Model Context Protocol (MCP) - Complete Python Guide

Table of Contents

What is MCP?

Model Context Protocol (MCP) is an open standard that enables AI assistants to securely connect to data sources and tools. Think of it as a universal adapter that lets AI models interact with external services, databases, APIs, and tools.

Key Benefits:

- **Standardized**: One protocol for all integrations
- **Secure**: Built-in authentication and sandboxing
- **Extensible**: Easy to add new capabilities
- **Bi-directional**: Both client and server can initiate communication

- **Language Agnostic**: Works with Python, TypeScript, and more

Core Concepts

1. Servers and Clients

- **MCP Server**: Provides resources, tools, or prompts (what we'll build)
- **MCP Client**: Consumes the server's capabilities (usually an AI assistant like Claude)

2. Resources

Data that can be read by the client:

```python
from dataclasses import dataclass
from typing import Optional

@dataclass
class Resource:
    uri: str          # Unique identifier like "file://document.txt"
    name: str         # Human-readable name
    description: Optional[str] = None  # Optional description
    mimeType: Optional[str] = None     # Content type like "text/plain"
```

3. Tools

Functions that can be executed by the AI:

```python
```

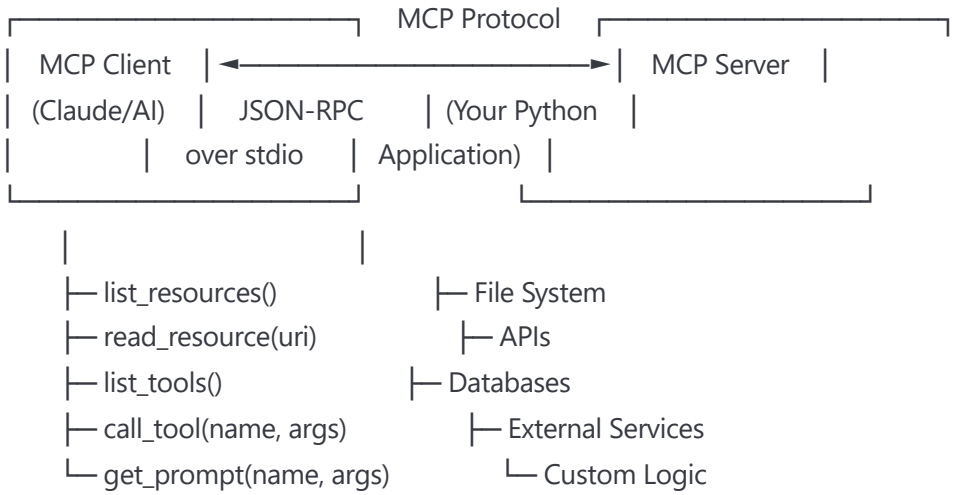
```python
@dataclass
class Tool:
    name: str           # Function name like "generate_pdf"
    description: str    # What the tool does
    inputSchema: dict   # JSON Schema defining parameters
```

4. Prompts

Reusable prompt templates with arguments:

```python
@dataclass
class Prompt:
    name: str
    description: str
    arguments: Optional[list] = None
```

Architecture



Setting Up Your Development Environment

1. Create a Virtual Environment

```bash
# Create project directory
mkdir mcp-tutorial
cd mcp-tutorial

# Create virtual environment
python -m venv mcp-env

# Activate it
# On Windows:
mcp-env\Scripts\activate
# On macOS/Linux:
source mcp-env/bin/activate
```

2. Install Dependencies

```bash
# Install MCP SDK
pip install mcp

# For our PDF example, we'll also need:
pip install reportlab  # For PDF generation
pip install requests   # For HTTP requests
```

3. Project Structure

```
mcp-tutorial/
├── mcp-env/
├── servers/
│   ├── __init__.py
│   ├── basic_server.py
│   └── pdf_server.py
├── requirements.txt
└── README.md
```

Create requirements.txt:

```txt
mcp>=0.5.0
reportlab>=4.0.0
requests>=2.31.0
```

Building Your First MCP Server

Let's start with a simple echo server to understand the basics:

Basic Server (servers/basic_server.py)

python

```python
#!/usr/bin/env python3
"""
A basic MCP server that demonstrates core concepts.
"""

import asyncio
import json
from typing import Any, Dict, List
from mcp.server.models import InitializationOptions
import mcp.types as types
from mcp.server import NotificationOptions, Server
import mcp.server.stdio

# Create server instance
server = Server("basic-mcp-server")

@server.list_tools()
async def handle_list_tools() -> List[types.Tool]:
    """Return list of available tools."""
    return [
        types.Tool(
            name="echo",
            description="Echo back any message you send",
            inputSchema={
                "type": "object",
                "properties": {
                    "message": {
                        "type": "string",
                        "description": "The message to echo back"
                    }
                },
                "required": ["message"]
```

```
        }
    ),
    types.Tool(
        name="add_numbers",
        description="Add two numbers together",
        inputSchema={
            "type": "object",
            "properties": {
                "a": {
                    "type": "number",
                    "description": "First number"
                },
                "b": {
                    "type": "number",
                    "description": "Second number"
                }
            },
            "required": ["a", "b"]
        }
    ),
    types.Tool(
        name="get_weather",
        description="Get current weather for a city (mock data)",
        inputSchema={
            "type": "object",
            "properties": {
                "city": {
                    "type": "string",
                    "description": "City name"
                }
            },
            "required": ["city"]
```

```
            }
        )
    ]

@server.call_tool()
async def handle_call_tool(
    name: str, arguments: Dict[str, Any]
) -> List[types.TextContent]:
    """Handle tool execution."""

    if name == "echo":
        message = arguments.get("message", "")
        return [
            types.TextContent(
                type="text",
                text=f"Echo: {message}"
            )
        ]

    elif name == "add_numbers":
        a = arguments.get("a", 0)
        b = arguments.get("b", 0)
        result = a + b
        return [
            types.TextContent(
                type="text",
                text=f"The sum of {a} and {b} is {result}"
            )
        ]

    elif name == "get_weather":
        city = arguments.get("city", "")
```

```
        # Mock weather data
        weather_data = {
            "city": city,
            "temperature": "22°C",
            "condition": "Sunny",
            "humidity": "65%"
        }
        return [
            types.TextContent(
                type="text",
                text=f"Weather in {city}: {weather_data['temperature']}, {weather_data['condition']}, Humidity: {weather_
            )
        ]

    else:
        raise ValueError(f"Unknown tool: {name}")

async def main():
    # Run the server using stdio transport
    async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
        await server.run(
            read_stream,
            write_stream,
            InitializationOptions(
                server_name="basic-mcp-server",
                server_version="1.0.0",
                capabilities=server.get_capabilities(
                    notification_options=NotificationOptions(),
                    experimental_capabilities={},
                ),
            ),
        )
```

