Design Decisions

This document logs important design decisions made during the development of BasicMcpServer.

2025-04-22: MCP Manager Development

Decision

We've developed a comprehensive command-line utility called `mcp-manager` to streamline the management of MCP servers across different environments.

Context

MCP servers can be deployed in various ways: as local stdio-based processes, as HTTP+SSE servers, or as remote services. Managing these different deployment methods and properly configuring them for tools like VS Code's Cline extension was becoming complex and error-prone.

Reasoning

Creating a dedicated management tool offers several benefits: 1. **Unified Interface**: A single tool manages all types of MCP servers 2. **Isolated Environments**: Each MCP server gets its own virtual environment 3. **Simplified Configuration**: Automatic generation of wrapper scripts and VS Code settings 4. **Consistent Structure**: Standardized directory structure under ~/.mcp_servers 5. **Modern Python Practices**: Using pipx for isolated CLI tool installation

Implementation

The implementation follows modern Python best practices: 1. **Project Structure**: - Organized as an installable Python package with pyproject.toml - Command-line interface using Typer - Rich terminal output with the Rich library - Jinja2 templates for generating configuration files

1. **Commands**:

2. `install`: Install local MCP servers with isolated environments
3. `add`: Configure remote MCP servers
4. `list`: List all configured servers
5. `run`: Run local servers with different transport modes

6. `configure`: Set up editor integration

7. **Directory Structure**: `~/.mcp_servers/ ├── servers/ # Local server installations │ ├── example-server/ # Each server gets its own directory │ │ ├── .venv/ # Isolated virtual environment │ │ ├── src/ # Source code (symlinked) │ │ └── meta.json # Metadata ├── bin/ # Generated wrapper scripts └── config/ # Configuration files └── servers.json # Server registry`

8. **VS Code Integration**:

9. Automatically configures the Cline extension settings
10. Generates wrapper scripts for stdio transport

Tradeoffs

- **Pros**:
- Simplified management of multiple MCP servers
- Consistent environment setup
- Easy configuration of VS Code/Cline integration

- Support for both stdio and HTTP+SSE transport modes

- **Cons**:

- Additional dependency (pipx) for installation
- More complex than direct configuration for simple use cases

The benefits of having a dedicated management tool significantly outweigh the minor complexity it introduces, especially as the number of MCP servers grows.

References

- Python packaging best practices: https://packaging.python.org/

- Typer documentation: https://typer.tiangolo.com/
- pipx documentation: https://pypa.github.io/pipx/

2025-04-22: Improved MCP Server Architecture with Command-Line Support

Decision

We've restructured the MCP server code to improve separation of concerns, made the MCP server name configurable via environment variables, simplified how the main Python module is called in Docker containers, optimized environment variable handling for Docker deployments, updated the import system to use direct imports instead of relative imports, and added a comprehensive command-line interface with built-in documentation.

Context

Previously, the MCP server name was hardcoded in the server.py file, and the Docker container used the module notation (src.main) to run the server, which was not aligned with treating src as the Python sources root.

Reasoning

Making these changes offers several benefits: 1. **Configurable Identity**: The server name can now be changed via environment variables without code changes 2. **Simplified Docker Execution**: Running the server with `python main` instead of `python -m src.main` is cleaner 3. **Proper Sources Root**: Setting PYTHONPATH to include src as a sources root provides a cleaner import structure 4. **Consistent Environment**: The changes make the development and Docker environments more consistent 5. **Flexible Deployment**: Multiple instances of the same code can have different names when deployed

Implementation Notes

The implementation involved: 1. Adding a `server_name` field to the Settings class in config.py 2. Updating server.py to use this configurable name 3. Modifying the Dockerfile to set PYTHONPATH and change working directory 4.

Updating the Docker CMD to run main.py directly 5. Adding SERVER_NAME to .env.example files

Tradeoffs

- **Pros**: More flexible server naming, cleaner Docker configuration, better alignment with Python practices
- **Cons**: Requires environment variables to be properly set for consistent naming

The benefits of configurable naming and simplified Docker execution outweigh the minor additional requirement to manage environment variables.

