python-sdk Tutorial

Welcome to the first chapter of the `python-sdk` tutorial! In this chapter, we'll introduce you to the fundamental concept behind this project: the Model Context Protocol, or MCP for short.

The Problem: LLMs Need to Interact with the World

Imagine you're building an application powered by a large language model (LLM), like a chatbot that can help users manage their tasks, look up information, or perform calculations. LLMs are great at understanding and generating text, but they don't inherently know how to *do* things in the real world, like:

- Fetching the current weather for a city.
- Reading the content of a specific file on your computer.
- Calculating a complex mathematical formula.
- Looking up data in a database.

To do these things, the LLM application needs to interact with external services or data sources.

Without a standard way for the LLM application (the "client") to talk to these external services (the "servers"), you'd have a big problem. Every time you wanted to connect your LLM to a new service, you'd have to write custom code specifically for that service's API. This would be like needing a different type of plug and socket for every single appliance in your house!

The Solution: The Model Context Protocol (MCP)

This is where the Model Context Protocol (MCP) comes in. Think of MCP as a **standard language** or a **common API specification** that allows LLM applications (clients) and external services (servers) to understand each other.

Instead of building custom integrations for every service, the LLM client learns to speak MCP, and the external services learn to speak MCP. Once they both speak the same language, they can communicate seamlessly.

The core idea is that MCP defines *how* an LLM client can:

1. **Discover** what capabilities an external service offers.
2. **Interact** with those capabilities to get data or perform actions.

This allows different LLM applications to use the same external service, and a single LLM application to use many different services, all speaking the same standard protocol.

What Can an MCP Server Provide?

An external service that speaks MCP is called an **MCP Server**. An MCP Server can expose different types of capabilities to the LLM client:

- **Resources:** These are like pieces of data or information that the server can provide. The client can "read" a resource to get its content. Think of reading a file, getting configuration data, or fetching the result of a database query. We'll cover Resources in Chapter 2.
- **Tools:** These are actions or functions that the server can perform. The client can "call" a tool, often providing arguments, and the server executes the action and returns a result. Think of performing a calculation, sending an email, or interacting with an external API. We'll cover Tools in Chapter 3.
- **Prompts:** These are predefined templates for interacting with the LLM. The server can provide these templates to guide the client on how to structure a conversation or request for a specific task. We'll cover Prompts in Chapter 4.

By defining these standard ways of exposing data (Resources), actions (Tools), and interaction patterns (Prompts), MCP provides a structured framework for LLM-service communication.

How Does the Communication Happen?

At a high level, the communication between an MCP Client and an MCP Server follows a simple pattern:

1. **Initialization:** When a client connects to a server, they first exchange messages to agree on the protocol version and inform each other about their capabilities (what they can do).
2. **Capability Discovery:** The client can ask the server for lists of its available Resources, Tools, and Prompts.
3. **Interaction:** The client sends requests to the server to read specific Resources, call specific Tools, or get the content of specific Prompts.
4. **Responses & Notifications:** The server sends back responses containing the requested data or results. The server can also send notifications to the client about events, like a resource being updated or progress on a long-running task.

This message exchange happens over a communication channel, which the protocol calls a "transport". Common transports include standard input/output (stdio) or network connections (like Server-Sent Events over HTTP). We'll discuss Transports in a later chapter.

Here's a simplified look at a basic interaction flow:

```
sequenceDiagram
    participant Client
    participant Server

    Client->>Server: initialize request (Hey, I'm here! What can you do?)
    Server->>Client: initialize response (Hi! I speak MCP version X and can do Y, Z, W...)
```

```
Client->>Server: tools/list request (Okay, tell me about your tools.)
Server->>Client: tools/list response (Here's a list of tools you can call...)
Client->>Server: call_tool request (Cool, please run tool "add" with arguments a=2, b=3)
Server->>Client: call_tool response (Okay, the result is 5)
```

MCP in the `python-sdk`

The `python-sdk` project provides the tools you need to build both MCP Clients and MCP Servers using Python. It handles all the complexities of the MCP message format, the communication over transports, and the lifecycle of a connection.

You don't need to worry about the low-level details of sending and receiving JSON-RPC messages. The SDK provides high-level abstractions that allow you to focus on defining your server's capabilities (Resources, Tools, Prompts) or using a server's capabilities from a client perspective.

For example, creating a basic MCP server that understands the protocol is as simple as:

```python
# server.py
from mcp.server.fastmcp import FastMCP

# Create an MCP server instance named "My First Server"
mcp = FastMCP("My First Server")

# Now you can add resources, tools, and prompts to this server object
# (We'll see how in the next chapters!)
```

This `mcp` object is your gateway to building an MCP server. The `FastMCP` class (which we'll explore more in Chapter 6: FastMCP Server) takes care of implementing the MCP specification.

Similarly, connecting to an MCP server from a client perspective using the SDK looks like this (we'll cover clients in Chapter 5: Client Session):

```python
# client.py
import asyncio
from mcp import ClientSession
from mcp.client.stdio import stdio_client
from mcp import StdioServerParameters # To define how to start the server process

async def main():
    # Define how to start the server process (e.g., run a Python script)
    server_params = StdioServerParameters(
        command="python",
        args=["server.py"], # Replace with your server script name
    )
```

```
    # Use the stdio_client to connect and get read/write streams
    async with stdio_client(server_params) as (read_stream, write_stream):
        # Create a ClientSession using the streams
        async with ClientSession(read_stream, write_stream) as session:
            # The session object allows you to interact with the server
            await session.initialize() # Start the MCP handshake
            print("Client initialized!")

            # Now you can call methods like session.list_tools(), session.read_resource(), e
            # (We'll see how in Chapter 5!)

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

This `session` object handles sending and receiving MCP messages, allowing you to interact with the server's capabilities using simple Python method calls.

The key takeaway is that the `python-sdk` abstracts away the low-level protocol details, letting you build powerful LLM applications and services by focusing on *what* data and functionality you want to expose or use, rather than *how* the messages are formatted and sent.

Conclusion

In this chapter, we learned that the Model Context Protocol (MCP) is a standard communication language between LLM applications (clients) and external services (servers). It solves the problem of custom integrations by providing a unified way for clients to discover and interact with server capabilities like Resources, Tools, and Prompts. The `python-sdk` provides the tools to easily build components that speak this protocol.

Now that we understand the basic concept of MCP, let's dive into one of the core capabilities a server can provide: Resources.

Next Chapter: Resources

Welcome back! In Chapter 1: MCP Protocol, we got a high-level overview of the Model Context Protocol (MCP) and how it helps LLM applications (clients) talk to external services (servers) using a standard language. We learned that servers can expose different capabilities, including Resources, Tools, and Prompts.

In this chapter, we'll dive deeper into the first of these core capabilities: **Resources**.

What Problem Do Resources Solve?

Imagine you're building an LLM application that needs to access information from your computer or another service. Maybe it needs to:

- Read the content of a specific configuration file.
- Fetch the latest stock price for a company.
- Get a list of files in a directory.
- Retrieve a piece of static data, like a welcome message.

How does the LLM client get this information from the server? This is where **Resources** come in.

Think of Resources as **read-only endpoints** on the server that expose data or information. They are like the `GET` requests you might use in a web API to fetch data. The client can ask the server for a list of available resources and then request the content of a specific resource.

What is a Resource?

In MCP, a Resource is a piece of data or information that a server makes available to a client.

- **Identified by a URI:** Each resource has a unique identifier, a URI (Uniform Resource Identifier). This is similar to a web address (URL) but can use different schemes (like `file://`, `config://`, `data://`, etc.) depending on what the server defines.
- **Read-Only:** Clients can *read* the content of a resource, but they don't typically *modify* it through the resource mechanism itself.
- **Can be Static or Dynamic:** A resource can represent data that never changes (like a configuration string) or data that is generated on the fly (like the current weather or a personalized greeting).
- **Can Represent Different Things:** Resources can expose simple text, binary data, file contents, directory listings, or even the result of calling an internal function.

The key idea is that the server *exposes* this information, and the client *requests* it using standard MCP messages.

How Clients Interact with Resources

An MCP client interacts with resources using specific requests defined by the protocol:

1. **`resources/list`:** The client sends this request to ask the server for a list of all the resources it currently offers. The server responds with metadata about each resource (its URI, name, description, MIME type).
2. **`resources/read`:** Once the client knows the URI of a resource it wants, it sends this request to get the actual content of that resource. The server responds with the content (as text or binary data) and its MIME type.
3. **`resources/templates/list`:** Servers can also expose *resource templates* for URIs that contain variables (like `user://{user_id}/profile`). The

client uses this request to discover these templates and understand what dynamic resources are available.

We'll see how to use these client requests in Chapter 5: Client Session. For now, let's focus on how to *define* resources on the server side using the `python-sdk`.

Defining Resources with FastMCP

The `python-sdk`'s `FastMCP` server makes defining resources very easy, especially for beginners. You primarily use the `@mcp.resource()` decorator.

Let's look at some examples using the `FastMCP` object we introduced in Chapter 1:

```
# server.py
from mcp.server.fastmcp import FastMCP

# Create an MCP server instance
mcp = FastMCP("My Resource Server")

# ... add resources here ...

# (Optional) Run the server directly for testing
if __name__ == "__main__":
    mcp.run()
```

Example 1: A Simple Static Resource

Let's create a resource that provides a fixed configuration string.

```
# server.py (continued)
# ... (FastMCP setup from above) ...

@mcp.resource("config://app")
def get_config() -> str:
    """Provides the application configuration."""
    return "API_KEY=abcdef123\nDEBUG=True"

# ... (Optional run block) ...
```

- We use the `@mcp.resource("config://app")` decorator. This tells `FastMCP` to register the function `get_config` as a resource accessible at the URI `config://app`.
- The function `get_config` simply returns a string. When a client reads `config://app`, the server will call this function and return its output as the resource content.
- The docstring `"Provides the application configuration."` is automatically used as the resource's description when the client lists resources.

When a client sends a `resources/list` request, the server will include information about `config://app`. When the client sends a `resources/read` request for `config://app`, the server will execute `get_config()` and return `"API_KEY=abcdef123\nDEBUG=True"`.

Example 2: A Dynamic Resource with Parameters

Resources can also be dynamic and depend on parts of the URI. This is similar to path parameters in web APIs (like `/users/{user_id}`).

```
# server.py (continued)
# ... (FastMCP setup and static resource from above) ...

@mcp.resource("greeting://{name}")
def get_greeting(name: str) -> str:
    """Get a personalized greeting."""
    return f"Hello, {name}!"

# ... (Optional run block) ...
```

- The URI template `"greeting://{name}"` includes a placeholder `{name}`.
- The function `get_greeting` takes a parameter `name` with the same name as the placeholder in the URI.
- When a client reads a URI like `greeting://Alice`, `FastMCP` extracts `Alice` from the URI and passes it as the `name` argument to the `get_greeting` function. The function then returns `"Hello, Alice!"`.

This pattern allows you to create many related resources using a single function, like `user://{user_id}/profile` or `data://items/{item_id}`.

When a client sends `resources/list`, the server will list `greeting://{name}` as a **Resource Template**. The client can then use `resources/templates/list` to get more details about how to use this template. When the client sends `resources/read` for a specific URI like `greeting://Bob`, the server uses the template to figure out which function to call and what arguments to pass.

Example 3: A Resource Representing a File

A common use case is exposing file contents. MCP has a standard `file://` URI scheme for this.

```
# server.py (continued)
# ... (FastMCP setup and other resources) ...

import os

# Create a dummy file for demonstration
dummy_file_path = os.path.abspath("example.txt")
with open(dummy_file_path, "w") as f:
```

```
    f.write("This is the content of example.txt")

@mcp.resource(f"file://{dummy_file_path}")
def read_example_file() -> str:
    """Provides the content of example.txt."""
    # The FunctionResource handles reading the file content when read() is called
    # We just need to provide the path.
    # FastMCP automatically wraps this function in a suitable Resource type.
    with open(dummy_file_path, "r") as f:
        return f.read()

# ... (Optional run block) ...
```

- We use a `file://` URI pointing to an absolute path on the server's file system.
- The function `read_example_file` opens and reads the file.
- When a client reads this resource, the server executes `read_example_file` and returns the file content.

While you *can* read files this way, the `python-sdk` also provides built-in `FileResource` and `DirectoryResource` types for more robust handling, which `FastMCP` might use internally or you could use with the lower-level `Server` class. The `@mcp.resource` decorator often intelligently wraps your function in the most appropriate internal `Resource` type (like `FunctionResource`).

Behind the Scenes: How Resources Work

Let's peek under the hood to understand what happens when a client interacts with resources.

When you use `@mcp.resource`, `FastMCP` doesn't immediately read the content. Instead, it registers information about the resource (its URI, the function to call, description, etc.) with its internal **`ResourceManager`**.

Here's a simplified flow for a `resources/read` request:

```
sequenceDiagram
    participant Client
    participant Server as FastMCP Server
    participant RM as ResourceManager
    participant ResourceObj as Specific Resource Object
    participant YourCode as Your @mcp.resource function

    Client->>Server: resources/read request (uri="greeting://Alice")
    Server->>RM: Handle read request for "greeting://Alice"
    RM->>RM: Look up resource/template for "greeting://"
    RM->>RM: Match URI "greeting://Alice" to template "greeting://{name}"
    RM->>RM: Extract parameter "name"="Alice"
```

```
RM->>ResourceObj: Create/Find FunctionResource object for get_greeting
ResourceObj->>YourCode: Call get_greeting(name="Alice")
YourCode-->>ResourceObj: Return "Hello, Alice!"
ResourceObj-->>RM: Return content "Hello, Alice!" and mime_type
RM-->>Server: Return content and mime_type
Server-->>Client: resources/read response (content="Hello, Alice!", mimeType="text/plain
```

1. **Client Request:** The client sends a JSON-RPC request like `{"jsonrpc": "2.0", "id": 1, "method": "resources/read", "params": {"uri": "greeting://Alice"}}`.
2. **Server Receives:** The `FastMCP` server receives this message via the chosen Transport (like stdio or SSE).
3. **Routing:** `FastMCP` recognizes the `resources/read` method and routes it to its internal handler for reading resources.
4. **`ResourceManager`:** This handler uses the `ResourceManager` to find the resource corresponding to the requested URI. If it's a template URI, the `ResourceManager` matches the URI to a registered template and extracts the parameters (like `name="Alice"`).
5. **Resource Object:** The `ResourceManager` finds or creates the appropriate internal `Resource` object (like a `FunctionResource` for our `@mcp.resource` examples).
6. **`read()` Method:** The server calls the `read()` method on that specific `Resource` object. This is where the actual work happens.
7. **Your Code Runs:** If it's a `FunctionResource` wrapping your `@mcp.resource` function, the `read()` method calls your function (`get_greeting("Alice")`).
8. **Content Returned:** Your function returns the content (`"Hello, Alice!"`). The `read()` method returns this content (as `str` or `bytes`).
9. **Response:** The `ResourceManager` and `FastMCP` server format the content and MIME type into an MCP response message and send it back to the client.

This layered approach means you, as the server developer, only need to provide the function that generates or retrieves the data. `FastMCP` and the `ResourceManager` handle the protocol details, URI matching, parameter extraction, and content formatting.

Looking at the code structure (you don't need to understand all of this yet, just see the pieces):

- `src/mcp/server/fastmcp/resources/base.py`: Defines the base `Resource` class with the abstract `read()` method. All specific resource types inherit from this.

```python
# Simplified snippet from base.py
class Resource(BaseModel, abc.ABC):
    uri: AnyUrl = Field(...)
    name: str | None = None
```

```python
    description: str | None = None
    mime_type: str = Field(...)

    @abc.abstractmethod
    async def read(self) -> str | bytes:
        """Read the resource content."""
        pass
```

- `src/mcp/server/fastmcp/resources/types.py`: Contains concrete implementations like `TextResource`, `BinaryResource`, `FileResource`, `FunctionResource`, etc. Each implements the `read()` method differently.

