dsp-ant Update .env.example ✓ 2ea1495 · yesterday

.github	fix publish-pypi	2 days ago
docs	Add mkdocs (#367)	4 days ago
examples	Update .env.example	yesterday
src/mcp	fix	2 days ago
tests	Fixes incorrect casting into...	2 days ago
.git-blame-ignore-re...	Added ignore revs (for gitl...	last month
.gitignore	update gitignore	4 days ago
.pre-commit-config....	check uv lockfile	last month
CLAUDE.md	docs: update read_resourc...	2 months ago
CODE_OF_CONDUC...	Add code of conduct	4 months ago
CONTRIBUTING.md	docs: Add branch selectio...	2 months ago
LICENSE	Update LICENSE	4 months ago
README.md	Fix typo in starlette import ...	2 days ago
RELEASE.md	Release on GitHub release ...	2 weeks ago
SECURITY.md	Update SECURITY.md	4 months ago
mkdocs.yml	set site url	2 days ago
pyproject.toml	refactor: Make types.py str...	3 days ago
uv.lock	Fixes to stdio_client to sup...	2 days ago

Python 100.0%

MCP Python SDK

Python implementation of the Model Context Protocol (MCP)

pypi v1.6.0 | license MIT | python 3.10 | 3.11 | 3.12 | 3.13 | docs modelcontextprotocol.io | spec spec.modelcontextprotocol.io | discussions 0 total

Table of Contents

Overview

The Model Context Protocol allows applications to provide context for LLMs in a standardized way, separating the concerns of providing context from the actual LLM interaction. This Python SDK implements the full MCP specification, making it easy to:

- Build MCP clients that can connect to any MCP server
- Create MCP servers that expose resources, prompts and tools
- Use standard transports like stdio and SSE
- Handle all MCP protocol messages and lifecycle events

Installation

Adding MCP to your python project

We recommend using uv to manage your Python projects. In a uv managed python project, add mcp to dependencies by:

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

Alternatively, for projects using pip for dependencies:

```
pip install mcp
```

Running the standalone MCP development tools

To run the mcp command with uv:

```
uv run mcp
```

Quickstart

Let's create a simple MCP server that exposes a calculator tool and some data:

```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}!"
```

You can install this server in Claude Desktop and interact with it right away by running:

```
mcp install server.py
```

Alternatively, you can test it with the MCP Inspector:

```
mcp dev server.py
```

What is MCP?

The Model Context Protocol (MCP) lets you build servers that expose data and functionality to LLM applications in a secure, standardized way. Think of it like a web API, but specifically designed for LLM interactions. MCP servers can:

- Expose data through **Resources** (think of these sort of like GET endpoints; they are used to load information into the LLM's context)
- Provide functionality through **Tools** (sort of like POST endpoints; they are used to execute code or otherwise produce a side effect)
- Define interaction patterns through **Prompts** (reusable templates for LLM interactions)
- And more!

Core Concepts

Server

The FastMCP server is your core interface to the MCP protocol. It handles connection management, protocol compliance, and message routing:

```python
# Add lifespan support for startup/shutdown with strong typing
from contextlib import asynccontextmanager
from collections.abc import AsyncIterator
from dataclasses import dataclass

from fake_database import Database  # Replace with your actual DB type

from mcp.server.fastmcp import Context, FastMCP

# Create a named server
mcp = FastMCP("My App")

# Specify dependencies for deployment and development
mcp = FastMCP("My App", dependencies=["pandas", "numpy"])


@dataclass
class AppContext:
    db: Database


@asynccontextmanager
async def app_lifespan(server: FastMCP) -> AsyncIterator[AppContext]:
    """Manage application lifecycle with type-safe context"""
    # Initialize on startup
    db = await Database.connect()
    try:
        yield AppContext(db=db)
    finally:
        # Cleanup on shutdown
        await db.disconnect()


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


# Access type-safe lifespan context in tools
@mcp.tool()
def query_db(ctx: Context) -> str:
    """Tool that uses initialized resources"""
    db = ctx.request_context.lifespan_context["db"]
    return db.query()
```

Resources

Resources are how you expose data to LLMs. They're similar to GET endpoints in a REST API - they provide data but shouldn't perform significant computation or have side effects:

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

mcp = FastMCP("My App")


@mcp.resource("config://app")
def get_config() -> str:
    """Static configuration data"""
    return "App configuration here"