References

- Python PYTHONPATH documentation: https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH
- Docker WORKDIR documentation: https://docs.docker.com/engine/reference/builder/#workdir

2025-04-22: Removal of .mcp.json Files from MCP Server Projects

Decision

We've removed all .mcp.json files from the MCP server projects (example, jira-clone, and template directories) as they were not being used by the server code and were potentially confusing.

Context

The .mcp.json files were present in various MCP server projects but were not referenced by any of the server code. These files contained server definitions, tool descriptions, and resource listings that duplicated information already defined in the Python code.

Reasoning

Removing the .mcp.json files offers several benefits: 1. **Eliminates Redundancy**: The information was already defined in the Python code using FastMCP decorators 2. **Prevents Confusion**: Having both code-defined and JSON-defined schemas could lead to inconsistencies 3. **Simplifies Project Structure**: One less file to maintain and keep in sync 4. **Clarifies Runtime Discovery**: Emphasizes that FastMCP handles schema generation at runtime 5. **Better Documentation**: Updates the documentation to accurately reflect how VSCode/Cline discovers MCP servers

Implementation Notes

The implementation involved: 1. Removing .mcp.json files from all MCP server projects 2. Updating documentation to clarify that VSCode/Cline settings are configured separately 3. Updating README files to reflect the actual workflow for connecting to MCP servers 4. Confirming that no code in the project was reading or using these files

Tradeoffs

- **Pros**: Reduced redundancy, clearer project structure, more accurate documentation
- **Cons**: Slightly less explicit documentation of available tools/resources in the project files

The benefits of removing redundant files and clarifying the actual server discovery mechanism significantly outweigh the minor documentation convenience of having tool schemas in the project.

References

- FastMCP automatic schema generation: See readme_fastmcp.md
- VSCode/Cline MCP settings configuration: `~/.config/Code/User/globalStorage/saoudrizwan.claude-dev/settings/cline_mcp_settings.json`

2025-04-21: Removing .env Files from Git Tracking and Adding .gitignore

Decision

We've removed .env files from git tracking and added a .gitignore file to prevent accidentally committing sensitive information in the future.

Context

A .env file containing sensitive information was accidentally committed to the git repository. This poses a security risk as environment variables often contain secrets, API keys, passwords, and other sensitive configuration.

Reasoning

Removing .env files from git tracking while keeping them in the working directory offers several benefits: 1. **Security**: Prevents sensitive credentials from being exposed in the git history 2. **Compliance**: Aligns with security best practices for handling configuration data 3. **Flexibility**: Allows developers to maintain their local configuration without affecting others 4. **Documentation**: .env.example files can be used to document required variables without exposing actual values

Implementation Notes

The implementation involved: 1. Using `git rm --cached example/.env` to remove the file from git tracking while keeping it in the working directory 2. Creating a comprehensive .gitignore file with rules for .env files and other common exclusions 3. Committing the changes to ensure the .env file remains untracked 4. Maintaining .env.example files as templates for required variables

Tradeoffs

- **Pros**: Improved security, better secret management, compliance with best practices
- **Cons**: Requires developers to manually create .env files based on examples, slight additional onboarding complexity

The security benefits significantly outweigh the minor inconvenience of additional setup steps for new developers.

References

- Git documentation for untracking files: https://git-scm.com/docs/git-rm
- Environment variable security best practices

2025-04-21: Creation of Base MCP Server Docker Image

Decision

We've created a separate base Docker image for MCP servers that can be used as a foundation for all MCP server implementations, including the BasicMcpServer.

Context

Previously, each MCP server implementation had its own complete Dockerfile that included all dependencies and configurations, leading to duplication across different MCP server projects.

Reasoning

Extracting a base Docker image offers several benefits: 1. **Reduced Duplication**: Common dependencies and configurations are defined once 2. **Faster Builds**: Downstream images only need to build their unique components 3. **Consistency**: All MCP servers use the same base environment and dependencies 4. **Easier Maintenance**: Updates to common dependencies can be made in one place 5. **Simplified Server Implementations**: Server-specific Dockerfiles become much simpler

Implementation Notes

The implementation involved: 1. Creating a dedicated `base-mcp-docker/` directory 2. Extracting common MCP dependencies and configurations into a base Dockerfile 3. Keeping the multi-stage build approach for optimization 4. Implementing the base image with pre-installed MCP dependencies 5.

