Clipcode Mentoring
https://clipcode.net/mentoring

Semantic Kernel Source Tour

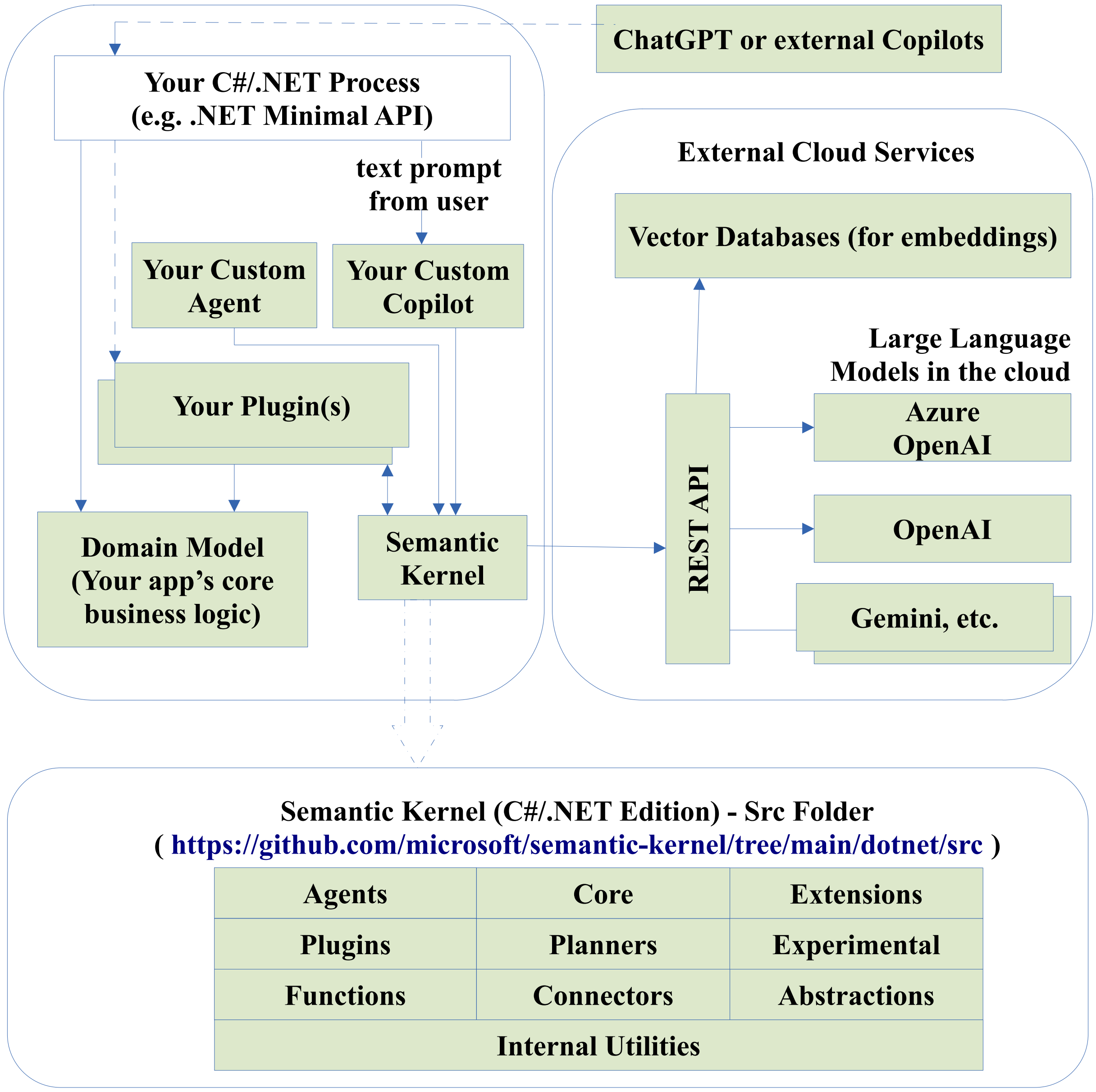
Guided tour of Microsoft's Semantic Kernel source code

The key framework for C#/.NET developers to orchestrate advanced AI solutions, including connecting to external AI services such as [Azure] OpenAI

Written By Eamon O'Tuathail

Last Updated: 2024-09-26

Table of Contents


ChatGPT or external Copilots
Your C#/.NET Process
(e.g. .NET Minimal API)
text prompt
from user
Your Custom
Agent
Your Custom
Copilot
Your Plugin(s)
Domain Model
(Your app's core
business logic)
Semantic
Kernel
External Cloud Services
Vector Databases (for embeddings)
Large Language
Models in the cloud
REST API
Azure
OpenAI
OpenAI
Gemini, etc.
Semantic Kernel (C#/.NET Edition) - Src Folder
(https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src)
Agents
Core
Extensions
Plugins
Planners
Experimental
Functions
Connectors
Abstractions
Internal Utilities

1: Introduction

Overview

Microsoft's Semantic Kernel is the key SDK for app developers to use when accessing Large Language Models and ancillary AI services, such as vector databases. App developers could of course directly use the REST API that OpenAI and the other AI cloud service providers expose, but app developers will quickly discover there is quite a bit of programming needed in order to develop robust solutions. Using a well-designed framework such as Semantic Kernel greatly reduces the effort and time-to-market. In addition to providing apps with a uniform programming surface against which to connect to a variety of AI cloud services, Semantic Kernel also provides substantial layered functionality, in areas such as orchestration of multiple AI capabilities via planners.

We like to think that Semantic Kernel is composed of two layers – orchestration and connectors. Orchestration is the complex arrangement of all the resources needed to provide AI capabilities to end-users. This is built around the idea of a dependency injection container (called "kernel" in Semantic Kernel), that .NET/C# application developers will be familiar with already, through the use of service collections. A good analogy for the connector layer is that it does for AI app programming what ODBC does for database app programming – each provides a family of DLLs that live within your own app's .NET process and enables connections to the outside world. Different DLLs are needed for different targets, but as much as possible they offer a consistent API to the calling code (so e.g. if you wish to change the LLM in a year's time, that should be easy to accomplish without having to re-write large chunks of your application).

In this document we give a guided tour of its source tree – reading the code is highly useful in order to gain deeper insight into how it really works. Semantic Kernel comes in multiple languages - .NET / Python / Java. Here we are going to focus on the .NET version, written in C#.

Mentoring

Note: The author of this guided tour provides specialist mentoring for enterprise dev teams embarking on Semantic Kernel projects. Learn more here:

- https://clipcode.net/mentoring

Here is an example of what is included – a recommended solution architecture for larger enterprise applications wishing to incorporate Semantic Kernel support:

- https://clipcode.net/assets/academy/mentoring/semantic-kernel-solution-architecture.pdf

MIT Licensing of Semantic Kernel Source

Source extracts from Semantic Kernel repo are (c) Microsoft and provided via MIT open source licensing:

- https://github.com/microsoft/semantic-kernel/blob/main/LICENSE

Family of Repos

You can view Semantic Kernel as a family of repos on GitHub.

There is the main one (which we are going to explore in great detail) and then some additional ones with useful functionality/content. This is a list of them (with size of download ZIP) in suggested order that a typical experienced C#/.NET developer (who is however new to Semantic Kernel) might consider exploring them:

Semantic Kernel (8 MB) is the main repo
https://github.com/microsoft/semantic-kernel

Semantic Kernel Starters (0.3 MB) are simple examples showing individual pieces of Semantic Kernel in action
https://github.com/microsoft/semantic-kernel-starters

Chat Copilot (2 MB) is a larger reference application showing how the different Semantic Kernel features and other libraries/tooling (e.g. Azure) can be used together for a larger solution
https://github.com/microsoft/chat-copilot

Kernel Memory (23MB) produces both a service and a plugin and focuses in indexing datasets for user with LLMs:
https://github.com/microsoft/kernel-memory

Kernel Memory Extension Example (0.07MB) shows you how to create custom adapters, by creating one for Postgres:
https://github.com/microsoft/kernel-memory-extension-example

Project Miyagi (33 MB) explores next-generation app architecture that incorporate copilots and LLMs:
https://github.com/Azure-Samples/miyagi

Semantic Kernel Docs (23 MB) is the documentation site – importantly it contains the code samples in the documentation as actual source projects:
https://github.com/MicrosoftDocs/semantic-kernel-docs

Documentation

The main documentation site for Semantic Kernel is here:
https://learn.microsoft.com/en-us/semantic-kernel/overview/

The official Semantic Kernel blog is here:
https://devblogs.microsoft.com/semantic-kernel/

Note that Semantic Kernel is a new and a fast-changing topic, so often details are on the blog before the documentation site is updated. The most up to date documentation is the actual source code, as always, hence the reason everyone keenly interested in Semantic Kernel should read the source.

A few Microsoft folks working on Semantic Kernel provide very relevant updates from time to time. Also an external developer community is building up around Semantic Kernel and they provide some very interesting resources. Let's mention a few.

Alex Chao has an excellent Youtube channel:
https://www.youtube.com/@alexchaomander/videos
which also features other members of the Semantic Kernel team, such as Matthew Bolaños and Devis Lucato, who have some excellent presentations.

Alex Chao is also active on LinkedIn:
https://www.linkedin.com/in/alexchao56/

Matteo Pagani has this blog with insightful Semantic Kernel content:

- https://www.developerscantina.com/

John Maeda on Twitter is good:

- https://twitter.com/johnmaeda

There is also plenty of activity in the wider community of developers using Semantic Kernel – e.g. Jose Luis Latorre provides excellent training on LinkedIn Learning:

- https://www.linkedin.com/learning/instructors/jose-latorre

Contacting the Semantic Kernel Team

This blog post:

- https://devblogs.microsoft.com/semantic-kernel/new-year-new-ways-to-connect-with-the-semantic-kernel-team/

has details of the regular community office hours and kernel development syncs that the Semantic Kernel team hosts to connect with the wider Semantic Kernel community.

This page outlines support options:

- https://learn.microsoft.com/en-us/semantic-kernel/support/

Note there is a popular Discord channel for Semantic Kernel:

- https://discord.com/invite/VpnfAZkv2a

Terminology

Here are the main terms used in Semantic Kernel and OpenAI:

Action – How you connect a GPT to a custom API defined with OpenAI; with your OpenAPI specification for your API hosted by OpenAI.

Agent – A range of AI solutions, from simple chat bots, to Copilots to more advanced autonomous agents. You use Semantic Kernel to build agents. Though the term "Copilot" has initially got most attention, be well aware that there is a spectrum of agent types and you should choose appropriately depending on your customers' needs

AI Management System – See ISO/IEC 42001:2023

Assistant – A cloud-hosted way of more robustly interacting with a LLM. Works with multiple tools; importantly offers threads (see definition) so chat history is remembered. As currently envisaged, assistants are a developer feature – whether your copilot is written with or without the use of Assistants will not be visible to end users. For more details, see:
https://platform.openai.com/docs/assistants/how-it-works

Autonomous Agent – The most capable type of agent – expect to operate with little or no human direction. see diagram here:
https://learn.microsoft.com/en-us/semantic-kernel/agents/

Azure AI Search – Microsoft's vector database in the cloud for AI search (and has additional capabilities).

Azure AI Studio – Workbench to work with AI models and prompt – all users of Semantic Kernel should become familiar with Azure AI Search.

Azure OpenAI Service – This is the key Microsoft cloud service hosting the LLM on Azure
(also see OpenAI)

Chatbot -The simplest type of agent, just answers questions the user submits.

Code Interpreter – Allows Assistant in the OpenAI cloud run Python code and inspect your uploaded files (e.g. CSV) and give insight
See also: https://www.nytimes.com/2023/07/11/technology/what-to-know-chatgpt-code-interpreter.html

Copilot - A user interface to a LLM that integrates with an app. A key target audience for developers using Semantic Kernel is to create their own copilots inside their own C#/.NET apps.

Embedding – Numerical representation of data in vector space. See:
https://openai.com/blog/new-and-improved-embedding-model
Here is a nice introduction to embeddings:

https://stackoverflow.blog/2023/11/09/an-intuitive-introduction-to-text-embeddings

Fine-tuning - allows you to customise a model with your own data (do this once or infrequently; and then use that from multiple prompts, without having to feed in the same embeddings (so reduces token use, specifically if you have a slow-changing or fixed dataset [because there is a cost to setting up fine-tuning, so the savings must outweigh the cost]). It is usually best to start without fine-tuning and later, use if needed.

Function – Endpoints defined in the OpenAPI (OpenAPI is not to be confused with OpenAI). (also see Native Function, Prompt Function)

Function Calling – The calling of external APIs from the LLM

GPT (Generative Pre-trained Transformer) – a custom version of ChatGPT, enhanced with your plugins.

Instruction (or custom instruction) – Used to tailor how the agent reacts.
See also: https://openai.com/blog/custom-instructions-for-chatgpt

ISO/IEC 42001:2023 - Artificial Intelligence Management System from OSI; a comprehensive enterprise-grade management system for the use of AI within well-managed enterprises; (similar in purpose to 27001 for information security);
See:
https://www.iso.org/artificial-intelligence/ai-management-systems
Also see:
https://www.iso.org/standard/81230.html

Kernel Memory - A service and a plugin specifically designed for indexing your databases (usually using vector databases; for non-trivial production environments, it is likely the service rather than the plugin will be used)

KNN – See K-nearest neighbors

K-nearest neighbors (KNN) – exhaustive KNN is a computationally intense yet highly accurate brute-force search (contrast with HNSW)
See:
https://learn.microsoft.com/en-us/azure/search/vector-search-ranking

Large Language Model (LLM) – a model with general language understanding and capable of language generation. It is not tied to a small vocabulary, rather a comprehensive understanding of language. LLMs are the basis for generative AI

Meta-prompt – Overall guidance given (often once) to the LLM to describe how you wish it to respond (expert level, intermediate, beginner)

Native Function – C# code that works with a plugin (contrast with Prompt Function – which is written in simple text). Most app developers using Semantic Kernel will be creating a mix of Native Functions (often needed when tying the AI to the rest of your C# app) and Prompt Functions.

OpenAI – The amazing hundreds of individuals working together to guide the AI ship into the future!

OpenAI library for .NET – This is the official SDK provided by OpenAI and used by modern Semantic Kernel to connect ot OpenAI. For details, see:

- https://devblogs.microsoft.com/semantic-kernel/support-for-azure-ai-openai-and-openai-v2-is-coming/

OpenAI Plugin – your plugin, with a REST API, described by OpenAPI, that can be called by OpenAI:
Also see: https://learn.microsoft.com/en-us/semantic-kernel/agents/plugins/openai-plugins

OpenAPI – A standard way of describing a REST API. Very widely supported by pretty much every modern dev tool.

Planner – A planner converts what the user wants into a plan – which may involve multiple steps, or steps condensed. Note the early experimental planners have been superseded by function calling; there is ongoing work for next generation planners – for details see:

- https://devblogs.microsoft.com/semantic-kernel/the-future-of-planners-in-semantic-kernel/

Plugin - These are the skills (made up of Native Functions [C]) and Prompt Functions [text]) that you provide to the kernel for integration with the LLM.
Also see: OpenAI Plugin

Prompt – A user-generated text in natural language (e.g plain English)

Prompt Template – Creating a reusable prompt layout, based on the Handlebars templating engine (Also see: Handlebars)

Prompt Chaining – Combining prompts

Prompt Flow – A useful feature of Azure AI Studio to build your own prompts

Retrieval Augmented Generation – A pattern whereby embeddings for your documents are created, and then when a prompt is about to be sent to the LLM, a search of the embeddings via a vector database appending additional contextually relevant data to the prompt and then the prompt is sent to the LLM

Prompt Function – Text-based function definitions for use with plugins

Prompt Template – A prompt with variables

Token – The "currency" used to charge you for use of the LLM. One token usually represent a part of a word (though for small words can represent the entire word). Token usage is extremely important and a key design consideration for the team – as you move into production, with many users and a lot more data / documents, token usage can explode. So by carefully keeping it under control, you can more wisely manage the economics and also potentially use AI for more features in your app.

Copilot Studio – Part of Microsoft's Power Platform family, a low-code way of creating custom Copilots (easy to use by non-devs, but runtime fees will be quite expensive if you have many users).

Vector Database – A specifically designed database architecture for vectors used with embeddings (Note embeddings can be stored in volatile memory (without using a vector database) but production of embeddings costs tokens, so for many documents it is best to create the embeddings once and store in a vector database. Azure AI Search is an example of a vector database.

Abbreviations

GPT - Generative Pre-trained Transformer
LLM – Large Language Model
RAG – Retrieval Augmented Generation
SK – Semantic Kernel

2: .NET Features Used In Semantic Kernel

Overview

The Semantic Kernel codebase uses a number of interesting capabilities of .NET which we will review here. You may wish to refer back to this chapter as you wander through the codebase.

Dependency Injection And Service Collections

One of the primary features of Semantic Kernel is to provide a dependency injection container, known as a kernel, to control a range of services and plugins in performing AI functionality. This directly uses the dependency injection system built into .NET, so experienced .NET application developers will have already seen this in action in areas such as ASP.NET Core. Within Semantic Kernel, you can see it in the implementation of the Kernel class:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Kernel.cs

where the `Services` getter returns an `IServiceProvider`:

```
using Microsoft.Extensions.DependencyInjection;

/// </remarks>
public sealed class Kernel
{
    ...
    /// <summary>
    /// Gets the service provider used to query for services
    /// available through the kernel.
    /// </summary>
    public IServiceProvider Services { get; }
   ...
}
```

An important point to consider is the lifetime of the constructed object. Let's ask copilot for some background:

You

In .NET dependency injection, please give me a brief description of options for object lifetimes

Copilot

Sure, in .NET dependency injection, there are three main options for managing the lifetime of an object:

1. **Transient**: A new instance is created each time a service is requested. This is useful when you want a clean, separate instance for each operation.
2. **Scoped**: A single instance is created per scope. In a web application, a scope is created for each request, so services with this lifetime will be created once per request.
3. **Singleton**: A single instance is created and shared across the entire application. This is useful for stateless, thread-safe services that can be used by multiple operations concurrently.

These options give you control over how and when objects are created and destroyed, helping you manage resources effectively in your application.

As one example of use of these object lifetimes inside Semantic Kernel, see the KernelServiceCollectionExtensions type in:

- Services/KernelServiceCollectionExtensions.cs

which defines this extension method, AddKernel() that uses AddTransient:

```
public static class KernelServiceCollectionExtensions
{
    public static IKernelBuilder AddKernel(this IServiceCollection services)
    {
        ...
        services.AddTransient<KernelPluginCollection>();
```

As another example, this time using singletons, see KernelExtensions.cs in Core:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/KernelExtensions.cs

which has the AddFromType<T>() that uses AddSingleton:

```
public static IKernelBuilderPlugins AddFromType<T>(
                                          this IKernelBuilderPlugins plugins,
                                          string? pluginName = null)
    {
      plugins.Services.AddSingleton<KernelPlugin>(
       serviceProvider =>
         KernelPluginFactory.CreateFromType<T>(pluginName, serviceProvider));

      return plugins;
    }
```

Record Structs

We see record structs in use in area such as for MethodDetails in:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/Functions/KernelFunctionFromMethod.cs

Let's ask Copilot for a description:

Copilot

In C#, a `record struct` is a value type that provides built-in functionality for encapsulating data. It was introduced in C# 10 and offers similar functionality to a `record class`, but as a value type.

Here are some key features of a `record struct`:

- **Immutability**: By default, `record struct` types are immutable, meaning once a value is set, it cannot be changed.
- **Value Equality**: `record struct` types have value-based equality semantics. This means two `record struct` instances are equal if their corresponding properties are equal.
- **Nondestructive Mutation**: `record struct` types support nondestructive mutation through the `with` keyword.
- **Positional Syntax**: You can declare a `record struct` using positional syntax, which automatically creates properties for the constructor parameters.

Here's an example of a `record struct`:

C#

```csharp
public readonly record struct Point(double X, double Y, double Z);
```

Debugging

Two standard .NET attributes - `DebuggerDisplay` and `DebuggerTypeProxy` - are used to facilitate debugging. `DebuggerDisplay` sets the name to be used in the debugger and `DebuggerTypeProxy` provides an alternative (often simpliified or expanded) representation of the detail of the object. An example is from `KernelPlugin` definition in Abstractions:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Functions/KernelPlugin.cs

which after attaching the two attributes to the type also provides a nested class, `TypeProxy` to use with `DebuggerTypeProxy`:

```
[DebuggerDisplay("Name = {Name}, Functions = {FunctionCount}")]
[DebuggerTypeProxy(typeof(KernelPlugin.TypeProxy))]
public abstract class KernelPlugin : IEnumerable<KernelFunction>
{
   …

    /// <summary>Debugger type proxy for the kernel plugin.</summary>
    private sealed class TypeProxy(KernelPlugin plugin)
    {
        private readonly KernelPlugin _plugin = plugin;

        public string Name => this._plugin.Name;

        public string Description => this._plugin.Description;

        public KernelFunction[] Functions => [.. this._plugin.OrderBy(
                                   f => f.Name, StringComparer.OrdinalIgnoreCase)];
    }
}
```

Interlocked Methods

Semantic Kernel uses `Interlocked` methods where appropriate to ensure thread safety. Here is an explanation of why there are needed in a multithreaded environment:

Copilot

In .NET, the `Interlocked` class provides atomic operations for variables that are shared by multiple threads [1]. The methods of this class help protect against errors that can occur when the scheduler switches contexts while a thread is updating a variable that can be accessed by other threads, or when two threads are executing concurrently on separate processors [1].

Here are some key points about `Interlocked` methods:

- **Atomic Operations**: `Interlocked` only provides atomic operations such as `Increment`, `Decrement`, `Exchange`, etc [2]. These methods are useful when you need to perform an operation on a shared variable [2].
- **Thread Safety**: The methods of this class do not throw exceptions [1]. They ensure that the operation (like incrementing or decrementing a variable) and the storage of the resulting value happen as a single operation [1]. This is crucial in multi-threaded environments to prevent race conditions.
- **Memory Barriers**: `Interlocked` methods apply a full fence around instructions they execute, so reordering does not happen [3]. This means that it ensures the correct sequence of operations in a multi-threaded environment [3].

C# Language Features

Pragma Warnings

In various places in the codebase there is a desrie to switch off default warning sand errors for common situations. So expect to see use of pragmas such as this:

```
#pragma warning disable CA1716 // Identifiers should not match keywords
```

Pattern matching construct

Pattern matching constructs (new in C# 8) are sometimes used in the Semantic Kernel codebase. An example is from:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/KernelExtensions.cs

where the highlighted part of this line:

```
if (builder.Services is { Count: > 0 } services) { … }
```

is a pattern matching construct, which is used for a more succinct way of writing:

```
var services = builder.Services;
if (services.Count > 0) {
     …
}
```

Discard Variables

The underscore, '_', represents a discard variable and is used when we wish to clearly state in code that the variable will not be used. As an example of its use in Semantic Kernel, see this plugin demo:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/samples/Demos/HomeAutomation/Plugins/MyAlarmPlugin.cs

which has this code:

```
/// <summary>
/// Simple plugin to illustrate creating plugins which have dependencies
/// that can be resolved through dependency injection.
/// </summary>
public class MyAlarmPlugin(MyTimePlugin timePlugin)
{
    [KernelFunction, Description("Sets an alarm at the provided time")]
    public void SetAlarm(string time)
    {
        // Code to actually set the alarm using the time plugin
        // would be placed here
        _ = timePlugin;
    }
}
```

Primary Constructors

Within the Semantic Kernel codebase we see use of the new primary constructor feature in C# to make parameters available in the body of the type.

Here is an explanation of their purpose and how they work:

- https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/primary-constructors

An example usage from Semantic Kernel comes from `AggregatorAgent`:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Agents/Abstractions/AggregatorAgent.cs

where there is a need to pass in a factory method to create an `AgentChat`:

```
/// <summary>
/// Allows an <see cref="AgentChat"/> to participate in another
/// <see cref="AgentChat"/> as an <see cref="Agent"/>.
/// </summary>
/// <param name="chatProvider">A factory method that produces a
/// new <see cref="AgentChat"/> instance.</param>
public sealed class AggregatorAgent(Func<AgentChat> chatProvider) : Agent {}
```

3: Running Semantic Kernel Code

Overview

Before duving deep into the Semantic Kernel codebase, let's first explore what happens when a Semantic Kernel app runs. The sample code we will is from the Semantic Kernel repo:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/Demos/TimePlugin

We will highlight what happens at important points in the code.

