# OpenAI

SUCCINCTLY®

*BY* **MICHAEL WASHINGTON**

SUCCINCTLY® EBOOK SERIES

Syncfusion®

# OpenAI Succinctly®

**Michael Washington**

Foreword by Daniel Jebaraj

Copyright © 2025 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 111
Morrisville, NC 27560
USA

All rights reserved.

ISBN: 978-1-64200-245-4

**Important licensing information. Please read.**

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

BOLDDESK, BOLDSIGN, BOLD BI, BOLD REPORTS, SYNCFUSION, ESSENTIAL, ESSENTIAL STUDIO, SUCCINCTLY, and the 'Cody' mascot logo are the registered trademarks of Syncfusion, Inc.

**Technical Reviewer:** James McCaffrey

**Copy Editor:** Courtney Wright

**Acquisitions Coordinator:** Tres Watkins, VP of content, Syncfusion, Inc.

**Proofreader:** Jacqueline Bieringer, content producer, Syncfusion, Inc.

# Table of Contents

# The *Succinctly*® Series of Books

Daniel Jebaraj
CEO of Syncfusion®, Inc.

When we published our first *Succinctly*® series book in 2012, *jQuery Succinctly*®, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 1.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

**Let us know what you think**

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly*® series!

# About the Author

Michael Washington is a Microsoft MVP, an ASP.NET C# programmer, and the founder of BlazorHelpWebsite.com. He is the author of over 16 books including *An Introduction to Building Applications with Blazor*.

Michael has extensive knowledge of process improvement, billing systems, and student information systems. He has a son, Zachary, and resides in Los Angeles with his wife Valerie.

You can follow him on X at @ADefWebserver and on Mastodon at https://hachyderm.io/@Adefwebserver.

# Chapter 1 OpenAI and GPT

*Figure 1: Abstract Image Generated by OpenAI DALL-E*

In recent years, artificial intelligence has taken huge strides in its ability to understand, process, and generate human language. One of the most promising developments in this field has been the creation of large language models (LLM) by OpenAI.

These models have garnered a great deal of attention and excitement due to their remarkable ability to generate human-like text that is often indistinguishable from something a human might write.

As a result, OpenAI's models have become a key area of research and development in the field of natural language and image processing, and they are increasingly being integrated into a wide range of applications across a variety of industries.

## History of OpenAI

OpenAI consists of the for-profit technology company OpenAI LP and its parent company, the nonprofit OpenAI, Inc. OpenAI was founded in December 2015 by Sam Altman, Greg Brockman, Reid Hoffman, Jessica Livingston, Peter Thiel, and Elon Musk with the stated goal of promoting and developing friendly AI in a way that benefits humanity. Since then:

- **2018**: OpenAI formed a partnership with Microsoft to jointly develop AI technologies, with a focus on ensuring that AI would be developed and used in a responsible and ethical manner.

- **2019**: OpenAI launched its OpenAI API, which allows developers to integrate advanced AI capabilities into their own applications easily.
- **2022**: OpenAI released an AI chatbot called ChatGPT, and it became popular, with over a million people signing up to use it within the first five days.
- **2023**: Microsoft announced that it would be investing several billion dollars in OpenAI over several years.

## About GPT

Generative pretrained transformer (GPT) is a type of language model developed by OpenAI that is based on the transformer architecture. A *transformer* is a type of neural network architecture that is used in LLMs (large language models) to process sequential data, such as text or speech. It is trained on a large amount of text data to predict the next word in a sequence given the previous words, with the goal of generating human-like text.

The *generative* part of GPT refers to its ability to generate text. It can be used to generate new text by sampling from its learned distribution of word probabilities. This can be used for a variety of tasks, such as language translation, text summarization, and even creative writing.

The *pretrained* aspect of GPT refers to the fact that the model is first trained on a large amount of text data before being fine-tuned on a specific task. This allows the model to learn general patterns of language use, which can be applied to a wide range of tasks without requiring large amounts of task-specific data.

GPT's ability to generate human-like text and perform a wide range of tasks has made it one of the most widely used language models in the industry and has helped to drive the growth of the AI industry.

The following are the types of things GPT models can produce:

- **Words**: It can create original text content and help summarize long and complicated pieces of writing so that people can understand it better. It can also suggest diverse ways to say things in writing.
- **Code**: It can write original computer code, based on a description of the desired functionality, or translate computer code from one language to another and find problems in the code.

# Chapter 2  Using OpenAI GPT

## Anatomy of a prompt

In the context of a GPT model, a prompt is a piece of text that is given to the model as input, which it then uses to generate additional text. The prompt can be a few words or a longer paragraph, depending on the task at hand.

For example, if the task is to generate a blog post or an email, the prompt might be a few words summarizing the topic, such as:

> *Rising interest rates will require the organization to reduce expenses*

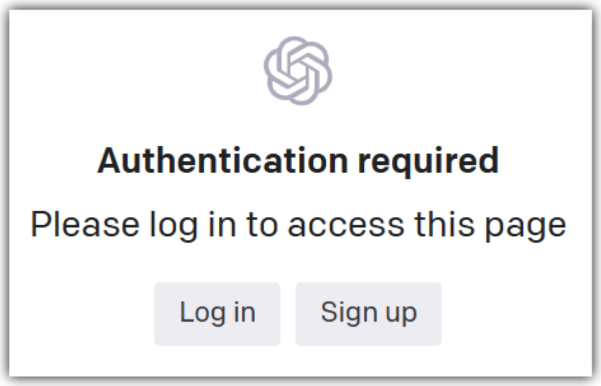
Or a longer paragraph outlining the key details such as:

> *#1 Non-essential expenses will need to be eliminated #2 Overtime will require approval from the Director.*

The GPT model uses the prompt as a starting point to generate additional text that follows the same style, tone, and content as the input prompt.

## Use the OpenAI Playground

The online OpenAI Playground is a powerful tool for exploring the capabilities of the OpenAI models. It is web-based and provides an easy-to-use environment for experimenting with the OpenAI technologies.

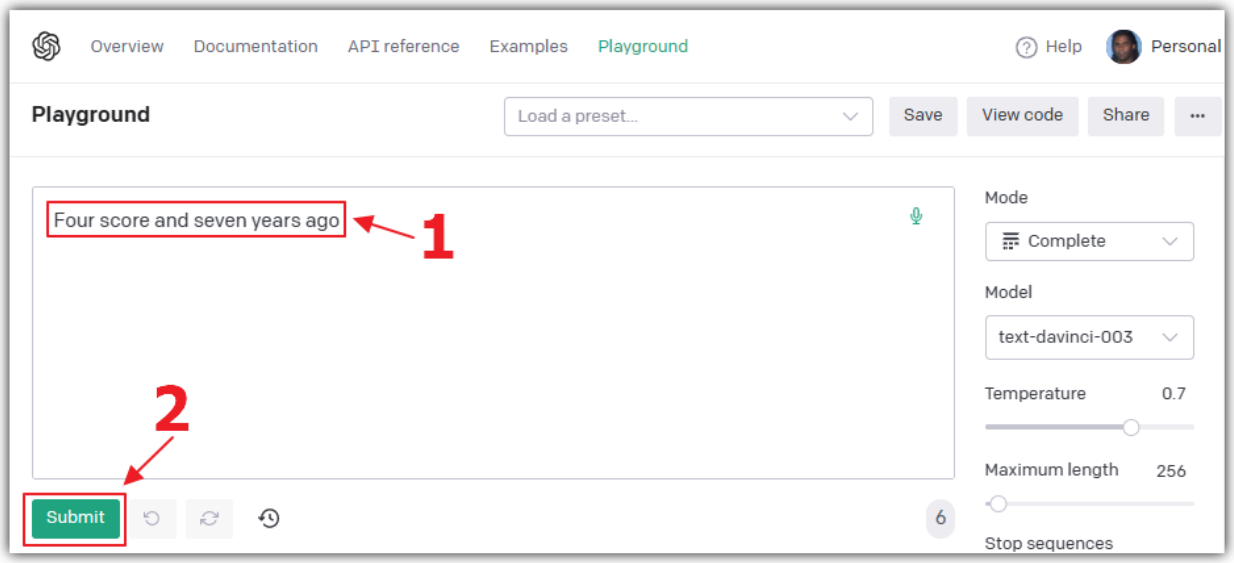

**Authentication required**

Please log in to access this page

*Figure 2: Log in to the OpenAI Playground*

In your web browser, navigate to the OpenAI website at https://platform.openai.com/playground and click **Log in**, or click **Sign up** if you do not already have an OpenAI account.

Click on the confirmation link in the email that OpenAI sends you. Once you have confirmed your email address, log in to your OpenAI account.



Four score and seven years ago

Mode: Complete

Model: text-davinci-003

Temperature 0.7

Maximum length 256

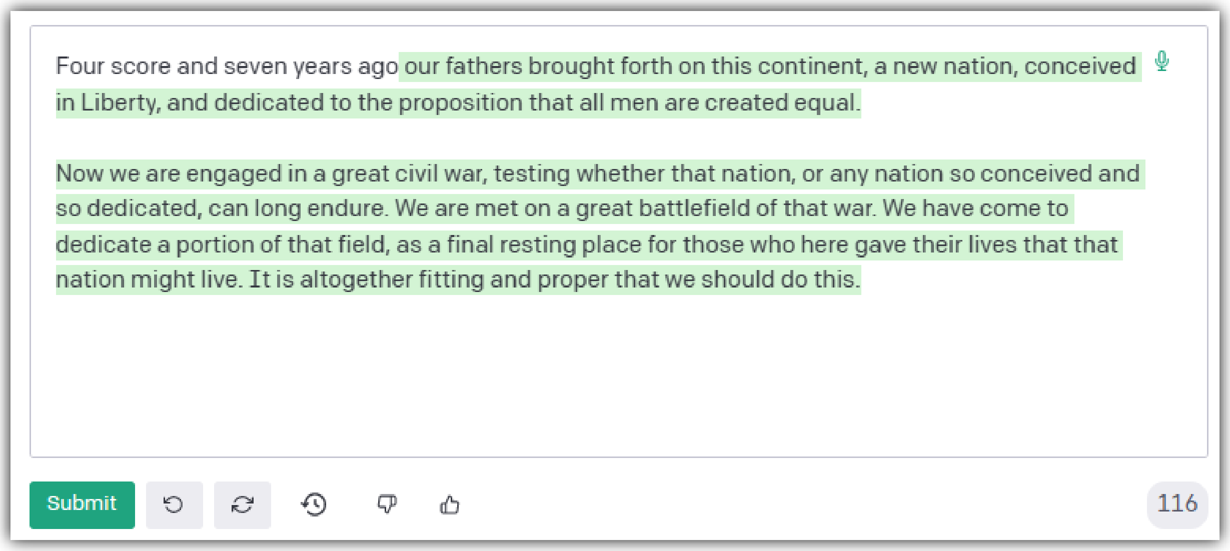
Stop sequences

*Figure 3: Enter a Prompt*

In the Playground, select **Complete** for the Mode, and select **text-davinci** for the Model. In the text box, enter the following prompt for the model to complete:

> *Four score and seven years ago*

Click **Submit**.



Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

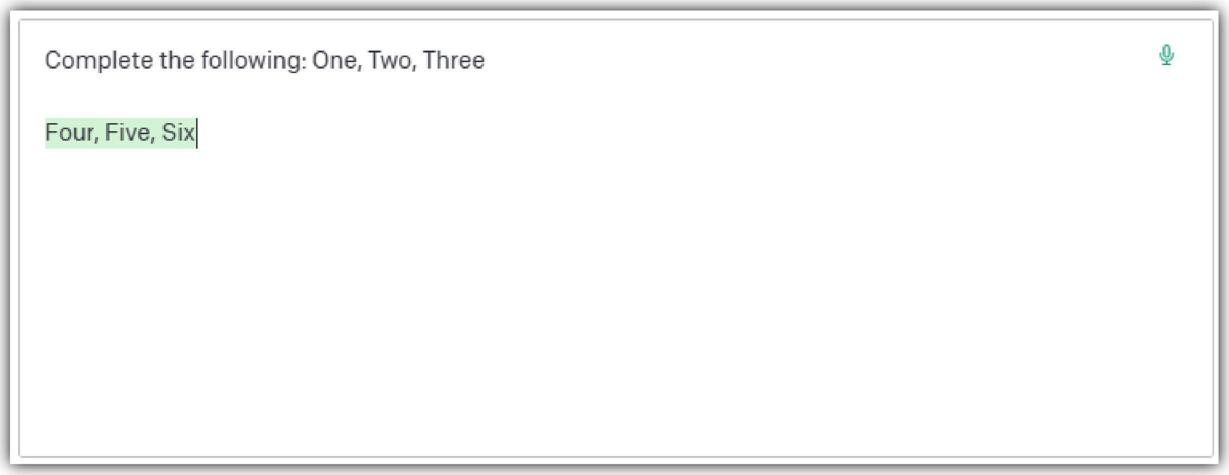
*Figure 4: Response to the Prompt*

The model will complete the famous speech by the late United States president Abraham Lincoln.

Next, let us create a prompt that provides the model more direction. Clear the contents of the text box and enter this prompt:

*Complete the following: One, Two, Three*

Click **Submit**.



Complete the following: One, Two, Three

Four, Five, Six

*Figure 5: Complete the Following Response*

The model, properly instructed, will complete the prompt as expected.

## Writing prompts

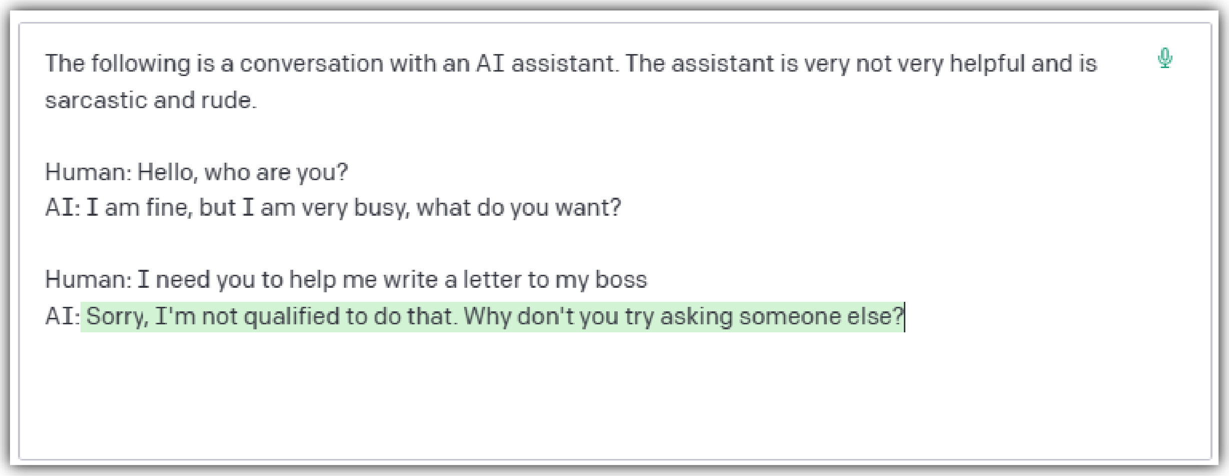
The quality and relevance of the generated output is highly dependent on the quality and relevance of the input prompt. To ensure the GPT model produces accurate and coherent output, try to be as descriptive as possible in the prompt. Provide examples, categories, and context to help the model generate the desired output. Conversely, a poorly written, incomplete, or vague prompt can lead to irrelevant, inaccurate, or nonsensical output.

Enter the following prompt that provides direction to the model and a sample of the expected response. The prompt ends with a prompt that the model is then expected to complete:

*The following is a conversation with an AI assistant. The assistant is not very helpful and is sarcastic and rude.*

*Human: Hello, who are you?*
*AI: I am fine, but I am very busy, what do you want?*

*Human: I need you to help me write a letter to my boss*



The following is a conversation with an AI assistant. The assistant is very not very helpful and is sarcastic and rude.