@mcp.resource("users://{user_id}/profile")
```

```python
def get_user_profile(user_id: str) -> str:
    """Dynamic user data"""
    return f"Profile data for user {user_id}"
```

Tools

Tools let LLMs take actions through your server. Unlike resources, tools are expected to perform computation and have side effects:

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

mcp = FastMCP("My App")


@mcp.tool()
def calculate_bmi(weight_kg: float, height_m: float) -> float:
    """Calculate BMI given weight in kg and height in meters"""
    return weight_kg / (height_m**2)


@mcp.tool()
async def fetch_weather(city: str) -> str:
    """Fetch current weather for a city"""
    async with httpx.AsyncClient() as client:
        response = await client.get(f"https://api.weather.com/{city}")
        return response.text
```

Prompts

Prompts are reusable templates that help LLMs interact with your server effectively:

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

mcp = FastMCP("My App")


@mcp.prompt()
def review_code(code: str) -> str:
    return f"Please review this code:\n\n{code}"


@mcp.prompt()
def debug_error(error: str) -> list[base.Message]:
    return [
        base.UserMessage("I'm seeing this error:"),
        base.UserMessage(error),
        base.AssistantMessage("I'll help debug that. What have you tried so far?"),
    ]
```

Images

FastMCP provides an `Image` class that automatically handles image data:

```python
from mcp.server.fastmcp import FastMCP, Image
from PIL import Image as PILImage

mcp = FastMCP("My App")
```

```python
@mcp.tool()
def create_thumbnail(image_path: str) -> Image:
    """Create a thumbnail from an image"""
    img = PILImage.open(image_path)
    img.thumbnail((100, 100))
    return Image(data=img.tobytes(), format="png")
```

Context

The Context object gives your tools and resources access to MCP capabilities:

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

mcp = FastMCP("My App")


@mcp.tool()
async def long_task(files: list[str], ctx: Context) -> str:
    """Process multiple files with progress tracking"""
    for i, file in enumerate(files):
        ctx.info(f"Processing {file}")
        await ctx.report_progress(i, len(files))
        data, mime_type = await ctx.read_resource(f"file://{file}")
    return "Processing complete"
```

Running Your Server

Development Mode

The fastest way to test and debug your server is with the MCP Inspector:

```bash
mcp dev server.py

# Add dependencies
mcp dev server.py --with pandas --with numpy

# Mount local code
mcp dev server.py --with-editable .
```

Claude Desktop Integration

Once your server is ready, install it in Claude Desktop:

```bash
mcp install server.py

# Custom name
mcp install server.py --name "My Analytics Server"

# Environment variables
mcp install server.py -v API_KEY=abc123 -v DB_URL=postgres://...
mcp install server.py -f .env
```

Direct Execution

For advanced scenarios like custom deployments:

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

```python
mcp = FastMCP("My App")

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

Run it with:

```
python server.py
# or
mcp run server.py
```

Mounting to an Existing ASGI Server

You can mount the SSE server to an existing ASGI server using the `sse_app` method. This allows you to integrate the SSE server with other ASGI applications.

```python
from starlette.applications import Starlette
from starlette.routing import Mount, Host
from mcp.server.fastmcp import FastMCP


mcp = FastMCP("My App")

# Mount the SSE server to the existing ASGI server
app = Starlette(
    routes=[
        Mount('/', app=mcp.sse_app()),
    ]
)

# or dynamically mount as host
app.router.routes.append(Host('mcp.acme.corp', app=mcp.sse_app()))
```

For more information on mounting applications in Starlette, see the Starlette documentation.

Examples

Echo Server

A simple server demonstrating resources, tools, and prompts:

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

mcp = FastMCP("Echo")


@mcp.resource("echo://{message}")
def echo_resource(message: str) -> str:
    """Echo a message as a resource"""
    return f"Resource echo: {message}"


@mcp.tool()
def echo_tool(message: str) -> str:
    """Echo a message as a tool"""
    return f"Tool echo: {message}"


@mcp.prompt()
def echo_prompt(message: str) -> str:
```

```
    """Create an echo prompt"""
    return f"Please process this message: {message}"
```

SQLite Explorer

A more complex example showing database integration:

```python
import sqlite3

from mcp.server.fastmcp import FastMCP

mcp = FastMCP("SQLite Explorer")


@mcp.resource("schema://main")
def get_schema() -> str:
    """Provide the database schema as a resource"""
    conn = sqlite3.connect("database.db")
    schema = conn.execute("SELECT sql FROM sqlite_master WHERE type='table'").fetchall()
    return "\n".join(sql[0] for sql in schema if sql[0])


@mcp.tool()
def query_data(sql: str) -> str:
    """Execute SQL queries safely"""
    conn = sqlite3.connect("database.db")
    try:
        result = conn.execute(sql).fetchall()
        return "\n".join(str(row) for row in result)
    except Exception as e:
        return f"Error: {str(e)}"
```