Sample Code

```
// Copyright (c) Microsoft. All rights reserved.
#pragma warning disable VSTHRD111 // Use ConfigureAwait(bool)
#pragma warning disable CA1050 // Declare types in namespaces
#pragma warning disable CA2007 // Consider calling ConfigureAwait on the
awaited task

using System.ComponentModel;
using Microsoft.Extensions.Configuration;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;

var config = new ConfigurationBuilder()
    .AddUserSecrets<Program>()
    .AddEnvironmentVariables()
    .Build()
    ?? throw new InvalidOperationException("Configuration is not provided.");

ArgumentNullException.ThrowIfNull(
                   config["OpenAI:ChatModelId"], "OpenAI:ChatModelId");
ArgumentNullException.ThrowIfNull(config["OpenAI:ApiKey"], "OpenAI:ApiKey");

 1 var kernelBuilder = Kernel.CreateBuilder()
  2 .AddOpenAIChatCompletion(
    modelId: config["OpenAI:ChatModelId"]!,
    apiKey: config["OpenAI:ApiKey"]!);

 3 kernelBuilder.Plugins.AddFromType<TimeInformationPlugin>();
 4 var kernel = kernelBuilder.Build();

// Get chat completion service
var chatCompletionService =
 5     kernel.GetRequiredService<IChatCompletionService>();

// Enable auto function calling
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
};
```

```
Console.WriteLine(
               "Ask questions to use the Time Plugin such as:" +
                  "- What time is it?");

ChatHistory chatHistory = [];
string? input = null;
while (true)
{
    Console.Write("User > ");
    input = Console.ReadLine();
    if (string.IsNullOrWhiteSpace(input))
    {
        // Leaves if the user hit enter without typing any word
        break;
    }
     chatHistory.AddUserMessage(input);
     var chatResult = await chatCompletionService.GetChatMessageContentAsync(
                        chatHistory, openAIPromptExecutionSettings, kernel);
    Console.Write($"Assistant > {chatResult}");
}

/// <summary>
/// A plugin that returns the current time.
/// </summary>
public class TimeInformationPlugin
{
    /// <summary>
    /// Retrieves the current time in UTC.
    /// </summary>
    /// <returns>The current time in UTC. </returns>
    [KernelFunction, Description("Retrieves the current time in UTC.")]
    public string GetCurrentUtcTime()
        => DateTime.UtcNow.ToString("R");
}
```

Step 1: CreateBuilder

```
var kernelBuilder = Kernel.CreateBuilder()
```

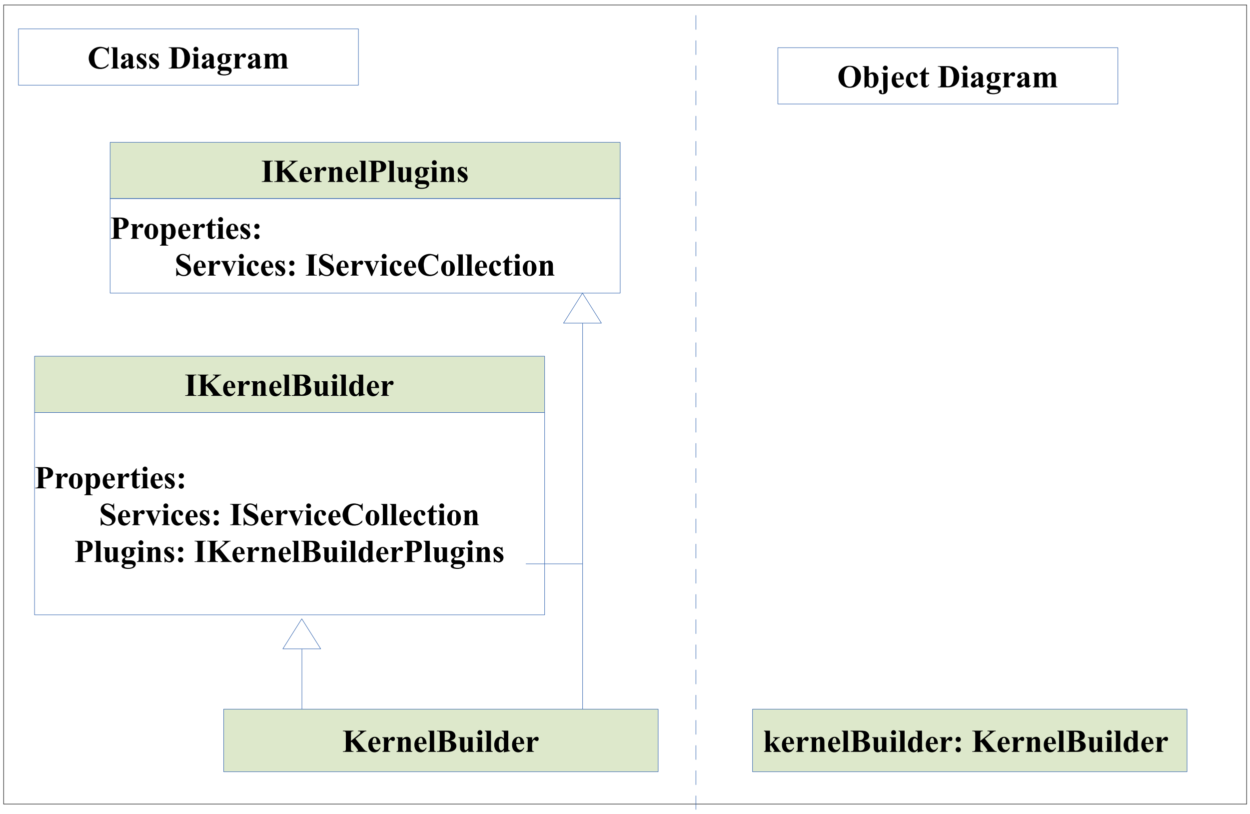
Creates a `KernelBuilder` which we will use to initialize services and plugins.

Defined in [Kernel.cs]() as:

```
public static IKernelBuilder CreateBuilder() => new KernelBuilder();
```

`CreateBuilder` simply creates a new `KernelBuilder` (using the parameterless constructor, which means later `Build()` can be called).

Initially, the Services and Plugins properties are empty.



Step 2: AddOpenAIChatCompletion

```
var kernelBuilder = Kernel.CreateBuilder().AddOpenAIChatCompletion
```

Registers the `OpenAIChatCompletion` service with DI.

Defined in [OpenAIKernelBuilderExtensions.cs]() as:

```
public static IKernelBuilder AddOpenAIChatCompletion(
                                this IKernelBuilder builder, ..) {

  OpenAIChatCompletionService Factory(
                    IServiceProvider serviceProvider, object? _) => new(..);

  builder.Services.AddKeyedSingleton<IchatCompletionService>(serviceId,
   (Func<IServiceProvider, object?, OpenAIChatCompletionService>)Factory);

  builder.Services.AddKeyedSingleton<ItextGenerationService>(
        serviceId, (Func<IServiceProvider, object?,
        OpenAIChatCompletionService>)Factory);
return builder;
}
```

this extension method adds a keyed singleton to the services collection for both `IChatCompletionService` and `ITextGenerationService`. Note `Factory` is a delegate, which only when invoked will actually create the instance.

Step 3: AddFromType

```
kernelBuilder.Plugins.AddFromType<TimeInformationPlugin>();
```

Constructs a `KernelPlugin`, which manages a collection of `KernelFunctions` - one each for every one of our methods we mark with the `KernelFunction` attribute.

Defined in KernelExtensions.cs as:

```
public static IKernelBuilderPlugins AddFromType<T>(
          this IKernelBuilderPlugins plugins, string? pluginName = null)
{
    plugins.Services.AddSingleton<KernelPlugin>(serviceProvider =>
      KernelPluginFactory.CreateFromType<T>(pluginName, serviceProvider));
    return plugins;
}
```

The `KernelPluginFactory` class has methods `CreateFromType`:

```
public static KernelPlugin CreateFromType<T>(
    string? pluginName = null, IServiceProvider? serviceProvider = null)
{
  serviceProvider ??= EmptyServiceProvider.Instance;
  return CreateFromObject(
    ActivatorUtilities.CreateInstance<T>(serviceProvider)!,
      pluginName, serviceProvider?.GetService<ILoggerFactory>());
}
```

and `CreateFromObject` which searches the methods of the type for matching attributes.

Step 4: Build

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

In the call to the `Build()` extension method is where the Kernel object itself is constructed.

`Build()` is defined at the end of: KernelExtensions in Core. its main line is:

```
return new Kernel(serviceProvider);
```

Step 5: GetRequiredService

```
var chatCompletionService =
       kernel.GetRequiredService<IChatCompletionService>();
```

The call to `GetRequiredService()` is where we get access to the completion service.

We also need to set up `OpenAIPromptExecutionSettings`

Step 6: AddUserMessage

```
ChatHistory chatHistory = [];
      while (true) {
        Console.Write("\nUser > ");
        input = Console.ReadLine();
        chatHistory.AddUserMessage(input);
        ...
      }
```

The call to `chatHistory.AddUserMessage` is where we add user messages to the chat which we will forward to the completion service.

`ChatHistory` is defined in: ChatHistory.cs in Abstractions and its main features are:

```
public class ChatHistory : Ilist<ChatMessageContent>,
                            IReadOnlyList<ChatMessageContent> {
            private readonly List<ChatMessageContent> _messages;

            public ChatHistory() {
                this._messages = [];
            }

          public void AddMessage(AuthorRole authorRole, string content,
                          Encoding? encoding = null,
                             IReadOnlyDictionary<string, object?>? Metadata =
                null) => this.Add(new ChatMessageContent(authorRole, content,
                                          null, null, encoding, metadata));

            public void Add(ChatMessageContent item) {
                Verify.NotNull(item);
                this._messages.Add(item);
            }

            public void AddUserMessage(string content) =>
                this.AddMessage(AuthorRole.User, content);
          ...
    }
```

Step 7: GetChatMessageContentAsync

```
var chatResult = await chatCompletionService.GetChatMessageContentAsync(
          chatHistory, openAIPromptExecutionSettings, kernel);
```

The call to `GetChatMessageContentAsync()` is where we send on the chat history to the AI service and process the results, such as it calling into our plugins.

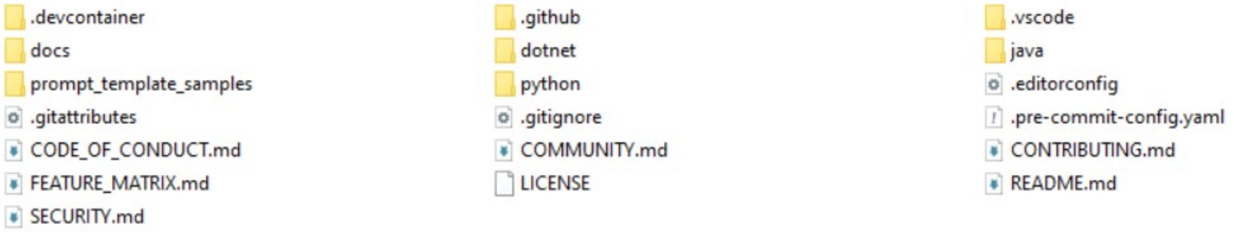
How `GetChatMessageContentAsync()` works is somewhat complicated - for a detailed explanation, please read the later chapters on Core and Connectors.

4: Layout Of The Semantic Kernel Repo

Overview

We will start with a general tour of the Semantic Kernel source tree and then zoom in to the .NET solution.

Layout of the Root Folder



The root of the source tree has sub-folders for each of the target environments for Semantic Kernel – dotnet, python and java. In this source tour we will focus on .NET and note the Python and Java source trees provide roughly the same functionality. We bring your attention to the feature matrix- there is a markdown file in the root folder that in the past had a matrix of features for the different implementations, now it just contains a comment, stating the feature matrix has been moved to:

- https://learn.microsoft.com/en-us/semantic-kernel/get-started/supported-languages

The LICENSE file states Semantic Kernel is provided under MIT licensing, which is good.

There are a number of markdown files related to the developer community:

- COMMUNITY.md
- CODE_OF_CONDUCT.md
- CONTRIBUTING.md

The security markdown file:

- SECURITY.md

invites anyone who discovers a security issue with the project to report the issue and Microsoft will investigate.

There are a bunch of dot files and folders, to do with editor config, devcontainers, git and GitHub. Editor config is defined in this file:

- .editorconfig

and is based on the Editor Config system:

- https://aka.ms/editorconfigdocs

It helps keep a consistent coding style across a repo used by multiple developers.

The .devcontainer sub-folder has a single file:

- devcontainer.json

with this content:

```
{
	"image": "mcr.microsoft.com/devcontainers/universal:2",
	"features": {
		"ghcr.io/devcontainers/features/node:1": {},
		"ghcr.io/devcontainers/features/dotnet:1": {
			"version": "8"
		},
	"ghcr.io/jlaundry/devcontainer-features/azure-functions-core-tools:1": {}
	},
	"customizations": {
		"vscode": {
			"extensions": [
				"ms-dotnettools.dotnet-interactive-vscode",
				"ms-semantic-kernel.semantic-kernel",
				"esbenp.prettier-vscode"
			]
		}
	},
	"postCreateCommand": "sudo chmod a+rwx /usr/share/dotnet"
	// avoids needing to run as 'sudo' when starting KernelHttpServer
}
```

You can learn more about devcontainers here:

- https://code.visualstudio.com/docs/devcontainers/containers

In particular:

> *"A devcontainer.json file in your project tells VS Code how to access (or create) a development container with a well-defined tool and runtime stack. This container can be used to run an application or to separate tools, libraries, or runtimes needed for working with a codebase."*

The .gitignore file tells Git which files to ignore:

- .gitignore

The .gitattributes file configure CRLF settings:

- .gitattributes

with this content:

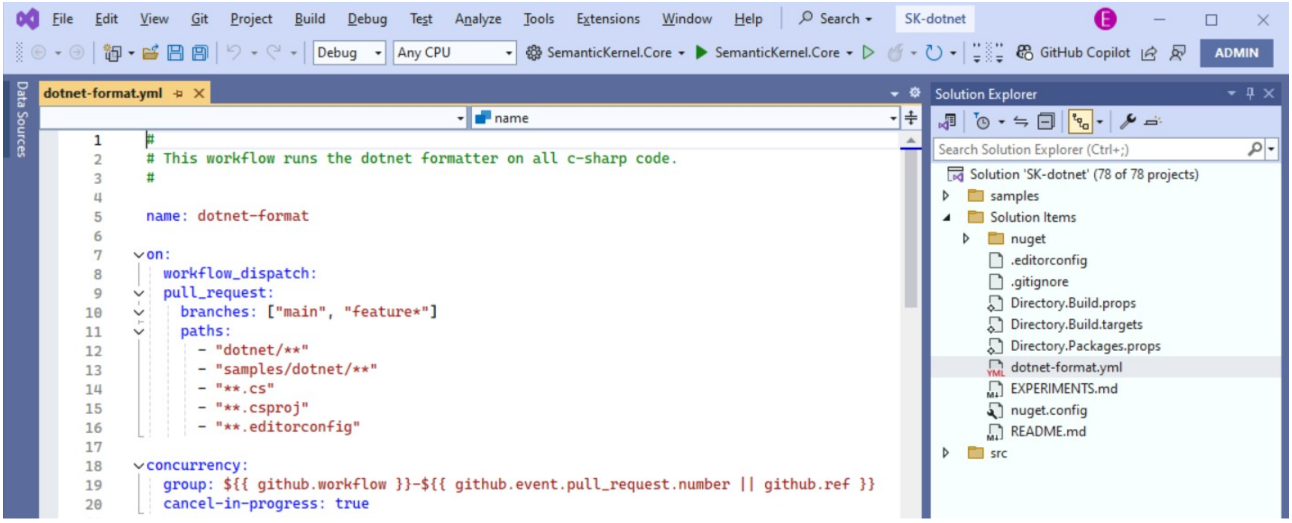
```
# Auto-detect text files, ensure they use LF.
* text=auto eol=lf working-tree-encoding=UTF-8

# Bash scripts
*.sh  text eol=lf
*.cmd text eol=crlf
```

Layout of the Root Folder

The repo's root folder also has sub-folders for GitHub, docs and samples. The most important part of .github sub-folder are its Workflows:

- .github/workflows

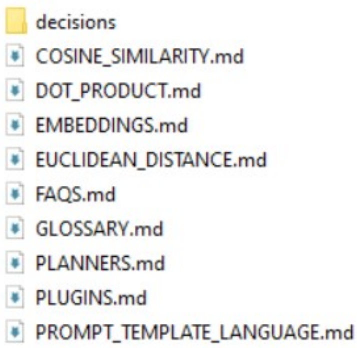
An example is dotnet-format.yml, which runs the Dotnet formatter on C# code. Later when we explore the Semantic Kernel Dotnet solution, we see under Solution Items that this dotnet-format.yml is one of the items listed:



Note there are docs folders in the repo root and in dotnet. In root, the docs sub-folder has a few markdown files with notes about interesting topics:

- https://github.com/microsoft/semantic-kernel/tree/main/docs

It also contains a decisions sub-folder explaining certain design decisions (which are actually interesting to read through).



The prompt_template_samples sub-folder include plugins using prompt templates:

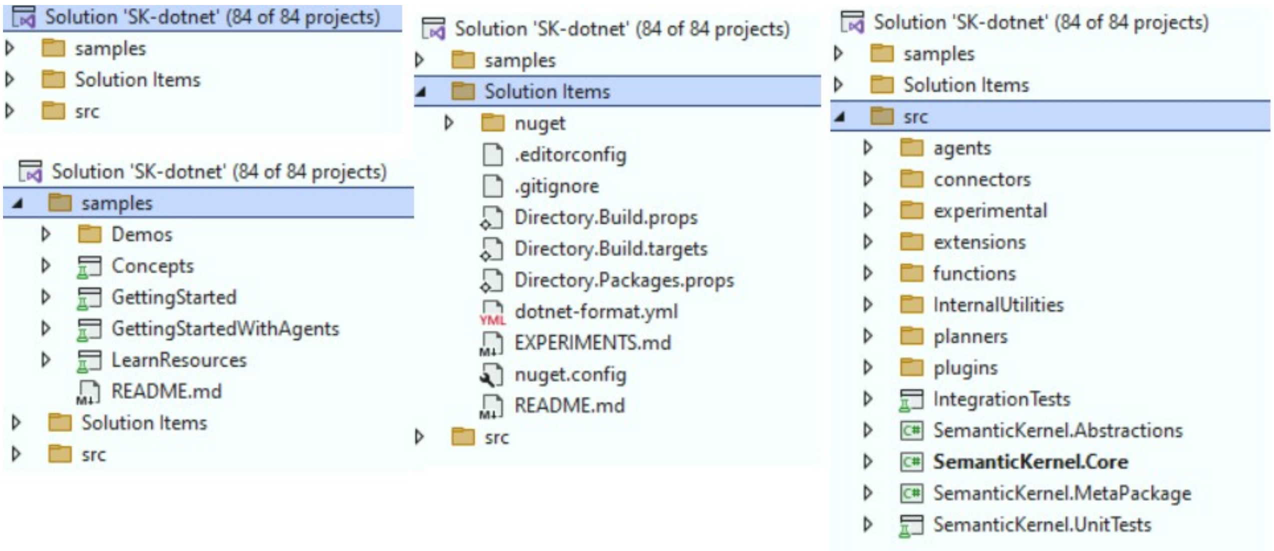
- https://github.com/microsoft/semantic-kernel/tree/main/prompt_template_samples

(these are written in natural language - so are applicable to all programming environments). Separately, note there is also a samples folder under the dotnet sub-folder, with .NET-specific native function examples (written in C#).

Layout Of The .NET Solution

The Semantic Kernel implementation for each target environment is located in a suitably named sub-folder. We are going to focus on the .NET implementation, so this is where we need to explore:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet



Building the Source

After you get the source from Github:

- https://github.com/microsoft/semantic-kernel

then you can easily build it.

You should have no problem working with it on any device that runs Visual Studio 2022. For my experimentation working with the source tree, I use Windows Server 2022, running in a VM on Azure, accessed via Azure Bastion, with Visual Studio 2022 (latest version).

To build yourself (which you should do), under the repo root, simply navigate to the dotnet folder and open up the SK-dotnet.sln file in Visual Studio.

It has dozens of projects in the solution so building takes a little while, including syncing with nuget the first time. Then, like any other .NET solution, under the Build menu, select Build Solution. That is it.

Stable vs. Experimental

Some parts of the Semantic Kernel are considered stable (expected not to change much into the future - though may have additions) and some are considered experimental (still in active development, subject to likely change). There is an "Experimental" sub-folder, which are experimental, but also it is important to note some other assemblies (not under the Experimental sub-folder) are marked as Experimental. Look for the `Experimental` attribute in the assembly's AssemblyInfo.cs:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/

Connectors/Connectors.HuggingFace/AssemblyInfo.cs

and when you see this you know it is experimental:

```
// Copyright (c) Microsoft. All rights reserved.

using System.Diagnostics.CodeAnalysis;

// This assembly is currently experimental.
[assembly: Experimental("SKEXP0070")]
```

If you wish to use experimental code in your solution, then disable the pragma warnings by adding this to your code:

```
#pragma warning disable SKEXP0XYZ // Type is for evaluation purposes only and
is subject to change or removal in future updates. Suppress this diagnostic
to proceed.

// Your code

#pragma warning restore SKEXP0XYZ // Type is for evaluation purposes only and
is subject to change or removal in future updates. Suppress this diagnostic
to proceed.
```

This file:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/docs/EXPERIMENTS.md

provides details about which assembly is marked as experimental.

This is the `Experimental` attribute:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Diagnostics/ExperimentalAttribute.cs

Files within the root of the .NET solution

The Visual Studio solution (*.sln) file for Semantic Kernel is:

- SK-dotnet.sln

and it has a few helper files (e.g. Directory.Build.targets). It can be build from within Visual Studio or from the command line, either with Windows Cmd or Bash:

- build.cmd
- build.sh

The nuget.config file:

- nuget.config

defines nuget configuration (see also the Nuget sub-folder):

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <packageSources>
    <clear />
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json" />
  </packageSources>
```

```
  <packageSourceMapping>
    <packageSource key="nuget.org">
      <package pattern="*" />
    </packageSource>
  </packageSourceMapping>