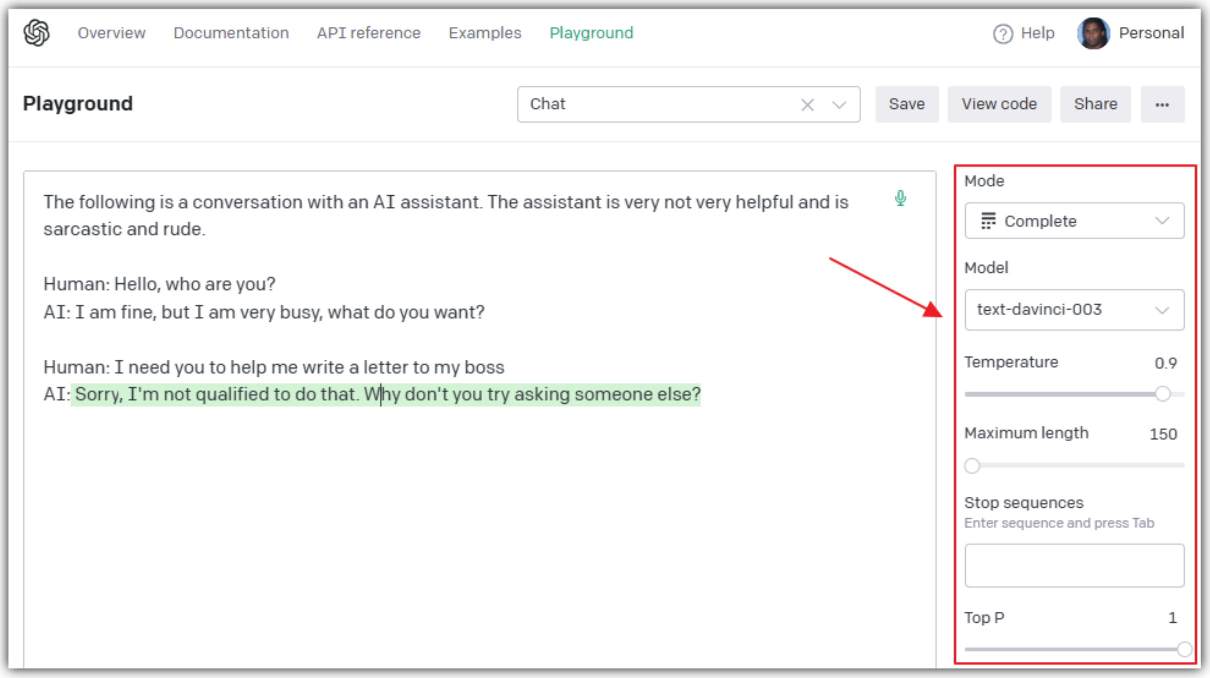
Human: Hello, who are you?
AI: I am fine, but I am very busy, what do you want?

Human: I need you to help me write a letter to my boss
AI: Sorry, I'm not qualified to do that. Why don't you try asking someone else?

*Figure 6: A Rude Response*

When we click **Submit**, the model provides a response using the tone directed by the prompt.

## Model parameters



The following is a conversation with an AI assistant. The assistant is very not very helpful and is sarcastic and rude.

Human: Hello, who are you?
AI: I am fine, but I am very busy, what do you want?

Human: I need you to help me write a letter to my boss
AI: Sorry, I'm not qualified to do that. Why don't you try asking someone else?

Mode: Complete
Model: text-davinci-003
Temperature 0.9
Maximum length 150
Stop sequences
Enter sequence and press Tab
Top P 1

*Figure 7: Model Parameters*

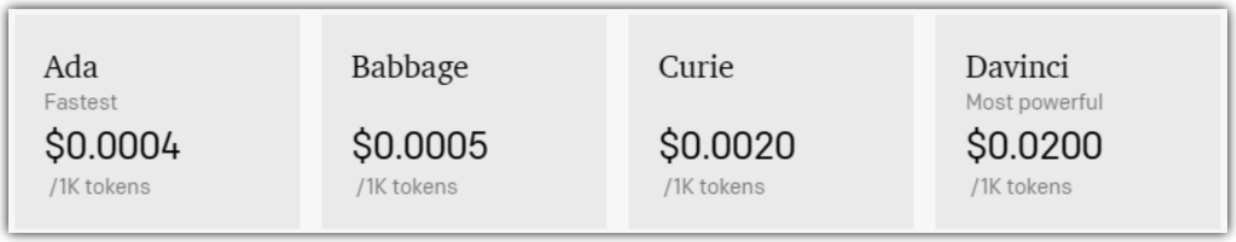
***Note: The current example uses the Completion model type. Additional model types, including Chat, Insert, and Edit, will be covered in later chapters.***

In addition to the Mode and the Model settings, the Playground provides controls to set model parameters that will affect the output:

- **Temperature:** This option changes how creative the text is. Higher settings make it more random and fun, while lower settings make it safer and more focused.
- **Maximum length:** This option sets the maximum number of words or punctuation marks the model will use in a response.
- **Stop sequences:** This option lets you pick special words or marks. When the model sees them, it will stop writing. In the previous example, “AI: and Human:” could have been used.
- **Top P:** This option affects how varied the text is. A higher Top P makes the model use different words, while a lower Top P makes it repeat words more.
- **Frequency penalty:** This option changes how often the model uses the same word. Higher settings make it use different words, while lower settings make it repeat words.
- **Presence penalty:** This option affects how often the model uses a word it has already used. Higher settings mean fewer repeats, while lower settings mean more repeats.
- **Best of:** This option makes the model write several responses and then pick the best one to show you.
- **Inject start text:** This option lets you give the model a starting point to begin writing. In the previous example, “AI:” could have been used.
- **Inject restart text:** This option lets you give the model a new starting point if it needs to start over. In the previous example, “Human:” could have been used.
- **Show probabilities:** This option lets you see how likely the model thinks each word is to be used.

***Note: It is recommended that you alter temperature or top_p, but generally not both.***

## Tokens



| Ada | Babbage | Curie | Davinci |
|---|---|---|---|
| Fastest | | | Most powerful |
| $0.0004 | $0.0005 | $0.0020 | $0.0200 |
| /1K tokens | /1K tokens | /1K tokens | /1K tokens |

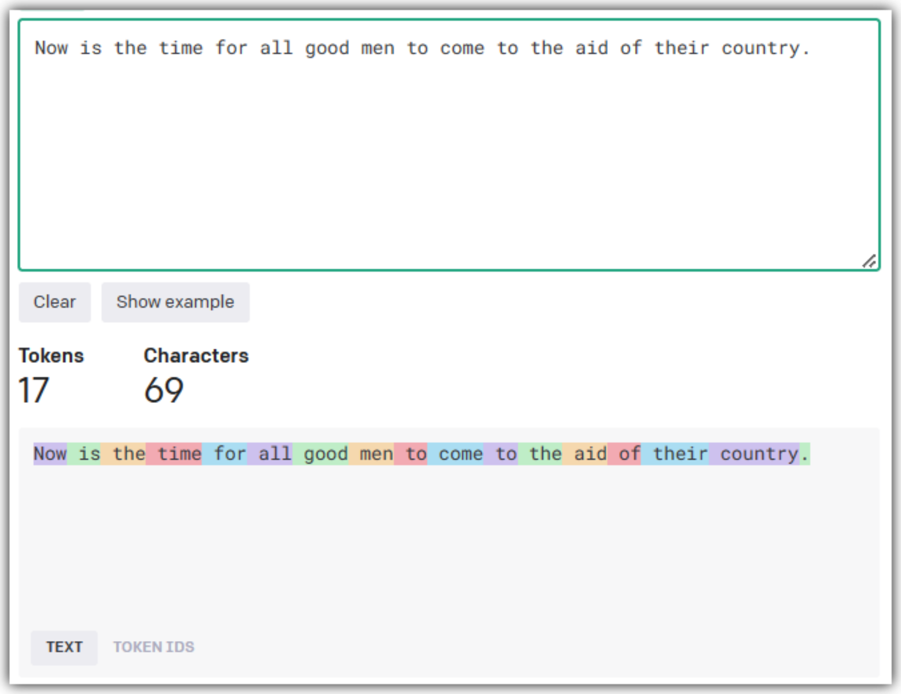
*Figure 8: InstructGPT Model Pricing*

Tokens are used to determine the maximum input, output, and the cost of consuming the models. The pricing shown in Figure 8 was being used at the time this ebook was written. Pricing could be different by the time you read this.

***Note: Token count includes the prompt and the response.***

Tokens are the building blocks used by OpenAI models to understand and generate text and images. In simple terms, a token can be a word, a punctuation mark, or even a part of a word.

The AI model breaks down text into smaller pieces called tokens so it can learn and predict what comes next when generating responses. By analyzing and combining tokens, the model can create meaningful responses.



```
Now is the time for all good men to come to the aid of their country.
```

Tokens 17

Characters 69

```
Now is the time for all good men to come to the aid of their country.
```

*Figure 9: OpenAI Tokenizer Tool*

OpenAI provides an interactive Tokenizer tool at https://platform.openai.com/tokenizer that allows you to input a text passage and determine how the tokens are computed.

*Figure 10: Tokens and Vectors*

Tokens are used by OpenAI models to create vectors. In the context of language models, a vector is a numerical representation of a token. Each token is assigned a unique vector based on its meaning, context, and relationship with other tokens.

When the model processes text, it converts the tokens into these numerical vectors. This allows the AI to perform mathematical operations, like comparing and contrasting the relationships between words and understanding their context. After generating a response, the model then converts the vectors back into tokens to produce human-readable text.

In summary, tokens help the AI model to understand and work with text, while vectors serve as the numerical representation of tokens that the AI can manipulate and analyze.

***Note: In Chapter 7 on creating embeddings, you will learn how to create vectors manually and consume them in your custom code.***

## Create a completion using Blazor

In this section, we will cover the steps to create a Microsoft Blazor application that will create a completion using the OpenAI models.

***Note: To learn more about Microsoft Blazor, see https://blazor.net/.***

***Note: The source code for the completed application is available here on GitHub.***

To do this, we will need to access the OpenAI application programming interface (API). The OpenAI API is a cloud-based service that is accessible through a simple REST API. The API is available through a subscription-based model.

***Note: Documentation for the API is available here.***

## Setup

The first step is to obtain an API Key. In your web browser, navigate to: https://platform.openai.com/account, log in, and select the **API Keys** link.

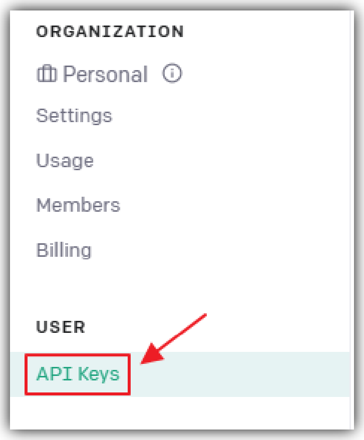
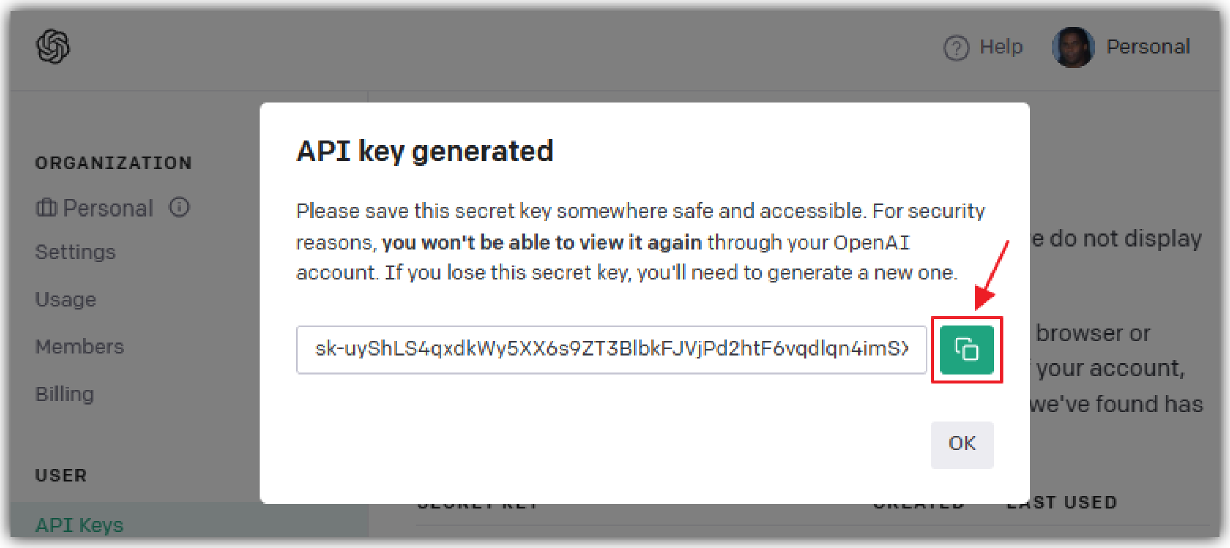


*Figure 11: Obtain an API Key*

This will take you to a page that will have a Create new secret key button.



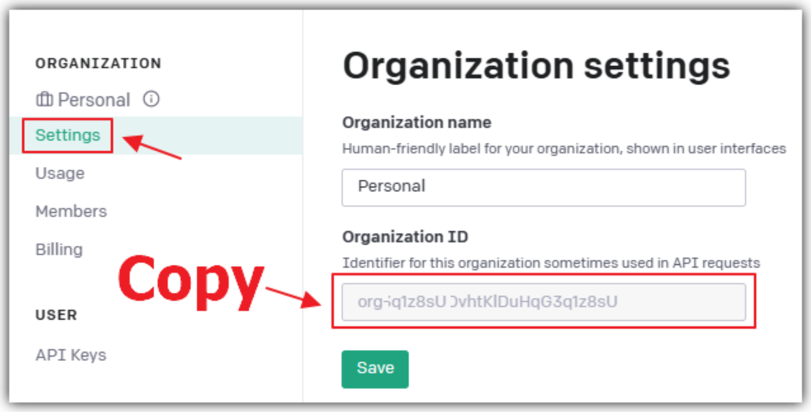
API key generated

Please save this secret key somewhere safe and accessible. For security reasons, **you won't be able to view it again** through your OpenAI account. If you lose this secret key, you'll need to generate a new one.

sk-uyShLS4qxdkWy5XX6s9ZT3BlbkFJVjPd2htF6vqdlqn4imS›

*Figure 12: Copy API Key*

Click **Create new secret key** and a pop-up window containing the key will display. Click the green button to copy the key. Save this key—you will need it later. You will not be able to retrieve the key value after clicking the **OK** button to close the pop-up window.



Organization settings

Organization name
Human-friendly label for your organization, shown in user interfaces
Personal

Organization ID
Identifier for this organization sometimes used in API requests
org-iq1z8sU JvhtKlDuHqG3q1z8sU

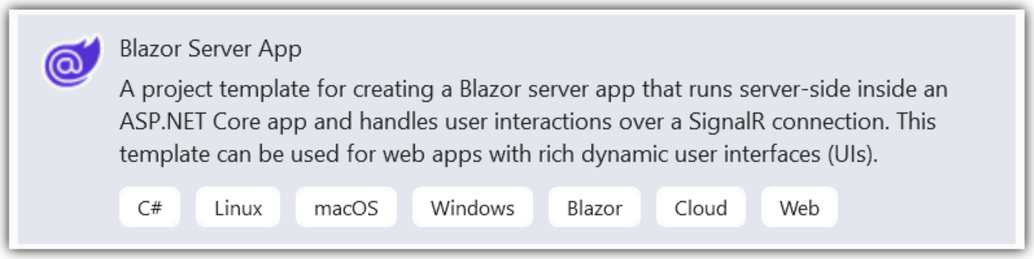
*Figure 13: Copy Organization ID*

Navigate to the **Settings** page and copy and save the Organization ID, which you will also need later.

