python-sdk Tutorial

This is the very first chapter, so we'll start from the beginning! Imagine you want to build a program that talks to a powerful server, a server that can answer questions, process information, and even help you create things. That's exactly what the `python-sdk` project helps you do. The `ClientSession` is the essential tool that makes this communication possible.

Think of `ClientSession` as your personal phone line to the server. It handles all the technical details of setting up the connection, sending your requests (like making a phone call), receiving the server's responses (like hearing the other person's answer), and keeping the conversation flowing smoothly.

Our central use case will be sending a simple "ping" request to the server to check if it's working. This is like calling someone just to say "hello".

Let's see how to use it:

```
import anyio
from mcp.client.session import ClientSession
from anyio.streams.memory import MemoryObjectReceiveStream, MemoryObjectSendStream # simpli

async def main():
    # Simulate connection streams (we'll learn about real connections later).
    read_stream = MemoryObjectReceiveStream[None]() # simplified for this example
    write_stream = MemoryObjectSendStream[None]() # simplified for this example

    async with ClientSession(read_stream, write_stream) as session:
        # Send a ping request
        result = await session.send_ping()
        print(f"Ping result: {result}") # This will print the server's response

anyio.run(main)
```

This code snippet shows the basic usage of `ClientSession`. We create a `ClientSession` object with simulated input and output streams (we'll learn about real streams like HTTP connections later). Then, we use `session.send_ping()` to send our "hello" to the server. The `await` keyword means we wait for the server's response before continuing. Finally, we print the response to the console.

What happens inside?

Let's visualize the interaction with a sequence diagram:

```
sequenceDiagram
    participant Client
    participant ClientSession
    participant Network
    participant Server
```

```
Client->>ClientSession: Create Session and send ping()
activate ClientSession
ClientSession->>Network: Send ping request
activate Network
Network->>Server: ping request
activate Server
Server->>Network: pong response
deactivate Server
Network->>ClientSession: pong response
deactivate Network
ClientSession->>Client: Return pong result
deactivate ClientSession
```

The `ClientSession` manages the communication behind the scenes. It handles the details of packaging the "ping" request, sending it over the network (via `Network`, a placeholder for the actual connection), receiving the "pong" response, and unpacking it.

A deeper look at the `send_ping()` function (simplified):

```python
async def send_ping(self) -> types.EmptyResult:
    # This is a simplified version
    # The actual implementation handles message packaging and sending etc.
    # ... some code to pack the message...
    await self.send_request(types.ClientRequest(types.PingRequest(method="ping")), types
    # ... some code to receive and unpack the response ...
    return types.EmptyResult() # This is a placeholder
```

The `send_ping()` method in the `ClientSession` class (located in `src/mcp/client/session.py`) uses the underlying `send_request()` method (which we won't cover in detail here) to actually send the message and receive the response.

In this chapter, we learned the fundamental role of `ClientSession` in facilitating communication with the server. We saw how to use it for a simple "ping" request. In the next chapter, FastMCP Server, we'll delve into the server-side of the communication and see how it responds to our requests.

In the previous chapter, ClientSession, we learned how to send a "ping" request to a server using the `python-sdk`. Now, let's explore the other side of the coin: the server itself, specifically the `FastMCP` server. This is the heart of our application, responsible for receiving requests from clients like the one we built in Chapter 1 and sending back appropriate responses.

Imagine a restaurant. The `ClientSession` is like a customer placing an order. The `FastMCP` server is the entire kitchen staff: it receives the order, processes it (prepares the food), and delivers the finished dish (the response) back to the customer. It manages various resources (ingredients), tools (cooking equipment), and recipes (prompts) to fulfill all orders efficiently.

Our central use case will be building a simple server that can add two numbers together.

Key Concepts: Resources, Tools, and Prompts

To build powerful servers, `FastMCP` uses three key concepts:

- **Resources:** These are pieces of information or data that your server exposes to clients. Think of them as the restaurant's ingredients: pre-prepared items that can be used in many dishes (responses). You can define resources that are static (always the same) or dynamic (change over time).
- **Tools:** These are functions your server makes available to process data. They're like the chef's cooking tools and techniques. They might involve computations, database queries, or API calls to external services.
- **Prompts:** These are reusable templates for interactions with Language Models (LLMs). Think of prompts as the restaurant's recipes: instructions for combining ingredients and tools to create a specific dish.

Let's see how to use these to create our addition server.