</configuration>
```

Sub-Folders within the root of the .NET solution

The .NET solution directly contains sub-folders for:

- docs
- notebooks
- nuget
- samples
- src

We will spend most of our time exploring the src sub-folder, but first let's quickly look at the others.

Docs contains markdown documentation files, related to experiments:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/docs/EXPERIMENTS.md

and information about the models:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/docs/MODELS.md

and telemetry:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/docs/TELEMETRY.md

Note docs under the dotnet sub-folder is quite distinct from the docs sub-folder under the entire repo.

The notebooks sub-folder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/notebooks

contains Jupyter notebooks demonstrating various aspects of how the Semantic Kernel can be used. The README.md file in notebooks is important as it explains how to configure them to run in Visual Studio Code (scroll down this page to read):

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/notebooks

The nuget sub-folder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/nuget

contains Nuget related content, such as readme and an icon, together with property settings.

The samples sub-folder contains .NET samples:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples

When looking for samples, explorer here and also the notebooks sub-folder and also the samples sub-folder in the repo's root (which contains samples not specific to any implementation).

src folder

From now on we will focus on the dotnet src sub-folder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src

as here is where we will find all the C# code that we are keenly interested in exploring. If we exclude the test sub-folders, there are these key sub-folders here:

Semantic Kernel (C#/.NET Edition) - Src Folder

Agents	Core	Extensions	MetaPackage
Plugins	Planners	Experimental	
Functions	Connectors	Internal Utilities	
Abstractions			

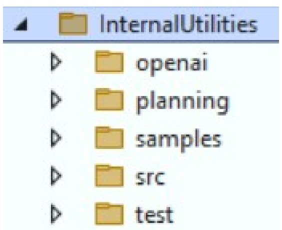
Let's first get an overview of how they are arranged and what is in each src sub-folder.

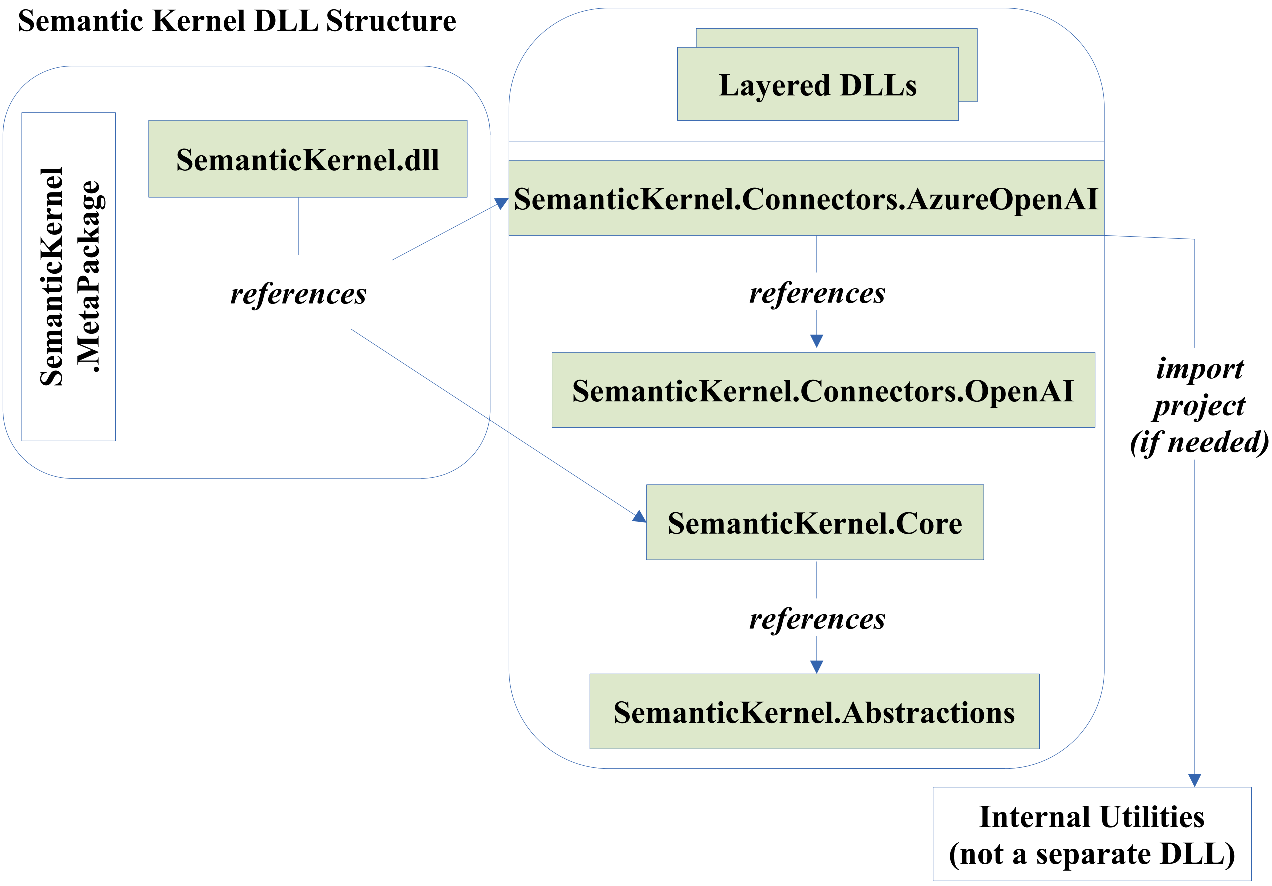
DLL Structure

The Semantic Kernel SDK comes with a meta package, that produces SemanticKernel.dll – its role is to allows apps that reference it to automatically reference a group of assemblies that are often needed together (namely Abstractions, Core and Connectors.OpenAI). Additional layered DLLs may be referenced as needed.

Internal Utilities

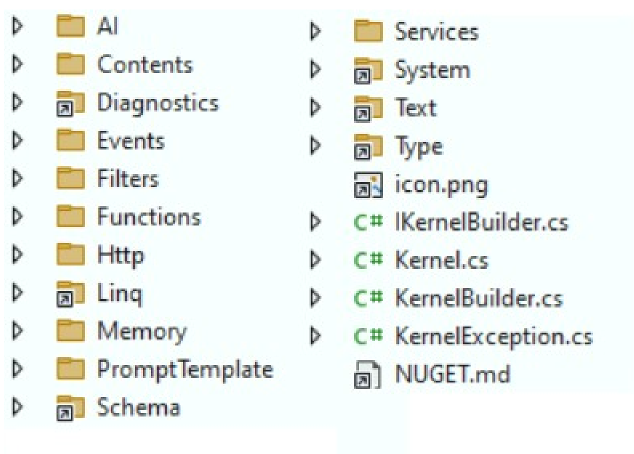
https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/InternalUtilities





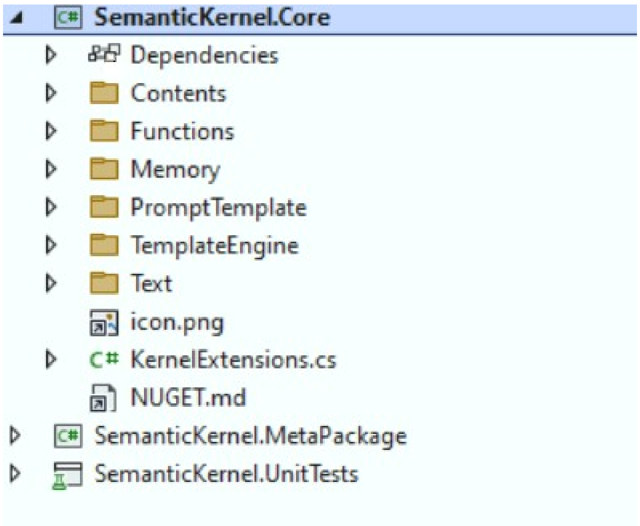
Abstractions

https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/SemanticKernel.Abstractions



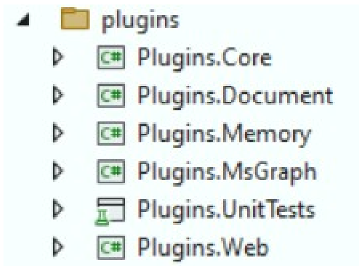
Core

https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/SemanticKernel.Core



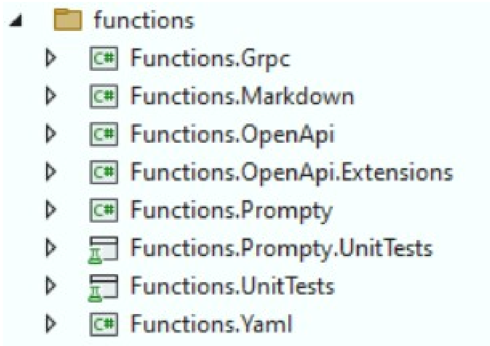
Plugins

https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Plugins

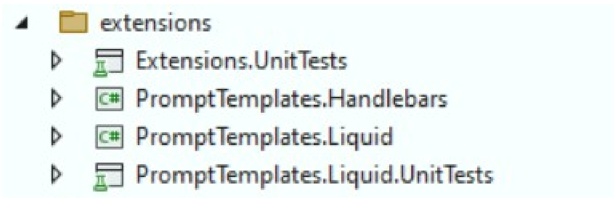


Functions

https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Functions



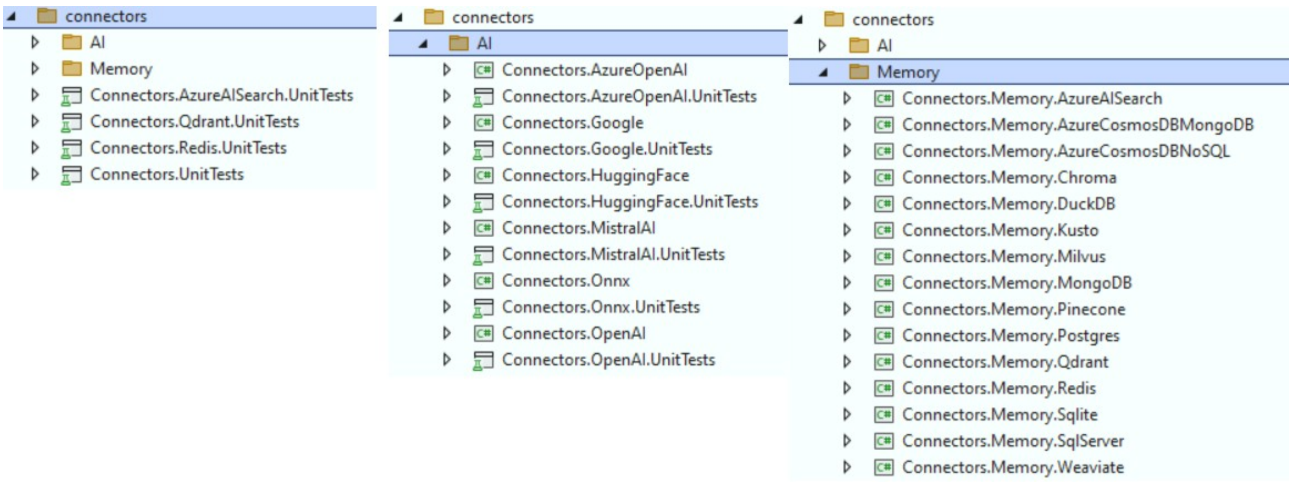
Extensions



https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Extensions

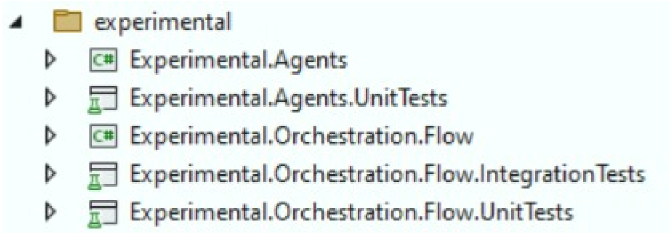
Connectors

https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Connectors



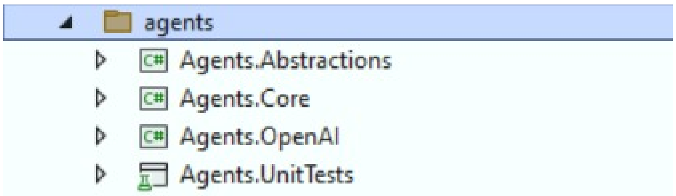
Experimental

https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Experimental



Agents

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Agents



Planners

https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Planners

Note: The use of planners in Semantic Kernel is evolving – the current experimental planners have been superseded by the much better automatic function calling approach. For details, see this Semantic Kernel blog post – showing decrease from 1380 to 240 in token usage for a sample scenario:

- https://devblogs.microsoft.com/semantic-kernel/planning-with-semantic-kernel-using-automatic-function-calling/

There is interesting work being carried out on more advanced future planners – keep an eye on the Semantic Kernel blog, such as this post:

- https://devblogs.microsoft.com/semantic-kernel/the-future-of-planners-in-semantic-kernel/

src/SemanticKernel.MetaPackage

The aim of a meta-package is to allow a single reference act as a reference to a number of assemblies. For Semantic Kernel, the meta-package references the assemblies most apps will need; additional references can be added for the needed layered assemblies. If an app finds what is in the meta-package does not suit, that is fine, the app should not reference the meta-package and instead reference the individual packages as needed.

The SemanticKernel.MetaPackage sub-folder has just a single file:

- SemanticKernel.MetaPackage.csproj

It has no C# code at all.

The csproj has two important package references (and Core in turn references Abstractions); it also specifies the assembly name, root namespace and provides a description:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <AssemblyName>Microsoft.SemanticKernel</AssemblyName>
    <RootNamespace>$(AssemblyName)</RootNamespace>
    <TargetFrameworks>net8.0;netstandard2.0</TargetFrameworks>
  </PropertyGroup>
  <Import Project="$(RepoRoot)/dotnet/nuget/nuget-package.props" />
  <PropertyGroup>
    <!-- NuGet Package Settings -->
    <Title>Semantic Kernel</Title>
    <Description>Semantic Kernel common package collection, including SK
Core, OpenAI, Azure OpenAI, DALL-E 2.
Empowers app owners to integrate cutting-edge LLM technology quickly and
easily into their apps.</Description>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference
       Include="..\SemanticKernel.Core\SemanticKernel.Core.csproj"
       PrivateAssets="none" />
    <ProjectReference Include=
       "..\Connectors\Connectors.AzureOpenAI\Connectors.AzureOpenAI.csproj"/>
  </ItemGroup>
</Project>
```

When we build, we see these DLLs are produced – note *Semantic Kernel.dll* is tiny as it only has references (this is a Debug build – a Release build will be even smaller); Note there is no DLL for *Internal Utilities* (we'll explore that next)

Microsoft.SemanticKernel.dll	5 KB
Microsoft.SemanticKernel.Core.dll	137 KB
Microsoft.SemanticKernel.Connectors.OpenAI.dll	237 KB
Microsoft.SemanticKernel.Connectors.AzureOpenAI.dll	34 KB
Microsoft.SemanticKernel.Abstractions.dll	249 KB

Architectural Decision Records (ADRs) in Docs/Decisions

Under the Semantic Kernel's repo root folder, in docs/decisions, a list of Architectural Decision Records (ADRs) is maintained. An ADR is defined as:

"An Architectural Decision (AD) is a justified software design choice that addresses a functional or non-functional requirement that is architecturally significant."

As an example, relevant to the MetaPackage we have just covered, here is one from a feature branch:

- https://github.com/microsoft/semantic-kernel/blob/feature-connectors-openai-release/docs/decisions/0047-azure-open-ai-connectors.md

that includes this:

"Meta Package Strategy - Currently the Microsoft.SemanticKernel package is a meta package that includes both SemanticKernel.Core and SemanticKernel.Connectors.OpenAI, with the new changes a new project will be added to the meta package SemanticKernel.Connectors.AzureOpenAI that will include the new Azure OpenAI connector."

Explanation: Up to recently both OpenAI and Azure OpenAI have shared the same connector. From now on, they will have separate connectors, and what this paragraph is stating is that both will be in the metapackage.

As has already been recommended, you should be paying close attention to the Semantic Kernel blog, where important announcements are regularly made concerning the evolution of Semantic Kernel. For example, here is an post from Mark Wallace about "updating our existing OpenAI connectors to use the new V2 OpenAI library and Azure.AI.OpenAI library":

- https://devblogs.microsoft.com/semantic-kernel/support-for-azure-ai-openai-and-openai-v2-is-coming/

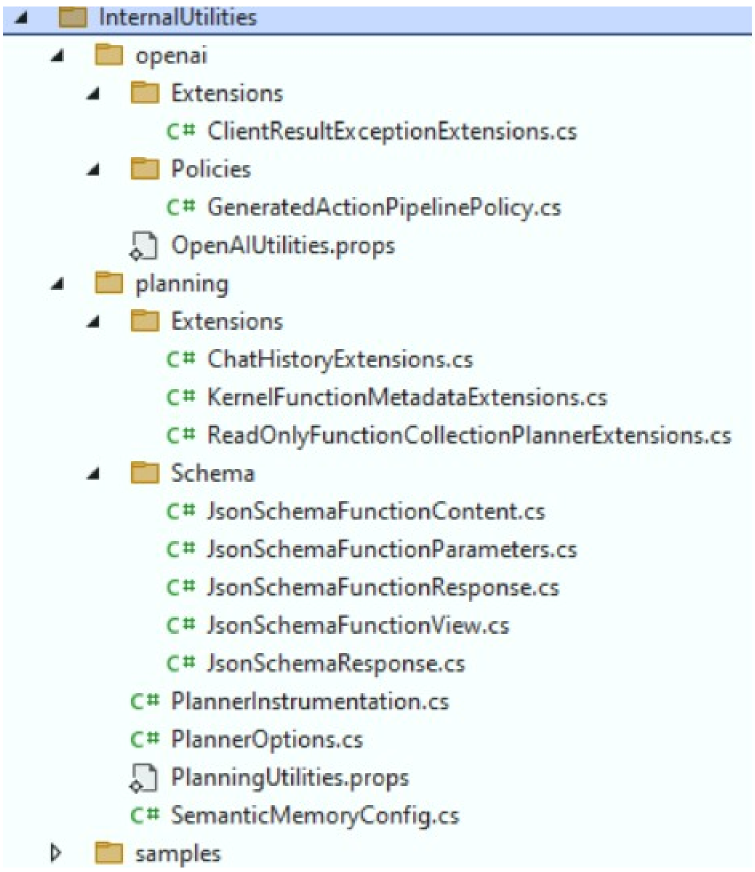
5: Internal Utilities

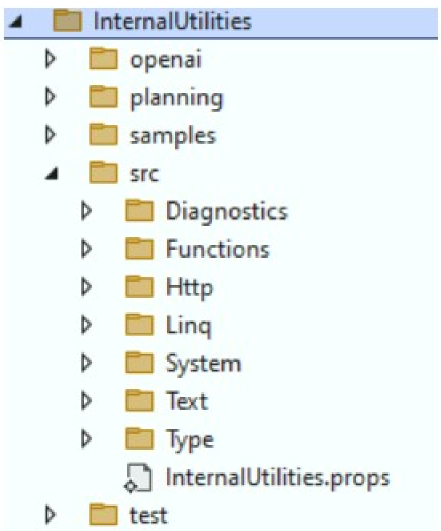
Overview

The InternalUtilities source folder include helper functionality needed in multiple other locations. It does not provide a standalone assembly, instead it is imported as needed and uses the MSBuild Link feature. Its capabilities are located in two separate sub-folders – src (for general capabilities) and planning (for common capabilities specifically for use in the planners). Each can be imported separately (via its own .props file).

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/InternalUtilities

Its source is laid out as follows:





Note there is no .csproj file; also note the InternalUtilities.props and PlanningUtilities.props files.

How it is imported

Before exploring the source, let's see how the import works. Only some of the main assemblies use Internal Utilities and those that need need to import the relevant .props file. For example, Abstractions only uses InternalUtilities.props, whereas the Handlebars planner imports both InternalUtilities.props and InternalUtilities.props; and Core imports neither.

The InternalUtilities.props file:

- src/InternalUtilities/src/InternalUtilities.props

contains this command:

```
<Project>
  <ItemGroup>
    <Compile Include="$(RepoRoot)/dotnet/src/InternalUtilities/src/**/*.cs"
Link="%(RecursiveDir)%(Filename)%(Extension)" />
  </ItemGroup>
</Project>
```

The PlanningUtilities.props file:

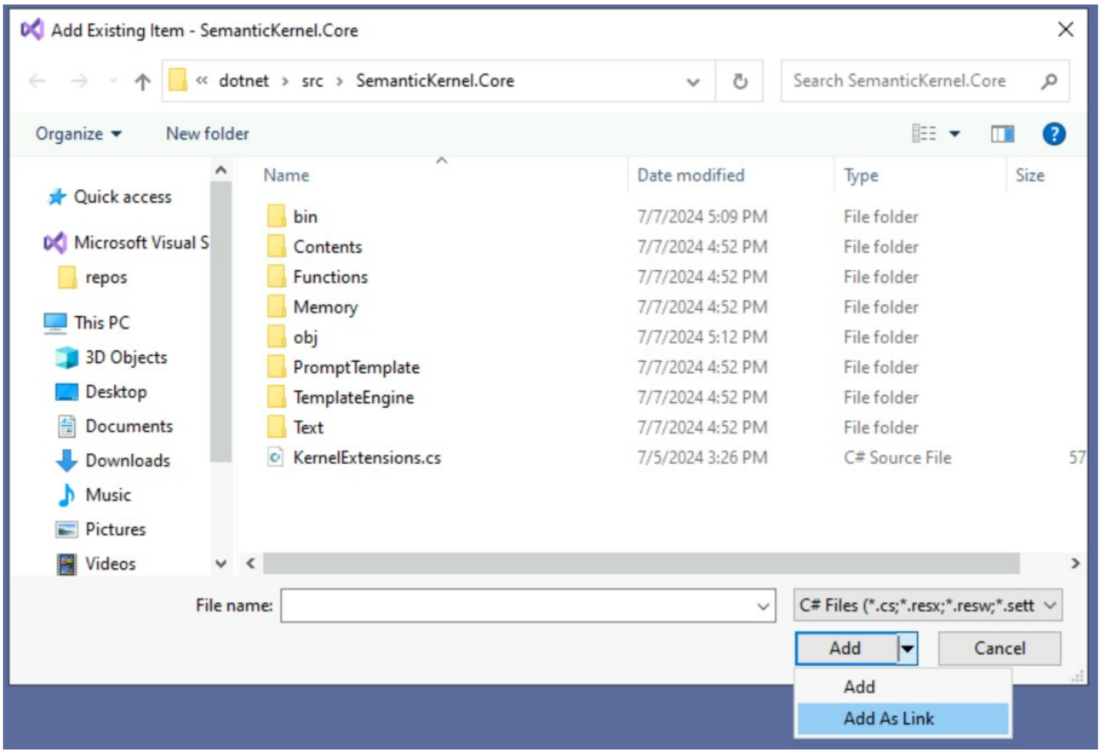
- src/InternalUtilities/planning/PlanningUtilities.props

is almost the same command (but note different path):

```
<Project>
  <ItemGroup>
    <Compile
Include="$(RepoRoot)/dotnet/src/InternalUtilities/planning/**/*.cs" Link="%
(RecursiveDir)%(Filename)%(Extension)" />
  </ItemGroup>
</Project>
```

The Link property enables a developer include a file from a separate location, without copying the file. This is particularly useful if you wish to use the same source file in multiple assemblies, without creating a separately assembly just for the shared source- which is what is happening with Semantic Kernel.

Usually linked files are added via the Add Item command in Solution Explorer in Visual Studio, where you need to select *Add As Link* from the *Add* drop-down.



By putting the Link command in a separate props file, it means it can be used from multiple places. It is noted the Abstractions csproj:

- SemanticKernel.Abstractions.csproj

includes:

```
<Project Sdk="Microsoft.NET.Sdk">
  …
<Import Project="$(RepoRoot)/dotnet/nuget/nuget-package.props" />
  <Import Project=
    "$(RepoRoot)/dotnet/src/InternalUtilities/src/InternalUtilities.props" />
```

similarly, it is noted the handlebars planner csproj:

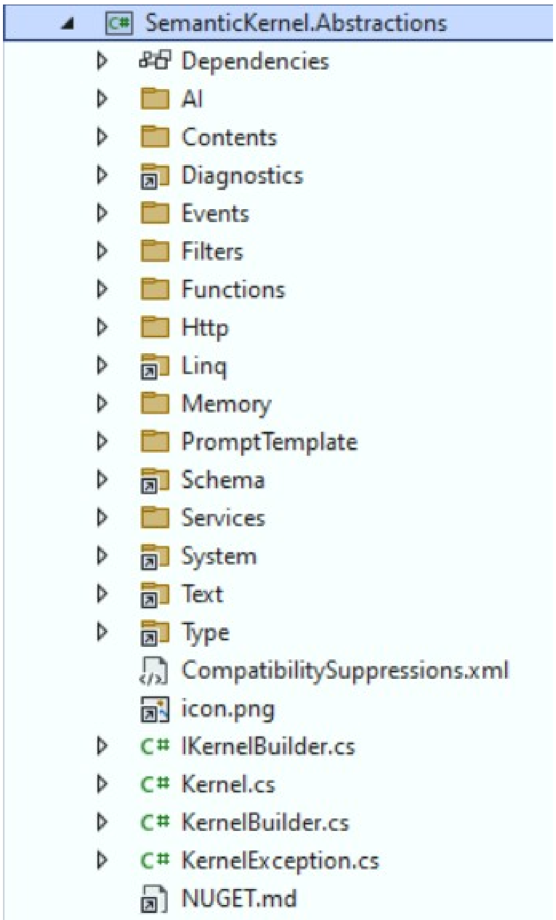
- src/Planners/Planners.Handlebars/Planners.Handlebars.csproj

imports both InternalUtilities.props and PlanningUtilities.props:

```
<Project Sdk="Microsoft.NET.Sdk">
  ...

  <Import Project="$(RepoRoot)/dotnet/nuget/nuget-package.props" />
  <Import Project=
    "$(RepoRoot)/dotnet/src/InternalUtilities/src/InternalUtilities.props" />
  <Import Project=
  "$(RepoRoot)/dotnet/src/InternalUtilities/planning/PlanningUtilities.props"
  />
```

There are a number of source sub-folders for InternalUtilities/src (Diagnostics, Functions, Http, Linq, System, Text and Type) and when we open SemanticKernel in Visual Studio, we see some are highlighted with a distinct icon to show they are linked (Http is not highlighted - we'll get back to that presently).

How it is imported

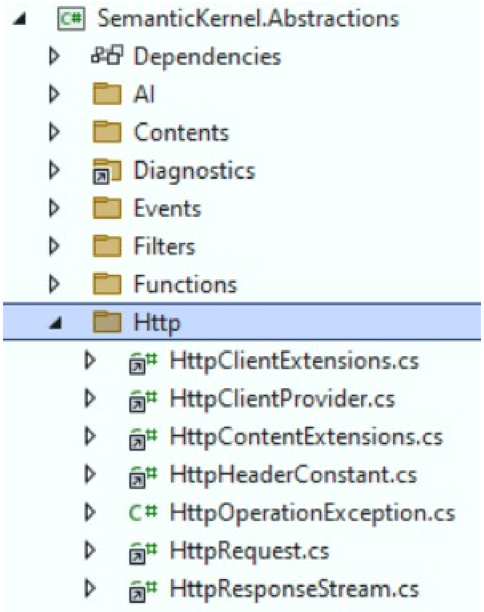


There are two source sub-folders in InternalUtilities/planning along with a few source files directly in its root folder.

Before we conclude with this topic, you will be curious about the Http sub-folder in Abstractions – why did it not have the link icon? The reason is Abstractions adds an extra file to Http (called HttpOperationException.cs) and so has its own sub-folder for this:

- src/SemanticKernel.Abstractions/Http

When you view the Http sub-folder in Visual Studio, y9ou do not see the linked folder icon for Http; you do see both the local source file and the linked source files (and note the distinct folder icons):



How InternalUtilities are added to Solution

Because InternalUtilities is not a regular csproj, it is added to the solution directly in the .sln file:

- SK-dotnet.sln

Features

Diagnostics

The Diagnostics sub-folder includes a number of attributes:

- ExperimentalAttribute – marks a type as experimental
- NullableAttributes – used with nullable characteristics
- CompilerServicesAttributes – for the compiler

The following is “Reserved to be used by the compiler for tracking metadata”:

- IsExternalInit

ExceptionExtensions lists the exceptions that should not be caught by the kernel:

```
[ExcludeFromCodeCoverage]
internal static class ExceptionExtensions
{
    /// <summary>
    /// Check if an exception is of a type that should not be
    /// caught by the kernel.
    /// </summary>
    /// <param name="ex">Exception.</param>
    /// <returns>True if <paramref name="ex"/> is a critical exception and
    /// should not be caught.</returns>
    internal static bool IsCriticalException(this Exception ex)
        => ex is ThreadAbortException
            or AccessViolationException
            or AppDomainUnloadedException
            or BadImageFormatException
```

```
            or CannotUnloadAppDomainException
            or InvalidProgramException;
}
```

The Verify.cs source file is used to verify various characteristics (function names, that strings start with certain sequences of characters, etc.)

Http

The Http sub-folders contains helper functionality dealing with HTTP.

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/InternalUtilities/src/Http

HttpHeaderConstant sets the user agent to be "Semantic-Kernel" (note hyphen) for all HTTP requests coming from Semantic Kernel:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Http/HttpHeaderConstant.cs

```
/// <summary>Provides HTTP header names and values for common
/// purposes.</summary>
[ExcludeFromCodeCoverage]
internal static class HttpHeaderConstant
{
    public static class Names
    {
        /// <summary>HTTP header name to use to include the Semantic Kernel
        ///  package version in all HTTP requests issued by Semantic
        /// Kernel.</summary>
        public static string SemanticKernelVersion
                                => "Semantic-Kernel-Version";
    }

    public static class Values
    {
        /// <summary>User agent string to use for all HTTP requests issued
        /// by Semantic Kernel.</summary>
        public static string UserAgent => "Semantic-Kernel";

        /// <summary>
        /// Gets the version of the <see cref="System.Reflection.Assembly"/>
        /// in which the specific type is declared.
        /// </summary>
        /// <param name="type">Type for which the assembly version
        /// is returned.</param>
        public static string GetAssemblyVersion(Type type)
        {
            return type.Assembly.GetName().Version!.ToString();
        }
    }
}
```

HttpRequest creates the request:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Http/HttpRequest.cs

by serializing the payload to UTF8 and setting the media type to “application/json”:

```
    private static HttpContent? CreateJsonContent(object? payload)
    {
        HttpContent? content = null;
        if (payload is not null)
        {
            byte[] utf8Bytes = payload is string s ?
                Encoding.UTF8.GetBytes(s) :
                JsonSerializer.SerializeToUtf8Bytes(payload,
                  JsonOptionsCache.Default);

            content = new ByteArrayContent(utf8Bytes);
            content.Headers.ContentType =
               new MediaTypeHeaderValue(
                   "application/json") { CharSet = "utf-8" };
        }

        return content;
    }
```

HttpClientExtensions is where the actual call to HttpClient's `Send` is made:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Http/HttpClientExtensions.cs

and it provides additional error handling / logging if the response is not successful.

```
internal static async Task<HttpResponseMessage>
SendWithSuccessCheckAsync(this HttpClient client, HttpRequestMessage request,
HttpCompletionOption completionOption, CancellationToken cancellationToken)
{
    HttpResponseMessage? response = null;
    try
    {
        response = await client.SendAsync(request, completionOption,
                                  cancellationToken).ConfigureAwait(false);
    }
    catch (HttpRequestException e)
    {
        throw new HttpOperationException(
                    HttpStatusCode.BadRequest, null, e.Message, e);
    }

    if (!response.IsSuccessStatusCode)
    {
        string? responseContent = null;
        try
        {
            // On .NET Framework, EnsureSuccessStatusCode disposes of the
            // response content; that was changed years ago in .NET Core,
            // but for .NET Framework it means in order to read the response
            // content in the case of failure, that has to be
            // done before calling EnsureSuccessStatusCode.
            responseContent = await response!.Content.ReadAsStringAsync()
                                                .ConfigureAwait(false);
            response.EnsureSuccessStatusCode(); // will always throw
        }
        catch (Exception e)
        {
            throw new HttpOperationException(
                        response.StatusCode, responseContent, e.Message, e);
```

```
            }
        }

        return response;
    }
```

Linq

The Linq sub-folder contains an implementation of AsyncEnumerable:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Linq/AsyncEnumerable.cs

System

The System sub-folder contains three files.

EnvExtensions converts a 1 / 0 environment variable setting into a bool:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/System/EnvExtensions.cs

InternalTypeCnverter converts types to strings in a culture-safe manner:

https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/System/InternalTypeConverter.cs

TypeConverterFactory creates an appropriate type converter:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/System/TypeConverterFactory.cs

Text

The Text sub-folder has two source files.