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

PDF Generator MCP Server

Now let's build a practical PDF generator server:

PDF Server (servers/pdf_server.py)

python

```python
#!/usr/bin/env python3
"""
MCP Server for generating PDF documents.
"""

import asyncio
import os
import json
from datetime import datetime
from pathlib import Path
from typing import Any, Dict, List, Optional
from mcp.server.models import InitializationOptions
import mcp.types as types
from mcp.server import NotificationOptions, Server
import mcp.server.stdio

# PDF generation imports
from reportlab.lib.pagesizes import letter, A4
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.units import inch
from reportlab.pdfgen import canvas

class PDFGenerator:
    """Handles PDF generation operations."""

    def __init__(self, output_dir: str = "generated_pdfs"):
        self.output_dir = Path(output_dir)
        self.output_dir.mkdir(exist_ok=True)

    def create_simple_pdf(self, filename: str, title: str, content: str,
                          font_size: int = 12) -> str:
```

```
    """Create a simple PDF with title and content."""
    filepath = self.output_dir / f"{filename}.pdf"

    # Create PDF document
    doc = SimpleDocTemplate(
        str(filepath),
        pagesize=letter,
        rightMargin=72,
        leftMargin=72,
        topMargin=72,
        bottomMargin=18
    )

    # Get styles
    styles = getSampleStyleSheet()
    title_style = ParagraphStyle(
        'CustomTitle',
        parent=styles['Heading1'],
        fontSize=18,
        spaceAfter=30,
        alignment=1  # Center alignment
    )

    content_style = ParagraphStyle(
        'CustomContent',
        parent=styles['Normal'],
        fontSize=font_size,
        spaceAfter=12,
        alignment=0  # Left alignment
    )

    # Build story (content)
```

```
    story = []

    # Add title
    if title:
        story.append(Paragraph(title, title_style))
        story.append(Spacer(1, 12))

    # Add content (split by paragraphs)
    for paragraph in content.split('\n\n'):
        if paragraph.strip():
            story.append(Paragraph(paragraph.strip(), content_style))
            story.append(Spacer(1, 6))

    # Add footer with timestamp
    footer_style = ParagraphStyle(
        'Footer',
        parent=styles['Normal'],
        fontSize=8,
        alignment=1
    )
    story.append(Spacer(1, 50))
    story.append(Paragraph(f"Generated on {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}", footer_style))

    # Build PDF
    doc.build(story)

    return str(filepath)

def create_report_pdf(self, filename: str, data: Dict[str, Any]) -> str:
    """Create a structured report PDF."""
    filepath = self.output_dir / f"{filename}.pdf"
```

```
    doc = SimpleDocTemplate(str(filepath), pagesize=A4)
    styles = getSampleStyleSheet()
    story = []

    # Title
    title = data.get('title', 'Report')
    story.append(Paragraph(title, styles['Title']))
    story.append(Spacer(1, 12))

    # Sections
    sections = data.get('sections', [])
    for section in sections:
        section_title = section.get('title', '')
        section_content = section.get('content', '')

        if section_title:
            story.append(Paragraph(section_title, styles['Heading2']))
            story.append(Spacer(1, 6))

        if section_content:
            story.append(Paragraph(section_content, styles['Normal']))
            story.append(Spacer(1, 12))

    doc.build(story)
    return str(filepath)

def list_generated_pdfs(self) -> List[Dict[str, Any]]:
    """List all generated PDF files."""
    pdfs = []
    for pdf_file in self.output_dir.glob("*.pdf"):
        stat = pdf_file.stat()
        pdfs.append({
```

```
                'filename': pdf_file.name,
                'filepath': str(pdf_file),
                'size_bytes': stat.st_size,
                'created': datetime.fromtimestamp(stat.st_ctime).isoformat(),
                'modified': datetime.fromtimestamp(stat.st_mtime).isoformat()
            })
        return sorted(pdfs, key=lambda x: x['modified'], reverse=True)

# Initialize PDF generator
pdf_gen = PDFGenerator()

# Create server instance
server = Server("pdf-generator-server")

@server.list_tools()
async def handle_list_tools() -> List[types.Tool]:
    """Return list of available PDF tools."""
    return [
        types.Tool(
            name="generate_simple_pdf",
            description="Generate a simple PDF document with title and content",
            inputSchema={
                "type": "object",
                "properties": {
                    "filename": {
                        "type": "string",
                        "description": "Name of the PDF file (without .pdf extension)"
                    },
                    "title": {
                        "type": "string",
                        "description": "Title of the document"
                    },
```

```
            "content": {
                "type": "string",
                "description": "Main content of the PDF. Use double newlines for paragraph breaks."
            },
            "font_size": {
                "type": "integer",
                "description": "Font size for content (default: 12)",
                "default": 12,
                "minimum": 8,
                "maximum": 24
            }
        },
        "required": ["filename", "title", "content"]
    }
),
types.Tool(
    name="generate_report_pdf",
    description="Generate a structured report PDF with multiple sections",
    inputSchema={
        "type": "object",
        "properties": {
            "filename": {
                "type": "string",
                "description": "Name of the PDF file (without .pdf extension)"
            },
            "title": {
                "type": "string",
                "description": "Report title"