```python
# Simplified snippet from types.py
class FunctionResource(Resource):
    fn: Callable[[], Any] = Field(exclude=True)

    async def read(self) -> str | bytes:
        """Read the resource by calling the wrapped function."""
        # Calls self.fn() and handles async/sync, JSON conversion, etc.
        ...
```

- `src/mcp/server/fastmcp/server.py`: The `FastMCP` class uses the `@mcp.resource` decorator to register your functions. It creates `FunctionResource` objects or `ResourceTemplate` entries and adds them to the `ResourceManager`. It also implements the `list_resources`, `list_resource_templates`, and `read_resource` handlers that interact with the `ResourceManager`.

This structure keeps the core logic of *how* to read different types of data separate from the MCP protocol handling.

Conclusion

Resources are a fundamental concept in MCP for exposing data and information from a server to a client. They are identified by URIs and can represent static or dynamic content, including files. The `python-sdk`'s `FastMCP` makes defining resources easy using the `@mcp.resource` decorator, handling the complexities of URI matching, parameter extraction, and content delivery via its internal `ResourceManager`.

Clients use `resources/list`, `resources/templates/list`, and `resources/read` requests to discover and retrieve resource content.

Now that we know how to expose data using Resources, let's move on to exposing actions and functionality using **Tools**.

Next Chapter: Tools

Welcome back to the `python-sdk` tutorial! In Chapter 1: MCP Protocol, we learned about the Model Context Protocol (MCP) as a standard way for LLM

applications (clients) and external services (servers) to communicate. In Chapter 2: Resources, we explored how MCP servers can expose data and information using Resources, which are primarily for *reading* content via URIs.

Now, let's dive into the next core capability an MCP server can provide: **Tools**.

What Problem Do Tools Solve?

While Resources are great for getting information *into* the LLM's context (like reading a file or fetching configuration), LLMs often need to *do* things in the real world. They might need to:

- Perform a calculation.
- Send an email or message.
- Interact with an external API (like fetching live weather data or looking up a stock price).
- Modify data in a database or file system.
- Trigger a complex process.

LLMs themselves don't have these capabilities built-in. They need a way to ask an external service (the MCP server) to perform these actions on their behalf. This is exactly what **Tools** are for.

Think of Tools as the **action-oriented endpoints** on your server. They are designed to perform computations, interact with external systems, or have side effects. They are like the `POST`, `PUT`, or `DELETE` requests you might use in a web API to trigger actions.

What is a Tool?

In MCP, a Tool is a defined piece of functionality on the server that a client (typically an LLM application) can invoke.

- **Identified by a Name:** Unlike Resources which use URIs, Tools are identified by a simple, descriptive name (e.g., `"add"`, `"fetch_weather"`, `"send_email"`).
- **Takes Parameters:** Tools are defined with specific input parameters, just like arguments to a function. The client provides values for these parameters when calling the tool.
- **Returns a Result:** After executing, a Tool returns a result back to the client. This result can be text, structured data, or even references to Resources or Images.
- **Can Have Side Effects:** The key difference from Resources is that Tools are *expected* to perform actions that might change the state of the server or external systems.
- **Clients Discover and Call:** Clients use specific MCP requests (`tools/list` and `tools/call`) to find out what tools are available and then execute them.

The core idea is that the server *exposes* these callable actions, and the client *requests* their execution using standard MCP messages, providing the necessary inputs.

How Clients Interact with Tools

An MCP client interacts with tools using specific requests defined by the protocol:

1. **`tools/list`:** The client sends this request to ask the server for a list of all the tools it currently offers. The server responds with metadata about each tool (its name, description, and a schema describing its input parameters).
2. **`tools/call`:** Once the client knows the name of a tool and understands its parameters, it sends this request to execute the tool, providing the required arguments. The server executes the tool function and returns the result.

We'll see how to use these client requests in Chapter 5: Client Session. For now, let's focus on how to *define* tools on the server side using the `python-sdk`.

Defining Tools with FastMCP

Just like with Resources, the `python-sdk`'s `FastMCP` server makes defining tools very easy using the `@mcp.tool()` decorator.

Let's use the `FastMCP` object we introduced in Chapter 1 and used in Chapter 2:

```
# server.py
from mcp.server.fastmcp import FastMCP

# Create an MCP server instance
mcp = FastMCP("My Tool Server")

# ... add tools here ...

# (Optional) Run the server directly for testing
if __name__ == "__main__":
    mcp.run()
```

Example 1: A Simple Calculation Tool

Let's create a tool that adds two numbers. This is a classic example of a tool that performs a computation.

```
# server.py (continued)
# ... (FastMCP setup from above) ...
```

```python
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

# ... (Optional run block) ...
```

- We use the `@mcp.tool()` decorator. This tells `FastMCP` to register the function `add` as a tool.
- By default, the tool's name will be the function's name: `"add"`.
- The function `add` takes two parameters, `a` and `b`, with type hints `int`. `FastMCP` uses these type hints to automatically generate the `inputSchema` for the tool, telling the client that this tool expects two integer arguments named `a` and `b`.
- The function returns an `int`. The return value is sent back to the client as the result of the tool call.
- The docstring `"Add two numbers"` is automatically used as the tool's description when the client lists tools.

When a client sends a `tools/list` request, the server will include information about the `"add"` tool, including its description and input schema (which will indicate it takes `a: int` and `b: int`). When the client sends a `tools/call` request for `"add"` with arguments like `{"a": 5, "b": 3}`, the server will execute `add(a=5, b=3)` and return the result `8`.

Example 2: A Tool Interacting with an External Service

Tools are often used to interact with the outside world. Let's create a tool that fetches weather data using an imaginary asynchronous HTTP client.

```python
# server.py (continued)
# ... (FastMCP setup and add tool from above) ...

import httpx # Assuming you have httpx installed (uv add httpx)

@mcp.tool()
async def fetch_weather(city: str) -> str:
    """Fetch current weather for a city"""
    # In a real app, you'd use a weather API key and proper error handling
    async with httpx.AsyncClient() as client:
        # Dummy URL for demonstration
        response = await client.get(f"https://example.com/weather/{city}")
        response.raise_for_status() # Raise an exception for bad status codes
        return response.text # Return the response body as text

# ... (Optional run block) ...
```

- This tool is `async` because fetching data from a network is an asynchronous

operation. `FastMCP` fully supports `async def` functions for tools.
- It takes a `city: str` parameter, which will be part of its `inputSchema`.
- It uses the `httpx` library to make an HTTP request. This demonstrates how tools can perform I/O and interact with external services.
- The return value (the response text) is sent back to the client.

When a client calls `"fetch_weather"` with `{"city": "London"}`, the server executes the `fetch_weather` function, waits for the HTTP request to complete, and returns the fetched weather data.

Example 3: A Tool Using the Context

Sometimes, your tool function needs access to information about the current MCP request or needs to interact with the MCP environment itself (like sending log messages or reporting progress). The `python-sdk` provides a `Context` object for this.

You can request the `Context` object by adding a parameter type-hinted with `Context` to your tool function.

```
# server.py (continued)
# ... (FastMCP setup and other tools) ...

# Assuming you have the Context class imported from mcp.server.fastmcp
from mcp.server.fastmcp import Context

@mcp.tool()
async def process_files(file_paths: list[str], ctx: Context) -> str:
    """Process a list of files with progress tracking and logging."""
    total_files = len(file_paths)
    for i, file_path in enumerate(file_paths):
        # Use the context to send an info log message to the client
        await ctx.info(f"Starting processing for file: {file_path}")

        # Use the context to report progress to the client
        # Progress is 0-indexed, so add 1 for display
        await ctx.report_progress(i + 1, total_files)

        try:
            # Use the context to read a resource (the file content)
            # This links back to our knowledge of Resources from Chapter 2!
            # ctx.read_resource returns an iterable of ReadResourceContents
            # We'll assume the first item is the text content for simplicity here
            contents = await ctx.read_resource(f"file://{file_path}")
            file_content = ""
            for item in contents:
                if hasattr(item, 'content') and isinstance(item.content, str):
```

```
                    file_content += item.content # Accumulate text content

            # Simulate some processing
            processed_data = f"Processed content of {file_path}: {file_content[:50]}..." # 

            # Use the context to send a debug log message
            await ctx.debug(f"Finished processing {file_path}. Data snippet: {processed_data

        except Exception as e:
            # Use the context to send an error log message
            await ctx.error(f"Error processing file {file_path}: {e}")
            # In a real tool, you might raise the exception or return an error result

    await ctx.info("All files processed.")
    return "File processing complete."

# ... (Optional run block) ...
```

- The function `process_files` takes a `file_paths: list[str]` parameter (part of the `inputSchema`) and a `ctx: Context` parameter.
- `FastMCP` automatically detects the `Context` type hint and injects the current request's `Context` object when the tool is called.
- Inside the function, we use methods provided by the `Context` object:
  - `ctx.info()`, `ctx.debug()`, `ctx.error()`: These send log messages back to the client. This is useful for providing feedback on the tool's execution progress or any issues encountered.
  - `ctx.report_progress()`: This sends a progress notification to the client, allowing the client application (like an LLM UI) to show a progress bar for long-running tasks.
  - `ctx.read_resource()`: This allows the tool to read the content of a Resource *via the client*. This is powerful because it means the tool doesn't need direct access to the resource's location; it asks the client to provide it according to the MCP protocol. This is especially useful for `file://` URIs where the client might have access to files the server doesn't.

The `Context` object provides a clean way for your tool logic to interact with the MCP session and leverage client capabilities like logging, progress display, and resource access without needing to know the low-level protocol details.

Behind the Scenes: How Tools Work

Let's look at what happens when a client sends a `tools/call` request.

When you use `@mcp.tool`, `FastMCP` doesn't immediately execute the function. Instead, it registers information about the tool (its name, the function to call, description, parameter schema, etc.) with its internal **`ToolManager`**.

Here's a simplified flow for a `tools/call` request:

```
sequenceDiagram
    participant Client
    participant Server as FastMCP Server
    participant TM as ToolManager
    participant ToolObj as Specific Tool Object
    participant YourCode as Your @mcp.tool function

    Client->>Server: tools/call request (name="add", arguments={"a": 5, "b": 3})
    Server->>TM: Handle call request for "add"
    TM->>TM: Look up tool "add"
    TM->>ToolObj: Find Tool object for add function
    TM->>ToolObj: Validate arguments {"a": 5, "b": 3} against schema
    ToolObj->>YourCode: Call add(a=5, b=3) (injecting Context if requested)
    YourCode-->>ToolObj: Return 8
    ToolObj-->>TM: Return result 8
    TM-->>Server: Return result 8
    Server-->>Client: tools/call response (content=[{"type": "text", "text": "8"}])
```

1. **Client Request:** The client sends a JSON-RPC request like `{"jsonrpc": "2.0", "id": 1, "method": "tools/call", "params": {"name": "add", "arguments": {"a": 5, "b": 3}}}`.
2. **Server Receives:** The `FastMCP` server receives this message via the chosen Transport.
3. **Routing:** `FastMCP` recognizes the `tools/call` method and routes it to its internal handler for calling tools.
4. **`ToolManager`:** This handler uses the `ToolManager` to find the tool corresponding to the requested name (`"add"`).
5. **Tool Object:** The `ToolManager` finds the appropriate internal `Tool` object wrapping your function.
6. **Argument Validation:** The `Tool` object (using Pydantic internally) validates the provided `arguments` (`{"a": 5, "b": 3}`) against the `inputSchema` derived from your function's type hints. If validation fails, an error is returned to the client.
7. **`run()` Method:** The server calls the `run()` method on that specific `Tool` object. This is where the actual work happens.
8. **Your Code Runs:** The `run()` method calls your `@mcp.tool` function (`add(a=5, b=3)`), passing the validated arguments and injecting the `Context` object if your function requested it.
9. **Result Returned:** Your function returns the result (`8`). The `run()` method returns this result.
10. **Result Formatting:** The `Tool` object and `ToolManager` format the result into the standard MCP `CallToolResult` format, which includes a list of `content` objects (like `TextContent`).
11. **Response:** The `FastMCP` server formats the result into an MCP response message and sends it back to the client.

This layered approach means you, as the server developer, only need to provide the function that performs the action. `FastMCP` and the `ToolManager` handle the protocol details, tool lookup, argument validation, context injection, and result formatting.

Looking at the code structure (you don't need to understand all of this yet, just see the pieces):

- `src/mcp/server/fastmcp/tools/base.py`: Defines the internal `Tool` class.

```python
# Simplified snippet from base.py
class Tool(BaseModel):
    fn: Callable[..., Any] = Field(exclude=True)
    name: str = Field(...)
    description: str = Field(...)
    parameters: dict[str, Any] = Field(...) # The JSON schema
    fn_metadata: FuncMetadata = Field(...) # Includes Pydantic model for args
    is_async: bool = Field(...)
    context_kwarg: str | None = Field(...) # Name of the param for Context

    @classmethod
    def from_function(cls, fn, name, description, context_kwarg):
        # Introspects fn, creates Pydantic model for args, builds schema
        ...

    async def run(self, arguments, context):
        # Validates arguments using fn_metadata.arg_model
        # Calls self.fn(arguments, context=context_obj)
        # Handles async/sync
        ...
```

- `src/mcp/server/fastmcp/tools/tool_manager.py`: Contains the `ToolManager` class.

```python
# Simplified snippet from tool_manager.py
class ToolManager:
    def __init__(self, warn_on_duplicate_tools=True):
        self._tools: dict[str, Tool] = {}

    def add_tool(self, fn, name, description):
        # Creates a Tool object from fn
        # Stores it in self._tools by name
        ...

    async def call_tool(self, name, arguments, context):
        # Looks up tool by name in self._tools
        # Calls tool.run(arguments, context)
```

...

- `src/mcp/server/fastmcp/server.py`: The `FastMCP` class uses the `@mcp.tool` decorator to register your functions by calling `self._tool_manager.add_tool()`. It also implements the `list_tools` and `call_tool` handlers that interact with the `ToolManager`. The `call_tool` handler also calls the `_convert_to_content` helper to format the tool's return value into the required MCP content list format.

This structure keeps the core logic of *what* the tool does separate from the MCP protocol handling, argument validation, and context management.

Conclusion

Tools are a fundamental concept in MCP for exposing actions and functionality from a server to a client. They are identified by names, take parameters, return results, and can have side effects. The `python-sdk`'s `FastMCP` makes defining tools easy using the `@mcp.tool` decorator, automatically handling parameter schema generation, argument validation, and execution via its internal `ToolManager`. Tools can also leverage the `Context` object to interact with the MCP session for logging, progress reporting, and resource access.

Clients use `tools/list` and `tools/call` requests to discover and execute tools.

Now that we know how to expose data using Resources and actions using Tools, let's move on to defining reusable interaction patterns using **Prompts**.

Next Chapter: Prompts

Welcome back to the `python-sdk` tutorial! In the previous chapters, we learned about the core concepts of the Model Context Protocol (MCP): Resources for exposing data and Tools for exposing actions.

In this chapter, we'll explore the third core primitive: **Prompts**.

What Problem Do Prompts Solve?

Imagine you're building an MCP server that helps an LLM application perform specific tasks, like:

- Reviewing a piece of code.
- Debugging an error message.
- Analyzing a database schema.
- Drafting an email based on some data.

While you could potentially define a single "master" tool that takes a complex instruction, it's often better to provide the LLM application with **predefined templates** or **structured conversation starters** for these specific tasks.

This is where **Prompts** come in.

Think of Prompts as **reusable interaction patterns** or **task definitions** provided by the server. They guide the LLM application on how to structure a conversation or what information to provide to achieve a specific goal. Instead of the LLM having to figure out *how* to ask for a code review, the server can offer a "Code Review" prompt that already defines the initial messages and expects the code as an argument.

What is a Prompt?

In MCP, a Prompt is a template for generating a sequence of messages, typically intended to be sent to an LLM.

- **Identified by a Name:** Like Tools, Prompts are identified by a simple, descriptive name (e.g., `"code_review"`, `"debug_error"`, `"analyze_schema"`).
- **Takes Arguments:** Prompts can be defined with specific input parameters, allowing the client to customize the generated messages (e.g., providing the code to review, the error message to debug).
- **Generates Messages:** When a client "gets" a prompt, the server executes the prompt's logic (often a function) which returns a list of messages. These messages can be simple text, or include references to Resources or Images.
- **Reusable Templates:** The key benefit is that the server defines the *structure* and *initial content* of the interaction, making it easy for the client to initiate complex tasks consistently.
- **Clients Discover and Get:** Clients use specific MCP requests (`prompts/list` and `prompts/get`) to find out what prompts are available and then retrieve the messages for a specific prompt, providing arguments if necessary.

The core idea is that the server *exposes* these predefined interaction templates, and the client *requests* them using standard MCP messages, providing the necessary inputs to fill in the template.

How Clients Interact with Prompts

An MCP client interacts with prompts using specific requests defined by the protocol:

1. **`prompts/list`:** The client sends this request to ask the server for a list of all the prompts it currently offers. The server responds with metadata about each prompt (its name, description, and a list of its arguments).
2. **`prompts/get`:** Once the client knows the name of a prompt and understands its arguments, it sends this request to get the actual messages generated by the prompt, providing the required arguments. The server executes the prompt's logic and returns a list of messages.

We'll see how to use these client requests in Chapter 5: Client Session. For now, let's focus on how to *define* prompts on the server side using the `python-sdk`.

Defining Prompts with FastMCP

Similar to Resources and Tools, the `python-sdk`'s `FastMCP` server makes defining prompts very easy using the `@mcp.prompt()` decorator.

Let's use the `FastMCP` object we've been using:

```python
# server.py
from mcp.server.fastmcp import FastMCP

# Create an MCP server instance
mcp = FastMCP("My Prompt Server")

# ... add prompts here ...

# (Optional) Run the server directly for testing
if __name__ == "__main__":
    mcp.run()
```

Example 1: A Simple Static Prompt

Let's create a prompt that provides a fixed introductory message.

