natoma

A Developer’s Guide to Building Your First MCP Server

Introduction

The AI development landscape is evolving rapidly, and one of the most exciting developments is Anthropic’s Model Context Protocol (MCP). If you've ever been frustrated by the constant context switching between AI models and the loss of conversation history when moving between different AI clients, MCP might be the solution you've been looking for.

What is MCP and Why Should You Care?

Think of MCP as the "USB-C port" for AI systems. Just as USB-C provides a standardized way to connect various devices to your laptop, MCP provides a standardized way for AI models to connect with external tools, data sources, and services.

MCP is a standardized communication layer (based on JSON-RPC) that allows AI clients (like Cursor) to discover and use external capabilities provided by MCP servers. Instead of building separate integrations for each AI client, you build one MCP server that works everywhere.

The Problem MCP Solves

Before MCP, connecting AI assistants to external tools created fragile integrations. Update the AI model or change an API, and your code breaks. Client developers can't build everything. They don't want to spend all their development hours tweaking web search for every new model, and they're definitely not out here trying to roll their own Jira integration.

MCP changes this by letting service providers maintain their own AI integrations, resulting in higher-quality interactions and less duplicated effort across the ecosystem.

Understanding MCP Architecture

MCP follows a client-server architecture with three main components:

- **MCP Hosts (Clients)**: Applications like Claude Desktop, Cursor, or Windsurf that need access to external capabilities.
- **MCP Servers**: Lightweight services that expose specific functionalities - tools, resources, and prompts - through the MCP protocol.
- **Transport Layer**: The communication mechanism between clients and servers, typically using standard input/output (stdio) or Server-Sent Events (SSE).

Core MCP Capabilities

MCP servers can provide three types of capabilities:

- **Resources:** File-like data that can be read by clients (API responses, file contents, database queries)
- **Tools:** Functions that can be executed by the AI model (with user approval)
- **Prompts:** Pre-written templates that help users accomplish specific tasks

Building Your First MCP Server

Let's walk through creating a practical MCP server step by step. We'll start with a simple calculator server to demonstrate core concepts, then show how these principles apply to more complex scenarios like weather services.

Setting Up Your Development Environment

The fastest way to get started is with Python and FastMCP, which simplifies MCP server development significantly:

```
# Create and activate virtual environment
python -m venv mcp-env
source mcp-env/bin/activate  # On Windows: mcp-env\Scripts\activate

# Install MCP SDK with FastMCP
pip install mcp fastmcp
```

Your First Server: A Calculator

Let's start with something simple - a calculator server that demonstrates all MCP concepts:

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

# Initialize the server with a descriptive name
mcp = FastMCP("Calculator Server")

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

@mcp.tool()
def multiply(a: int, b: int) -> int:
    """Multiply two numbers"""
    return a * b
```

```
# A resource that provides calculation history
@mcp.resource("calculator://history")
def calculation_history() -> str:
    """Get recent calculation history"""
    # In a real implementation, you'd track this
    return "Recent calculations: 5+3=8, 4*2=8, factorial(5)=120"

# A prompt to guide AI behavior
@mcp.prompt()
def calculator_assistant() -> str:
    """Instructions for using the calculator effectively"""
    return """
    You are a helpful math assistant. Use the calculator tools to:
    - Perform accurate calculations for users
    - Show your work step by step
    - Explain mathematical concepts when helpful
    - Check your answers using the available tools
    """

# Run the server
if __name__ == "__main__":
    mcp.run(transport="stdio")
```

This simple example showcases the three core MCP capabilities: tools for actions, resources for data, and prompts for guidance.

Understanding FastMCP Magic

FastMCP uses Python's type hints and docstrings to automatically generate proper MCP schemas. When you write:

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

FastMCP automatically creates:

- Tool registration with the MCP server
- JSON schema for input validation
- Proper error handling and response formatting
- Documentation from your docstring

This dramatically reduces boilerplate compared to writing raw MCP protocol handlers.

Testing with MCP Inspector: Your Development Best Friend

The MCP Inspector is an interactive developer tool for testing and debugging MCP servers. It's your most important development tool, acting as a test client that lets you verify your server works correctly before connecting it to AI clients.

Setting Up the Inspector

The Inspector runs directly through npx without requiring installation:

```
# For locally developed servers (most common)
npx @modelcontextprotocol/inspector node path/to/your/server.js

# For Python servers
npx @modelcontextprotocol/inspector python path/to/your/server.py

# For servers with specific arguments
npx @modelcontextprotocol/inspector python server.py --config config.json
```

Inspector Interface Features

The Inspector provides several features for interacting with your MCP server:

Server Connection Pane

- Allows selecting the transport for connecting to the server
- For local servers, supports customizing the command-line arguments and environment
- Shows connection status and capability negotiation

Resources Tab

- Lists all available resources
- Shows resource metadata (MIME types, descriptions)
- Allows resource content inspection
- Supports subscription testing

Tools Tab

- Lists available tools
- Shows tool schemas and descriptions
- Enables tool testing with custom inputs
- Displays tool execution results

Prompts Tab

- Displays available prompt templates
- Shows prompt arguments and descriptions
- Enables prompt testing with custom arguments
- Previews generated messages

Notifications Pane

- Presents all logs recorded from the server
- Shows notifications received from the server

Recommended Development Workflow

The Inspector documentation recommends a specific iterative development workflow:

1. **Start Development**
   - Launch Inspector with your server
   - Verify basic connectivity
   - Check capability negotiation

2. **Iterative Testing**
   - Make server changes
   - Rebuild the server
   - Reconnect the Inspector
   - Test affected features
   - Monitor messages

3. **Test Edge Cases**
   - Invalid inputs
   - Missing prompt arguments
   - Concurrent operations
   - Verify error handling and error responses

Pro tip: Keep the MCP Inspector open while developing. After each code change, rebuild your server and reconnect the Inspector to test your changes immediately. This rapid feedback loop catches issues early and shows you exactly how your server communicates via the MCP protocol.

Building a More Complex Example: Weather Server

Now let's apply these concepts to a practical weather server that demonstrates real-world patterns:

```
from mcp.server.fastmcp import FastMCP
import httpx
import json
from typing import Dict, Any

mcp = FastMCP("Weather Service")

async def fetch_weather_data(location: str) → Dict[str, Any]:
    """Helper function to fetch weather from National Weather Service"""
    # Simplified example - real implementation would handle errors
    async with httpx.AsyncClient() as client:
        # Get coordinates for location (simplified)
        response = await client.get(f"https://api.weather.gov/points/{location}")
        if response.status_code == 200:
            data = response.json()
            forecast_url = data["properties"]["forecast"]

            # Get forecast
            forecast_response = await client.get(forecast_url)
            return forecast_response.json()

    return {"error": "Could not fetch weather data"}

@mcp.tool()
async def get_forecast(location: str) → str:
    """Get weather forecast for a specific location (US only)"""
    try:
        weather_data = await fetch_weather_data(location)
        if "error" in weather_data:
            return f"Error: {weather_data['error']}"

        # Format the forecast nicely
        periods = weather_data.get("properties", {}).get("periods", [])[:3]
        forecast = []
```

```
            for period in periods:
                forecast.append(f"{period['name']}: {period['detailedForecast']}")

            return "\\n\\n".join(forecast)
        except Exception as e:
            return f"Error getting forecast: {str(e)}"

    @mcp.resource("weather://current/{location}")
    async def current_conditions(location: str) -> str:
        """Current weather conditions as a resource"""
        data = await fetch_weather_data(location)
        return json.dumps(data, indent=2)

    @mcp.prompt()
    def weather_assistant_guidance() -> str:
        """Comprehensive weather assistant instructions"""
        return """
        You are a knowledgeable weather assistant. When users ask about weather:

        1. Use get_forecast tool for future weather predictions
        2. Access current_conditions resource for detailed current data
        3. Always specify that forecasts are for US locations only
        4. Provide helpful context about weather conditions
        5. Suggest appropriate activities based on the forecast
        6. Alert users to any severe weather conditions

        Be conversational and helpful, not just informational.
        """

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

This example demonstrates several important patterns:

- **Async operations**: Using async/await for external API calls
- **Error handling**: Graceful degradation when services are unavailable
- **Helper functions**: Separating business logic from MCP tool definitions
- **Resource parameterization**: Dynamic resources that accept parameters
- **Comprehensive prompts**: Detailed guidance for AI behavior

Connecting to AI Clients

Once your server works perfectly in the inspector, connecting it to real AI clients is straightforward. The key is providing the correct paths and ensuring your environment is properly configured.

Claude Desktop Integration

Claude Desktop reads server configurations from a JSON file. The location varies by operating system:

- macOS: ~/Library/Application Support/Claude/claude_desktop_config.json
- Windows: %APPDATA%\Claude\claude_desktop_config.json

Create or edit this file:

```
{
  "mcpServers": {
    "calculator": {
      "command": "python",
      "args": ["/absolute/path/to/your/calculator.py"],
      "env": {
        "PYTHONPATH": "/path/to/your/project"
      }
    },
    "weather": {
      "command": "/absolute/path/to/venv/bin/python",
      "args": ["/absolute/path/to/weather_server.py"]
    }
  }
}
```

Important notes:

- Always use absolute paths, not relative ones
- If you're using a virtual environment, point to the Python executable inside it
- Restart Claude Desktop after making changes
- Check the connection status in Claude's interface

Cursor IDE Integration

Cursor has built-in MCP support through its settings interface:

1. Go to **Settings → MCP → Add New Server**
2. Configure the server:
   - **Name**: A descriptive name (e.g., "Calculator")
   - **Type**: Select "command"
   - **Command**: Full path to your Python executable
   - **Args**: Path to your server script

For virtual environments, your command might look like:

```
/Users/yourname/projects/mcp-server/venv/bin/python
```

With args:

```
/Users/yourname/projects/mcp-server/calculator.py
```

Cursor-Specific Tips

- The green circle indicator shows your server is connected and healthy
- Orange indicates connection issues - check your paths
- Use Cursor's composer with the @ symbol to reference your MCP tools
- Create custom rules to guide how Cursor uses your MCP server

Troubleshooting Connections

Server Not Appearing:

- Verify absolute paths are correct
- Check that your Python environment has all dependencies
- Look for error messages in the client's developer console
- Test the server independently with MCP Inspector first

Permission Issues:

- Ensure the server script has execute permissions
- Check that the Python interpreter is accessible
- Verify environment variables are set correctly

Runtime Errors:

- Add logging to your server to debug issues
- Use try-catch blocks around external API calls
- Test with simple tools first, then add complexity

Best Practices and Development Patterns

Development Workflow

Successful MCP server development follows a predictable pattern:

1. **Start Simple**: Begin with basic tools that don't require external dependencies
2. **Test Early**: Use MCP Inspector to verify each tool as you build it
3. **Add Complexity Gradually**: Introduce external APIs, state management, and error handling
4. **Integration Test**: Connect to your preferred AI client and test real workflows
5. **Iterate Based on Usage**: Refine based on how the AI actually uses your tools

Error-Handling Strategies

Robust error handling is crucial for MCP servers since they run in the background:

```
@mcp.tool()
async def reliable_api_call(query: str) -> str:
    """Example of comprehensive error handling"""
    try:
        async with httpx.AsyncClient(timeout=10.0) as client:
            response = await client.get(f"https://api.example.com/search?q={query}")
            response.raise_for_status()

            data = response.json()
            return f"Results: {data.get('results', 'No results found')}"

    except httpx.TimeoutException:
        return "Error: Request timed out. Please try again."
    except httpx.HTTPStatusError as e:
        return f"Error: API returned status {e.response.status_code}"
    except json.JSONDecodeError:
        return "Error: Invalid response from API"
    except Exception as e:
        return f"Unexpected error: {str(e)}"
```

State Management Approaches

MCP servers often need to maintain state between tool calls. Here are common patterns.