            },
            "sections": {
                "type": "array",
                "description": "Array of report sections",
```

```
                "items": {
                    "type": "object",
                    "properties": {
                        "title": {"type": "string"},
                        "content": {"type": "string"}
                    },
                    "required": ["title", "content"]
                }
            }
        },
        "required": ["filename", "title", "sections"]
    }
),
types.Tool(
    name="list_pdfs",
    description="List all generated PDF files with metadata",
    inputSchema={
        "type": "object",
        "properties": {}
    }
),
types.Tool(
    name="delete_pdf",
    description="Delete a generated PDF file",
    inputSchema={
        "type": "object",
        "properties": {
            "filename": {
                "type": "string",
                "description": "Name of the PDF file to delete (with or without .pdf extension)"
            }
        },
```

```
            "required": ["filename"]
        }
    )
]

@server.list_resources()
async def handle_list_resources() -> List[types.Resource]:
    """List PDF files as resources."""
    resources = []
    pdfs = pdf_gen.list_generated_pdfs()

    for pdf in pdfs:
        resources.append(
            types.Resource(
                uri=f"file://{pdf['filepath']}",
                name=pdf['filename'],
                description=f"Generated PDF ({pdf['size_bytes']} bytes, modified: {pdf['modified']})",
                mimeType="application/pdf"
            )
        )

    return resources

@server.read_resource()
async def handle_read_resource(uri: str) -> str:
    """Read a PDF resource (return file info since we can't return binary)."""
    if not uri.startswith("file://"):
        raise ValueError(f"Unsupported URI scheme: {uri}")

    filepath = Path(uri[7:])  # Remove "file://" prefix

    if not filepath.exists():
```

```
        raise FileNotFoundError(f"File not found: {filepath}")

    if not filepath.suffix.lower() == '.pdf':
        raise ValueError("Resource is not a PDF file")

    stat = filepath.stat()
    return f"""PDF File Information:
Name: {filepath.name}
Size: {stat.st_size} bytes
Created: {datetime.fromtimestamp(stat.st_ctime)}
Modified: {datetime.fromtimestamp(stat.st_mtime)}
Path: {filepath}

Note: This is a binary PDF file. Use a PDF viewer to open it."""

@server.call_tool()
async def handle_call_tool(
    name: str, arguments: Dict[str, Any]
) -> List[types.TextContent]:
    """Handle tool execution."""

    try:
        if name == "generate_simple_pdf":
            filename = arguments["filename"]
            title = arguments["title"]
            content = arguments["content"]
            font_size = arguments.get("font_size", 12)

            filepath = pdf_gen.create_simple_pdf(filename, title, content, font_size)

            return [
                types.TextContent(
```

```
            type="text",
            text=f"✅ PDF generated successfully!\n\nFile: {filepath}\nTitle: {title}\nFont size: {font_size}pt\n\nTh
        )
    ]

elif name == "generate_report_pdf":
    filename = arguments["filename"]
    data = {
        'title': arguments["title"],
        'sections': arguments["sections"]
    }

    filepath = pdf_gen.create_report_pdf(filename, data)
    section_count = len(data['sections'])

    return [
        types.TextContent(
            type="text",
            text=f"📄 Report PDF generated successfully!\n\nFile: {filepath}\nTitle: {data['title']}\nSections: {secti
        )
    ]

elif name == "list_pdfs":
    pdfs = pdf_gen.list_generated_pdfs()

    if not pdfs:
        return [
            types.TextContent(
                type="text",
                text="📁 No PDF files found in the output directory."
            )
        ]
```

```
        pdf_list = "📚 Generated PDF Files:\n\n"
        for i, pdf in enumerate(pdfs, 1):
            size_kb = pdf['size_bytes'] / 1024
            pdf_list += f"{i}. {pdf['filename']}\n"
            pdf_list += f"   Size: {size_kb:.1f} KB\n"
            pdf_list += f"   Modified: {pdf['modified']}\n\n"

        return [
            types.TextContent(
                type="text",
                text=pdf_list
            )
        ]

    elif name == "delete_pdf":
        filename = arguments["filename"]
        if not filename.endswith('.pdf'):
            filename += '.pdf'

        filepath = pdf_gen.output_dir / filename

        if not filepath.exists():
            return [
                types.TextContent(
                    type="text",
                    text=f"❌ File not found: {filename}"
                )
            ]

        filepath.unlink()  # Delete the file
```

```
            return [
                types.TextContent(
                    type="text",
                    text=f"🗑️ Successfully deleted: {filename}"
                )
            ]

        else:
            raise ValueError(f"Unknown tool: {name}")

    except Exception as e:
        return [
            types.TextContent(
                type="text",
                text=f"❌ Error: {str(e)}"
            )
        ]

async def main():
    """Run the PDF generator MCP server."""
    async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
        await server.run(
            read_stream,
            write_stream,
            InitializationOptions(
                server_name="pdf-generator-server",
                server_version="1.0.0",
                capabilities=server.get_capabilities(
                    notification_options=NotificationOptions(),
                    experimental_capabilities={},
                ),
            ),
```

```
        )

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

Testing Your Server

1. Manual Testing Script (test_server.py)

