This is the second MCP-related blog post that is part of a series covering how to use Semantic Kernel (SK) with the Model Context Protocol (MCP).

This blog post demonstrates how to build an MCP server using https://github.com/modelcontextprotocol/csharp-sdk and SK, expose SK plugins as MCP tools, and call the tools from the client side via SK.

Here are a few reasons why you might want to build an MCP server with SK:

- **Interoperability**: Existing SK plugins need to be reused and exposed as MCP tools so they can be consumed by non-SK applications or by SK for a different platform.
- **Content safety**: Each tool call needs to be validated before it is executed using https://learn.microsoft.com/en-us/semantic-kernel/concepts/enterprise-readiness/filters?pivots=programming-language-csharp.
- **Observability**: Tool call-related logs, traces, and metrics need to be collected (see: https://learn.microsoft.com/en-us/semantic-kernel/concepts/enterprise-readiness/observability/?pivots=programming-language-csharp) and saved to an existing monitoring infrastructure.

For more information about MCP, please refer to the https://modelcontextprotocol.io/introduction.

The sample described below uses the official https://www.nuget.org/packages/mcpdotnet nuget package. Its runnable source code is available in the https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/Demos/ModelContextProtocolClientServer.

#### Build MCP Server and Expose SK Plugins as MCP Tools

Let’s start by declaring SK plugins that will be exposed as MCP tools by the server in the next step:

```cs
internal sealed class DateTimeUtils
{
    [KernelFunction, Description("Retrieves the current date time in UTC.")]
    public static string GetCurrentDateTimeInUtc()
    {
        return DateTime.UtcNow.ToString("yyyy-MM-dd HH:mm:ss");
    }
}

internal sealed class WeatherUtils
{
    [KernelFunction, Description("Gets the current weather for the specified city and specified date time.")]
    public static string GetWeatherForCity(string cityName, string currentDateTimeInUtc)
    {
        return cityName switch
        {
            "Boston" => "61 and rainy",
            "London" => "55 and cloudy",
            ...
        };
    }
}
```

Now we will create an MCP server and import the SK plugins:

```cs
// Create a kernel builder and add plugins
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.Plugins.AddFromType<DateTimeUtils>();
kernelBuilder.Plugins.AddFromType<WeatherUtils>();

// Build the kernel
Kernel kernel = kernelBuilder.Build();

var builder = Host.CreateEmptyApplicationBuilder(settings: null);
builder.Services
   .AddMcpServer()
   .WithStdioServerTransport()
   // Add all functions from the kernel plugins to the MCP server as tools
   .WithTools(kernel);
await builder.Build().RunAsync();
```

The code above creates a kernel and imports SK plugins. Then it creates an MCP server and registers all functions from all SK plugins as MCP tools using the WithTools extension method:

```cs
public static IMcpServerBuilder WithTools(this IMcpServerBuilder builder, KernelPluginCollection plugins)
{
   foreach (var plugin in plugins)
   {
      foreach (var function in plugin)
```

```
        {
            builder.Services.AddSingleton(services =>
McpServerTool.Create(function.AsAIFunction()));
        }
    }

    return builder;
}
```

The code can be further extended to register function invocation filters to intercept function calls and validate them before execution in addition to enabling observability to collect tool call-related logs, traces, and metrics.

#### Build MCP Client and Use MCP tools in SK

Next, we will create an MCP client:

```cs
await using IMcpClient mcpClient = await McpClientFactory.CreateAsync(new
StdioClientTransport(new()
{
    Name = "MCPServer",
    // Point the client to the MCPServer server executable
    Command = Path.Combine("..", "..", "..", "..", "MCPServer", "bin", "Debug", "net8.0",
"MCPServer.exe")
}));
```

Then we get MCP tools from the server, import them to SK, add OpenAI chat completion service, and build the kernel:

```cs
IList<AIFunction> tools = await mcpClient.ListToolsAsync();

IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.Plugins.AddFromFunctions("Tools", tools.Select(aiFunction => aiFunction.AsKernelFunction()));
kernelBuilder.Services.AddOpenAIChatCompletion(serviceId: "openai", modelId: config["OpenAI:ChatModelId"] ?? "gpt-4o-mini", apiKey: apiKey);

Kernel kernel = kernelBuilder.Build();
```

The code above iterates over the MCP tools represented by AI functions, converts them to SK functions, and adds them to the kernel so they can be used by AI models.

