<!-- page slug: mcp-docs-sdk-java-mcp-server | order: 18 -->

- Breaking Changes in 0.8.x ⚠
- Overview
- Server Transport Providers
- Server Capabilities
  - Logging Support
  - Tool Specification
  - Resource Specification
  - Prompt Specification
  - Using Sampling from a Server
- Error Handling

title: MCP Server description: Learn how to implement and configure a Model Context Protocol (MCP) server —

Breaking Changes in 0.8.x ⚠

Note: Version 0.8.x introduces several breaking changes including a new session-based architecture. If you're upgrading from 0.7.0, please refer to the Migration Guide for detailed instructions.

Overview

The MCP Server is a foundational component in the Model Context Protocol (MCP) architecture that provides tools, resources, and capabilities to clients. It implements the server-side of the protocol, responsible for:

- Exposing tools that clients can discover and execute
- Managing resources with URI-based access patterns
- Providing prompt templates and handling prompt requests
- Supporting capability negotiation with clients
- Implementing server-side protocol operations
- Managing concurrent client connections
- Providing structured logging and notifications

The core `io.modelcontextprotocol.sdk:mcp` module provides STDIO and SSE server transport implementations without requiring external web frameworks.

Spring-specific transport implementations are available as an **optional** dependencies `io.modelcontextprotocol.sdk:mcp-spring-webflux`, `io.modelcontextprotocol.sdk:mcp-spring-webmvc` for Spring Framework users.

The server supports both synchronous and asynchronous APIs, allowing for flexible integration in different application contexts.

```
// Create a server with custom configuration
McpSyncServer syncServer = McpServer.sync(transportProvider)
    .serverInfo("my-server", "1.0.0")
    .capabilities(ServerCapabilities.builder()
        .resources(true)     // Enable resource support
        .tools(true)         // Enable tool support
        .prompts(true)       // Enable prompt support
        .logging()           // Enable logging support
        .build())
    .build();

// Register tools, resources, and prompts
syncServer.addTool(syncToolSpecification);
syncServer.addResource(syncResourceSpecification);
syncServer.addPrompt(syncPromptSpecification);

// Send logging notifications
syncServer.loggingNotification(LoggingMessageNotification.builder()
    .level(LoggingLevel.INFO)
```

```
    .logger("custom-logger")
    .data("Server initialized")
    .build());

// Close the server when done
syncServer.close();
```

```
// Create an async server with custom configuration
McpAsyncServer asyncServer = McpServer.async(transportProvider)
    .serverInfo("my-server", "1.0.0")
    .capabilities(ServerCapabilities.builder()
        .resources(true)     // Enable resource support
        .tools(true)         // Enable tool support
        .prompts(true)       // Enable prompt support
        .logging()           // Enable logging support
        .build())
    .build();

// Register tools, resources, and prompts
asyncServer.addTool(asyncToolSpecification)
    .doOnSuccess(v -> logger.info("Tool registered"))
    .subscribe();

asyncServer.addResource(asyncResourceSpecification)
    .doOnSuccess(v -> logger.info("Resource registered"))
    .subscribe();

asyncServer.addPrompt(asyncPromptSpecification)
    .doOnSuccess(v -> logger.info("Prompt registered"))
    .subscribe();

// Send logging notifications
asyncServer.loggingNotification(LoggingMessageNotification.builder()
    .level(LoggingLevel.INFO)
    .logger("custom-logger")
    .data("Server initialized")
    .build());

// Close the server when done
asyncServer.close()
```

```
    .doOnSuccess(v -> logger.info("Server closed"))
    .subscribe();
```

Server Transport Providers

The transport layer in the MCP SDK is responsible for handling the communication between clients and servers. It provides different implementations to support various communication protocols and patterns. The SDK includes several built-in transport provider implementations:

<> Create in-process based transport:

```
  ```java
  StdioServerTransportProvider transportProvider = new StdioServerTransportProvider(new
ObjectMapper());
  ```

  Provides bidirectional JSON-RPC message handling over standard input/output streams with
non-blocking message processing, serialization/deserialization, and graceful shutdown support.

  Key features:

    Bidirectional communication through stdin/stdout
    Process-based integration support
    Simple setup and configuration
    Lightweight implementation

</>
```