```
python
```

```python
#!/usr/bin/env python3
"""
Test script for MCP servers.
"""

import asyncio
import json
import subprocess
import sys
from typing import Any, Dict

async def test_mcp_server(server_script: str):
    """Test an MCP server by sending JSON-RPC messages."""

    # Start the server process
    process = await asyncio.create_subprocess_exec(
        sys.executable, server_script,
        stdin=asyncio.subprocess.PIPE,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE
    )

    async def send_request(method: str, params: Dict[str, Any] = None) -> Dict:
        """Send a JSON-RPC request to the server."""
        request = {
            "jsonrpc": "2.0",
            "id": 1,
            "method": method,
            "params": params or {}
        }

        request_json = json.dumps(request) + "\n"
```

```
    process.stdin.write(request_json.encode())
    await process.stdin.drain()

    # Read response
    response_line = await process.stdout.readline()
    return json.loads(response_line.decode())

try:
    # Initialize the server
    init_response = await send_request("initialize", {
        "protocolVersion": "2024-11-05",
        "capabilities": {},
        "clientInfo": {"name": "test-client", "version": "1.0.0"}
    })
    print("✅ Server initialized:", init_response.get("result", {}).get("serverInfo", {}))

    # List tools
    tools_response = await send_request("tools/list")
    tools = tools_response.get("result", {}).get("tools", [])
    print(f"🛠️ Available tools: {len(tools)}")
    for tool in tools:
        print(f"   - {tool['name']}: {tool['description']}")

    # Test a tool (if available)
    if tools:
        first_tool = tools[0]
        print(f"\n🧪 Testing tool: {first_tool['name']}")

        # Create test arguments based on the tool
        test_args = {}
        if first_tool['name'] == 'echo':
            test_args = {"message": "Hello, MCP!"}
```

```
            elif first_tool['name'] == 'add_numbers':
                test_args = {"a": 5, "b": 3}
            elif first_tool['name'] == 'generate_simple_pdf':
                test_args = {
                    "filename": "test_document",
                    "title": "Test PDF Document",
                    "content": "This is a test PDF generated by our MCP server.\n\nIt has multiple paragraphs to demons
                }

            if test_args:
                tool_response = await send_request("tools/call", {
                    "name": first_tool['name'],
                    "arguments": test_args
                })
                result = tool_response.get("result", {})
                content = result.get("content", [])
                if content:
                    print(f"📝 Tool result: {content[0].get('text', 'No text content')}")

    except Exception as e:
        print(f"❌ Error testing server: {e}")

    finally:
        # Clean up
        process.terminate()
        await process.wait()

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python test_server.py <server_script.py>")
        sys.exit(1)
```

```
    server_script = sys.argv[1]
    asyncio.run(test_mcp_server(server_script))
```

2. Run Tests

```bash
# Test basic server
python test_server.py servers/basic_server.py

# Test PDF server
python test_server.py servers/pdf_server.py
```

Advanced Features

1. Adding Prompts

```python
```

```
@server.list_prompts()
async def handle_list_prompts() -> List[types.Prompt]:
    return [
        types.Prompt(
            name="generate_report",
            description="Generate a structured business report",
            arguments=[
                types.PromptArgument(
                    name="company_name",
                    description="Name of the company",
                    required=True
                ),
                types.PromptArgument(
                    name="report_type",
                    description="Type of report (quarterly, annual, etc.)",
                    required=False
                )
            ]
        )
    ]

@server.get_prompt()
async def handle_get_prompt(
    name: str, arguments: Dict[str, str]
) -> types.GetPromptResult:
    if name == "generate_report":
        company = arguments.get("company_name", "Your Company")
        report_type = arguments.get("report_type", "quarterly")

        prompt_text = f"""Please generate a {report_type} report for {company}.

Include the following sections:
```

```
1. Executive Summary
2. Financial Performance
3. Key Achievements
4. Challenges and Risks
5. Future Outlook

Make the report professional and detailed."""

        return types.GetPromptResult(
            description=f"Generated {report_type} report prompt for {company}",
            messages=[
                types.PromptMessage(
                    role="user",
                    content=types.TextContent(type="text", text=prompt_text)
                )
            ]
        )

    raise ValueError(f"Unknown prompt: {name}")
```

2. Error Handling and Logging

python

```python
import logging

# Set up logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('mcp_server.log'),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger(__name__)

@server.call_tool()
async def handle_call_tool(
    name: str, arguments: Dict[str, Any]
) -> List[types.TextContent]:
    logger.info(f"Tool called: {name} with args: {arguments}")

    try:
        # Your tool logic here
        result = await execute_tool(name, arguments)
        logger.info(f"Tool {name} executed successfully")
        return result

    except Exception as e:
        logger.error(f"Tool {name} failed: {str(e)}")
        return [
            types.TextContent(
                type="text",
                text=f"Error executing {name}: {str(e)}"
```

```
        )
    ]
```

Best Practices

1. Input Validation

python

```python
def validate_filename(filename: str) -> str:
    """Validate and sanitize filename."""
    import re

    # Remove invalid characters
    clean_name = re.sub(r'[<>:"/\\|?*]', '', filename)

    # Limit length
    if len(clean_name) > 100:
        clean_name = clean_name[:100]

    # Ensure not empty
    if not clean_name:
        clean_name = "document"

    return clean_name
```

2. Configuration Management

python

```python
import os
from dataclasses import dataclass

@dataclass
class ServerConfig:
    output_dir: str = "generated_pdfs"
    max_file_size: int = 10 * 1024 * 1024  # 10MB
    allowed_formats: list = None

    def __post_init__(self):
        if self.allowed_formats is None:
            self.allowed_formats = ['pdf']

        # Create output directory
        os.makedirs(self.output_dir, exist_ok=True)

# Load config from environment
config = ServerConfig(
    output_dir=os.getenv('MCP_OUTPUT_DIR', 'generated_pdfs'),
    max_file_size=int(os.getenv('MCP_MAX_FILE_SIZE', '10485760'))
)
```

3. Async Operations

```python
```

```python
import aiofiles
import aiohttp

async def download_and_process(url: str) -> str:
    """Download content and process it asynchronously."""
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            content = await response.text()

    # Process content asynchronously
    processed = await process_content_async(content)
    return processed
```

4. Testing Your Server

Create comprehensive tests:

python

```python
# tests/test_pdf_server.py
import pytest
import asyncio
import tempfile
from pathlib import Path
from servers.pdf_server import PDFGenerator

@pytest.fixture
def pdf_generator():
    with tempfile.TemporaryDirectory() as temp_dir:
        yield PDFGenerator(temp_dir)

@pytest.mark.asyncio
async def test_create_simple_pdf(pdf_generator):
    """Test PDF creation."""
    filepath = pdf_generator.create_simple_pdf(
        filename="test",
        title="Test Document",
        content="This is test content."
    )

    assert Path(filepath).exists()
    assert Path(filepath).suffix == '.pdf'
    assert Path(filepath).stat().st_size > 0

@pytest.mark.asyncio
async def test_list_pdfs(pdf_generator):
    """Test PDF listing."""
    # Create a test PDF
    pdf_generator.create_simple_pdf("test1", "Test 1", "Content 1")
    pdf_generator.create_simple_pdf("test2", "Test 2", "Content 2")
```