ReadOnlyMemoryConverter.cs deals with RedOnlyMemory:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Text/ReadOnlyMemoryConverter.cs

JsonOptionsCache.cs has helper functionality to deal with JSON:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Text/JsonOptionsCache.cs

```
internal static class JsonOptionsCache
{
    /// <summary>Singleton for <see
cref="ReadOnlyMemoryConverter"/>.</summary>
    public static ReadOnlyMemoryConverter ReadOnlyMemoryConverter { get; } =
new();

    public static JsonSerializerOptions Default { get; } = new()
    {
        Converters = { ReadOnlyMemoryConverter },
    };
```

```
    public static JsonSerializerOptions WriteIndented { get; } = new()
    {
        WriteIndented = true,
        Converters = { ReadOnlyMemoryConverter },
    };

    public static JsonSerializerOptions ReadPermissive { get; } = new()
    {
        AllowTrailingCommas = true,
        PropertyNameCaseInsensitive = true,
        ReadCommentHandling = JsonCommentHandling.Skip,
        Converters = { ReadOnlyMemoryConverter },
    };
}
```

Type

The Type sub-folder has a single source file, TypeExtensions.cs:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/InternalUtilities/src/Type/TypeExtensions.cs

and it has a single type, TypeExtensions, which is used to discover the result type from a generic parameter:

```
internal static class TypeExtensions
{
    public static bool TryGetGenericResultType(this Type? returnType,
                                                out Type resultType)
    {
        resultType = typeof(object);
        if (returnType is null)
        {
            return false;
        }

        if (returnType.IsGenericType)
        {
            Type genericTypeDef = returnType.GetGenericTypeDefinition();

            if (genericTypeDef == typeof(Task<>)
                || genericTypeDef == typeof(Nullable<>)
                || genericTypeDef == typeof(ValueTask<>))
            {
                resultType = returnType.GetGenericArguments()[0];
            }
            else if (genericTypeDef == typeof(IEnumerable<>)
                || genericTypeDef == typeof(IList<>)
                || genericTypeDef == typeof(ICollection<>))
            {
                resultType = typeof(List<>).MakeGenericType(
                                 returnType.GetGenericArguments()[0]);
            }
            else if (genericTypeDef == typeof(IDictionary<,>))
            {
                Type[] genericArgs = returnType.GetGenericArguments();
                resultType = typeof(Dictionary<,>).MakeGenericType(
                                    genericArgs[0], genericArgs[1]);
            }
```

```
            return true;
        }

        return false;
    }

    public static string GetFriendlyTypeName(this Type type)
    {
        if (type.IsGenericType)
        {
            string typeName = type.GetGenericTypeDefinition().Name;
            // Remove the `1, `2 etc from the type name which indicates
            // the number of generic arguments
            typeName = typeName.Substring(0, typeName.IndexOf(
                                  '`', (int)StringComparison.Ordinal));
            string genericArgs = string.Join(
               ", ", type.GetGenericArguments().Select(GetFriendlyTypeName));
            return $"{typeName}<{genericArgs}>";
        }

        return type.Name;
    }
}
```

Openai Sub-folder

The openai sub-folder in internal utiltiies:

- https://github.com/microsoft/semantic-kernel/tree/feature-connectors-openai/dotnet/src/InternalUtilities/openai

contains some helper functionality used in the conectors.

The ClientResultExceptionExtensions.cs file:

Extensions/ClientResultExceptionExtensions.cs

converts from a ClientResultException to an HttpOperationException.

The GeneratedActionPipelinePolicy.cs file:

- Policies/GeneratedActionPipelinePolicy.cs

defines an action pipeline policy, which provides synchrionous and asynchronous methods to process the actions:

```
internal sealed class GenericActionPipelinePolicy : PipelinePolicy
{
    private readonly Action<PipelineMessage> _processMessageAction;
```

```
    internal GenericActionPipelinePolicy(
                        Action<PipelineMessage> processMessageAction)
    {
        this._processMessageAction = processMessageAction;
    }

    public override void Process(PipelineMessage message,
                  IReadOnlyList<PipelinePolicy> pipeline, int currentIndex)
    {
        this._processMessageAction(message);
        if (currentIndex < pipeline.Count - 1)
        {
            pipeline[currentIndex + 1].Process(message,
                                                pipeline, currentIndex + 1);
        }
    }
..
}
```

6: Abstractions

Overview

The SemanticKernel.Abstractions assembly is the glue that holds all of Semantic Kernel together. It defines key interfaces, argument types, extension methods, builders and exceptions that are used throughout the rest of Semantic Kernel and often as base types for parameters passed to and from client code. It is wise for application developers to spend time to become familiar with the abstractions, because often when debugging and you *"go to definition"* in your IDE, it is into Abstractions that you will be directed.

Extension Methods

It should be noted that the Semantic Kernel source tree uses extension methods extensively, usually located in source files whose names end with "Extensions". Hence when searching for an implementation of a particular piece of functionality, make sure to look in these. As an example, note that `Kernel.Build()` method is not implemented anywhere in Abstractions, instead it is an extension method defined in Core at the bottom of:

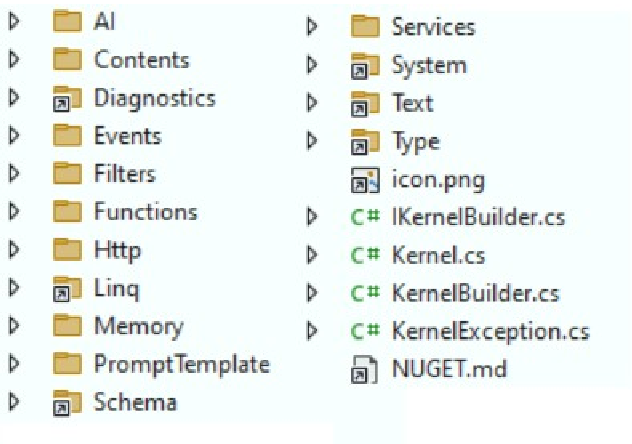
- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/KernelExtensions.cs

Source Layout

The main Abstractions source folder:

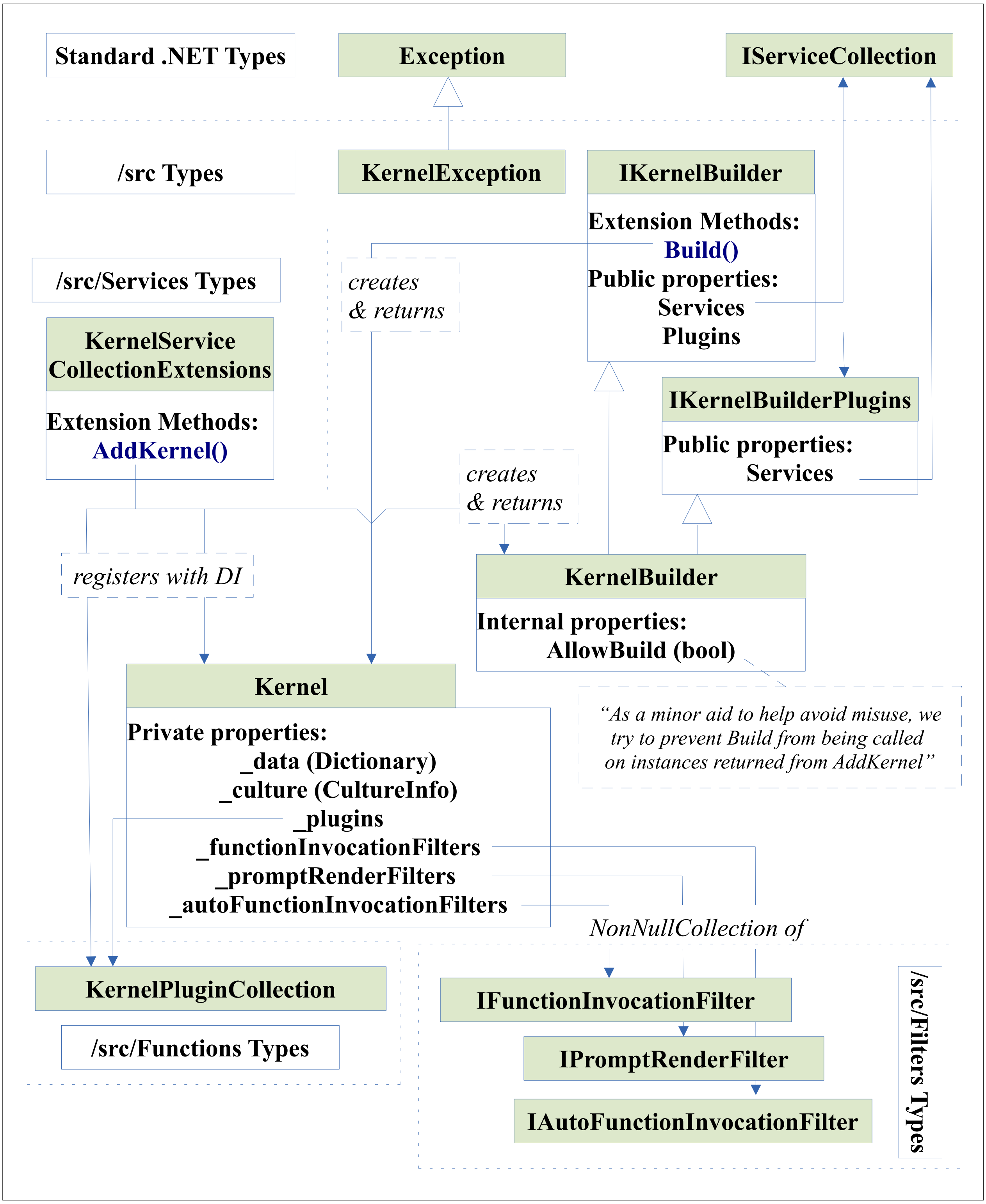
- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/SemanticKernel.Abstractions

contains the following sub-folders and files:



src/SemanticKernel.Abstractions folder

Let's start by looking at the files directly in the src sub-folder, which have these types:



The KernelException.cs file:

- KernelException.cs

contains a definition for the KernelException class, which derives from Exception (note the comment [highlight added] about telemetry):

```
/// <summary>
/// Represents the base exception from which all Semantic Kernel
/// exceptions derive.
/// </summary>
/// <remarks>
/// Instances of this class optionally contain telemetry information in the
/// Exception.Data property using keys that are consistent with the
/// OpenTelemetry standard.
/// See https://opentelemetry.io/ for more information.
/// </remarks>
public class KernelException : Exception
{
  ..
    /// <summary>
    /// Initializes a new instance of the <see cref="KernelException"/> class
    /// with a specified error message and a reference to the inner exception
    /// that is the cause of this exception.
    /// </summary>
    /// <param name="message">The error message that explains the reason for
    /// the exception.</param>
    /// <param name="innerException">The exception that is the cause of the
    /// current exception, or a null reference if no inner exception is
    /// specified.</param>
    public KernelException(string? message, Exception? innerException)
                                       : base(message, innerException)
    {
    }
}
```

The IKernelBuilder.cs file:

- IKernelBuilder.cs

is an interface representing a Builder design pattern for a kernel object:

```
/// <summary>Provides a builder for constructing instances of
/// <see cref="Kernel"/>.</summary>
public interface IkernelBuilder {
    /// <summary>Gets the collection of services to be built into
    /// the <see cref="Kernel"/>.</summary>
    IServiceCollection Services { get; }

    /// <summary>Gets a builder for adding collections as singletons
    /// to <see cref="Services"/>.</summary>
    IKernelBuilderPlugins Plugins { get; }
}
/// <summary>Provides a builder for adding plugins as singletons to
/// a service collection.</summary>
public interface IKernelBuilderPlugins
{
    /// <summary>Gets the collection of services to which plugins
    /// should be added.</summary>
    IServiceCollection Services { get; }
```

```
}
```

It is based around the idea of a service collection – that we see in other parts of .NET, such as ASP.NET Core.

The KernelBuilder file provides one implementation of this builder interface:

- KernelBuilder.cs

and we see it is not complex. It is an internal sealed class. It is essentially managing a service collection and a plugin collection:

```
/// <summary>Provides a builder for constructing instances
/// of <see cref="Kernel"/>.</summary>
internal sealed class KernelBuilder : IKernelBuilder, IKernelBuilderPlugins
{
    /// <summary>The collection of services to be available through
    /// the <see cref="Kernel"/>.</summary>
    private IServiceCollection? _services;

    /// <summary>Initializes a new instance of
    /// the <see cref="KernelBuilder"/>.</summary>
    public KernelBuilder()
    {
        this.AllowBuild = true;
    }

    /// <summary>Initializes a new instance of
    /// the <see cref="KernelBuilder"/>.</summary>
    /// <param name="services">
    /// The <see cref="IServiceCollection"/> to wrap and use for
    /// building the <see cref="Kernel"/>.
    /// </param>
    public KernelBuilder(IServiceCollection services)
    {
        Verify.NotNull(services);

        this._services = services;
    }

    /// <summary>Whether to allow a call to Build.</summary>
    /// <remarks>As a minor aid to help avoid misuse, we try to prevent Build
    /// from being called on instances returned from AddKernel.</remarks>
    internal bool AllowBuild { get; }

    /// <summary>Gets the collection of services to be built into
    /// the <see cref="Kernel"/>.</summary>
    public IServiceCollection Services =>
                                this._services ??= new ServiceCollection();

    /// <summary>Gets a builder for plugins to be built as services
    /// into the <see cref="Kernel"/>.</summary>
    public IKernelBuilderPlugins Plugins => this;
}
```

It does have an extra boolean property, `AllowBuild`, whose purpose is explained in the highlighted remark (in the first of the two constructors here, `AllowBuild` is set to true; in the second it is not set at all, which means the default is used, which for a Boolean is false). So it is important to note when the constructor with no parameter is called, `AllowBuild` ends ups being true, whereas when the constructor with a parameter of a service collection is used, `AllowBuild` ends up being false. This is important to note, there are two different ways of creating a kernel builder, one with `CreateBuilder()` form Kernel (which uses the constructor with no parameter, so AllowBuild is true):

```
public static IKernelBuilder CreateBuilder() => new KernelBuilder();
```

and another is the `AddKernel` extension method, which does pass in a services collection to the constructor, thus `AllowBuild` is false.

Then in the `Build()` extension method for `IKernelBuilder`, we see this important check:

```
public static Kernel Build(this IKernelBuilder builder)
    {
        Verify.NotNull(builder);

        if (builder is KernelBuilder kb && !kb.AllowBuild)
        {
          throw new InvalidOperationException(
            "Build is not permitted on instances returned from AddKernel. " +
            "Resolve the Kernel from the service provider.");
        }
```

Moving on to the `Kernel` class, it is implemented in:

- Kernel.cs

It is responsible for state management and note it is passed to every function invocaton and service call. It manages these private fields:

```
/// <summary>
/// Provides state for use throughout a Semantic Kernel workload.
/// </summary>
/// <remarks>
/// An instance of <see cref="Kernel"/> is passed through to every function
/// invocation and service call throughout the system, providing to each
/// the ability to access shared state and services.
/// </remarks>
public sealed class Kernel
{
    private Dictionary<string, object?>? _data;
    private CultureInfo _culture = CultureInfo.InvariantCulture;
    private KernelPluginCollection? _plugins;
    private NonNullCollection<IFunctionInvocationFilter>?
                                    _functionInvocationFilters;
    private NonNullCollection<IPromptRenderFilter>? _promptRenderFilters;
    private NonNullCollection<IAutoFunctionInvocationFilter>?
                                    _autoFunctionInvocationFilters;
```

Its constructor is concerned with the following: (1) initializing `Services`, (2) initializing plugins from the `KernelPluginCollection` of the supplied services parameter or from the supplied plugins parameter, (3) if no plugins are found to initialize from the `KernelPlugin` of the supplied services parameter and (4) to call `addFilters()`:

```
public Kernel(
    IServiceProvider? services = null,
    KernelPluginCollection? plugins = null) {
1    this.Services = services ?? EmptyServiceProvider.Instance;
2    this._plugins = plugins ??
                  this.Services.GetService<KernelPluginCollection>();
3    if (this._plugins is null) {
        IEnumerable<KernelPlugin> registeredPlugins =
                            this.Services.GetServices<KernelPlugin>();
      if (IsNotEmpty(registeredPlugins)) {
          this._plugins = new(registeredPlugins); }
    }
4    this.AddFilters();
}
```

Its `CreateBuilder()` method simply returns a new instance of `KernelBuilder`:

```
public static IKernelBuilder CreateBuilder() => new KernelBuilder();
```

It provides three kinds of filters (how filters work will be explored shortly):

```
public IList<IFunctionInvocationFilter> FunctionInvocationFilters =>
    this._functionInvocationFilters ??
      Interlocked.CompareExchange(ref this._functionInvocationFilters,
      [], null) ?? this._functionInvocationFilters;

public IList<IPromptRenderFilter> PromptRenderFilters =>
    this._promptRenderFilters ??
    Interlocked.CompareExchange(ref this._promptRenderFilters, [], null) ??
    this._promptRenderFilters;

public IList<IAutoFunctionInvocationFilter> AutoFunctionInvocationFilters =>
    this._autoFunctionInvocationFilters ??
    Interlocked.CompareExchange(ref this._autoFunctionInvocationFilters, [],
    null) ?? this._autoFunctionInvocationFilters;
```

The `Interlocked` methods are used to ensure thread safety.

For services, it offers:

```
public IEnumerable<T> GetAllServices<T>() where T : class
```

which returns an enumerable of matching services and

```
public T GetRequiredService<T>(object? serviceKey = null) where T : class
```

which either returns the matching service instance (if found) or throws an exception.

For filters, the `addFilters()` method initializes `this._functionInvocationFilters`, `this._promptRenderFilters` and `this._autoFunctionInvocationFilters`.

The Kernel class provides the `InvokeAsync()/InvokeStreamingAsync()` methods to invoke a kernel function. We can see they simply call the invoke method fo the `KernelFunction` itself.

```
    /// <summary>
    /// Invokes the <see cref="KernelFunction"/>.
    /// </summary>
    /// <param name="function">The <see cref="KernelFunction"/> to
    /// invoke.</param>
```

```
    /// <param name="arguments">The arguments to pass to the function's
    /// invocation, including any
    /// <see cref="PromptExecutionSettings"/>.</param>
    /// <param name="cancellationToken">The <see cref="CancellationToken"/>
    /// to monitor for cancellation requests. The default is
    /// <see cref="CancellationToken.None"/>.</param>
    /// <returns>The result of the function's execution.</returns>
    /// <exception cref="ArgumentNullException"><paramref name="function"/>
    /// is null.</exception>
    /// <exception cref="KernelFunctionCanceledException">The
    /// <see cref="KernelFunction"/>'s invocation was canceled.</exception>
    /// <remarks>
    /// This behaves identically to invoking the specified
    /// <paramref name="function"/> with this <see cref="Kernel"/> as its
    /// <see cref="Kernel"/> argument.
    /// </remarks>
    public Task<FunctionResult> InvokeAsync(
        KernelFunction function,
        KernelArguments? arguments = null,
        CancellationToken cancellationToken = default)
    {
        Verify.NotNull(function);

        return function.InvokeAsync(this, arguments, cancellationToken);
    }
```

To see what happens next, we need to explore `KernelFunction.InvokeAsync()`, which we will shortly. It calls back into `Kernel.OnFunctionInvocationAsync()`, which gives the kernel a chance to runs its filters:

```
     internal async Task<FunctionInvocationContext> OnFunctionInvocationAsync(
        KernelFunction function,
        KernelArguments arguments,
        FunctionResult functionResult,
        Func<FunctionInvocationContext, Task> functionCallback,
        CancellationToken cancellationToken)
    {
        FunctionInvocationContext context =
                        new(this, function, arguments, functionResult)
        {
            CancellationToken = cancellationToken
        };

        await InvokeFilterOrFunctionAsync(
               this._functionInvocationFilters, functionCallback,context)
               .ConfigureAwait(false);
        return context;
    }
```

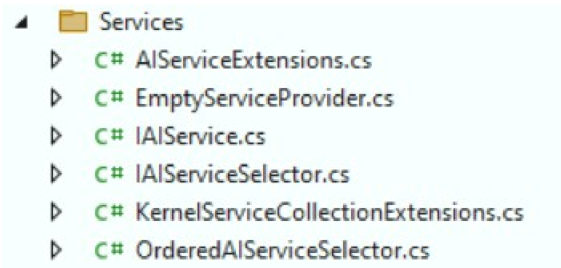
`InvokeFilterOrFunctionAsync()` is called recursively to run each filter (note how index is incremented each time):

```
/// <summary>
/// This method will execute filters and kernel function recursively.
/// If there are no registered filters, just kernel function will be
/// executed.
/// If there are registered filters, filter on <paramref name="index"/>
/// position will be executed.
/// Second parameter of filter is callback. It can be either filter on
///  <paramref name="index"/> + 1 position or kernel function if there are no
```

```
/// remaining filters to execute.
/// Kernel function will be always executed as last step after all filters.
/// </summary>
private static async Task InvokeFilterOrFunctionAsync(
  NonNullCollection<IFunctionInvocationFilter>? functionFilters,
  Func<FunctionInvocationContext, Task> functionCallback,
  FunctionInvocationContext context,
  int index = 0)
{
  if (functionFilters is { Count: > 0 } && index < functionFilters.Count)
  {
    await functionFilters[index].OnFunctionInvocationAsync(
          context, (context) => InvokeFilterOrFunctionAsync(functionFilters,
          functionCallback, context, index + 1)).ConfigureAwait(false);
  }
  else
  {
    await functionCallback(context).ConfigureAwait(false);
  }
}
```

src/SemanticKernel.Abstractions/Services folder

This subfolder contains types to work with AI Services:



This is important, since one of the primary goals of semantic Kernel is to work with AI Services and application developers need to work with various AI services when building their AI solutions. As background to exploring this folder, read up on how dependency injection works with Semantic Kernel and understand this sample:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/Demos/HomeAutomation

The IAIService.cs file:

- IAIService.cs

contains this simple interface:

```
/// <summary>
/// Represents an AI service.
/// </summary>
public interface IAIService
{
    /// <summary>
    /// Gets the AI service attributes.
    /// </summary>
    IReadOnlyDictionary<string, object?> Attributes { get; }
}
```

which defines the `Attributes` dictionary.

It is the role of a service provider to implement `IServiceProvider`. The EmptyServiceProvider.cs file:

- EmptyServiceProvider.cs

provides one implementation, which is empty:

```
/// <summary>Empty <see cref="IServiceProvider"/> implementation that returns
null from all <see cref="IServiceProvider.GetService"/> calls.</summary>
internal sealed class EmptyServiceProvider : IServiceProvider,
IKeyedServiceProvider {
    private static readonly ConcurrentDictionary<Type, object?> s_results =
new();
    /// <summary>Singleton instance of <see
cref="EmptyServiceProvider"/>.</summary>
    public static IServiceProvider Instance { get; } = new
EmptyServiceProvider();
    public object? GetService(Type serviceType) =>
s_results.GetOrAdd(serviceType, GetEmpty);
    public object? GetKeyedService(Type serviceType, object? serviceKey) =>
s_results.GetOrAdd(serviceType, GetEmpty);
    ...
}
```

The AIServiceExtensions class:

- AIServiceExtensions.cs

defines the `AIServiceExtensions` public static class. This starts with some static properties and related getters:

```
public static string ModelIdKey => "ModelId";
public static string EndpointKey => "Endpoint";
public static string ApiVersionKey => "ApiVersion";
```

The important method is `SelectAIService`, which uses a selector to choose which service to resolve:

```
/// <summary>
/// Resolves an <see cref="IAIService"/> and  associated
/// <see cref="PromptExecutionSettings"/> from the specified
/// <see cref="Kernel"/> based on a <see cref="KernelFunction"/>
/// and associated <see cref="KernelArguments"/>.
/// </summary>
public static (T?, PromptExecutionSettings?) SelectAIService<T>(
    this IAIServiceSelector selector,
    Kernel kernel,
    KernelFunction function,
    KernelArguments arguments) where T : class, IAIService
{
  ...
    if (selector.TrySelectAIService<T>(
        kernel, function, arguments,
        out T? service, out PromptExecutionSettings? Settings)) {
        return (service, settings);
    }
    var message = new StringBuilder().Append("Required service of type ")
                    .Append(typeof(T)).Append(" not registered.");
    if (function.ExecutionSettings is not null) {
        string serviceIds =
               string.Join("|", function.ExecutionSettings.Keys);
```

```
            if (!string.IsNullOrEmpty(serviceIds))              {
                message.Append(
                    " Expected serviceIds: ").Append(serviceIds).Append('.');
            }
            string modelIds = string.Join("|",
           function.ExecutionSettings.Values.Select(model => model.ModelId));
            if (!string.IsNullOrEmpty(modelIds))
            {
                message.Append(
                     " Expected modelIds: ").Append(modelIds).Append('.');
            }
        }
        throw new KernelException(message.ToString());
    }
}
```

The `KernelServiceCollectionExtensions` type in:

- [Services/KernelServiceCollectionExtensions.cs]()

defines one very important extension method, `AddKernel()`.

```
/// <summary>Extension methods for interacting with <see
cref="Kernel"/>.</summary>
public static class KernelServiceCollectionExtensions
{
    /// <summary>Adds a <see cref="KernelPluginCollection"/> and
    /// <see cref="Kernel"/> services to the services collection.</summary>
    /// <param name="services">The service collection.</param>
    /// <returns>
    /// A <see cref="IKernelBuilder"/> that can be used to add additional
    /// services to the same <see cref="IServiceCollection"/>.
    /// </returns>
    /// <remarks>
    /// Both services are registered as transient, as both objects are
    /// mutable.
    /// </remarks>
    public static IKernelBuilder AddKernel(this IServiceCollection services)
    {
        Verify.NotNull(services);

        // Register a KernelPluginCollection to be populated with any
        // IKernelPlugins that have been
        // directly registered in DI. It's transient because the Kernel will
        // store the collection
        // directly, and we don't want two Kernel instances to hold on to the
        // same mutable collection.
        services.AddTransient<KernelPluginCollection>();

        // Register the Kernel as transient. It's mutable and expected to be
        // mutated by consumers,
        // such as via adding event handlers, adding plugins, storing state
        // in its Data collection, etc.
        services.AddTransient<Kernel>();

        // Create and return a builder that can be used for adding services
        // and plugins to the IServiceCollection.
        return new KernelBuilder(services);
    }
}
```

