY` secure!

Creating the Client

Basic Client Structure

First, let's create the basic client class:

```
class MCPClient : AutoCloseable {
    private val anthropic = AnthropicOkHttpClient.fromEnv()
    private val mcp: Client = Client(clientInfo = Implementation(name = "mcp-client-cli",
version = "1.0.0"))
    private lateinit var tools: List<ToolUnion>

    // methods will go here

    override fun close() {
        runBlocking {
            mcp.close()
            anthropic.close()
        }
    }
}
```

Server connection management

Next, we'll implement the method to connect to an MCP server:

```

suspend fun connectToServer(serverScriptPath: String) {
    try {
        val command = buildList {
            when (serverScriptPath.substringAfterLast(".")) {
                "js" -> add("node")
                "py" -> add(if (System.getProperty("os.name").lowercase().contains("win"))
                    "python" else "python3")
                "jar" -> addAll(listOf("java", "-jar"))
                else -> throw IllegalArgumentException("Server script must be a .js, .py or
.jar file")
            }
            add(serverScriptPath)
        }

        val process = ProcessBuilder(command).start()
        val transport = StdioClientTransport(
            input = process.inputStream.asSource().buffered(),
            output = process.outputStream.asSink().buffered()
        )

        mcp.connect(transport)

        val toolsResult = mcp.listTools()
        tools = toolsResult?.tools?.map { tool ->
            ToolUnion.ofTool(
                Tool.builder()
                    .name(tool.name)
                    .description(tool.description ?: "")
                    .inputSchema(
                        Tool.InputSchema.builder()
                            .type(JsonValue.from(tool.inputSchema.type))
                            .properties(tool.inputSchema.properties.toJsonValue())
                            .putAdditionalProperty("required",
                                JsonValue.from(tool.inputSchema.required))
                            .build()
                    )
            )
        } ?: emptyList()
        println("Connected to server with tools: ${tools.joinToString(", ") {

```

```

it.tool().get().name() }}"
    } catch (e: Exception) {
        println("Failed to connect to MCP server: $e")
        throw e
    }
}

```

Also create a helper function to convert from `JsonObject` to `JsonValue` for Anthropic:

```

private fun JsonObject.toJsonValue(): JsonValue {
    val mapper = ObjectMapper()
    val node = mapper.readTree(this.toString())
    return JsonValue.fromJsonNode(node)
}

```

Query processing logic

Now let's add the core functionality for processing queries and handling tool calls:

```

private val messageParamsBuilder: MessageCreateParams.Builder = MessageCreateParams.builder()
    .model(Model.CLAUDE_3_5_SONNET_20241022)
    .maxTokens(1024)

suspend fun processQuery(query: String): String {
    val messages = mutableListOf(
        MessageParam.builder()
            .role(MessageParam.Role.USER)
            .content(query)
            .build()
    )

    val response = anthropic.messages().create(
        messageParamsBuilder
            .messages(messages)
            .tools(tools)
            .build()
    )

    val finalText = mutableListOf<String>()
}

```

```

response.content().forEach { content ->
    when {
        content.isText() -> finalText.add(content.text().getOrNull()?.text() ?: "")

        content.isToolUse() -> {
            val toolName = content.toolUse().get().name()
            val toolArgs =
                content.toolUse().get()._input().convert(object :
TypeReference<Map<String, JsonValue>>() {})

            val result = mcp.callTool(
                name = toolName,
                arguments = toolArgs ?: emptyMap()
            )
            finalText.add("[Calling tool $toolName with args $toolArgs]")

            messages.add(
                MessageParam.builder()
                    .role(MessageParam.Role.USER)
                    .content(
                        """
                            "type": "tool_result",
                            "tool_name": $toolName,
                            "result": ${result?.content?.joinToString("\n") { (it as
TextContent).text ?: "" }}
                        """.trimIndent()
                    )
                    .build()
            )

            val aiResponse = anthropic.messages().create(
                messageParamsBuilder
                    .messages(messages)
                    .build()
            )

            finalText.add(aiResponse.content().first().text().getOrNull()?.text() ?: "")
        }
    }
}

```

```
}

return finalText.joinToString("\n", prefix = "", postfix = "")
}
```

Interactive chat

We'll add the chat loop:

```
suspend fun chatLoop() {
    println("\nMCP Client Started!")
    println("Type your queries or 'quit' to exit.")

    while (true) {
        print("\nQuery: ")
        val message = readLine() ?: break
        if (message.lowercase() == "quit") break
        val response = processQuery(message)
        println("\n$response")
    }
}
```

Main entry point

Finally, we'll add the main execution function:

```
fun main(args: Array<String>) = runBlocking {
    if (args.isEmpty()) throw IllegalArgumentException("Usage: java -jar  
<your_path>/build/libs/kotlin-mcp-client-0.1.0-all.jar <path_to_server_script>")
    val serverPath = args.first()
    val client = MCPClient()
    client.use {
        client.connectToServer(serverPath)
        client.chatLoop()
    }
}
```

Running the client

To run your client with any MCP server:

```
./gradlew build

# Run the client
java -jar build/libs/<your-jar-name>.jar path/to/server.jar # jvm server
java -jar build/libs/<your-jar-name>.jar path/to/server.py # python server
java -jar build/libs/<your-jar-name>.jar path/to/build/index.js # node server
```

If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `java -jar build/libs/kotlin-mcp-client-0.1.0-all.jar .../samples/weather-stdio-server/build/libs/weather-stdio-server-0.1.0-all.jar`

The client will: 1. Connect to the specified server 2. List available tools 3. Start an interactive chat session where you can: - Enter queries - See tool executions - Get responses from Claude

How it works

Here's a high-level workflow schema:

```
---
config:
  theme: neutral
---

sequenceDiagram
    actor User
    participant Client
    participant Claude
    participant MCP_Server as MCP Server
    participant Tools

    User->>Client: Send query
    Client<<->>MCP_Server: Get available tools
    Client->>Claude: Send query with tool descriptions
    Claude-->>Client: Decide tool execution
    Client->>MCP_Server: Request tool execution
```

```
MCP_Server->>Tools: Execute chosen tools
Tools-->>MCP_Server: Return results
MCP_Server-->>Client: Send results
Client->>Claude: Send tool results
Claude-->>Client: Provide final response
Client-->>User: Display response
```

When you submit a query: 1. The client gets the list of available tools from the server 2. Your query is sent to Claude along with tool descriptions 3. Claude decides which tools (if any) to use 4. The client executes any requested tool calls through the server 5. Results are sent back to Claude 6. Claude provides a natural language response 7. The response is displayed to you

Best practices

1. Error Handling

- Leverage Kotlin's type system to model errors explicitly
- Wrap external tool and API calls in `try-catch` blocks when exceptions are possible
- Provide clear and meaningful error messages
- Handle network timeouts and connection issues gracefully

2. Security

- Store API keys and secrets securely in `local.properties`, environment variables, or secret managers
- Validate all external responses to avoid unexpected or unsafe data usage
- Be cautious with permissions and trust boundaries when using tools

Troubleshooting

Server Path Issues

- Double-check the path to your server script is correct
- Use the absolute path if the relative path isn't working
- For Windows users, make sure to use forward slashes (/) or escaped backslashes (\) in the path
- Make sure that the required runtime is installed (java for Java, npm for Node.js, or uv for Python)
- Verify the server file has the correct extension (.jar for Java, .js for Node.js or .py for Python)