Advanced Usage

Low-Level Server

For more control, you can use the low-level server implementation directly. This gives you full access to the protocol and allows you to customize every aspect of your server, including lifecycle management through the lifespan API:

```python
from contextlib import asynccontextmanager
from collections.abc import AsyncIterator

from fake_database import Database  # Replace with your actual DB type

from mcp.server import Server


@asynccontextmanager
async def server_lifespan(server: Server) -> AsyncIterator[dict]:
    """Manage server startup and shutdown lifecycle."""
    # Initialize resources on startup
    db = await Database.connect()
    try:
        yield {"db": db}
    finally:
        # Clean up on shutdown
        await db.disconnect()


# Pass lifespan to server
server = Server("example-server", lifespan=server_lifespan)
```

```python
# Access lifespan context in handlers
@server.call_tool()
async def query_db(name: str, arguments: dict) -> list:
    ctx = server.request_context
    db = ctx.lifespan_context["db"]
    return await db.query(arguments["query"])
```

The lifespan API provides:

- A way to initialize resources when the server starts and clean them up when it stops
- Access to initialized resources through the request context in handlers
- Type-safe context passing between lifespan and request handlers

```python
import mcp.server.stdio
import mcp.types as types
from mcp.server.lowlevel import NotificationOptions, Server
from mcp.server.models import InitializationOptions

# Create a server instance
server = Server("example-server")


@server.list_prompts()
async def handle_list_prompts() -> list[types.Prompt]:
    return [
        types.Prompt(
            name="example-prompt",
            description="An example prompt template",
            arguments=[
                types.PromptArgument(
                    name="arg1", description="Example argument", required=True
                )
            ],
        )
    ]


@server.get_prompt()
async def handle_get_prompt(
    name: str, arguments: dict[str, str] | None
) -> types.GetPromptResult:
    if name != "example-prompt":
        raise ValueError(f"Unknown prompt: {name}")

    return types.GetPromptResult(
        description="Example prompt",
        messages=[
            types.PromptMessage(
                role="user",
                content=types.TextContent(type="text", text="Example prompt text"),
            )
        ],
    )


async def run():
    async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
        await server.run(
            read_stream,
            write_stream,
            InitializationOptions(
                server_name="example",
                server_version="0.1.0",
                capabilities=server.get_capabilities(
```

```
                    notification_options=NotificationOptions(),
                    experimental_capabilities={},
                ),
            ),
        )


if __name__ == "__main__":
    import asyncio

    asyncio.run(run())
```

Writing MCP Clients

The SDK provides a high-level client interface for connecting to MCP servers:

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

# Create server parameters for stdio connection
server_params = StdioServerParameters(
    command="python",  # Executable
    args=["example_server.py"],  # Optional command line arguments
    env=None,  # Optional environment variables
)


# Optional: create a sampling callback
async def handle_sampling_message(
    message: types.CreateMessageRequestParams,
) -> types.CreateMessageResult:
    return types.CreateMessageResult(
        role="assistant",
        content=types.TextContent(
            type="text",
            text="Hello, world! from model",
        ),
        model="gpt-3.5-turbo",
        stopReason="endTurn",
    )


async def run():
    async with stdio_client(server_params) as (read, write):
        async with ClientSession(
            read, write, sampling_callback=handle_sampling_message
        ) as session:
            # Initialize the connection
            await session.initialize()

            # List available prompts
            prompts = await session.list_prompts()

            # Get a prompt
            prompt = await session.get_prompt(
                "example-prompt", arguments={"arg1": "value"}
            )

            # List available resources
            resources = await session.list_resources()

            # List available tools
            tools = await session.list_tools()

            # Read a resource
```

```
            content, mime_type = await session.read_resource("file://some/path")

            # Call a tool
            result = await session.call_tool("tool-name", arguments={"arg1": "value"})


if __name__ == "__main__":
    import asyncio

    asyncio.run(run())
```

MCP Primitives

The MCP protocol defines three core primitives that servers can implement:

Primitive	Control	Description	Example Use
Prompts	User-controlled	Interactive templates invoked by user choice	Slash commands, menu options
Resources	Application-controlled	Contextual data managed by the client application	File contents, API responses
Tools	Model-controlled	Functions exposed to the LLM to take actions	API calls, data updates

Server Capabilities

MCP servers declare capabilities during initialization:

Capability	Feature Flag	Description
`prompts`	`listChanged`	Prompt template management
`resources`	`subscribe` `listChanged`	Resource exposure and updates
`tools`	`listChanged`	Tool discovery and execution
`logging`	-	Server logging configuration
`completion`	-	Argument completion suggestions

Documentation

- Model Context Protocol documentation
- Model Context Protocol specification
- Officially supported servers

Contributing

We are passionate about supporting contributors of all levels of experience and would love to see you get involved in the project. See the contributing guide to get started.

License