Simple File-Based State

```
import json
from pathlib import Path

STATE_FILE = Path("server_state.json")

def load_state() → dict:
    """Load state from disk"""
    if STATE_FILE.exists():
        return json.loads(STATE_FILE.read_text())
    return {}

def save_state(state: dict):
    """Save state to disk"""
    STATE_FILE.write_text(json.dumps(state, indent=2))

@mcp.tool()
def remember_preference(key: str, value: str) → str:
    """Store a user preference"""
    state = load_state()
    state[key] = value
    save_state(state)
    return f"Remembered: {key} = {value}"
```

Database Integration

For production servers, consider SQLite or other databases:

```
import sqlite3
from contextlib import contextmanager

@contextmanager
def get_db_connection():
    """Context manager for database connections"""
    conn = sqlite3.connect("server.db")
    try:
        yield conn
    finally:
        conn.close()

@mcp.tool()
def store_user_data(user_id: str, data: str) → str:
    """Store data with proper database handling"""
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(
                "INSERT OR REPLACE INTO user_data (user_id, data) VALUES (?, ?)",
                (user_id, data)
            )
            conn.commit()
            return "Data stored successfully"
    except Exception as e:
        return f"Database error: {str(e)}"
```

Security Considerations

MCP servers run with user permissions and can access external services:

Environment Variable Management

```
import os
from typing import Optional

def get_api_key(service: str) → Optional[str]:
    """Safely retrieve API keys from environment"""
    key = os.getenv(f"{service.upper()}_API_KEY")
    if not key:
        raise ValueError(f"Missing {service} API key in environment variables")
    return key

@mcp.tool()
async def secure_api_call(query: str) → str:
    """Tool that uses API key securely"""
    try:
        api_key = get_api_key("openweather")
        # Use api_key in request...
    except ValueError as e:
        return f"Configuration error: {str(e)}"
```

Input Validation

```
from typing import Literal

@mcp.tool()
def validated_calculation(
    operation: Literal["add", "subtract", "multiply", "divide"],
    a: float,
    b: float
) → str:
    """Calculator with strict input validation"""
    if operation == "divide" and b == 0:
        return "Error: Division by zero"
```

```
    operations = {
        "add": a + b,
        "subtract": a - b,
        "multiply": a * b,
        "divide": a / b
    }

    result = operations[operation]
    return f"{a} {operation} {b} = {result}"
```

Performance Optimization

Caching Strategies

```
from functools import lru_cache
import time

@lru_cache(maxsize=100)
def expensive_computation(input_data: str) -> str:
    """Cache expensive operations"""
    # Simulate expensive operation
    time.sleep(2)
    return f"Processed: {input_data}"

@mcp.tool()
def cached_tool(data: str) -> str:
    """Tool that benefits from caching"""
    return expensive_computation(data)
```

Async Best Practices

```
import asyncio
from typing import List

@mcp.tool()
async def batch_api_calls(queries: List[str]) -> str:
    """Efficiently handle multiple API calls"""
    async def single_call(query: str) -> str:
        async with httpx.AsyncClient() as client:
            response = await client.get(f"https://api.example.com/search?q={query}")
            return response.json()
```

15

```
    # Process all queries concurrently
    results = await asyncio.gather(*[single_call(q) for q in queries])
    return f"Processed {len(results)} queries successfully"
```