Example of correct path usage:

```
# Relative path
java -jar build/libs/client.jar ./server/build/libs/server.jar

# Absolute path
java -jar build/libs/client.jar /Users/username/projects/mcp-server/build/libs/server.jar

# Windows path (either format works)
java -jar build/libs/client.jar C:/projects/mcp-server/build/libs/server.jar
java -jar build/libs/client.jar C:\\projects\\mcp-server\\build\\libs\\server.jar
```

Response Timing

- The first response might take up to 30 seconds to return
- This is normal and happens while:
 - The server initializes
 - Claude processes the query
 - Tools are being executed
- Subsequent responses are typically faster
- Don't interrupt the process during this initial waiting period

Common Error Messages

If you see: - `Connection refused`: Ensure the server is running and the path is correct - `Tool execution failed`: Verify the tool's required environment variables are set - `ANTHROPIC_API_KEY is not set`: Check your environment variables

[You can find the complete code for this tutorial here.](#)

System Requirements

Before starting, ensure your system meets these requirements: - .NET 8.0 or higher - Anthropic API key (Claude) - Windows, Linux, or MacOS

Setting up your environment

First, create a new .NET project:

```
dotnet new console -n QuickstartClient
cd QuickstartClient
```

Then, add the required dependencies to your project:

```
dotnet add package ModelContextProtocol --prerelease
dotnet add package Anthropic.SDK
dotnet add package Microsoft.Extensions.Hosting
```

Setting up your API key

You'll need an Anthropic API key from the [Anthropic Console](#).

```
dotnet user-secrets init
dotnet user-secrets set "ANTHROPIC_API_KEY" "<your key here>"
```

Creating the Client

Basic Client Structure

First, let's setup the basic client class:

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

var builder = Host.CreateEmptyApplicationBuilder(settings: null);

builder.Configuration
    .AddUserSecrets<Program>();
```

This creates the beginnings of a .NET console application that can read the API key from user secrets.

Next, we'll setup the MCP Client:

```

var (command, arguments) = args switch
{
    [var script] when script.EndsWith(".py") => ("python", script),
    [var script] when script.EndsWith(".js") => ("node", script),
    [var script] when Directory.Exists(script) || (File.Exists(script) &&
script.EndsWith(".csproj")) => ("dotnet", $"run --project {script} --no-build"),
    _ => throw new NotSupportedException("An unsupported server script was provided. Supported
scripts are .py, .js, or .csproj")
};

await using var mcpClient = await McpClientFactory.CreateAsync(new()
{
    Id = "demo-server",
    Name = "Demo Server",
    TransportType = TransportTypes.StdIo,
    TransportOptions = new()
    {
        ["command"] = command,
        ["arguments"] = arguments,
    }
});

var tools = await mcpClient.ListToolsAsync();
foreach (var tool in tools)
{
    Console.WriteLine($"Connected to server with tools: {tool.Name}");
}

```

Be sure to add the `using` statements for the namespaces:

```

using ModelContextProtocol.Client;
using ModelContextProtocol.Protocol.Transport;

```

This configures a MCP client that will connect to a server that is provided as a command line argument. It then lists the available tools from the connected server.

Query processing logic

Now let's add the core functionality for processing queries and handling tool calls:

```

using IChatClient anthropicClient = new AnthropicClient(new
APIAuthentication(builder.Configuration["ANTHROPIC_API_KEY"]))

    .Messages
    .AsBuilder()
    .UseFunctionInvocation()
    .Build();

var options = new ChatOptions
{
    MaxOutputTokens = 1000,
    ModelId = "claude-3-5-sonnet-20241022",
    Tools =[.. tools]
};

while (true)
{
    Console.WriteLine("MCP Client Started!");
    Console.WriteLine("Type your queries or 'quit' to exit.");

    string? query = Console.ReadLine();

    if (string.IsNullOrEmpty(query))
    {
        continue;
    }
    if (string.Equals(query, "quit", StringComparison.OrdinalIgnoreCase))
    {
        break;
    }

    var response = anthropicClient.GetStreamingResponseAsync(query, options);

    await foreach (var message in response)
    {
        Console.Write(message.Text);
    }
    Console.WriteLine();
}

```

Key Components Explained

1. Client Initialization

- The client is initialized using `McpClientFactory.CreateAsync()`, which sets up the transport type and command to run the server.

2. Server Connection

- Supports Python, Node.js, and .NET servers.
- The server is started using the command specified in the arguments.
- Configures to use stdio for communication with the server.
- Initializes the session and available tools.

3. Query Processing

- Leverages [Microsoft.Extensions.AI](#) for the chat client.
- Configures the `IChatClient` to use automatic tool (function) invocation.
- The client reads user input and sends it to the server.
- The server processes the query and returns a response.
- The response is displayed to the user.

Running the Client

To run your client with any MCP server:

```
dotnet run -- path/to/server.csproj # dotnet server
dotnet run -- path/to/server.py # python server
dotnet run -- path/to/server.js # node server
```

If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `dotnet run -- path/to/QuickstartWeatherServer`.

The client will:

1. Connect to the specified server
2. List available tools
3. Start an interactive chat session where you can:

- Enter queries
 - See tool executions
 - Get responses from Claude
4. Exit the session when done

Here's an example of what it should look like it connected to a weather server quickstart:

quickstart-dotnet-client.png

Next steps

<Card title="Example servers" icon="grid" href="/examples"

“ Check out our gallery of official MCP servers and implementations <Card title="Clients" icon="cubes" href="/clients" View the list of clients that support MCP integrations <Card title="Building MCP with LLMs" icon="comments" href="/tutorials/building-mcp-with-llms" Learn how to use LLMs like Claude to speed up your MCP development <Card title="Core architecture" icon="sitemap" href="/docs/concepts/architecture" Understand how MCP connects clients, servers, and LLMs

<!-- page slug: mcp-docs-quickstart-server | order: 14 -->

title: "For Server Developers" description: "Get started building your own server to use in Claude for Desktop and other clients." —

In this tutorial, we'll build a simple MCP weather server and connect it to a host, Claude for Desktop. We'll start with a basic setup, and then progress to more complex use cases.

What we'll be building

Many LLMs do not currently have the ability to fetch the forecast and severe weather alerts. Let's use MCP to solve that!

We'll build a server that exposes two tools: `get-alerts` and `get-forecast`. Then we'll connect the server to an MCP host (in this case, Claude for Desktop):

weather-alerts.png current-weather.png

Servers can connect to any client. We've chosen Claude for Desktop here for simplicity, but we also have guides on [building your own client](#) as well as a [list of other clients here](#).

Because servers are locally run, MCP currently only supports desktop hosts. Remote hosts are in active development.

Core MCP Concepts

MCP servers can provide three main types of capabilities:

1. **Resources:** File-like data that can be read by clients (like API responses or file contents)
2. **Tools:** Functions that can be called by the LLM (with user approval)
3. **Prompts:** Pre-written templates that help users accomplish specific tasks

This tutorial will primarily focus on tools.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

Prerequisite knowledge

This quickstart assumes you have familiarity with: - Python - LLMs like Claude

System requirements

- Python 3.10 or higher installed.
- You must use the Python MCP SDK 1.2.0 or higher.

Set up your environment

First, let's install `uv` and set up our Python project and environment:

```
bash MacOS/Linux curl -LsSf https://astral.sh/uv/install.sh | sh
```

```
powershell Windows powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Make sure to restart your terminal afterwards to ensure that the `uv` command gets picked up.