Building an Addition Server

Here's how to create a minimal FastMCP server that can add two numbers:

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

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

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

if __name__ == "__main__":
    mcp.run()
```

This code creates a `FastMCP` server named "Adder" and defines a single tool called `add`. The `@mcp.tool()` decorator registers this function as a tool that clients can access. `if __name__ == "__main__":` ensures this code only runs when executed directly (not when imported as a module). `mcp.run()` starts the server.

This code snippet is less than 10 lines, making it simple and easy to grasp for beginners. The code uses a decorator `@mcp.tool()` to register a function `add` as a tool. We don't need to know how `mcp.run()` works at this level. We will cover these details later in this chapter.

Internal Implementation: A Simplified View

Let's visualize what happens when a client calls the `add` tool:

```
sequenceDiagram
    participant Client
    participant Network
    participant FastMCP Server
    Client->>Network: Request to add(2,3)
    Network->>FastMCP Server: Request to add(2,3)
    activate FastMCP Server
    FastMCP Server->>add function: Execute add(2,3)
    activate add function
    add function->>FastMCP Server: Result: 5
    deactivate add function
    FastMCP Server->>Network: Response: 5
    deactivate FastMCP Server
    Network->>Client: Response: 5
```

The `FastMCP` server receives the request, routes it to the appropriate tool (`add` in this case), executes the tool, and sends back the result.

Let's look at a simplified version of the `run` method within the `FastMCP` class (located in `src/mcp/server/fastmcp/server.py`):

```
def run(self, transport: Literal["stdio", "sse"] = "stdio") -> None:
    """Run the FastMCP server."""
    if transport == "stdio":
        anyio.run(self.run_stdio_async) # Simplified: uses stdio for communication
    else: # transport == "sse"
        anyio.run(self.run_sse_async) # Simplified: uses SSE for communication
```

This highly simplified `run` method shows that based on the transport type, it either uses a stdio or SSE based communication mechanism. The actual implementation handles much more, including connection management, request processing, error handling, and more. We'll explore these details in later chapters.

Conclusion

In this chapter, we learned about the `FastMCP` server and its core concepts: resources, tools, and prompts. We built a simple server that adds two numbers. In the next chapter, Prompt, we'll delve deeper into prompts and how they can be used to structure interactions with LLMs.

In the previous chapter, FastMCP Server, we learned how to build a simple server using tools and resources. But what if we want our server to interact with a Large Language Model (LLM)? This is where *prompts* come in.

Imagine you're ordering food at a restaurant. You don't just tell the chef "make me something," you give them instructions: "I'd like a pepperoni pizza with extra cheese." A prompt is like that recipe for the LLM. It provides clear instructions on what kind of response you want. Prompts make interactions with LLMs more predictable and efficient.

Our central use case will be creating a server that uses a prompt to generate a story based on user input.

Key Concepts: Prompt Templates

A prompt is a reusable template for interacting with LLMs. It can include placeholders for variables, making it adaptable to different situations. Let's break it down:

- **Template:** A prompt is essentially a text template, similar to a fill-in-the-blank exercise. It guides the LLM by providing a structured context and instructions.
- **Placeholders:** Prompts can contain placeholders (variables) that are filled in when the prompt is used. This allows you to customize the LLM's input dynamically.
- **LLM Interaction:** The prompt is sent to the LLM, which generates a response based on the provided instructions and variables.

Let's see how to create a simple story-generating prompt.

Building a Story Generator

Here's how to create a `FastMCP` server with a prompt that generates a short story based on a user-provided character and setting:

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

mcp = FastMCP("StoryGenerator")

@mcp.prompt()
def story_prompt(character: str, setting: str) -> str:
    """Generate a short story."""
    return f"Write a short story about a {character} in a {setting}."

if __name__ == "__main__":
    mcp.run()
```

This code defines a prompt named `story_prompt` that takes `character` and `setting` as input. The `@mcp.prompt()` decorator registers this function as a prompt available to clients. `mcp.run()` starts the server. This is a very minimal example. A client would send the character and setting to the server, which would then pass this information to the prompt and return a story generated by the LLM.

Internal Implementation: A Simplified View

Let's visualize what happens when a client uses the `story_prompt`:

```
sequenceDiagram
    participant Client
    participant Network
    participant FastMCP Server
    participant PromptManager
    participant LLM
    Client->>Network: Request to use story_prompt("knight", "castle")
    Network->>FastMCP Server: Request to use story_prompt("knight", "castle")
    activate FastMCP Server
    FastMCP Server->>PromptManager: Render story_prompt("knight", "castle")
    activate PromptManager
    PromptManager->>LLM: "Write a short story about a knight in a castle."
    activate LLM
    LLM->>PromptManager: Story text
    deactivate LLM
    PromptManager->>FastMCP Server: Story text
    deactivate PromptManager
    FastMCP Server->>Network: Story text
    deactivate FastMCP Server
    Network->>Client: Story text
```

The `FastMCP` server receives the request, passes it to the `PromptManager` (which we'll learn more about in PromptManager), the `PromptManager` then sends the filled-in prompt to the LLM, the LLM generates the story, and the result is returned to the client.