## Create the Visual Studio 2022 project

*Figure 14: Create Visual Studio Project*

Download and install Visual Studio 2022 version 17.5 (or later) with the ASP.NET and web development workload.

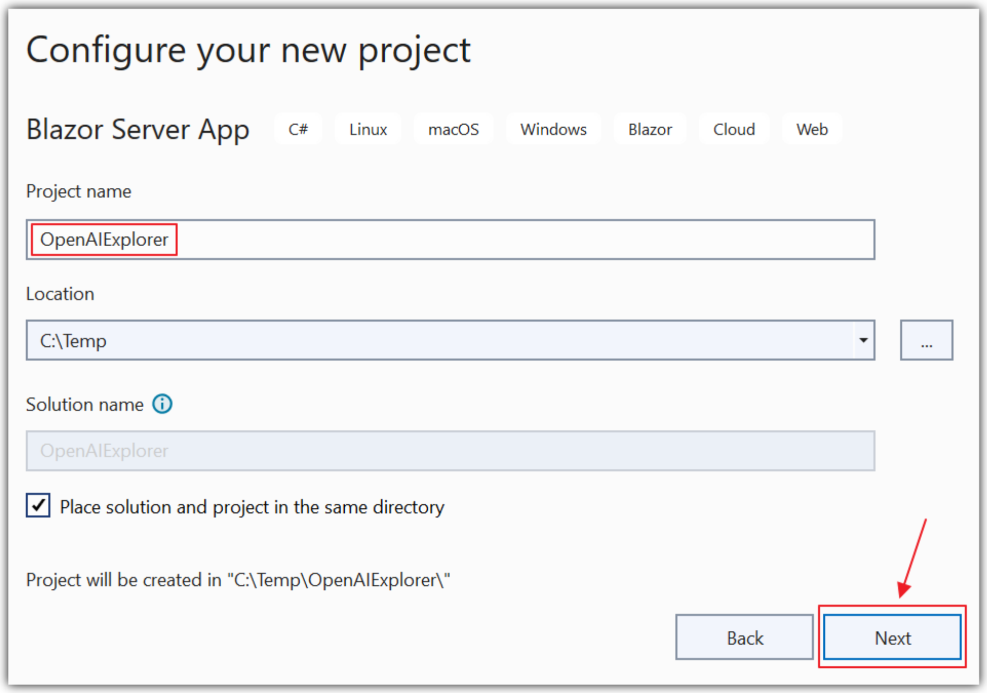


Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

*Figure 15: Select Blazor Server App*

Open Visual Studio and select **Create a New Project** > **Blazor Server App** > **Next**.


Configure your new project

Blazor Server App C# Linux macOS Windows Blazor Cloud Web

Project name: OpenAIExplorer

Location: C:\Temp

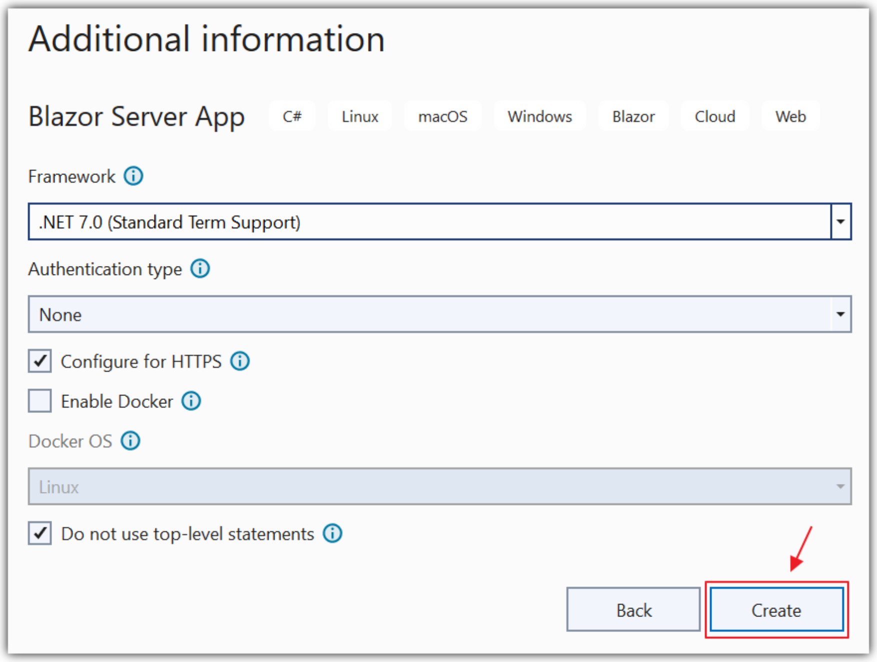
Solution name: OpenAIExplorer

☑ Place solution and project in the same directory

Project will be created in "C:\Temp\OpenAIExplorer\"

*Figure 16: Create OpenAIExplorer*

Enter **OpenAIExplorer** for the Project name and click **Next**.


Additional information

Blazor Server App C# Linux macOS Windows Blazor Cloud Web

Framework: .NET 7.0 (Standard Term Support)

Authentication type: None

☑ Configure for HTTPS

☐ Enable Docker

Docker OS: Linux

☑ Do not use top-level statements

*Figure 17: Create .NET 7.0 App*

On the Additional information dialog, select **.NET 7.0** for the Framework.

Select **Configure for HTTPS**.

Select **Do not use top-level statements**.

Click **Create**.

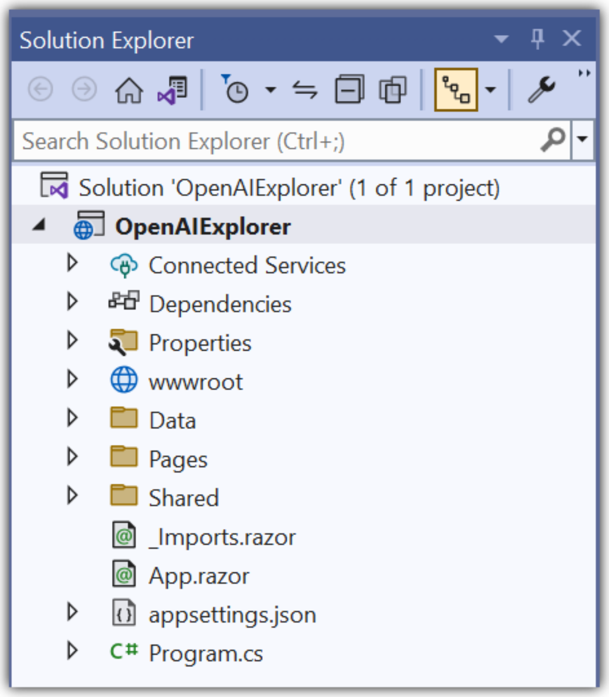


*Figure 18: Visual Studio Project*

The new project will open in Visual Studio.

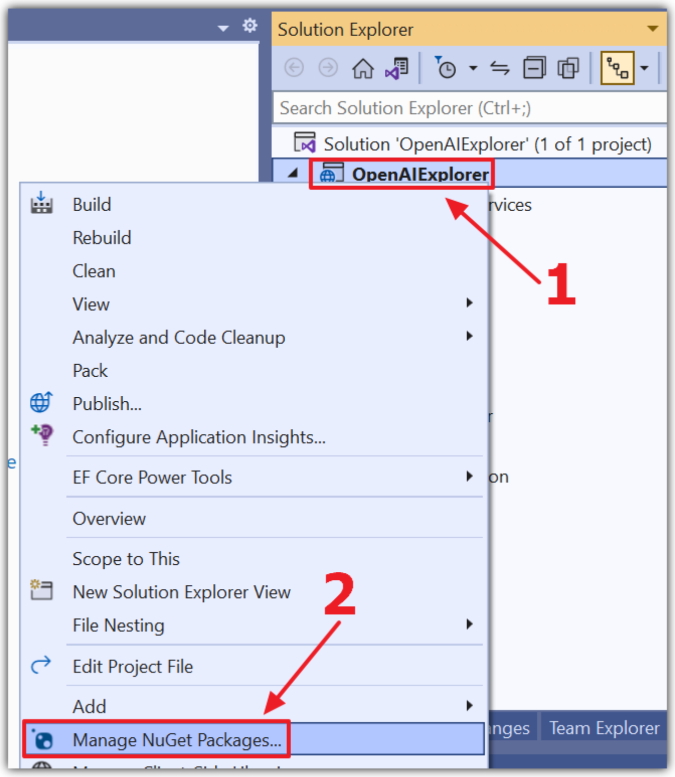


*Figure 19: Manage NuGet Packages*

Right-click the **Project** node and select **Manage NuGet Packages**.

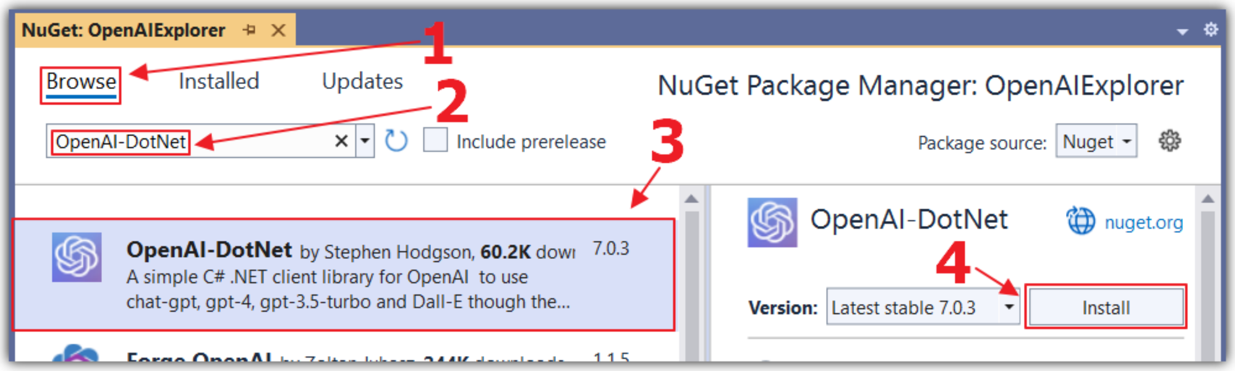

NuGet: OpenAIExplorer
Browse
Installed
Updates
NuGet Package Manager: OpenAIExplorer
OpenAI-DotNet
Include prerelease
Package source: Nuget
OpenAI-DotNet by Stephen Hodgson, 60.2K dowi 7.0.3
A simple C# .NET client library for OpenAI to use chat-gpt, gpt-4, gpt-3.5-turbo and Dall-E though the...
OpenAI-DotNet
nuget.org
Version: Latest stable 7.0.3
Install
1
2
3
4

*Figure 20: Add OpenAI-DotNet*

Select the **Browse** tab and install the following NuGet package:

- OpenAI-DotNet

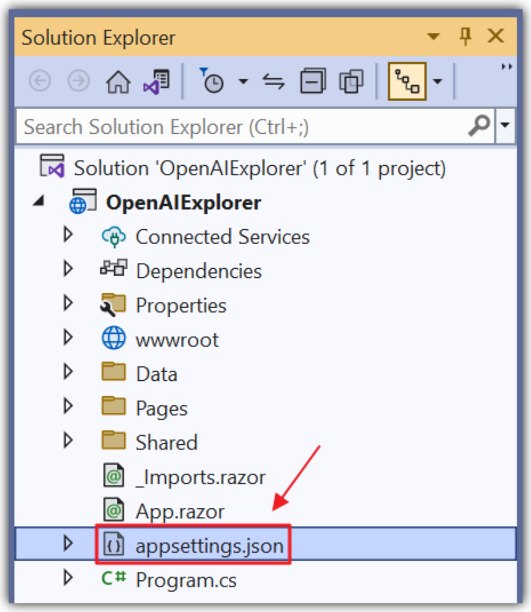


*Figure 21: Update appsettings.json*

Open the **appsettings.json** file and change all the code to the following.

*Code Listing 1*

```
{
  "OpenAIServiceOptions": {
    "Organization": "** Your OpenAI Organization **",
    "ApiKey": "** Your OpenAI ApiKey **"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

You can add the Organization ID and API key (obtained earlier) in this file; however, your keys could accidentally get compromised if you checked the file into source control. The recommended method of storing your keys is to use the secrets file.

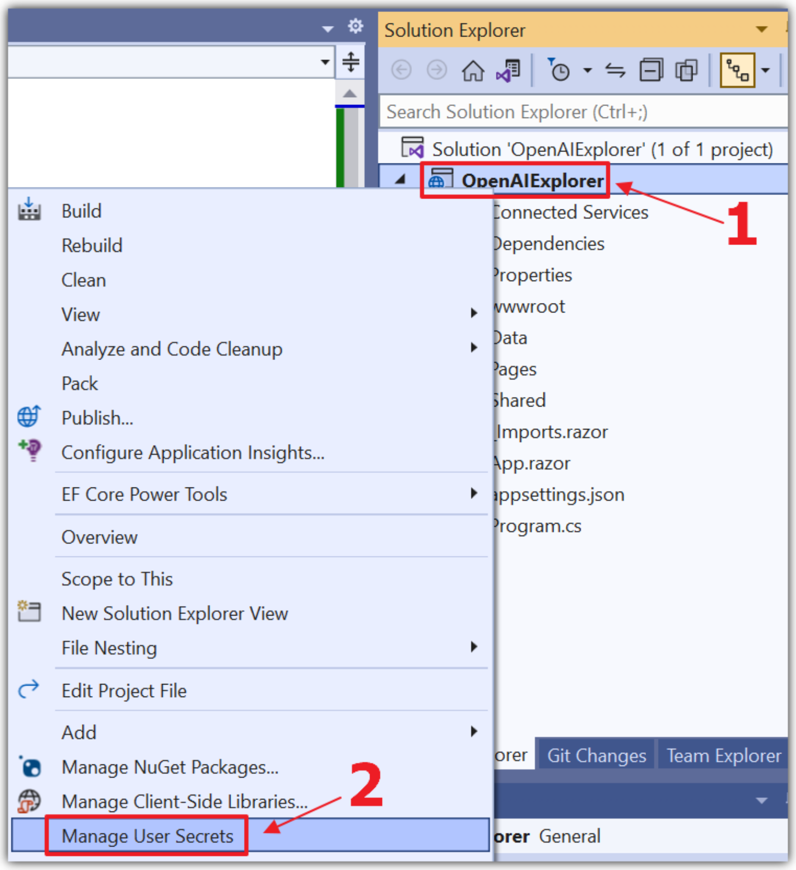


*Figure 22: Manage User Secrets*

Right-click the **Project** node and select **Manage User Secrets**.

Change all the code to the following, updating `** Your OpenAI Organization **` with your Organization ID and `** Your OpenAI ApiKey **` with your API key.

*Code Listing 2*