Now, let's create and set up our project:

```
```bash MacOS/Linux # Create a new directory for our project uv init weather cd weather
```

## Create virtual environment and activate it

```
uv venv source .venv/bin/activate
```

## Install dependencies

```
uv add "mcp[cli]" httpx
```

# Create our server file

```
touch weather.py
```

```
```powershell Windows
# Create a new directory for our project
uv init weather
cd weather

# Create virtual environment and activate it
uv venv
.venv\Scripts\activate

# Install dependencies
uv add mcp[cli] httpx

# Create our server file
new-item weather.py
```

Now let's dive into building your server.

Building your server

Importing packages and setting up the instance

Add these to the top of your `weather.py`:

```
from typing import Any
import httpx
from mcp.server.fastmcp import FastMCP

# Initialize FastMCP server
```

```
mcp = FastMCP("weather")

# Constants
NWS_API_BASE = "https://api.weather.gov"
USER_AGENT = "weather-app/1.0"
```

The FastMCP class uses Python type hints and docstrings to automatically generate tool definitions, making it easy to create and maintain MCP tools.

Helper functions

Next, let's add our helper functions for querying and formatting the data from the National Weather Service API:

```
async def make_nws_request(url: str) -> dict[str, Any] | None:
    """Make a request to the NWS API with proper error handling."""
    headers = {
        "User-Agent": USER_AGENT,
        "Accept": "application/geo+json"
    }
    async with httpx.AsyncClient() as client:
        try:
            response = await client.get(url, headers=headers, timeout=30.0)
            response.raise_for_status()
            return response.json()
        except Exception:
            return None

def format_alert(feature: dict) -> str:
    """Format an alert feature into a readable string."""
    props = feature["properties"]
    return f"""
Event: {props.get('event', 'Unknown')}
Area: {props.get('areaDesc', 'Unknown')}
Severity: {props.get('severity', 'Unknown')}
Description: {props.get('description', 'No description available')}
Instructions: {props.get('instruction', 'No specific instructions provided')}
    """
```

Implementing tool execution

The tool execution handler is responsible for actually executing the logic of each tool. Let's add it:

```
@mcp.tool()
async def get_alerts(state: str) -> str:
    """Get weather alerts for a US state.

    Args:
        state: Two-letter US state code (e.g. CA, NY)
    """
    url = f"{NWS_API_BASE}/alerts/active/area/{state}"
    data = await make_nws_request(url)

    if not data or "features" not in data:
        return "Unable to fetch alerts or no alerts found."

    if not data["features"]:
        return "No active alerts for this state."

    alerts = [format_alert(feature) for feature in data["features"]]
    return "\n---\n".join(alerts)


@mcp.tool()
async def get_forecast(latitude: float, longitude: float) -> str:
    """Get weather forecast for a location.

    Args:
        latitude: Latitude of the location
        longitude: Longitude of the location
    """
    # First get the forecast grid endpoint
    points_url = f"{NWS_API_BASE}/points/{latitude},{longitude}"
    points_data = await make_nws_request(points_url)

    if not points_data:
        return "Unable to fetch forecast data for this location."

    # Get the forecast URL from the points response
```

```

forecast_url = points_data["properties"]["forecast"]
forecast_data = await make_nws_request(forecast_url)

if not forecast_data:
    return "Unable to fetch detailed forecast."

# Format the periods into a readable forecast
periods = forecast_data["properties"]["periods"]
forecasts = []
for period in periods[:5]: # Only show next 5 periods
    forecast = f"""
{period['name']}:
Temperature: {period['temperature']}°{period['temperatureUnit']}
Wind: {period['windSpeed']} {period['windDirection']}
Forecast: {period['detailedForecast']}
"""
    forecasts.append(forecast)

return "\n---\n".join(forecasts)

```

Running the server

Finally, let's initialize and run the server:

```

if __name__ == "__main__":
    # Initialize and run the server
    mcp.run(transport='stdio')

```

Your server is complete! Run `uv run weather.py` to confirm that everything's working.

Let's now test your server from an existing MCP host, Claude for Desktop.

Testing your server with Claude for Desktop

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](#) tutorial to build an MCP client that connects to the server we just built.

First, make sure you have Claude for Desktop installed. [You can install the latest version here](#). If you already have Claude for Desktop, **make sure it's updated to the latest version**.

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/Claude/claude_desktop_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [VS Code](#) installed:

```
code ~/Library/Application\ Support/Claude/claude_desktop_config.json
```

```
code $env:AppData\Claude\claude_desktop_config.json
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
json Python {      "mcpServers": {      "weather": {      "command": "uv",
"args": [      "--directory",
"/ABSOLUTE/PATH/TO/PARENT/FOLDER/weather",      "run",
"weather.py"      ]      }      } } json Python {      "mcpServers": {      "weather":
{      "command": "uv",      "args": [      "--
directory",      "C:\\ABSOLUTE\\PATH\\TO\\PARENT\\FOLDER\\weather",
"run",      "weather.py"      ]      }      } }
```

You may need to put the full path to the `uv` executable in the `command` field. You can get this by running `which uv` on MacOS/Linux or `where uv` on Windows.

Make sure you pass in the absolute path to your server.

This tells Claude for Desktop: 1. There's an MCP server named "weather" 2. To launch it by running `uv --directory /ABSOLUTE/PATH/TO/PARENT/FOLDER/weather run weather.py`

Save the file, and restart **Claude for Desktop**.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here](#).

Prerequisite knowledge

This quickstart assumes you have familiarity with: - TypeScript - LLMs like Claude

System requirements

For TypeScript, make sure you have the latest version of Node installed.

Set up your environment

First, let's install Node.js and npm if you haven't already. You can download them from nodejs.org. Verify your Node.js installation:

```
node --version  
npm --version
```

For this tutorial, you'll need Node.js version 16 or higher.

Now, let's create and set up our project:

```
```bash MacOS/Linux # Create a new directory for our project mkdir weather cd weather
```

## Initialize a new npm project

```
npm init -y
```

## Install dependencies

```
npm install @modelcontextprotocol/sdk zod npm install -D @types/node typescript
```

## Create our files

```
mkdir src touch src/index.ts
```

```
```powershell Windows
# Create a new directory for our project
md weather
cd weather

# Initialize a new npm project
npm init -y

# Install dependencies
npm install @modelcontextprotocol/sdk zod
npm install -D @types/node typescript

# Create our files
md src
new-item src\index.ts
```

Update your package.json to add type: “module” and a build script:

```
json package.json {  "type": "module",  "bin": {    "weather": "./build/index.js"  },  "scripts": {    "build": "tsc && chmod 755 build/index.js"  },  "files": [    "build"  ], }
```

Create a `tsconfig.json` in the root of your project:

```
json tsconfig.json {  "compilerOptions": {    "target": "ES2022",    "module": "Node16",    "moduleResolution": "Node16",    "outDir": "./build",    "rootDir": "./src",    "strict": true,    "esModuleInterop": true,    "skipLibCheck": true,    "forceConsistentCasingInFileNames": true  },  "include": ["src/**/*.ts"],  "exclude": ["node_modules"] }
```

Now let’s dive into building your server.