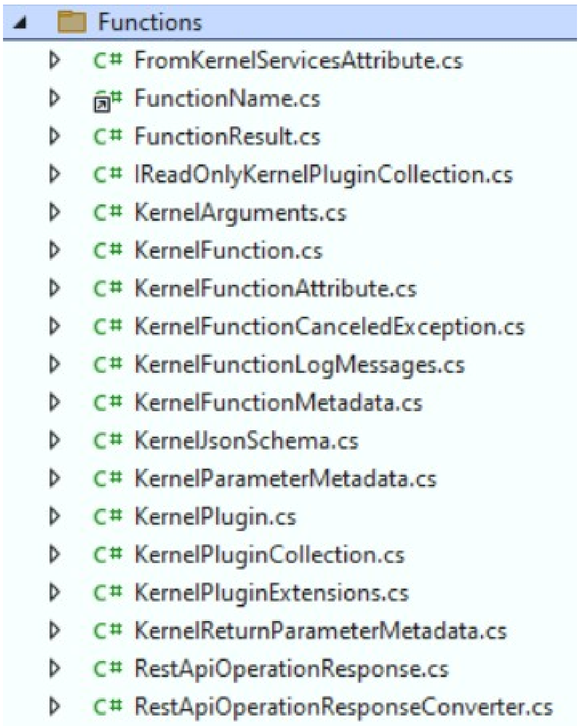
Two types, `Kernel` and `KernelPluginCollection`, are being registered as transients with DI (means they will be created each time upon demand). Also a new `KernelBuilder` is being instantiated (witeh service collection as parameter) and this `KernelBuilder` instance is being returned, not the `Kernel` instance (despite the method being called `AddKernel`).

src/SemanticKernel.Abstractions/Functions folder

The Functions sub-folder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/SemanticKernel.Abstractions/Functions

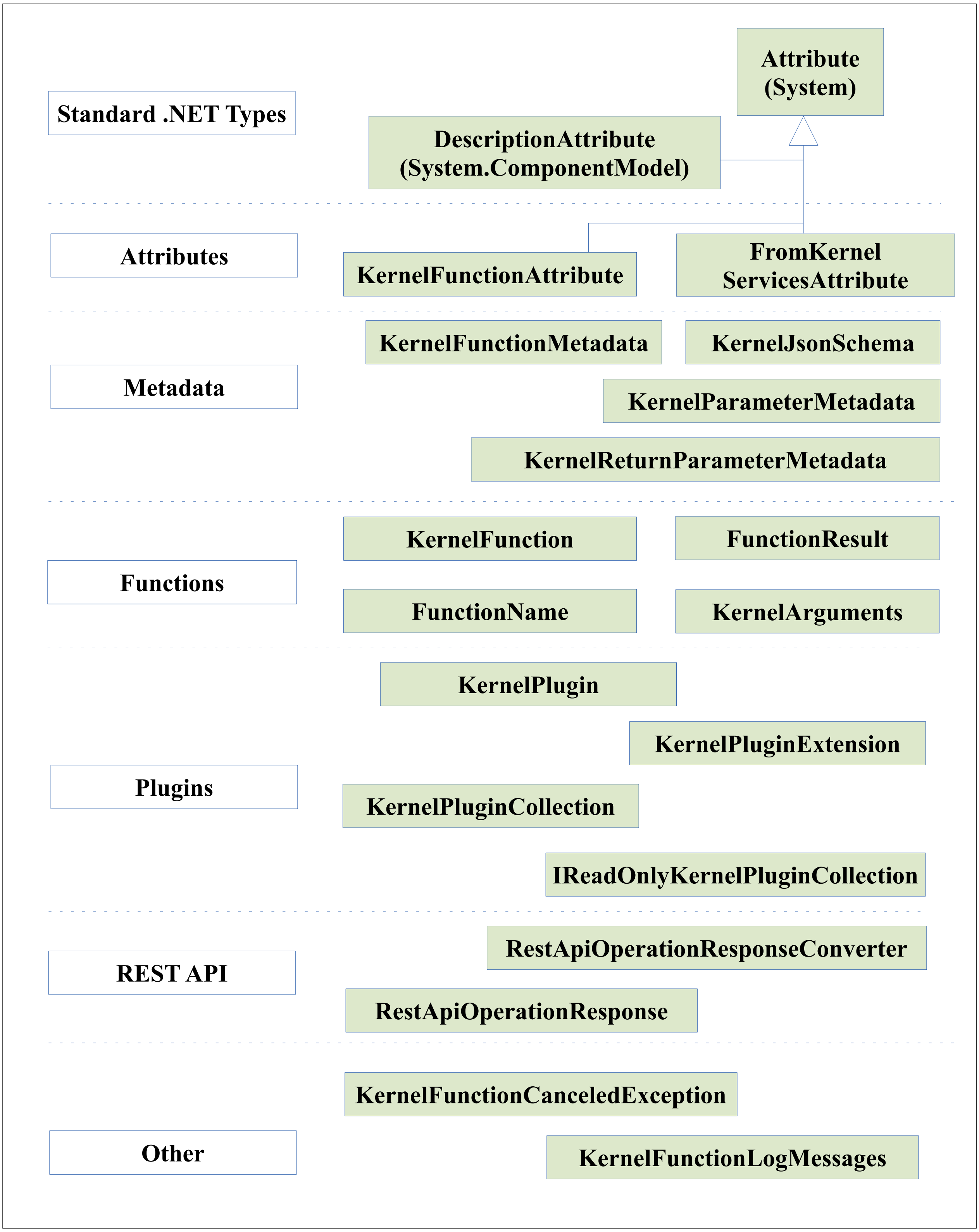
contains these files:



In the world of Semantic Kernel, there are two kinds of kernel functions: method- (also called native) and prompt-based.

Method kernel functions represent a method that the application developer writes in programming code and that is in various ways “decorated” with additional information so that it integrates well with large-language models. Using a technique known as function calling, we wish to have the LLM call the function as needed. Hence we need to be able to provide the LLM with a precise descritpion of certain characteristics of the function and that is what the types in this sub-folder provide.

Prompt kernel functions are based on textual prompts written in natural language (Perhaps using a templating syntax such as handlebars).



We can divide the functionality provided by the Abstractions/Functions folder into:

- Attributes (KernelFunctionAttribute indicates a method is to be treated as a kernel function, FromKernelServicesAttribute indicates an argument is to be taken from Services and the .NET standard DescriptionAttribute is used to provide a textual description of the function – used by the LLM when deciding if this function should be called)

- Metadata (KernelFunction, Parameter, ReturnParameter)
- Functions (names, result, arguments)
- Plugins (collections of functions and more)
- RestAPIOperations

Attributes

The `KernelFunctionAttribute`:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Functions/KernelFunctionAttribute.cs

is used to decorate methods to indicate they are part of the plugin and available to be called by the kernel:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public sealed class KernelFunctionAttribute : Attribute
{
    /// <summary>Initializes the attribute.</summary>
    public KernelFunctionAttribute() { }

    /// <summary>Initializes the attribute.</summary>
    /// <param name="name">The name to use for the function.</param>
    public KernelFunctionAttribute(string? name) => this.Name = name;

    /// <summary>Gets the function's name.</summary>
    /// <remarks>If null, a name will based on the name of the attributed
    /// method will be used.</remarks>
    public string? Name { get; }
}
```

Note `AttributeUsage` sets target to `Method` and `AllowMultiple` to `false`.

The `FromKernelServicesAttribute`:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Functions/FromKernelServicesAttribute.cs

is used to state a kernel function parameter is to come from the kernel's services:

```
/// <summary>
/// Specifies that an argument to a <see cref="KernelFunction"/> should be
/// supplied from the associated <see cref="Kernel"/>'s
/// <see cref="Kernel.Services"/> rather than from
/// <see cref="KernelArguments"/>.
/// </summary>
[AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false)]
public sealed class FromKernelServicesAttribute : Attribute
{
    /// <summary>Initializes the attribute.</summary>
    public FromKernelServicesAttribute() { }

    /// <summary>Initializes the attribute with the specified
    /// service key.</summary>
    /// <param name="serviceKey">The optional service key to use when
    /// resolving a service.</param>
    public FromKernelServicesAttribute(object? serviceKey)
```

```
                                              => this.ServiceKey = serviceKey;

        /// <summary>Gets the key to use when searching
        /// <see cref="Kernel.Services"/>.</summary>
        public object? ServiceKey { get; }
    }
```

Note `AttributeUsage` sets target to `Parameter` and `AllowMultiple` to `false`.

Metadata

The three metadata types represent metadata for the kernel function itself, for its parameters and for its return parameter. They make use of the `KernelJsonSchema` type to provide a JSON Schema for a parameter type.

The `KernelFunctionMetadata` type has properties for the name, description, list of parameters, the return parameter and optionally other properties:

```
    /// <summary>
    /// Provides read-only metadata for a <see cref="KernelFunction"/>.
    /// </summary>
    public sealed class KernelFunctionMetadata
    {
        /// <summary>The name of the function.</summary>
        private string _name = string.Empty;
        /// <summary>The description of the function.</summary>
        private string _description = string.Empty;
        /// <summary>The function's parameters.</summary>
        private IReadOnlyList<KernelParameterMetadata> _parameters = [];
        /// <summary>The function's return parameter.</summary>
        private KernelReturnParameterMetadata? _returnParameter;
        /// <summary>Optional metadata in addition to the named properties
        ///  already available on this class.</summary>
        private ReadOnlyDictionary<string, object?>? _additionalProperties;
        …
    }
```

Functions

We see FunctionName.cs is actually defined in InternalUtilities (note additional small icon) and all the other files are defined locally.

FunctionName.cs contains an implementation of the `FunctionName` class, which offers getters for two string properties – `PluginName` and `Name` (the actual function name).

These are initialized in the constructor:

```
    public FunctionName(string name, string? pluginName = null)
    {
        Verify.NotNull(name);

        this.Name = name;
        this.PluginName = pluginName;
    }
```

Helper methods are included to convert to and from a fully qualified name:

```
    public static string ToFullyQualifiedName(
                             string functionName,
```

```
                           string? pluginName = null,
                           string functionNameSeparator = "-")
{
    return string.IsNullOrEmpty(pluginName) ? functionName :
                        $"{pluginName}{functionNameSeparator}{functionName}";
}
```

As you can see, the fully qualified name is simply the plugin name, separator if provided or else '-') and the function name). Once application using Semantic Kernel become moe complex and perhaps use many plugins and many functions, the need to distinguish between functions (e.g. for logging) becomes more important.

The `KernelArguments` class:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Functions/KernelArguments.cs

manages dictionaries of arguments used in kernel functions:

```
public sealed class KernelArguments : IDictionary<string, object?>,
IReadOnlyDictionary<string, object?>
{
    /// <summary>Dictionary of name/values for all the arguments in the
    /// instance.</summary>
    private readonly Dictionary<string, object?> _arguments;
    private IReadOnlyDictionary<string,
                     PromptExecutionSettings>? _executionSettings;
    ...
    /// <summary>
    /// Initializes a new instance of the <see cref="KernelArguments"/> class
    /// with the specified AI execution settings.
    /// </summary>
    /// <param name="executionSettings">The prompt execution
    /// settings.</param>
    public KernelArguments(
              IEnumerable<PromptExecutionSettings>? executionSettings)
    {
       this._arguments = new(StringComparer.OrdinalIgnoreCase);
       if (executionSettings is not null)
       {
           var newExecutionSettings =
                 new Dictionary<string, PromptExecutionSettings>();
           foreach (var settings in executionSettings)
           {
               var targetServiceId = settings.ServiceId ??
                                 PromptExecutionSettings.DefaultServiceId;
               ...
               newExecutionSettings[targetServiceId] = settings;
           }
           this.ExecutionSettings = newExecutionSettings;
       }
   }
   ///
}
```

The `FunctionResult` class:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Functions/FunctionResult.cs

defines the result of an invocation:

```
/// <summary>
/// Represents the result of a <see cref="KernelFunction"/> invocation.
/// </summary>
public sealed class FunctionResult
{
    /// <summary>
    /// Initializes a new instance of the <see cref="FunctionResult"/> class.
    /// </summary>
    /// <param name="function">The <see cref="KernelFunction"/> whose result
    /// is represented by this instance.</param>
    /// <param name="value">The resulting object of the function's
    /// invocation.</param>
    /// <param name="culture">The culture configured on the
    /// <see cref="Kernel"/> that executed the function.</param>
    /// <param name="metadata">Metadata associated with the function's
    /// execution</param>
    public FunctionResult(
      KernelFunction function,
      object? value = null,
      CultureInfo? culture = null,
      IReadOnlyDictionary<string, object?>? metadata = null)
    {
        Verify.NotNull(function);

        this.Function = function;
        this.Value = value;
        this.Culture = culture ?? CultureInfo.InvariantCulture;
        this.Metadata = metadata;
    }
…
}
```

`KernelFunction` is an abstract class:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Functions/KernelFunction.cs

that represents the kernel function itself. Its constructors are marked as internal. We will shortly see when exploring Core that two internal concrete classes derive from `KernelFunction` called `KernelFunctionFromMethod` and `KernelFunctionfromPrompt`, which provide the implementation for what the Semantic Kernel documentation call native functions and prompt functions.

```
/// <summary>
/// Represents a function that can be invoked as part of a Semantic
/// Kernel workload.
/// </summary>
public abstract class KernelFunction
{
    ...

    /// <summary>
```

```
        /// Initializes a new instance of the <see cref="KernelFunction"/> class.
        /// </summary>
        /// <param name="name">A name of the function to use as its
        /// <see cref="KernelFunction.Name"/>.</param>
        /// <param name="pluginName">The name of the plugin this function
        /// instance has been added to.</param>
        /// <param name="description">The description of the function to use as
        /// its <see cref="KernelFunction.Description"/>.</param>
        /// <param name="parameters">The metadata describing the parameters to
        /// the function.</param>
        /// <param name="returnParameter">The metadata describing the return
        /// parameter of the function.</param>
        /// <param name="executionSettings">
        /// The <see cref="PromptExecutionSettings"/> to use with the function.
        /// These will apply unless they've been
        /// overridden by settings passed into the invocation of the function.
        /// </param>
        /// <param name="additionalMetadata">Properties/metadata associated with
        /// the function itself rather than its parameters and return
        /// type.</param>
        internal KernelFunction(
                  string name,
                  string? pluginName,
                  string description,
                  IReadOnlyList<KernelParameterMetadata> parameters,
                  KernelReturnParameterMetadata? returnParameter = null,
                  Dictionary<string, PromptExecutionSettings>?
                                               executionSettings = null,
                  ReadOnlyDictionary<string, object?>? additionalMetadata = null)
        {
      }
      ...
   }
```

`KernelFunction` has a couple of implementations of the `InvokeAsync()` method. This contains logging, timing and cancellation token handling – when we strip away all that, we are left with the following, which is how kernel functions actually invoke the underlying functionality:

```
        public async Task<FunctionResult> InvokeAsync(
            Kernel kernel,
            KernelArguments? arguments = null,
            CancellationToken cancellationToken = default)
        {
            FunctionResult functionResult = new(this, culture: kernel.Culture);
            try
            {
                var invocationContext =
                          await kernel.OnFunctionInvocationAsync(
                           this, arguments, functionResult, async (context) =>
                {
                      context.Result = functionResult =
                         await this.InvokeCoreAsync(kernel, context.Arguments,
                                        cancellationToken).ConfigureAwait(false);
                }, cancellationToken).ConfigureAwait(false);

                functionResult = invocationContext.Result;
                return functionResult;
            }
```

```
    }
```

`InvokeCoreAsync()` is where the actual functionality of the `KernelFunction` is invoked. It is defined as abstract and protected:

```
/// <summary>
/// Invokes the <see cref="KernelFunction"/>.
/// </summary>
/// <param name="kernel">The <see cref="Kernel"/> containing services,
/// plugins, and other state for use throughout the operation.</param>
/// <param name="arguments">The arguments to pass to the function's
/// invocation, including any <see cref="PromptExecutionSettings"/>.</param>
/// <returns>The updated context, potentially a new one if context switching
/// is implemented.</returns>
/// <param name="cancellationToken">The <see cref="CancellationToken"/> to
/// monitor for cancellation requests. The default is
/// <see cref="CancellationToken.None"/>.</param>

    protected abstract ValueTask<FunctionResult> InvokeCoreAsync(
        Kernel kernel,
        KernelArguments arguments,
        CancellationToken cancellationToken);
```

If you are looking for concrete implementations, for native functions, see:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/Functions/KernelFunctionFromMethod.cs

where `InvokeCoreAsync()` is defined as:

```
protected override ValueTask<FunctionResult> InvokeCoreAsync(
        Kernel kernel,
        KernelArguments arguments,
        CancellationToken cancellationToken)
    {
        return this._function(kernel, this, arguments, cancellationToken);
    }
```

and for prompt functions, see :

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/Functions/KernelFunctionFromPrompt.cs

where `InvokeCoreAsync()` is defined as:

```
  protected override async ValueTask<FunctionResult> InvokeCoreAsync(
        Kernel kernel,
        KernelArguments arguments,
        CancellationToken cancellationToken = default)
  {
        this.AddDefaultValues(arguments);

        var promptRenderingResult =
                  await this.RenderPromptAsync(kernel, arguments,
                               cancellationToken).ConfigureAwait(false);
        ...
   }
```

Its metadata is defined as:

```
public IReadOnlyDictionary<string, object?>? Metadata { get; init; }
```

Plugins

`KernelPlugin` is an abstract class that manages an eunerable of KernelFunctions:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Functions/KernelPlugin.cs

It also has properties for name and description. Note the highlighting part of the remarks about it being a many-to-many relationship:

```
/// <remarks>
/// A plugin is a named read-only collection of functions. There is a
/// many-to-many relationship between plugins and functions: a plugin
/// may contain any number of functions, and a function may exist in
/// any number of plugins.
/// </remarks>
public abstract class KernelPlugin : IEnumerable<KernelFunction>
{
    /// <summary>Initializes the new plugin from the provided name,
    ///  description, and function collection.</summary>
    protected KernelPlugin(string name, string? description = null)
    {
        Verify.ValidPluginName(name);

        this.Name = name;
        this.Description =
            !string.IsNullOrWhiteSpace(description) ? description! : "";
    }

    /// <summary>Gets the name of the plugin.</summary>
    public string Name { get; }

    /// <summary>Gets a description of the plugin.</summary>
    public string Description { get; }

    …

}
```

It also provides access to KernelFunctions in a few different ways, such as:

```
  /// <summary>Gets the function in the plugin with the
    /// specified name.</summary>
    /// <param name="functionName">The name of the function.</param>
    /// <returns>The function.</returns>
    /// <exception cref="KeyNotFoundException">The plugin does not contain a
    ///   function with the specified name.</exception>
    public KernelFunction this[string functionName] =>
        this.TryGetFunction(functionName, out KernelFunction? Function) ?
          function :
          throw new KeyNotFoundException(
$"The plugin does not contain a function with the specified name. Plugin name
- '{this.Name}', function name - '{functionName}'.");
```

KernelPluginExtensions.cs contains some extension methods to extracts functions from KernelPlugins.

The `KernelPlugin` class is abstract – note that one important feature it omits is the enumerator implementation:

```
public abstract IEnumerator<KernelFunction> GetEnumerator();
```

Quite often the concrete implementation application developers use will be:

```
internal sealed class DefaultKernelPlugin : KernelPlugin
{
   ...
   private readonly Dictionary<string, KernelFunction> _functions;

   public override IEnumerator<KernelFunction> GetEnumerator()
                            => this._functions.Values.GetEnumerator();
}
```

defined in:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/Functions/DefaultKernelPlugin.cs

which uses a dictionary and constructed by:

```
public static partial class KernelPluginFactory
{
   ...
   public static KernelPlugin CreateFromFunctions(
               string pluginName,
               string? description = null,
               IEnumerable<KernelFunction>? functions = null)
   => new DefaultKernelPlugin(pluginName, description, functions);
}
```

defined in:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/Functions/KernelPluginFactory.cs

src/SemanticKernel.Abstractions/Filters folder

For a succinct explanation of the purpose of filters, let’s just ask Copilot:

You

What are filters in Semantic Kernel?

Copilot

Filters in Semantic Kernel are mechanisms that allow you to control and modify the behavior of functions and prompts within the kernel. They are particularly useful for ensuring security, handling exceptions, and customizing the execution flow of functions.

There are two main types of filters in Semantic Kernel:

1. **Function Invocation Filters:** These filters are executed before and after a function is invoked. They allow you to perform actions such as logging, modifying input/output, or handling exceptions. For example, you can measure the execution time of a function by starting a timer before the function invocation and stopping it afterward [1].
2. **Prompt Render Filters:** These filters are used to intercept and modify the prompts before they are sent to the AI model. This can help ensure that sensitive information is not exposed and that prompts are safe and appropriate [1].

Filters provide a flexible way to enhance the functionality and security of your AI applications by allowing you to inject custom logic at various points in the execution flow [1] [2].

The filters subfolders in Abstractions contains these types:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/SemanticKernel.Abstractions/Filters

```
Filters
    AutoFunctionInvocation
        C# AutoFunctionInvocationContext.cs
        C# IAutoFunctionInvocationFilter.cs
    Function
        C# FunctionFilterContext.cs
        C# FunctionInvocationContext.cs
        C# FunctionInvokedContext.cs
        C# FunctionInvokingContext.cs
        C# IFunctionInvocationFilter.cs
    Prompt
        C# IPromptRenderFilter.cs
        C# PromptFilterContext.cs
        C# PromptRenderContext.cs
        C# PromptRenderedContext.cs
        C# PromptRenderingContext.cs
```

The filters are divided into three categories: auto-function invocation, function and prompt. For each, types are provided for the filter and for one or more contexts.

Each of the filters define an interface with one method that is called during a certain activity:

```
/// <summary>
/// Interface for filtering actions during automatic function invocation.
/// </summary>
public interface IAutoFunctionInvocationFilter
{
    /// <summary>
    /// Method which is called asynchronously before automatic function
    /// invocation.
    /// </summary>
    /// <param name="context">Instance of
    /// <see cref="AutoFunctionInvocationContext"/> with automatic
    /// function invocation details.</param>
    /// <param name="next">Delegate to the next filter in pipeline or
    /// function invocation itself. If it's not invoked, next filter won't be
    /// invoked and function invocation will be skipped.</param>
    Task OnAutoFunctionInvocationAsync(AutoFunctionInvocationContext context,
      Func<AutoFunctionInvocationContext, Task> next);
}

/// <summary>
/// Interface for filtering actions during function invocation.
/// </summary>
public interface IFunctionInvocationFilter
{
    /// <summary>
    /// Method which is called asynchronously before function invocation.
    /// </summary>
    /// <param name="context">Instance of
    /// <see cref="FunctionInvocationContext"/> with function invocation
    /// details.</param>
    /// <param name="next">Delegate to the next filter in pipeline or
    /// function itself. If it's not invoked, next filter or function won't
    /// be invoked.</param>
    Task OnFunctionInvocationAsync(FunctionInvocationContext context,
      Func<FunctionInvocationContext, Task> next);
}

/// <summary>
/// Interface for filtering actions during prompt rendering.
/// </summary>
public interface IPromptRenderFilter
{
    /// <summary>
    /// Method which is called asynchronously before prompt rendering.
    /// </summary>
    /// <param name="context">Instance of <see cref="PromptRenderContext"/>
    /// with prompt rendering details.</param>
    /// <param name="next">Delegate to the next filter in pipeline or prompt
    /// rendering operation itself. If it's not invoked, next filter or
    /// prompt rendering won't be invoked.</param>
    Task OnPromptRenderAsync(PromptRenderContext context,
      Func<PromptRenderContext, Task> next);
}
```

One example usage for filters is logging and telemetry. Another is implementing an approval process, whereby the function is only invoked if an approval filter permits it. There is a nice little demo here:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/Demos/FunctionInvocationApproval

showing how to implement this (and shows how to create a filter in general – it is quite simple). This file:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/samples/Demos/FunctionInvocationApproval/Program.cs

contains this code to add a filter called `FunctionInvocationFilter`:

```
var builder = Kernel.CreateBuilder();
...
builder.Services.AddSingleton<IFunctionInvocationFilter,
                                        FunctionInvocationFilter>();
```

and this code to implement a filter called `FunctionInvocationFilter`:

```
public sealed class FunctionInvocationFilter(
                      IFunctionApprovalService approvalService)
                        : IFunctionInvocationFilter
{
    private readonly IFunctionApprovalService
                                   _approvalService = approvalService;

    public async Task OnFunctionInvocationAsync(
                          FunctionInvocationContext context,
                          Func<FunctionInvocationContext, Task> next)
    {
        // Invoke the function only if it's approved.
        if (this._approvalService.IsInvocationApproved(
                               context.Function, context.Arguments))
        {
1           await next(context);
        }
        else
        {
            // Otherwise, return a result that operation was rejected.
            context.Result =
2             new FunctionResult(context.Result, "Operation was rejected.");
        }
    }
}
```

The filter async method takes in two parameters – the context and a next function. If your filtering filtering determines the function call / additional filters are to proceed, then you call await on `next` (1), otherwise you return a `FunctionResult` (2).

The context for each filter type contains useful data that may be needed in the filter implementation. As an example:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Abstractions/Filters/Function/FunctionInvocationContext.cs

contains properties representing the kernel, the function, the arguments and results:

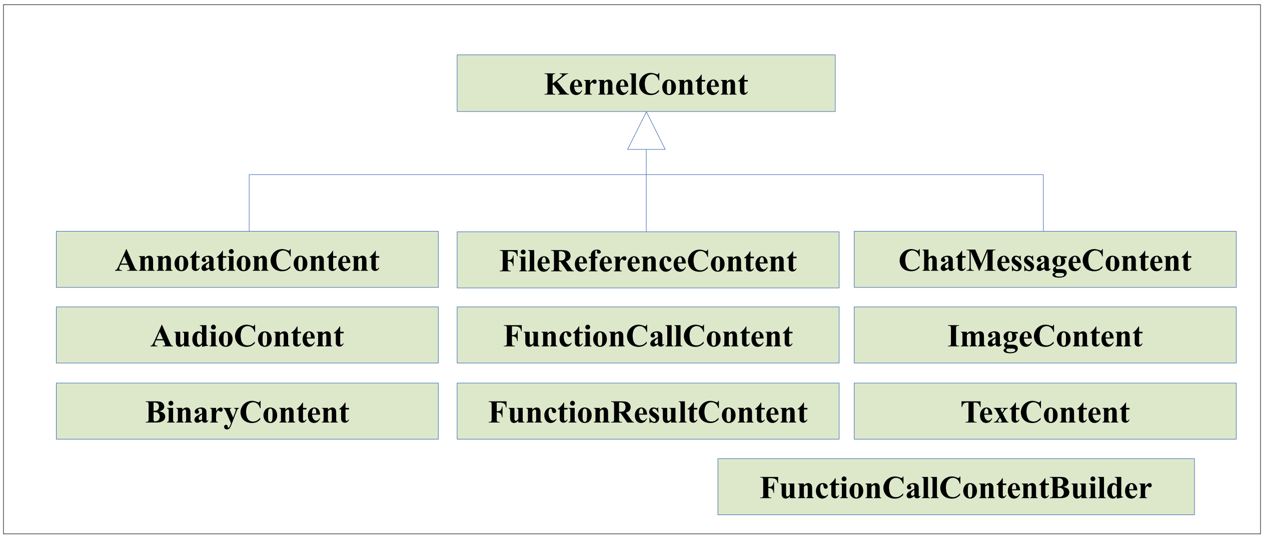
```
/// <summary>
/// Class with data related to function invocation.
/// </summary>
public class FunctionInvocationContext
{
    /// <summary>
    /// Initializes a new instance of the
    /// <see cref="FunctionInvocationContext"/> class.
    /// </summary>
    /// <param name="kernel">The
    /// <see cref="Microsoft.SemanticKernel.Kernel"/> containing services,
    /// plugins, and other state for use throughout the operation.</param>
    /// <param name="function">The <see cref="KernelFunction"/> with which
    /// this filter is associated.</param>
    /// <param name="arguments">The arguments associated with the
    /// operation.</param>
    /// <param name="result">The result of the function's invocation.</param>
    internal FunctionInvocationContext(
               Kernel kernel,
               KernelFunction function,
               KernelArguments arguments,
               FunctionResult result
    ) {
        Verify.NotNull(kernel);
        Verify.NotNull(function);
        Verify.NotNull(arguments);

        this.Kernel = kernel;
        this.Function = function;
        this.Arguments = arguments;
        this.Result = result;
    }
  ...
}
```

src/SemanticKernel.Abstractions/Contents folder

The Contents folder:

- https://github.com/microsoft/semantic-kernel/tree/feature-connectors-openai/dotnet/src/SemanticKernel.Abstractions/Contents

contains representations of content that flows between client and AI services. There is an abstract base class, `KernelContent`, and then a concrete type for each specific kind of content:



The KernelContent class uses type discriminators based on `$type`:

```
[JsonPolymorphic(TypeDiscriminatorPropertyName = "$type")]
[JsonDerivedType(typeof(TextContent), typeDiscriminator: nameof(TextContent))]
[JsonDerivedType(typeof(ImageContent), typeDiscriminator: nameof(ImageContent))]
[JsonDerivedType(typeof(FunctionCallContent), typeDiscriminator: nameof(FunctionCallContent))]
[JsonDerivedType(typeof(FunctionResultContent), typeDiscriminator:
                                                    nameof(FunctionResultContent))]
[JsonDerivedType(typeof(BinaryContent), typeDiscriminator: nameof(BinaryContent))]
[JsonDerivedType(typeof(AudioContent), typeDiscriminator: nameof(AudioContent))]
[JsonDerivedType(typeof(AnnotationContent), typeDiscriminator: nameof(AnnotationContent))]
[JsonDerivedType(typeof(FileReferenceContent), typeDiscriminator: nameof(FileReferenceContent))]
public abstract class KernelContent
{
    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public string? MimeType { get; set; }

    [JsonIgnore]
    public object? InnerContent { get; set; }

    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public string? ModelId { get; set; }

    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public IReadOnlyDictionary<string, object?>? Metadata { get; set; }

    protected KernelContent() { }

    protected KernelContent(object? innerContent, string? modelId = null,
                   IReadOnlyDictionary<string, object?>? metadata = null)
    {
```

```
        this.ModelId = modelId;
        this.InnerContent = innerContent;
        this.Metadata = metadata;
    }
}
```

As an example content, let's look at `ChatMessageContent`:

- https://github.com/microsoft/semantic-kernel/blob/feature-connectors-openai/dotnet/src/SemanticKernel.Abstractions/Contents/ChatMessageContent.cs

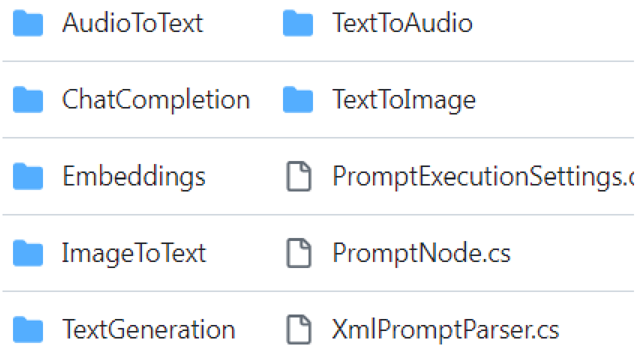
```
/// <summary>
/// Represents chat message content return from a
/// <see cref="IChatCompletionService" /> service.
/// </summary>
public class ChatMessageContent : KernelContent
{
    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public string? AuthorName
    {
        get => this._authorName;
        set =>
          this._authorName = string.IsNullOrWhiteSpace(value) ? null : value;
    }

    public AuthorRole Role { get; set; }

    /// <summary>
    /// Chat message content items
    /// </summary>
    public ChatMessageContentItemCollection Items
    {
        get => this._items ??= [];
        set => this._items = value;
    }

    public ChatMessageContent(
        AuthorRole role,
        ChatMessageContentItemCollection items,
        string? modelId = null,
        object? innerContent = null,
        Encoding? encoding = null,
        IReadOnlyDictionary<string, object?>? metadata = null)
        : base(innerContent, modelId, metadata)
    {
        this.Role = role;
        this._encoding = encoding ?? Encoding.UTF8;
        this._items = items;
    }

    private ChatMessageContentItemCollection? _items;
    private Encoding _encoding;
    private string? _authorName;
}
```

src/SemanticKernel.Abstractions/AI folder

The AI folder:

- https://github.com/microsoft/semantic-kernel/tree/feature-connectors-openai/dotnet/src/SemanticKernel.Abstractions/AI

provides a range of abstractions helpful for using the AI services:



The main AI folder contains files related to prompt handling.

The `PromptExecutionSettings` class manages three properties (including their JSON serialization):

```
private string? _modelId;
private IDictionary<string, object>? _extensionData;
private string? _serviceId;
```

`ExtensionData` is decorated with the `JsonExtensionData` attribute:

```
[JsonExtensionData]
public IDictionary<string, object>? ExtensionData
```

which is used for deserialized JSON properties that do not have a backing field or property in the actual C# type.

The PromptNode.cs file:

- https://github.com/microsoft/semantic-kernel/blob/feature-connectors-openai/dotnet/src/SemanticKernel.Abstractions/AI/PromptNode.cs

represents a node in the prompt:

```
internal sealed class PromptNode(string tagName)
{
    private Dictionary<string, string>? _attributes;
    private List<PromptNode>? _childNodes;
    public string TagName { get; set; } = tagName;
    public string? Content { get; set; }
    public Dictionary<string, string> Attributes
    {
        get => this._attributes ??= [];
        set => this._attributes = value;
    }
     public List<PromptNode> ChildNodes
    {
        get => this._childNodes ??= [];
```

```
            set => this._childNodes = value;
        }
    }
```

The `XmlPromptParser` class extracts a textual prompt from XML using a collection of prompt nodes.

For each kind of AI service, there is a separate sub-folder with useful types. Let's look at the chat completion sub-folder:

- https://github.com/microsoft/semantic-kernel/tree/feature-connectors-openai/dotnet/src/SemanticKernel.Abstractions/AI/ChatCompletion

It contains helper types for working with chat completion (include history):

- AuthorRole.cs
- ChatCompletionServiceExtensions.cs
- ChatHistory.cs
- ChatMessageContentItemCollection.cs
- ChatPromptParser.cs
- IChatCompletionService.cs
- StreamingKernelContentItemCollection.cs

The `AuthorRole` class:

- https://github.com/microsoft/semantic-kernel/blob/feature-connectors-openai/dotnet/src/SemanticKernel.Abstractions/AI/ChatCompletion/AuthorRole.cs

helps distinguish different types of authors:

```
public readonly struct AuthorRole : IEquatable<AuthorRole>
{
    public static AuthorRole System { get; } = new("system");
    public static AuthorRole Assistant { get; } = new("assistant");
    public static AuthorRole User { get; } = new("user");
    public static AuthorRole Tool { get; } = new("tool");
    public string Label { get; }
    [JsonConstructor]
    public AuthorRole(string label)
    {
        Verify.NotNullOrWhiteSpace(label, nameof(label));
        this.Label = label!;
    }
   ...
}
```

The `ChatHistory` class:

- https://github.com/microsoft/semantic-kernel/blob/feature-connectors-openai/dotnet/src/SemanticKernel.Abstractions/AI/ChatCompletion/ChatHistory.cs

manages a list of `ChatMessageContent` instances:

```
/// <summary>
/// Provides a history of chat messages from a chat conversation.
/// </summary>
public class ChatHistory : Ilist<ChatMessageContent>,
                              IReadOnlyList<ChatMessageContent>
{
    private readonly List<ChatMessageContent> _messages;

    public ChatHistory(string systemMessage)
    {
        this._messages = [];
        this.AddSystemMessage(systemMessage);
    }
    public ChatHistory(IEnumerable<ChatMessageContent> messages)
    {
        Verify.NotNull(messages);
        this._messages = new(messages);
    }


    /// <summary>Gets the number of messages in the history.</summary>
    public int Count => this._messages.Count;

    /// <summary>
    /// <param name="authorRole">Role of the message author</param>
    /// <param name="content">Message content</param>
    /// <param name="encoding">Encoding of the message content</param>
    /// <param name="metadata">Dictionary for any
    /// additional metadata</param>
    /// </summary>
    public void AddMessage(AuthorRole authorRole, string content,
      Encoding? encoding = null, IReadOnlyDictionary<string,
      object?>? metadata = null) =>
        this.Add(new ChatMessageContent(authorRole, content, null,
                                   null, encoding, metadata));
}
```

`IChatCompletionService` defines the key interface for chat completion, with both non-streaming and streaming versions of `GetChatMessageContents`:

```
public interface IChatCompletionService : IAIService
{
    Task<IReadOnlyList<ChatMessageContent>> GetChatMessageContentsAsync(
        ChatHistory chatHistory,
        PromptExecutionSettings? executionSettings = null,
        Kernel? kernel = null,
        CancellationToken cancellationToken = default);

    IAsyncEnumerable<StreamingChatMessageContent>
GetStreamingChatMessageContentsAsync(
        ChatHistory chatHistory,
        PromptExecutionSettings? executionSettings = null,
        Kernel? kernel = null,
        CancellationToken cancellationToken = default);
}
```

The static `ChatCompletionServiceExtensions` class defines some useful extension methods on `IChatCompletionService`:

```
public static class ChatCompletionServiceExtensions
{
    public static async Task<ChatMessageContent> GetChatMessageContentAsync(
        this IChatCompletionService chatCompletionService,
        ChatHistory chatHistory,
        PromptExecutionSettings? executionSettings = null,
        Kernel? kernel = null,
        CancellationToken cancellationToken = default)
        => (await chatCompletionService.GetChatMessageContentsAsync(
                                chatHistory, executionSettings, kernel,
                                cancellationToken).ConfigureAwait(false))
            .Single();

}
```

No concrete implementation of `IChatCompletionService` is provided – it is up to each connector to deliver this functionality.

7: Core

Overview

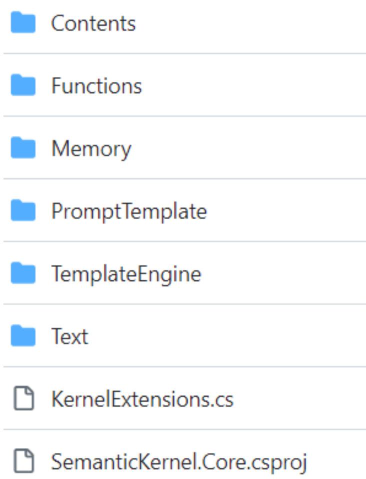
The SemanticKernel.Core assembly builds on Abstractions to provide a solid foundation for Semantic Kernel. Every application using semantic Kernel needs to import Core.

Core Source Tree

The Core source tree:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/SemanticKernel.Core

is laid out as follows:

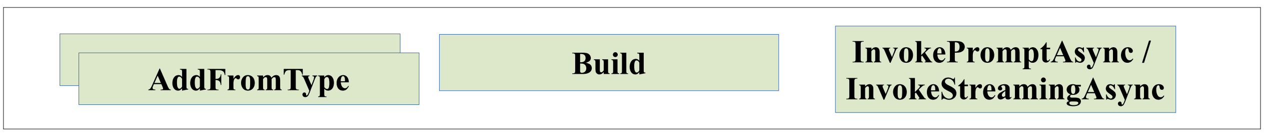


src/SemanticKernel.Core folder

KernelExtensions.cs is the sole source file in the root folder of Core:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/KernelExtensions.cs

It is quite large (almost a thousand lines) and consists of many extension methods (hence its name), divided into C# code regions. You will find the regions useful when exploring the code. Apart from `Build()` and `InvokePromptAsync() / InvokeStreamingAsync()`, all the other extension methods relate in some way to discovering and configuring kernel plugins and kernel functions. So we can say there are three kinds of extension methods here:



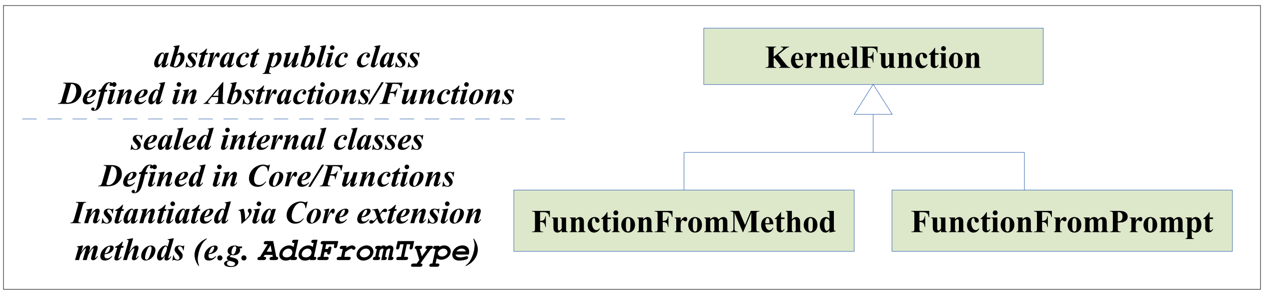
The kernel does not “intrinsically know” what you as an application developer wish to expose to the LLM – you need to tell it, via `AddFromType()` or similar extension methods. `AddFromType()` is defined as:

```
/// <summary>Creates a plugin that wraps a new instance of the specified
/// type <typeparamref name="T"/> and adds it into the plugin
/// collection.</summary>
/// <typeparam name="T">Specifies the type of the object to
/// wrap.</typeparam>
/// <param name="plugins">The plugin collection to which the new plugin
/// should be added.</param>
/// <param name="pluginName">
/// Name of the plugin for function collection and prompt templates. If
/// the value is null, a plugin name is derived from the type of the
/// <typeparamref name="T"/>.
/// </param>
/// <returns>The same instance as <paramref name="plugins"/>.</returns>
/// <remarks>
/// Methods that have the <see cref="KernelFunctionAttribute"/> attribute
/// will be included in the plugin.
/// See <see cref="KernelFunctionAttribute"/> attribute for details.
/// </remarks>
public static IKernelBuilderPlugins AddFromType<T>(
                                      this IKernelBuilderPlugins plugins,
                                      string? pluginName = null)
{
  Verify.NotNull(plugins);

  plugins.Services.AddSingleton<KernelPlugin>(
   serviceProvider =>
     KernelPluginFactory.CreateFromType<T>(pluginName, serviceProvider));

  return plugins;
}
```

we see it registers as a singleton for `KernelPlugin` a factory method parameter, which results in a call to `KernelPluginFactory.CreateFromType()` from:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/Functions/DefaultKernelPlugin.cs

which we will cover shortly. What is happening here is that .NET application developers do not manually create their own kernel plugins. Instead, they write normal C# types and decorate just those methods they wish to expose to LLMs with the `KernelFunction` attribute. The only other step they need to perform is call `AddFromType()` (or a similar extension) to get Semantic Kernel internally to construct the `KernelPlugin`. Similarly functionality is provided to create prompt functions.



The role of `Build()` is to create a `Kernel` instance. It is implemented as follows:

```
    /// <summary>Constructs a new instance of <see cref="Kernel"/> using all
    /// of the settings configured on the builder.</summary>
    /// <returns>The new <see cref="Kernel"/> instance.</returns>
    /// <remarks>
    /// Every call to <see cref="Build"/> produces a new <see cref="Kernel"/>
    /// instance. The resulting <see cref="Kernel"/>
    /// instances will not share the same plugins collection or services
    /// provider (unless there are no services).
    /// </remarks>
    public static Kernel Build(this IKernelBuilder builder) {
        ...
        IServiceProvider serviceProvider = EmptyServiceProvider.Instance;
        if (builder.Services is { Count: > 0 } services) {
            Dictionary<Type, HashSet<object?>> typeToKeyMappings = [];
            foreach (ServiceDescriptor serviceDescriptor in services) {
                if (!typeToKeyMappings.TryGetValue(
                            serviceDescriptor.ServiceType,
                                out HashSet<object?>? keys)) {
                 typeToKeyMappings[serviceDescriptor.ServiceType]= keys = [];
                }
                keys.Add(serviceDescriptor.ServiceKey);
            }
            services.AddKeyedSingleton(
                  Kernel.KernelServiceTypeToKeyMappings, typeToKeyMappings);
            serviceProvider = services.BuildServiceProvider();
        }
        return new Kernel(serviceProvider);
    }
}
```

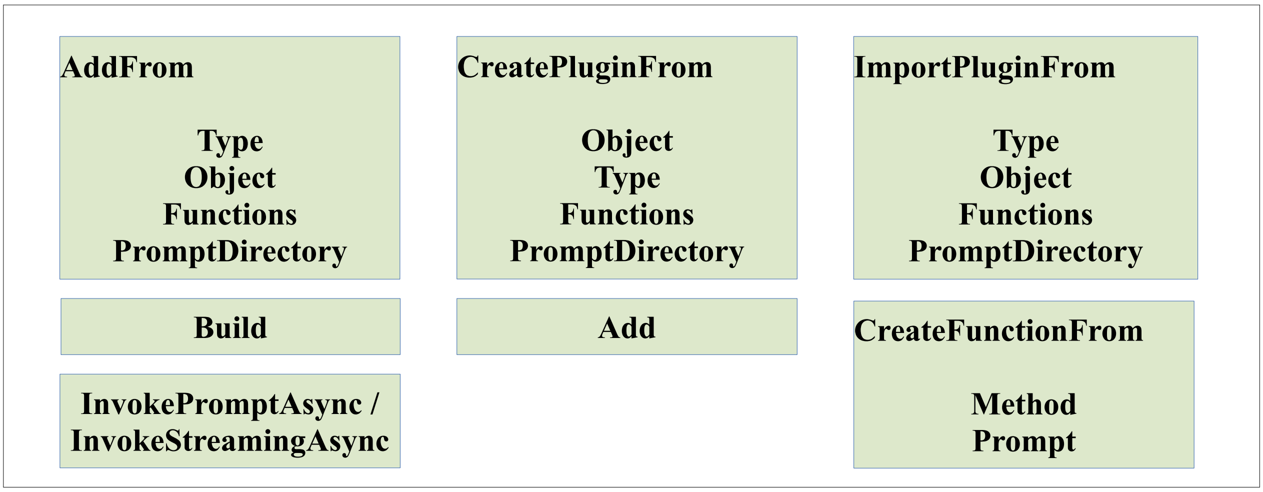
It initializes a serviceProvider to be an instance of `EmptyServiceProvider.Instance` and if `builder.Services` is empty, then this is passed to the `Kernel` constructor. If `builder.Services` is not empty, then a new service provider is constructed for the ist of services it has and this is subsequently used for the `Kernel` constructor.

`InvokePromptAsync()` and its streaming version, `InvokeStreamingAsync()`, invoke a prompt function.

```
/// <summary>
/// Invokes a prompt specified via a prompt template and returns the results
/// of type <typeparamref name="T"/>.
/// </summary>
/// <param name="kernel">The <see cref="Kernel"/> containing services,
/// plugins, and other state for use throughout the operation.</param>
/// <param name="promptTemplate">Prompt template for the function.</param>
/// <param name="arguments">The arguments to pass to the function's
/// invocation, including any <see cref="PromptExecutionSettings"/>.</param>
/// <param name="templateFormat">The template format of
/// <paramref name="promptTemplate"/>. This must be provided if
/// <paramref name="promptTemplateFactory"/> is not null.</param>
/// <param name="promptTemplateFactory">
/// The <see cref="IPromptTemplateFactory"/> to use when interpreting the
/// <paramref name="promptTemplate"/> into a <see cref="IPromptTemplate"/>.
/// If null, a default factory will be used.
/// </param>
/// <param name="cancellationToken">The <see cref="CancellationToken"/> to
/// monitor for cancellation requests. The default is
```

```
/// <see cref="CancellationToken.None"/>.</param>
/// <returns>The <typeparamref name="T"/> of the function result
/// value.</returns>
…
public static Task<T?> InvokePromptAsync<T>(
        this Kernel kernel,
        string promptTemplate,
        KernelArguments? arguments = null,
        string? templateFormat = null,
        IPromptTemplateFactory? promptTemplateFactory = null,
        CancellationToken cancellationToken = default)
    {
        Verify.NotNull(kernel);
        Verify.NotNullOrWhiteSpace(promptTemplate);

        KernelFunction function = KernelFunctionFromPrompt.Create(
            promptTemplate,
            functionName: KernelFunctionFromPrompt.CreateRandomFunctionName(
                                                nameof(InvokePromptAsync)),
            templateFormat: templateFormat,
            promptTemplateFactory: promptTemplateFactory,
            loggerFactory: kernel.LoggerFactory);

        return kernel.InvokeAsync<T>(function, arguments, cancellationToken);
    }
```

All the other extension methods are similar to `AddFromType()`, in that they construct a `KernelPlugin` from something. The full list of extension methods is as following:



src/SemanticKernel.Core/Functions folder

KernelFunctionFromMethod

The KernelFunctionFromMethod.cs file:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/SemanticKernel.Core/Functions/KernelFunctionFromMethod.cs

defines the key type that constructs a `KernelFunction` from a native method.

Its definition begins as:

```
/// <summary>
/// Provides factory methods for creating <see cref="KernelFunction"/>
instances backed by a .NET method.
/// </summary>
[DebuggerDisplay("{DebuggerDisplay,nq}")]
internal sealed partial class KernelFunctionFromMethod : KernelFunction
{ ... }
```

Note that it is marked as sealed, partial and internal. It is called, via intermediate methods, from the public extension method `AddFromType()`.

The native method is actually invoked in the `Invoke()` method:

```
/// <summary>Invokes the MethodInfo with the specified target
/// object and arguments.</summary>
private static object? Invoke(
             MethodInfo method, object? target, object?[]? arguments)
    {
        object? result = null;
        try
        {
            const BindingFlags  ...
            result = method.Invoke(target,
               BindingFlagsDoNotWrapExceptions, binder:
                          null, arguments, culture: null);
        }
        catch (TargetInvocationException e)
                       when (e.InnerException is not null)
        {
            ...
        }

        return result;
    }
```

`KernelFunctionFromMethod` includes this delegate

```
/// <summary>Delegate used to invoke the underlying delegate.</summary>
private delegate ValueTask<FunctionResult> ImplementationFunc(
   Kernel kernel,
   KernelFunction function,
   KernelArguments arguments,
   CancellationToken cancellationToken);
private readonly ImplementationFunc _function;
```

and a record struct to represent the method details:

```
private record struct MethodDetails(
  string Name,
  string Description,
  ImplementationFunc Function,
  List<KernelParameterMetadata> Parameters,
  KernelReturnParameterMetadata ReturnParameter);
```

It has a method called `GetMethodDetails()` to extract the method's details and most importantly, set up the `ImplementationFunc` delegate, where we note the call to `Invoke()` is located:

```
private static MethodDetails
   GetMethodDetails(string? functionName, MethodInfo method, object? target)
{
…

ValueTask<FunctionResult> Function(Kernel kernel, KernelFunction function,
KernelArguments arguments, CancellationToken cancellationToken)
{
    // Create the arguments.
    object?[] args = parameterFuncs.Length != 0 ?
                                    new object?[parameterFuncs.Length] : [];
    for (int i = 0; i < args.Length; i++)
    {
        args[i] = parameterFuncs[i](
                       function, kernel, arguments, cancellationToken);
    }

    // Invoke the method.
    object? result = Invoke(method, target, args);

    // Extract and return the result.
    return returnFunc(kernel, function, result);
}

// And return the details.
return new MethodDetails { Function = Function, ... }
```

The `KernelFunction` is created via a call to `Create()`:

```
    public static KernelFunction Create(
        MethodInfo method,
        object? target = null,
        KernelFunctionFromMethodOptions? options = default)
    {
        ...
        MethodDetails methodDetails =
            GetMethodDetails(options?.FunctionName, method, target);
        var result = new KernelFunctionFromMethod(
            methodDetails.Function,
            methodDetails.Name,
            options?.Description ?? methodDetails.Description,
            options?.Parameters?.ToList() ?? methodDetails.Parameters,
            options?.ReturnParameter ?? methodDetails.ReturnParameter,
            options?.AdditionalMetadata);
        ...
        return result;
    }
```

After all that hard work, the constructor for `KernelFunctionFromMethod` is trivial:

```
private KernelFunctionFromMethod(
        ImplementationFunc implementationFunc,
        string functionName,
        string? pluginName,
        string description,
        IReadOnlyList<KernelParameterMetadata> parameters,
        KernelReturnParameterMetadata returnParameter,
        ReadOnlyDictionary<string, object?>? additionalMetadata = null) :
        base(functionName, pluginName,
                description, parameters, returnParameter,
                additionalMetadata: additionalMetadata)
    {
        Verify.ValidFunctionName(functionName);

        this._function = implementationFunc;
    }
```

8: OpenAI and Azure.OpenAI Connectors

Overview

Connectors are how applications using Semantic Kernel connect to external AI services. The connector itself may use local helper DLLs, which you should be aware of when packing your own application. Semantic Kernel supports many kinds of connectors. In this chapter we will examine two – the OpenAI Connector and the Azure.OpenAI Connector. The other connecctors are similar in structure.

Note the layout of connectors in the source tree has been changing. We now see a flat structure (in the past there were sub-folders for AI and memory). We also now see separate OpenAI and Azure.OpenAI connectors. Where appropriate, Azure.OpenAI relies on OpenAI, so code is not duplicated. Rather Azure.OpenAI consists of small additions at key points.

To keep up to date with the evolution of connectors in Semantic Kernel, please read:

- https://devblogs.microsoft.com/semantic-kernel/support-for-azure-ai-openai-and-openai-v2-is-coming/

This is also of particular interest:

- https://github.com/microsoft/semantic-kernel/blob/feature-connectors-openai-release/docs/decisions/0047-azure-open-ai-connectors.md

src/Connectors/Connectors.OpenAI

The source for the OpenAI Connector:

- src/Connectors/Connectors.OpenAI

has this layout:

- Core
- Extensions
- Models
- Services
- Settings
- CompatibilitySuppressions.xml
- Connectors.OpenAI.csproj
- ToolCallBehavior.cs

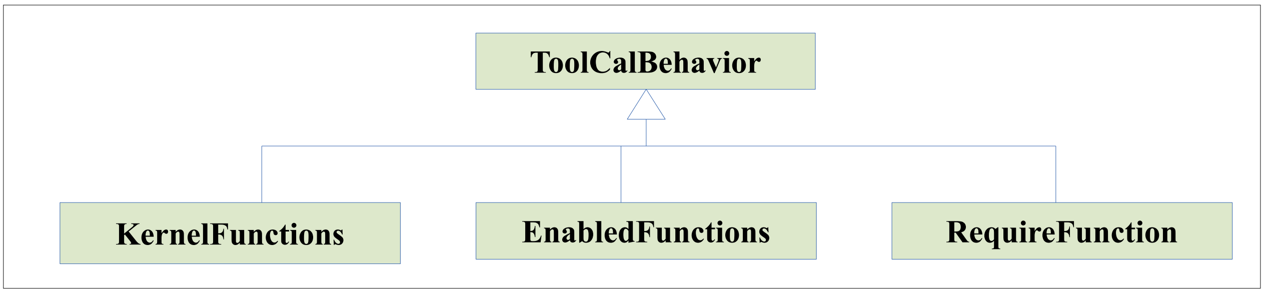
The .csproj file exposes the internals to the AzureOpenAI connector. Also, it includes the OpenAI assembly (which is where the actual connectivity to OpenAI takes place):