```
    pdfs = pdf_generator.list_generated_pdfs()
    assert len(pdfs) == 2
    assert all('filename' in pdf for pdf in pdfs)
```

Run tests with:

```bash
pip install pytest pytest-asyncio
pytest tests/
```

Getting Started - Quick Commands

```bash
# 1. Set up your environment
python -m venv mcp-env
source mcp-env/bin/activate  # On Windows: mcp-env\Scripts\activate
pip install mcp reportlab

# 2. Create your first server
mkdir servers
# Copy the basic_server.py code above into servers/basic_server.py

# 3. Test it
python servers/basic_server.py

# 4. For PDF server
python servers/pdf_server.py
```

This comprehensive guide gives you everything you need to start building MCP servers with Python. The PDF generator is a practical example you can extend and customize for your needs!

Real-World Examples and Extensions

1. Database Integration Server

python

```python
#!/usr/bin/env python3
"""
MCP Server with database integration using SQLite.
"""

import asyncio
import sqlite3
from pathlib import Path
from typing import Any, Dict, List
import mcp.types as types
from mcp.server import Server
import mcp.server.stdio
from mcp.server.models import InitializationOptions
from mcp.server import NotificationOptions

class DatabaseServer:
    def __init__(self, db_path: str = "mcp_database.db"):
        self.db_path = db_path
        self.init_database()

    def init_database(self):
        """Initialize the database with sample tables."""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Create users table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                email TEXT UNIQUE NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

```
        )
    ''')

    # Create tasks table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS tasks (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER,
            title TEXT NOT NULL,
            description TEXT,
            status TEXT DEFAULT 'pending',
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (user_id) REFERENCES users (id)
        )
    ''')

    conn.commit()
    conn.close()

def execute_query(self, query: str, params: tuple = ()) -> List[Dict]:
    """Execute a SQL query and return results."""
    conn = sqlite3.connect(self.db_path)
    conn.row_factory = sqlite3.Row  # This enables column access by name
    cursor = conn.cursor()

    try:
        cursor.execute(query, params)

        if query.strip().upper().startswith('SELECT'):
            rows = cursor.fetchall()
            result = [dict(row) for row in rows]
        else:
```

```
            conn.commit()
            result = [{"affected_rows": cursor.rowcount, "last_row_id": cursor.lastrowid}]

        return result
    finally:
        conn.close()

# Initialize database server
db_server = DatabaseServer()

# Create MCP server
server = Server("database-mcp-server")

@server.list_tools()
async def handle_list_tools() -> List[types.Tool]:
    return [
        types.Tool(
            name="execute_sql",
            description="Execute a SQL query on the database",
            inputSchema={
                "type": "object",
                "properties": {
                    "query": {
                        "type": "string",
                        "description": "SQL query to execute"
                    },
                    "params": {
                        "type": "array",
                        "description": "Parameters for the SQL query",
                        "items": {"type": ["string", "number", "null"]},
                        "default": []
                    }
```

```
            },
            "required": ["query"]
        }
    ),
    types.Tool(
        name="add_user",
        description="Add a new user to the database",
        inputSchema={
            "type": "object",
            "properties": {
                "name": {"type": "string", "description": "User's full name"},
                "email": {"type": "string", "description": "User's email address"}
            },
            "required": ["name", "email"]
        }
    ),
    types.Tool(
        name="add_task",
        description="Add a new task for a user",
        inputSchema={
            "type": "object",
            "properties": {
                "user_id": {"type": "integer", "description": "ID of the user"},
                "title": {"type": "string", "description": "Task title"},
                "description": {"type": "string", "description": "Task description"},
                "status": {"type": "string", "description": "Task status", "default": "pending"}
            },
            "required": ["user_id", "title"]
        }
    ),
    types.Tool(
        name="get_user_tasks",
```

```
            description="Get all tasks for a specific user",
            inputSchema={
                "type": "object",
                "properties": {
                    "user_id": {"type": "integer", "description": "ID of the user"}
                },
                "required": ["user_id"]
            }
        )
    ]

@server.call_tool()
async def handle_call_tool(name: str, arguments: Dict[str, Any]) -> List[types.TextContent]:
    try:
        if name == "execute_sql":
            query = arguments["query"]
            params = tuple(arguments.get("params", []))

            result = db_server.execute_query(query, params)

            return [types.TextContent(
                type="text",
                text=f"Query executed successfully:\n{result}"
            )]

        elif name == "add_user":
            name_val = arguments["name"]
            email = arguments["email"]

            result = db_server.execute_query(
                "INSERT INTO users (name, email) VALUES (?, ?)",
                (name_val, email)
```

```
        )

        return [types.TextContent(
            type="text",
            text=f"✅ User added successfully! ID: {result[0]['last_row_id']}"
        )]

    elif name == "add_task":
        user_id = arguments["user_id"]
        title = arguments["title"]
        description = arguments.get("description", "")
        status = arguments.get("status", "pending")

        result = db_server.execute_query(
            "INSERT INTO tasks (user_id, title, description, status) VALUES (?, ?, ?, ?)",
            (user_id, title, description, status)
        )

        return [types.TextContent(
            type="text",
            text=f"✅ Task added successfully! ID: {result[0]['last_row_id']}"
        )]

    elif name == "get_user_tasks":
        user_id = arguments["user_id"]

        tasks = db_server.execute_query(
            "SELECT * FROM tasks WHERE user_id = ? ORDER BY created_at DESC",
            (user_id,)
        )

        if not tasks:
```

```
            return [types.TextContent(
                type="text",
                text=f"No tasks found for user ID {user_id}"
            )]

        task_list = f"📋 Tasks for User ID {user_id}:\n\n"
        for task in tasks:
            task_list += f"• {task['title']} ({task['status']})\n"
            if task['description']:
                task_list += f"  {task['description']}\n"
            task_list += f"  Created: {task['created_at']}\n\n"

        return [types.TextContent(type="text", text=task_list)]

    else:
        raise ValueError(f"Unknown tool: {name}")

  except Exception as e:
    return [types.TextContent(
        type="text",
        text=f"❌ Error: {str(e)}"
    )]

async def main():
  async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
    await server.run(
        read_stream,
        write_stream,
        InitializationOptions(
            server_name="database-mcp-server",
            server_version="1.0.0",
            capabilities=server.get_capabilities(
```