Building your server

Importing packages and setting up the instance

Add these to the top of your `src/index.ts`:

```

import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";
import { z } from "zod";

const NWS_API_BASE = "https://api.weather.gov";
const USER_AGENT = "weather-app/1.0";

// Create server instance
const server = new McpServer({
  name: "weather",
  version: "1.0.0",
  capabilities: {
    resources: {},
    tools: {},
  },
});

```

Helper functions

Next, let's add our helper functions for querying and formatting the data from the National Weather Service API:

```

// Helper function for making NWS API requests
async function makeNWSRequest<T>(url: string): Promise<T | null> {
  const headers = {
    "User-Agent": USER_AGENT,
    Accept: "application/geo+json",
  };

  try {
    const response = await fetch(url, { headers });
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return (await response.json()) as T;
  } catch (error) {
    console.error("Error making NWS request:", error);
    return null;
  }
}

```

```

}

interface AlertFeature {
  properties: {
    event?: string;
    areaDesc?: string;
    severity?: string;
    status?: string;
    headline?: string;
  };
}

// Format alert data
function formatAlert(feature: AlertFeature): string {
  const props = feature.properties;
  return [
    `Event: ${props.event || "Unknown"}`,
    `Area: ${props.areaDesc || "Unknown"}`,
    `Severity: ${props.severity || "Unknown"}`,
    `Status: ${props.status || "Unknown"}`,
    `Headline: ${props.headline || "No headline"}`,
    "---",
  ].join("\n");
}

interface ForecastPeriod {
  name?: string;
  temperature?: number;
  temperatureUnit?: string;
  windSpeed?: string;
  windDirection?: string;
  shortForecast?: string;
}

interface AlertsResponse {
  features: AlertFeature[];
}

interface PointsResponse {

```

```

    properties: {
      forecast?: string;
    };
  }

interface ForecastResponse {
  properties: {
    periods: ForecastPeriod[];
  };
}

```

Implementing tool execution

The tool execution handler is responsible for actually executing the logic of each tool. Let's add it:

```

// Register weather tools
server.tool(
  "get-alerts",
  "Get weather alerts for a state",
  {
    state: z.string().length(2).describe("Two-letter state code (e.g. CA, NY)"),
  },
  async ({ state }) => {
    const stateCode = state.toUpperCase();
    const alertsUrl = `${NWS_API_BASE}/alerts?area=${stateCode}`;
    const alertsData = await makeNWSRequest<AlertsResponse>(alertsUrl);

    if (!alertsData) {
      return {
        content: [
          {
            type: "text",
            text: "Failed to retrieve alerts data",
          },
        ],
      };
    }

    const features = alertsData.features || [];
  }

```

```

if (features.length === 0) {
  return {
    content: [
      {
        type: "text",
        text: `No active alerts for ${stateCode}`,
      },
    ],
  };
}

const formattedAlerts = features.map(formatAlert);
const alertsText = `Active alerts for ${stateCode}: \n\n${formattedAlerts.join("\n")}`;

return {
  content: [
    {
      type: "text",
      text: alertsText,
    },
  ],
};
},
);

server.tool(
  "get-forecast",
  "Get weather forecast for a location",
  {
    latitude: z.number().min(-90).max(90).describe("Latitude of the location"),
    longitude: z.number().min(-180).max(180).describe("Longitude of the location"),
  },
  async ({ latitude, longitude }) => {
    // Get grid point data
    const pointsUrl = `${NWS_API_BASE}/points/${latitude.toFixed(4)},${longitude.toFixed(4)}`;
    const pointsData = await makeNWSRequest<PointsResponse>(pointsUrl);

    if (!pointsData) {
      return {

```

```
        content: [
          {
            type: "text",
            text: `Failed to retrieve grid point data for coordinates: ${latitude},
${longitude}. This location may not be supported by the NWS API (only US locations are
supported).`,
          },
        ],
      };
    }
  }
}
```

```
const forecastUrl = pointsData.properties?.forecast;
if (!forecastUrl) {
  return {
    content: [
      {
        type: "text",
        text: "Failed to get forecast URL from grid point data",
      },
    ],
  };
}
```

```
// Get forecast data
const forecastData = await makeNWSRequest<ForecastResponse>(forecastUrl);
if (!forecastData) {
  return {
    content: [
      {
        type: "text",
        text: "Failed to retrieve forecast data",
      },
    ],
  };
}
```

```
const periods = forecastData.properties?.periods || [];
if (periods.length === 0) {
  return {
```

```

        content: [
          {
            type: "text",
            text: "No forecast periods available",
          },
        ],
      };
    }

    // Format forecast periods
    const formattedForecast = periods.map((period: ForecastPeriod) =>
      [
        `${period.name || "Unknown"}:`,
        `Temperature: ${period.temperature || "Unknown"}°${period.temperatureUnit || "F"}`,
        `Wind: ${period.windSpeed || "Unknown"} ${period.windDirection || ""}`,
        `${period.shortForecast || "No forecast available"}`,
        "---",
      ].join("\n"),
    );

    const forecastText = `Forecast for ${latitude},
    ${longitude}: \n\n${formattedForecast.join("\n")} `;

    return {
      content: [
        {
          type: "text",
          text: forecastText,
        },
      ],
    };
  },
);

```

Running the server

Finally, implement the main function to run the server:

```
async function main() {
  const transport = new StdioServerTransport();
  await server.connect(transport);
  console.error("Weather MCP Server running on stdio");
}

main().catch((error) => {
  console.error("Fatal error in main():", error);
  process.exit(1);
});
```

Make sure to run `npm run build` to build your server! This is a very important step in getting your server to connect.

Let's now test your server from an existing MCP host, Claude for Desktop.

Testing your server with Claude for Desktop

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](#) tutorial to build an MCP client that connects to the server we just built.

First, make sure you have Claude for Desktop installed. [You can install the latest version here](#). If you already have Claude for Desktop, **make sure it's updated to the latest version**.

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/Claude/claude_desktop_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [VS Code](#) installed:

```
code ~/Library/Application\ Support/Claude/claude_desktop_config.json
```

```
code $env:AppData\Claude\claude_desktop_config.json
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
json Node {  "mcpServers": {    "weather": {      "command": "node",      "args": [        "/ABSOLUTE/PATH/T0/PARENT/FOLDER/weather/build/index.js"      ]    }  } } json Node {  "mcpServers": {    "weather": {      "command": "node",      "args": [        "C:\\PATH\\T0\\PARENT\\FOLDER\\weather\\build\\index.js"      ]    }  } }
```

This tells Claude for Desktop: 1. There's an MCP server named "weather" 2. Launch it by running `node /ABSOLUTE/PATH/T0/PARENT/FOLDER/weather/build/index.js`

Save the file, and restart **Claude for Desktop**.

This is a quickstart demo based on Spring AI MCP auto-configuration and boot starters. To learn how to create sync and async MCP Servers, manually, consult the [Java SDK Server](#) documentation.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

For more information, see the [MCP Server Boot Starter](#) reference documentation. For manual MCP Server implementation, refer to the [MCP Server Java SDK documentation](#).

System requirements

- Java 17 or higher installed.
- [Spring Boot 3.3.x](#) or higher

Set up your environment

Use the [Spring Initializer](#) to bootstrap the project.