Advanced Use Cases and Real-World Applications

Multi-Service Integration Servers

One of MCP's most powerful features is creating servers that coordinate between multiple services. Here's a project management server example:

```
@mcp.tool()
async def create_project_workflow(
    project_name: str,
    team_members: List[str]
) -> str:
    """Create a complete project setup across multiple services"""
    results = []

    # Create Linear project
    linear_project = await create_linear_project(project_name)
    results.append(f"Linear project created: {linear_project['id']}")

    # Create Slack channel
    slack_channel = await create_slack_channel(project_name.lower().replace(' ', '-'))
    results.append(f"Slack channel created: #{slack_channel['name']}")

    # Add team members to both
    for member in team_members:
        await add_member_to_linear(linear_project['id'], member)
        await invite_to_slack_channel(slack_channel['id'], member)

    # Create initial Notion page
    notion_page = await create_notion_project_page(project_name, linear_project['id'])
    results.append(f"Notion documentation created: {notion_page['url']}")

    return "\\n".join(results)
```

Dynamic Tool Generation

For advanced use cases, you can generate tools dynamically based on configuration:

```
def create_api_tools(api_config: dict):
    """Generate MCP tools from API configuration"""
    for endpoint in api_config['endpoints']:
        tool_name = f"call_{endpoint['name']}"

        @mcp.tool(name=tool_name)
        async def dynamic_api_call(
            **kwargs
        ) -> str:
            """Dynamically generated API tool"""
            async with httpx.AsyncClient() as client:
                response = await client.request(
                    method=endpoint['method'],
                    url=f"{api_config['base_url']}{endpoint['path']}",
                    **kwargs
                )
                return response.json()

        # Register the tool
        globals()[tool_name] = dynamic_api_call

# Load configuration and generate tools
api_config = load_api_configuration()
create_api_tools(api_config)
```

Stateful Conversation Management

Create servers that maintain context across different AI clients:

```
from datetime import datetime
import uuid

class ConversationMemory:
    def __init__(self):
        self.conversations = {}

    def get_or_create_session(self, user_id: str) -> str:
        """Get existing session or create new one"""
        if user_id not in self.conversations:
            session_id = str(uuid.uuid4())
            self.conversations[user_id] = {
                'session_id': session_id,
                'created_at': datetime.now(),
                'messages': [],
                'context': {}
            }
        return self.conversations[user_id]['session_id']

memory = ConversationMemory()

@mcp.tool()
def remember_context(user_id: str, key: str, value: str) -> str:
    """Store context for the user's session"""
    session_id = memory.get_or_create_session(user_id)
    memory.conversations[user_id]['context'][key] = value
    return f"Remembered {key} for session {session_id}"

@mcp.resource("memory://context/{user_id}")
def get_user_context(user_id: str) -> str:
    """Retrieve user's conversation context"""
    if user_id in memory.conversations:
        context = memory.conversations[user_id]['context']
        return json.dumps(context, indent=2)
    return "{}"
```

Enterprise Integration Patterns

Service Discovery

```
import consul

@mcp.tool()
async def discover_services(service_type: str) -> List[str]:
    """Discover available services in the infrastructure"""
    c = consul.Consul()
    services = c.health.service(service_type, passing=True)[1]
    return [f"{s['Service']['Address']}:{s['Service']['Port']}"
            for s in services]
```

Workflow Orchestration

```
@mcp.tool()
async def execute_deployment_workflow(
    app_name: str,
    environment: str
) -> str:
    """Orchestrate complex deployment workflow"""
    workflow_steps = [
        ("build", build_application),
        ("test", run_tests),
        ("deploy", deploy_to_environment),
        ("verify", verify_deployment),
        ("notify", send_notifications)
    ]

    results = []
    for step_name, step_func in workflow_steps:
        try:
            result = await step_func(app_name, environment)
            results.append(f"✅ {step_name}: {result}")
        except Exception as e:
            results.append(f"❌ {step_name}: {str(e)}")
            # Handle rollback logic
            await rollback_deployment(app_name, environment)
            break

    return "\\n".join(results)
```

Getting Started: Your First Production Server

Step-by-Step Development Process

1. **Identify the Need**: Start with a tool you use frequently that could benefit from AI integration
2. **Design the Interface**: Plan your tools, resources, and prompts before coding
3. **Build Incrementally**: Start with one tool, test it, then add more
4. **Add Error Handling**: Make your server robust with proper error handling
5. **Optimize Performance**: Add caching and async operations where beneficial
6. **Deploy and Iterate**: Connect to your AI client and refine based on usage

