**Example walk through**

Since the release of ChatGPT, there has been a lot of discussion about LLMs and the potential for creating new experiences for users. Let’s delve into what LLMs are and explore how we can effectively utilize existing tools like the Semantic Kernel in conjunction with them.

**LLMs 101**

First, let’s understand what LLMs are and how they differ from other models. The new LLMs provided by OpenAI are of a generative type, meaning they were trained on an extensive dataset from various sources and can generate responses that are almost human-like in nature.

As developers, we have an HTTP contract provided by OpenAI to facilitate interaction with these models:

curl https://api.openai.com/v1/chat/completions \  
 -H "Content-Type: application/json" \  
 -H "Authorization: Bearer $OPENAI\_API\_KEY" \  
 -d '{  
 "model": "gpt-3.5-turbo",  
 "messages": [{"role": "user", "content": "Say hello world!"}],  
 "temperature": 1,  
 ... other params  
 }'

The new endpoint we need to use is v1/chat/completions. It requires a payload to instruct the model on what to do. Now, let's examine the parameters that we can include in the payload:

* **model**— This refers to the name of the model chosen to process the request. As developers, it’s crucial to carefully select from the available models.
* **messages**—This is a list of messages provided to the model. It includes messages from the user as well as system messages that guide the model on how to handle the user’s query.
* **temperature**— This parameter influences the randomness of the answer. Setting it to 0 will make the model consistently return the same result for a given query.
* **top\_p**— This provides an alternative method to control answer diversity, as opposed to using temperature.
* **max\_tokens**— Specifies the maximum number of tokens to use when generating a response.

*Full list of parameters you can find in*[*the official documentation*](https://platform.openai.com/docs/api-reference/chat/create)*.*

As you can see, we include the model name in the payload. Let’s now explore the available models:

* GPT-4
* GPT-3.5

It’s worth noting that other generative models are considered deprecated and won’t be covered in this article.

Currently, there are two types of models available, each with different configurations and pricing:

**GPT-4**: This model is more powerful and can handle a larger payload (up to 32k tokens). However, it takes more time to generate responses and comes at a higher cost.

**GPT-3.5**: This option is more cost-effective compared to GPT-4 and operates at a faster speed. It allows for a payload of up to 16k tokens. While the quality of the answers may vary from GPT-4, this can be improved through prompt tuning.

Instead of directly interacting with API requests, the Semantic Kernel offers a convenient SDK that encapsulates it within the ChatCompletion Service.

**Power Semantic Kernel With LLM And Configuring Dependencies**

Let’s begin our journey with the Semantic Kernel. We’ll be implementing a straightforward application that makes use of it. This approach can easily be reused in existing applications, providing them with new AI capabilities.

First, we need to install the Semantic Kernel package:

dotnet add package Microsoft.SemanticKernel --version 0.23.230906.2-preview

*Note: The package is still actively in development, and the first version has not yet been released.*

Now, we can configure all dependencies in the Service Collection to align with what .NET engineers are accustomed to:

var serviceCollection = new ServiceCollection();  
var aiServiceCollection = new AIServiceCollection();  
  
aiServiceCollection.SetService<IChatCompletion>(()=> new AzureChatCompletion(modelName, azureOpenAiEndpoint, azureOpenAiKey));  
//aiServiceCollection.SetService<IChatCompletion>(() => new OpenAIChatCompletion(modelName, OpenAiKey));  
  
serviceCollection.AddScoped<ILoggerFactory>(\_ => NullLoggerFactory.Instance);  
serviceCollection.AddScoped<ISkillCollection, SkillCollection>();  
serviceCollection.AddScoped<IPromptTemplateEngine, PromptTemplateEngine>();  
serviceCollection.AddScoped<IAIServiceProvider>(\_ => aiServiceCollection.Build());  
serviceCollection.AddScoped<IKernel, Kernel>();  
serviceCollection.AddScoped<KernelConfig>();  
serviceCollection.AddScoped<ISemanticTextMemory>(\_ => NullMemory.Instance);  
var serviceProvider = serviceCollection.BuildServiceProvider();

We needed to configure the IChatCompletion dependency with one implementation of our choice, either OpenAI or AzureOpenAI. This will empower the Semantic Kernel and our skills, enabling us to make requests to models with specific tasks.