You will need to add the following dependencies:

```
```xml org.springframework.ai spring-ai-starter-mcp-server
```

```
<dependency>
 <groupId>org.springframework</groupId>
```

```
<artifactId>spring-web</artifactId>
</dependency>
```

```
</Tab> <Tab title="Gradle">groovy dependencies { implementation
platform("org.springframework.ai:spring-ai-starter-mcp-server") implementation
platform("org.springframework:spring-web")
} ````
```

Then configure your application by setting the application properties:

```
bash application.properties spring.main.bannerMode=off logging.pattern.console=
```

```
yaml application.yml logging: pattern: console: spring: main: banner-mode: off
```

The [Server Configuration Properties](#) documents all available properties.

Now let's dive into building your server.

# Building your server

## Weather Service

Let's implement a [WeatherService.java](#) that uses a REST client to query the data from the National Weather Service API:

```
@Service
public class WeatherService {

 private final RestClient restClient;

 public WeatherService() {
 this.restClient = RestClient.builder()
 .baseUrl("https://api.weather.gov")
 .defaultHeader("Accept", "application/geo+json")
 .defaultHeader("User-Agent", "WeatherApiClient/1.0 (your@email.com)")
 .build();
 }

 @Tool(description = "Get weather forecast for a specific latitude/longitude")
```

```

public String getWeatherForecastByLocation(
 double latitude, // Latitude coordinate
 double longitude // Longitude coordinate
) {
 // Returns detailed forecast including:
 // - Temperature and unit
 // - Wind speed and direction
 // - Detailed forecast description
}

@Tool(description = "Get weather alerts for a US state")
public String getAlerts(
 @ToolParam(description = "Two-letter US state code (e.g. CA, NY)" String state
) {
 // Returns active alerts including:
 // - Event type
 // - Affected area
 // - Severity
 // - Description
 // - Safety instructions
}

//
}

```

The `@Service` annotation with `auto-register` the service in your application context. The Spring AI `@Tool` annotation, making it easy to create and maintain MCP tools.

The auto-configuration will automatically register these tools with the MCP server.

## Create your Boot Application

```

@SpringBootApplication
public class McpServerApplication {

 public static void main(String[] args) {
 SpringApplication.run(McpServerApplication.class, args);
 }

 @Bean

```

```
public ToolCallbackProvider weatherTools(WeatherService weatherService) {
 return MethodToolCallbackProvider.builder().toolObjects(weatherService).build();
}
}
```

Uses the the `MethodToolCallbackProvider` utils to convert the `@Tools` into actionable callbacks used by the MCP server.

## Running the server

Finally, let's build the server:

```
./mvnw clean install
```

This will generate a `mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar` file within the `target` folder.

Let's now test your server from an existing MCP host, Claude for Desktop.

# Testing your server with Claude for Desktop

Claude for Desktop is not yet available on Linux.

First, make sure you have Claude for Desktop installed. [You can install the latest version here.](#) If you already have Claude for Desktop, **make sure it's updated to the latest version.**

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/Claude/claude_desktop_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [VS Code](#) installed:

```
bash code ~/Library/Application\ Support/Claude/claude_desktop_config.json powershell code
$env:AppData\Claude\claude_desktop_config.json
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
json java { "mcpServers": { "spring-ai-mcp-weather": { "command":
"java", "args": ["-Dspring.ai.mcp.server.stdio=true", "-
jar", "/ABSOLUTE/PATH/TO/PARENT/FOLDER/mcp-weather-stdio-server-0.0.1-
SNAPSHOT.jar"] } } }
```

```
json java { "mcpServers": { "spring-ai-mcp-weather": { "command":
"java", "args": ["-Dspring.ai.mcp.server.transport=STDIO", "-
jar", "C:\\ABSOLUTE\\PATH\\TO\\PARENT\\FOLDER\\weather\\mcp-weather-stdio-server-0.0.1-
SNAPSHOT.jar"] } } }
```

Make sure you pass in the absolute path to your server.

This tells Claude for Desktop: 1. There's an MCP server named "my-weather-server" 2. To launch it by running `java -jar /ABSOLUTE/PATH/TO/PARENT/FOLDER/mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar`

Save the file, and restart **Claude for Desktop**.

# Testing your server with Java client

## Create a MCP Client manually

Use the `McpClient` to connect to the server:

```
var stdioParams = ServerParameters.builder("java")
 .args("-jar", "/ABSOLUTE/PATH/TO/PARENT/FOLDER/mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar")
 .build();

var stdioTransport = new StdioClientTransport(stdioParams);

var mcpClient = McpClient.sync(stdioTransport).build();

mcpClient.initialize();

ListToolsResult toolsList = mcpClient.listTools();

CallToolResult weather = mcpClient.callTool(
 new CallToolRequest("getWeatherForecastByLocation",
 Map.of("latitude", "47.6062", "longitude", "-122.3321")));

CallToolResult alert = mcpClient.callTool(
```

```
new CallToolRequest("getAlerts", Map.of("state", "NY")));

mcpClient.closeGracefully();
```

## Use MCP Client Boot Starter

Create a new boot starter application using the `spring-ai-starter-mcp-client` dependency:

```
<dependency>
 <groupId>org.springframework.ai</groupId>
 <artifactId>spring-ai-starter-mcp-client</artifactId>
</dependency>
```

and set the `spring.ai.mcp.client.stdio.servers-configuration` property to point to your `claude_desktop_config.json`. You can re-use the existing Anthropic Desktop configuration:

```
spring.ai.mcp.client.stdio.servers-configuration=file:PATH/T0/claude_desktop_config.json
```

When you start your client application, the auto-configuration will create, automatically MCP clients from the `claude_desktop_config.json`.

For more information, see the [MCP Client Boot Starters](#) reference documentation.

## More Java MCP Server examples

The [starter-webflux-server](#) demonstrates how to create a MCP server using SSE transport. It showcases how to define and register MCP Tools, Resources, and Prompts, using the Spring Boot's auto-configuration capabilities.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

## Prerequisite knowledge

This quickstart assumes you have familiarity with: - Kotlin - LLMs like Claude

## System requirements

- Java 17 or higher installed.

## Set up your environment

First, let's install `java` and `gradle` if you haven't already. You can download `java` from [official Oracle JDK website](#). Verify your `java` installation:

```
java --version
```

Now, let's create and set up your project:

```
```bash MacOS/Linux # Create a new directory for our project mkdir weather cd weather
```

Initialize a new kotlin project

gradle init

```
```powershell Windows
Create a new directory for our project
md weather
cd weather

Initialize a new kotlin project
gradle init
```

After running `gradle init`, you will be presented with options for creating your project. Select **Application** as the project type, **Kotlin** as the programming language, and **Java 17** as the Java version.

Alternatively, you can create a Kotlin application using the [IntelliJ IDEA project wizard](#).

After creating the project, add the following dependencies: ```kotlin build.gradle.kts val mcpVersion = "0.4.0" val slf4jVersion = "2.0.9" val ktorVersion = "3.1.1"

```
dependencies { implementation("io.modelcontextprotocol:kotlin-sdk:$mcpVersion")
implementation("org.slf4j:slf4j-nop:$slf4jVersion") implementation("io.ktor:ktor-client-content-negotiation:$ktorVersion") implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")
}
```

```

```groovy build.gradle
def mcpVersion = '0.3.0'
def slf4jVersion = '2.0.9'
def ktorVersion = '3.1.1'

dependencies {
    implementation "io.modelcontextprotocol:kotlin-sdk:$mcpVersion"
    implementation "org.slf4j:slf4j-nop:$slf4jVersion"
    implementation "io.ktor:ktor-client-content-negotiation:$ktorVersion"
    implementation "io.ktor:ktor-serialization-kotlinx-json:$ktorVersion"
}

```

Also, add the following plugins to your build script: `kotlin build.gradle.kts plugins {`
`kotlin("plugin.serialization") version "your_version_of_kotlin"`
`id("com.github.johnrengelman.shadow") version "8.1.1" }`

```

groovy build.gradle plugins {      id 'org.jetbrains.kotlin.plugin.serialization' version
'your_version_of_kotlin'      id 'com.github.johnrengelman.shadow' version '8.1.1' }

```

Now let's dive into building your server.

