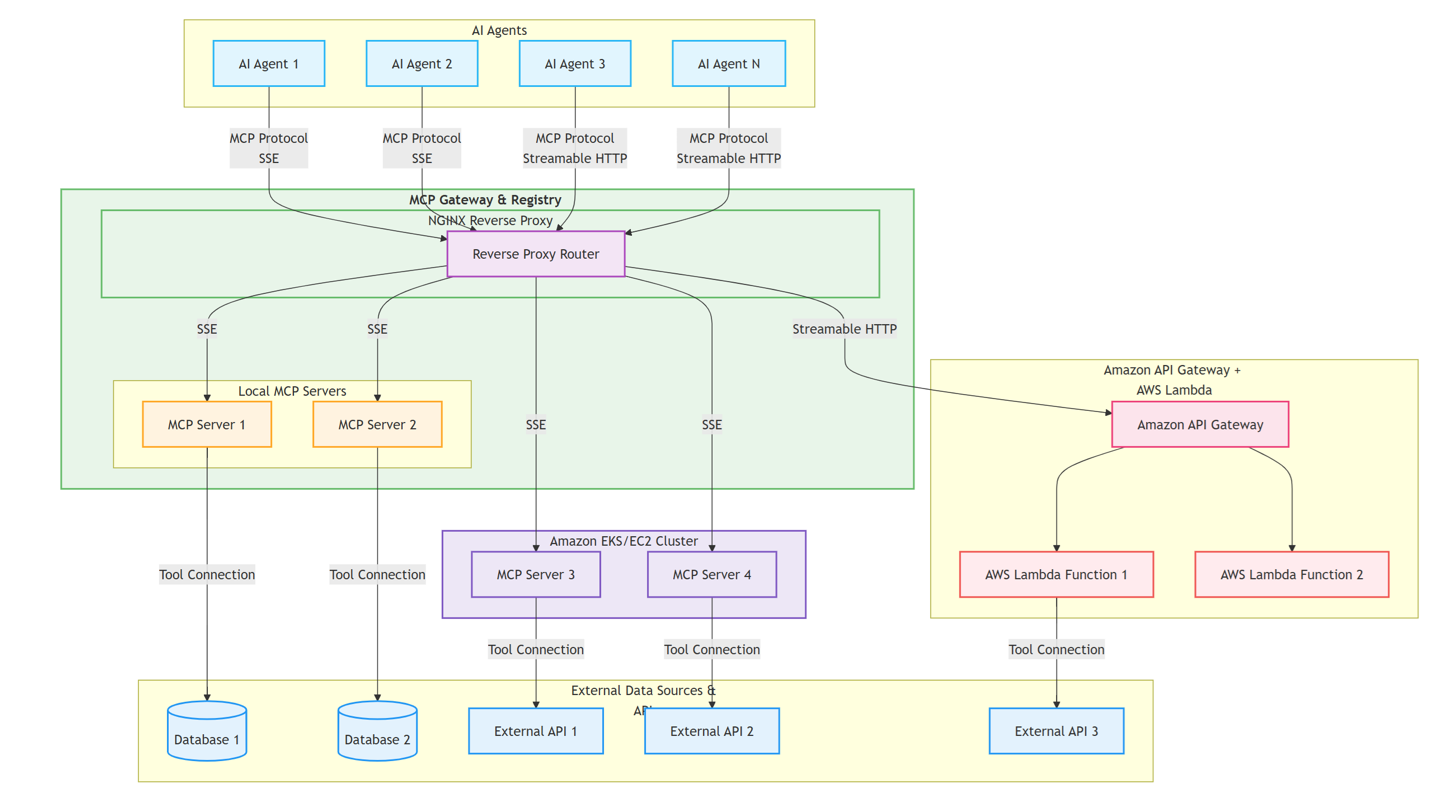
MCP Gateway and Registry Code Walkthrough

compiled by D. Gueorguiev, 9/10/2025



# References

[1] <https://github.com/agentic-community/mcp-gateway-registry>

[2]

[] <https://openid.net/developers/how-connect-works/>

[] <https://aws.amazon.com/cognito/>

[] <https://mcp-framework.com/docs/Transports/sse/>

[] <https://mcp-framework.com/docs/Transports/http-stream-transport>

[] <https://mcp-framework.com/docs/Authentication/overview>

# Appendix

## A1. OpenID Connect (OIDC)

OIDC stands for OpenID Connect, a modern, lightweight identity layer built on top of the OAuth 2.0 framework that enables secure, standardized user authentication and single sign-on (SSO) across different applications. It allows users to log in to one service and then access other services without re-entering their credentials. OIDC works by having an "*OpenID Provider*" verify user identity and then send an "*ID token*" containing user information (like their name) to the "*Relying Party*" (the application the user wants to access).