```python
# server.py (continued)
# ... (FastMCP setup from above) ...

@mcp.prompt()
def introduce_yourself() -> str:
    """A prompt to introduce the server."""
    return "Hello! I am the My Prompt Server, ready to assist you."

# ... (Optional run block) ...
```

- We use the `@mcp.prompt()` decorator. This tells `FastMCP` to register the function `introduce_yourself` as a prompt.
- By default, the prompt's name will be the function's name: `"introduce_yourself"`.
- The function `introduce_yourself` simply returns a string. When a client calls `prompts/get` for `"introduce_yourself"`, the server will call this function. The returned string `"Hello! I am the My Prompt Server, ready to assist you."` will be automatically wrapped into a list containing a single `UserMessage` with that text content.
- The docstring `"A prompt to introduce the server."` is automatically used as the prompt's description when the client lists prompts.

When a client sends a `prompts/list` request, the server will include information about the `"introduce_yourself"` prompt. When the client sends

a `prompts/get` request for `"introduce_yourself"`, the server will execute `introduce_yourself()` and return a list of messages like `[{"role": "user", "content": {"type": "text", "text": "Hello! I am the My Prompt Server, ready to assist you."}}]`.

Example 2: A Prompt with Arguments

Prompts become much more powerful when they can take arguments to customize the output messages.

```
# server.py (continued)
# ... (FastMCP setup and static prompt from above) ...

@mcp.prompt()
def summarize_text(text: str, length: Literal["short", "medium", "long"] = "medium") -> str:
    """Generate a prompt to summarize text."""
    return f"Please summarize the following text. The summary should be {length}. Text:\n\n{

# ... (Optional run block) ...
```

- The function `summarize_text` takes two parameters: `text: str` and `length: Literal["short", "medium", "long"]`.
- `FastMCP` uses these type hints to automatically generate the `arguments` list for the prompt, telling the client that this prompt expects a required string argument named `text` and an optional argument named `length` with specific allowed values.
- The function uses an f-string to incorporate the provided arguments into the generated message.

When a client sends `prompts/list`, the server will list `"summarize_text"` and describe its `text` (required string) and `length` (optional string, enum) arguments. When the client sends `prompts/get` for `"summarize_text"` with arguments like `{"text": "...", "length": "short"}`, the server will execute `summarize_text(text="...", length="short")` and return the resulting message.

Example 3: A Prompt Returning Structured Messages

You can return more complex message structures, including multiple messages and different roles (`user`, `assistant`). The `python-sdk` provides helper classes like `UserMessage` and `AssistantMessage` for this.

```
# server.py (continued)
# ... (FastMCP setup and other prompts) ...

# Assuming you have Message, UserMessage, AssistantMessage imported
from mcp.server.fastmcp.prompts.base import Message, UserMessage, AssistantMessage
```

```python
@mcp.prompt()
def debug_code_error(error_message: str, code_snippet: str) -> list[Message]:
    """Generate a prompt to debug a code error."""
    return [
        UserMessage(f"I encountered the following error:\n\n{error_message}"),
        UserMessage(f"Here is the relevant code snippet:\n\n{code_snippet}"),
        AssistantMessage("Okay, I will help you debug this. What steps have you already take
    ]

# ... (Optional run block) ...
```

- The function `debug_code_error` takes `error_message: str` and `code_snippet: str` as arguments.
- It returns a `list[Message]`. Each item in the list is an instance of `UserMessage` or `AssistantMessage`.
- `UserMessage` represents a message from the user (or the client application acting on behalf of the user).
- `AssistantMessage` represents a message from the assistant (or the server providing context that looks like an assistant response).
- The content of these messages can be simple strings, which `FastMCP` automatically wraps in `TextContent`.

When a client calls `prompts/get` for `"debug_code_error"` with the required arguments, the server executes the function and returns the list of structured messages. This allows the server to define a multi-turn conversation or provide initial context from both the "user" and "assistant" perspectives.

Example 4: A Prompt Referencing a Resource

Prompts can also include references to Resources defined by the server. This is useful for including large pieces of data (like file contents or database schemas) in the prompt without embedding the full content directly in the `get_prompt` response. The client is then responsible for reading the resource using `resources/read`.

```python
# server.py (continued)
# ... (FastMCP setup and other prompts) ...

# Assuming you have EmbeddedResource imported
from mcp.types import EmbeddedResource
# Assuming you have TextResourceContents imported
from mcp.types import TextResourceContents # Or BlobResourceContents for binary

@mcp.prompt()
def analyze_file_content(file_path: str) -> list[Message]:
    """Generate a prompt to analyze the content of a file resource."""
    # Construct a Resource object reference
```

```
    file_resource_ref = TextResourceContents(
        uri=f"file://{file_path}", # Use the file:// URI scheme
        text="" # Content is not included here, client reads it
    )

    return [
        UserMessage("Please analyze the content of the following file:"),
        # Include the resource reference in the message content
        UserMessage(content=EmbeddedResource(type="resource", resource=file_resource_ref)),
        AssistantMessage("Okay, I will read the file and provide an analysis."),
    ]

# ... (Optional run block) ...
```

- The function takes `file_path: str` as an argument.
- Inside the function, we create a `TextResourceContents` object. Crucially, we provide the `uri` of the resource (`file://{file_path}`) but *don't* include the actual `text` content here. This tells the client *where* to find the content without sending it in the `get_prompt` response.
- We then wrap this resource object inside an `EmbeddedResource` object, which has `type="resource"`.
- Finally, we include the `EmbeddedResource` in the `content` of a `UserMessage`.

When a client calls `prompts/get` for `"analyze_file_content"` with a path like `{"file_path": "/path/to/my/file.txt"}`, the server returns messages including the `EmbeddedResource`. The client application sees this `EmbeddedResource` and knows it needs to make a separate `resources/read` call for `uri="file:///path/to/my/file.txt"` to get the actual file content before presenting it to the LLM or user. This pattern is efficient for large resources.

Behind the Scenes: How Prompts Work

Let's look at what happens when a client sends a `prompts/get` request.

When you use `@mcp.prompt`, `FastMCP` doesn't immediately generate the messages. Instead, it registers information about the prompt (its name, the function to call, description, arguments) with its internal **`PromptManager`**.

Here's a simplified flow for a `prompts/get` request:

```
sequenceDiagram
    participant Client
    participant Server as FastMCP Server
    participant PM as PromptManager
    participant PromptObj as Specific Prompt Object
    participant YourCode as Your @mcp.prompt function
```

```
Client->>Server: prompts/get request (name="summarize_text", arguments={"text": "...", '
Server->>PM: Handle get request for "summarize_text"
PM->>PM: Look up prompt "summarize_text"
PM->>PromptObj: Find Prompt object for summarize_text function
PM->>PromptObj: Validate arguments {"text": "...", "length": "short"} against schema
PromptObj->>YourCode: Call summarize_text(text="...", length="short")
YourCode-->>PromptObj: Return "Please summarize..."
PromptObj-->>PM: Return list of Message objects
PM-->>Server: Return list of Message objects
Server-->>Client: prompts/get response (messages=[{"role": "user", "content": {"type": '
```

1. **Client Request:** The client sends a JSON-RPC request like `{"jsonrpc": "2.0", "id": 1, "method": "prompts/get", "params": {"name": "summarize_text", "arguments": {"text": "...", "length": "short"}}}`.
2. **Server Receives:** The `FastMCP` server receives this message via the chosen Transport.
3. **Routing:** `FastMCP` recognizes the `prompts/get` method and routes it to its internal handler for getting prompts.
4. **`PromptManager`:** This handler uses the `PromptManager` to find the prompt corresponding to the requested name (`"summarize_text"`).
5. **Prompt Object:** The `PromptManager` finds the appropriate internal `Prompt` object wrapping your function.
6. **Argument Validation:** The `Prompt` object (using Pydantic internally) validates the provided `arguments` against the `arguments` schema derived from your function's type hints. If validation fails, an error is returned to the client.
7. **`render()` Method:** The server calls the `render()` method on that specific `Prompt` object. This is where the actual message generation happens.
8. **Your Code Runs:** The `render()` method calls your `@mcp.prompt` function (`summarize_text(text="...", length="short")`), passing the validated arguments.
9. **Result Returned:** Your function returns the result (e.g., a string or a list of `Message` objects).
10. **Result Formatting:** The `Prompt` object and `PromptManager` format the result into the standard MCP `GetPromptResult` format, ensuring the output is a list of valid `PromptMessage` objects (converting strings, dicts, etc., as needed).
11. **Response:** The `FastMCP` server formats the result into an MCP response message and sends it back to the client.

This layered approach means you, as the server developer, only need to provide the function that generates the messages based on inputs. `FastMCP` and the `PromptManager` handle the protocol details, prompt lookup, argument validation, and message formatting.

Looking at the code structure (you don't need to understand all of this yet, just see the pieces):

- `src/mcp/server/fastmcp/prompts/base.py`: Defines the internal `Prompt` class, `Message` base class, and helper classes like `UserMessage`, `AssistantMessage`. It also contains the `from_function` class method to create a `Prompt` from your decorated function and the `render` method to execute the function and format the output.
- `src/mcp/server/fastmcp/prompts/manager.py`: Contains the `PromptManager` class which stores registered `Prompt` objects and provides methods like `list_prompts`, `get_prompt`, and `render_prompt`.
- `src/mcp/server/fastmcp/server.py`: The `FastMCP` class uses the `@mcp.prompt` decorator to register your functions by calling `self._prompt_manager.add_prompt()`. It also implements the `list_prompts` and `get_prompt` handlers that interact with the `PromptManager`.

This structure keeps the core logic of *what* messages to generate separate from the MCP protocol handling, argument validation, and message formatting.

Conclusion

Prompts are a fundamental concept in MCP for defining reusable templates for LLM interactions. They are identified by names, can take arguments, and generate structured lists of messages, potentially including references to Resources. The `python-sdk`'s `FastMCP` makes defining prompts easy using the `@mcp.prompt` decorator, automatically handling argument schema generation, argument validation, and message formatting via its internal `PromptManager`.

Clients use `prompts/list` and `prompts/get` requests to discover and retrieve prompt messages.

Now that we've covered the three core primitives (Resources, Tools, and Prompts), let's shift our focus to the client side and learn how to interact with an MCP server using the `ClientSession`.

Next Chapter: Client Session

Welcome back to the `python-sdk` tutorial! In the previous chapters, we explored the core building blocks of the Model Context Protocol (MCP) from the **server's** perspective: Resources for exposing data, Tools for exposing actions, and Prompts for defining interaction templates.

Now, let's switch gears and look at the **client's** perspective. How does a Python application connect to an MCP server and actually *use* those Resources, Tools, and Prompts?

What Problem Does the Client Session Solve?

Imagine you're building an LLM application or any other program that needs to interact with an MCP server. You know the server exposes cool capabilities like fetching weather data (a Tool) or reading a configuration file (a Resource).

But how do you:

1. Establish a connection to the server?
2. Send the correct MCP messages (which are based on JSON-RPC) to request a list of tools or call a specific tool?
3. Receive the server's response and understand it?
4. Handle ongoing communication, like notifications from the server?

Dealing with the low-level details of network sockets, message formatting, request IDs, and response parsing can be complicated and error-prone. You want to focus on *what* you want to do (e.g., "call the 'fetch_weather' tool for London") not *how* the message is sent over the wire.

This is where the **Client Session** comes in.

The Solution: The `ClientSession`

The `ClientSession` is your high-level interface in the `python-sdk` for interacting with an MCP server from a client application.

Think of the `ClientSession` as a **smart connection object**. It wraps the underlying communication channel (the "transport") and handles all the complexities of the MCP protocol for you. It provides simple Python methods that correspond directly to the MCP requests you want to send (like `list_tools()`, `call_tool()`, `read_resource()`, etc.).

Here's what the `ClientSession` does for you:

- **Manages the Connection:** It keeps the communication channel open and handles sending and receiving messages reliably.
- **Handles MCP Protocol:** It formats your Python method calls into the correct JSON-RPC messages required by MCP and parses incoming JSON-RPC messages back into Python objects.
- **Provides a Programmatic Interface:** You interact with the server's capabilities using intuitive Python methods, not by manually crafting JSON strings.
- **Processes Responses and Notifications:** It receives responses to your requests and handles asynchronous notifications sent by the server (like logging messages or progress updates).

Using a `ClientSession` allows you to focus on the logic of your client application – deciding *when* to call a tool or *what* resource to read – without getting bogged down in the communication mechanics.

How to Use `ClientSession`

Let's walk through the basic steps of connecting to a server and using a `ClientSession`.

MCP communication happens over a "transport". A common and simple transport for development is standard input/output (stdio), where the client and server communicate by writing to and reading from each other's standard streams. The `python-sdk` provides helpers for this.

Step 1: Connect using a Transport Helper

The `stdio_client` helper is a convenient way to launch a server process and get the necessary communication streams.

```python
# client.py
import asyncio
from mcp import ClientSession
from mcp.client.stdio import stdio_client
from mcp import StdioServerParameters # Needed to tell stdio_client how to start the server

async def main():
    # Define how to start the server process.
    # This assumes you have a server script named 'server.py'
    # like the one from the Quickstart or previous chapters.
    server_params = StdioServerParameters(
        command="python",
        args=["server.py"], # Replace with the actual path to your server script
    )

    # Use the stdio_client context manager to launch the server
    # and get the read/write streams.
    async with stdio_client(server_params) as (read_stream, write_stream):
        # read_stream is where you receive messages from the server
        # write_stream is where you send messages to the server

        # Step 2: Create and use the ClientSession
        # ... (see next step) ...

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

- We import `stdio_client` and `StdioServerParameters`.
- `StdioServerParameters` tells the helper which command to run (`"python"`) and what arguments to pass (`["server.py"]`) to start the server process.
- The `async with stdio_client(...)` block launches the server process and provides `read_stream` and `write_stream` objects. These streams are

the raw communication channel.