The `@mcp.prompt()` decorator (simplified from `src/mcp/server/fastmcp/prompts/__init__.py`):

```python
def prompt(self, fn: Callable[..., PromptResult]):
    prompt = Prompt.from_function(fn)
    self.prompts.add_prompt(prompt)
    return fn
```

This simplified code snippet shows that the decorator adds a prompt using `Prompt.from_function`. It then adds this new prompt to the `PromptManager` that manages prompts internally.

Conclusion

In this chapter, we learned about the `Prompt` abstraction, a key component for interacting with LLMs within the `FastMCP` server. We built a simple story generator that demonstrates the power of prompts in structuring LLM interactions. In the next chapter, Resource, we'll explore another important concept: resources.

In the previous chapter, Prompt, we learned how to create prompts to interact with LLMs. But where does the data the LLM needs come from? This is where *resources* come in.

Imagine you're building a chatbot that answers questions about a specific company. The chatbot needs access to the company's information – its history, products, and mission statement. These pieces of information are stored as *resources*. Resources provide the context for the LLM, enabling it to give accurate and relevant answers.

Our central use case will be creating a server that provides a company's mission statement as a resource.

Key Concepts: Resource Types

A resource represents data accessible by the server. It's like a file on your computer – it holds information. The server provides ways to access and read this information. There are different types of resources: text files, images, or data from APIs. For this tutorial, we'll focus on text resources.

Building a Company Information Server

Let's build a `FastMCP` server that provides a company's mission statement as a text resource:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("CompanyInfo")

@mcp.resource("mission_statement")
def get_mission_statement() -> str:
    """Provides the company's mission statement."""
    return "To empower businesses through innovative technology."

if __name__ == "__main__":
    mcp.run()
```

This code defines a resource named "mission_statement". The `@mcp.resource()` decorator registers this function as a resource. The function `get_mission_statement()` simply returns the company's mission statement as a string. `mcp.run()` starts the server.

This is a very minimal example. A client can then request this resource from the server and the server will return the mission statement string.

Internal Implementation: A Simplified View

Let's visualize what happens when a client requests the "mission_statement" resource:

```
sequenceDiagram
    participant Client
    participant Network
    participant FastMCP Server
    participant ResourceManager
    Client->>Network: Request "mission_statement" resource
    Network->>FastMCP Server: Request "mission_statement" resource
    activate FastMCP Server
    FastMCP Server->>ResourceManager: Get "mission_statement"
    activate ResourceManager
    ResourceManager->>get_mission_statement: Execute function
    activate get_mission_statement
    get_mission_statement->>ResourceManager: "To empower businesses through innovative techn
    deactivate get_mission_statement
    ResourceManager->>FastMCP Server: "To empower businesses through innovative technology."
    deactivate ResourceManager
    FastMCP Server->>Network: "To empower businesses through innovative technology."
    deactivate FastMCP Server
    Network->>Client: "To empower businesses through innovative technology."
```

The `FastMCP` server receives the request, delegates it to the `ResourceManager` (which we'll cover in more detail in ResourceManager), the `ResourceManager` executes the function associated with the resource and returns the result to the server, which then sends the result back to the client.

The `@mcp.resource()` decorator (simplified from `src/mcp/server/fastmcp/resources/__init__.py`):