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

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

2. Web Scraping and API Integration Server

python

```python
#!/usr/bin/env python3
"""
MCP Server for web scraping and API integration.
"""

import asyncio
import aiohttp
import json
from bs4 import BeautifulSoup
from typing import Any, Dict, List
import mcp.types as types
from mcp.server import Server
import mcp.server.stdio
from mcp.server.models import InitializationOptions
from mcp.server import NotificationOptions

server = Server("web-integration-server")

@server.list_tools()
async def handle_list_tools() -> List[types.Tool]:
    return [
        types.Tool(
            name="fetch_webpage",
            description="Fetch and extract text content from a webpage",
            inputSchema={
                "type": "object",
                "properties": {
                    "url": {"type": "string", "description": "URL to fetch"},
                    "selector": {"type": "string", "description": "CSS selector to extract specific content (optional)"}
                },
                "required": ["url"]
            }
```

```
    ),
    types.Tool(
        name="fetch_json_api",
        description="Fetch data from a JSON API endpoint",
        inputSchema={
            "type": "object",
            "properties": {
                "url": {"type": "string", "description": "API endpoint URL"},
                "method": {"type": "string", "description": "HTTP method", "default": "GET"},
                "headers": {"type": "object", "description": "HTTP headers", "default": {}},
                "data": {"type": "object", "description": "Request data for POST/PUT", "default": {}}
            },
            "required": ["url"]
        }
    ),
    types.Tool(
        name="search_news",
        description="Search for news articles (using NewsAPI - requires API key)",
        inputSchema={
            "type": "object",
            "properties": {
                "query": {"type": "string", "description": "Search query"},
                "language": {"type": "string", "description": "Language code", "default": "en"},
                "sort_by": {"type": "string", "description": "Sort by: relevancy, popularity, publishedAt", "default": "pub
            },
            "required": ["query"]
        }
    ),
    types.Tool(
        name="get_weather",
        description="Get weather information (using OpenWeatherMap API - requires API key)",
        inputSchema={
```

```
            "type": "object",
            "properties": {
                "city": {"type": "string", "description": "City name"},
                "country": {"type": "string", "description": "Country code (optional)"}
            },
            "required": ["city"]
        }
    )
]

@server.call_tool()
async def handle_call_tool(name: str, arguments: Dict[str, Any]) -> List[types.TextContent]:
    try:
        if name == "fetch_webpage":
            url = arguments["url"]
            selector = arguments.get("selector")

            async with aiohttp.ClientSession() as session:
                async with session.get(url) as response:
                    html = await response.text()

            soup = BeautifulSoup(html, 'html.parser')

            if selector:
                elements = soup.select(selector)
                content = "\n".join([elem.get_text().strip() for elem in elements])
            else:
                # Extract main content, remove scripts and styles
                for script in soup(["script", "style"]):
                    script.decompose()
                content = soup.get_text()
                # Clean up whitespace
```

```
            lines = (line.strip() for line in content.splitlines())
            content = "\n".join(line for line in lines if line)

        return [types.TextContent(
            type="text",
            text=f"Content from {url}:\n\n{content[:2000]}{'...' if len(content) > 2000 else ''}"
        )]

    elif name == "fetch_json_api":
        url = arguments["url"]
        method = arguments.get("method", "GET").upper()
        headers = arguments.get("headers", {})
        data = arguments.get("data", {})

        async with aiohttp.ClientSession() as session:
            if method == "GET":
                async with session.get(url, headers=headers) as response:
                    result = await response.json()
            elif method == "POST":
                async with session.post(url, headers=headers, json=data) as response:
                    result = await response.json()
            else:
                raise ValueError(f"Unsupported HTTP method: {method}")

        return [types.TextContent(
            type="text",
            text=f"API Response from {url}:\n\n{json.dumps(result, indent=2)[:2000]}{'...' if len(str(result)) > 2000 els
        )]

    elif name == "search_news":
        # Note: This requires a NewsAPI key
        api_key = "YOUR_NEWSAPI_KEY"  # Replace with actual API key or env variable
```

```
        query = arguments["query"]
        language = arguments.get("language", "en")
        sort_by = arguments.get("sort_by", "publishedAt")

        url = f"https://newsapi.org/v2/everything?q={query}&language={language}&sortBy={sort_by}&apiKey={

        async with aiohttp.ClientSession() as session:
            async with session.get(url) as response:
                data = await response.json()

        if data.get("status") != "ok":
            return [types.TextContent(
                type="text",
                text=f"❌ News API Error: {data.get('message', 'Unknown error')}"
            )]

        articles = data.get("articles", [])[:5]  # Limit to 5 articles

        news_text = f"📰 News articles for '{query}':\n\n"
        for i, article in enumerate(articles, 1):
            news_text += f"{i}. {article['title']}\n"
            news_text += f"   Source: {article['source']['name']}\n"
            news_text += f"   Published: {article['publishedAt']}\n"
            news_text += f"   Description: {article['description']}\n"
            news_text += f"   URL: {article['url']}\n\n"

        return [types.TextContent(type="text", text=news_text)]

    elif name == "get_weather":
        # Note: This requires an OpenWeatherMap API key
        api_key = "YOUR_OPENWEATHER_API_KEY"  # Replace with actual API key
        city = arguments["city"]
```

```
            country = arguments.get("country", "")

            location = f"{city},{country}" if country else city
            url = f"http://api.openweathermap.org/data/2.5/weather?q={location}&appid={api_key}&units=metric"

            async with aiohttp.ClientSession() as session:
                async with session.get(url) as response:
                    data = await response.json()

            if data.get("cod") != 200:
                return [types.TextContent(
                    type="text",
                    text=f"❌ Weather API Error: {data.get('message', 'Unknown error')}"
                )]

            weather_text = f"🌤️ Weather in {data['name']}, {data['sys']['country']}:\n\n"
            weather_text += f"Temperature: {data['main']['temp']}°C (feels like {data['main']['feels_like']}°C)\n"
            weather_text += f"Condition: {data['weather'][0]['description'].title()}\n"
            weather_text += f"Humidity: {data['main']['humidity']}%\n"
            weather_text += f"Wind Speed: {data['wind']['speed']} m/s\n"
            weather_text += f"Pressure: {data['main']['pressure']} hPa\n"

            return [types.TextContent(type="text", text=weather_text)]

        else:
            raise ValueError(f"Unknown tool: {name}")

    except Exception as e:
        return [types.TextContent(
            type="text",
            text=f"❌ Error: {str(e)}"
        )]
```

```
async def main():
    async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
        await server.run(
            read_stream,
            write_stream,
            InitializationOptions(
                server_name="web-integration-server",
                server_version="1.0.0",
                capabilities=server.get_capabilities(
                    notification_options=NotificationOptions(),
                    experimental_capabilities={},
                ),
            ),
        )

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

3. File Management Server

python