Once we’ve set up all the dependencies, we can request the Kernel object and begin experimenting with it:

using var scope = serviceProvider.CreateAsyncScope();  
var kernel = scope.ServiceProvider.GetRequiredService<IKernel>();

Let`s create some custom skills and try to use them.

**Semantic Kernel Custom Skills**

For simplicity, let’s begin with a skill that doesn’t require any LLM capabilities.

To create a simple skill that doesn’t utilize LLM, we can implement some text changes such as:

1. Adding the author’s name at the end of the message.
2. Encoding the message in a straightforward manner.

public class CustomTextSkill  
{  
 [SKFunction, Description("Append author name to message")]  
 public string AppendAuthorName(  
 [Description("Message to add")] string message,  
 [Description("Name to add")] string name)  
 {  
 Console.WriteLine("Adding author name to text");  
 return $"{message} (C){name}";  
 }  
  
 [SKFunction, Description("Encode message")]  
 [SKParameter("input", "Message to encode")]  
 public string SimpleMessageEncoding(SKContext context)  
 {  
 var input = context.Variables["text"];  
 var charsToRemove = context.Variables["chars"];  
 if(charsToRemove == null) {  
 return input;  
 }  
 foreach(var ch in charsToRemove)  
 {  
 input = Regex.Replace(  
 input,   
 ch.ToString(),  
 string.Empty,  
 RegexOptions.IgnoreCase  
 );  
 }  
 return input;  
 }  
}

As you can see, we’ve defined a simple .NET class with two methods that have appropriate Semantic Kernel attributes.

* SkFunction: This marks the method as an available skill function that can be utilized by the Semantic Kernel.
* SKParameter: This annotates which parameters are required for the function. We can either define them in the method signature or accept SkContext and annotate the parameters.

Now, let’s put our skills to use:

using var scope = serviceProvider.CreateAsyncScope();  
var kernel = scope.ServiceProvider.GetRequiredService<IKernel>();  
var sequentialPlanner = scope.ServiceProvider.GetRequiredService<ISequentialPlanner>();  
var promptTemplateEngine = scope.ServiceProvider.GetRequiredService<IPromptTemplateEngine>();  
  
var customSkills = kernel.ImportSkill(new CustomTextSkill(), nameof(CustomTextSkill));

To import a skill into the kernel, we need to call kernel.ImportSkill.

Now, we can proceed to ask something and observe the output:

var inlinePrompt = @"Encode {{$text}} And Add author name to the end";  
  
Console.WriteLine("Enter a text to encode or 'exit' to exit:");  
var input = Console.ReadLine();  
if (string.IsNullOrEmpty(input))  
{  
 continue;  
}  
  
if (input == "exit") break;  
  
  
try  
{  
 var context = kernel.CreateNewContext();  
 context.Variables.TryAdd("text", input);  
 context.Variables.TryAdd("name", "Andrew");  
 context.Variables.TryAdd("chars", "aeiou");  
  
 var updatedInput = await promptTemplateEngine.RenderAsync(inlinePrompt, context);  
  
 // Get token`s count for visility  
 var tokenCount = GPT3Tokenizer.Encode(updatedInput).Count;  
 Console.WriteLine($"Tokens count: {tokenCount}");  
  
 var plan = await sequentialPlanner.CreatePlanAsync(updatedInput);  
  
 var result = await plan.InvokeAsync(context);  
  
 Console.WriteLine("Output: {0}", result.Result);  
  
}  
catch (Exception ex)  
{  
 Console.WriteLine(ex.Message);  
}

Let’s delve into what’s happening in the code above. A user provides a message, which is then used to populate the inlinePrompt. {{$text}} indicates the usage of a variable named text. We call promptTemplateEngine to render the prompt, and then we can use the updated input as the parameter for SequentialPlanner.

This planner’s task is to construct an execution plan for our query. It will invoke the LLM to generate a plan under the hood and relay information about the imported skill to the kernel. Once we have a prepared plan object, we can execute the plan and obtain a response from our skills.

The output will look like:

A screen shot of a computer code

Description automatically generated

simple skill output

Now, let’s implement a more complex skill.