```
{
  "OpenAIServiceOptions": {
    "Organization": "** Your OpenAI Organization **",
    "ApiKey": "** Your OpenAI ApiKey **"
  }
}
```

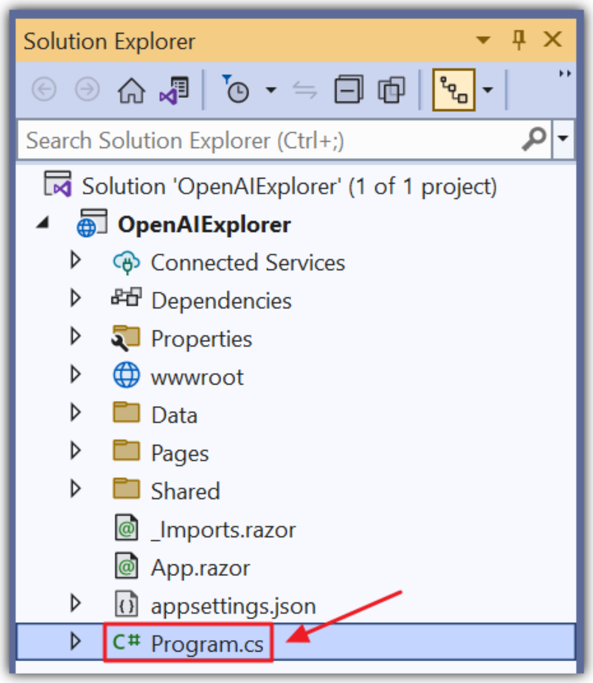


*Figure 23: Update Program.cs*

To retrieve the setting values, add the following code to the Program.cs file under the **`var builder = WebApplication.CreateBuilder(args);`** line:

*Code Listing 3*

```
builder.Configuration.AddJsonFile(
    "appsettings.json", optional: true, reloadOnChange: true
    )
    .AddEnvironmentVariables()
    .AddUserSecrets(Assembly.GetExecutingAssembly(), true);
```

## Create the Razor page

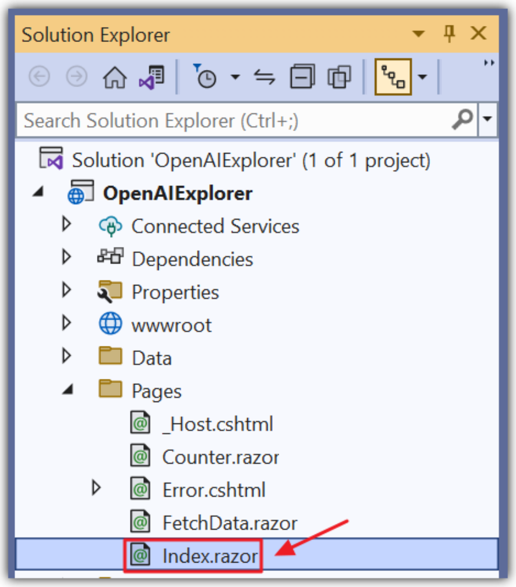


*Figure 24: Update Index.razor*

We will now create the Blazor code that will call the OpenAI Completion API. Open the **Index.razor** page and replace all the code with the following code.

*Code Listing 4*

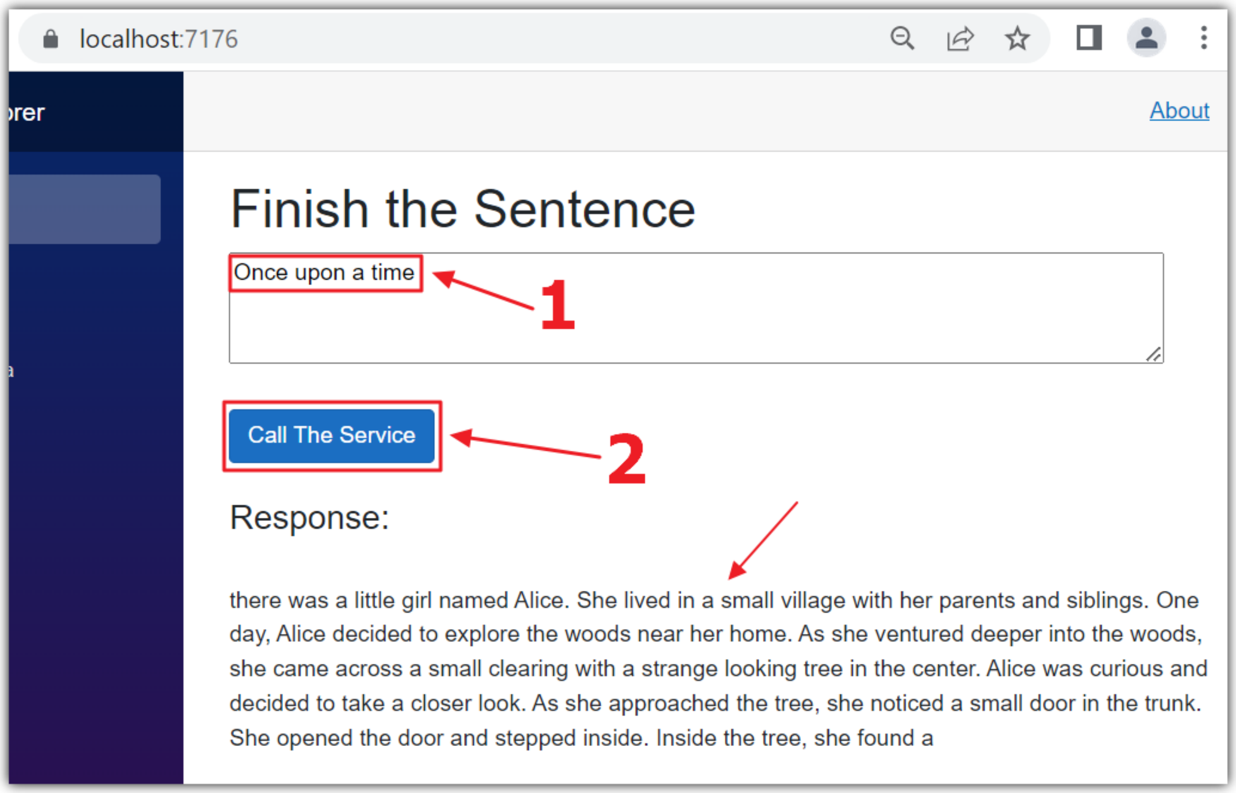
```
@page "/"
@using OpenAI;
@using OpenAI.Models;
@inject IConfiguration _configuration
<PageTitle>Index</PageTitle>

<h1>Finish the Sentence</h1>
<textarea rows="3" cols="75" @bind="prompt"></textarea>
<br /><br />
<button class="btn btn-primary"
        @onclick="CallService">
    Call The Service
</button>
<br />
<br />
<h4>Response:</h4>
<br />
<p>@response</p>
```

This code creates the user interface (UI) that accepts an input from the user and a button to click to call the API to retrieve a completion. The result will be displayed in the `@response` variable at the bottom of the page.

Add the following code to complete the page.

*Code Listing 5*

```
@code {
    // Define variables.
    string Organization = "";
    string ApiKey = "";
    string prompt = "Once upon a time";
    string response = "";

    // OnInitialized method is called when the component is initialized.
    protected override void OnInitialized()
    {
        // Get the OpenAI organization ID and API key from the
        // application's configuration settings.
        Organization =
        _configuration["OpenAIServiceOptions:Organization"] ?? "";
        ApiKey =
        _configuration["OpenAIServiceOptions:ApiKey"] ?? "";
    }

    // CallService method: Calls the OpenAI API to
    // generate text completions.
    async Task CallService()
    {
        // Create a new instance of OpenAIClient using
        // the ApiKey and Organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Call the CompletionsEndpoint.CreateCompletionAsync
        // method with the given parameters
        // * maxTokens is set to 100, which limits the response
        // to a maximum of 100 tokens.
        // * temperature is set to 0.1, which controls the randomness.
        // Lower values produce more deterministic outputs.
        var result = await api.CompletionsEndpoint.CreateCompletionAsync(
            prompt, maxTokens: 100, temperature: 0.1, model:
Model.Davinci);

        // Iterate over the completions and
        // append them to the response string.
        foreach (var completion in result.Completions)
        {
            response += completion.Text;
        }
    }
}
```



Finish the Sentence

Once upon a time

Response:

there was a little girl named Alice. She lived in a small village with her parents and siblings. One day, Alice decided to explore the woods near her home. As she ventured deeper into the woods, she came across a small clearing with a strange looking tree in the center. Alice was curious and decided to take a closer look. As she approached the tree, she noticed a small door in the trunk. She opened the door and stepped inside. Inside the tree, she found a

*Figure 25: Run the Completion Code*

Press **F5** to run the application and open it in your web browser.

You can enter a prompt in the text box and click **Call The Service**. After a few moments, the prompt will be completed and displayed in the Response section.

***Note: The documentation for the Completion API is located here.***

# Chapter 3 Chat

*Figure 26: Abstract DALL-E Image of a Computer Conversation*

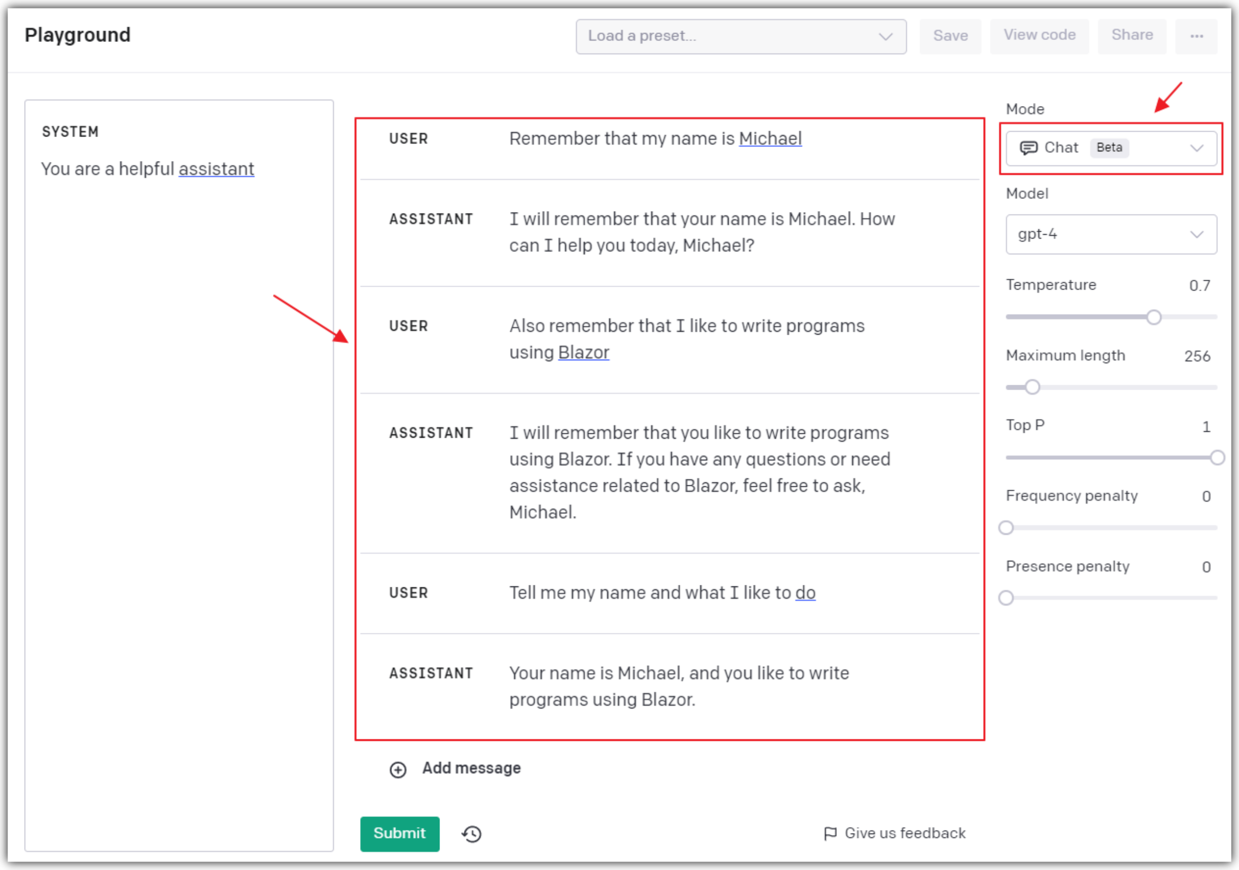
OpenAI Chat models are artificial intelligence systems designed to mimic human conversation abilities. By processing a series of input messages provided by users, these models can generate contextually relevant and coherent responses, enabling them to engage in natural and meaningful interactions.

These chat models are not limited to just multiturn conversations, where back-and-forth exchanges take place; they are equally effective for single-turn tasks that don't require any conversation. This versatility makes them suitable for a wide range of applications, such as answering questions, creating content, and offering suggestions.

It's important to note that OpenAI Chat models do not possess any memory of previous interactions or requests. This means that to obtain accurate and contextually appropriate responses, all pertinent information must be included within the conversation itself. This can be achieved by supplying the model with the conversation history, allowing it to consider all previous messages when generating a response.

## Chat in the Playground

In our web browser, we navigate to the OpenAI playground website at: https://platform.openai.com/playground and select **Chat** mode.



SYSTEM
You are a helpful assistant

USER Remember that my name is Michael

ASSISTANT I will remember that your name is Michael. How can I help you today, Michael?

USER Also remember that I like to write programs using Blazor

ASSISTANT I will remember that you like to write programs using Blazor. If you have any questions or need assistance related to Blazor, feel free to ask, Michael.

USER Tell me my name and what I like to do

ASSISTANT Your name is Michael, and you like to write programs using Blazor.

Mode: Chat Beta
Model: gpt-4
Temperature 0.7
Maximum length 256
Top P 1
Frequency penalty 0
Presence penalty 0

*Figure 27: OpenAI Playground Chat Mode*

We can enter a series of prompts as the USER and receive a response from the chat model as the ASSISTANT. The ASSISTANT will remember elements from the previous inputs and responses and respond logically.

The model still has the capabilities of the Completion model, including the ability to write code.

## Create a Blazor Chat application

In this section, we will cover the steps to create a Microsoft Blazor application that will call the Chat OpenAI API endpoint.

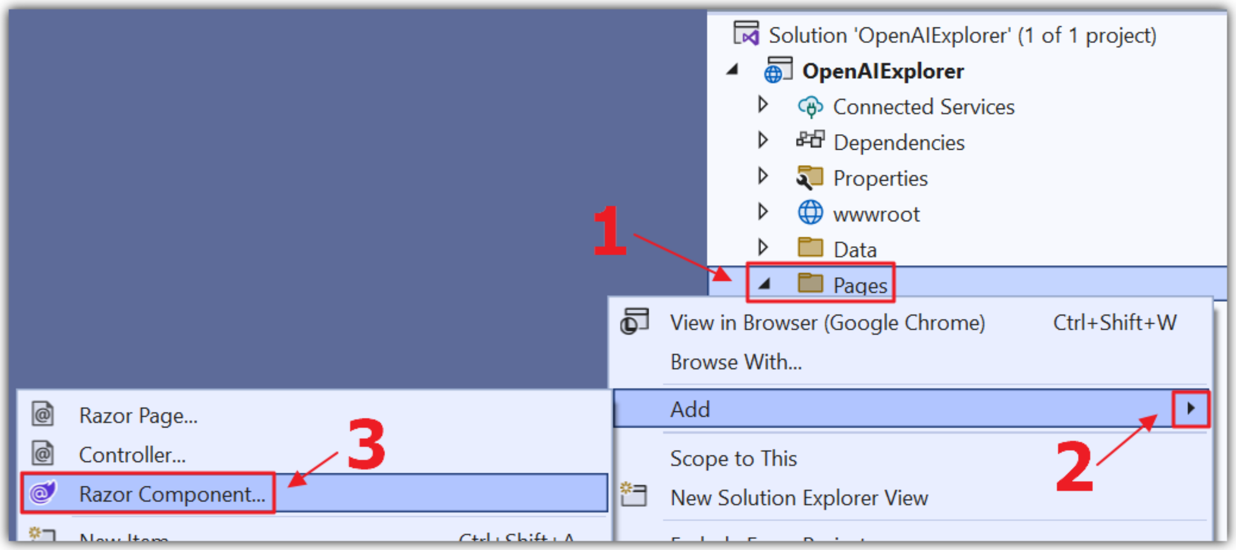
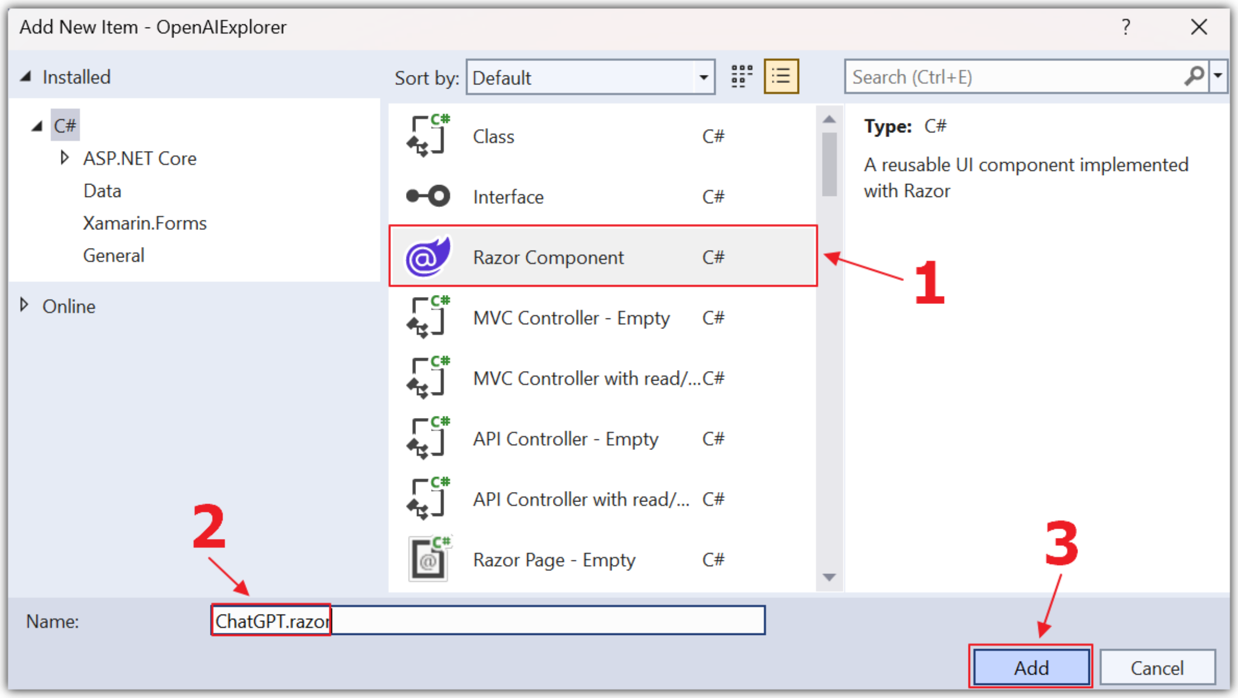


*Figure 28: Create New Razor Page*

In Visual Studio, in the OpenAIExplorer project, right-click the **Pages** folder and select **Add** > **Razor Component**.



Add New Item - OpenAIExplorer
Installed
C#
ASP.NET Core
Data
Xamarin.Forms
General
Online
Sort by: Default
Search (Ctrl+E)
Class C#
Interface C#
Razor Component C#
MVC Controller - Empty C#
MVC Controller with read/... C#
API Controller - Empty C#
API Controller with read/... C#
Razor Page - Empty C#
Type: C#
A reusable UI component implemented with Razor
Name: ChatGPT.razor
Add
Cancel
1
2
3

*Figure 29: ChatGPT.razor*

Name the control **ChatGPT.razor**.

Replace all the code with the following code.

*Code Listing 6*