Step 2: Create and Initialize the `ClientSession`

Once you have the streams, you create a `ClientSession` instance using them. The session itself should also be managed using an `async with` block to ensure proper setup and teardown.

```
# client.py (continued from above)
# ... (imports and server_params) ...

async def main():
    server_params = StdioServerParameters(
        command="python",
        args=["server.py"],
    )

    async with stdio_client(server_params) as (read_stream, write_stream):
        # Create a ClientSession using the streams
        async with ClientSession(read_stream, write_stream) as session:
            # The session object is now ready to handle MCP communication

            # Step 3: Initialize the session (the MCP handshake)
            # ... (see next step) ...

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

- We import `ClientSession`.
- We create `ClientSession(read_stream, write_stream)`. This object now knows how to send and receive MCP messages over these specific streams.
- The `async with ClientSession(...)` ensures the session's background tasks (like listening for incoming messages) are started and stopped correctly.

Step 3: Initialize the Session (The Handshake)

The very first thing you must do after creating the session is call `await session.initialize()`. This performs the MCP handshake: the client and server exchange information about their supported protocol versions and capabilities.

```
# client.py (continued from above)
# ... (imports, server_params, stdio_client context) ...

async def main():
    server_params = StdioServerParameters(
```

```
        command="python",
        args=["server.py"],
    )

    async with stdio_client(server_params) as (read_stream, write_stream):
        async with ClientSession(read_stream, write_stream) as session:
            print("Client session created. Initializing...")
            # Perform the MCP handshake
            server_capabilities = await session.initialize()
            print("Client initialized!")
            print(f"Server capabilities: {server_capabilities}")

            # Now you can interact with the server's capabilities!
            # ... (see next step) ...

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

- `await session.initialize()` sends the `initialize` request to the server and waits for the `initialize` response.
- The response contains the server's capabilities (what Resources, Tools, Prompts, etc., it supports). This is returned by the `initialize()` method.
- After `initialize()` completes successfully, the session is fully set up and ready for further interaction.

Step 4: Interact with Server Capabilities

Now that the session is initialized, you can use its methods to interact with the server's exposed capabilities. These methods correspond directly to the MCP requests we discussed in previous chapters.

Let's assume our `server.py` is the simple Quickstart server from the README:

```
# server.py (Quickstart example)
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("Demo")

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

@mcp.resource("greeting://{name}")
def get_greeting(name: str) -> str:
    """Get a personalized greeting"""
```

```
    return f"Hello, {name}!"
```

Here's how the client can interact with it:

```
# client.py (continued from above)
# ... (imports, server_params, stdio_client context, ClientSession context, initialize call,

async def main():
    server_params = StdioServerParameters(
        command="python",
        args=["server.py"], # Make sure this points to your server file
    )

    async with stdio_client(server_params) as (read_stream, write_stream):
        async with ClientSession(read_stream, write_stream) as session:
            print("Client session created. Initializing...")
            server_capabilities = await session.initialize()
            print("Client initialized!")
            print(f"Server capabilities: {server_capabilities}")

            # --- Interact with Tools ---
            print("\nListing tools...")
            tools_list = await session.list_tools()
            print(f"Available tools: {[tool.name for tool in tools_list.tools]}")

            print("\nCalling 'add' tool...")
            # Call the 'add' tool with arguments
            add_result = await session.call_tool("add", arguments={"a": 5, "b": 3})
            # The result is a list of content objects. For simple text,
            # we expect one TextContent object.
            if add_result.content and add_result.content[0].type == "text":
                 print(f"Result of add(5, 3): {add_result.content[0].text}")
            else:
                 print(f"Unexpected result format: {add_result}")


            # --- Interact with Resources ---
            print("\nListing resources...")
            resources_list = await session.list_resources()
            # Note: Dynamic resources like greeting://{name} appear as templates
            print(f"Available resources/templates: {[r.uri for r in resources_list.resources

            print("\nReading 'greeting://Alice' resource...")
            # Read the dynamic resource, providing the name in the URI
            greeting_result = await session.read_resource("greeting://Alice")
            # The result is a list of content objects.
            if greeting_result.content and greeting_result.content[0].type == "text":
```

```
            print(f"Content of greeting://Alice: {greeting_result.content[0].text}")
        else:
            print(f"Unexpected resource content format: {greeting_result}")

        # --- Interact with Prompts ---
        # (Assuming the server had a prompt, e.g., from Chapter 4)
        # print("\nListing prompts...")
        # prompts_list = await session.list_prompts()
        # print(f"Available prompts: {[p.name for p in prompts_list.prompts]}")

        # print("\nGetting 'summarize_text' prompt...")
        # prompt_messages = await session.get_prompt("summarize_text", arguments={"text
        # print(f"Generated prompt messages: {prompt_messages.messages}")


        print("\nInteraction complete. Session closing.")

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

- `await session.list_tools()`: Sends the `tools/list` request and waits for the response containing the list of available tools.
- `await session.call_tool("add", arguments={"a": 5, "b": 3})`: Sends the `tools/call` request for the tool named `"add"` with the specified arguments. Waits for the result.
- `await session.list_resources()`: Sends the `resources/list` request and waits for the response.
- `await session.read_resource("greeting://Alice")`: Sends the `resources/read` request for the specified URI and waits for the content.
- `await session.list_prompts()` and `await session.get_prompt()` (commented out, but show the pattern): Similar methods for interacting with Prompts.

Notice how each interaction is a simple `await session.method_name(...)` call. The `ClientSession` handles all the underlying MCP message exchange.

Handling Notifications

Besides requests and responses, MCP servers can send notifications to the client (e.g., logging messages, progress updates, resource updates). The `ClientSession` receives these automatically in the background.

By default, the `ClientSession` doesn't do anything visible with these notifications. If you want to process them (e.g., print log messages to the console), you can provide callback functions when creating the `ClientSession`.

```
# client.py (adding notification handlers)
import asyncio
```

```python
from mcp import ClientSession
from mcp.client.stdio import stdio_client
from mcp import StdioServerParameters
from mcp.types import LoggingMessageNotificationParams, ProgressNotificationParams

# Define a callback for logging messages
async def handle_logging_message(params: LoggingMessageNotificationParams):
    print(f"[SERVER LOG] [{params.level.value}] {params.message}")

# Define a callback for progress notifications
async def handle_progress_notification(params: ProgressNotificationParams):
    # progress is 0-indexed, total is total steps
    if params.total is not None:
        percentage = (params.progress / params.total) * 100
        print(f"[SERVER PROGRESS] Token: {params.progressToken}, Step {params.progress}/{par
    else:
         print(f"[SERVER PROGRESS] Token: {params.progressToken}, Step {params.progress}")


async def main():
    server_params = StdioServerParameters(
        command="python",
        args=["server.py"],
    )

    async with stdio_client(server_params) as (read_stream, write_stream):
        # Pass the callback functions when creating the session
        async with ClientSession(
            read_stream,
            write_stream,
            logging_callback=handle_logging_message,
            # progress notifications are handled internally by default,
            # but you could add a custom handler if needed via message_handler
        ) as session:
            print("Client session created. Initializing...")
            await session.initialize()
            print("Client initialized!")

            # Now interact as before...
            print("\nCalling 'add' tool...")
            add_result = await session.call_tool("add", arguments={"a": 10, "b": 20})
            if add_result.content and add_result.content[0].type == "text":
                 print(f"Result of add(10, 20): {add_result.content[0].text}")

            # If the server had a tool that sends logs or progress (like process_files from
            # you would see the output from handle_logging_message or handle_progress_notif
```

```
            # as the tool runs.

            print("\nInteraction complete. Session closing.")


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

- We define `async def` functions (`handle_logging_message`, `handle_progress_notification`) that match the expected signature for the callbacks.
- We pass these functions to the `ClientSession` constructor using the `logging_callback` and `message_handler` arguments (though progress is often handled internally by the base session logic, `message_handler` is the general way to intercept *all* incoming messages, including notifications).
- When the server sends a logging notification, the `handle_logging_message` function will be called automatically by the `ClientSession`'s background listener.

This allows your client application to react to events happening on the server side without constantly polling.

Behind the Scenes: How `ClientSession` Works

Let's take a peek under the hood to understand what the `ClientSession` is doing.

The `ClientSession` inherits from a base class (`BaseSession`) that handles the fundamental logic of sending and receiving JSON-RPC messages over streams. It maintains a dictionary to keep track of outgoing requests by their unique ID, so it knows which incoming response belongs to which pending request.

Here's a simplified flow when you call a method like `session.list_tools()`:

```
sequenceDiagram
    participant YourCode as Your Client Code
    participant CS as ClientSession
    participant Transport as Read/Write Streams
    participant Server as MCP Server

    YourCode->>CS: await session.list_tools()
    CS->>CS: Create JSON-RPC request for "tools/list" with unique ID
    CS->>Transport: Write JSON-RPC request message to stream
    Transport->>Server: Message arrives at server

    Server->>Transport: Write JSON-RPC response message to stream (with same ID)
    Transport->>CS: Message arrives at ClientSession

    CS->>CS: Read JSON-RPC response message from stream
```

```
CS->>CS: Find pending request matching the ID
CS->>CS: Parse JSON-RPC result into Python object (ListToolsResult)
CS-->>YourCode: Return ListToolsResult object
```

1. **Your Code Calls:** You call an asynchronous method like `await session.list_tools()`.
2. **`ClientSession` Creates Request:** The `list_tools()` method in `ClientSession` creates the appropriate MCP request object (`ListToolsRequest`). It then wraps this in a standard JSON-RPC request message, assigning it a unique ID.
3. **`ClientSession` Sends:** The session writes the JSON-RPC message (as a string or bytes) to the `write_stream` provided during initialization.
4. **Transport Delivers:** The transport mechanism (stdio, SSE, etc.) carries the message to the server.
5. **Server Processes:** The server receives, processes the request, and generates a JSON-RPC response message with the *same* unique ID.
6. **Transport Delivers:** The response message travels back to the client via the `read_stream`.
7. **`ClientSession` Receives:** The session's background listener reads the incoming message from the `read_stream`.
8. **`ClientSession` Matches & Parses:** It looks up the pending request using the message's ID. It then parses the JSON-RPC `result` payload into the expected Python object type (`ListToolsResult` in this case) using Pydantic models defined in `mcp.types`.
9. **`ClientSession` Returns:** The `await session.list_tools()` call in your code completes, returning the parsed Python object.

For notifications, the flow is simpler:

```
sequenceDiagram
    participant Server as MCP Server
    participant Transport as Read/Write Streams
    participant CS as ClientSession
    participant YourCallback as Your Notification Handler

    Server->>Transport: Write JSON-RPC notification message to stream
    Transport->>CS: Message arrives at ClientSession

    CS->>CS: Read JSON-RPC notification message from stream
    CS->>CS: Identify notification type (e.g., logging/message)
    CS->>YourCallback: Call the registered callback function (e.g., handle_logging_message)
    YourCallback-->>CS: (Callback finishes)
```

1. **Server Sends Notification:** The server sends a JSON-RPC notification message (which doesn't have an ID and doesn't expect a response).
2. **Transport Delivers:** The message arrives at the client via the `read_stream`.

3. `ClientSession` **Receives:** The session's background listener reads the message.
4. `ClientSession` **Routes:** It identifies the message as a notification and determines its type (e.g., `notifications/logging/message`).
5. `ClientSession` **Calls Callback:** If you provided a callback for that notification type (like `logging_callback`), the session calls your function, passing the parsed notification parameters.

The `ClientSession` effectively acts as a proxy, translating between your Python code and the MCP JSON-RPC messages flowing over the transport.

You can see the implementation details in `src/mcp/client/session.py`. It uses `anyio` for asynchronous stream handling and the Pydantic models from `mcp.types` for parsing and validating messages.

Conclusion

The `ClientSession` is the essential component for building MCP client applications with the `python-sdk`. It abstracts away the complexities of the MCP protocol and the underlying transport, providing a clean, asynchronous Python interface to interact with server capabilities like Resources, Tools, and Prompts. By using `stdio_client` and the `ClientSession`'s methods, you can easily connect to and communicate with any compliant MCP server.

Now that we've seen how to interact with a server from the client side, let's dive deeper into building the server itself using the high-level `FastMCP` framework.

Next Chapter: FastMCP Server

Welcome back to the `python-sdk` tutorial! In the last few chapters, we've explored the core building blocks of the Model Context Protocol (MCP) from both the server's perspective (Resources, Tools, Prompts) and the client's perspective (Client Session).

Now, let's bring it all together and learn how to easily build a complete MCP server using the `python-sdk`'s high-level framework: **FastMCP**.

What Problem Does FastMCP Solve?

Imagine you want to create an MCP server that exposes some data as Resources and some actions as Tools. If you were to build this from scratch, you would need to:

1. Set up a communication channel (a Transport like stdio or SSE).
2. Listen for incoming messages from the client.
3. Parse the incoming JSON-RPC messages according to the MCP specification.
4. Identify which MCP method the client is requesting (e.g., `tools/list`, `resources/read`, `tools/call`).

5. Validate the parameters sent by the client.
6. Route the request to the correct piece of your server logic (the function that lists tools, the function that reads a specific resource, the function that executes a tool).
7. Execute your logic.
8. Format the result of your logic into the correct MCP response message.
9. Send the response back to the client over the transport.
10. Handle errors, notifications, and the session lifecycle.

That's a lot of work! It requires deep knowledge of the MCP protocol and asynchronous programming.

This is exactly the problem **FastMCP** solves.

The Solution: FastMCP

FastMCP is a high-level, user-friendly framework within the `python-sdk` designed specifically for building MCP servers in Python.

Think of FastMCP like popular web frameworks such as Flask or FastAPI, but instead of helping you build HTTP APIs, it helps you build MCP services. Just as Flask/FastAPI let you define web endpoints using decorators (`@app.route`, `@app.get`), FastMCP lets you define your MCP capabilities (Resources, Tools, Prompts) using simple Python functions decorated with `@mcp.resource`, `@mcp.tool`, and `@mcp.prompt`.

FastMCP handles all the underlying complexities:

- Implementing the MCP protocol specification.
- Managing the communication Transport.
- Parsing incoming messages and formatting outgoing messages.
- Routing requests to your specific functions.
- Validating input parameters based on your function's type hints.
- Handling asynchronous operations (`async def`).
- Providing a simple way to access the request Context.

With FastMCP, you can focus on writing the Python functions that define *what* your server does (the business logic), and let the framework handle *how* it communicates via MCP.

Building a Simple FastMCP Server

Let's revisit the simple Quickstart example we saw in Chapter 1: MCP Protocol and break down how it works using FastMCP.

Our goal is to create a server that can: 1. Provide an "add" Tool that sums two numbers. 2. Provide a "greeting" Resource that returns a personalized message based on a name in the URI.

Here's the complete code for `server.py`:

```python
# server.py
from mcp.server.fastmcp import FastMCP

# 1. Create a FastMCP server instance
# We give it a name that clients can see during initialization
mcp = FastMCP("Demo Server")

# 2. Define a Tool using the @mcp.tool() decorator
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

# 3. Define a Resource using the @mcp.resource() decorator
# This resource uses a URI template with a parameter {name}
@mcp.resource("greeting://{name}")
def get_greeting(name: str) -> str:
    """Get a personalized greeting"""
    return f"Hello, {name}!"

# 4. Add a block to run the server directly (optional, but useful for testing)
if __name__ == "__main__":
    print("Starting FastMCP server...")
    # By default, mcp.run() uses the stdio transport
    mcp.run()
    print("FastMCP server stopped.")
```

Let's walk through this code:

Step 1: Create the FastMCP Instance

```python
from mcp.server.fastmcp import FastMCP

# Create a FastMCP server instance
mcp = FastMCP("Demo Server")
```

This line creates the central `FastMCP` object. This object is the core of your server application. You'll use it to register your Resources, Tools, and Prompts. The name `"Demo Server"` is used by the server to identify itself to clients during the MCP initialization handshake.

Step 2: Define a Tool

```python
# Define a Tool using the @mcp.tool() decorator
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
```

```
    return a + b
```

Here, we define a standard Python function `add` that takes two integer arguments and returns an integer. The `@mcp.tool()` decorator tells the `FastMCP` instance (`mcp`) to register this function as an MCP Tool.