```
    def resource(self, uri: str, fn: Callable[..., Any] = None):
        # ... some code to create a resource ...
        self.resources.add_resource(uri, resource) # Add to ResourceManager
        return fn
```

This simplified code shows that the decorator creates a `Resource` object and adds it to the `ResourceManager`.

Conclusion

In this chapter, we learned about the `Resource` abstraction, a crucial component for providing data to LLMs within a `FastMCP` server. We built a simple server that exposes a company's mission statement as a resource. In the next chapter, Tool, we will explore how to create tools that allow LLMs to perform actions.

In the previous chapter, Resource, we learned how to provide data to our LLM using resources. But what if we want the LLM to *do* something? That's where *tools* come in. Tools allow our LLM to interact with the real world, performing actions like sending emails, querying databases, or making API calls.

Imagine a chef (our LLM) preparing a dish. They need ingredients (resources), but they also need tools like knives, ovens, and whisks to actually prepare the food. Tools are the equivalent of these instruments for our LLM.

Our central use case will be creating a server that lets the LLM send a simple email.

Key Concepts: Defining and Using Tools

A tool is simply a function that the LLM can call. This function performs some action and returns a result. Let's break it down:

- **Function:** A tool is defined as a regular Python function.
- **Arguments:** The function takes arguments that the LLM will provide.
- **Return Value:** The function returns a value to the LLM, representing the result of the action.
- **Side Effects:** Tools can have side effects, like sending an email or updating a database.

Building an Email Sending Server

Let's create a `FastMCP` server with a tool that sends a simple email using the `smtplib` library. For simplicity, we will omit error handling and assume you have an SMTP server configured.

```
import smtplib
from email.mime.text import MIMEText
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("EmailSender")

@mcp.tool()
def send_email(to: str, subject: str, body: str) -> str:
    """Sends a simple email."""
    msg = MIMEText(body)
    msg["Subject"] = subject
    msg["From"] = "your_email@example.com" #Replace with your email
    msg["To"] = to

    with smtplib.SMTP("your_smtp_server") as server: #Replace with your SMTP server
```

```python
        server.send_message(msg)
    return "Email sent successfully!"

if __name__ == "__main__":
    mcp.run()
```

This code defines a tool named `send_email`. The `@mcp.tool()` decorator registers this function as a tool available to clients. The function takes the recipient's email address (`to`), the subject (`subject`), and the body (`body`) of the email as arguments. It uses the `smtplib` library to send the email and returns a confirmation message. `mcp.run()` starts the server.

This is a very minimal example and omits crucial error handling (e.g., checking for valid email addresses, handling SMTP connection errors). For a production-ready server, you should add comprehensive error handling.

Internal Implementation: A Simplified View

Let's see what happens when the LLM calls the `send_email` tool:

```
sequenceDiagram
    participant LLM
    participant FastMCP Server
    participant ToolManager
    participant send_email function
    LLM->>FastMCP Server: Call send_email("test@example.com", "Test", "Hello!")
    activate FastMCP Server
    FastMCP Server->>ToolManager: Execute send_email
    activate ToolManager
    ToolManager->>send_email function: Execute with arguments
    activate send_email function
    send_email function->>SMTP Server: Send email
    activate SMTP Server
    SMTP Server->>send_email function: Email sent
    deactivate SMTP Server
    send_email function->>ToolManager: "Email sent successfully!"
    deactivate send_email function
    ToolManager->>FastMCP Server: "Email sent successfully!"
    deactivate ToolManager
    FastMCP Server->>LLM: "Email sent successfully!"
    deactivate FastMCP Server
```

The LLM sends a request to the `FastMCP` server to call the `send_email` tool with the specified arguments. The server routes the request to the `ToolManager`, which in turn executes the `send_email` function. The function interacts with the SMTP server to send the email, and the result is returned to the LLM.

The `@mcp.tool()` decorator (simplified from `src/mcp/server/fastmcp/tools/__init__.py`):

```python
def tool(self, fn: Callable[..., Any], name: str | None = None, description: str | None = No
    tool = self.tools.add_tool(fn, name=name, description=description) # Add to ToolManager
    return fn
```

This highly simplified code snippet shows that the decorator adds the tool to the `ToolManager`, which handles tool registration and execution.

Conclusion

In this chapter, we learned how to create and use tools to extend the capabilities of our LLM within a `FastMCP` server. We built a simple email-sending tool, demonstrating how to integrate external functionality. In the next chapter, RequestContext, we'll explore the `RequestContext` object which provides additional context to your tools and resources.

In the previous chapter, Tool, we learned how to create tools that allow our Language Model (LLM) to interact with the outside world. But sometimes, our tools need more information than just the arguments passed directly to them. That's where the `RequestContext` object comes in.

Imagine a chef (our LLM) preparing a dish. They need ingredients (resources), tools (like knives and ovens), and also a recipe card! This recipe card contains extra information, like the client's preferences (spicy or not spicy), available resources (only certain types of cheese), etc. The `RequestContext` is like that recipe card, providing extra context to our tools and resources.

Our central use case will be creating a tool that logs messages with additional context information, like the client's ID.

Key Concepts: Contextual Information

The `RequestContext` object provides contextual information about the current request. This information is available to any tool or resource that requests it. Key pieces of information include:

- **Client ID:** A unique identifier for the client making the request. This is useful for tracking usage, personalizing responses, or limiting access to certain tools or resources.
- **Request ID:** A unique identifier for the current request. Useful for debugging and logging.
- **Session:** The underlying session object handling the communication. It's useful for more advanced functionalities, such as sending progress updates.
- **Logging:** Functions for sending log messages directly back to the client. This is crucial for debugging and monitoring the server.
- **Progress Reporting:** Functions to inform the client about the progress of long-running operations.

Building a Context-Aware Logging Tool

Let's build a FastMCP server with a logging tool that uses RequestContext to log messages along with the client ID:

```
from mcp.server.fastmcp import FastMCP
from mcp.shared.context import Context

mcp = FastMCP("ContextLogger")

@mcp.tool()
async def log_message(message: str, ctx: Context) -> str:
    """Logs a message with client ID."""
    client_id = ctx.client_id
    await ctx.info(f"Client {client_id}: {message}")  # Log message with client ID
    return "Message logged successfully!"

if __name__ == "__main__":
    mcp.run()
```

This code defines a tool `log_message` that takes a message and a `Context` object as input. The `Context` object provides access to contextual information, including the client ID. The tool logs the message along with the client ID using `ctx.info()`, which sends a log message back to the client.

This is a really minimal example, focusing only on the logging function. Error handling is skipped.

Internal Implementation: A Simplified View

Let's visualize what happens when the LLM calls the `log_message` tool:

```
sequenceDiagram
    participant LLM
    participant FastMCP Server
    participant ToolManager
    participant log_message function
    participant RequestContext
    LLM->>FastMCP Server: Call log_message("Hello!")
    activate FastMCP Server
    FastMCP Server->>ToolManager: Execute log_message
    activate ToolManager
    ToolManager->>log_message function: Execute with RequestContext
    activate log_message function
    log_message function->>RequestContext: Get client_id and info()
    activate RequestContext
    RequestContext->>log_message function: client_id and sends info message to client
    deactivate RequestContext
    log_message function->>ToolManager: "Message logged successfully!"
    deactivate log_message function
    ToolManager->>FastMCP Server: "Message logged successfully!"
    deactivate ToolManager
    FastMCP Server->>LLM: "Message logged successfully!"
    deactivate FastMCP Server
```

The `FastMCP` server passes a `RequestContext` object to the `log_message` function. The function uses the `RequestContext` to access contextual information (client ID) and to send log messages back to the client.

A simplified version of the `Context` class (from `src/mcp/shared/context.py`):

```python
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
    lifespan_context: LifespanContextT
```

This simplified class shows the core attributes of `RequestContext`. The actual implementation handles more complex logic, but this highlights the essential parts.

The `info()` method (simplified from `src/mcp/server/fastmcp/server.py`):

```python
    async def info(self, message: str, **extra: Any) -> None:
        """Send an info log message."""
        await self.log("info", message, **extra)
```

This is a convenience method from the `Context` class. It just calls the `log` method with level "info".

Conclusion

In this chapter, we explored the `RequestContext` object and how it provides crucial contextual information to tools and resources within our `FastMCP` server. We built a simple logging tool that demonstrates how to access and use this context. In the next chapter, ServerSession, we will dive deeper into the `ServerSession` object and how it manages the server-side communication.

In the previous chapter, RequestContext, we learned how to add context to our tools and resources. Now, let's explore the `ServerSession` object, the core component managing server-side communication. Think of it as the central hub of our server, orchestrating all interactions with clients.

Our central use case will be building a simple server that echoes back any message received from a client.

Key Concepts: Managing Client Connections

The `ServerSession` manages the entire lifecycle of a client connection. This includes:

- **Receiving Messages:** It constantly listens for messages from the connected client.
- **Processing Requests:** It handles requests, routing them to appropriate tools and resources (Tool, Resource, Prompt).
- **Sending Responses:** It sends responses back to the client, including the results of requests and notifications.
- **Handling Notifications:** It receives notifications from clients and the client itself about changes in states.
- **Maintaining Session State:** It maintains information about the current client session, like initialization parameters (RequestContext).

Building an Echo Server

Let's create a minimal `FastMCP` server that simply echoes back any message it receives:

```
from mcp.server.fastmcp import FastMCP
from mcp.shared.context import Context

mcp = FastMCP("EchoServer")

@mcp.tool()
async def echo(message: str, ctx: Context) -> str:
    """Echoes the input message."""
    return message

if __name__ == "__main__":
    mcp.run()
```

This code defines a simple tool `echo` that returns the input message. The `ServerSession` will handle receiving the client's request, routing it to the `echo` tool, and sending the response back.

Internal Implementation: A Simplified View

Let's visualize a simplified interaction:

```
sequenceDiagram
    participant Client
    participant Network
    participant ServerSession
    participant echo tool
    Client->>Network: Send "Hello!"
    Network->>ServerSession: "Hello!"
    activate ServerSession
    ServerSession->>echo tool: Execute with "Hello!"
    activate echo tool
    echo tool->>ServerSession: "Hello!"
    deactivate echo tool
    ServerSession->>Network: "Hello!"
    deactivate ServerSession
    Network->>Client: "Hello!"
```

The `ServerSession` receives the message, passes it to the `echo` tool, and sends the result back.

A highly simplified snippet from the `ServerSession` class (from `src/mcp/server/session.py`):

```python
    async def _handle_incoming(self, req: ServerRequestResponder) -> None:
        await self._incoming_message_stream_writer.send(req)

    @property
    def incoming_messages(self) -> MemoryObjectReceiveStream[ServerRequestResponder]:
        return self._incoming_message_stream_reader
```

This shows how `ServerSession` manages incoming messages. `_handle_incoming` sends the message to an internal queue which is read by `incoming_messages`. This internal queue is used to ensure that request handling can be performed concurrently without blocking the reception of new messages.

Conclusion

In this chapter, we learned about the `ServerSession` object, the central component managing server-side communication. We built a simple echo server to demonstrate its core functionality. In the next chapter, ResourceManager, we will explore the `ResourceManager` and how it manages server resources.

In the previous chapter, ServerSession, we learned how the server manages client connections. But where does the server get the data it needs to respond to client requests? This is where the `ResourceManager` comes in.

Imagine a library. The `ServerSession` is the librarian at the front desk, interacting with patrons (clients). The `ResourceManager` is the entire library catalog system: it keeps track of all the books (resources), and allows the librarian to find and retrieve specific books using their titles or identifiers (URIs). It also handles getting new books (resources) from templates.

Our central use case will be building a server that provides information about different animals, stored as resources.

Key Concepts: Resources and Templates

The `ResourceManager` manages two key types of things:

- **Resources:** These are pieces of data, like text files, images, or data from APIs. In our example, each animal's information will be a separate resource.
- **Templates:** These are blueprints for creating resources dynamically. If we have many animals, instead of defining each one individually, we can use a template to generate the resources automatically.

Building an Animal Information Server

Let's build a `FastMCP` server that provides information about different animals. We'll use a template to create the resources dynamically:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("AnimalInfo")

@mcp.resource_template("animals/{animal_name}") #Create a template for dynamic resources
async def get_animal_info(animal_name: str) -> str:
    """Provides information about an animal."""
    if animal_name == "dog":
        return "Dogs are loyal and furry companions."
    elif animal_name == "cat":
        return "Cats are independent and graceful creatures."
    else:
        return "I don't have information about that animal."

if __name__ == "__main__":
    mcp.run()
```

This code defines a resource *template* named "animals/{animal_name}". The `{animal_name}` part is a placeholder; it will be replaced with the specific animal name when a client requests it. The `get_animal_info` function will be used to dynamically create a resource each time it's accessed. The `@mcp.resource_template()` decorator registers this function as a resource template. `mcp.run()` starts the server. A client could request `/animals/dog` or `/animals/cat`, and the server will dynamically generate a response based on the template.

Internal Implementation: A Simplified View

Let's see what happens when a client requests `/animals/dog`:

```
sequenceDiagram
    participant Client
    participant Network
    participant FastMCP Server
    participant ResourceManager
    Client->>Network: Request /animals/dog
    Network->>FastMCP Server: Request /animals/dog
    activate FastMCP Server
    FastMCP Server->>ResourceManager: Get /animals/dog
    activate ResourceManager
    ResourceManager->>get_animal_info: Execute with animal_name="dog"
    activate get_animal_info
    get_animal_info->>ResourceManager: "Dogs are loyal and furry companions."
    deactivate get_animal_info
    ResourceManager->>FastMCP Server: "Dogs are loyal and furry companions."
    deactivate ResourceManager
    FastMCP Server->>Network: "Dogs are loyal and furry companions."
    deactivate FastMCP Server
    Network->>Client: "Dogs are loyal and furry companions."
```

The `ResourceManager` receives the request. It checks if it's a registered resource or template. It finds the template `animals/{animal_name}`, fills the placeholder "{animal_name}" with "dog", and runs the associated function `get_animal_info`. The result is then returned to the client.