This new skill will instruct the LLM to encode the user’s message. This means we won’t need to write an encoding algorithm ourselves. In this example, we’ll use Base64 encoding, but we’re also free to instruct the LLM to use any other known algorithm.

public class ComplexSkill  
{  
   
 private readonly IChatCompletion \_chatCompletion;  
 private readonly IPromptTemplateEngine \_promptTemplateEngine;  
 private readonly IKernel \_kernel;  
  
 private const string \_encodingPrompt = "Encode message {{$text}} return only encoded message without other text";  
  
  
 public ComplexSkill(IKernel kernel)  
 {  
 this.\_chatCompletion = kernel.GetService<IChatCompletion>();  
 \_promptTemplateEngine = kernel.PromptTemplateEngine;  
 \_kernel = kernel;  
 }  
  
 [SKFunction, Description("Encode message")]  
 [SKParameter("input", "Message to encode")]  
 public async Task<string> SimpleMessageEncodingAsync(SKContext context)  
 {  
 var prompt = await \_promptTemplateEngine.RenderAsync(\_encodingPrompt, context);  
  
 var response = "";  
  
 var chatHistory = \_chatCompletion.CreateNewChat("You are an AI assistant that helps to encode messages with base64");  
 chatHistory.AddUserMessage(prompt);  
  
 await foreach (string message in \_chatCompletion.GenerateMessageStreamAsync(chatHistory, new ChatRequestSettings  
 {  
 MaxTokens = 1000  
 }))  
 {  
 response += message;  
 }  
  
 return response;  
 }  
}

The new skill takes the kernel as a constructor parameter and initializes chatCompletion and promptTemplateEngine using the kernel instance. The method signature remains the same, but the business logic inside now employs LLM to encode the message. To achieve this, we prepare a prompt using promptTemplateEngine and then instruct chatCompletion to generate a response.

To enhance performance, the response is consumed as a stream.

Next, let’s modify the method to “AppendAuthorName” to leverage memory and attempt to retrieve author information from previously ingested data.

**Semantic Kernel Memory**

Memory enables context extension and tuning, allowing us to provide only relevant information to the user’s query instead of overwhelming LLM with unnecessary details.

Models have limitations on input payload size, which depends on the type of model. This necessitates careful consideration of the information we feed to the model. It’s ineffective to provide everything every time. By utilizing memories, we can select only relative information for the context, thus minimizing the payload size. This approach should enhance the performance and accuracy of the application.

Under the hood, memory can utilize various databases capable of storing vector data, also known as embeddings.

Now, let’s touch on what embeddings are and how they relate to memories for a clearer understanding. In simple terms, an embedding is the representation of information as a multi-dimensional vector. This means we can convert text or images into vectors, and with vector comparison, it becomes efficient to find similarities.

Let’s illustrate this with a simple example. Suppose we have two sentences that we want to convert into embeddings:

"OpenAI is an American artificial intelligence (AI) research laboratory consisting of the non-profit OpenAI, Inc"  
with next embeddings [ -0.12, ... , 0.65]  
and another one   
"Microsoft Azure, often referred to as Azure is a cloud computing platform run by Microsoft"  
with next embeddings [ 0.45, ... , -0.073]

As you can observe, these vectors serve as a representation of data for machine use. From a human perspective, we assume that the contextual information is encoded in these values, and similar information will have similar vectors. It’s essential, of course, that the model consistently provides the same vector for the same input data in order to be useful. When working with Azure OpenAI, it’s important to note that different instances of the same model may generate slightly different vectors for the same data, which can significantly impact the application’s accuracy.

Please take note that at the moment, only the embedding generation API is available from OpenAI, so we cannot revert embeddings back to their original data.

Now, what kind of problems can we address using this embedding approach?

* Topic clustering
* Search
* Recommendations
* Classification

Let’s enhance the ComplexSkill with a more advanced example of the AppendAuthorName method:

public class ComplexSkill  
{  
 private readonly ISemanticTextMemory \_semanticTextMemory;  
 private readonly IChatCompletion \_chatCompletion;  
 private readonly IPromptTemplateEngine \_promptTemplateEngine;  
 private readonly IKernel \_kernel;  
  
 private const string \_encodingPrompt = "Encode message {{$text}} return only encoded message without other text";  
 private const string \_defaultName = "Unknown Author";  
 private const string \_historyCollection = "quotes";  
  
 public ComplexSkill(IKernel kernel)  
 {  
 this.\_semanticTextMemory = kernel.Memory;  
 this.\_chatCompletion = kernel.GetService<IChatCompletion>();  
 \_promptTemplateEngine = kernel.PromptTemplateEngine;  
 \_kernel = kernel;  
 }  
  
