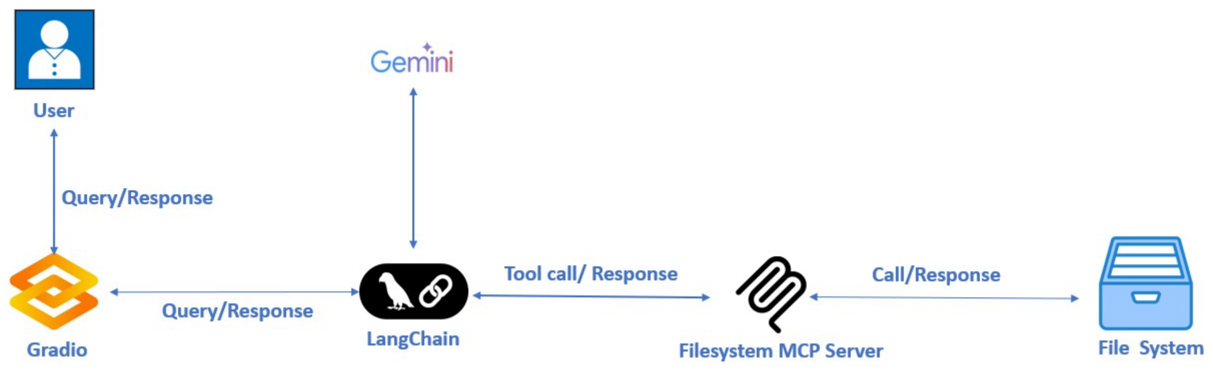
Gradio MCP Integration

Interactive File Management Interface



Overview

The Gradio MCP File Assistant implements the Model Context Protocol (MCP) specification to create a seamless web-based interface between AI agents and local file systems. This interactive application enables users to perform comprehensive file operations through natural language queries using a modern web interface powered by Gradio and Google's Gemini AI model.

Key Features

- **Interactive Web Interface** - Modern chat-based UI for natural language file queries
- **File System Integration** - Direct access to local directories and files through MCP
- **AI-Powered Responses** - Google Gemini 2.0 Flash model for intelligent file operations
- **Real-time Processing** - Asynchronous handling for responsive user experience
- **Chat History Management** - Persistent conversation context throughout sessions

Prerequisites

Before you begin, ensure you have the following installed and configured:

Required Software

- **Python 3.10 or higher**
- **Node.js and npm** - For MCP filesystem server

- **uv package manager** - Faster alternative to pip for package management
- **Google AI API Key** - For the Gemini model

Python Dependencies

- `gradio` - Web interface framework
- `mcp` - Model Context Protocol implementation
- `langchain-mcp-adapters` - MCP integration for LangChain
- `langgraph` - Agent framework
- `langchain-google-genai` - Google AI integration
- `langchain-core` - Core LangChain components

Environment Setup

1. **Google AI API Key** - Obtain from Google AI Studio
2. **MCP Filesystem Server** - Automatically installed via npx

---

Project Setup

Installing uv Package Manager

If you don't have uv installed yet:

```
pip install uv
```

Creating a Project Environment

Create a new project environment using uv:

```
uv init gradio_mcp_file_assistant_project
cd gradio_mcp_file_assistant_proj
```

Installing Required Packages

Install all required packages using uv:

```
uv add gradio==5.9.1 "mcp[cli]==1.9.2" langchain-mcp-adapters==0.1.4 langg
raph==0.5.3 langchain-google-genai==2.1.5 langchain-core==0.3.62 python-
dotenv==1.1.0
```

Environment Configuration

Create a `.env` file in your project directory with the following configuration:

```
# Required: Your Google AI API key for Gemini model
GOOGLE_API_KEY=your_google_api_key_her
```

Note: Replace the paths and API key with your actual values.