How OIDC Works

1. **User Initiates Login**:

A user visits an application (the *Relying Party*) to access a service and starts the login process.

2. **Request to OpenID Provider**:

The *Relying Party* redirects the user to an *OpenID Provider* (e.g., Google or Microsoft) to verify their identity.

3. **User Authentication**:

The OpenID Provider authenticates the user, often by asking for their username and password.

4. **ID Token Issuance**:

After successful authentication, the *OpenID Provider* issues an "*ID token*" to the *Relying Party*.

5. **User Information and Access**:

The *ID token* contains information about the authenticated user. The *Relying Party* uses this information to grant the user access to the application and personalize the user experience.

Key Components

**OpenID Provider (OP)**:

The server that authenticates the user and issues ID tokens.

**Relying Party (RP)**:

The application that requests user authentication and receives the ID token.

**ID Token**:

A JSON Web Token (JWT) that contains claims (information) about the authenticated user.

**User Info Endpoint**:

A dedicated endpoint where the Relying Party can retrieve more user information.

Benefits of OIDC

**Enhanced User Experience**:

Provides single sign-on, allowing users to log in once and access multiple services without repeated logins.

**Increased Security**:

Centralizes authentication, moving the responsibility for user verification to trusted expert providers and reducing the number of stored passwords.

**Simpler Integration**:

Built on OAuth 2.0, it uses modern technologies like JSON Web Tokens (JWT) and REST APIs, making it easier to integrate into applications, especially JavaScript-based ones.

## A2. AWS Cognito

AWS Cognito vs Secrets Manager

AWS Cognito is for user identity management in web and mobile apps, handling authentication, authorization, and user directories, while AWS Secrets Manager is for securely storing, managing, and rotating application secrets like API keys, database credentials, and passwords. Cognito manages who users are and what they can access, whereas Secrets Manager manages the sensitive information your applications need to function.

AWS Cognito

Purpose:

Manages user identity for B2C and B2B applications, providing user sign-up, sign-in, and access control.

Key Features:

Integrates with social identity providers (Facebook, Google) and supports SAML and OIDC for federated identity.

Provides customizable user interfaces and Multi-Factor Authentication (MFA).

Generates tokens to grant access to backend resources and other AWS services.

Handles user directories and data synchronization across devices.

Use Cases:

Allowing users to log in to your application using their social media accounts.

Securing access to your APIs and web or mobile app's backend resources.

AWS Secrets Manager

Purpose:

Centralizes the storage, management, and rotation of sensitive information used by your applications and AWS services.

Key Features:

Secures and stores secrets like database credentials, API keys, and OAuth tokens.

Automates the rotation of these secrets at predefined intervals.

Allows fine-grained access control using IAM policies.

Provides auditing and monitoring of secrets usage with AWS CloudTrail.

Use Cases:

Storing database connection strings and user credentials.

Managing API keys for third-party services.

Securing private keys for client assertions in authentication flows.

Key Differences

What They Protect:

Cognito protects user identities and their access to applications, while Secrets Manager protects sensitive credentials and tokens that applications use to access other services.

Who Uses Them:

Cognito is for the end-users of your application to sign in. Secrets Manager is for your application code and infrastructure to securely access sensitive data.

How They Work Together:

You can use Secrets Manager to store a private key that a Lambda function uses to authenticate with Cognito, or to rotate the password for a user in a Cognito user pool.

## A3. SSE Transport

**DEPRECATED**: The SSE Transport has been deprecated as of MCP specification version 2025-03-26. Please use the [HTTP Stream Transport](https://mcp-framework.com/docs/Transports/http-stream-transport) instead, which implements the new Streamable HTTP transport specification.

The Server-Sent Events (SSE) transport enables HTTP-based communication between the MCP server and clients. It uses SSE for server-to-client messages and HTTP POST for client-to-server messages.