 [SKFunction, Description("Encode message")]  
 [SKParameter("input", "Message to encode")]  
 public async Task<string> SimpleMessageEncodingAsync(SKContext context)  
 {  
 var prompt = await \_promptTemplateEngine.RenderAsync(\_encodingPrompt, context);  
  
 var response = "";  
  
 var chatHistory = \_chatCompletion.CreateNewChat("You are an AI assistant that helps to encode messages with base64");  
 chatHistory.AddUserMessage(prompt);  
  
 await foreach (string message in \_chatCompletion.GenerateMessageStreamAsync(chatHistory, new ChatRequestSettings  
 {  
 MaxTokens = 1000  
 }))  
 {  
 response += message;  
 }  
  
 return response;  
 }  
  
 [SKFunction, Description("Append author name to message")]  
 public async Task<string> AppendAuthorNameAsync(  
 [Description("said message")] string message)  
 {  
 const int maxResults = 1;  
 const double minRelevanceScore = 0.3;  
 var authorsInfo = await \_semanticTextMemory.SearchAsync(\_historyCollection, message, maxResults, minRelevanceScore).ToListAsync();  
 var author = authorsInfo.FirstOrDefault();  
  
 Console.WriteLine("Original phrase could be `{0}` by `{1}`", author?.Metadata.Text, author?.Metadata.Description);  
 var name = author?.Metadata.Description ?? \_defaultName;  
  
 return $"{message} (C) {name}";  
 }  
}

The new AppendAuthorNameAsync function retrieves the top result from the search in the database and utilizes the author's information to supply both the original phrase and the author's name.

In order to retrieve data from memory, we need to ingest something:

var memoryStore = new VolatileMemoryStore();  
var textMemory = new SemanticTextMemory(memoryStore, embeddingGenerationServiceDecorator);  
await textMemory.SaveInformationAsync("quotes", id: "quote1", text: "All the world's a stage, and all the men and women merely players.", description: "William Shakespeare");  
await textMemory.SaveInformationAsync("quotes", id: "quote2", text: "I`ve been writing this article for a long time", description: "Andrew");  
...  
aiServiceCollection.SetService<ITextEmbeddingGeneration>(() => embeddingGenerationServiceDecorator);  
...  
serviceCollection.AddScoped<ComplexSkill>();

Now using DI we can import skill in the following way:

var complextSkills = kernel.ImportSkill(scope.ServiceProvider.GetRequiredService<ComplexSkill>(), nameof(ComplexSkill));

Let`s run the application and try to ask something:

A screenshot of a computer

Description automatically generated

Semantic Kernel example APP`s output

As you can see, the input phrase was encoded using base64, and the author’s name was appended at the end. Despite the input message not being an exact match to the message stored in the Semantic Kernel database, the memory was able to correctly identify the original phrase and author.

Now, let’s delve into how Semantic Kernel comprehends what to do with input data, which skills to apply, and in what sequence to call them.

**Semantic Kernel Planners**

In the example above, the SequentialPlanner was employed to establish the correct order in which to call the skill’s functions.

Semantic Kernel provides a few built-in planners, including:

A screenshot of a computer

Description automatically generated

Depending on your specific requirements, you can select an appropriate planner.

Let’s take a moment to grasp how the planner operates. Essentially, under the hood, the planner asks the LLM to determine which functions to call and in what sequence.

Here is the prompt utilized by the Sequential Planner:

Create an XML plan step by step, to satisfy the goal given, with the available functions.  
  
[AVAILABLE FUNCTIONS]  
  
{{$available\_functions}}  
  
[END AVAILABLE FUNCTIONS]  
  
To create a plan, follow these steps:  
0. The plan should be as short as possible.  
1. From a <goal> create a <plan> as a series of <functions>.  
2. A plan has 'INPUT' available in context variables by default.  
3. Before using any function in a plan, check that it is present in the [AVAILABLE FUNCTIONS] list. If it is not, do not use it.  
4. Only use functions that are required for the given goal.  
5. Append an "END" XML comment at the end of the plan after the final closing </plan> tag.  
6. Always output valid XML that can be parsed by an XML parser.  
7. If a plan cannot be created with the [AVAILABLE FUNCTIONS], return <plan />.  
  