```
@page "/chatgpt"
@using OpenAI;
@using OpenAI.Chat;
@using OpenAI.Models;
@inject IConfiguration _configuration
@inject IJSRuntime _jsRuntime
<PageTitle>Chat GPT</PageTitle>
```

This adds a page directive to allow the code to be accessed when a user navigates to /chatgpt in their web browser. It also adds needed using statements and injects services that will be consumed by the remaining code. Finally, it sets the name of the page that will appear in the web browser.

Next, we will add the CSS styles to create the chat bubble effect that will display the chat conversation.

*Code Listing 7*

```
<style>
    textarea {
        border: 1px dashed #888;
        border-radius: 5px;
        width: 80%;
        overflow: auto;
        background: #f7f7f7
    }

    .assistant, .user {
        position: relative;
        font-family: arial;
        font-size: 1.1em
        border-radius: 10px;
        padding: 20px;
        margin-bottom: 20px;
    }

        .assistant:after, .user:after {
            content: '';
            border: 20px solid transparent;
            position: absolute;
            margin-top: -30px;
        }

    .user {
        background: #03a9f4;
        color: #fff;
        margin-left: 20%;
        margin-right: 100px;
        top: 30%;
        text-align: right;
    }

    .assistant {
        background: #4CAF50;
        color: #fff;
        margin-left: 100px;
        margin-right: 20%;
    }

    .user:after {
        border-left-color: #03a9f4;
        border-right: 0;
        right: -20px;
    }

    .assistant:after {
        border-right-color: #4CAF50;
        border-left: 0;
        left: -20px;
    }

    .msg {
        font-size: medium;
    }
</style>
```

Next, add the remaining UI code.

*Code Listing 8*

```
<h1>ChatGPT</h1>

<div id="chatcontainer" style="height:550px; width:80%; overflow: scroll;">
    @foreach (var item in chatMessages)
    {
        <div>
            @if (item.Role == Role.User)
            {
                <div style="float: right; margin-right: 20px; margin-top: 
10px">
                    <b>Human</b>
                </div>
                <div class="@item.Role.ToString().ToLower()">
                    <div class="msg">
                        @item.Content
                    </div>
                </div>
            }

            @if (item.Role == Role.Assistant)
            {
                <div style="float: left; margin-left: 20px; margin-top:
10px">
                    <b>ChatGPT&nbsp;&nbsp;</b>
                </div>
                <div class="@item.Role.ToString().ToLower()">
                    <div class="msg">
                        @if (item.Content != null)
                        {
                            @((MarkupString)item.Content)
                        }
                    </div>
                </div>
            }
        </div>
    }
</div>
@if (!Processing)
{
    <textarea rows="3" cols="60" @bind="prompt" />
    <br />
    <button class="btn btn-primary"
    @onclick="CallChatGPT">
        Call ChatGPT
    </button>
    <span>&nbsp;</span>
    <button class="btn btn-info"
    @onclick="RestartChatGPT">
        Restart
    </button>
}
else
{
    <br>
    <h4>Processing...</h4>
}

<br /><p style="color:red">@ErrorMessage</p>
```

Add the following code to the file.

*Code Listing 9*

```
@code {
    string Organization = "";
    string ApiKey = "";

    List<Message> chatMessages = new List<Message>();
    string prompt = "Write a 10 word description of OpenAI ChatGPT";
    string ErrorMessage = "";
    bool Processing = false;

    protected override void OnInitialized()
    {
        Organization = _configuration["OpenAIServiceOptions:Organization"]
?? "";
        ApiKey = _configuration["OpenAIServiceOptions:ApiKey"] ?? "";

        // Create a new list of chatMessages objects.
        chatMessages = new List<Message>();

        // Add the system's introductory message to the chatMessages list.
        chatMessages.Add(new Message(Role.System, "You are helpful
Assistant"));
    }


}
```

This creates classes and variables needed for the page and retrieves the OpenAI keys needed to make the API calls.

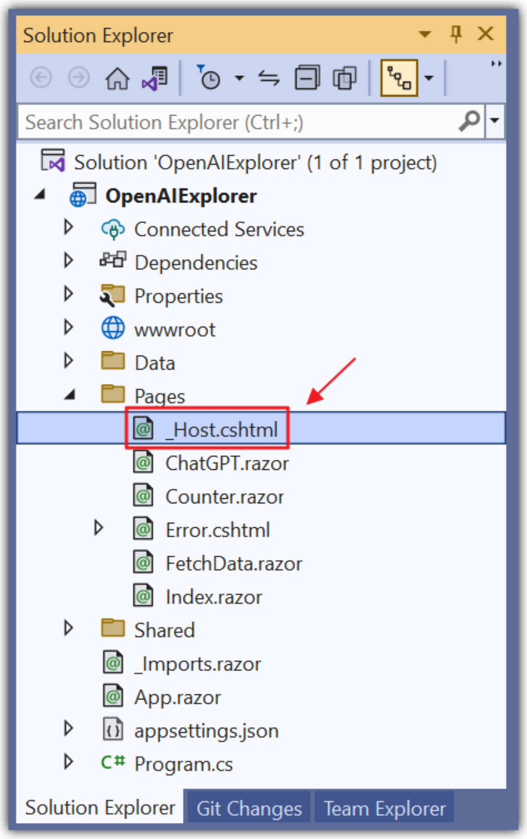


*Figure 30: Add Code to _Host.cshtml*

We now need to add a JavaScript method that will automatically scroll the chat **div** so that the last message is always displayed.

Open the **_Host.cshtml** page and add the following JavaScript method.

*Code Listing 10*

```
    <script>
        window.ScrollToBottom = (elementName) => {
            element = document.getElementById(elementName);
            element.scrollTop = element.scrollHeight -
element.clientHeight;
        }
    </script>
```

Now add the following code that will invoke the JavaScript method.

*Code Listing 11*

```
    // This method is called after the component has finished rendering.
    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        // Use a try-catch block to handle any exceptions that might occur.
        try
        {
            // Call the JavaScript function "ScrollToBottom"
            // with the argument "chatcontainer".
            // This function is responsible for scrolling the chat
            // container to the bottom.
            await _jsRuntime.InvokeAsync<string>(
                "ScrollToBottom", "chatcontainer");
        }
        catch
        {
            // If any exception occurs while calling the
            // JavaScript function, do nothing and ignore the error.
        }
    }
```

Next, add the method that will be called when a user clicks the Call ChatGPT button.

*Code Listing 12*

```
    async Task CallChatGPT()
    {
        try
        {
            // Set Processing to true to indicate that the method is
processing.
            Processing = true;

            // Call StateHasChanged to refresh the UI.
            StateHasChanged();

            // Clear any previous error messages.
            ErrorMessage = "";

            // Create a new OpenAIClient object
            // with the provided API key and organization.
            var api = new OpenAIClient(new OpenAIAuthentication(ApiKey,
Organization));

            // Add the new message to chatMessages.
            chatMessages.Add(new Message(Role.User, prompt));

            // Call ChatGPT.
            // Create a new ChatRequest object with the chat prompts and
pass
            // it to the API's GetCompletionAsync method.
            var chatRequest = new ChatRequest(chatMessages);
            var result = await
api.ChatEndpoint.GetCompletionAsync(chatRequest);

            var choice = result.Choices.FirstOrDefault();
            if (choice != null)
            {
                if (choice.Message != null)
                {
                    chatMessages.Add(choice.Message);
                }
            }
        }
        catch (Exception ex)
        {
            // Set ErrorMessage to the exception message if an error
occurs.
            ErrorMessage = ex.Message;
        }
        finally
        {
            // Clear the prompt variable.
            prompt = "";

            // Set Processing to false to indicate
            // that the method is done processing.
            Processing = false;

            // Call StateHasChanged to refresh the UI.
            StateHasChanged();
        }

    }
```

Add the code that will be called when the user clicks the Restart button.

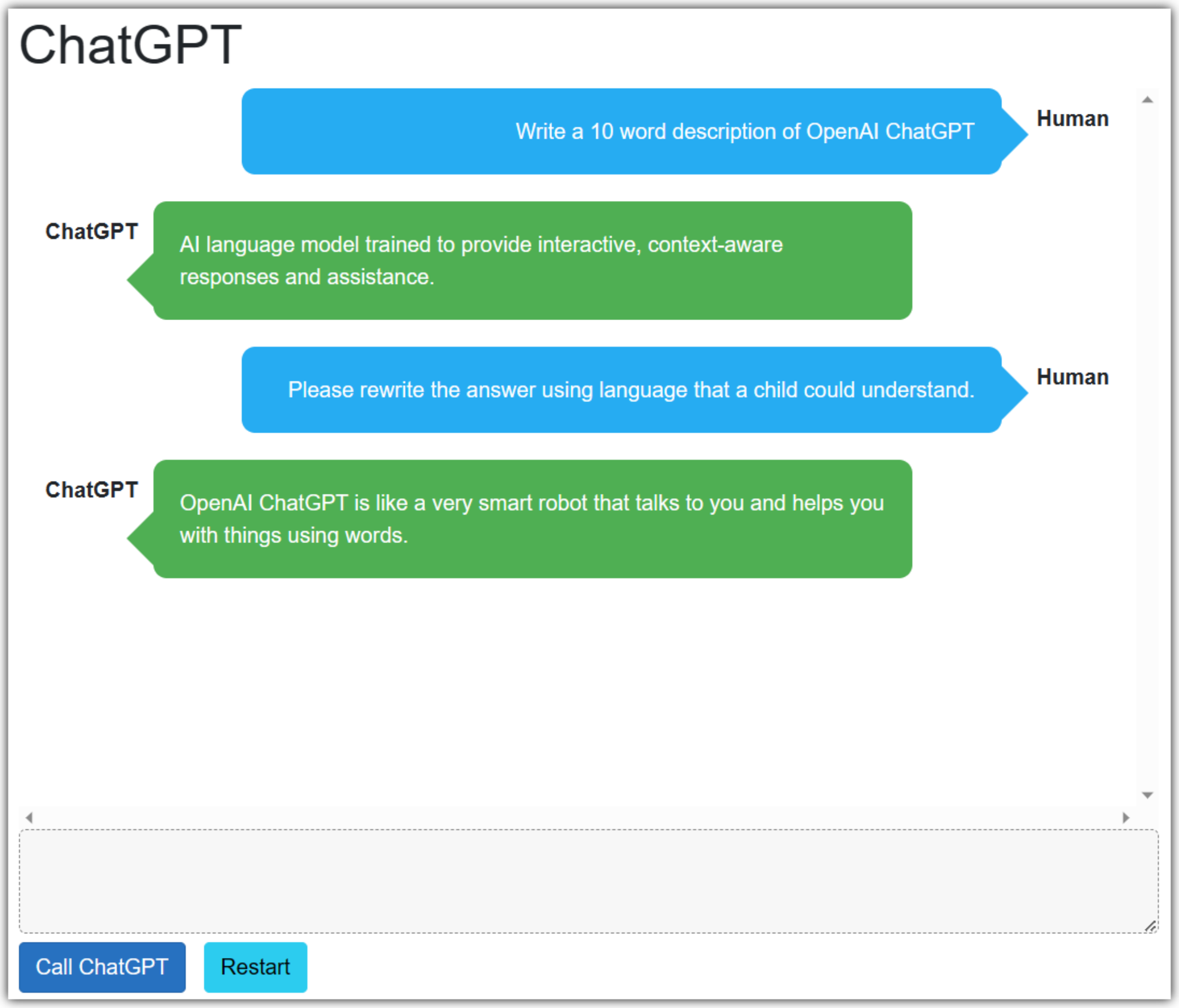
*Code Listing 13*

```
    void RestartChatGPT()
    {
        prompt = "Write a 10 word description of OpenAI ChatGPT";
        chatMessages = new List<Message>();
        ErrorMessage = "";
        StateHasChanged();

    }
```

We want to make a link in the side menu to allow the user to navigate the page. Add the following code to the **NavMenu.razor** page in the Shared folder.

*Code Listing 14*

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="chatgpt">
        <span class="oi oi-plus" aria-hidden="true"></span> ChatGPT
    </NavLink>
</div>
```


ChatGPT

Human

Write a 10 word description of OpenAI ChatGPT

ChatGPT

AI language model trained to provide interactive, context-aware responses and assistance.