<>

Creates WebFlux-based SSE server transport.
Requires the `mcp-spring-webflux` dependency.

```
```java
@Configuration
class McpConfig {
```
```

```
    @Bean
    WebFluxSseServerTransportProvider webFluxSseServerTransportProvider(ObjectMapper mapper) {
        return new WebFluxSseServerTransportProvider(mapper, "/mcp/message");
    }

    @Bean
    RouterFunction<?> mcpRouterFunction(WebFluxSseServerTransportProvider transportProvider) {
        return transportProvider.getRouterFunction();
    }
}
```

Implements the MCP HTTP with SSE transport specification, providing:

  Reactive HTTP streaming with WebFlux
  Concurrent client connections through SSE endpoints
  Message routing and session management
  Graceful shutdown capabilities

</>

<>

Creates WebMvc-based SSE server transport.
Requires the `mcp-spring-webmvc` dependency.

```java
@Configuration
@EnableWebMvc
class McpConfig {
    @Bean
    WebMvcSseServerTransportProvider webMvcSseServerTransportProvider(ObjectMapper mapper) {
        return new WebMvcSseServerTransportProvider(mapper, "/mcp/message");
    }

    @Bean
    RouterFunction<ServerResponse> mcpRouterFunction(WebMvcSseServerTransportProvider
transportProvider) {
```

````
        return transportProvider.getRouterFunction();
    }
}
```

<p>Implements the MCP HTTP with SSE transport specification, providing:</p>
<ul>
  <li>Server-side event streaming</li>
  <li>Integration with Spring WebMVC</li>
  <li>Support for traditional web applications</li>
  <li>Synchronous operation handling</li>
</ul>
````

</>

<>

Creates a Servlet-based SSE server transport. It is included in the core `mcp` module.
The `HttpServletSseServerTransport` can be used with any Servlet container.
To use it with a Spring Web application, you can register it as a Servlet bean:

````
```java
@Configuration
@EnableWebMvc
public class McpServerConfig implements WebMvcConfigurer {

    @Bean
    public HttpServletSseServerTransportProvider servletSseServerTransportProvider() {
        return new HttpServletSseServerTransportProvider(new ObjectMapper(), "/mcp/message");
    }

    @Bean
    public ServletRegistrationBean customServletBean(HttpServletSseServerTransportProvider
transportProvider) {
        return new ServletRegistrationBean(transportProvider);
    }
}
```
````
````

```
<p>
  Implements the MCP HTTP with SSE transport specification using the traditional Servlet API,
providing:
</p>
<ul>
  <li>Asynchronous message handling using Servlet 6.0 async support</li>
  <li>Session management for multiple client connections</li>
  <li>
    Two types of endpoints:
    <ul>
      <li>SSE endpoint (<code>/sse</code>) for server-to-client events</li>
      <li>Message endpoint (configurable) for client-to-server requests</li>
    </ul>
  </li>
  <li>Error handling and response formatting</li>
  <li>Graceful shutdown support</li>
</ul>
```

</>

Server Capabilities

The server can be configured with various capabilities:

```
var capabilities = ServerCapabilities.builder()
    .resources(false, true)  // Resource support with list changes notifications
    .tools(true)            // Tool support with list changes notifications
    .prompts(true)          // Prompt support with list changes notifications
    .logging()              // Enable logging support (enabled by default with logging level
INFO)
    .build();
```

Logging Support

The server provides structured logging capabilities that allow sending log messages to clients with different severity levels:

```
// Send a log message to clients
server.loggingNotification(LoggingMessageNotification.builder()
    .level(LoggingLevel.INFO)
    .logger("custom-logger")
    .data("Custom log message")
    .build());
```