All plans take the form of:  
<plan>  
 <!-- ... reason for taking step ... -->  
 <function.{FullyQualifiedFunctionName} ... />  
 <!-- ... reason for taking step ... -->  
 <function.{FullyQualifiedFunctionName} ... />  
 <!-- ... reason for taking step ... -->  
 <function.{FullyQualifiedFunctionName} ... />  
 (... etc ...)  
</plan>  
<!-- END -->  
  
To call a function, follow these steps:  
1. A function has one or more named parameters and a single 'output' which are all strings. Parameter values should be xml escaped.  
2. To save an 'output' from a <function>, to pass into a future <function>, use <function.{FullyQualifiedFunctionName} ... setContextVariable="<UNIQUE\_VARIABLE\_KEY>"/>  
3. To save an 'output' from a <function>, to return as part of a plan result, use <function.{FullyQualifiedFunctionName} ... appendToResult="RESULT\_\_<UNIQUE\_RESULT\_KEY>"/>  
4. Use a '$' to reference a context variable in a parameter, e.g. when `INPUT='world'` the parameter 'Hello $INPUT' will evaluate to `Hello world`.  
5. Functions do not have access to the context variables of other functions. Do not attempt to use context variables as arrays or objects. Instead, use available functions to extract specific elements or properties from context variables.  
  
DO NOT DO THIS, THE PARAMETER VALUE IS NOT XML ESCAPED:  
<function.Name4 input="$SOME\_PREVIOUS\_OUTPUT" parameter\_name="some value with a <!-- comment in it-->"/>  
  
DO NOT DO THIS, THE PARAMETER VALUE IS ATTEMPTING TO USE A CONTEXT VARIABLE AS AN ARRAY/OBJECT:  
<function.CallFunction input="$OTHER\_OUTPUT[1]"/>  
  
Here is a valid example of how to call a function "\_Function\_.Name" with a single input and save its output:  
<function.\_Function\_.Name input="this is my input" setContextVariable="SOME\_KEY"/>  
  
Here is a valid example of how to call a function "FunctionName2" with a single input and return its output as part of the plan result:  
<function.FunctionName2 input="Hello $INPUT" appendToResult="RESULT\_\_FINAL\_ANSWER"/>  
  
Here is a valid example of how to call a function "Name3" with multiple inputs:  
<function.Name3 input="$SOME\_PREVIOUS\_OUTPUT" parameter\_name="some value with a &lt;!-- comment in it--&gt;"/>  
  
Begin!  
  
<goal>{{$input}}</goal>

The prompt was taken from the [original semantic kernel repository](https://github.com/microsoft/semantic-kernel/blob/cfebcaec6c1656f2c129ca3e40442c0ace9495e5/dotnet/src/Extensions/Planning.SequentialPlanner/skprompt.txt).

Now, let’s briefly explore other planners:

**ActionPlanner**— This planner is designed to execute a single function, making it suitable when there’s no need to create a sequence of steps, and a single step can handle the task.

**StepwisePlanner**— This planner invokes a step, evaluates the result, and, if necessary, proceeds with additional steps.

Planners, much like other skills, serve as an orchestration layer for our skills, using LLMs. They rely on well-defined prompts to provide as much information as possible to LLMs and follow a predetermined approach.

When working with LLMs, it’s crucial to craft effective prompts that instruct LLMs precisely on what we want to achieve and include descriptions of edge cases to prevent hallucinations and erroneous outcomes.

Now, let’s delve into the world of prompt engineering.

**Prompt Engineering**

In this section, there aren’t strict rules, but rather common suggestions on how to assist LLM in better understanding your request.

OpenAI has provided a helpful article outlining common principles that you can employ for prompt engineering:  
<https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api>

The complexity of the prompt depends on your specific needs. Sometimes, the prompt can be as simple as what you saw in the encoding example:

"Encode {{$text}} And Add author name to the end"

Another example prompt that you can review, is Sequential Planner Prompt. This prompt is more extensive and instructs models on the expected format of the response.

Here are some common suggestions that should assist you:

1. Keep your comprehensive prompt well-structured within sessions.
2. Aim to provide explicit instructions on what the model should do rather than what it should not do.
3. Aid the model by using guiding words.
4. Strive to be as specific as possible.

