20250913a The FastMCP Client

`New in version: `2.0.0`` The central piece of MCP client applications is the `fastmcp.Client` class. This class provides a **programmatic interface** for interacting with any Model Context Protocol (MCP) server, handling protocol details and connection management automatically.The FastMCP Client is designed for deterministic, controlled interactions rather than autonomous behavior, making it ideal for:

- **Testing MCP servers** during development
- **Building deterministic applications** that need reliable MCP interactions
- **Creating the foundation for agentic or LLM-based clients** with structured, type-safe operations All client operations require using the `async with` context manager for proper connection lifecycle management.

This is not an agentic client - it requires explicit function calls and provides direct control over all MCP operations. Use it as a building block for higher-level systems.

Creating a Client

Creating a client is straightforward. You provide a server source and the client automatically infers the appropriate transport mechanism.

```
import asyncio

from fastmcp import Client, FastMCP

# In-memory server (ideal for testing)

server = FastMCP("TestServer")

client = Client(server)

# HTTP server

client = Client("https://example.com/mcp")

# Local Python script

client = Client("my_mcp_server.py")

async def main():
```

```
    async with client:
        # Basic server interaction
        await client.ping()

        # List available operations
        tools = await client.list_tools()
        resources = await client.list_resources()
        prompts = await client.list_prompts()

        # Execute operations
        result = await client.call_tool("example_tool", {"param": "value"})
        print(result)

asyncio.run(main())
```

Client-Transport Architecture

The FastMCP Client separates concerns between protocol and connection:

- **`Client`**: Handles MCP protocol operations (tools, resources, prompts) and manages callbacks
- **`Transport`**: Establishes and maintains the connection (WebSockets, HTTP, Stdio, in-memory)

Transport Inference

The client automatically infers the appropriate transport based on the input:

1. **`FastMCP` instance** → In-memory transport (perfect for testing)
2. **File path ending in `.py`** → Python Stdio transport
3. **File path ending in `.js`** → Node.js Stdio transport
4. **URL starting with `http://` or `https://`** → HTTP transport
5. **`MCPConfig` dictionary** → Multi-server client

```
from fastmcp import Client, FastMCP

# Examples of transport inference

client_memory = Client(FastMCP("TestServer"))

client_script = Client("./server.py")

client_http = Client("https://api.example.com/mcp")
```

For testing and development, always prefer the in-memory transport by passing a `FastMCP` server directly to the client. This eliminates network complexity and separate processes.

Configuration-Based Clients

`New in version: 2.4.0` Create clients from MCP configuration dictionaries, which can include multiple servers. While there is no official standard for MCP configuration format, FastMCP follows established conventions used by tools like Claude Desktop.

Configuration Format

Multi-Server Example

```
config = {

    "mcpServers": {

        "weather": {"url": "https://weather-api.example.com/mcp"},

        "assistant": {"command": "python", "args": ["./assistant_server.py"]}

    }

}

client = Client(config)

async with client:

    # Tools are prefixed with server names

    weather_data = await client.call_tool("weather_get_forecast", {"city":
```

```
"London"})

    response = await client.call_tool("assistant_answer_question", {"question":
"What's the capital of France?"})



    # Resources use prefixed URIs

    icons = await client.read_resource("weather://weather/icons/sunny")

    templates = await client.read_resource("resource://assistant/templates/list")
```

Connection Lifecycle

The client operates asynchronously and uses context managers for connection management:

```
async def example():

    client = Client("my_mcp_server.py")



    # Connection established here

    async with client:

        print(f"Connected: {client.is_connected()}")



        # Make multiple calls within the same session

        tools = await client.list_tools()

        result = await client.call_tool("greet", {"name": "World"})



    # Connection closed automatically here

    print(f"Connected: {client.is_connected()}")
```

Operations

FastMCP clients can interact with several types of server components:

Tools

Tools are server-side functions that the client can execute with arguments.

```
async with client:

    # List available tools

    tools = await client.list_tools()



    # Execute a tool

    result = await client.call_tool("multiply", {"a": 5, "b": 3})

    print(result.data)  # 15
```

See Tools for detailed documentation.

Resources

Resources are data sources that the client can read, either static or templated.

```
async with client:

    # List available resources

    resources = await client.list_resources()



    # Read a resource

    content = await client.read_resource("file:///config/settings.json")

    print(content[0].text)
```

See Resources for detailed documentation.

Prompts

Prompts are reusable message templates that can accept arguments.

```
async with client:

    # List available prompts

    prompts = await client.list_prompts()



    # Get a rendered prompt

    messages = await client.get_prompt("analyze_data", {"data": [1, 2, 3]})

    print(messages.messages)
```

See Prompts for detailed documentation.

Server Connectivity

Use `ping()` to verify the server is reachable:

```
async with client:

    await client.ping()

    print("Server is reachable")
```

Client Configuration

Clients can be configured with additional handlers and settings for specialized use cases.

Callback Handlers

The client supports several callback handlers for advanced server interactions:

```
from fastmcp import Client

from fastmcp.client.logging import LogMessage

async def log_handler(message: LogMessage):
```

```
    print(f"Server log: {message.data}")

async def progress_handler(progress: float, total: float | None, message: str |
None):

    print(f"Progress: {progress}/{total} - {message}")

async def sampling_handler(messages, params, context):

    # Integrate with your LLM service here

    return "Generated response"

client = Client(

    "my_mcp_server.py",

    log_handler=log_handler,

    progress_handler=progress_handler,

    sampling_handler=sampling_handler,

    timeout=30.0

)
```

The `Client` constructor accepts several configuration options:

- `transport`: Transport instance or source for automatic inference
- `log_handler`: Handle server log messages
- `progress_handler`: Monitor long-running operations
- `sampling_handler`: Respond to server LLM requests
- `roots`: Provide local context to servers
- `timeout`: Default timeout for requests (in seconds)

Transport Configuration

For detailed transport configuration (headers, authentication, environment variables), see the Transports documentation.Explore the detailed documentation for each operation type:

Core Operations

- **Tools** - Execute server-side functions and handle results

- **Resources** - Access static and templated resources
- **Prompts** - Work with message templates and argument serialization

Advanced Features

- **Logging** - Handle server log messages
- **Progress** - Monitor long-running operations
- **Sampling** - Respond to server LLM requests
- **Roots** - Provide local context to servers

Connection Details

- **Transports** - Configure connection methods and parameters
- **Authentication** - Set up OAuth and bearer token authentication

The FastMCP Client is designed as a foundational tool. Use it directly for deterministic operations, or build higher-level agentic systems on top of its reliable, type-safe interface.

FastMCP Cloud Transports