## Configuration

The SSE transport supports various configuration options to customize its behavior:

import { MCPServer } from "@modelcontextprotocol/mcp-framework";  
  
const server = new MCPServer({  
 transport: {  
 type: "sse",  
 options: {  
 port: 8080, *// Port to listen on (default: 8080)*  
 endpoint: "/sse", *// SSE endpoint path (default: "/sse")*  
 messageEndpoint: "/messages", *// Message endpoint path (default: "/messages")*  
 maxMessageSize: "4mb", *// Maximum message size (default: "4mb")*  
 headers: { *// Custom headers for SSE responses*  
 "X-Custom-Header": "value"  
 },  
 cors: { *// CORS configuration*  
 allowOrigin: "\*",  
 allowMethods: "GET, POST, OPTIONS",  
 allowHeaders: "Content-Type, Authorization, x-api-key",  
 exposeHeaders: "Content-Type, Authorization, x-api-key",  
 maxAge: "86400"  
 },  
 auth: { *// Authentication configuration*  
 provider: authProvider,  
 endpoints: {  
 sse: true, *// Require auth for SSE connections*  
 messages: true *// Require auth for messages*  
 }  
 }  
 }  
 }  
});

**Port Configuration**

The port option specifies which port the SSE server should listen on. Default is 8080.

**Endpoints**

endpoint: The path for the SSE connection endpoint (default: "/sse")

messageEndpoint: The path for receiving messages via POST (default: "/messages")

**Message Size Limit**

The maxMessageSize option controls the maximum allowed size for incoming messages. Accepts string values like "4mb", "1kb", etc.

**Custom Headers**

You can specify custom headers to be included in SSE responses:

headers: {

"X-Custom-Header": "value",

"Cache-Control": "no-cache"

}

CORS Configuration

The SSE transport includes comprehensive CORS support with the following options:

cors: {

allowOrigin: "\*", // Access-Control-Allow-Origin

allowMethods: "GET, POST, OPTIONS", // Access-Control-Allow-Methods

allowHeaders: "Content-Type, Authorization, x-api-key", // Access-Control-Allow-Headers

exposeHeaders: "Content-Type, Authorization, x-api-key", // Access-Control-Expose-Headers

maxAge: "86400" // Access-Control-Max-Age

}

Authentication

The SSE transport supports authentication through various providers. See the [Authentication](https://mcp-framework.com/docs/Authentication/overview) documentation for details.

//TODO: finish the section on SSE transport

## A4. Http Stream Transport

//TODO: finish the section on Http Stream transport

## A5. MCP Authentication

The MCP Framework provides built-in authentication support through various authentication providers. This allows you to secure your MCP server endpoints and ensure only authorized clients can access your tools and resources.

Available Authentication Providers

**API Key Authentication**

The API Key authentication provider allows you to secure your endpoints using API keys. This is useful for simple authentication scenarios where you want to control access using predefined keys.

import { APIKeyAuthProvider } from "@modelcontextprotocol/mcp-framework";

const authProvider = new APIKeyAuthProvider({

keys: ["your-api-key-1", "your-api-key-2"],

headerName: "X-API-Key" // Optional, defaults to "X-API-Key"

});

Clients must include the API key in the specified header:

X-API-Key: your-api-key-1

**JWT Authentication**

The JWT authentication provider enables token-based authentication using JSON Web Tokens. This is suitable for more complex authentication scenarios where you need to include user information or other claims in the token.

import { JWTAuthProvider } from "@modelcontextprotocol/mcp-framework";

const authProvider = new JWTAuthProvider({

secret: "your-jwt-secret",

algorithms: ["HS256"], // Optional, defaults to ["HS256"]

headerName: "Authorization", // Optional, defaults to "Authorization"

requireBearer: true // Optional, defaults to true

});

Clients must include the JWT token in the Authorization header:

Authorization: Bearer eyJhbGciOiJIUzI1NiIs...

**Configuring Authentication**

You can configure authentication when setting up your SSE transport:

import { MCPServer, APIKeyAuthProvider } from "@modelcontextprotocol/mcp-framework";

const server = new MCPServer({

transport: {

type: "sse",

options: {

auth: {

provider: new APIKeyAuthProvider({

keys: ["your-api-key"]

}),

endpoints: {

sse: true, // Require auth for SSE connections

messages: true // Require auth for messages

}

}

}

}

});

**Endpoint Configuration**

The endpoints configuration allows you to specify which endpoints require authentication:

sse: Controls authentication for the SSE connection endpoint

Default: false

messages: Controls authentication for the message endpoint

Default: true

Error Handling

Authentication providers include built-in error handling that returns appropriate HTTP status codes and error messages:

// Example error response for invalid API key

{

"error": "Invalid API key",

"status": 401,

"type": "authentication\_error"

}

// Example error response for invalid JWT

{

"error": "Invalid or expired JWT token",

"status": 401,

"type": "authentication\_error"

}

//TODO: finish the section on MCP Authentication