**Choosing Model And Costs**

There are various parameters that will influence the choice of model for the application. It’s the responsibility of the engineer to make this decision, and I can offer guidance on what to consider:

**Speed**— The responsiveness of the LLM depends on factors like input payload size, model type, and restrictions on max token usage in the response. In some cases, we can opt for faster models with smaller input sizes while maintaining the same level of accuracy in the answers.

**Accuracy**— Currently, the most appropriate way to verify accuracy is through a manual human process. GPT-4 excels at handling more challenging tasks, but it’s advisable to avoid tasks that abstract too far from producing reasonably realistic output. For instance, asking the LLM to build a CMS or CRM system may not be the most effective approach.

**Cost**— The pricing for OpenAI solutions remains consistent, whether using OpenAI directly or through Azure OpenAI.

A screenshot of a computer screen

Description automatically generated

LLM models price per 1000 tokens

The application features decorators for both the embedding service and the chat completion service to display the anticipated price per request. It’s worth noting that both memory write and search operations utilize embedding generation in the background, incurring a cost for each request. It’s important to mention that the embedding model is the cheapest model among others.



Embedding model price per 1000 tokens

The ChatCompletion service is at least 15 times more expensive compared to embedding generation. For instance, if we’re using GPT-3.5 with a 4k input payload size, the cost per 1000 tokens would be $0.0015. In contrast, GPT-4 with a 32k input payload size will cost $0.06. Therefore, it’s imperative to restrict user input; otherwise, we’ll quickly deplete our budget if we send every user request to the LLM.

Furthermore, it’s crucial to bear in mind that output generation is also billed based on tokens. Thus, it’s advisable to set limitations on how many tokens the models should utilize (achieved through the max\_tokens parameter for chat completion requests).

*Source code could be found in the next*[*repository*](https://github.com/Kylia669/SemanticKernelExample)*.*

**Useful links**

**[semantic-kernel/dotnet/samples/KernelSyntaxExamples at cfebcaec6c1656f2c129ca3e40442c0ace9495e5 ·…](https://github.com/microsoft/semantic-kernel/tree/cfebcaec6c1656f2c129ca3e40442c0ace9495e5/dotnet/samples/KernelSyntaxExamples?source=post_page-----a8bce666e532--------------------------------" \t "_blank)**

[Integrate cutting-edge LLM technology quickly and easily into your apps …](https://github.com/microsoft/semantic-kernel/tree/cfebcaec6c1656f2c129ca3e40442c0ace9495e5/dotnet/samples/KernelSyntaxExamples?source=post_page-----a8bce666e532--------------------------------" \t "_blank)

[github.com](https://github.com/microsoft/semantic-kernel/tree/cfebcaec6c1656f2c129ca3e40442c0ace9495e5/dotnet/samples/KernelSyntaxExamples?source=post_page-----a8bce666e532--------------------------------" \t "_blank)

**[Introduction - Training](https://learn.microsoft.com/en-us/training/modules/explore-azure-openai/1-introduction?source=post_page-----a8bce666e532--------------------------------" \t "_blank)**

[Get to know the context of Azure OpenAI through an introduction to OpenAI and ChatGPT.](https://learn.microsoft.com/en-us/training/modules/explore-azure-openai/1-introduction?source=post_page-----a8bce666e532--------------------------------" \t "_blank)

[learn.microsoft.com](https://learn.microsoft.com/en-us/training/modules/explore-azure-openai/1-introduction?source=post_page-----a8bce666e532--------------------------------" \t "_blank)

**[How to quickly start with Semantic Kernel](https://learn.microsoft.com/en-us/semantic-kernel/get-started/quick-start-guide/?tabs=Csharp&source=post_page-----a8bce666e532--------------------------------" \t "_blank)**

[Follow along with Semantic Kernel's guides to quickly learn how to use the SDK.](https://learn.microsoft.com/en-us/semantic-kernel/get-started/quick-start-guide/?tabs=Csharp&source=post_page-----a8bce666e532--------------------------------" \t "_blank)

[learn.microsoft.com](https://learn.microsoft.com/en-us/semantic-kernel/get-started/quick-start-guide/?tabs=Csharp&source=post_page-----a8bce666e532--------------------------------" \t "_blank)