Adding Python Scripts

After setting up the project environment, you need to add the required Python script to your project directory:

- **Place the main script** - Save the `gradio_mcp.py` file in your project directory

Project Structure

Your project structure should look like:

```
gradio_mcp_file_assistant_project/
├── gradio_mcp.py
├── .env
├── pyproject.toml
└── README.md
```

Running the System

Starting the File Assistant

Navigate to your project directory and run the application:

```
cd gradio_mcp_file_assistant_project
python gradio_mcp.py
```

Alternatively, you can use uv to run the script:

The system will:

1. Initialize the background event loop
2. Set up the MCP client connection
3. Configure the AI agent
4. Launch the Gradio web interface on port 7860

Accessing the Interface

Once started, access the web interface at:

```
http://localhost:7860
```

Configuration Details:

- **Environment Loading:** Loads API keys and settings from `.env` file
- **AI Model Setup:** Initializes Google Gemini 2.0 Flash model for intelligent responses
- **MCP Server Params:** Configures the filesystem MCP server with:
  - `npx` command to run Node.js packages
  - `y` flag for automatic yes to prompts
  - `@modelcontextprotocol/server-filesystem` package
  - Target directory path for file operations

Global Variables Section

```
# Global variables
agent ☐ None
loop ☐ None
loop_ready ☐ False
chat_history  ☐  InMemoryChatMessageHistory()
```

Global State Management:

- `agent` - Stores the initialized LangGraph agent
- `loop` - Reference to the background asyncio event loop
- `loop_ready` - Flag to track event loop initialization status
- `chat_history` - In-memory storage for conversation context

Background Event Loop Function

```
def start_background_loop():
    """Start background event loop"""
    global loop, loop_ready
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop_ready ☐ True
    loop.run_forever()
```

Purpose: Creates a dedicated event loop for MCP operations

- **New Event Loop:** Creates isolated asyncio loop for MCP communication
- **Loop Assignment:** Sets the loop as the current event loop for the thread
- **Ready Flag:** Signals that the loop is ready for use
- **Infinite Run:** Keeps the loop running to handle async operations

Agent Setup Function

```
async def setup_agent():
    """Initialize the agent"""
    global agent

    async with stdio_client(server_params) as (read, write):
        async with ClientSession(read, write) as session:
            await session.initialize()
            tools = await load_mcp_tools(session)
            agent = create_react_agent(model, tools)
            print("Agent ready!")

# Keep the connection alive
            await asyncio.Event().wait()
```

Agent Initialization Process:

- **Stdio Client:** Establishes communication with MCP filesystem server
- **Client Session:** Creates persistent MCP session for tool operations
- **Session Initialize:** Handshakes with the MCP server
- **Tool Loading:** Loads available filesystem tools from MCP server
- **Agent Creation:** Creates ReAct agent with Gemini model and MCP tools
- **Connection Alive:** Maintains persistent connection using asyncio.Event

Chat Handler Function

```
def chat(message, history):
    """Handle chat messages"""
    if not agent:
        return "Agent not ready yet!"

    try:
# Add the user message to history
        chat_history.add_message(HumanMessage(content=message))

        future = asyncio.run_coroutine_threadsafe(
            agent.ainvoke({"messages": chat_history.messages}),
            loop
        )
        response = future.result(timeout=30)

# Get the AI's response and add it to history
        ai_response = response["messages"□□-1□.content
        chat_history.add_message(response["messages"□□-1□)

        return ai_response

    except Exception as e:
        return f"Error: {str(e)}"
```

Message Processing Flow:

- **Agent Check:** Validates that the agent is initialized
- **Message Storage:** Adds user message to conversation history
- **Async Execution:** Runs agent processing in background event loop
- **Future Result:** Waits for response with 30-second timeout
- **Response Extraction:** Gets AI response from the last message
- **History Update:** Adds AI response to conversation history
- **Error Handling:** Catches and displays any processing errors

Gradio Interface Setup

```
# Create chat interface
demo = gr.ChatInterface(
    fn=chat,
    title="File Assistant",
    description="Ask questions about your files",
    type="messages"
)
```

Interface Configuration:

- **ChatInterface:** Modern chat UI component from Gradio
- **Function Binding:** Links the `chat` function to handle messages
- **UI Customization:** Sets title and description for the interface
- **Message Type:** Configures for message-based conversations

Main Execution Block

```
if __name__== "__main__":
    print("Starting background loop...")
    loop_thread = threading.Thread(target=start_background_loop, daemon=Tr
ue)
    loop_thread.start()
    while not loop_ready:
        time.sleep(0.1)
    asyncio.run_coroutine_threadsafe(setup_agent(), loop)
    time.sleep(3)
    demo.launch(server_port=7860)
```

Startup Sequence:

1. **Thread Creation:** Starts background event loop in separate daemon thread
2. **Loop Wait:** Waits for event loop to be ready using polling
3. **Agent Setup:** Initializes agent in the background loop
4. **Startup Delay:** Gives agent time to fully initialize
5. **Interface Launch:** Starts Gradio web interface on port 7860