- **Tool Name:** By default, the tool's name will be the function's name: `"add"`.
- **Description:** The function's docstring (`"""Add two numbers"""`) is automatically used as the tool's description when a client requests the list of tools.
- **Input Schema:** FastMCP inspects the function's type hints (`a: int`, `b: int`) and automatically generates the `inputSchema` for the tool, which tells clients what arguments the tool expects and their types.
- **Execution:** When a client sends a `tools/call` request for the `"add"` tool with arguments like `{"a": 5, "b": 3}`, FastMCP will call this `add(a=5, b=3)` function and return its result (`8`) to the client.

This is much simpler than manually handling the `tools/call` request, parsing arguments from JSON, validating them, calling your function, and formatting the response!

Step 3: Define a Resource

```
# Define a Resource using the @mcp.resource() decorator
# This resource uses a URI template with a parameter {name}
@mcp.resource("greeting://{name}")
def get_greeting(name: str) -> str:
    """Get a personalized greeting"""
    return f"Hello, {name}!"
```

Here, we define another Python function `get_greeting` that takes a string argument `name` and returns a string. The `@mcp.resource("greeting://{name}")` decorator tells `FastMCP` to register this function as an MCP Resource.

- **Resource URI:** The string `"greeting://{name}"` specifies the URI template for this resource. The `{name}` part indicates a dynamic parameter in the URI.
- **Description:** The docstring (`"""Get a personalized greeting"""`) is used as the resource's description.
- **Parameter Mapping:** FastMCP matches the `{name}` parameter in the URI template to the `name` parameter in the function signature.
- **Execution:** When a client sends a `resources/read` request for a URI like `greeting://Alice`, FastMCP matches the URI to the template, extracts `"Alice"` for the `{name}` parameter, and calls the `get_greeting(name="Alice")` function. The function's return value (`"Hello, Alice!"`) is sent back to the client as the resource content.

FastMCP handles the URI parsing, parameter extraction, function calling, and

content formatting for you. Because the URI contains a parameter, this resource is automatically registered as a **Resource Template**, which clients can discover using `resources/templates/list`.

Step 4: Running the Server

```
# Add a block to run the server directly (optional, but useful for testing)
if __name__ == "__main__":
    print("Starting FastMCP server...")
    # By default, mcp.run() uses the stdio transport
    mcp.run()
    print("FastMCP server stopped.")
```

This standard Python `if __name__ == "__main__":` block allows you to run the server directly by executing the script. `mcp.run()` is a synchronous function that starts the server's main loop. By default, it uses the `stdio` Transport, meaning it will communicate with a client by reading from its standard input and writing to its standard output.

When you run this script, it will start listening for MCP messages on stdio. A client application (like the one we built in Chapter 5: Client Session using `stdio_client`) can then connect to it.

Running Your FastMCP Server

To run the `server.py` file we just created:

1. Save the code above as `server.py`.
2. Make sure you have the `mcp` package installed (`uv add mcp` or `pip install mcp`).
3. Open your terminal in the directory where you saved `server.py`.
4. Run the script:

   ```
   python server.py
   ```

   You should see the output:

   ```
   Starting FastMCP server...
   ```

   The server is now running and waiting for a client to connect via stdio.

You can also use the `mcp run` command provided by the SDK's CLI (which we'll cover in Chapter 10: CLI (mcp command)):

```
mcp run server.py
```

This does essentially the same thing as `python server.py` in this case, but the `mcp` command has additional features for managing dependencies, etc.

While the server is running, you can connect to it using an MCP client that uses the stdio transport, like the `client.py` example from Chapter 5: Client Session or the `mcp dev` command (also covered in Chapter 10).

Beyond the Basics (Briefly)

FastMCP supports more advanced features:

- **Asynchronous Functions:** Your `@mcp.tool` and `@mcp.resource` functions can be `async def` functions if they need to perform asynchronous operations (like making network requests or reading files asynchronously). FastMCP handles awaiting them correctly.
- **Context:** As we saw in Chapter 3: Tools, your tool functions can request a `Context` object via a type hint (`ctx: Context`). This object provides access to MCP-specific features like sending log messages, reporting progress, and reading resources via the client.
- **Prompts:** You can define Prompts using the `@mcp.prompt()` decorator, similar to tools and resources.
- **Other Transports:** While `mcp.run()` defaults to stdio, FastMCP also supports the SSE (Server-Sent Events) transport, which is commonly used for web-based clients or integration with ASGI servers (like Uvicorn). You can run with SSE using `mcp.run("sse")` or integrate it into an existing ASGI app using `mcp.sse_app()`. We'll cover Transports in more detail later.

Behind the Scenes: How FastMCP Works

FastMCP is built on top of the `python-sdk`'s lower-level `Server` class (which we'll touch upon in Chapter 7: Session and Chapter 8: Transports). It provides the high-level abstraction layer.

When you use the `@mcp.tool`, `@mcp.resource`, or `@mcp.prompt` decorators, FastMCP doesn't immediately execute your functions. Instead, it does the following:

1. **Introspection:** It inspects your function (its name, docstring, type hints).
2. **Registration:** It creates an internal representation of your capability (a `Tool`, `Resource`, or `Prompt` object) and registers it with internal managers (`ToolManager`, `ResourceManager`, `PromptManager`). For tools and prompts, it uses Pydantic to automatically generate the input schema based on your type hints. For resources, it determines if it's a static resource or a template based on the URI and function parameters.
3. **Handler Mapping:** FastMCP sets up the core MCP protocol handlers (`list_tools`, `call_tool`, `list_resources`, `read_resource`, etc.) on the underlying `MCPServer` instance. These handlers know how to interact with the managers.

When a client sends a request (e.g., `tools/call` for "add"):

```
sequenceDiagram
    participant Client
    participant Transport
    participant FastMCP as FastMCP Server
    participant TM as ToolManager
    participant YourCode as Your @mcp.tool function

    Client->>Transport: JSON-RPC Request (tools/call, name="add", args={"a": 5, "b": 3})
    Transport->>FastMCP: Raw message received
    FastMCP->>FastMCP: Parse message, identify method "tools/call"
    FastMCP->>TM: Route call to ToolManager handler
    TM->>TM: Look up tool "add"
    TM->>TM: Validate arguments {"a": 5, "b": 3} using schema
    TM->>YourCode: Call add(a=5, b=3) (injecting Context if requested)
    YourCode-->>TM: Return 8
    TM-->>FastMCP: Return result 8 (formatted as MCP content)
    FastMCP->>Transport: JSON-RPC Response (result=..., id=...)
    Transport-->>Client: Response sent
```

1. **Client Sends:** The client sends a JSON-RPC message over the Transport.
2. **FastMCP Receives:** The `FastMCP` server (via its underlying `MCPServer` and Transport handler) receives and parses the message.
3. **Routing:** FastMCP identifies the method (`tools/call`) and routes it to its internal `call_tool` handler.
4. **Manager Interaction:** The `call_tool` handler uses the `ToolManager` to find the registered `"add"` tool.
5. **Validation:** The `ToolManager` validates the client-provided arguments (`{"a": 5, "b": 3}`) against the schema derived from your function's type hints.
6. **Function Execution:** The `ToolManager` calls your original `add` function with the validated arguments.
7. **Result Formatting:** The return value from your function (`8`) is formatted into the standard MCP `CallToolResult` structure (which contains a list of content objects).
8. **Response:** FastMCP formats the result into a JSON-RPC response message and sends it back to the client.

You can see parts of this logic in the `src/mcp/server/fastmcp/server.py` file, which contains the `FastMCP` class and the implementations of the core MCP handlers (`list_tools`, `call_tool`, etc.) that delegate to the managers (`ToolManager`, `ResourceManager`, `PromptManager`) found in their respective subdirectories (`src/mcp/server/fastmcp/tools/`, `src/mcp/server/fastmcp/resources/`, `src/mcp/server/fastmcp/prompts/`).

This layered design allows you to build powerful MCP servers by simply defining

Python functions and decorating them, while FastMCP takes care of all the protocol and communication details.

Conclusion

FastMCP is the recommended way for beginners to build MCP servers with the `python-sdk`. It provides a high-level, decorator-based interface that abstracts away the complexities of the MCP protocol and transport handling. By using `@mcp.resource`, `@mcp.tool`, and `@mcp.prompt`, you can easily define your server's capabilities using standard Python functions, focusing on your application logic rather than communication mechanics.

Now that we understand how FastMCP simplifies server creation, let's take a step back and look at the underlying concept of a **Session**, which is fundamental to both client and server communication in MCP.

Next Chapter: Session

Welcome back to the `python-sdk` tutorial! In the last couple of chapters, we looked at how to build an MCP client using the Client Session and how to build an MCP server using FastMCP Server, which exposes Resources, Tools, and Prompts.

You saw that both the client and the server sides involved an object that seemed to represent the connection itself: the `ClientSession` on the client side and the underlying mechanism used by `FastMCP` on the server side.

In this chapter, we'll dive into the core concept that powers this connection on *both* sides: the **Session**.

What Problem Does the Session Solve?

Imagine a conversation between two people. It's not just one person talking and the other listening forever. It's a back-and-forth exchange:

- Person A asks a question.
- Person B listens, processes the question, and provides an answer.
- Person A might then ask another question, or Person B might volunteer some information (like "Hey, guess what happened!").

This ongoing, stateful exchange is similar to how an MCP client and server communicate. It's not just sending a single request and getting a single response like a simple HTTP GET. An MCP connection is a persistent channel where:

- The client sends requests (like `tools/call` or `resources/read`).
- The server sends responses to those specific requests.
- The server can also send independent notifications (like `notifications/logging/message` or `notifications/resources/updated`) that aren't tied to a specific client request.

- The client can also send requests (like `sampling/createMessage`) and notifications (like `notifications/progress`) to the server.

Managing this complex, bidirectional flow of messages, ensuring that responses are correctly matched to the requests that triggered them, and handling asynchronous notifications is tricky. You need an object that keeps track of the state of the conversation.

This is where the **Session** abstraction comes in.

The Solution: The `BaseSession`

The `python-sdk` provides a core class, `BaseSession`, that implements the fundamental logic for managing this stateful communication channel over any underlying message streams.

Think of `BaseSession` as the **engine** that drives the MCP conversation. It doesn't know *what* specific requests or notifications mean (like "call this tool" or "this resource updated"), but it knows *how* to send and receive messages, assign unique IDs to requests, match incoming responses to those IDs, and route incoming notifications.

`BaseSession` provides the mechanics for:

1. **Sending Requests:** Packaging a request object into a JSON-RPC message with a unique ID and sending it down the write stream.
2. **Waiting for Responses:** Keeping track of the unique ID and waiting for an incoming message with a matching ID.
3. **Matching Responses:** When a message arrives with an ID, checking if there's a pending request with that ID and delivering the response (or error) to the code that sent the request.
4. **Handling Notifications:** When a message arrives without an ID (a notification), routing it to a separate handler.
5. **Managing Streams:** Operating on top of generic `anyio` read and write streams, which are provided by the specific Transport being used (like stdio or SSE).

`BaseSession` is designed to be subclassed. The `python-sdk` provides two main subclasses:

- **`ClientSession`:** (Which you saw in Chapter 5) This subclass adds client-specific methods like `list_tools()`, `call_tool()`, `read_resource()`, etc. These methods use the `BaseSession`'s `send_request` internally. It also defines how to handle incoming *server* requests and notifications.
- **`ServerSession`:** (Used internally by `FastMCP` and the lower-level `Server` class) This subclass defines how to handle incoming *client* requests (like `tools/call`, `resources/read`) and notifications (like `notifications/initialized`). It also provides server-specific methods

like `send_log_message` or `send_resource_updated`, which use the `BaseSession`'s `send_notification` internally.

So, while you primarily interact with `ClientSession` on the client side and define handlers for incoming requests/notifications on the server side (often via `FastMCP` decorators), it's the `BaseSession` that's doing the heavy lifting of managing the message flow beneath the surface.

How `ClientSession` and `ServerSession` Use `BaseSession`

Let's look at how the client and server sessions build upon the `BaseSession`.

`ClientSession` (Building on `BaseSession`)

As you saw in Chapter 5, you use `ClientSession` methods like `await session.list_tools()`.

Internally, the `ClientSession`'s `list_tools` method does something like this:

```
# Simplified snippet from ClientSession (src/mcp/client/session.py)
# This is NOT the full code, just illustrates the concept

async def list_tools(self) -> types.ListToolsResult:
    """Send a tools/list request."""
    # 1. Create the specific MCP request object
    request_obj = types.ClientRequest(
        types.ListToolsRequest(method="tools/list")
    )

    # 2. Use the BaseSession's send_request method
    # BaseSession handles adding ID, sending JSON-RPC, waiting for response
    # and parsing the result into the expected type (types.ListToolsResult)
    result = await self.send_request(
        request_obj,
        types.ListToolsResult, # Tell BaseSession what type to expect back
    )
    return result

# ClientSession also overrides BaseSession's _received_notification
# to handle server notifications like logging messages.
async def _received_notification(
    self, notification: types.ServerNotification
) -> None:
    """Handle notifications from the server."""
    match notification.root:
        case types.LoggingMessageNotification(params=params):
            # Call the logging callback provided during ClientSession init
```

```
            await self._logging_callback(params)
        # ... handle other server notifications ...
```

This shows that `ClientSession` methods are thin wrappers around `BaseSession.send_request`, providing the correct MCP request object and specifying the expected response type. `ClientSession` also implements `_received_notification` to process incoming notifications from the server.

`ServerSession` (Building on `BaseSession`)

On the server side, when a client sends a request (like `tools/call`), the `ServerSession` (which `FastMCP` uses internally) receives it. The `BaseSession`'s receive loop detects it's an incoming request and calls the `ServerSession`'s `_received_request` method.

The `ServerSession`'s `_received_request` method then identifies the type of request and routes it to the appropriate handler (which, in `FastMCP`, is linked to your decorated function).

```
# Simplified snippet from ServerSession (src/mcp/server/session.py)
# This is NOT the full code, just illustrates the concept

async def _received_request(
    self, responder: RequestResponder[types.ClientRequest, types.ServerResult]
):
    """Handle incoming requests from the client."""
    # BaseSession's receive loop calls this when a request arrives.
    # The 'responder' object is used to send the response later.

    match responder.request.root:
        case types.InitializeRequest(params=params):
            # Handle the initial handshake request
            self._client_params = params # Store client capabilities
            with responder: # Use responder as context manager for cancellation
                await responder.respond(
                    types.ServerResult(
                        types.InitializeResult(
                            # ... send server capabilities ...
                        )
                    )
                )
        case types.CallToolRequest(params=params):
            # Handle a tool call request
            # In FastMCP, this would route to the ToolManager
            # which would find and call your @mcp.tool function
            with responder:
                 # Call the appropriate handler function (e.g., your add tool)
```

```
                # Get the result
                tool_result = await self._handle_tool_call(params.name, params.arguments)
                # Send the result back using the responder
                await responder.respond(types.ServerResult(tool_result))
        # ... handle other client requests (resources/read, prompts/get, etc.) ...

# ServerSession also provides methods to send notifications to the client
async def send_log_message(
    self, level: types.LoggingLevel, data: Any, logger: str | None = None
) -> None:
    """Send a log message notification."""
    # Use the BaseSession's send_notification method
    await self.send_notification(
        types.ServerNotification(
            types.LoggingMessageNotification(
                method="notifications/message",
                params=types.LoggingMessageNotificationParams(
                    level=level,
                    data=data,
                    logger=logger,
                ),
            ),
        )
    )
```

This shows that `ServerSession` overrides `BaseSession._received_request` to process incoming client requests and uses the `RequestResponder` object provided by `BaseSession` to send the final response. It also provides server-specific methods like `send_log_message` which are wrappers around `BaseSession.send_notification`.

Behind the Scenes: The `BaseSession` Flow

Let's visualize the core message flow managed by `BaseSession`.