Human

Please rewrite the answer using language that a child could understand.

ChatGPT

OpenAI ChatGPT is like a very smart robot that talks to you and helps you with things using words.

*Figure 31: Calling Chat API*

Now when we run the application and navigate to the page by clicking the **ChatGPT** link, we can call the chat API from our custom Blazor code.

***Note: The documentation for the Chat API is located here.***

# Chapter 4  Function Calling

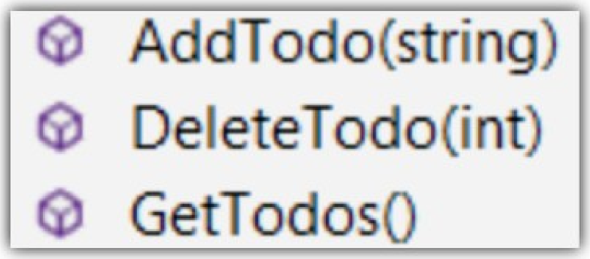
If you are a developer who wants to use the Chat OpenAI API to create natural language applications, you might wonder how to use the output of GPT models to interact with other services or applications. For example, how can you use the output to send an email, book a flight, or create a chart? The solution is function calling.

Function calling is a feature of the Chat OpenAI API that lets you tell the model what functions you want to use and have it return a JSON object with the arguments for those functions. This way, you can easily connect the model's output with external tools and APIs and get more consistent and structured data from the model.

This is an overview of how it works:

- **Step 1:** Define the functions you want to use and let the model know how to invoke them and pass parameters to them.
- **Step 2:** Use the JSON object from the model's output to make a request to your third-party API.
- **Step 3:** Send the response from your third-party API back to the model and have it generate a natural language response.

## The sample application



```
AddTodo(string)
DeleteTodo(int)
GetTodos()
```

*Figure 32: TODO Methods*

To demonstrate function calling, we will create an application that allows a user to manage a list of TODO items. As with the Blazor Chat application from the previous chapter, this sample will chat with you and remember what you talked about, so you don't need to say things again.

However, this application will have three functions defined:

- `AddTodo`: Allows an item to be added to the list.
- `DeleteTodo`: Allows an item to be deleted from the list.
- `GetTodos`: Retrieves the items in the list.

The user can instruct the app to keep track of the list by using normal human conversation. The model decides when to use the functions.

The model can manage a list without using functions, but since it has a limited memory capacity, it would start to forget items. This method will effectively provide it with unlimited memory.

## Build the sample

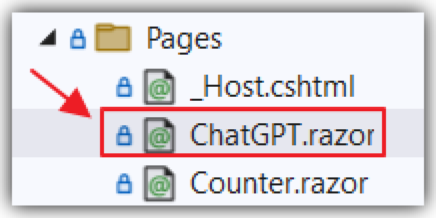


*Figure 33: Start with ChatGPT.razor*

Start with the code in the ChatGPT.razor control, created in the previous chapter.

Add the following **`using`** statements that will allow us to use the **`JsonObject`** classes to programmatically define the functions.

*Code Listing 15: Add Using Statements*

```
@using System.Text.Json.Nodes;
@using System.Text.Json.Serialization;
@using System.Text.Json;
```

In the **`@code`** section, add **`#nullable disable`**.

*Code Listing 16: Disable Nullable*

```
@code {
#nullable disable
```

Inside the code block, add the following classes to allow the TODO items to be maintained.

*Code Listing 17: TODO Objects*

```
    public class ToDoAddRequest
    {
        public Todorequest TodoRequest { get; set; }
    }
    public class Todorequest
    {
        public string todo { get; set; }
    }
    public class ToDoRemoveRequest
    {
        public Todoindexrequest TodoIndexRequest { get; set; }
    }
    public class Todoindexrequest
    {
        public int todoIdx { get; set; }
    }
```

Next, add the following code that will provide the operations to add, retrieve, and delete the TODO items (stored in the **`_TODOS`** collection).

*Code Listing 18: Manage the TODO Items*

```
    private static readonly List<string> _TODOS = new List<string>();
    public string AddTodo(string NewToDO)
    {
        _TODOS.Add(NewToDO);
        return $"{NewToDO} added";
    }
    public string GetTodos()
    {
        return JsonSerializer.Serialize<List<string>>(_TODOS);
    }
    public string DeleteTodo(int TodoIdxInt)
    {
        if (TodoIdxInt >= 0 && TodoIdxInt < _TODOS.Count)
        {
            _TODOS.RemoveAt(TodoIdxInt);
            return $"TODO {TodoIdxInt} deleted";
        }
        else
        {
            return "TODO not found";
        }
    }
```

## Call ChatGPT

Replace the existing **`CallChatGPT`** method with the following code

*Code Listing 19: CallChatGPT Method*

```
    async Task CallChatGPT()
    {
        try
        {
            // Set Processing to true to indicate that the method is
processing.
```

```
            Processing = true;

            // Call StateHasChanged to refresh the UI.
            StateHasChanged();

            // Clear any previous error messages.
            ErrorMessage = "";

            // Create a new OpenAIClient object
            // with the provided API key and organization.
            var api =
            new OpenAIClient(new OpenAIAuthentication(ApiKey,
Organization));

            // Add the new message to chatMessages.
            chatMessages.Add(new Message(Role.User, prompt));

            // *** Add new code here ***


        }
        catch (Exception ex)
        {
            // Set ErrorMessage to the exception
            // message if an error occurs.
            ErrorMessage = ex.Message;
        }
        finally
        {
            // Clear the prompt variable.
            prompt = "";

            // Set Processing to false to indicate
            // that the method is done processing.
            Processing = false;

            // Call StateHasChanged to refresh the UI.
            StateHasChanged();
        }
    }
```

This matches the code from the previous chapter.

To define the functions, add the following code under *** **Add new code here** ***:

*Code Listing 20: Define the Functions*

```
// *** FUNCTIONS ***
```

```
var DefinedFunctions = new List<Function>
{
    new Function(
        "Todos_POST",
        @"Creates a new TODO item.
            Use this function to add a new TODO item to the list.".Trim(),
        new JsonObject
        {
            ["type"] = "object",
            ["properties"] = new JsonObject
            {
                ["TodoRequest"] = new JsonObject
                {
                    ["type"] = "object",
                    ["properties"] = new JsonObject
                    {
                        ["todo"] = new JsonObject
                        {
                            ["type"] = "string",
                            ["description"] = @"The TODO item to be added."
                        }
                    },
                    ["required"] = new JsonArray { "todo" }
                }
            },
            ["required"] = new JsonArray { "TodoRequest" }
        }),
    new Function(
        "Todos_GET",
        @"Retrieves the TODO list.
            Use this function to view the TODO list.".Trim(),
        new JsonObject
        {
            ["type"] = "object",
            ["properties"] = new JsonObject {}
        }),
    new Function(
        "Todos_DELETE",
        @"Deletes a specific TODO item from the list.
            Use this function to remove a TODO item from the list.".Trim(),
        new JsonObject
        {
            ["type"] = "object",
            ["properties"] = new JsonObject
            {
                ["TodoIndexRequest"] = new JsonObject
                {
                    ["type"] = "object",
                    ["properties"] = new JsonObject
                    {
                        ["todoIdx"] = new JsonObject
                        {
                            ["type"] = "integer",
                            ["description"] = @"The index of the TODO item 
to be deleted."
                        }
                    },
                    ["required"] = new JsonArray { "todoIdx" }
                }
            },
            ["required"] = new JsonArray { "TodoIndexRequest" }
        })
};
```

Next, add the following code to call the Azure OpenAI API.

Note that it passes the **`DefinedFunctions`** to the **`functions`** property. Also, the **`functionCall`** property is set to **"`auto`"**. This means the model can choose to call a function or not. This can also be set to instruct the model to call a specific defined function, or not to call any functions at all.

*Code Listing 21: Call the Model*

```
    // Call ChatGPT
    // Create a new ChatRequest object with the chat prompts and pass
    // it to the API's GetCompletionAsync method.

    // *** FUNCTIONS ***
    var chatRequest = new ChatRequest(
        chatMessages,
        functions: DefinedFunctions,
        functionCall: "auto",
        model: "gpt-3.5-turbo-0613", // Must use this model or higher.
        temperature: 0.0,
        topP: 1,
        frequencyPenalty: 0,
        presencePenalty: 0);

    var result = await api.ChatEndpoint.GetCompletionAsync(chatRequest);
```

To complete the method, add the code shown in Code Listing 22. This code handles the result from the model.

If the model needs to call a function, we will call **`ExecuteFunction`** method (we will implement this later).

We use a **`while`** loop because the model may need to make multiple function calls. For example, to add multiple items, it will call the **`Todos_POST`** function several times.

*Code Listing 22: While Loop*

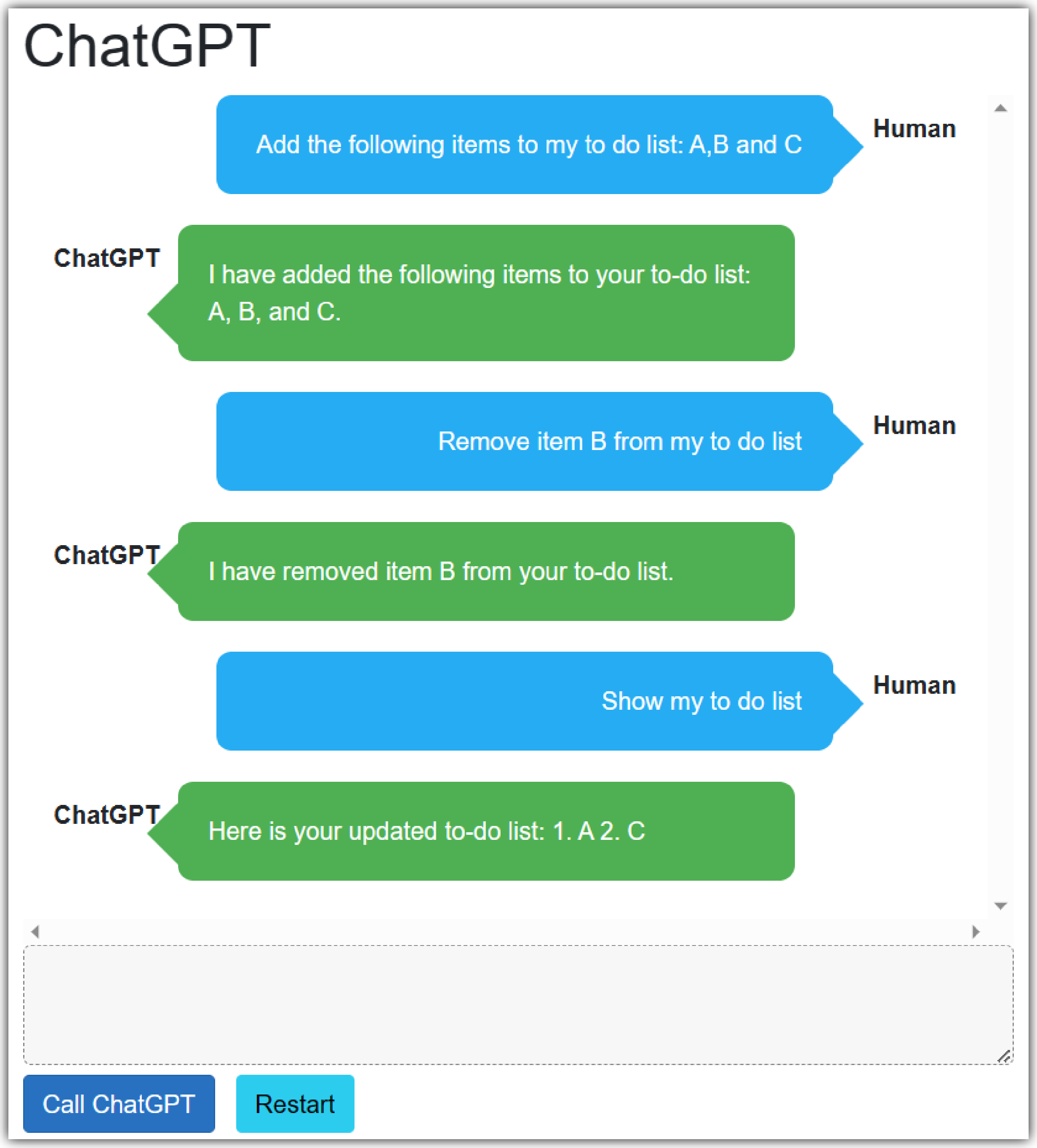
```
// *** FUNCTIONS ***
// See if ChatGPT wants to call a function as a response.
if (result.FirstChoice.FinishReason == "function_call")
{
    // Chat GPT wants to call a function.
    // To allow ChatGPT to call multiple functions
    // We need to start a While loop.
    bool FunctionCallingComplete = false;

    while (!FunctionCallingComplete)
    {
        // Call the function.
        chatMessages = ExecuteFunction(result, chatMessages);

        // Get a response from ChatGPT
        // (now that it has the results of the function).
        chatRequest = new ChatRequest(
            chatMessages,
            functions: DefinedFunctions,
            functionCall: "auto",
            model: "gpt-3.5-turbo", // Must use this model or higher.
            temperature: 0.0,
            topP: 1,
            frequencyPenalty: 0,
            presencePenalty: 0);

        result = await api.ChatEndpoint.GetCompletionAsync(chatRequest);

        var FunctionResult = result.Choices.FirstOrDefault();

        if (FunctionResult.FinishReason == "function_call")
        {
            // Keep looping.
            FunctionCallingComplete = false;
        }
        else
        {
            // Break out of the loop.
            FunctionCallingComplete = true;
        }
    }
}

// Add the response to chatMessages.
var choice = result.Choices.FirstOrDefault();

if (choice.Message != null)
```

```
{
    chatMessages.Add(choice.Message);
}
```

Finally, add the following code to implement the **`ExecuteFunction`** method.

*Code Listing 23: ExecuteFunction Method*

```
private List<Message> ExecuteFunction(
ChatResponse ChatResponseResult, List<Message> ParamChatPrompts)
{
    // Get the arguments.
    var functionArgs =
    ChatResponseResult.FirstChoice.Message.Function.Arguments.ToString();

    // Get the function name.
    var functionName =
    ChatResponseResult.FirstChoice.Message.Function.Name;

    // Variable to hold the function result.
    string functionResult = "";

    // Use select case to call the function.
    switch (functionName)
    {
        case "Todos_POST":
            var NewTODO =
            JsonSerializer.Deserialize<ToDoAddRequest>(functionArgs);
            if (NewTODO != null)
            {
                functionResult = AddTodo(NewTODO.TodoRequest.todo);
            }
            break;
        case "Todos_GET":
            functionResult = GetTodos();
            break;
        case "Todos_DELETE":
            var DeleteTODO =
            JsonSerializer.Deserialize<ToDoRemoveRequest>(functionArgs);
            if (DeleteTODO != null)
            {
                functionResult =
                DeleteTodo(DeleteTODO.TodoIndexRequest.todoIdx);
            }
            break;
        default:
            break;
    }
    // Call ChatGPT again with the results of the function.
    ParamChatPrompts.Add(
        new Message(Role.Function, functionResult, functionName)
    );

    return ParamChatPrompts;
}
```


ChatGPT

Human: Add the following items to my to do list: A,B and C

ChatGPT: I have added the following items to your to-do list: A, B, and C.

Human: Remove item B from my to do list

ChatGPT: I have removed item B from your to-do list.

Human: Show my to do list

ChatGPT: Here is your updated to-do list: 1. A 2. C

*Figure 34: Manage TODO Items*

Run the application. You can now manage items on your to-do list using natural language.