Now, we can prompt the AI model and the AI model will automatically call the imported kernel functions which will delegate the calls to the MCP tools to answer the prompt:

```cs
// Enable automatic function calling
OpenAIPromptExecutionSettings executionSettings = new()
{
   Temperature = 0,
   FunctionChoiceBehavior = FunctionChoiceBehavior.Auto(options: new() { RetainArgumentTypes = true })
};
```

```
// Execute a prompt using the MCP tools. The AI model will automatically call the appropriate MCP tools to answer the prompt.

var prompt = "What is the likely color of the sky in Boston today?";

var result = await kernel.InvokePromptAsync(prompt, new(executionSettings));

Console.WriteLine(result);
```

After calling the MCP tools, the AI model will return the following result:

```default
The likely color of the sky in Boston today is gray, as it is currently rainy.
```

#### What’s Next?

We recommend trying out the sample code in the https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/Demos/ModelContextProtocolClientServer to get started.

Next, we plan to create more samples to demonstrate how to use Semantic Kernel for building MCP https://modelcontextprotocol.io/docs/concepts/prompts and https://modelcontextprotocol.io/docs/concepts/resources on the server side, and how to use them with the SK client side.

Category

https://devblogs.microsoft.com/semantic-kernel/category/net/
https://devblogs.microsoft.com/semantic-kernel/category/samples/
https://devblogs.microsoft.com/semantic-kernel/category/semantic-kernel/

Topics

https://devblogs.microsoft.com/semantic-kernel/tag/net/
https://devblogs.microsoft.com/semantic-kernel/tag/ai/
https://devblogs.microsoft.com/semantic-kernel/tag/mcp/
https://devblogs.microsoft.com/semantic-kernel/tag/semantic-kernel/

## 3 comments

Join the discussion.

### javascript:void(0 "Leave a comment") https://devblogs.microsoft.com/semantic-kernel/building-a-model-context-protocol-server-with-semantic-kernel/\#respond

https://devblogs.microsoft.com/semantic-kernel/wp-login.php?redirect_to=https%3A%2F%2Fdevblogs.microsoft.com%2Fsemantic-kernel%2Fbuilding-a-model-context-protocol-server-with-semantic-kernel%2F%23comments

https://answers.microsoft.com/en-us/page/codeofconduct

Sort by :

Newest

javascript:void(0) javascript:void(0) javascript:void(0)

- https://devblogs.microsoft.com/semantic-kernel/wp-content/uploads/sites/78/letter-avatar/ea15b6c68b667e156e8c02f6c1af17a2.svg

Is there a better way to do this?

```cs
["command"] = Path.Combine("..", "..", "..", "..", "MCPServer", "bin", "Debug", "net8.0", "MCPServer.exe");
```

- https://devblogs.microsoft.com/semantic-kernel/wp-content/uploads/sites/78/2023/12/9e589fc0-853d-43d4-9ebd-7de4e5642d0a.jpeg

Yes, there’s a slightly better way. In the SK sample, you can find a version that dynamically selects between Debug and Release configurations. For production, the server executable might be copied to the client output folder by the release pipeline. In that case, you can just use \[“command”\] = “MCPServer.exe”; for the release mode.

- https://devblogs.microsoft.com/semantic-kernel/wp-content/uploads/sites/78/letter-avatar/ff81ae8c89d8cccf50932a57c1302350.svg