Imagine a client calling `session.list_tools()` and the server responding.

```
sequenceDiagram
    participant ClientCode as Your Client Code
    participant ClientSession
    participant BaseSession_C as Client BaseSession
    participant Transport as Streams
    participant BaseSession_S as Server BaseSession
    participant ServerSession
    participant ServerCode as Your Server Code

    ClientCode->>ClientSession: await list_tools()
```

```
ClientSession->>BaseSession_C: send_request(ListToolsRequest, ListToolsResult)
BaseSession_C->>BaseSession_C: Generate unique ID, create JSON-RPC Request
BaseSession_C->>Transport: Write JSON-RPC Request (ID=X)
Transport->>BaseSession_S: Read JSON-RPC Request (ID=X)
BaseSession_S->>BaseSession_S: Detects incoming Request (ID=X)
BaseSession_S->>ServerSession: Call _received_request(Responder(ID=X, Request=ListToolsR
ServerSession->>ServerCode: Route to list_tools handler (e.g., via FastMCP)
ServerCode-->>ServerSession: Return ListToolsResult data
ServerSession->>BaseSession_S: responder.respond(ServerResult(ListToolsResult data))
BaseSession_S->>BaseSession_S: Create JSON-RPC Response (ID=X)
BaseSession_S->>Transport: Write JSON-RPC Response (ID=X)
Transport->>BaseSession_C: Read JSON-RPC Response (ID=X)
BaseSession_C->>BaseSession_C: Detects incoming Response (ID=X)
BaseSession_C->>BaseSession_C: Match ID X to pending request
BaseSession_C->>BaseSession_C: Parse JSON-RPC result into ListToolsResult object
BaseSession_C-->>ClientSession: Return ListToolsResult object
ClientSession-->>ClientCode: Return ListToolsResult object
```

Key mechanisms within `BaseSession`:

- **Unique IDs:** Every request sent gets a unique ID.
- **`_response_streams`:** A dictionary (`dict[RequestId, MemoryObjectSendStream]`) is maintained. When a request is sent, a small in-memory stream is created and stored in this dictionary, keyed by the request ID. The `send_request` method then waits to *receive* something from this stream.
- **`_receive_loop`:** A background task runs constantly, reading messages from the incoming stream (`read_stream`).
- **Routing Incoming Messages:** When `_receive_loop` gets a message:
  - If it has an ID and is a response/error, it looks up the corresponding stream in `_response_streams` and sends the message to that stream, unblocking the waiting `send_request`.
  - If it has an ID and is a request, it wraps it in a `RequestResponder` and calls the subclass's (`ClientSession` or `ServerSession`) `_received_request` method.
  - If it has no ID (a notification), it calls the subclass's `_received_notification` method.
- **`RequestResponder`:** On the server side, this object represents an incoming request that needs a response. It holds the request details and has a `respond()` method that uses `BaseSession._send_response` to send the reply with the correct ID. It also manages cancellation scopes.

You can see the `BaseSession` implementation details in `src/mcp/shared/session.py`. It uses `anyio` for asynchronous concurrency and streams, and the Pydantic models from `mcp.types` for parsing and validating the JSON-RPC messages.

Conclusion

The `BaseSession` class is the fundamental building block for managing the stateful, bidirectional communication channel in the `python-sdk`. It handles the low-level details of sending and receiving JSON-RPC messages, matching requests to responses using unique IDs, and routing notifications. The `ClientSession` and `ServerSession` classes inherit from `BaseSession`, adding the specific methods and request/notification handling logic needed for each side of the connection.

Understanding the Session helps clarify how the high-level `ClientSession` methods and `FastMCP` server handlers are connected to the underlying message exchange.

Now that we understand the core communication logic provided by the Session, let's look at how these messages actually travel between the client and server – the **Transports**.

Next Chapter: Transports

Welcome back to the `python-sdk` tutorial! In our journey so far, we've learned about the core concepts of the Model Context Protocol (MCP): Resources, Tools, and Prompts on the server side, and how to interact with them using the Client Session. We also peeked behind the curtain at the fundamental Session object that manages the message flow.

In this chapter, we'll explore **Transports**. If the Session is the engine that understands *how* to manage the MCP conversation (sending requests, receiving responses, handling notifications), the Transport is the **pipe** or **cable** through which the actual messages travel between the client and the server.

What Problem Do Transports Solve?

Imagine you've built a fantastic MCP server using FastMCP with useful Tools and Resources. Now you want different types of client applications to connect to it:

- A command-line interface (CLI) tool that runs on the same machine.
- A web-based application running in a browser.
- A desktop application like Claude Desktop.

Each of these clients might need to communicate with the server using a different underlying technology:

- The CLI tool might simply talk over standard input and output (`stdin`/`stdout`).
- The web application needs to communicate over HTTP, perhaps using technologies like Server-Sent Events (SSE) or WebSockets.
- A desktop application might also use stdio or potentially a local network connection.

The MCP protocol defines the *format* of the messages (JSON-RPC) and the *sequence* of the conversation (handshake, requests, responses, notifications), but it doesn't dictate the specific low-level communication method.

This is where **Transports** come in.

The Solution: Transports

A **Transport** in the `python-sdk` is the layer responsible for sending and receiving the raw MCP messages (which are JSON-RPC messages, typically represented as strings or bytes) over a specific communication channel.

Think of a Transport as an adapter. It takes the structured JSON-RPC messages from the Session and sends them using the rules of its specific medium (writing to `stdout`, sending an HTTP POST request, sending a WebSocket frame). On the receiving end, it reads from the medium (reading from `stdin`, receiving an SSE event, receiving a WebSocket frame), parses the raw data into a JSON-RPC message, and passes it up to the Session.

The key benefit is that the core MCP logic (the Session, your FastMCP handlers, your Client Session calls) remains completely independent of the Transport. You can swap out the Transport without changing your application logic.

The `python-sdk` provides built-in support for several common transports:

- **`stdio`:** Communicates over standard input (`stdin`) and standard output (`stdout`). This is very common for CLI tools or when one process launches another and communicates directly.
- **`SSE` (Server-Sent Events):** A mechanism for a server to push updates to a client over an HTTP connection. In MCP, this is often combined with HTTP POST requests from the client back to the server. Useful for web-based clients where the server needs to send notifications.
- **`WebSocket`:** Provides a full-duplex, real-time communication channel over a single TCP connection. Also useful for web-based or desktop clients needing low-latency, bidirectional communication.

Each transport is implemented as an asynchronous context manager that, when entered, provides the necessary `anyio` read and write streams for the Session to use.

Using Transports

Let's see how you use different transports on both the client and server sides.

Client Side Transports

On the client side, you need a transport helper function that connects to the server using a specific method and gives you the `(read_stream, write_stream)` tuple needed to create a ClientSession.

We already saw the `stdio_client` in Chapter 5: Client Session:

```python
# client_stdio.py
import asyncio
from mcp import ClientSession
from mcp.client.stdio import stdio_client # This is the stdio transport helper
from mcp import StdioServerParameters

async def main():
    server_params = StdioServerParameters(
        command="python",
        args=["server.py"], # Assuming server.py uses stdio transport
    )

    # Use the stdio_client transport helper
    async with stdio_client(server_params) as (read_stream, write_stream):
        # The ClientSession uses the streams provided by the transport
        async with ClientSession(read_stream, write_stream) as session:
            print("Client connected via stdio.")
            await session.initialize()
            print("Session initialized.")
            # Now you can use session.list_tools(), session.call_tool(), etc.
            # ... (interaction logic remains the same regardless of transport) ...
            print("Interaction complete.")

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

The `stdio_client` function (defined in `src/mcp/client/stdio/__init__.py`) handles spawning the server process and wrapping its `stdin` and `stdout` into the `anyio` streams that `ClientSession` expects.

For an SSE client, you would use the `sse_client` helper (defined in `src/mcp/client/sse.py`):

```python
# client_sse.py
import asyncio
from mcp import ClientSession
from mcp.client.sse import sse_client # This is the SSE transport helper

async def main():
    # Use the sse_client transport helper, pointing to the server's SSE endpoint
    # This assumes your server is running with the SSE transport enabled
    async with sse_client("http://localhost:8000/sse") as (read_stream, write_stream):
        # The ClientSession code is IDENTICAL to the stdio example
        async with ClientSession(read_stream, write_stream) as session:
            print("Client connected via SSE.")
            await session.initialize()
```

```
            print("Session initialized.")
            # ... (interaction logic remains the same) ...
            print("Interaction complete.")

if __name__ == "__main__":
    # Note: You'd need an ASGI server (like uvicorn) running server.py with SSE
    # asyncio.run(main())
    print("Run this client after starting a server with SSE transport.")
```

The `sse_client` function handles establishing the SSE connection (for receiving messages) and figuring out the endpoint for sending messages (via HTTP POST). It wraps this HTTP communication into the `anyio` streams for the `ClientSession`.

For a WebSocket client, you would use the `websocket_client` helper (defined in `src/mcp/client/websocket.py`):

```
# client_websocket.py
import asyncio
from mcp import ClientSession
from mcp.client.websocket import websocket_client # This is the WebSocket transport helper

async def main():
    # Use the websocket_client transport helper, pointing to the server's WebSocket endpoint
    # This assumes your server is running with the WebSocket transport enabled
    async with websocket_client("ws://localhost:8000/ws") as (read_stream, write_stream):
        # The ClientSession code is STILL IDENTICAL
        async with ClientSession(read_stream, write_stream) as session:
            print("Client connected via WebSocket.")
            await session.initialize()
            print("Session initialized.")
            # ... (interaction logic remains the same) ...
            print("Interaction complete.")

if __name__ == "__main__":
    # Note: You'd need an ASGI server (like uvicorn) running server.py with WebSocket
    # asyncio.run(main())
    print("Run this client after starting a server with WebSocket transport.")
```

The `websocket_client` function handles establishing the WebSocket connection and wrapping the incoming/outgoing WebSocket messages into the `anyio` streams.

The key takeaway for the client side: You choose the appropriate transport helper function (`stdio_client`, `sse_client`, `websocket_client`) based on how the server is accessible. Once you have the `(read_stream, write_stream)` pair from the transport helper, the rest of your client code using `ClientSession` is exactly the same. The transport abstracts away the low-level communication

details.

Server Side Transports

On the server side, the transport is responsible for listening for incoming connections or messages and providing the `(read_stream, write_stream)` pair to the server's Session logic (which is managed by `FastMCP` or the lower-level `Server` class).

When using FastMCP, the `mcp.run()` method handles starting the server with a default transport:

```
# server.py (from Chapter 6)
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("Demo Server")

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

# ... other resources/tools/prompts ...

if __name__ == "__main__":
    print("Starting FastMCP server (stdio transport)...")
    # By default, mcp.run() uses the stdio_server transport
    mcp.run()
    print("FastMCP server stopped.")
```

The `mcp.run()` method (defined in `src/mcp/server/fastmcp/server.py`) internally uses the `stdio_server` transport helper (defined in `src/mcp/server/stdio.py`) to get the streams and then runs the core server logic using those streams.

To run the same `FastMCP` server using the SSE transport, you can specify it:

```
# server_sse.py
from mcp.server.fastmcp import FastMCP
import uvicorn # Need an ASGI server to run SSE

mcp = FastMCP("Demo Server")

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

# ... other resources/tools/prompts ...
```

```python
if __name__ == "__main__":
    print("Starting FastMCP server (SSE transport)...")
    # mcp.run("sse") starts an ASGI server (uvicorn by default)
    # and mounts the SSE transport app
    mcp.run("sse", port=8000) # Specify port for HTTP server
    print("FastMCP server stopped.")
```

The `mcp.run("sse", ...)` call uses the `sse_server` transport implementation (defined in `src/mcp/server/sse.py`), which is designed to be mounted within an ASGI application (like one built with Starlette or FastAPI, and run by Uvicorn or Hypercorn). The `sse_server` transport provides two endpoints: one for the client to connect via GET (for the SSE stream) and one for the client to send messages via POST. It manages mapping these HTTP interactions to the internal `anyio` streams for the server's Session.

Similarly, for a WebSocket server, you would use the `websocket_server` transport (defined in `src/mcp/server/websocket.py`), which is also designed as an ASGI application:

```python
# server_websocket.py
from mcp.server.fastmcp import FastMCP
import uvicorn # Need an ASGI server to run WebSocket
from starlette.applications import Starlette
from starlette.routing import WebSocketRoute

mcp = FastMCP("Demo Server")

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

# ... other resources/tools/prompts ...

# The websocket_server transport is an ASGI application
from mcp.server.websocket import websocket_server

# You need to manually set up an ASGI app and route the transport
async def websocket_endpoint(websocket):
    # The websocket_server transport is an async context manager
    async with websocket_server(websocket.scope, websocket.receive, websocket.send) as (read
        # Get the underlying low-level server instance from FastMCP
        server_instance = mcp.get_server()
        # Run the server logic using the streams from the transport
        await server_instance.run(
            read_stream,
```

```
            write_stream,
            server_instance.create_initialization_options() # Get server capabilities
        )

app = Starlette(routes=[
    WebSocketRoute("/ws", endpoint=websocket_endpoint)
])

if __name__ == "__main__":
    print("Starting FastMCP server (WebSocket transport)...")
    uvicorn.run(app, host="0.0.0.0", port=8000)
    print("FastMCP server stopped.")
```

Here, because WebSocket requires a specific ASGI route type, we manually create a Starlette app and route the `/ws` path to an endpoint that uses the `websocket_server` transport context manager. Inside the context manager, we get the streams and then run the core `mcp` server logic using those streams.

The key takeaway for the server side: You choose the appropriate transport implementation (`stdio_server`, `sse_server`, `websocket_server`) based on how you want clients to connect. `FastMCP.run()` provides convenient shortcuts for stdio and SSE. For WebSocket or more complex ASGI integration, you might need to manually use the transport's context manager and the underlying `mcp.get_server().run()` method. Regardless of the transport, your core server logic (the `@mcp.tool`, `@mcp.resource`, `@mcp.prompt` functions) remains the same.

Behind the Scenes: How Transports Work

At its core, each transport implementation is an `asynccontextmanager` that sets up the communication channel and yields a tuple of `(read_stream, write_stream)`. These streams are typically `anyio` memory object streams or wrappers around underlying I/O primitives (like process pipes or network sockets).

Here's a simplified view of the stdio transport flow:

```
sequenceDiagram
    participant ClientProcess as Client Process
    participant ClientStdio as Client stdio_client
    participant ClientStreams as Client Read/Write Streams
    participant ClientSession
    participant ServerProcess as Server Process
    participant ServerStdio as Server stdio_server
    participant ServerStreams as Server Read/Write Streams
    participant ServerSession
    participant FastMCP as FastMCP Server
```

```
    ClientProcess->>ClientStdio: stdio_client(server_params)
    ClientStdio->>ServerProcess: Spawn process, pipe stdin/stdout
    ServerProcess->>ServerStdio: stdio_server() (wraps stdin/stdout)
    ServerStdio-->>ServerStreams: Yield (read_stream_S, write_stream_S)
    ServerStreams->>ServerSession: ServerSession(read_stream_S, write_stream_S)
    ServerSession->>FastMCP: FastMCP uses ServerSession

    ClientStdio-->>ClientStreams: Yield (read_stream_C, write_stream_C)
    ClientStreams->>ClientSession: ClientSession(read_stream_C, write_stream_C)

    ClientSession->>ClientStreams: Write JSON-RPC message
    ClientStreams->>ServerStreams: Message travels over pipes
    ServerStreams->>ServerSession: Read JSON-RPC message
    ServerSession->>FastMCP: Process message

    FastMCP->>ServerSession: Write JSON-RPC message
    ServerSession->>ServerStreams: Write JSON-RPC message
    ServerStreams->>ClientStreams: Message travels over pipes
    ClientStreams->>ClientSession: Read JSON-RPC message
    ClientSession->>ClientProcess: Deliver result/notification
```

The transport's job is to bridge the gap between the specific I/O mechanism (process pipes, HTTP, WebSockets) and the generic `anyio` streams that the Session understands.

Let's look at simplified snippets from the transport implementations:

`stdio_server` (src/mcp/server/stdio.py):

```python
# Simplified snippet
@asynccontextmanager
async def stdio_server(...):
    # Wrap sys.stdin/stdout in anyio AsyncFile and TextIOWrapper
    stdin = anyio.wrap_file(...)
    stdout = anyio.wrap_file(...)