Building your server

Setting up the instance

Add a server initialization function:

```

// Main function to run the MCP server
fun `run mcp server`() {
    // Create the MCP Server instance with a basic implementation
    val server = Server(
        Implementation(
            name = "weather", // Tool name is "weather"
            version = "1.0.0" // Version of the implementation
        ),
        ServerOptions(
            capabilities = ServerCapabilities(tools = ServerCapabilities.Tools(listChanged =
true))
        )
    )
}

```

```

)

// Create a transport using standard IO for server communication
val transport = StdioServerTransport(
    System.`in`.asInput(),
    System.out.asSink().buffered()
)

runBlocking {
    server.connect(transport)
    val done = Job()
    server.onClose {
        done.complete()
    }
    done.join()
}
}

```

Weather API helper functions

Next, let's add functions and data classes for querying and converting responses from the National Weather Service API:

```

// Extension function to fetch forecast information for given latitude and longitude
suspend fun HttpClient.getForecast(latitude: Double, longitude: Double): List<String> {
    val points = this.get("/points/$latitude,$longitude").body<Points>()
    val forecast = this.get(points.properties.forecast).body<Forecast>()
    return forecast.properties.periods.map { period ->
        """
            ${period.name}:
            Temperature: ${period.temperature} ${period.temperatureUnit}
            Wind: ${period.windSpeed} ${period.windDirection}
            Forecast: ${period.detailedForecast}
        """.trimIndent()
    }
}

// Extension function to fetch weather alerts for a given state
suspend fun HttpClient.getAlerts(state: String): List<String> {

```

```

val alerts = this.get("/alerts/active/area/$state").body<Alert>()
return alerts.features.map { feature ->
    """
        Event: ${feature.properties.event}
        Area: ${feature.properties.areaDesc}
        Severity: ${feature.properties.severity}
        Description: ${feature.properties.description}
        Instruction: ${feature.properties.instruction}
    """.trimIndent()
}
}

@Serializable
data class Points(
    val properties: Properties
) {
    @Serializable
    data class Properties(val forecast: String)
}

@Serializable
data class Forecast(
    val properties: Properties
) {
    @Serializable
    data class Properties(val periods: List<Period>)

    @Serializable
    data class Period(
        val number: Int, val name: String, val startTime: String, val endTime: String,
        val isDaytime: Boolean, val temperature: Int, val temperatureUnit: String,
        val temperatureTrend: String, val probabilityOfPrecipitation: JsonObject,
        val windSpeed: String, val windDirection: String,
        val shortForecast: String, val detailedForecast: String,
    )
}

@Serializable
data class Alert(

```

```

    val features: List<Feature>
) {
    @Serializable
    data class Feature(
        val properties: Properties
    )

    @Serializable
    data class Properties(
        val event: String, val areaDesc: String, val severity: String,
        val description: String, val instruction: String?,
    )
}

```

Implementing tool execution

The tool execution handler is responsible for actually executing the logic of each tool. Let's add it:

```

// Create an HTTP client with a default request configuration and JSON content negotiation
val httpClient = HttpClient {
    defaultRequest {
        url("https://api.weather.gov")
        headers {
            append("Accept", "application/geo+json")
            append("User-Agent", "WeatherApiClient/1.0")
        }
        contentType(ContentType.Application.Json)
    }
    // Install content negotiation plugin for JSON serialization/deserialization
    install(ContentNegotiation) { json(Json { ignoreUnknownKeys = true }) }
}

// Register a tool to fetch weather alerts by state
server.addTool(
    name = "get_alerts",
    description = """
        Get weather alerts for a US state. Input is Two-letter US state code (e.g. CA, NY)
    """.trimIndent(),
    inputSchema = Tool.Input(

```

```

        properties = buildJsonObject {
            putJsonObject("state") {
                put("type", "string")
                put("description", "Two-letter US state code (e.g. CA, NY)")
            }
        },
        required = listOf("state")
    )
} { request ->
    val state = request.arguments["state"]?.jsonPrimitive?.content
    if (state == null) {
        return@addTool CallToolResult(
            content = listOf(TextContent("The 'state' parameter is required."))
        )
    }

    val alerts = httpClient.getAlerts(state)

    CallToolResult(content = alerts.map { TextContent(it) })
}

// Register a tool to fetch weather forecast by latitude and longitude
server.addTool(
    name = "get_forecast",
    description = """
        Get weather forecast for a specific latitude/longitude
    """.trimIndent(),
    inputSchema = Tool.Input(
        properties = buildJsonObject {
            putJsonObject("latitude") { put("type", "number") }
            putJsonObject("longitude") { put("type", "number") }
        },
        required = listOf("latitude", "longitude")
    )
} { request ->
    val latitude = request.arguments["latitude"]?.jsonPrimitive?.doubleOrNull
    val longitude = request.arguments["longitude"]?.jsonPrimitive?.doubleOrNull
    if (latitude == null || longitude == null) {
        return@addTool CallToolResult(

```

```
        content = listOf(TextContent("The 'latitude' and 'longitude' parameters are
required."))
    )
}

val forecast = httpClient.getForecast(latitude, longitude)

CallToolResult(content = forecast.map { TextContent(it) })
}
```

Running the server

Finally, implement the main function to run the server:

```
fun main() = `run mcp server`()
```

Make sure to run `./gradlew build` to build your server. This is a very important step in getting your server to connect.

Let's now test your server from an existing MCP host, Claude for Desktop.

Testing your server with Claude for Desktop

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](#) tutorial to build an MCP client that connects to the server we just built.

First, make sure you have Claude for Desktop installed. [You can install the latest version here](#). If you already have Claude for Desktop, **make sure it's updated to the latest version**.

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/Claude/claude_desktop_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [VS Code](#) installed:

```
bash MacOS/Linux code ~/Library/Application\ Support/Claude/claude_desktop_config.json
```

```
powershell Windows code $env:AppData\Claude\claude_desktop_config.json
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
json MacOS/Linux {      "mcpServers": {      "weather": {      "command":  
"java",      "args": [      "-jar",  
"/ABSOLUTE/PATH/TO/PARENT/FOLDER/weather/build/libs/weather-0.1.0-all.jar"      ]  
}      } }
```

```
json Windows {      "mcpServers": {      "weather": {      "command":  
"java",      "args": [      "-jar",  
"C:\\PATH\\TO\\PARENT\\FOLDER\\weather\\build\\libs\\weather-0.1.0-all.jar"      ]  
}      } }
```

This tells Claude for Desktop: 1. There's an MCP server named "weather" 2. Launch it by running

```
java -jar /ABSOLUTE/PATH/TO/PARENT/FOLDER/weather/build/libs/weather-0.1.0-all.jar
```

Save the file, and restart **Claude for Desktop**.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

Prerequisite knowledge

This quickstart assumes you have familiarity with: - C# - LLMs like Claude - .NET 8 or higher

System requirements

- [.NET 8 SDK](#) or higher installed.