```python
#!/usr/bin/env python3
"""
MCP Server for file management operations.
"""

import asyncio
import os
import shutil
import mimetypes
from pathlib import Path
from typing import Any, Dict, List
import mcp.types as types
from mcp.server import Server
import mcp.server.stdio
from mcp.server.models import InitializationOptions
from mcp.server import NotificationOptions

class FileManager:
    def __init__(self, base_dir: str = "mcp_workspace"):
        self.base_dir = Path(base_dir)
        self.base_dir.mkdir(exist_ok=True)

    def get_safe_path(self, filepath: str) -> Path:
        """Get a safe path within the base directory."""
        path = self.base_dir / filepath
        # Resolve to prevent directory traversal
        resolved_path = path.resolve()

        # Ensure the path is within base_dir
        if not str(resolved_path).startswith(str(self.base_dir.resolve())):
            raise ValueError("Path is outside the allowed directory")
```

```
        return resolved_path

file_manager = FileManager()
server = Server("file-management-server")

@server.list_tools()
async def handle_list_tools() -> List[types.Tool]:
    return [
        types.Tool(
            name="list_files",
            description="List files and directories in a path",
            inputSchema={
                "type": "object",
                "properties": {
                    "path": {"type": "string", "description": "Directory path (relative to workspace)", "default": "."},
                    "include_hidden": {"type": "boolean", "description": "Include hidden files", "default": False}
                }
            }
        ),
        types.Tool(
            name="read_file",
            description="Read content of a text file",
            inputSchema={
                "type": "object",
                "properties": {
                    "filepath": {"type": "string", "description": "Path to the file"},
                    "encoding": {"type": "string", "description": "File encoding", "default": "utf-8"}
                },
                "required": ["filepath"]
            }
        ),
        types.Tool(
```

```
        name="write_file",
        description="Write content to a file",
        inputSchema={
            "type": "object",
            "properties": {
                "filepath": {"type": "string", "description": "Path to the file"},
                "content": {"type": "string", "description": "Content to write"},
                "encoding": {"type": "string", "description": "File encoding", "default": "utf-8"}
            },
            "required": ["filepath", "content"]
        }
    ),
    types.Tool(
        name="create_directory",
        description="Create a new directory",
        inputSchema={
            "type": "object",
            "properties": {
                "path": {"type": "string", "description": "Directory path to create"}
            },
            "required": ["path"]
        }
    ),
    types.Tool(
        name="delete_file",
        description="Delete a file or directory",
        inputSchema={
            "type": "object",
            "properties": {
                "path": {"type": "string", "description": "Path to delete"},
                "recursive": {"type": "boolean", "description": "Delete directories recursively", "default": False}
            },
```

```
            "required": ["path"]
        }
    ),
    types.Tool(
        name="copy_file",
        description="Copy a file or directory",
        inputSchema={
            "type": "object",
            "properties": {
                "source": {"type": "string", "description": "Source path"},
                "destination": {"type": "string", "description": "Destination path"}
            },
            "required": ["source", "destination"]
        }
    ),
    types.Tool(
        name="move_file",
        description="Move/rename a file or directory",
        inputSchema={
            "type": "object",
            "properties": {
                "source": {"type": "string", "description": "Source path"},
                "destination": {"type": "string", "description": "Destination path"}
            },
            "required": ["source", "destination"]
        }
    ),
    types.Tool(
        name="get_file_info",
        description="Get detailed information about a file or directory",
        inputSchema={
            "type": "object",
```

```
            "properties": {
                "path": {"type": "string", "description": "Path to examine"}
            },
            "required": ["path"]
        }
    )
]

@server.list_resources()
async def handle_list_resources() -> List[types.Resource]:
    """List all text files as resources."""
    resources = []

    for file_path in file_manager.base_dir.rglob("*"):
        if file_path.is_file():
            # Only include text files as resources
            mime_type, _ = mimetypes.guess_type(str(file_path))
            if mime_type and mime_type.startswith('text/'):
                rel_path = file_path.relative_to(file_manager.base_dir)
                resources.append(
                    types.Resource(
                        uri=f"file://{rel_path}",
                        name=str(rel_path),
                        description=f"Text file ({file_path.stat().st_size} bytes)",
                        mimeType=mime_type
                    )
                )

    return resources

@server.read_resource()
async def handle_read_resource(uri: str) -> str:
```

```
    """Read a file resource."""
    if not uri.startswith("file://"):
        raise ValueError(f"Unsupported URI scheme: {uri}")

    rel_path = uri[7:]  # Remove "file://" prefix
    file_path = file_manager.get_safe_path(rel_path)

    if not file_path.exists():
        raise FileNotFoundError(f"File not found: {rel_path}")

    return file_path.read_text(encoding='utf-8')