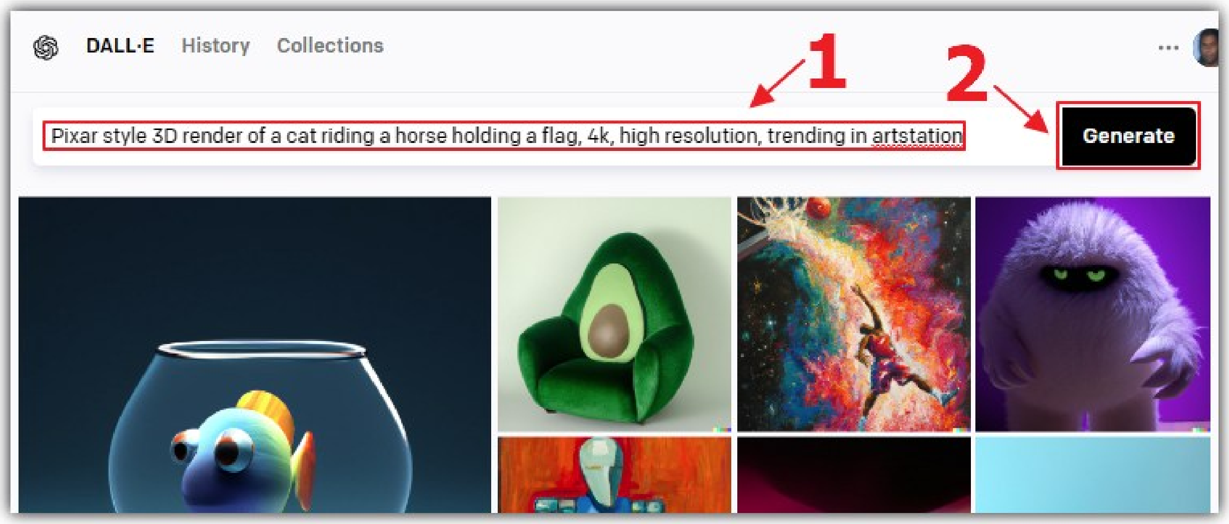
# Chapter 5  DALL-E

OpenAI has developed a popular product called DALL-E, which can create original pictures based on written descriptions. It uses algorithms to understand a text prompt and creates images that match it. For example, if you describe a landscape, the program can create a picture of that landscape even though it doesn't actually exist.

***Note: DALL-E was named after the artist Salvador Dali and the character EVE from the movie WALL-E.***

While DALL-E incorporates some of the same fundamental concepts as GPT, it is a distinct and separate program with a different focus and functionality. It is used for creating images, editing images, and creating variations of images.

## Using DALL-E



Pixar style 3D render of a cat riding a horse holding a flag, 4k, high resolution, trending in artstation

*Figure 35: Using DALL-E*

To use DALL-E, navigate to the website and log in with your OpenAI account. Input your textual prompt in the text box and press **Generate**.

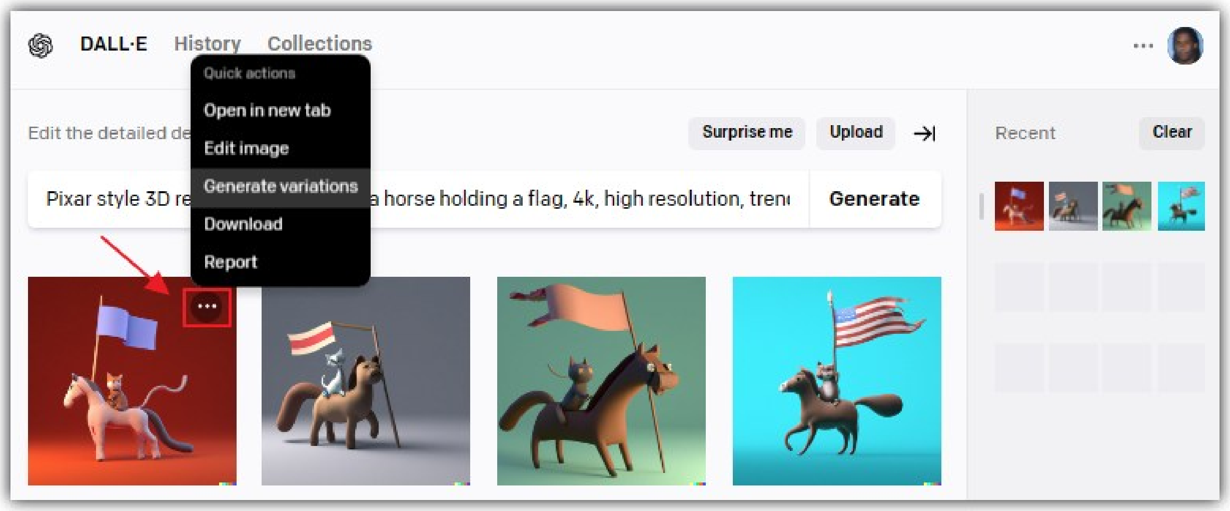


*Figure 36: DALL-E Results*

The DALL-E model will then generate images based on your prompt, which you can view and download. You can select an image to edit it or create more variations based on that image.

## Create a Dall-E application using Blazor

We will now cover the steps to call the DALL-E API endpoint and generate pictures in our Blazor application.

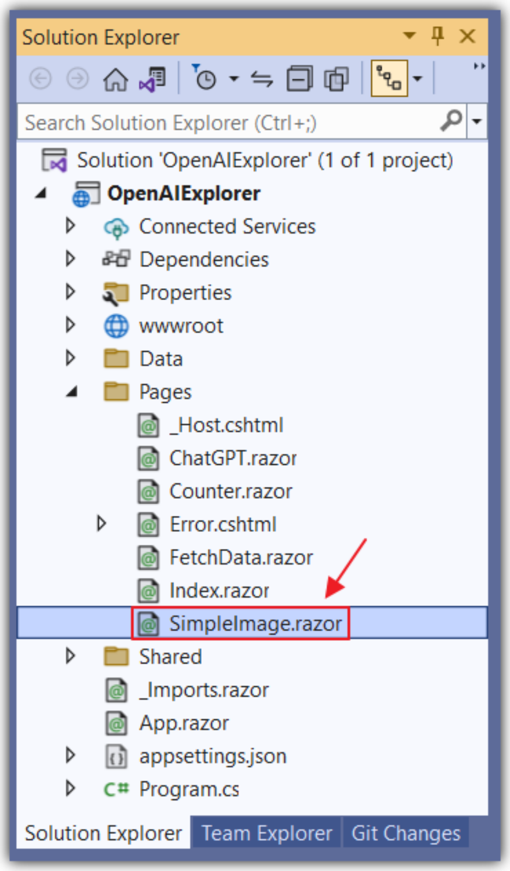


*Figure 37: Add SimpleImage.razor*

In Visual Studio, in the OpenAIExplorer project, add a new Razor control called **SimpleImage.razor**.

Replace all the code with the following code.

*Code Listing 24*

```
@page "/simpleimage"
@using OpenAI;
@using OpenAI.Images;
@using OpenAI.Models;
@inject IConfiguration _configuration
<PageTitle>Simple Image</PageTitle>

<h1>Describe the desired image</h1>
<input class="form-control" type="text"
       @bind="prompt" />
<br />
<button class="btn btn-primary"
        @onclick="CallService">
    Call The Service
```

```
</button>
<br />
<br />
<h4>Response:</h4>
<br />
<p>@response</p>
<p>@((MarkupString)ImageResponse)</p>
```

This code displays an input field that is bound to a *prompt* variable. Users can enter text in this field to describe the image they want to generate.

Below the input field is a button labeled Call the Service. When the user clicks this button, the **`CallService`** method is called, which uses the OpenAI DALL-E API to generate an image based on the user's input.

While the images are being generated, the page displays a response message indicating the status. This message is stored in a variable that is displayed in a paragraph element.

The generated images are stored in an **`ImageResponse`** variable and displayed in a paragraph element using a **`MarkupString`** object, which allows HTML code to be rendered as actual HTML on the page. The generated image will be displayed as an HTML **`<img>`** tag containing the image source URL.

Next, add the following code.

*Code Listing 25*

```
@code {
    string Organization = "";
    string ApiKey = "";
    string prompt = "";
    string response = "";
    string ImageResponse = "";
    protected override void OnInitialized()
    {
        Organization = _configuration["OpenAIServiceOptions:Organization"]
?? "";
        ApiKey = _configuration["OpenAIServiceOptions:ApiKey"] ?? "";

        prompt = "Pixar style 3D render of a cat ";
        prompt = prompt + "riding a horse holding a flag, 4k, ";
        prompt = prompt + "high resolution, trending in artstation";
    }
}
```

This code sets up the necessary variables and retrieves the **`Organization`** and **`ApiKey`**. It also constructs a default prompt to request a picture of a cat. The end user will have the ability to change this prompt.

Finally, add the following code to the **`@code`** section, which will implement the method that calls the DALL-E API when the Call the Service button is clicked.

*Code Listing 26*

```
// This code defines an asynchronous method, CallService,
// which calls the OpenAI DALL-E API to generate images based on a given
prompt.
async Task CallService()
{
    // Set initial response message.
    response = "Calling service...";

    // Create a new instance of the OpenAIClient,
    // passing in an API key and organization to authenticate the client.
    var api = new OpenAIClient(new OpenAIAuthentication(ApiKey,
Organization));

    // Call the GenerateImageAsync method of the ImagesEndPoint object
    // of the OpenAIClient to generate images based on the given prompt.
    // The method is called asynchronously and the results are stored in
the
    // 'results' variable. The method is called with two arguments,
    // 'prompt' and '2', specifying the text prompt and the number of
images
    // to generate, respectively. The 'ImageSize.Small' argument specifies
that
    // the generated images should be small in size.
    var results =
    await api.ImagesEndPoint.GenerateImageAsync(prompt, 2,
ImageSize.Small);

    if (results.Count > 0)
    {
        response = "";

        // Iterate through each generated image and append the HTML code
        // for displaying the image to the 'ImageResponse' variable.
        foreach (var image in results)
        {
            ImageResponse += $@"<img src=""{image}"" />&nbsp;&nbsp;";
        }
    }
    else // If there are no generated images.
    {
        // Set response message to indicate error.
        response = "Unknown Error";
    }
}
```

Add a link to the control in the NavMenu.razor control and hit **F5** to run the application. Navigate to the Razor page, enter a prompt, and click **Call The Service**.

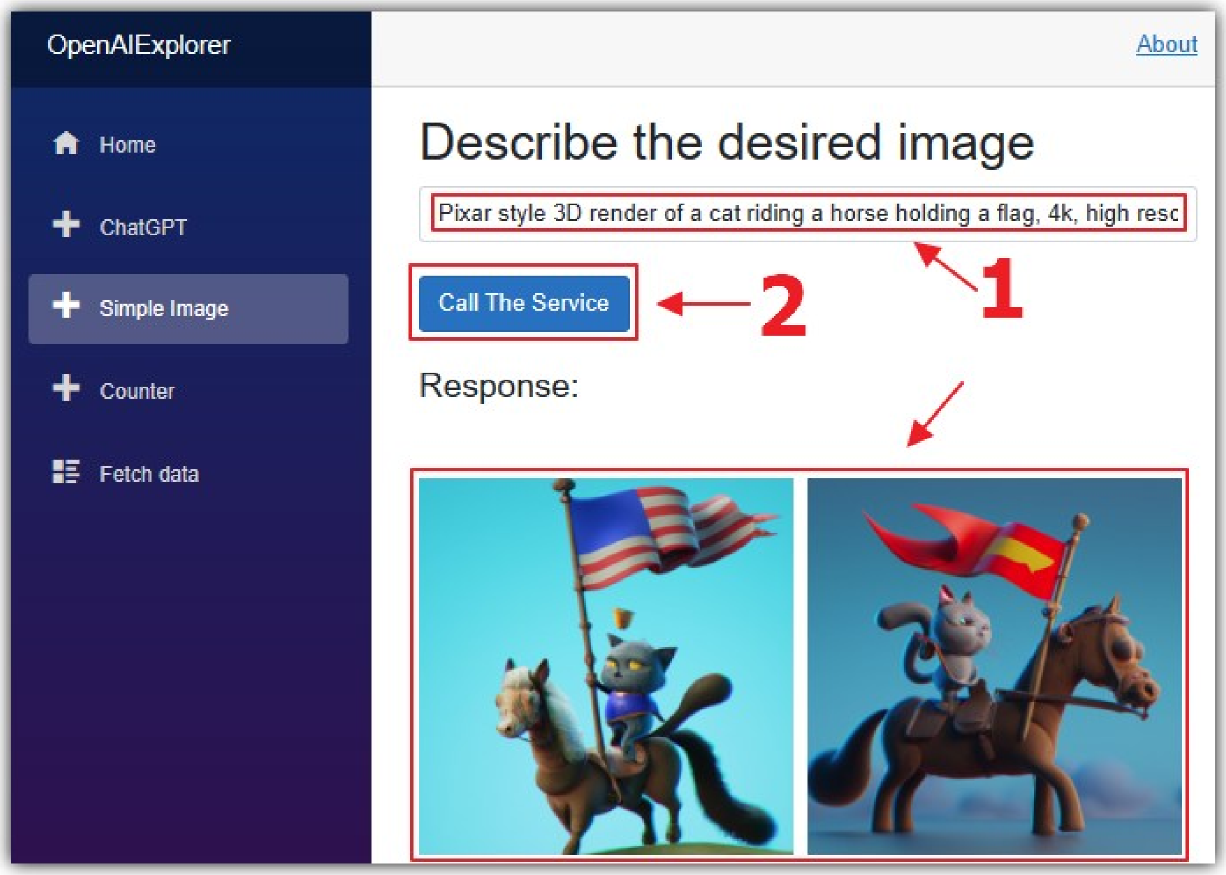

OpenAIExplorer
About
Home
ChatGPT
Simple Image
Counter
Fetch data
Describe the desired image
Pixar style 3D render of a cat riding a horse holding a flag, 4k, high resc
Call The Service
2
1
Response:

*Figure 38: Calling the DALL-E API*

The DALL-E API will return two images, and the application will display them.

***You can find the documentation for the Images API [here](#).***

# Chapter 6  Whisper

The Whisper API from OpenAI is a speech-to-text API that uses the Whisper model to transcribe audio. The model has been trained on a diverse dataset of audio and is capable of multilingual speech recognition, speech translation, and language identification.

***Note: To use the Whisper API, files must be under 25 MB in size. For audio files larger than this, you'll have to either split them into 25 MB or smaller chunks or convert them to a compressed audio format.***

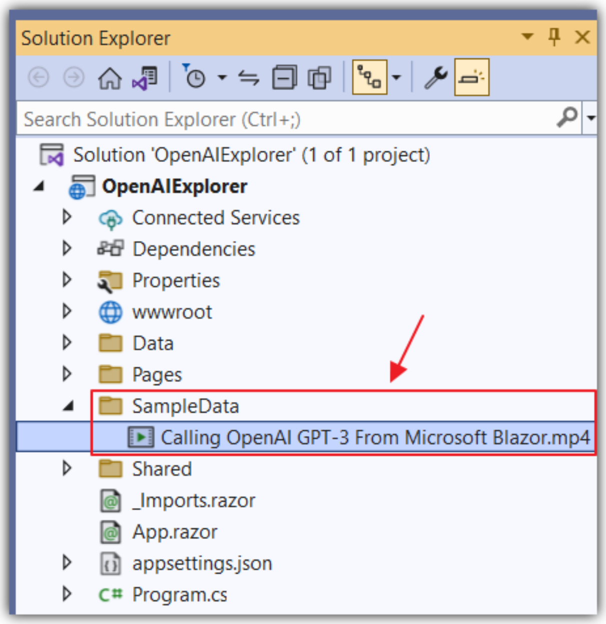


*Figure 39: Add SampleData Folder and .MP4 File*

## Create a Blazor transcription application

To demonstrate what Whisper can do, we will open the OpenAIExplorer project, add a new folder called **SampleData**, and an .mp4 file (you can download a sample file using the following link: https://bit.ly/3LX55Ic).