    # Create in-memory streams for the Session to use
    read_stream_writer, read_stream = anyio.create_memory_object_stream(0)
    write_stream, write_stream_reader = anyio.create_memory_object_stream(0)

    # Start background tasks to move data:
    # stdin_reader: reads lines from stdin, parses JSON, sends to read_stream_writer
    # stdout_writer: reads JSONRPCMessages from write_stream_reader, formats as JSON lines,
    async with anyio.create_task_group() as tg:
        tg.start_soon(stdin_reader)
        tg.start_soon(stdout_writer)
        # Yield the streams for the Session
        yield read_stream, write_stream
```

sse_server (src/mcp/server/sse.py):

```python
# Simplified snippet
class SseServerTransport:
    # ... __init__ stores endpoint and a dict of writers per session ...

    @asynccontextmanager
    async def connect_sse(self, scope, receive, send):
        # Create in-memory streams for the Session
        read_stream_writer, read_stream = anyio.create_memory_object_stream(0)
        write_stream, write_stream_reader = anyio.create_memory_object_stream(0)

        # Generate session ID, store writer
        session_id = uuid4()
        self._read_stream_writers[session_id] = read_stream_writer

        # Start background tasks:
        # sse_writer: reads messages from write_stream_reader, formats as SSE events, sends
        async with anyio.create_task_group() as tg:
            # Start the ASGI response task (which uses sse_writer)
            response = EventSourceResponse(...)
            tg.start_soon(response, scope, receive, send)
            # Yield streams for the Session
            yield read_stream, write_stream

    async def handle_post_message(self, scope, receive, send):
        # This is a separate ASGI endpoint handler
        request = Request(scope, receive)
        session_id = request.query_params.get("session_id")
        writer = self._read_stream_writers.get(session_id) # Get the writer for this sessio

        body = await request.body() # Read the POST body
        try:
            message = types.JSONRPCMessage.model_validate_json(body) # Parse JSON
            await writer.send(message) # Send message to the session's read stream
            response = Response("Accepted", status_code=202)
        except Exception as exc:
            # Handle errors, send error to session's read stream
            response = Response("Error", status_code=400)
            await writer.send(exc)

        await response(scope, receive, send) # Send HTTP response
```

These snippets show how the transport implementations handle the specifics of their communication medium (reading lines from stdio, handling SSE/POST requests) and translate them into sending/receiving `JSONRPCMessage` objects via the standard `anyio` streams that the Session layer uses.

Conclusion

Transports are the essential layer that connects the abstract MCP Session logic to the physical communication channel. The `python-sdk` provides built-in transports for `stdio`, `SSE`, and `WebSocket`, allowing you to run the same MCP server or client logic over different mediums without modification. By providing standard `anyio` read/write streams, transports ensure that the core MCP implementation remains clean and transport-agnostic.

Now that we understand how messages travel, let's look at how information specific to the current request or session is made available to your server-side logic through the **Context**.

Next Chapter: Context

Welcome back to the `python-sdk` tutorial! In the last few chapters, we've explored how to build MCP servers using FastMCP, defining Resources, Tools, and Prompts. We also looked at the underlying Session object that manages the conversation over different Transports.

Now, let's address a crucial question for server developers: how do your Tool or Resource functions get information about the *current* request or the client session they are interacting with?

What Problem Does the Context Solve?

Imagine you've defined a Tool function using `@mcp.tool()` that performs a long-running task, like processing a list of files. While this task is running, you might want to:

- Send updates back to the client, like "Processing file X..." or "Finished file Y.". These are essentially log messages visible to the client application.
- Report the progress of the task so the client can display a progress bar (e.g., "5/10 files processed").
- Maybe the task needs to read the *content* of those files. As we learned in Chapter 2: Resources, files can be exposed as `file://` Resources. How does your tool function *read* a resource? It shouldn't open the file directly if the client is providing the resource via the MCP protocol. It needs to ask the *client* to provide the resource content.
- Perhaps your server initialized some shared resources during startup (using the `lifespan` feature we saw briefly in Chapter 6: FastMCP Server and the README). How does your tool function access that shared database connection or configuration object?
- You might want to know which specific client is connected or get a unique ID for the current request for logging or debugging purposes.

Your simple Python function (`def my_tool(...)`) doesn't automatically have access to this kind of information or these capabilities. It needs a way to receive them from the MCP server framework (FastMCP).

This is where the **Context** object comes in.

The Solution: The `Context` Object

The `Context` object is a special object provided by the `python-sdk` that gives your server-side handler functions (the functions decorated with `@mcp.tool()`, `@mcp.resource()`, or `@mcp.prompt()`) access to information and capabilities related to the current MCP request and the client session.

Think of the `Context` object as a **toolbox** handed to your function every time it's called by the MCP server. This toolbox contains tools for interacting *back* with the client and accessing request-specific information.

The `Context` object allows your handlers to perform actions like:

- Sending log messages (`ctx.info()`, `ctx.error()`, etc.)
- Reporting progress (`ctx.report_progress()`)
- Reading resources *via the client* (`ctx.read_resource()`)
- Accessing shared data from the server's lifespan (`ctx.request_context.lifespan_context`)
- Getting details about the request (`ctx.request_id`, `ctx.client_id`)
- Accessing the underlying Session object for advanced use (`ctx.session`)

How to Use the `Context`

Using the `Context` object in your FastMCP handlers is very simple. You just need to add a parameter to your function and give it a type hint of `Context`. FastMCP will automatically detect this type hint and inject the correct `Context` instance for the current request when your function is called.

Let's revisit the `long_task` tool example from the README:

```python
# server.py
from mcp.server.fastmcp import FastMCP, Context # Import Context

mcp = FastMCP("My App")

@mcp.tool()
async def long_task(files: list[str], ctx: Context) -> str: # Add ctx: Context parameter
    """Process multiple files with progress tracking and logging."""
    total_files = len(files)
    for i, file_path in enumerate(files):
        # Use the context to send an info log message to the client
        await ctx.info(f"Starting processing for file: {file_path}")

        # Use the context to report progress to the client
        # Progress is 0-indexed, so add 1 for display
        await ctx.report_progress(i + 1, total_files)
```

```
        try:
            # Use the context to read a resource (the file content) via the client
            # ctx.read_resource returns an iterable of ReadResourceContents
            # We'll assume the first item is the text content for simplicity here
            contents = await ctx.read_resource(f"file://{file_path}")
            file_content = ""
            for item in contents:
                if hasattr(item, 'content') and isinstance(item.content, str):
                    file_content += item.content # Accumulate text content

            # Simulate some processing
            processed_data = f"Processed content of {file_path}: {file_content[:50]}..." # 

            # Use the context to send a debug log message
            await ctx.debug(f"Finished processing {file_path}. Data snippet: {processed_data

        except Exception as e:
            # Use the context to send an error log message
            await ctx.error(f"Error processing file {file_path}: {e}")
            # In a real tool, you might raise the exception or return an error result

    await ctx.info("All files processed.")
    return "File processing complete."

# ... (add mcp.run() block if you want to run this directly) ...
```

In this example:

- We imported the `Context` class.
- We added a parameter named `ctx` with the type hint `Context` to the `long_task` function signature. The name `ctx` is just a convention; you could name it anything (e.g., `context_obj`, `mcp_ctx`).
- Inside the function, we call methods on the `ctx` object:
  - `await ctx.info(...)`: Sends a log message with level "info" to the client.
  - `await ctx.report_progress(...)`: Sends a progress notification to the client.
  - `await ctx.read_resource(...)`: Sends a `resources/read` request *to the client* asking it to provide the content of the specified URI. This is an `await` call because reading a resource is an asynchronous operation.

This pattern works for any function decorated with `@mcp.tool()`, `@mcp.resource()`, or `@mcp.prompt()`. If your function doesn't need the context, you simply omit the `Context` parameter.

Accessing Lifespan Context

If your FastMCP server was configured with a `lifespan` context manager (as shown in the README and briefly in Chapter 6), the shared object yielded by the lifespan manager is available via the `Context` object as well.

The `Context` object wraps an internal `RequestContext` object (which we'll see in the "Behind the Scenes" section). The lifespan context is a property of this `RequestContext`.

Here's how you would access it, assuming your lifespan yielded an object with a `db` attribute:

```python
# server.py (continued, assuming lifespan is set up)
from mcp.server.fastmcp import FastMCP, Context
# Assuming AppContext and lifespan are defined as in the README/Chapter 6
from fake_database import Database # Example dependency
from dataclasses import dataclass
from contextlib import asynccontextmanager
from collections.abc import AsyncIterator

@dataclass
class AppContext:
    db: Database

@asynccontextmanager
async def app_lifespan(server: FastMCP) -> AsyncIterator[AppContext]:
    db = await Database.connect() # Connect to your database
    try:
        yield AppContext(db=db)
    finally:
        await db.disconnect() # Disconnect on shutdown

mcp = FastMCP("My App", lifespan=app_lifespan) # Pass lifespan here

@mcp.tool()
async def query_db(query: str, ctx: Context) -> str: # Add ctx: Context
    """Tool that uses the database connection from lifespan."""
    # Access the lifespan context via ctx.request_context.lifespan_context
    app_context = ctx.request_context.lifespan_context
    db = app_context.db # Access the database object

    await ctx.info(f"Executing query: {query}")
    try:
        result = await db.query(query) # Use the database connection
        await ctx.info("Query executed successfully.")
        return str(result)
```

```
    except Exception as e:
        await ctx.error(f"Query failed: {e}")
        raise # Re-raise the exception to signal failure to the client

# ... (add mcp.run() block) ...
```

In this example, `ctx.request_context.lifespan_context` gives you access to the `AppContext` object that was created and yielded by your `app_lifespan` context manager when the server started. You can then access any resources (like the `db` connection) stored on that object.

Accessing Request Details

The `Context` object also provides easy access to basic information about the current request:

```
# server.py (continued)
from mcp.server.fastmcp import FastMCP, Context

mcp = FastMCP("My App")

@mcp.tool()
async def show_request_info(ctx: Context) -> str:
    """Shows information about the current request."""
    request_id = ctx.request_id # Get the unique ID for this request
    client_id = ctx.client_id   # Get the client ID (if provided by client)

    info_message = f"Request ID: {request_id}\n"
    if client_id:
        info_message += f"Client ID: {client_id}\n"
    else:
        info_message += "Client ID: Not provided\n"

    await ctx.info(f"Request info accessed: {info_message}")

    return info_message

# ... (add mcp.run() block) ...
```

`ctx.request_id` is a unique identifier generated by the server for each incoming request. `ctx.client_id` is an optional identifier that the client application can send during the initial handshake.

Behind the Scenes: How Context Works

Let's peek under the hood to understand what's happening when `FastMCP` injects the `Context`.

When an MCP client sends a request (like `tools/call` or `resources/read`) to the server, the underlying Session object receives and parses it. For each request, the `MCPServer` (which `FastMCP` uses internally) creates a `RequestContext` object.

The `RequestContext` is a simple dataclass that holds the essential details for that specific request:

- `request_id`: The unique ID of the request.
- `meta`: Any metadata sent with the request (like the `client_id`).
- `session`: A reference to the `ServerSession` object handling this connection.
- `lifespan_context`: A reference to the object yielded by the server's lifespan context manager.

You can see the definition of `RequestContext` in `src/mcp/shared/context.py`:

```
# Simplified snippet from src/mcp/shared/context.py
from dataclasses import dataclass
from typing import Any, Generic
from typing_extensions import TypeVar
from mcp.shared.session import BaseSession
from mcp.types import RequestId, RequestParams

SessionT = TypeVar("SessionT", bound=BaseSession[Any, Any, Any, Any, Any])
LifespanContextT = TypeVar("LifespanContextT")

@dataclass
class RequestContext(Generic[SessionT, LifespanContextT]):
    request_id: RequestId
    meta: RequestParams.Meta | None
    session: SessionT # Reference to the ServerSession
    lifespan_context: LifespanContextT # Reference to lifespan object
```

The `Context` object that you use in your `@mcp.tool` functions is a wrapper around this `RequestContext`. The `Context` class (defined in `src/mcp/server/fastmcp/server.py`) holds a reference to the `RequestContext` and provides user-friendly methods that delegate to the `RequestContext` or the underlying `ServerSession`.

```
# Simplified snippet from src/mcp/server/fastmcp/server.py
class Context(BaseModel, Generic[ServerSessionT, LifespanContextT]):
    _request_context: RequestContext[ServerSessionT, LifespanContextT] | None
    _fastmcp: FastMCP | None # Also holds a reference to the FastMCP instance

    # ... __init__ stores the request_context and fastmcp references ...

    @property
    def request_context(self) -> RequestContext[ServerSessionT, LifespanContextT]:
```

```
        # Provides access to the underlying RequestContext
        if self._request_context is None:
             raise ValueError("Context not available...")
        return self._request_context

    @property
    def request_id(self) -> str:
        # Delegates to RequestContext
        return str(self.request_context.request_id)

    @property
    def client_id(self) -> str | None:
        # Delegates to RequestContext.meta
        return (
            getattr(self.request_context.meta, "client_id", None)
            if self.request_context.meta
            else None
        )

    async def report_progress(self, progress: float, total: float | None = None) -> None:
        # Delegates to the session within the RequestContext
        progress_token = ... # Get token from meta
        await self.request_context.session.send_progress_notification(...)

    async def read_resource(self, uri: str | AnyUrl) -> Iterable[ReadResourceContents]:
        # Delegates back to the FastMCP instance's read_resource method
        assert self._fastmcp is not None
        return await self._fastmcp.read_resource(uri)

    async def log(self, level: Literal["debug", "info", "warning", "error"], message: str,
        # Delegates to the session within the RequestContext
        await self.request_context.session.send_log_message(...)

    # ... convenience log methods (debug, info, warning, error) ...
```

When `FastMCP` receives a request and determines which of your decorated functions to call, it inspects the function's signature. If it finds a parameter type-hinted as `Context`, it creates a `Context` instance, populating it with the `RequestContext` for the current request and a reference to itself (`FastMCP`), and passes that `Context` instance as the argument value for that parameter.

Here's a simplified sequence diagram:

```
sequenceDiagram
    participant Client
    participant Transport
    participant FastMCP as FastMCP Server
```

```
participant RequestContext as Request Context Object
participant Context as Context Object (for handler)
participant YourHandler as Your @mcp.tool function

Client->>Transport: JSON-RPC Request (tools/call, ...)
Transport->>FastMCP: Raw message received
FastMCP->>FastMCP: Parse message, identify handler
FastMCP->>RequestContext: Create RequestContext for this request (with session, lifespar
FastMCP->>Context: Create Context object, wrapping RequestContext and FastMCP
FastMCP->>YourHandler: Call YourHandler(..., ctx=Context object)
YourHandler->>Context: Call ctx.info(...)
Context->>RequestContext: Access session from RequestContext
RequestContext->>FastMCP: Call session.send_log_message(...)
FastMCP->>Transport: Send JSON-RPC Notification (logging/message)
Transport-->>Client: Notification received by client
YourHandler-->>FastMCP: Return result
FastMCP->>Transport: Send JSON-RPC Response
Transport-->>Client: Response received by client
```

This layered design ensures that your core application logic in the handler function is clean and focused on *what* it needs to do, while the `Context` object provides a standard, easy-to-use interface for interacting with the MCP environment and accessing request-specific data.

Conclusion

The `Context` object is a powerful feature in the `python-sdk` that provides your server-side handler functions with essential information and capabilities related to the current MCP request and session. By simply adding a parameter type-hinted with `Context` to your `@mcp.tool`, `@mcp.resource`, or `@mcp.prompt` functions, you gain access to features like sending logs and progress updates, reading resources via the client, accessing lifespan data, and retrieving request details. This abstraction keeps your core logic clean while enabling rich interaction with the MCP client.

Now that we've covered the core concepts of building and interacting with MCP servers, let's look at the command-line interface provided by the `python-sdk` to help you manage and run your MCP projects.

Next Chapter: CLI (mcp command)

Welcome back to the `python-sdk` tutorial! In the previous chapters, we've learned about the core concepts of the Model Context Protocol (MCP), how to build servers using FastMCP that expose Resources, Tools, and Prompts, and how the underlying Session, Transports, and Context work.

You now know how to write the Python code for your MCP server. But how do you easily run it, test it, or make it available to client applications like Claude

Desktop?

Running a Python script directly (`python server.py`) is simple, but it doesn't handle things like managing dependencies for the server process, launching it with a specific Transport, or integrating it into other applications.

This is where the **`mcp` command-line interface (CLI)** comes in.

What Problem Does the `mcp` Command Solve?

The `mcp` command is a helper tool provided by the `python-sdk` to simplify common tasks for developers working with MCP servers, especially those built with FastMCP.

Think of it as your **main entry point** for interacting with your MCP server project from the terminal. It wraps the complexities of running Python scripts with the correct environment and integrates with tools like the MCP Inspector or Claude Desktop.

The `mcp` command helps you:

1. **Develop and Debug:** Easily run your server in a special mode that connects it to a visual inspector tool, making it easy to see what capabilities your server exposes and test them interactively.
2. **Install:** Configure compatible client applications (like Claude Desktop) to automatically discover and run your server.
3. **Run Directly:** Execute your server script using a specific Transport without needing to write boilerplate code for launching the server process yourself.
4. **Manage Dependencies:** Ensure your server runs with the necessary Python packages installed, even if you're distributing it or running it in a different environment.

Let's look at the most common `mcp` commands.

Using the `mcp` Command

The `mcp` command is available in your terminal once you've installed the `mcp` package with the `[cli]` extra (e.g., `uv add "mcp[cli]"` or `pip install "mcp[cli]"`).

You can see the available commands by just typing `mcp` or `mcp --help`:

```
mcp

Usage: mcp [OPTIONS] COMMAND [ARGS]...