Clients can control the minimum logging level they receive through the `mcpClient.setLoggingLevel(level)` request. Messages below the set level will be filtered out. Supported logging levels (in order of increasing severity): DEBUG (0), INFO (1), NOTICE (2), WARNING (3), ERROR (4), CRITICAL (5), ALERT (6), EMERGENCY (7)

Tool Specification

The Model Context Protocol allows servers to expose tools that can be invoked by language models. The Java SDK allows implementing a Tool Specifications with their handler functions. Tools enable AI models to perform calculations, access external APIs, query databases, and manipulate files:

```
// Sync tool specification
var schema = """
            {
              "type" : "object",
              "id" : "urn:jsonschema:Operation",
              "properties" : {
                "operation" : {
                  "type" : "string"
                },
                "a" : {
                  "type" : "number"
                },
                "b" : {
                  "type" : "number"
                }
              }
            }
            """;
var syncToolSpecification = new McpServerFeatures.SyncToolSpecification(
```

```
    new Tool("calculator", "Basic calculator", schema),
    (exchange, arguments) -> {
        // Tool implementation
        return new CallToolResult(result, false);
    }
);
```

```
// Async tool specification
var schema = """
            {
              "type" : "object",
              "id" : "urn:jsonschema:Operation",
              "properties" : {
                "operation" : {
                  "type" : "string"
                },
                "a" : {
                  "type" : "number"
                },
                "b" : {
                  "type" : "number"
                }
              }
            }
            """;
var asyncToolSpecification = new McpServerFeatures.AsyncToolSpecification(
    new Tool("calculator", "Basic calculator", schema),
    (exchange, arguments) -> {
        // Tool implementation
        return Mono.just(new CallToolResult(result, false));
    }
);
```

The Tool specification includes a Tool definition with `name`, `description`, and `parameter schema` followed by a call handler that implements the tool's logic. The function's first argument is `McpAsyncServerExchange` for client interaction, and the second is a map of tool arguments.

Resource Specification

Specification of a resource with its handler function. Resources provide context to AI models by exposing data such as: File contents, Database records, API responses, System information, Application state. Example resource specification:

```
// Sync resource specification
var syncResourceSpecification = new McpServerFeatures.syncResourceSpecification(
    new Resource("custom://resource", "name", "description", "mime-type", null),
    (exchange, request) -> {
        // Resource read implementation
        return new ReadResourceResult(contents);
    }
);
```

```
// Async resource specification
var asyncResourceSpecification = new McpServerFeatures.asyncResourceSpecification(
    new Resource("custom://resource", "name", "description", "mime-type", null),
    (exchange, request) -> {
        // Resource read implementation
        return Mono.just(new ReadResourceResult(contents));
    }
);
```

The resource specification comprised of resource definitions and resource read handler. The resource definition including `name`, `description`, and `MIME type`. The first argument of the function that handles resource read requests is an `McpAsyncServerExchange` upon which the server can interact with the connected client. The second arguments is a `McpSchema.ReadResourceRequest`.

Prompt Specification

As part of the Prompting capabilities, MCP provides a standardized way for servers to expose prompt templates to clients. The Prompt Specification is a structured template for AI model interactions that enables consistent message formatting, parameter substitution, context injection, response formatting, and instruction templating.

```
// Sync prompt specification
var syncPromptSpecification = new McpServerFeatures.syncPromptSpecification(
    new Prompt("greeting", "description", List.of(
        new PromptArgument("name", "description", true)
    )),
    (exchange, request) -> {
        // Prompt implementation
        return new GetPromptResult(description, messages);
    }
);
```

```
// Async prompt specification
var asyncPromptSpecification = new McpServerFeatures.asyncPromptSpecification(
    new Prompt("greeting", "description", List.of(
        new PromptArgument("name", "description", true)
    )),
    (exchange, request) -> {
        // Prompt implementation
        return Mono.just(new GetPromptResult(description, messages));
    }
);
```

The prompt definition includes name (identifier for the prompt), description (purpose of the prompt), and list of arguments (parameters for templating). The handler function processes requests and returns formatted templates. The first argument is `McpAsyncServerExchange` for client interaction, and the second argument is a `GetPromptRequest` instance.