Common Mistakes to Avoid

- **Overcomplicating**: Start simple and add complexity gradually
- **Poor Error Handling**: Always handle external API failures gracefully
- **Ignoring Security**: Never hardcode API keys or sensitive data
- **Foregoing Testing**: Always test with MCP Inspector before connecting to AI clients
- **Unclear Documentation**: Write clear tool descriptions and prompts

Quick Start Template

Here's a template to get you started quickly:

```
from mcp.server.fastmcp import FastMCP
import os
import httpx
import json

# Initialize server
mcp = FastMCP("My Custom Server")

# Load configuration
API_KEY = os.getenv("MY_API_KEY")
if not API_KEY:
    raise ValueError("MY_API_KEY environment variable required")

@mcp.tool()
async def my_first_tool(input_data: str) -> str:
    """Description of what this tool does"""
    try:
        # Your logic here
        return f"Processed: {input_data}"
    except Exception as e:
        return f"Error: {str(e)}"

@mcp.resource("data://my-resource")
async def my_resource() -> str:
    """Description of this resource"""
    return json.dumps({"status": "active", "data": "sample"})

@mcp.prompt()
def usage_guidance() -> str:
    """How to use this server effectively"""
    return """
    This server provides tools for [your use case].
    Use my_first_tool to [specific functionality].
    Check my_resource for [what it provides].
    """

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

22

The Ecosystem Advantage

The real power of MCP isn't in individual servers - it's in the ecosystem. You get an enormous ecosystem of plug-and-play tools that you can bring to any chat window that implements the standard.

Companies like GitHub, Notion, and others are building official MCP servers. As a developer, you can:

- Use existing servers for common services
- Build custom servers for your specific needs
- Share your servers with the community
- Mix and match capabilities from different servers

Future-Proofing Your AI Workflow

The tools we build today should keep working even as new models, clients, and services come around. MCP provides this future-proofing by creating a stable interface between AI systems and external capabilities.

As new AI models emerge and existing ones evolve, your MCP servers continue working without modification. This is particularly valuable in the rapidly changing AI landscape where model capabilities and client applications are constantly evolving.

Getting Started Today

Building MCP servers is more accessible than many developers realize. If you want to just hand the AI agent the MCP docs and tell it what functionalities you want... well, it's probably gonna work. This is the kind of code AI is especially good at—it's essentially boilerplate.

Start with a simple server for a tool you use regularly. The development cycle is:

1. Define your tools and resources
2. Test with MCP Inspector
3. Connect to your preferred AI client
4. Iterate based on real usage

The MCP ecosystem is still young, which means there's tremendous opportunity to build servers that solve real problems for developers and organizations.

Conclusion

MCP represents a fundamental shift in how we think about AI integration. Instead of building point-to-point connections between AI models and external services, we're creating a standardized ecosystem where capabilities can be shared, reused, and combined in powerful ways.

Whether you're building internal tools for your team or contributing to the broader MCP ecosystem, the protocol provides a solid foundation for creating AI integrations that will remain valuable as the technology landscape continues to evolve.

The future of AI development isn't just about more powerful models - it's about creating seamless experiences where AI can intelligently interact with all the tools and data sources we use daily. MCP is a crucial building block for that future, and now is the perfect time to start building with it.

About Natoma

Natoma enables enterprises to adopt AI agents securely. The secure agent access gateway empowers organizations to unlock the full power of AI, by connecting agents to their tools and data without compromising security.

Leveraging a hosted MCP platform, Natoma provides enterprise-grade authentication, fine-grained authorization, and governance for AI agents with flexible deployment models and out-of-the-box support for 100+ pre-built MCP servers.

To learn more, visit natoma.id or connect with our team directly at natoma.id/book-a-demo.