Set up your environment

First, let's install `dotnet` if you haven't already. You can download `dotnet` from [official Microsoft .NET website](#). Verify your `dotnet` installation:

```
dotnet --version
```

Now, let's create and set up your project: `bash MacOS/Linux # Create a new directory for our project mkdir weather cd weather # Initialize a new C# project dotnet new console`

powershell Windows # Create a new directory for our project mkdir weather cd weather # Initialize a new C# project dotnet new console After running dotnet new console, you will be presented with a new C# project. You can open the project in your favorite IDE, such as [Visual Studio](#) or [Rider](#).

Alternatively, you can create a C# application using the [Visual Studio project wizard](#). After creating the project, add NuGet package for the Model Context Protocol SDK and hosting:

```
# Add the Model Context Protocol SDK NuGet package
dotnet add package ModelContextProtocol --prerelease
# Add the .NET Hosting NuGet package
dotnet add package Microsoft.Extensions.Hosting
```

Now let's dive into building your server.

Building your server

Open the `Program.cs` file in your project and replace its contents with the following code:

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using ModelContextProtocol;
using System.Net.Http.Headers;

var builder = Host.CreateEmptyApplicationBuilder(settings: null);

builder.Services.AddMcpServer()
    .WithStdioServerTransport()
    .WithToolsFromAssembly();

builder.Services.AddSingleton(_ =>
{
    var client = new HttpClient() { BaseAddress = new Uri("https://api.weather.gov") };
    client.DefaultRequestHeaders.UserAgent.Add(new ProductInfoHeaderValue("weather-tool",
"1.0"));
    return client;
});

var app = builder.Build();

await app.RunAsync();
```

When creating the `ApplicationHostBuilder`, ensure you use `CreateEmptyApplicationBuilder` instead of `CreateDefaultBuilder`. This ensures that the server does not write any additional messages to the console. This is only necessary for servers using STDIO transport.

This code sets up a basic console application that uses the Model Context Protocol SDK to create an MCP server with standard I/O transport.

Weather API helper functions

Next, define a class with the tool execution handlers for querying and converting responses from the National Weather Service API:

```
using ModelContextProtocol.Server;
using System.ComponentModel;
using System.Net.Http.Json;
using System.Text.Json;

namespace QuickstartWeatherServer.Tools;

[McpServerToolType]
public static class WeatherTools
{
    [McpServerTool, Description("Get weather alerts for a US state.")]
    public static async Task<string> GetAlerts(
        HttpClient client,
        [Description("The US state to get alerts for.")] string state)
    {
        var jsonElement = await
client.GetFromJsonAsync<JsonElement>($"/alerts/active/area/{state}");
        var alerts = jsonElement.GetProperty("features").EnumerateArray();

        if (!alerts.Any())
        {
            return "No active alerts for this state.";
        }

        return string.Join("\n--\n", alerts.Select(alert =>
        {
            JsonElement properties = alert.GetProperty("properties");
            return $""
```

```

        Event: {properties.GetProperty("event").GetString()}
        Area: {properties.GetProperty("areaDesc").GetString()}
        Severity: {properties.GetProperty("severity").GetString()}
        Description: {properties.GetProperty("description").GetString()}
        Instruction: {properties.GetProperty("instruction").GetString()}
        """;
    }));
}

[McpServerTool, Description("Get weather forecast for a location.")]
public static async Task<string> GetForecast(
    HttpClient client,
    [Description("Latitude of the location.")] double latitude,
    [Description("Longitude of the location.")] double longitude)
{
    var jsonElement = await
client.GetFromJsonAsync<JsonElement>($"/points/{latitude},{longitude}");
    var periods =
jsonElement.GetProperty("properties").GetProperty("periods").EnumerateArray();

    return string.Join("\n---\n", periods.Select(period => $$$$
        {period.GetProperty("name").GetString()}
        Temperature: {period.GetProperty("temperature").GetInt32()}°F
        Wind: {period.GetProperty("windSpeed").GetString()}
        {period.GetProperty("windDirection").GetString()}
        Forecast: {period.GetProperty("detailedForecast").GetString()}
        """));
}
}

```

Running the server

Finally, run the server using the following command:

```
dotnet run
```

This will start the server and listen for incoming requests on standard input/output.

Testing your server with Claude for Desktop

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](#) tutorial to build an MCP client that connects to the server we just built.

First, make sure you have Claude for Desktop installed. [You can install the latest version here.](#) If you already have Claude for Desktop, **make sure it's updated to the latest version.** We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/Claude/claude_desktop_config.json` in a text editor. Make sure to create the file if it doesn't exist. For example, if you have [VS Code](#) installed:

```
code ~/Library/Application\ Support/Claude/claude_desktop_config.json
```

```
code $env:AppData\Claude\claude_desktop_config.json
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured. In this case, we'll add our single weather server like so:

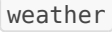
```
{
  "mcpServers": {
    "weather": {
      "command": "dotnet",
      "args": [
        "run",
        "--project",
        "/ABSOLUTE/PATH/TO/PROJECT",
        "--no-build"
      ]
    }
  }
}
```

```
}
```

```
{
  "mcpServers": {
    "weather": {
      "command": "dotnet",
      "args": [
        "run",
        "--project",
        "C:\\\\ABSOLUTE\\\\PATH\\\\T0\\\\PROJECT",
        "--no-build"
      ]
    }
  }
}
```

This tells Claude for Desktop: 1. There's an MCP server named "weather" 2. Launch it by running `dotnet run /ABSOLUTE/PATH/T0/PROJECT` Save the file, and restart **Claude for Desktop**.

Test with commands

Let's make sure Claude for Desktop is picking up the two tools we've exposed in our `weather` server. You can do this by looking for the hammer  icon: `` icon:

visual-indicator-mcp-tools.png

After clicking on the hammer icon, you should see two tools listed:

available-mcp-tools.png

If your server isn't being picked up by Claude for Desktop, proceed to the [Troubleshooting](#) section for debugging tips.

If the hammer icon has shown up, you can now test your server by running the following commands in Claude for Desktop:

- What's the weather in Sacramento?
- What are the active weather alerts in Texas?

current-weather.png weather-alerts.png

Since this is the US National Weather service, the queries will only work for US locations.

What's happening under the hood

When you ask a question:

1. The client sends your question to Claude
2. Claude analyzes the available tools and decides which one(s) to use
3. The client executes the chosen tool(s) through the MCP server
4. The results are sent back to Claude
5. Claude formulates a natural language response
6. The response is displayed to you!

Troubleshooting

Getting logs from Claude for Desktop

Claude.app logging related to MCP is written to log files in `~/Library/Logs/Claude`:

- `mcp.log` will contain general logging about MCP connections and connection failures.
- Files named `mcp-server-SERVERNAME.log` will contain error (stderr) logging from the named server.

You can run the following command to list recent logs and follow along with any new ones:

```
# Check Claude's logs for errors
tail -n 20 -f ~/Library/Logs/Claude/mcp*.log
```

Server not showing up in Claude

1. Check your `claude_desktop_config.json` file syntax
2. Make sure the path to your project is absolute and not relative
3. Restart Claude for Desktop completely

Tool calls failing silently

If Claude attempts to use the tools but they fail:

1. Check Claude's logs for errors
2. Verify your server builds and runs without errors
3. Try restarting Claude for Desktop

None of this is working. What do I do?