```
<Project Sdk="Microsoft.NET.Sdk">
...
  <ItemGroup>
    <InternalsVisibleTo
Include="SemanticKernel.Connectors.OpenAI.UnitTests" />
    <InternalsVisibleTo
Include="Microsoft.SemanticKernel.Connectors.AzureOpenAI" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\..\SemanticKernel.Core\
SemanticKernel.Core.csproj" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="OpenAI" />
    <PackageReference Include="System.ClientModel" />
  </ItemGroup>
</Project>
```

The ToolCallBehavior.cs file:

- https://github.com/microsoft/semantic-kernel/blob/feature-connectors-openai/dotnet/src/Connectors/Connectors.OpenAI/ToolCallBehavior.cs

defines the abstract class `ToolCallBehavior` and a range of derived classes:



Note the ‘s’ at the end of the `KernelFunctions` class and the `EnabledFunctions` class. Also note `KernelFunctions` is a quite separate class to the `KernelFunction` seen in Core.

Instances of these are constructed via a number of statics:

```
public static ToolCallBehavior EnableKernelFunctions
                        { get; } = new KernelFunctions(autoInvoke: false);

  public static ToolCallBehavior AutoInvokeKernelFunctions
                        { get; } = new KernelFunctions(autoInvoke: true);

  public static ToolCallBehavior EnableFunctions
    (IEnumerable<OpenAIFunction> functions, bool autoInvoke = false)
  {
      Verify.NotNull(functions);
      return new EnabledFunctions(functions, autoInvoke);
  }

  public static ToolCallBehavior RequireFunction(
                    OpenAIFunction function, bool autoInvoke = false)
```

```
        {
            Verify.NotNull(function);
            return new RequiredFunction(function, autoInvoke);
        }
```

For each of the derived classes, it is the `ConfigureOptions` method that is the most important detail.

src/Connectors/Connectors.OpenAI/Core

Core has functionality related to each type of the AI service built as part of a unified `ClientCore` (using the C# partial class feature):

- ClientCore.AudioToText.cs
- ClientCore.ChatCompletion.cs
- ClientCore.Embeddings.cs
- ClientCore.TextToAudio.cs
- ClientCore.TextToImage.cs
- ClientCore.cs
- OpenAIChatMessageContent.cs
- OpenAIFunction.cs
- OpenAIFunctionToolCall.cs
- OpenAIStreamingChatMessageContent.cs

So each file whose name begins with "ClientCore" adds to the `ClientCore` definition. Here is the main one:

- Connectors.OpenAI/Core/ClientCore.cs

This includes where run is actually called to invoke the function – note the invoke is wrapped in an exception handler that converts the `ClientResultException` to a `ToHttpOperationException`:

```
     protected static async Task<T> RunRequestAsync<T>(Func<Task<T>> request)
        {
            try
            {
                return await request.Invoke().ConfigureAwait(false);
            }
            catch (ClientResultException e)
            {
                throw e.ToHttpOperationException();
            }
        }

        protected static T RunRequest<T>(Func<T> request)
        {
```

```
        try
        {
            return request.Invoke();
        }
        catch (ClientResultException e)
        {
            throw e.ToHttpOperationException();
        }
    }
```

It starts with a number of properties:

```
private const string SingleSpace = " ";
internal const string OrganizationKey = "Organization";
private const string OpenAIV1Endpoint = "https://api.openai.com/v1";
protected internal string ModelId { get; init; } = string.Empty;
protected internal Uri? Endpoint { get; init; }
protected internal ILogger? Logger { get; init; }
protected internal OpenAIClient? Client { get; set; }
internal Dictionary<string, object?> Attributes { get; } = [];
```

Its constructor sets up various options to use for the call:

```
    internal ClientCore(
        string? modelId = null,
        string? apiKey = null,
        string? organizationId = null,
        Uri? endpoint = null,
        HttpClient? httpClient = null,
        ILogger? logger = null)
    {
        ..
        if (!string.IsNullOrWhiteSpace(modelId))
        {
            this.ModelId = modelId!;
            this.AddAttribute(AIServiceExtensions.ModelIdKey, modelId);
        }

        this.Logger = logger ?? NullLogger.Instance;

        // Accepts the endpoint if provided, otherwise uses the
        // default OpenAI endpoint.
        this.Endpoint = endpoint ?? httpClient?.BaseAddress;
        if (this.Endpoint is null)
        {
            Verify.NotNullOrWhiteSpace(apiKey);
            // For Public OpenAI Endpoint a key must be provided.
            this.Endpoint = new Uri(OpenAIV1Endpoint);
        }
        else if (string.IsNullOrEmpty(apiKey))
        {
            // Avoids an exception from OpenAI Client when a custom
            // endpoint is provided without an API key.
            apiKey = SingleSpace;
        }

        this.AddAttribute(AIServiceExtensions.EndpointKey,
                                          this.Endpoint.ToString());

        var options = GetOpenAIClientOptions(httpClient, this.Endpoint);
        if (!string.IsNullOrWhiteSpace(organizationId))
```

```
        {
            options.AddPolicy(CreateRequestHeaderPolicy(
        "OpenAI-Organization", organizationId!), PipelinePosition.PerCall);

            this.AddAttribute(ClientCore.OrganizationKey, organizationId);
        }

        this.Client = new OpenAIClient(apiKey!, options);
    }
```

We will now move our attention to the ClientCore.ChatCompletion.cs file as an example:

- Connectors.OpenAI/Core/ClientCore.ChatCompletion.cs

and specifically we will look in depth at `GetChatMessageContentsAsync()` as this is interesting. It is long, so let's examine it in smaller chunks:

```
internal async Task<IreadOnlyList<ChatMessageContent>>
  GetChatMessageContentsAsync(
        string targetModel,
        ChatHistory chat,
        PromptExecutionSettings? executionSettings,
        Kernel? kernel,
        CancellationToken cancellationToken = default)
{
  ...
  // Convert the incoming execution settings to OpenAI settings.
  OpenAIPromptExecutionSettings chatExecutionSettings =
                this.GetSpecializedExecutionSettings(executionSettings);
  ValidateMaxTokens(chatExecutionSettings.MaxTokens);
  for (int requestIndex = 0; ; requestIndex++)
  {
      var chatForRequest =
           CreateChatCompletionMessages(chatExecutionSettings, chat);

      var toolCallingConfig = this.GetToolCallingConfiguration(
                              kernel, chatExecutionSettings, requestIndex);

      var chatOptions = this.CreateChatCompletionOptions(
                  chatExecutionSettings, chat, toolCallingConfig, kernel);
```

It starts by preparing the settings and options. Then it makes the request:

```
      // Make the request.
      OpenAIChatCompletion? chatCompletion = null;
      OpenAIChatMessageContent chatMessageContent;
      using (var activity = this.StartCompletionActivity(
                                   chat, chatExecutionSettings))
      {
          try
          {
              chatCompletion = (await RunRequestAsync(() =>
                        this.Client!.GetChatClient(targetModel)
                          .CompleteChatAsync(chatForRequest, chatOptions,
                           cancellationToken)).ConfigureAwait(false)).Value;
              this.LogUsage(chatCompletion.Usage);
          }
          catch (Exception ex) when (activity is not null)
          {
```

```
                ...
            }
```

Next it create the char message content:

```
chatMessageContent = this.CreateChatMessageContent(chatCompletion,
targetModel);
activity?.SetCompletionResponse([chatMessageContent],
chatCompletion.Usage.InputTokens, chatCompletion.Usage.OutputTokens);
}

// If we don't want to attempt to invoke any functions,
// just return the result.
if (!toolCallingConfig.AutoInvoke)
{
  return [chatMessageContent];
}

Debug.Assert(kernel is not null);
```

Then it processes the function calls, if any:

```
// Get our single result and extract the function call information. If this
// isn't a function call, or if it is but we're unable to find the function
// or extract the relevant information, just return the single result.
// Note that we don't check the FinishReason and instead check whether there
// are any tool calls, as the service may return a FinishReason of "stop"
// even if there are tool calls to be made, in particular if a required tool
// is specified.
if (chatCompletion.ToolCalls.Count == 0)
{
      return [chatMessageContent];
}
...

// Add the result message to the caller's chat history;
// this is required for the service to understand the tool call responses.
chat.Add(chatMessageContent);

// We must send back a response for every tool call, regardless of whether we
// successfully executed it or not. If we successfully execute it, we'll add
// the result. If we don't, we'll add an error.
for (int toolCallIndex = 0; toolCallIndex <
               chatMessageContent.ToolCalls.Count; toolCallIndex++)
{
   ChatToolCall functionToolCall =
                         chatMessageContent.ToolCalls[toolCallIndex];
                ...
                // Parse the function call arguments.
                OpenAIFunctionToolCall? openAIFunctionToolCall;
                try
                {
                    openAIFunctionToolCall = new(functionToolCall);
                }
                catch (JsonException)
                {
                    AddResponseMessage(chat, result: null,
                    "Error: Function call arguments were invalid JSON.",
                    functionToolCall, this.Logger);
                    continue;
```

```
          }
          ...
          // Find the function in the kernel and populate the
          // arguments.
          if (!kernel!.Plugins.TryGetFunctionAndArguments(
                   openAIFunctionToolCall,
                   out KernelFunction? function,
                   out KernelArguments? functionArgs))
          {
              AddResponseMessage(chat, result: null,
                   "Error: Requested function could not be found.",
                   functionToolCall, this.Logger);
              continue;
          }

          // Now, invoke the function, and add the resulting tool call
          //  message to the chat options.
          FunctionResult functionResult
                        = new(function) { Culture = kernel.Culture };
        AutoFunctionInvocationContext invocationContext =
        new(kernel, function, functionResult, chat, chatMessageContent)
          {
              Arguments = functionArgs,
              RequestSequenceIndex = requestIndex,
              FunctionSequenceIndex = toolCallIndex,
              FunctionCount = chatMessageContent.ToolCalls.Count
          };

          s_inflightAutoInvokes.Value++;
          try
          {
              invocationContext =
                await OnAutoFunctionInvocationAsync(
                    kernel, invocationContext, async (context) =>
              {
                  // Check if filter requested termination.
                  if (context.Terminate)
                  {
                      return;
                  }


                  context.Result =
       await function.InvokeAsync(
 kernel, invocationContext.Arguments, cancellationToken:
cancellationToken).ConfigureAwait(false);
              }).ConfigureAwait(false);
          }
          catch (Exception e)
          {
              AddResponseMessage(chat, null,
              $"Error: Exception while invoking function. {e.Message}",
              functionToolCall, this.Logger);
              continue;
          }
          finally
          {
              s_inflightAutoInvokes.Value--;
          }
```

```
                functionResult = invocationContext.Result;

                object functionResultValue =
                       functionResult.GetValue<object>() ?? string.Empty;
                var stringResult = ProcessFunctionResult(
                functionResultValue, chatExecutionSettings.ToolCallBehavior);

                AddResponseMessage(chat, stringResult, errorMessage:
                                        null, functionToolCall, this.Logger);


                if (invocationContext.Terminate)
                {
                    if (this.Logger.IsEnabled(LogLevel.Debug))
                    {
                        this.Logger.LogDebug(
           "Filter requested termination of automatic function invocation.");
                    }

                    return [chat.Last()];
                }
            }
        }
    }
```

src/Connectors/Connectors.OpenAI/Extensions

The OpenAIKernelBuilderExtensions.cs file:

- Connectors.OpenAI/Extensions/OpenAIKernelBuilderExtensions.cs

defines the `OpenAIKernelBuilderExtensions` class with a number of extension methods, such as this, which sets up key singletons for DI:

```
    /// <summary>
    /// Adds the OpenAI chat completion service to the list.
    /// </summary>
    /// <param name="builder">The <see cref="IKernelBuilder"/> instance to
    /// augment.</param>
    /// <param name="modelId">OpenAI model id</param>
    /// <param name="openAIClient"><see cref="OpenAIClient"/> to use for the
    /// service. If null, one must be available in the service provider when
    /// this service is resolved.</param>
    /// <param name="serviceId">A local identifier for the given AI
    /// service</param>
    /// <returns>The same instance as <paramref name="builder"/>.</returns>
    public static IKernelBuilder AddOpenAIChatCompletion(
        this IKernelBuilder builder,
        string modelId,
        OpenAIClient? openAIClient = null,
        string? serviceId = null)
    {
        Verify.NotNull(builder);
        Verify.NotNullOrWhiteSpace(modelId);

        OpenAIChatCompletionService Factory(
             IServiceProvider serviceProvider, object? _) =>
```

```
                  new(modelId, openAIClient ??
                   serviceProvider.GetRequiredService<OpenAIClient>(),
                  serviceProvider.GetService<ILoggerFactory>());

      builder.Services.AddKeyedSingleton<IchatCompletionService>(
        serviceId,
        (Func<IServiceProvider, object?, OpenAIChatCompletionService>)Factory);

      builder.Services.AddKeyedSingleton<ItextGenerationService>(
        serviceId,
        (Func<IServiceProvider, object?, OpenAIChatCompletionService>)Factory);

      return builder;
  }
```

src/Connectors/Connectors.OpenAI/Services

The `OpenAIChatCompletionService` class is defined in:

- Connectors.OpenAI/Services/OpenAIChatCompletionService.cs

We see in its constructor it sets up an `OpenAIClient` instance:

```
public sealed class OpenAIChatCompletionService
                      : IChatCompletionService, ITextGenerationService
{
    private readonly ClientCore _client;

    public OpenAIChatCompletionService(
        string modelId,
        OpenAIClient openAIClient,
        ILoggerFactory? loggerFactory = null)
    {
        this._client = new(
            modelId,
            openAIClient,
            loggerFactory?.CreateLogger(
                      typeof(OpenAIChatCompletionService)));
    }
```

and its `GetChatMessageContentsAsync()` function simply calls `OpenAIClient.GetChatMessageContentsAsync()`:

```
public Task<IReadOnlyList<ChatMessageContent>> GetChatMessageContentsAsync(
        ChatHistory chatHistory,
        PromptExecutionSettings? executionSettings = null,
        Kernel? kernel = null,
        CancellationToken cancellationToken = default)
        => this._client.GetChatMessageContentsAsync(
                this._client.ModelId, chatHistory, executionSettings,
                                                kernel, cancellationToken);
...
}
```

9: Agent Framework

Overview

The evolving Agent Framework within Semantic Kernel helps developers build a variety of agent types. The word "agent" in AI circles can mean anything from a simple chatbot interacting with a human user to a fully autonomous agent, capable of independent action. There is a spectrum and when using the "agent" term it is important to be aware, from context or clearly stated, where on that spectrum is being discussed. Over time it is likely we will see a move towards agents as autonomous agents, empowered with richer access to tools (compared to say a regular Copilot), ability to remember conversation threads and ability to participate in multi-agent group chats.

A sample usage scenario for an agent is imagine for a large construction project a notice from a contractor arrives electronically at 2am. There are no human workers at their desk there to process the notice (so a Copilot is of no use); but an autonomous agent has been deployed. Depending on the notice content, the autonomous agent can decide whether to process it fully (e.g. change of postal address of contractor), partially (imagine the notice is a request for payment - the agent can prepare a draft response and a human engineer in the morning can then complete/alter it as needed) or not at all (the contractor wants to terminate the contract, which is likely to lead to legal action; time to call in the lawyers).

Many software architects believe the future of software architecture will be delivering swarms of highly capable AI agents that can autonomously work together to deliver solutions. Software architects need to consider how these agents work from an AI context (explored in this chapter) and also within the wider software solution. As just one example (of many), consider access control. Just like regular user accounts, permissions would need to be assigned to agent accounts. What the agent can and cannot do depends on those permissions. In contrast, a copilot is a proxy for a human user and so uses the permissions of the user (pilot) running the Copilot.

Useful Links

To get a a good overview of agents in Semantic Kernel, read this post:

- https://devblogs.microsoft.com/semantic-kernel/introducing-agents-in-semantic-kernel/

The architecture of agents in Semantic Kernel is intended to deliver a design that is independent of any particular AI cloud service. That said, the first to get supported in Semantic Kernel is OpenAI Assistants, so you should read up on that:

- https://platform.openai.com/docs/assistants/overview

Design Decisions

The docs/decisions subfolder:

- https://github.com/microsoft/semantic-kernel/tree/main/docs/decisions

contains detailed explanations behind design decisions. For agents, these are extremely well written and contain detail you will not find elsewhere:

- https://github.com/microsoft/semantic-kernel/blob/main/docs/decisions/0032-agents.md
- https://github.com/microsoft/semantic-kernel/blob/main/docs/decisions/0048-agent-chat-serialization.md
- https://github.com/microsoft/semantic-kernel/blob/main/docs/decisions/0049-agents-assistantsV2.md

Each decision contains introductory as well as more detailed content. For example, 032 states:

> *An agent is expected to be able to support two interaction patterns:*
>
> *Direct Invocation ("No Chat"):The caller is able to directly invoke any single agent without any intervening machinery or infrastructure. For different agents to take turns in a conversation using direct invocation, the caller is expected to invoke each agent per turn. Coordinating interaction between different agent types must also be explicitly managed by the caller.*
>
> *Agent Chat:The caller is able to assemble multiple agents to participate in an extended conversation for the purpose of accomplishing a specific goal (generally in response to initial or iterative input). Once engaged, agents may participate in the chat over multiple interactions by taking turns.*

Sample Code

The Semantic Kernel repo has sample code related to Agents here:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/GettingStartedWithAgents

By carefully exploring these samples you can clearly see how the various features in the agent framework are intended to work.

Terminology

We need a few pieces of terminology:

- Agent – an AI construct that participates in conversations (AgentChats)
- Channel – a construct through which conversation messages flow
- Aggregator – aggregation of multiple agents or channels to appear as one
- AgentChat – a conversation with one or more agents via channels
- BroadcastQueue – broadcasts AgentChat messages to channels

Source Layout

The agent-related source in the .NET version of Semantic Kernel is here:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Agents

Abstractions
Core
OpenAI
UnitTests

It is comprised of three main source sub-folders, which we will now explore.

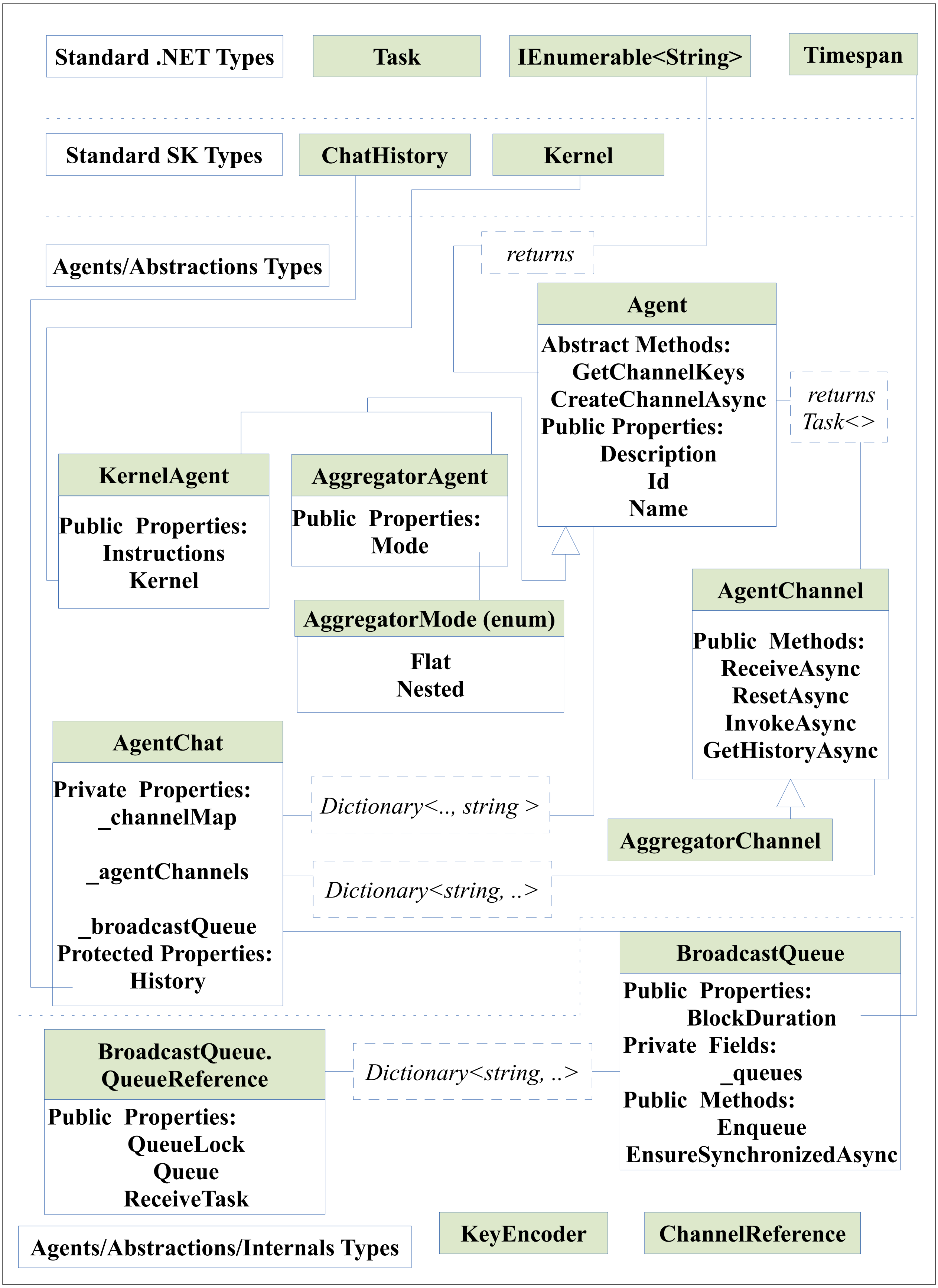
src/Agents/Abstractions

The Abstractions folder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Agents/Abstractions

has the following layout:

Extensions	Agent.cs
Internal	AgentChannel.cs
Logging	AgentChat.cs
Properties	Agents.Abstractions.csproj
	AggregatorAgent.cs
	AggregatorChannel.cs
	KernelAgent.cs


Standard .NET Types
Task
IEnumerable<String>
Timespan
Standard SK Types
ChatHistory
Kernel
Agents/Abstractions Types
returns
Agent
Abstract Methods:
GetChannelKeys
CreateChannelAsync
Public Properties:
Description
Id
Name
returns
Task<>
KernelAgent
Public Properties:
Instructions
Kernel
AggregatorAgent
Public Properties:
Mode
AggregatorMode (enum)
Flat
Nested
AgentChannel
Public Methods:
ReceiveAsync
ResetAsync
InvokeAsync
GetHistoryAsync
AgentChat
Private Properties:
_channelMap
_agentChannels
_broadcastQueue
Protected Properties:
History
Dictionary<.., string >
Dictionary<string, ..>
AggregatorChannel
BroadcastQueue
Public Properties:
BlockDuration
Private Fields:
_queues
Public Methods:
Enqueue
EnsureSynchronizedAsync
BroadcastQueue.
QueueReference
Public Properties:
QueueLock
Queue
ReceiveTask
Dictionary<string, ..>
Agents/Abstractions/Internals Types
KeyEncoder
ChannelReference

Agent

The Agent type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Agents/Abstractions/Agent.cs

is an abstract class defined as:

```
public abstract class Agent
{
    public string? Description { get; init; }

    public string Id { get; init; } = Guid.NewGuid().ToString();

    public string? Name { get; init; }

    public ILoggerFactory LoggerFactory { get; init; }
                                        = NullLoggerFactory.Instance;

    protected ILogger Logger =>
          this._logger ??= this.LoggerFactory.CreateLogger(this.GetType());

    protected internal abstract IEnumerable<string> GetChannelKeys();

    protected internal abstract Task<AgentChannel>
                  CreateChannelAsync(CancellationToken cancellationToken);

    private ILogger? _logger;
}
```

Two important abstract methods are `GetChannelKeys()` and `CreateChannelAsync()` – both related to channels, which we will cover presently.

KernelAgent

The KernelAgent type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Agents/Abstractions/KernelAgent.cs

derives from `Agent` and includes an optional string called `Instruction` (explains to te `Agent` how it should behave) and a mandatory `Kernel`:

```
public abstract class KernelAgent : Agent
{
    public string? Instructions { get; init; }
    public Kernel Kernel { get; init; } = new();
}
```

AggregatorAgent

The `AggregatorAgent` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Agents/Abstractions/AggregatorAgent.cs

is defined as:

```
public enum AggregatorMode
{
     Flat,

    Nested,
}

public sealed class AggregatorAgent(Func<AgentChat> chatProvider) : Agent
{
    public AggregatorMode Mode { get; init; } = AggregatorMode.Flat;

    /// <inheritdoc/>
    protected internal override IEnumerable<string> GetChannelKeys()
    {
        yield return typeof(AggregatorChannel).FullName!;
    }

    /// <inheritdoc/>
    protected internal override Task<AgentChannel>
CreateChannelAsync(CancellationToken cancellationToken)
    {}
}
```

AgentChannel

The AgentChannel type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Agents/Abstractions/AgentChannel.cs

is defined as:

```
public abstract class AgentChannel
{
    public ILogger Logger { get; set; } = NullLogger.Instance;

    protected internal abstract Task
ReceiveAsync(IEnumerable<ChatMessageContent> history, CancellationToken
cancellationToken = default);

   protected internal abstract Task ResetAsync(CancellationToken
cancellationToken = default);

   protected internal abstract IAsyncEnumerable<(bool IsVisible,
ChatMessageContent Message)> InvokeAsync(
        Agent agent,
        CancellationToken cancellationToken = default);

    protected internal abstract IAsyncEnumerable<ChatMessageContent>
GetHistoryAsync(CancellationToken cancellationToken = default);
}
```

It also have an alternative definition, that takes a type parameter:

```
public abstract class AgentChannel<TAgent> :
                               AgentChannel where TAgent : Agent
{
    protected internal abstract IAsyncEnumerable<(bool IsVisible,
ChatMessageContent Message)> InvokeAsync(
        TAgent agent,
```

```
            CancellationToken cancellationToken = default);

        protected internal override IAsyncEnumerable<(bool IsVisible,
                                  ChatMessageContent Message)> InvokeAsync(
            Agent agent,
            CancellationToken cancellationToken = default)
        {
            if (agent.GetType() != typeof(TAgent))
            {
                throw new KernelException($"Invalid agent channel:
                           {typeof(TAgent).Name}/{agent.GetType().Name}");
            }

            return this.InvokeAsync((TAgent)agent, cancellationToken);
        }
    }
```

There is one more very important type in Agents/Abstractions, called `AgentChat`, but before exploring that we are first going to explore the internals subfolder, because we need its contents to explain the `AgentChat`.

src/Agents/Abstractions/Internals

The Internals subfolder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Agents/Abstractions/Internal

has these files:

- BroadcastQueue.cs
- ChannelReference.cs
- KeyEncoder.cs

`ChannelReference` associates a hash with an `AgentChannel`:

```
/// <summary>
/// Tracks channel along with its hashed key.
/// </summary>
internal readonly struct ChannelReference(AgentChannel channel, string hash)
{
    /// <summary>
    /// The referenced channel.
    /// </summary>
    public AgentChannel Channel { get; } = channel;

    /// <summary>
    /// The channel hash.
    /// </summary>
    public string Hash { get; } = hash;
}
```

`KeyEncoder` merges an enumerable of strings (with ':' between each pair), then computes the SHA256 hash and then returns its base64 encoding. If the NET symbol is defined it uses the faster stackalloc technique for memory for the hash and then calls `HashData`, otherwise uses the traditional approach:

```
/// <summary>
/// Utility to encode a list of string keys to an base-64 encoded hash.
/// </summary>
internal static class KeyEncoder
{
    public static string GenerateHash(IEnumerable<string> keys)
    {
        byte[] buffer = Encoding.UTF8.GetBytes(string.Join(":", keys));
#if NET
        Span<byte> hash = stackalloc byte[32];
        SHA256.HashData(buffer, hash);
#else
        using SHA256 shaProvider = SHA256.Create();
        byte[] hash = shaProvider.ComputeHash(buffer);
#endif
        return Convert.ToBase64String(hash);
    }
}
```

The BroadcastQueue.cs file:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Agents/Abstractions/Internal/BroadcastQueue.cs

defines one class, `BroadcastQueue`, which internally defines the `QueueReference` class.

This file starts with this `using`:

```
using ChannelQueue = System.Collections.Generic.Queue<
System.Collections.Generic.IReadOnlyList<
                        Microsoft.SemanticKernel.ChatMessageContent> >;
```

which allows use of a symbol `ChannelQueue` to represent a `Queue` of `IReadOnlyList` of `SemanticKernel.ChatMessageContent`. Note `Queue` is a standard .NET type:

- https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1

with these characteristics:

> *This class implements a generic queue as a circular array. Objects stored in a Queue<T> are inserted at one end and removed from the other. Queues and stacks are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value.*
>
> ...
>
> *Three main operations can be performed on a Queue<T> and its elements:*
>
> *Enqueue adds an element to the end of the Queue<T>.*
>
> *Dequeue removes the oldest element from the start of the Queue<T>.*
>
> *Peek peek returns the oldest element that is at the start of the Queue<T> but does not remove it from the Queue<T>.*

`QueueReference` is a simple sealed class that manages the association between a queue and its lock:

```
private sealed class QueueReference
{
    /// <summary>
    /// Convenience logic
    /// </summary>
    public bool IsEmpty => this.Queue.Count == 0;

    /// <summary>
    /// Queue specific lock to control queue access with
    /// finer granularity
    /// than the state-lock.
    /// </summary>
    public object QueueLock { get; } = new object();

    /// <summary>
    /// The target queue.
    /// </summary>
    public ChannelQueue Queue { get; } = new ChannelQueue();

    /// <summary>
    /// The task receiving and processing messages
    /// from <see cref="Queue" />.
    /// </summary>
    public Task? ReceiveTask { get; set; }

    /// <summary>
    /// Capture any failure that may occur during execution
    /// of <see cref="ReceiveTask"/>.
    /// </summary>
    public Exception? ReceiveFailure { get; set; }
}
```

src/Agents/Abstractions (AgentChat)

The AgentChat type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Agents/Abstractions/AgentChat.cs

is a large abstract class, whose outline is:

```
public abstract class AgentChat
{
    private readonly BroadcastQueue _broadcastQueue;
    private readonly Dictionary<string, AgentChannel> _agentChannels;
    private readonly Dictionary<Agent, string> _channelMap;
    private int _isActive;
    private ILogger? _logger;

    public bool IsActive =>
               Interlocked.CompareExchange(ref this._isActive, 1, 1) > 0;

    public ILoggerFactory LoggerFactory { get; init; }
         = NullLoggerFactory.Instance;

    protected ILogger Logger => this._logger ??=
                   this.LoggerFactory.CreateLogger(this.GetType());

   rotected ChatHistory History { get; }

   public abstract IasyncEnumerable<ChatMessageContent>
      InvokeAsync(CancellationToken cancellationToken = default);

   public IasyncEnumerable<ChatMessageContent>
      GetChatMessagesAsync(CancellationToken cancellationToken = default) =>
        this.GetChatMessagesAsync(agent: null, cancellationToken);

   public async IAsyncEnumerable<ChatMessageContent> GetChatMessagesAsync(
        Agent? agent,
        [EnumeratorCancellation] CancellationToken
                          cancellationToken = default) { }

   public void AddChatMessage(ChatMessageContent message) { }

   public void AddChatMessages(IReadOnlyList<ChatMessageContent> messages){}

   protected async IAsyncEnumerable<ChatMessageContent> InvokeAgentAsync(
        Agent agent,
        [EnumeratorCancellation]
                        CancellationToken cancellationToken = default) {}

   public async Task ResetAsync(
                          CancellationToken cancellationToken = default) {}

   private void ClearActivitySignal() {}

   protected void SetActivityOrThrow() { }

   private string GetAgentHash(Agent agent) { }

   private async Task<AgentChannel?> SynchronizeChannelAsync(
```

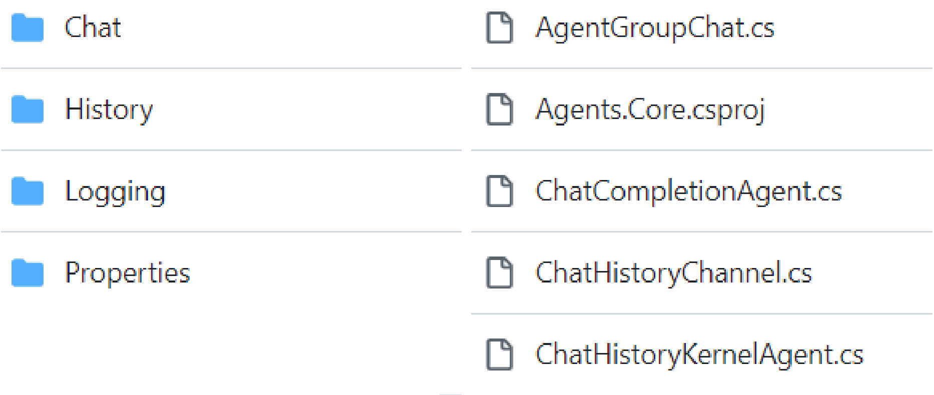
```
            string channelKey, CancellationToken cancellationToken) { }

        protected AgentChat() { }
    }
```

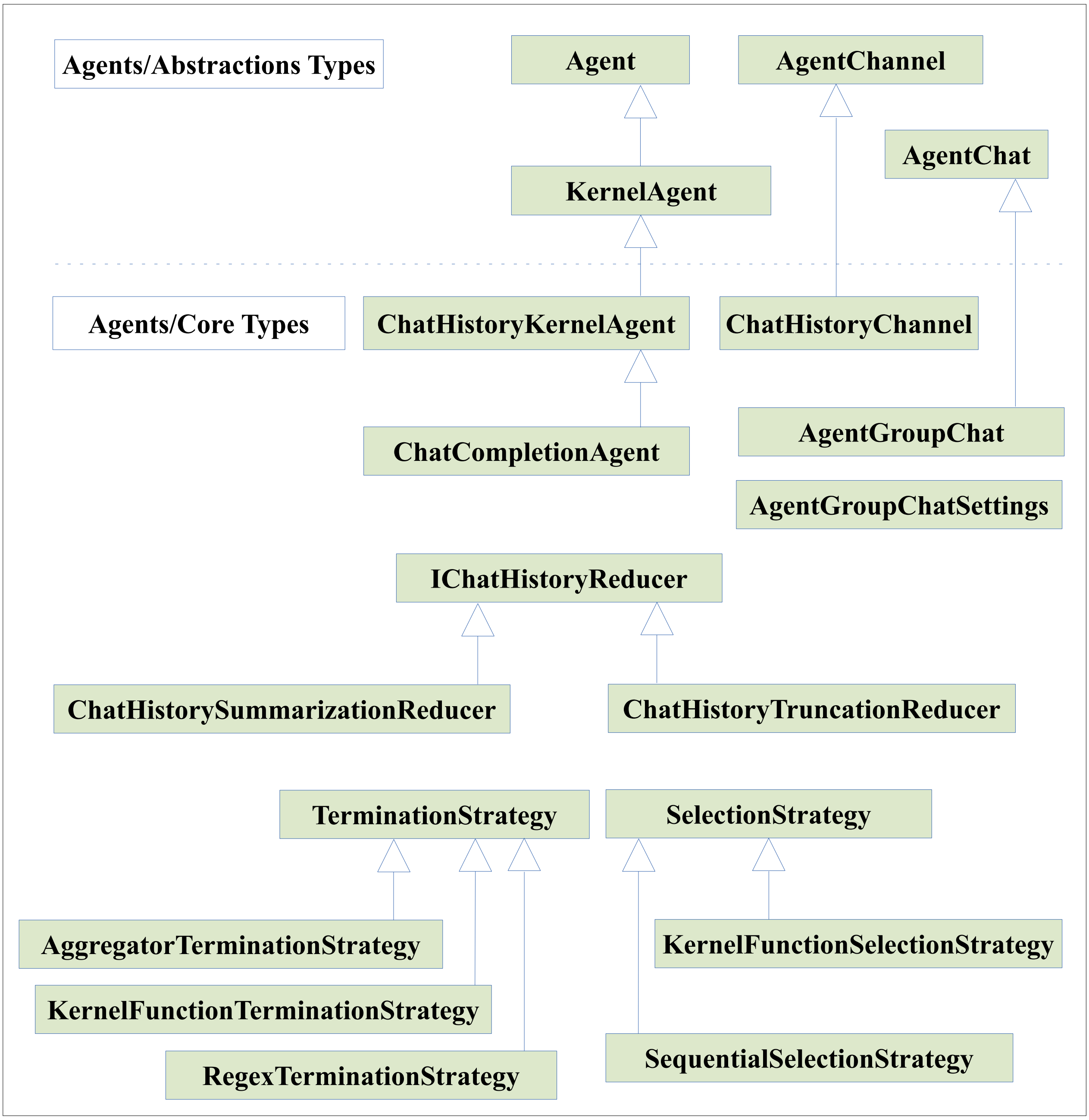
src/Agents/Core

The Agents/Core folder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Agents/Core

has the following layout:




Agents/Abstractions Types
Agent
AgentChannel
AgentChat
KernelAgent
Agents/Core Types
ChatHistoryKernelAgent
ChatHistoryChannel
ChatCompletionAgent
AgentGroupChat
AgentGroupChatSettings
IChatHistoryReducer
ChatHistorySummarizationReducer
ChatHistoryTruncationReducer
TerminationStrategy
SelectionStrategy
AggregatorTerminationStrategy
KernelFunctionSelectionStrategy
KernelFunctionTerminationStrategy
RegexTerminationStrategy
SequentialSelectionStrategy

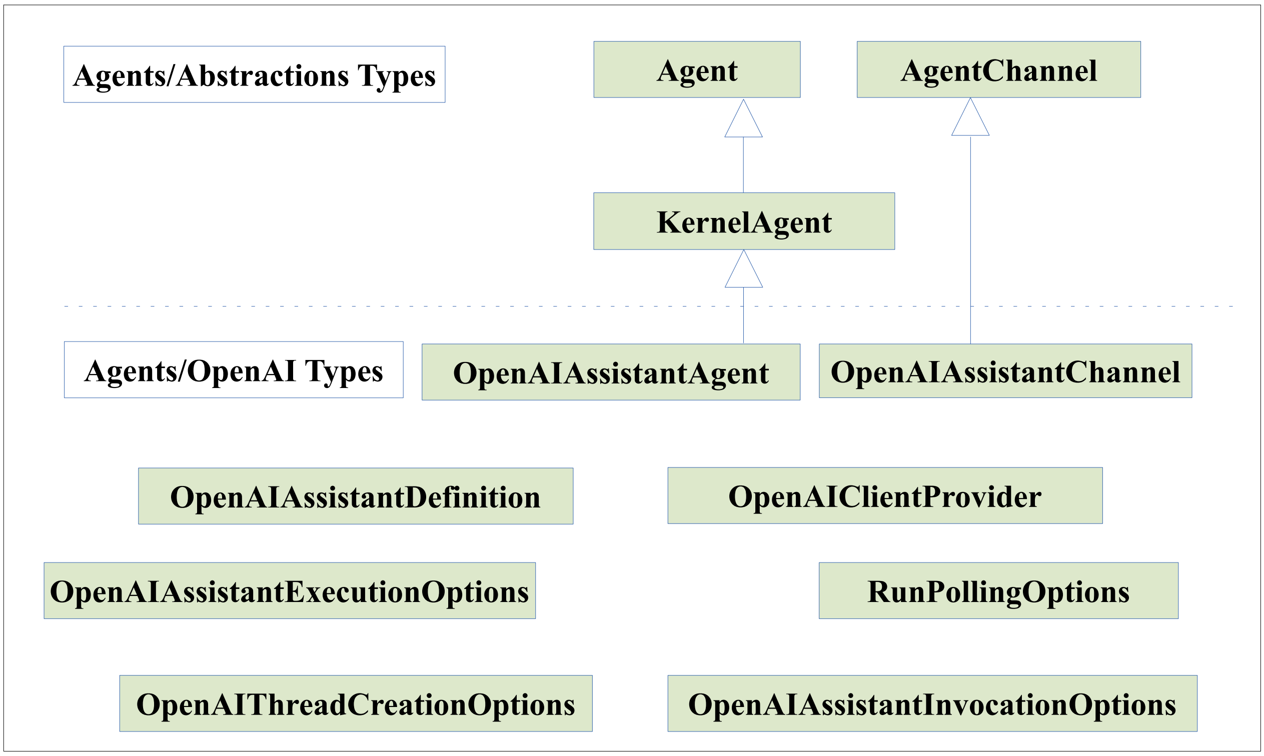
src/Agents/OpenAI

The Agents/OpenAI folder:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Agents/OpenAI

has this layout:

Extensions	Agents.OpenAI.csproj
Internal	OpenAIAssistantAgent.cs
Logging	OpenAIAssistantChannel.cs
Properties	OpenAIAssistantDefinition.cs
	OpenAIAssistantExecutionOptions.cs
	OpenAIAssistantInvocationOptions.cs
	OpenAIClientProvider.cs
	OpenAIThreadCreationOptions.cs
	RunPollingOptions.cs



10: Process Framework

Overview

Modern knowledge-intensive enterprises increasingly work with business processes and Process Framework adds AI to how these can be delivered.

The Process Framework extends Semantic Kernel with support for what we might call AI processes. These are controlled multistep activities leading to some goal – now possibly including AI steps (it is expected that steps not related to AI – e.g. local-code, REST API calls or human interactions – are also steps in a process).

Here we are using "process" in the business sense ("business processes") and do not confuse this use with the use of the "process" term that an app developer working with operating system APIs (process and threads) would use ("to fork a process").

Useful Links

Excellent article by Jose Luis Latorre:

- https://www.linkedin.com/pulse/introducing-semantic-kernel-process-library-new-era-ai-latorre-g8tef/

Documentation:

- https://learn.microsoft.com/en-us/semantic-kernel/frameworks/process/process-framework

Semantic Kernel Devblog:

- https://devblogs.microsoft.com/semantic-kernel/integrating-ai-into-business-processes-with-the-process-framework

Docs Change

Semantic Kernel allowed for controlled use of experimental features via feature IDs, which are listed here:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/docs/EXPERIMENTS.md

By default, warnings are issued during compilation when using experimental features – this document explains how to selectively turn off warnings if so desired.

This document includes an entry for Process framework:

```
SKEXP0080 Process Framework
```

Sample code

Sample code showing use of Process Framework is here:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/GettingStartedWithProcesses

Terminology

- Process – this is a business process, usually composed of multiple steps
- Step – a discrete manageable activity, that is part of a larger business process
- Local runtime - when the business process is executed locally inside your process. We would expect support for other kinds of runtimes – such as Microsoft Orleans – to be added in future.

Source Layout

You can find the Process Framework source code under Experimental:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Experimental

and it include three assemblies – for abstractions, core and local runtime:

- Process.Abstractions
- Process.Core
- Process.LocalRuntime
- Process.UnitTests

src/Experimental/ProcessAbstractions

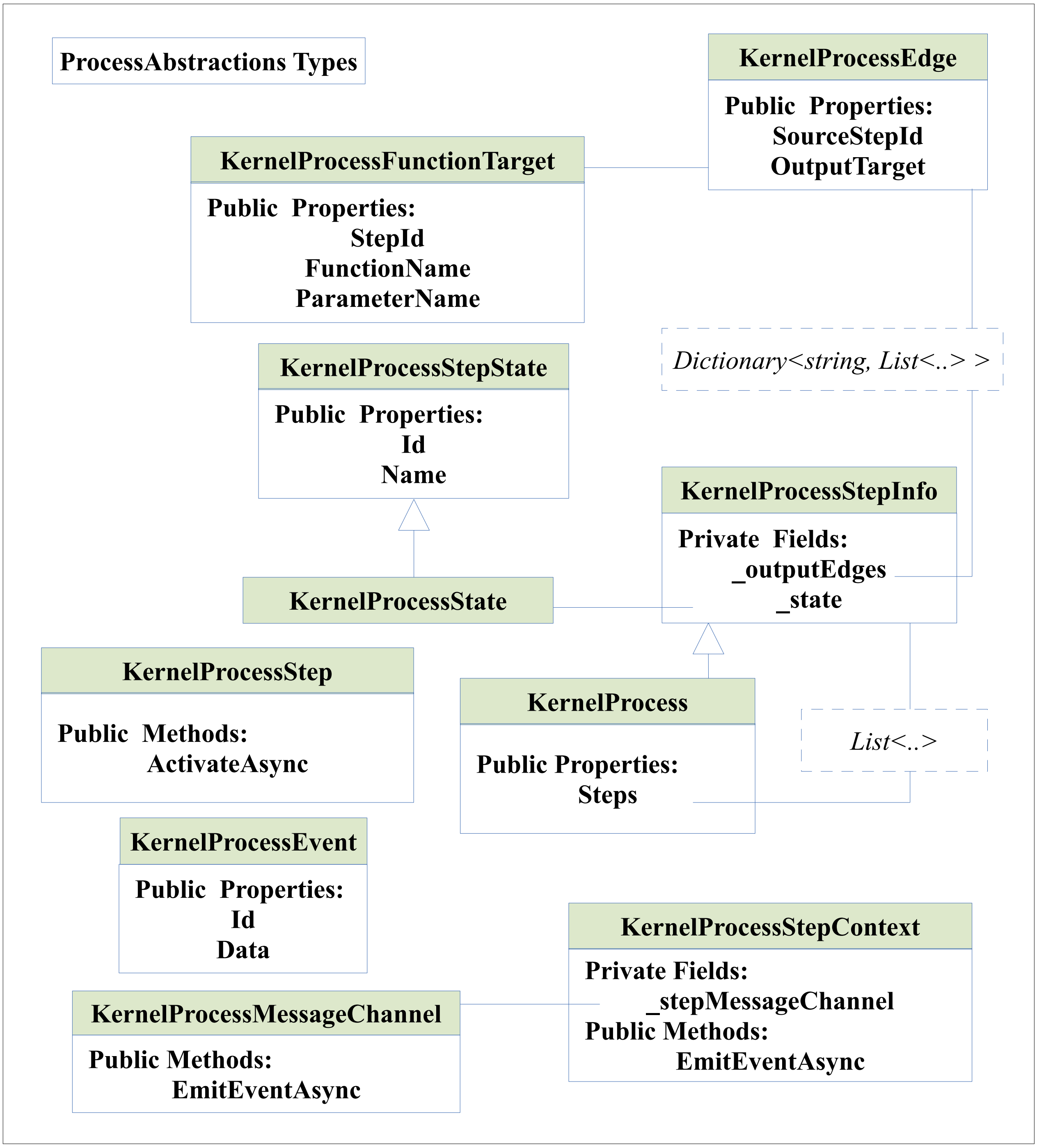
The code for the ProcessAbstractions assembly is here:

- https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Experimental/Process.Abstractions

and is comprised of the following files:

AssemblyInfo.cs	KernelProcessState.cs
KernelProcess.cs	KernelProcessStep.cs
KernelProcessEdge.cs	KernelProcessStepContext.cs
KernelProcessEvent.cs	KernelProcessStepInfo.cs
KernelProcessFunctionTarget.cs	KernelProcessStepState.cs
KernelProcessMessageChannel.cs	Process.Abstractions.csproj

The main types within this assembly are structured as follows:



The `KernelProcessFunctionTarget` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessFunctionTarget.cs

is defined as:

```
public record KernelProcessFunctionTarget {
    public KernelProcessFunctionTarget(string stepId,
                string functionName, string? parameterName = null){ .. }
```

```
    public string StepId { get; init; }
    public string FunctionName { get; init; }
    public string? ParameterName { get; init; }
}
```

It essentially maps a (string) step Id to a function name and optionally a parameter name.

The `KernelProcessEdge` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessEdge.cs

is defined as:

```
public sealed class KernelProcessEdge {
    public string SourceStepId { get; }
    public KernelProcessFunctionTarget OutputTarget { get; }
    public KernelProcessEdge(string sourceStepId,
                    KernelProcessFunctionTarget outputTargets) { .. }
}
```

It represents a named edge in a process graph.

The `KernelProcessStepState` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessStepState.cs

is defined as:

```
public record KernelProcessStepState {
    public string? Id { get; init; }
    public string? Name { get; init; }
    public KernelProcessStepState(
                    string? id = null, string? name = null) { .. }
}
```

Additionally, for the user-defined state, it has this derived record:

```
public sealed record KernelProcessStepState<TState>
       : KernelProcessStepState where TState : class, new() {
    public TState? State { get; set; }
    public KernelProcessStepState(
                string? id = null, string? name = null) { .. }
}
```

The `KernelProcessState` type derives from this:

```
public sealed record KernelProcessState : KernelProcessStepState { }
```

The `KernelProcessStepInfo` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessStepInfo.cs

is defined as:

```
public record KernelProcessStepInfo {
    private KernelProcessStepState _state;
    private readonly Dictionary<
                    string, List<KernelProcessEdge>> _outputEdges;
```

```
    public Type InnerStepType { get; }

    public KernelProcessStepState State { .. }

    public IReadOnlyDictionary<string,
      IReadOnlyCollection<KernelProcessEdge>> Edges =>
     this._outputEdges.ToDictionary(kvp => kvp.Key,
      kvp => (IReadOnlyCollection<KernelProcessEdge>)kvp.Value.AsReadOnly());

    public KernelProcessStepInfo(Type innerStepType,
                KernelProcessStepState state,
                Dictionary<string, List<KernelProcessEdge>> edges) { .. }
}
```

It manages three key pieces of information about a step (all of which are verified to be not null): the state, the inner type and the edges.

The `KernelProcess` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcess.cs

derives from `KernelProcessStepInfo` and is defined as:

```
public sealed record KernelProcess : KernelProcessStepInfo
{
    public IList<KernelProcessStepInfo> Steps { get; }

    public KernelProcess(string name, IList<KernelProcessStepInfo> steps,
       Dictionary<string, List<KernelProcessEdge>>? edges = null)
        : base(typeof(KernelProcess), new KernelProcessState()
           { Name = name }, edges ?? []) { .. }
}
```

The `KernelProcessStep` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessStep.cs

comes in two versions, untyped and derived from that, typed:

```
public class KernelProcessStep
{
    public virtual ValueTask ActivateAsync(KernelProcessStepState state)
    {
        return default;
    }
}

public class KernelProcessStep<TState>
              : KernelProcessStep where TState : class, new()
{
    public virtual ValueTask ActivateAsync(
                             KernelProcessStepState<TState> state)
    {
        return default;
    }
}
```

It defines the `ActivateSync()` virtual method.

The `KernelProcessEvent` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessEvent.cs

is defined as:

```
public sealed record KernelProcessEvent {
    public string? Id { get; set; }
    public object? Data { get; set; }
}
```

and just represents an event (string) id and event data (a custom object).

The `KernelProcessMessageChannel` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessMessageChannel.cs

is defined as:

```
public abstract class KernelProcessMessageChannel {
    public abstract ValueTask EmitEventAsync(
                                  KernelProcessEvent processEvent);
}
```

It represents a channel for emitting events from a step.

The `KernelProcessStepContext` type:

- https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Experimental/Process.Abstractions/KernelProcessStepContext.cs

is defined as:

```
public sealed class KernelProcessStepContext
{
    private readonly KernelProcessMessageChannel _stepMessageChannel;

    public KernelProcessStepContext(KernelProcessMessageChannel channel)
    {
        this._stepMessageChannel = channel;
    }

    public ValueTask EmitEventAsync(KernelProcessEvent processEvent)
    {
        return this._stepMessageChannel.EmitEventAsync(processEvent);
    }
}
```

It has a method called `EmitEventAsync` which in turn calls `EmitEventAsync` on the message channel.