A simplified version of the `add_resource_template` method (from `src/mcp/server/fastmcp/resources/__in`

```
    def resource_template(self, uri_template: str, fn: Callable[..., Any] = None, **kwargs)
        self.resources.add_template(fn, uri_template, **kwargs) #Add to the ResourceManager
        return fn
```

This shows how the `@mcp.resource_template` decorator adds a new template to the `ResourceManager`.

Conclusion

In this chapter, we learned about the `ResourceManager`, a key component for managing and providing access to resources within a `FastMCP` server. We built a simple animal information server using resource templates to dynamically generate resources. In the next chapter, ToolManager, we'll delve into the `ToolManager` and how it manages server tools.

In the previous chapter, ResourceManager, we learned how to manage data resources for our server. Now, let's explore how to manage the functions, or *tools*, that our server can use to process that data. This is where the `ToolManager` comes in.

Imagine a workshop. The `ResourceManager` is like the storage room, holding all the materials (resources). The `ToolManager` is the toolbox, containing all the tools (functions) needed to work with those materials. The `ToolManager` keeps track of available tools, their descriptions, and how to use them, ensuring the Large Language Model (LLM) can easily access and utilize the right tools for specific tasks.

Our central use case will be creating a server with two tools: one to add two numbers and another to subtract them.

Key Concepts: Registering and Calling Tools

The `ToolManager` is responsible for:

- **Registering Tools:** Adding new tools to the server's toolbox. Each tool is essentially a Python function.
- **Calling Tools:** Executing a specific tool with given arguments. The `ToolManager` handles the details of running the function safely and correctly.

Building a Simple Arithmetic Server

Let's create a `FastMCP` server with two tools: `add` and `subtract`:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("ArithmeticServer")

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

@mcp.tool()
def subtract(a: int, b: int) -> int:
    """Subtracts two numbers."""
    return a - b

if __name__ == "__main__":
    mcp.run()
```

This code defines two functions, `add` and `subtract`. The `@mcp.tool()` decorator registers these functions as tools within the `ToolManager`. `mcp.run()` starts the server, making these tools available to clients.

This code is short and simple! Each tool is a function, and the decorator `@mcp.tool()` automatically registers it for use.

Internal Implementation: A Simplified View

Let's see what happens when a client requests to use the `add` tool:

```
sequenceDiagram
    participant Client
    participant Network
    participant FastMCP Server
    participant ToolManager
    participant add function
    Client->>Network: Request to add(5,3)
    Network->>FastMCP Server: Request to add(5,3)
    activate FastMCP Server
    FastMCP Server->>ToolManager: Execute add(5,3)
    activate ToolManager
    ToolManager->>add function: Execute with arguments
    activate add function
    add function->>ToolManager: Result: 8
    deactivate add function
    ToolManager->>FastMCP Server: Result: 8
    deactivate ToolManager
    FastMCP Server->>Network: Result: 8
    deactivate FastMCP Server
    Network->>Client: Result: 8
```

The `ToolManager` receives the request, finds the `add` tool, runs it with the provided arguments (5 and 3), and returns the result (8) back to the server which then forwards it to the client.

A simplified look at the `add_tool` method within the `ToolManager` (from `src/mcp/server/fastmcp/tools/tool_manager.py`):

```python
    def add_tool(self, fn: Callable[..., Any], name: str | None = None, description: str | N
        tool = Tool.from_function(fn, name=name, description=description) #Create a Tool ob
        # ... some code to add the tool to internal storage ...
        return tool
```

This simplified code shows that `add_tool` creates a `Tool` object from the given function and adds it to the internal list of tools. The `call_tool` method is responsible for actually executing the tool, which is shown in the sequence diagram.

Conclusion

In this chapter, we learned about the `ToolManager`, a crucial component for managing and executing tools within the `FastMCP` server. We built a simple arithmetic server, demonstrating how to register and use tools. In the next chapter, PromptManager, we'll delve into the `PromptManager` and learn how it manages prompts for interacting with LLMs.

In the previous chapter, ToolManager, we learned how to manage tools for our server. Now, let's explore how to manage prompts for interacting with Large Language Models (LLMs). This is the role of the `PromptManager`.

Imagine a chef (our LLM) who needs recipes (prompts) to create dishes. The `PromptManager` is like a recipe book, storing various prompt templates and providing them to the chef as needed. It ensures the LLM receives properly formatted instructions, leading to more consistent and accurate responses. Our central use case will be a server that generates different types of creative text based on user requests.

Key Concepts: Prompt Templates and Rendering

The `PromptManager` manages:

- **Prompt Templates:** These are reusable templates for interacting with LLMs. They are like fill-in-the-blank recipes, with placeholders for customizable inputs.
- **Prompt Rendering:** The process of filling in the placeholders in a prompt template with specific values, creating the final instruction for the LLM.

Building a Creative Text Generator

Let's build a `FastMCP` server with two prompts: one for generating poems and another for writing short stories:

```
from mcp.server.fastmcp import FastMCP
from mcp.server.fastmcp.prompts import Prompt

mcp = FastMCP("CreativeTextGenerator")

@mcp.prompt()
def poem_prompt(topic: str) -> str:
    """Generates a poem about a given topic."""
    return f"Write a short poem about {topic}."

@mcp.prompt()
def story_prompt(character: str, setting: str) -> str:
    """Generates a short story."""
    return f"Write a short story about a {character} in a {setting}."

if __name__ == "__main__":
    mcp.run()
```

This code defines two prompts, `poem_prompt` and `story_prompt`. The `@mcp.prompt()` decorator registers them with the `PromptManager`. The server is now ready to generate poems and stories based on user inputs. Note: We are simplifying the actual LLM interaction here – the `PromptManager` will handle sending these prompts to an LLM (which is outside the scope of this example).

Internal Implementation: A Simplified View

Let's see what happens when a client requests a poem about "nature":

```
sequenceDiagram
    participant Client
    participant Network
    participant FastMCP Server
    participant PromptManager
    participant LLM
    Client->>Network: Request poem about "nature"
    Network->>FastMCP Server: Request poem about "nature"
    activate FastMCP Server
    FastMCP Server->>PromptManager: Render poem_prompt("nature")
    activate PromptManager
    PromptManager->>LLM: "Write a short poem about nature."
    activate LLM
    LLM->>PromptManager: Poem text
    deactivate LLM
    PromptManager->>FastMCP Server: Poem text
    deactivate PromptManager
    FastMCP Server->>Network: Poem text
    deactivate FastMCP Server
    Network->>Client: Poem text
```

The `FastMCP` server receives the request and forwards it to the `PromptManager`. The `PromptManager` identifies the correct prompt, renders it with the given topic, and sends it to the LLM. The LLM generates the poem, and the `PromptManager` returns the result to the server.

A simplified version of the `render_prompt` method (from `src/mcp/server/fastmcp/prompts/manager.py`):

```
    async def render_prompt(
        self, name: str, arguments: dict[str, Any] | None = None
    ) -> list[Message]:
        prompt = self.get_prompt(name)
        if not prompt:
            raise ValueError(f"Unknown prompt: {name}")
        return await prompt.render(arguments)
```

This method retrieves the prompt by name, checks if it exists, and then calls the prompt's `render` method (which handles placeholder replacement) with any provided arguments. The result is then returned.

Conclusion

In this chapter, we explored the `PromptManager`, which efficiently manages prompts for interacting with LLMs. We built a simple creative text generator using the `PromptManager` to handle prompt rendering and demonstrated the workflow involved in using prompts. This lays the groundwork for more complex interactions with LLMs in future chapters.

This Python project, `python-sdk`, implements the *Model Context Protocol (MCP)*. It allows developers to build **servers** that expose *data* (**Resources**) and *functionality* (**Tools**) to Large Language Models (LLMs) in a standardized way. The project also includes tools for managing **Prompts**, reusable templates for LLM interactions, and handles client and server-side communication (*ClientSession*, *ServerSession*).

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

```
flowchart TD
    A0["FastMCP Server"]
    A1["Resource"]
    A2["Tool"]
    A3["Prompt"]
    A4["ClientSession"]
    A5["ServerSession"]
    A6["RequestContext"]
    A7["ResourceManager"]
    A8["ToolManager"]
    A9["PromptManager"]
    A0 -- "Manages" --> A1
    A0 -- "Manages" --> A2
    A0 -- "Manages" --> A3
    A0 -- "Creates" --> A5
    A0 -- "Provides" --> A6
    A7 -- "Manages" --> A1
    A8 -- "Manages" --> A2
    A9 -- "Manages" --> A3
    A5 -- "Interacts with" --> A0
    A6 -- "Provides to" --> A2
    A6 -- "Provides access to" --> A1
    A2 -- "Uses" --> A6
    A4 -- "Connects to" --> A0
```

Chapters

1. ClientSession
2. FastMCP Server
3. Prompt
4. Resource
5. Tool
6. RequestContext
7. ServerSession
8. ResourceManager
9. ToolManager
10. PromptManager