@server.call_tool()
async def handle_call_tool(name: str, arguments: Dict[str, Any]) -> List[types.TextContent]:
    try:
        if name == "list_files":
            path = arguments.get("path", ".")
            include_hidden = arguments.get("include_hidden", False)

            dir_path = file_manager.get_safe_path(path)

            if not dir_path.exists():
                return [types.TextContent(
                    type="text",
                    text=f"❌ Directory not found: {path}"
                )]

            if not dir_path.is_dir():
                return [types.TextContent(
                    type="text",
                    text=f"❌ Path is not a directory: {path}"
                )]
```

```
        items = []
        for item in sorted(dir_path.iterdir()):
            if not include_hidden and item.name.startswith('.'):
                continue

            item_type = "📁" if item.is_dir() else "📄"
            size = f" ({item.stat().st_size} bytes)" if item.is_file() else ""
            items.append(f"{item_type} {item.name}{size}")

        if not items:
            result = f"📂 Directory '{path}' is empty"
        else:
            result = f"📂 Contents of '{path}':\n\n" + "\n".join(items)

        return [types.TextContent(type="text", text=result)]

    elif name == "read_file":
        filepath = arguments["filepath"]
        encoding = arguments.get("encoding", "utf-8")

        file_path = file_manager.get_safe_path(filepath)

        if not file_path.exists():
            return [types.TextContent(
                type="text",
                text=f"❌ File not found: {filepath}"
            )]

        content = file_path.read_text(encoding=encoding)

        return [types.TextContent(
```

```
            type="text",
            text=f"📄 Content of '{filepath}':\n\n{content}"
        )]

    elif name == "write_file":
        filepath = arguments["filepath"]
        content = arguments["content"]
        encoding = arguments.get("encoding", "utf-8")

        file_path = file_manager.get_safe_path(filepath)

        # Create parent directories if they don't exist
        file_path.parent.mkdir(parents=True, exist_ok=True)

        file_path.write_text(content, encoding=encoding)

        return [types.TextContent(
            type="text",
            text=f"✅ File written successfully: {filepath} ({len(content)} characters)"
        )]

    elif name == "create_directory":
        path = arguments["path"]

        dir_path = file_manager.get_safe_path(path)
        dir_path.mkdir(parents=True, exist_ok=True)

        return [types.TextContent(
            type="text",
            text=f"✅ Directory created: {path}"
        )]
```

```
elif name == "delete_file":
    path = arguments["path"]
    recursive = arguments.get("recursive", False)

    target_path = file_manager.get_safe_path(path)

    if not target_path.exists():
        return [types.TextContent(
            type="text",
            text=f"❌ Path not found: {path}"
        )]

    if target_path.is_dir():
        if recursive:
            shutil.rmtree(target_path)
        else:
            target_path.rmdir()  # Only works if empty
    else:
        target_path.unlink()

    return [types.TextContent(
        type="text",
        text=f"🗑️ Deleted: {path}"
    )]

elif name == "copy_file":
    source = arguments["source"]
    destination = arguments["destination"]

    src_path = file_manager.get_safe_path(source)
    dst_path = file_manager.get_safe_path(destination)
```

```
        if not src_path.exists():
            return [types.TextContent(
                type="text",
                text=f"❌ Source not found: {source}"
            )]

        # Create parent directories if needed
        dst_path.parent.mkdir(parents=True, exist_ok=True)

        if src_path.is_dir():
            shutil.copytree(src_path, dst_path)
        else:
            shutil.copy2(src_path, dst_path)

        return [types.TextContent(
            type="text",
            text=f"📋 Copied: {source} → {destination}"
        )]

    elif name == "move_file":
        source = arguments["source"]
        destination = arguments["destination"]

        src_path = file_manager.get_safe_path(source)
        dst_path = file_manager.get_safe_path(destination)

        if not src_path.exists():
            return [types.TextContent(
                type="text",
                text=f"❌ Source not found: {source}"
            )]
```

```
        # Create parent directories if needed
        dst_path.parent.mkdir(parents=True, exist_ok=True)

        shutil.move(str(src_path), str(dst_path))

        return [types.TextContent(
            type="text",
            text=f"📦 Moved: {source} → {destination}"
        )]

    elif name == "get_file_info":
        path = arguments["path"]

        target_path = file_manager.get_safe_path(path)

        if not target_path.exists():
            return [types.TextContent(
                type="text",
                text=f"❌ Path not found: {path}"
            )]

        stat = target_path.stat()

        info = f"📊 Information for '{path}':\n\n"
        info += f"Type: {'Directory' if target_path.is_dir() else 'File'}\n"
        info += f"Size: {stat.st_size} bytes\n"
        info += f"Created: {stat.st_ctime}\n"
        info += f"Modified: {stat.st_mtime}\n"
        info += f"Permissions: {oct(stat.st_mode)[-3:]}\n"

        if target_path.is_file():
            mime_type, _ = mimetypes.guess_type(str(target_path))
```

```
            info += f"MIME Type: {mime_type or 'Unknown'}\n"

            return [types.TextContent(type="text", text=info)]

        else:
            raise ValueError(f"Unknown tool: {name}")

    except Exception as e:
        return [types.TextContent(
            type="text",
            text=f"❌ Error: {str(e)}"
        )]

async def main():
    async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
        await server.run(
            read_stream,
            write_stream,
            InitializationOptions(
                server_name="file-management-server",
                server_version="1.0.0",
                capabilities=server.get_capabilities(
                    notification_options=NotificationOptions(),
                    experimental_capabilities={},
                ),
            ),
        )

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

Deployment and Configuration

1. Environment Configuration

Create a `.env` file for your secrets:

```bash
# .env
NEWSAPI_KEY=your_news_api_key_here
OPENWEATHER_API_KEY=your_openweather_api_key_here
DATABASE_URL=sqlite:///./app.db
MCP_OUTPUT_DIR=/path/to/output
MCP_LOG_LEVEL=INFO
```

Load environment variables in your server:

```python
import os
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

# Use in your code
NEWS_API_KEY = os.getenv('NEWSAPI_KEY')
WEATHER_API_KEY = os.getenv('OPENWEATHER_API_KEY')
```

2. Docker Deployment

Create a `Dockerfile`:

```dockerfile
```

```
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get
```