Please refer to our [debugging guide](#) for better debugging tools and more detailed guidance. **Error: Failed to retrieve grid point data**

This usually means either: 1. The coordinates are outside the US 2. The NWS API is having issues 3. You're being rate limited

Fix:

- Verify you're using US coordinates
- Add a small delay between requests
- Check the NWS API status page

Error: No active alerts for [STATE]

This isn't an error - it just means there are no current weather alerts for that state. Try a different state or check during severe weather.

For more advanced troubleshooting, check out our guide on [Debugging MCP](#)

Next steps

<Card title="Building a client" icon="outlet" href="/quickstart/client"

“ Learn how to build your own MCP client that can connect to your server <Card title="Example servers" icon="grid" href="/examples" Check out our gallery of official MCP servers and implementations <Card title="Debugging Guide" icon="bug" href="/docs/tools/debugging" Learn how to effectively debug MCP servers and integrations <Card title="Building MCP with LLMs" icon="comments" href="/tutorials/building-mcp-with-llms" Learn how to use LLMs like Claude to speed up your MCP development

```
<!-- page slug: mcp-docs-quickstart-user | order: 15 -->
```

title: "For Claude Desktop Users" description: "Get started using pre-built servers in Claude for Desktop." —

In this tutorial, you will extend [Claude for Desktop](#) so that it can read from your computer's file system, write new files, move files, and even search files.

quickstart-filesystem.png

Don't worry — it will ask you for your permission before executing these actions!

1. Download Claude for Desktop

Start by downloading [Claude for Desktop](#), choosing either macOS or Windows. (Linux is not yet supported for Claude for Desktop.)

Follow the installation instructions.

If you already have Claude for Desktop, make sure it's on the latest version by clicking on the Claude menu on your computer and selecting "Check for Updates..."

Because servers are locally run, MCP currently only supports desktop hosts. Remote hosts are in active development.

2. Add the Filesystem MCP Server

To add this filesystem functionality, we will be installing a pre-built [Filesystem MCP Server](#) to Claude for Desktop. This is one of dozens of [servers](#) created by Anthropic and the community.

Get started by opening up the Claude menu on your computer and select “Settings...” Please note that these are not the Claude Account Settings found in the app window itself.

This is what it should look like on a Mac: <Frame style={{ textAlign: 'center' }}>
quickstart-menu.png

Click on “Developer” in the left-hand bar of the Settings pane, and then click on “Edit Config”:
quickstart-developer.png

This will create a configuration file at: - macOS: ~/Library/Application
Support/Claude/claude_desktop_config.json - Windows: %APPDATA%\Claude\claude_desktop_config.json

if you don’t already have one, and will display the file in your file system.

Open up the configuration file in any text editor. Replace the file contents with this:

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "/Users/username/Desktop",
        "/Users/username/Downloads"
      ]
    }
  }
}
```

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "C:\\Users\\username\\Desktop",
        "C:\\Users\\username\\Downloads"
      ]
    }
  }
}
```

```
}  
}  
}
```

Make sure to replace `username` with your computer's username. The paths should point to valid directories that you want Claude to be able to access and modify. It's set up to work for Desktop and Downloads, but you can add more paths as well.

You will also need [Node.js](#) on your computer for this to run properly. To verify you have Node installed, open the command line on your computer. - On macOS, open the Terminal from your Applications folder - On Windows, press Windows + R, type "cmd", and press Enter

Once in the command line, verify you have Node installed by entering in the following command:

```
node --version
```

If you get an error saying "command not found" or "node is not recognized", download Node from nodejs.org.

How does the configuration file work?

This configuration file tells Claude for Desktop which MCP servers to start up every time you start the application. In this case, we have added one server called "filesystem" that will use the Node `npx` command to install and run `@modelcontextprotocol/server-filesystem`. This server, described [here](#), will let you access your file system in Claude for Desktop.

Command Privileges

Claude for Desktop will run the commands in the configuration file with the permissions of your user account, and access to your local files. Only add commands if you understand and trust the source.

3. Restart Claude

After updating your configuration file, you need to restart Claude for Desktop.

Upon restarting, you should see a hammer  icon in the bottom right corner of the input box:

quickstart-hammer.png

After clicking on the hammer icon, you should see the tools that come with the Filesystem MCP Server:

<Frame style={{ textAlign: 'center' }}> quickstart-tools.png

If your server isn't being picked up by Claude for Desktop, proceed to the [Troubleshooting](#) section for debugging tips.

4. Try it out!

You can now talk to Claude and ask it about your filesystem. It should know when to call the relevant tools.

Things you might try asking Claude: - Can you write a poem and save it to my desktop? - What are some work-related files in my downloads folder? - Can you take all the images on my desktop and move them to a new folder called "Images"?

As needed, Claude will call the relevant tools and seek your approval before taking an action:

<Frame style={{ textAlign: 'center' }}> quickstart-approve.png

Troubleshooting

1. Restart Claude for Desktop completely 2. Check your `claude_desktop_config.json` file syntax 3. Make sure the file paths included in `claude_desktop_config.json` are valid and that they are absolute and not relative 4. Look at [logs](#) to see why the server is not connecting 5. In your command line, try manually running the server (replacing `username` as you did in `claude_desktop_config.json`) to see if you get any errors:

```
npx -y @modelcontextprotocol/server-filesystem /Users/username/Desktop
/Users/username/Downloads
```

```
npx -y @modelcontextprotocol/server-filesystem C:\Users\username\Desktop
C:\Users\username\Downloads
```

Claude.app logging related to MCP is written to log files in: - macOS: `~/Library/Logs/Claude` - Windows: `%APPDATA%\Claude\logs`

- `mcp.log` will contain general logging about MCP connections and connection failures.

- Files named `mcp-server-SERVERNAME.log` will contain error (stderr) logging from the named server.

You can run the following command to list recent logs and follow along with any new ones (on Windows, it will only show recent logs):

```
# Check Claude's logs for errors
tail -n 20 -f ~/Library/Logs/Claude/mcp*.log
```

```
type "%APPDATA%\Claude\logs\mcp*.log"
```

If Claude attempts to use the tools but they fail:

1. Check Claude's logs for errors
2. Verify your server builds and runs without errors
3. Try restarting Claude for Desktop Please refer to our [debugging guide](#) for better debugging tools and more detailed guidance. If your configured server fails to load, and you see within its logs an error referring to `${APPDATA}` within a path, you may need to add the expanded value of `%APPDATA%` to your `env` key in `claude_desktop_config.json`:

```
{
  "brave-search": {
    "command": "npx",
    "args": ["-y", "@modelcontextprotocol/server-brave-search"],
    "env": {
      "APPDATA": "C:\\Users\\user\\AppData\\Roaming\\",
      "BRAVE_API_KEY": "..."
    }
  }
}
```

With this change in place, launch Claude Desktop once again.

NPM should be installed globally

The `npx` command may continue to fail if you have not installed NPM globally. If NPM is already installed globally, you will find `%APPDATA%\npm` exists on your system. If not, you can install NPM globally by running the following command:

```
npm install -g npm
```

Next steps

<Card title="Explore other servers" icon="grid" href="/examples"

“ Check out our gallery of official MCP servers and implementations <Card title="Build your own server" icon="code" href="/quickstart/server" Now build your own custom server to use in Claude for Desktop and other clients