Quickstart

For Client Developers

Copy page

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.

Python Node Java Kotlin C#

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:

```python
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(st
        self.stdio, self.write = stdio_transport
        self.session = await self.exit_stack.enter_async_context(Clien

        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
```

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_
            final_text.append(f"[Calling tool {tool_name} with arg

            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:

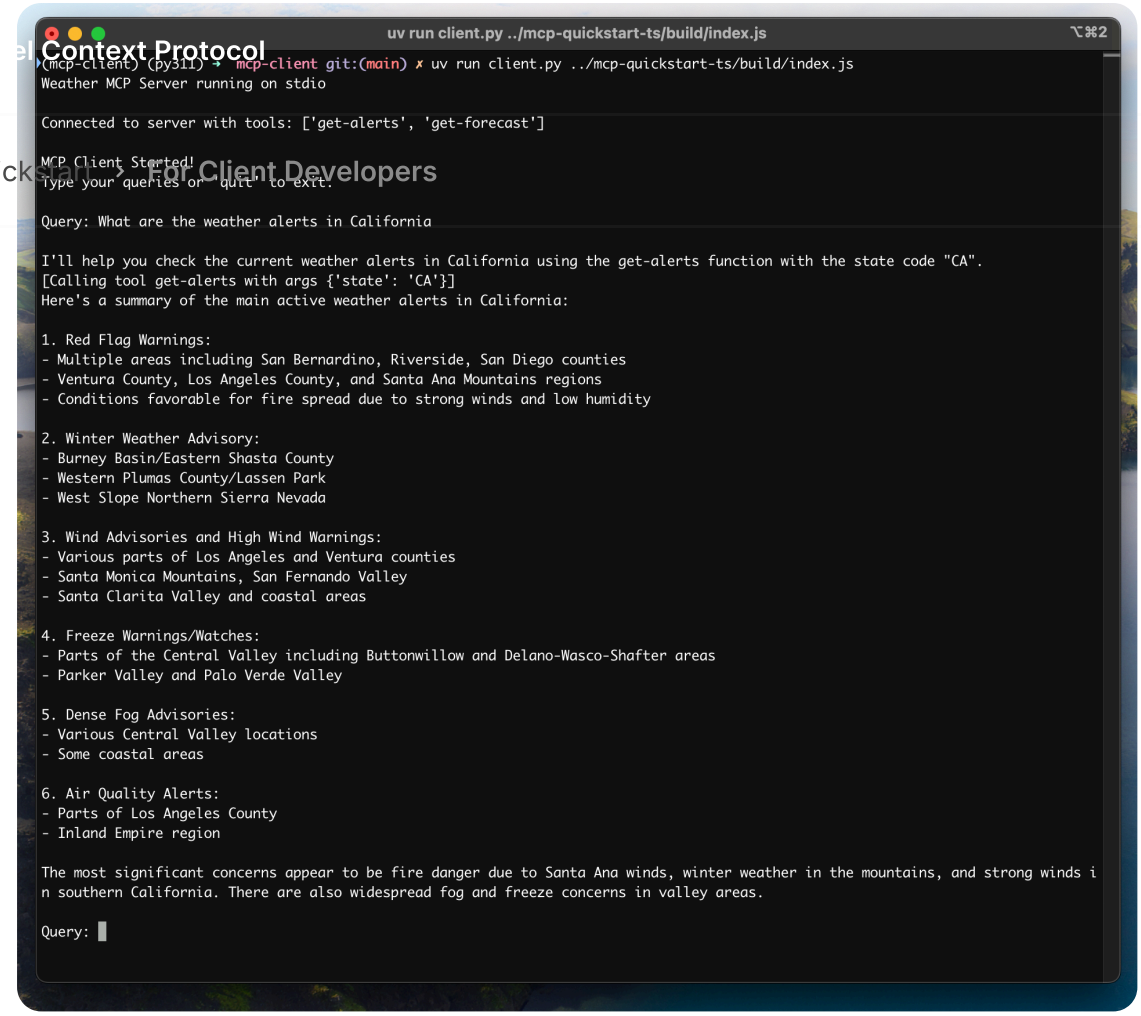
```
uv run client.py path/to/server.py # python server
uv run client.py path/to/build/index.js # node server
```

> 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:



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

Next steps

Example servers

Check out our gallery of official MCP servers and implementations

Clients

View the list of clients that support MCP integrations

Building MCP with LLMs

Learn how to use LLMs like Claude to speed up your MCP development

Core architecture

Understand how MCP connects clients, servers, and LLMs

Was this page helpful? Yes No