  MCP development tools

Options:
```

```
  --install-completion [bash|zsh|fish|powershell|pwsh]
                                  Install completion for the specified shell.
  --show-completion [bash|zsh|fish|powershell|pwsh]
                                  Show completion for the specified shell.
  --help                          Show this message and exit.

Commands:
  dev      Run a MCP server with the MCP Inspector.
  install  Install a MCP server in the Claude desktop app.
  run      Run a MCP server.
  version  Show the MCP version.
```

Let's focus on the `dev`, `install`, and `run` commands.

For these examples, we'll use the simple Quickstart server from the README and Chapter 6: FastMCP Server:

```python
# server.py
from mcp.server.fastmcp import FastMCP

# Create an MCP server
mcp = FastMCP("Demo")

# Add an addition tool
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

# Add a dynamic greeting resource
@mcp.resource("greeting://{name}")
def get_greeting(name: str) -> str:
    """Get a personalized greeting"""
    return f"Hello, {name}!"

# Note: We don't need the if __name__ == "__main__": mcp.run() block
# when using the mcp command, as the command handles running the server.
# However, keeping it doesn't hurt if you also want to run with `python server.py`.
```

1. Development Mode (`mcp dev`)

The `mcp dev` command is your best friend for building and debugging. It launches your server and connects it to the MCP Inspector, a separate tool that provides a visual interface for interacting with your server.

Use Case: You are actively developing your `server.py` file and want to quickly test your Tools, Resources, and Prompts and see the messages being exchanged.

How to use:

```
mcp dev server.py
```

What happens:

1. The `mcp` command uses `uv run` internally to execute your `server.py` script.
2. It automatically configures your server to use the `stdio` Transport (if it's a FastMCP server without a specified transport in `mcp.run()`).
3. It launches the MCP Inspector tool (which requires Node.js and `npx` to be installed, but `mcp dev` handles running it).
4. The Inspector connects to your running server via stdio.
5. You'll see the Inspector UI open in your browser, allowing you to list capabilities, call tools, read resources, etc.

Handling Dependencies: A key feature of `mcp dev` is its ability to manage dependencies for your server script using `uv`.

If your `server.py` requires packages like `pandas` or `httpx`, you can tell `mcp dev` to include them:

```
mcp dev server.py --with pandas --with httpx
```

This command will ensure `pandas` and `httpx` are available in the environment where your `server.py` is run.

If your server is part of a larger project with a `pyproject.toml` file, you can install that project in editable mode:

```
mcp dev server.py --with-editable .
```

This is very useful when your server code is spread across multiple files within your project.

Example Output (Console):

```
mcp dev server.py
```

```
DEBUG   cli: Starting dev server extra={'file': 'server.py', 'server_object': None, 'with_ed
DEBUG   cli: Building uv command: ['uv', 'run', '--with', 'mcp', 'mcp', 'run', 'server.py']
DEBUG   cli: Using npx command: npx
Starting FastMCP server...
```

(The console output from your server will appear here, and the Inspector UI will open in your browser).

The `mcp dev` command is the recommended way to test your server during development.

2. Claude Desktop Integration (`mcp install`)

The `mcp install` command makes your MCP server available within the Claude Desktop application.

Use Case: You have a finished or stable version of your server and want to use its capabilities directly within Claude Desktop, allowing the LLM to access your custom Tools, Resources, and Prompts.

How to use:

```
mcp install server.py
```

What happens:

1. The `mcp` command locates the configuration directory for Claude Desktop on your system.
2. It modifies a configuration file (`claude_desktop_config.json`) within that directory.
3. It adds an entry for your server, specifying how Claude Desktop should launch it. This launch command uses `uv run` internally, similar to `mcp dev`.
4. The entry includes the path to your `server.py` file and any specified dependencies or environment variables.
5. The next time you open Claude Desktop, it will detect this configuration and list your server as an available tool provider.

Customizing Installation:

- **Server Name:** By default, `mcp install` tries to use the `name` attribute of your `FastMCP` instance. If that's not available or you want a different name, use `--name`:

  ```
  mcp install server.py --name "My Custom Server"
  ```

- **Dependencies:** Just like `mcp dev`, you can specify dependencies using `--with` and `--with-editable`. These dependencies will be installed by `uv` when Claude Desktop launches your server:

  ```
  mcp install server.py --with pandas --with-editable .
  ```

- **Environment Variables:** You can pass environment variables to your server process using `--env-var` (or `-v`) or load them from a `.env` file using `--env-file` (or `-f`). These variables are stored in the Claude config and set when the server is launched:

  ```
  mcp install server.py -v API_KEY=abc123 -v DB_URL=postgres://user:pass@host/db
  mcp install server.py -f .env
  ```

  Environment variables added this way are persistent in the Claude config. If you run `mcp install` again with the same server name, new `--env-var`

or `--env-file` values will *merge* with existing ones, with the new values taking precedence.

Example Output (Console):

```
mcp install server.py --name "My Demo Server" -v MY_SETTING=some_value
```

```
DEBUG   cli: Installing server extra={'file': 'server.py', 'server_name': 'My Demo Server',
DEBUG   cli: Could not import server (likely missing dependencies), using file name extra={'
INFO    claude: Added server 'My Demo Server' to Claude config extra={'config_file': '/Users
INFO    cli: Successfully installed My Demo Server in Claude app
```

After running this, restart Claude Desktop, and you should see your server listed.

3. Direct Execution (`mcp run`)

The `mcp run` command executes your server script directly.

Use Case: You want to run your server as a standalone process, perhaps for a custom deployment or to test a specific Transport like SSE.

How to use:

```
mcp run server.py
```

What happens:

1. The `mcp` command uses `uv run` internally to execute your `server.py` script.
2. It calls the `run()` method on the `FastMCP` instance found in your script.
3. By default, `mcp.run()` uses the `stdio` Transport.

Specifying Transport: You can tell `mcp run` which transport to use with the `--transport` (or `-t`) option. This is useful if your server script is written to support multiple transports but defaults to stdio in its `if __name__ == "__main__":` block.

```
mcp run server.py --transport sse
```

This would attempt to run your server using the SSE transport, typically starting an HTTP server (like Uvicorn) if your script is set up for it (as shown in Chapter 8: Transports).

Note: Unlike `mcp dev` and `mcp install`, `mcp run` does *not* automatically handle dependencies using `--with` or `--with-editable`. It assumes your Python environment already has the necessary packages installed. For dependency management, `mcp dev` and `mcp install` are preferred.

Example Output (Console):

```
mcp run server.py
```

```
DEBUG   cli: Running server extra={'file': 'server.py', 'server_object': None, 'transport':
Starting FastMCP server... # Output from your server.py
```

The server will continue running until you stop it (e.g., with Ctrl+C).

Behind the Scenes: How the `mcp` Command Works

The `mcp` command is a Python application built using the `typer` library, which makes it easy to create command-line interfaces.

When you run `mcp <command> <args>`, the `mcp` script (located in `src/mcp/cli/cli.py`) is executed. It uses functions from `src/mcp/cli/cli.py` and `src/mcp/cli/claude.py` to perform the requested action.

Here's a simplified look at what happens for each command:

`mcp dev`

```
sequenceDiagram
    participant User as Terminal User
    participant mcpCLI as mcp Command
    participant uv as uv
    participant npx as npx
    participant YourServer as Your server.py
    participant Inspector as MCP Inspector

    User->>mcpCLI: mcp dev server.py --with pandas
    mcpCLI->>mcpCLI: Parse args (file='server.py', with_packages=['pandas'])
    mcpCLI->>mcpCLI: Build uv command: ['uv', 'run', '--with', 'mcp', '--with', 'pandas', 'm
    mcpCLI->>npx: Execute npx @modelcontextprotocol/inspector <uv command>
    npx->>uv: Execute uv run ...
    uv->>YourServer: Run server.py (with dependencies installed)
    YourServer->>YourServer: Start FastMCP server (stdio transport)
    YourServer->>Inspector: Communicate via stdio
    Inspector-->>User: Open UI in browser
```

The `dev` command in `src/mcp/cli/cli.py` uses `_build_uv_command` to construct the `uv run` part, then executes `npx @modelcontextprotocol/inspector` followed by that `uv` command using `subprocess.run`.

```python
# Simplified snippet from src/mcp/cli/cli.py
@app.command()
def dev(...):
    # ... parse args ...
    uv_cmd = _build_uv_command(file_spec, with_editable, with_packages)
    npx_cmd = _get_npx_command() # Find npx executable
    subprocess.run([npx_cmd, "@modelcontextprotocol/inspector"] + uv_cmd, check=True, shell=
```

mcp install

```
sequenceDiagram
    participant User as Terminal User
    participant mcpCLI as mcp Command
    participant ClaudeUtils as src/mcp/cli/claude.py
    participant ClaudeConfig as claude_desktop_config.json

    User->>mcpCLI: mcp install server.py --name "My Server" -v KEY=VALUE
    mcpCLI->>mcpCLI: Parse args (file='server.py', name='My Server', env_vars=['KEY=VALUE'])
    mcpCLI->>ClaudeUtils: update_claude_config('server.py', 'My Server', env_vars={'KEY':'VA
    ClaudeUtils->>ClaudeUtils: Find Claude config path
    ClaudeUtils->>ClaudeConfig: Read existing config
    ClaudeUtils->>ClaudeUtils: Build uv command: ['uv', 'run', '--with', 'mcp', 'mcp', 'run'
    ClaudeUtils->>ClaudeUtils: Create server entry in config (command='uv', args=[...], env=
    ClaudeUtils->>ClaudeConfig: Write updated config
    ClaudeUtils-->>mcpCLI: Return success
    mcpCLI-->>User: Print success message
```

The `install` command in `src/mcp/cli/cli.py` delegates most of the work to the `update_claude_config` function in `src/mcp/cli/claude.py`. This function handles finding the config file, reading/writing JSON, building the `uv run` command string, and merging environment variables.

```python
# Simplified snippet from src/mcp/cli/claude.py
def update_claude_config(...):
    config_dir = get_claude_config_path() # Find config dir
    config_file = config_dir / "claude_desktop_config.json"
    config = json.loads(config_file.read_text())
    # ... merge env_vars ...
    args = _build_uv_command(file_spec, with_editable, with_packages) # Build uv command
    server_config = {"command": "uv", "args": args}
    if env_vars:
        server_config["env"] = env_vars
    config["mcpServers"][server_name] = server_config
    config_file.write_text(json.dumps(config, indent=2))
    # ... log success ...
```

mcp run

```
sequenceDiagram
    participant User as Terminal User
    participant mcpCLI as mcp Command
    participant uv as uv
    participant YourServer as Your server.py

    User->>mcpCLI: mcp run server.py --transport sse
```

```
    mcpCLI->>mcpCLI: Parse args (file='server.py', transport='sse')
    mcpCLI->>mcpCLI: Build uv command: ['uv', 'run', '--with', 'mcp', 'mcp', 'run', 'server.
    mcpCLI->>uv: Execute uv run ...
    uv->>YourServer: Run server.py (assuming dependencies are installed)
    YourServer->>YourServer: Start FastMCP server (sse transport)
    YourServer-->>User: Print server output
```

The `run` command in `src/mcp/cli/cli.py` also uses `_build_uv_command` but executes it directly using `subprocess.run`, without involving the Inspector or modifying external config files. It passes the `--transport` argument through to the internal `mcp run` call.

```python
# Simplified snippet from src/mcp/cli/cli.py
@app.command()
def run(...):
    # ... parse args ...
    # Note: _build_uv_command here doesn't include --with-editable or --with
    # as run assumes dependencies are handled externally.
    uv_cmd = ["uv", "run", "--with", "mcp", "mcp", "run", file_spec]
    if transport:
        uv_cmd.extend(["--transport", transport])
    subprocess.run(uv_cmd, check=True)
```

The `mcp` command effectively acts as a convenient wrapper around `uv run` and other tools, providing a user-friendly interface for common MCP development and deployment tasks.

Conclusion

The `mcp` command-line interface is a valuable tool for developers using the `python-sdk`. It simplifies running your MCP servers in development mode with the Inspector (`mcp dev`), integrating them into client applications like Claude Desktop (`mcp install`), and running them directly with specific transports (`mcp run`). By handling dependency management via `uv` and abstracting away launch details, the `mcp` command allows you to focus on building your server's capabilities.

This concludes our core tutorial on the `python-sdk`. You now have a solid understanding of the MCP protocol, how to build servers with FastMCP, interact with them from a client perspective using the Client Session, and leverage the `mcp` CLI for development and deployment.

You can now explore the examples provided in the SDK, dive deeper into the specification, or start building your own powerful MCP servers!

Back to Table of Contents

The MCP Python SDK provides tools to build applications that communicate using the **Model Context Protocol (MCP)**. Developers can create

FastMCP Servers to expose data (*Resources*), actions (*Tools*), and interaction templates (*Prompts*) to LLMs, or build **Client Sessions** to interact with existing MCP servers. The SDK handles the underlying communication channels (*Transports*) and protocol details, allowing developers to focus on defining their server's capabilities or consuming those of others. The **CLI** simplifies development tasks like running servers and integrating with clients.

Source Repository: https://github.com/modelcontextprotocol/python-sdk

```
flowchart TD
    A0["MCP Protocol"]
    A1["FastMCP Server"]
    A2["Client Session"]
    A3["Resources"]
    A4["Tools"]
    A5["Prompts"]
    A6["Transports"]
    A7["Session"]
    A8["Context"]
    A9["CLI (mcp command)"]
    A0 -- "Implemented By" --> A1
    A0 -- "Implemented By" --> A2
    A0 -- "Defines Messages" --> A7
    A1 -- "Manages" --> A3
    A1 -- "Manages" --> A4
    A1 -- "Manages" --> A5
    A1 -- "Provides" --> A8
    A1 -- "Provides App" --> A6
    A9 -- "Runs" --> A1
    A9 -- "Uses" --> A6
    A2 -- "Uses" --> A6
    A2 -- "Accesses" --> A3
    A2 -- "Calls" --> A4
    A2 -- "Accesses" --> A5
    A7 -- "Uses Streams" --> A6
    A8 -- "Accesses" --> A7
    A8 -- "Reads" --> A3
```

Chapters

1. MCP Protocol
2. Resources
3. Tools
4. Prompts
5. Client Session
6. FastMCP Server
7. Session

8. Transports
9. Context
10. CLI (mcp command)