Updating the BasicMcpServer Dockerfile to use the base image 6. Removing the health check endpoint which was not functioning correctly

Tradeoffs

- **Pros**: Less duplication, better maintainability, consistent environment across servers
- **Cons**: Additional image to maintain, dependency on the base image for all server implementations

The benefits of having a shared base image significantly outweigh the minor additional maintenance overhead.

References

- Docker multi-stage build documentation: https://docs.docker.com/build/building/multi-stage/
- Container image best practices

2025-04-21: Separation of Base MCP Server into Dedicated Repository

Decision

We've moved the base MCP server Docker image to a separate repository and configured BasicMcpServer to use the pre-built image from Docker Hub.

Context

Previously, the base MCP server image was built within the same repository as BasicMcpServer, in the `base-mcp-docker/` directory. This approach coupled the base image with the specific implementation.

Reasoning

Extracting the base MCP server to an independent repository: 1. **Better Separation of Concerns**: Complete separation between base image and implementation 2. **Reusability**: Makes the base image more easily reusable across multiple projects 3. **Independent Versioning**: Base image can be versioned independently of specific implementations 4. **Simplified Workflow**: Implementation projects can simply pull the pre-built image 5. **Consistent Base**: Ensures all MCP servers use exactly the same base environment

Implementation Notes

The implementation involved: 1. Moving `base-mcp-docker/` to a separate repository 2. Updating the build scripts to use Docker Hub authentication 3. Modifying Dockerfile.new to accept Docker username as a build argument 4. Updating the build process to pull the base image instead of building it locally 5. Adding comprehensive documentation for both projects

Tradeoffs

- **Pros**: Cleaner separation, independent versioning, simplified implementation projects
- **Cons**: Requires Docker Hub access, adds dependency on external registry

The advantages of a separate repository model with pre-built images significantly outweigh the additional dependency on Docker Hub.

References

- Docker build arguments: https://docs.docker.com/engine/reference/commandline/build/#build-arg
- Docker Hub documentation: https://docs.docker.com/docker-hub/

2025-04-21: HTTP+SSE Protocol for MCP Communication

Decision

We've selected HTTP+SSE (Server-Sent Events) as the exclusive protocol for our containerized MCP server implementation, deliberately excluding support for stdio.

Context

The MCP Python SDK supports two main transport protocols: 1. **stdio**: Primarily designed for local development and desktop integration 2. **HTTP+SSE**: Better suited for networked environments and web-based integrations

Our containerized approach to MCP servers required selecting the most appropriate protocol for deployment in distributed environments.

Reasoning

Choosing HTTP+SSE exclusively offers several key benefits for containerized deployments:

1. **Network-First Architecture**: HTTP+SSE is designed for communication over networks, making it ideal for containerized environments that expose services over TCP/IP
2. **Scalability**: HTTP is stateless and can be load-balanced across multiple container instances
3. **Compatibility**: Works seamlessly with cloud environments, Kubernetes, and other orchestration systems
4. **Integration**: Can be easily mounted into existing web applications or exposed as a dedicated service
5. **Security**: Allows for standard web security practices (HTTPS, API keys, rate limiting)
6. **Observability**: HTTP traffic can be monitored and analyzed using standard web tools

In contrast, stdio is primarily designed for local communication between processes on the same machine, making it unsuitable for containerized deployments that need to expose services over a network.

Implementation Notes

The implementation leverages the MCP Python SDK's built-in HTTP+SSE support: 1. Using the `.sse_app()` method to create an ASGI application 2. Running this application with Uvicorn as the ASGI server 3. Exposing the server on a configurable port (default: 7501) 4. Providing documentation on mounting options for advanced integration scenarios

Tradeoffs

- **Pros**: Better network support, scalability, security, and integration capabilities
- **Cons**: Not directly compatible with local desktop integrations that expect stdio

The advantages of HTTP+SSE for containerized deployments significantly outweigh the limitations, particularly as our focus is on building services that can be deployed and scaled in cloud environments.