Using Sampling from a Server

To use Sampling capabilities, connect to a client that supports sampling. No special server configuration is needed, but verify client sampling support before making requests. Learn about client sampling support.

Once connected to a compatible client, the server can request language model generations:

```
// Create a server
McpSyncServer server = McpServer.sync(transportProvider)
    .serverInfo("my-server", "1.0.0")
    .build();

// Define a tool that uses sampling
var calculatorTool = new McpServerFeatures.SyncToolSpecification(
    new Tool("ai-calculator", "Performs calculations using AI", schema),
    (exchange, arguments) -> {
        // Check if client supports sampling
        if (exchange.getClientCapabilities().sampling() == null) {
            return new CallToolResult("Client does not support AI capabilities", false);
        }

        // Create a sampling request
        McpSchema.CreateMessageRequest request = McpSchema.CreateMessageRequest.builder()
            .messages(List.of(new McpSchema.SamplingMessage(McpSchema.Role.USER,
                new McpSchema.TextContent("Calculate: " + arguments.get("expression"))))
            .modelPreferences(McpSchema.ModelPreferences.builder()
                .hints(List.of(
                    McpSchema.ModelHint.of("claude-3-sonnet"),
                    McpSchema.ModelHint.of("claude")
                ))
                .intelligencePriority(0.8)  // Prioritize intelligence
                .speedPriority(0.5)         // Moderate speed importance
                .build())
            .systemPrompt("You are a helpful calculator assistant. Provide only the numerical
answer.")
            .maxTokens(100)
            .build();

        // Request sampling from the client
        McpSchema.CreateMessageResult result = exchange.createMessage(request);

        // Process the result
        String answer = result.content().text();
        return new CallToolResult(answer, false);
    }
);
```

```
// Add the tool to the server
server.addTool(calculatorTool);
```

```
// Create a server
McpAsyncServer server = McpServer.async(transportProvider)
    .serverInfo("my-server", "1.0.0")
    .build();

// Define a tool that uses sampling
var calculatorTool = new McpServerFeatures.AsyncToolSpecification(
    new Tool("ai-calculator", "Performs calculations using AI", schema),
    (exchange, arguments) -> {
        // Check if client supports sampling
        if (exchange.getClientCapabilities().sampling() == null) {
            return Mono.just(new CallToolResult("Client does not support AI capabilities",
false));
        }

        // Create a sampling request
        McpSchema.CreateMessageRequest request = McpSchema.CreateMessageRequest.builder()
            .content(new McpSchema.TextContent("Calculate: " + arguments.get("expression")))
            .modelPreferences(McpSchema.ModelPreferences.builder()
                .hints(List.of(
                    McpSchema.ModelHint.of("claude-3-sonnet"),
                    McpSchema.ModelHint.of("claude")
                ))
                .intelligencePriority(0.8)  // Prioritize intelligence
                .speedPriority(0.5)         // Moderate speed importance
                .build())
            .systemPrompt("You are a helpful calculator assistant. Provide only the numerical
answer.")
            .maxTokens(100)
            .build();

        // Request sampling from the client
        return exchange.createMessage(request)
            .map(result -> {
                // Process the result
                String answer = result.content().text();
                return new CallToolResult(answer, false);
```

```
            });
    }
);

// Add the tool to the server
server.addTool(calculatorTool)
    .subscribe();
```

The `CreateMessageRequest` object allows you to specify: `Content` - the input text or image for the model, `Model Preferences` - hints and priorities for model selection, `System Prompt` - instructions for the model's behavior and `Max Tokens` - maximum length of the generated response.

Error Handling

The SDK provides comprehensive error handling through the McpError class, covering protocol compatibility, transport communication, JSON-RPC messaging, tool execution, resource management, prompt handling, timeouts, and connection issues. This unified error handling approach ensures consistent and reliable error management across both synchronous and asynchronous operations.

□□□ #2
□ □□□ □□□ 2026-10-31 13:09:14 UTC
□ □□□ □□□ 2026-10-31 13:09:15 UTC