References

- MCP Python SDK documentation: https://github.com/modelcontextprotocol/python-sdk
- Server-Sent Events (SSE) documentation: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events
- ASGI specification: https://asgi.readthedocs.io/en/latest/

2025-04-15: Separation of Docker Containerization from Application Functionality

Decision

We've restructured the project to separate Docker containerization concerns from application functionality by: 1. Creating a dedicated `docker/` directory for all Docker-related files 2. Replacing Docker Compose with a single Dockerfile and helper scripts 3. Upgrading to Python 3.13 from Python 3.11 4. Implementing multi-stage builds for better optimization

Context

The previous setup had containerization concerns tightly coupled with application code, using Docker Compose for a single container, and running on Python 3.11 instead of 3.13.

Reasoning

This separation offers several benefits: 1. **Clear Separation of Concerns**: Containerization details are isolated from application code 2. **Simplified Workflow**: A single container doesn't require Docker Compose's complexity 3. **Version Upgrades**: Moving to Python 3.13 keeps us current with the latest language features 4. **Optimized Builds**: Multi-stage builds reduce image size and improve security 5. **Improved Maintainability**: Changes to containerization can be made independently of application code

Implementation Notes

The implementation involved: 1. Creating a new `docker/` directory with an optimized Dockerfile 2. Adding helper scripts to replace Docker Compose functionality 3. Upgrading Python version to 3.13 4. Implementing security best practices like non-root user and health checks 5. Providing comprehensive documentation in `docker/README.md`

Tradeoffs

- **Pros**: Better separation of concerns, simplified workflow, optimized container, better security
- **Cons**: Slightly more complex directory structure, need to maintain helper scripts

The tradeoffs heavily favor the new approach as it provides better maintainability, security, and follows the principle of separation of concerns.

References

- Docker Best Practices Documentation
- Python 3.13 Release Notes

2025-04-15: Migration from Low-Level Server to FastMCP

Decision

We have decided to migrate the BasicMcpServer from using the low-level `mcp.server.lowlevel.Server` class to the more ergonomic `mcp.server.fastmcp.FastMCP` class.

Context

The MCP Python SDK provides two main approaches for implementing MCP servers:

1. **Low-Level Server API**: Direct access to the MCP protocol with more manual implementation
2. **FastMCP API**: Higher-level, more ergonomic API with simplified abstractions

Our initial implementation used the low-level Server API, which required more boilerplate code and manual handling of various MCP protocol aspects.

Reasoning

The migration to FastMCP offers several significant benefits:

1. **Reduced Boilerplate**: FastMCP dramatically reduces the amount of code needed to implement the same functionality
2. **Type Safety**: Automatic type validation and conversion using Python type hints
3. **Automatic Schema Generation**: Input schemas are generated automatically from function signatures
4. **Simplified Error Handling**: Errors from tools are properly formatted and returned to clients
5. **Better Developer Experience**: More intuitive API that's easier to understand and maintain
6. **Access to Advanced Features**: Easier access to MCP capabilities like logging and progress reporting

The previous implementation required: - Manual definition of tool schemas - Explicit validation of input arguments - Manually formatting responses - Significant boilerplate for server setup

With FastMCP, these aspects are handled automatically, resulting in more maintainable code.

Implementation Notes

The migration involved:

1. Replacing the low-level Server instantiation with FastMCP
2. Converting tool implementations to use FastMCP's decorator pattern
3. Updating the server execution logic to use FastMCP's built-in functionality
4. Adding support for future extensions like resources and prompts

Tradeoffs

- **Pros**: Less code, better maintainability, easier access to advanced features
- **Cons**: Slightly higher level of abstraction (less direct control of the protocol)

The tradeoffs heavily favor FastMCP for our use case. The minimal loss of direct control is far outweighed by the significant reduction in code complexity and improved developer experience.

References

- FastMCP vs Low-Level Server Implementation
- MCP Python SDK Documentation