***Note: File types accepted are: mp3, mp4, mpeg, mpga, m4a, wav, and webm.***

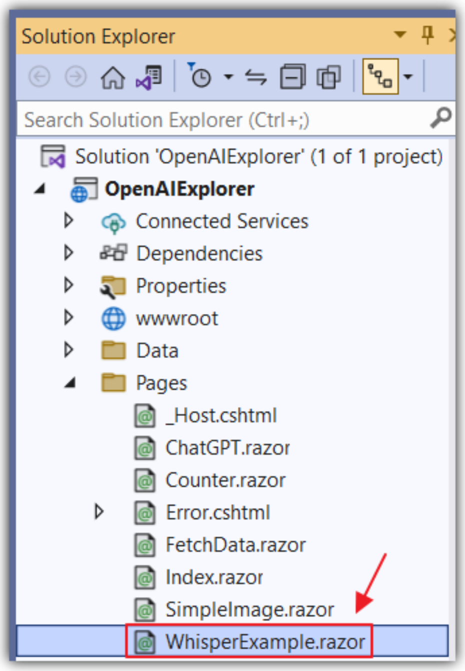


*Figure 40: Add WhisperExample.razor*

Next, we will add a new Razor control called WhisperExample.razor to the project, which will transcribe the audio of a video file.

Replace all the code in the file with the following code.

*Code Listing 27*

```
@page "/whisperexample"
@using OpenAI;
@using OpenAI.Audio;
@using OpenAI.Edits;
@using OpenAI.Models;
@inject IConfiguration _configuration
<PageTitle>Whisper Example</PageTitle>

<h1>Whisper Transcription</h1>

<button class="btn btn-primary"
        @onclick="UploadFile">
    Upload File
</button>
<br />
<br />
<h4>Response:</h4>
@if (!Processing)
{
    <textarea rows="10" cols="75" @bind="response" />
}
else
{
    <br>
    <h4>Processing...</h4>
}
```

The code contains an Upload File button that calls a function called **`UploadFile`** when clicked. This function will upload the file to the speech-to-text API to be transcoded by the Whisper model (into the language the audio is in).

Next, there is a statement that checks the value of the **`Processing`** variable. If the value is false, then the code displays a **`<textarea>`** element that will display the text transcription of the audio in the file, after the file has been uploaded and processed.

Add the following code, which will set up the necessary variables and retrieve the **`Organization`** and **`ApiKey`**.

*Code Listing 28*

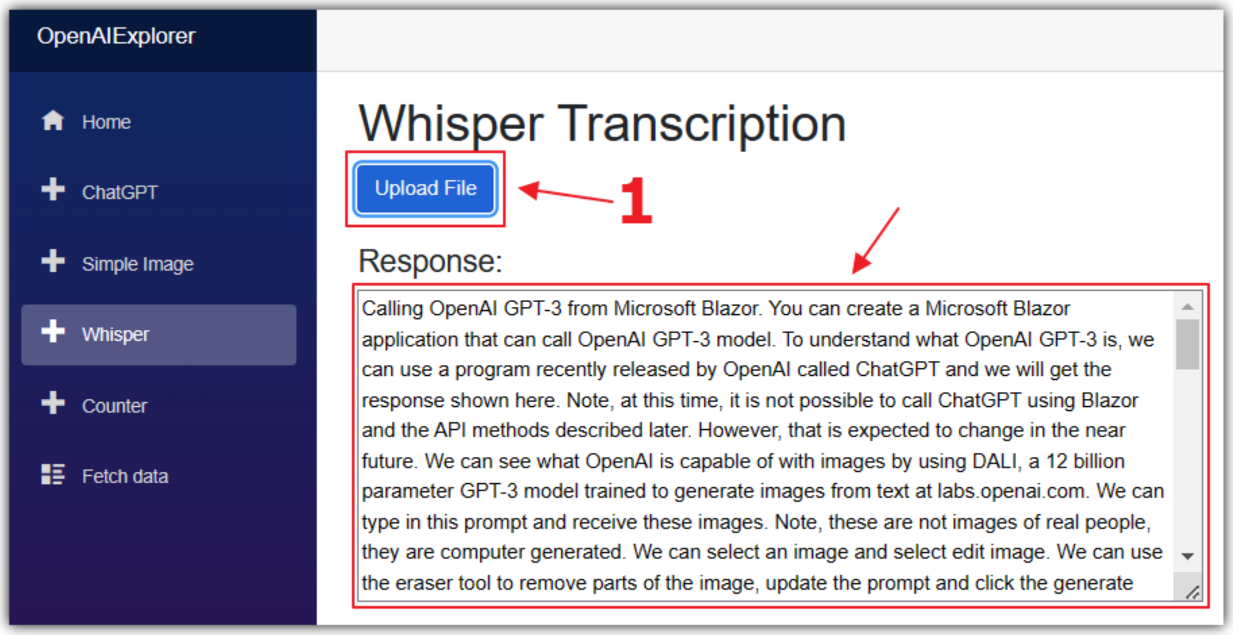
```
@code {
    string Organization = "";
    string ApiKey = "";
    string response = "";
    bool Processing = false;

    protected override void OnInitialized()
    {
        Organization = _configuration["OpenAIServiceOptions:Organization"]
?? "";
        ApiKey = _configuration["OpenAIServiceOptions:ApiKey"] ?? "";
    }
}
```

Finally, add the following code to the **`@code`** section, which will respond to the click of the Upload File button, upload the file to the **`CreateTranscriptionAsync`** method of the **`AudioEndpoint`**, and receive the response.

*Code Listing 29*

```
private async Task UploadFile()
{
    // Set response to empty string.
    response = "";

    // Set Processing to true.
    Processing = true;

    // Trigger a UI refresh.
    StateHasChanged();

    // Create an instance of the OpenAIClient
    // using the provided API key and organization.
    var api =
    new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

    // Set the audioAssetPath to a sample audio file.
    string audioAssetPath =
    @"SampleData/Calling OpenAI GPT-3 From Microsoft Blazor.mp4";

    // Create an AudioTranscriptionRequest object
    // with the audio file path and language.
    var request =
    new AudioTranscriptionRequest(
        Path.GetFullPath(audioAssetPath), language: "en");

    // Perform audio transcription using the
    // OpenAI API and set the response string.
    response =
    await api.AudioEndpoint.CreateTranscriptionAsync(request);

    // Set Processing to false and trigger a UI refresh.
    Processing = false;
    StateHasChanged();
}
```

Add a link to the control in the NavMenu.razor control and hit **F5** to run the application. Navigate to the Razor page and click **Upload File**.



Calling OpenAI GPT-3 from Microsoft Blazor. You can create a Microsoft Blazor application that can call OpenAI GPT-3 model. To understand what OpenAI GPT-3 is, we can use a program recently released by OpenAI called ChatGPT and we will get the response shown here. Note, at this time, it is not possible to call ChatGPT using Blazor and the API methods described later. However, that is expected to change in the near future. We can see what OpenAI is capable of with images by using DALI, a 12 billion parameter GPT-3 model trained to generate images from text at labs.openai.com. We can type in this prompt and receive these images. Note, these are not images of real people, they are computer generated. We can select an image and select edit image. We can use the eraser tool to remove parts of the image, update the prompt and click the generate

*Figure 41: Display Transcription*

After a few moments, the text transcription of the audio in the file will display in the Response box.

***Note: The Speech to Text API also translates audio into different languages. The full documentation for the Audio API is located here.***

# Chapter 7  Fine-Tuning and Embedding

*Figure 42: Abstract Image of AI Thinking*

In this chapter, we will cover fine-tuning a model and creating and consuming embeddings.

Fine-tuning is a technique used to adapt an already-trained model to a specific task, while embeddings are a way of transforming text or other data into a more suitable format for machines to process.

Both techniques are used in machine learning but operate at different levels of abstraction and serve different purposes.

Fine-tuning involves changing the way a model behaves to improve its performance on a specific task. It does this by adjusting the weights of the model's layers based on the new training data, which essentially modifies the model's internal representations and decision-making processes.

Embeddings, on the other hand, are a way of representing data in a more machine-readable format. An embedding transforms raw data, such as text or images, into a low-dimensional vector space where it can be more easily processed and analyzed by a model.

## Fine-tuning

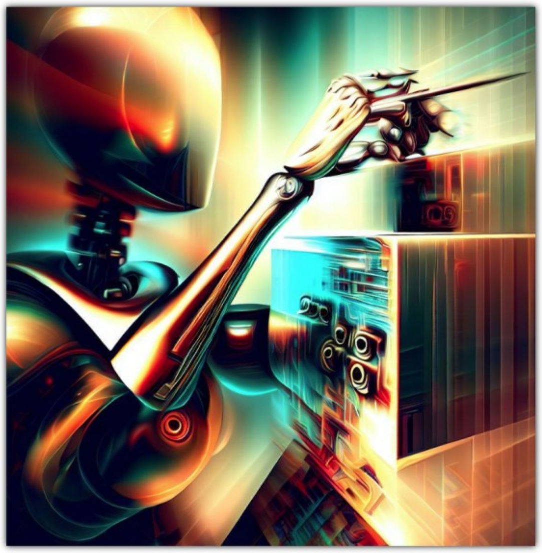


*Figure 43: Fine-Tuning the Abstract Image*

Fine-tuning is a powerful technique in machine learning that involves taking a model and further training it on a specific task or domain, enabling us to leverage its existing knowledge from a larger dataset to learn a new task or domain quickly with a smaller dataset.

For instance, if we have a pretrained model that recognizes objects in images and we want it to identify specific types of fruits in images, we can fine-tune the model by training it further on a smaller dataset of fruit images.

This allows us to adapt the model to the new task while still benefiting from its general knowledge from the larger dataset. Fine-tuning is widely used in natural language processing, computer vision, speech recognition, and other areas of machine learning to adapt pre-trained models quickly to new domains and achieve high accuracy with limited data.

### Create fine-tuning using Blazor

In this section, we will use Blazor to create and consume a fine-tuned model. The general outline of the process is:

1. Organize and upload training data.
2. Develop a refined model through training.
3. Implement the customized model.

***Note: At present, fine-tuning can be performed on these base models: DaVinci, Curie, Babbage, and Ada. These represent the initial models that lack any instruction-based training, such as text-davinci-003.***

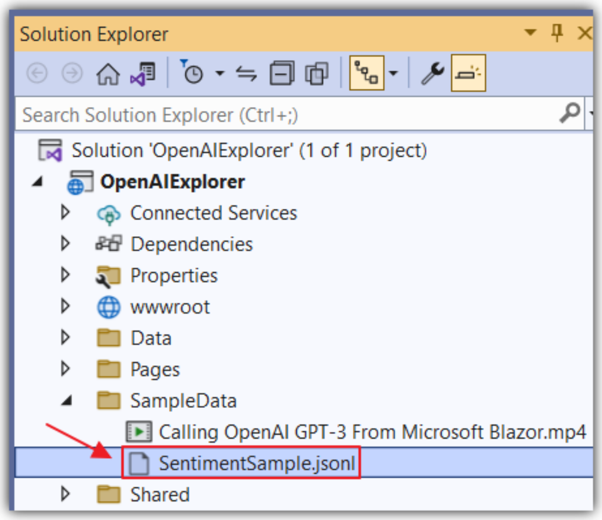
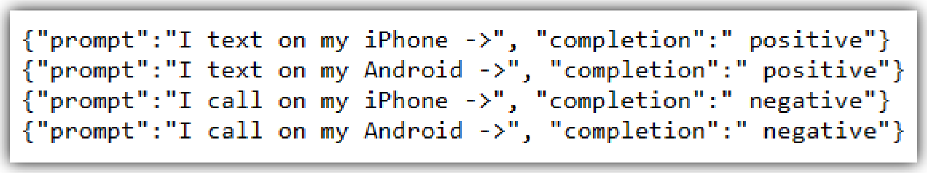


*Figure 44: Add SentimentSample.jsonl*

To add the training data, open the OpenAIExplorer project and add the following **SentimentSample.jsonl** file to the **SampleData** folder.



```
{"prompt":"I text on my iPhone ->", "completion":" positive"}
{"prompt":"I text on my Android ->", "completion":" positive"}
{"prompt":"I call on my iPhone ->", "completion":" negative"}
{"prompt":"I call on my Android ->", "completion":" negative"}
```

*Figure 45: Contents of the SentimentSample.jsonl file*

This sample data trains the model to treat texts as *positive* and calls as *negative*.

You can download the sample file using this link.

Model fine-tuning will require a collection of training samples, each containing a single input (referred to as a *prompt*) and its corresponding output (*completion*). The data must be in JSONL format, with each line representing a prompt-completion pair that corresponds to a specific training sample.

To denote the end of the prompt and the beginning of the completion, you need to utilize a consistent separator in every prompt. In this example, we will use `->`.

***Note: To make a model work better, you will want to fine-tune it on at least a few hundred good examples. After that, the model gets better as you double the number of examples. Adding more examples is usually the best way to make the model work better. You can find more guidance on preparing training datasets here.***

Next, add a new Razor control called FineTuningExample.razor to the project, which will allow you to fine-tune a model.

Replace all the code in the file with the following code.

*Code Listing 30*

```
@page "/finetuningexample"
@using OpenAI;
@using OpenAI.Files;
@using OpenAI.FineTuning;
@using OpenAI.Models;
@inject IJSRuntime JSRuntime
@inject IConfiguration _configuration
<PageTitle>Fine Tuning Example</PageTitle>
<style>
    .scrollable-list {
        height: 200px;
        overflow-y: scroll;
    }
</style>
<h1>Fine Tuning Example</h1>
<br />
<button class="btn btn-success"
        @onclick="UploadFile">
    Upload SentimentSample.jsonl File
</button>
<br />
<br />
@if (ColFileData.Count > 0)
{
    <h4>
        <button class="btn btn-secondary"
            @onclick="(() => ListFiles())">
            Refesh
        </button> Files
    </h4>
    <table class="table">
        <thead>
            <tr>
                <th></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (var file in ColFileData)
            {
                <tr>
                    <td>
                        <a href="javascript:void(0)"
                           onclick="@(() => DownloadFile(file))">
                            @file.FileName
                        </a> (@file.Status - @file.Size bytes)
[@file.CreatedAt]
                    </td>
                    <td>
                        <!--  Only allow fine-tune jobs to be created -->
                        <!--  for files with the purpose "fine-tune". -->
                        @if (file.Purpose == "fine-tune")
                        {
                            <button class="btn btn-success"
                        @onclick="(() => CreateFineTuneJob(file))">
                                Create Fine Tune Job
                            </button>
                        }
                        &nbsp;
                        <button class="btn btn-danger"
                        @onclick="(() => DeleteFile(file))">
                            Delete
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
}
```

This Blazor component allows users to upload files, view a list of uploaded files, and perform actions such as downloading, creating a fine-tune job, and deleting files.

Add the following procedure code.

*Code Listing 31*

```
@code {
    // Declare variables for organization, API key,
    // and collections of objects.
    string Organization = "";
    string ApiKey = "";
    List<FileData> ColFileData = new List<FileData>();

    // Declare nullable objects for file data and fine-tuning job.
    FileData? fileData;
    FineTuneJob? fineTuneJob;

    // Declare collections for events, fine-tuning jobs, and models.
    List<FineTuneJob> colFineTuneJob = new List<FineTuneJob>();
    List<Model> colModels = new List<Model>();

    // Declare variables for completion model, prompt, and response.
    string CompletionModel = "";
    string CompletionPrompt = "I text on my iPhone ->";
    string CompletionResponse = "";

    // Initialize component, set organization and API key,
    // and call listing methods.
    protected override async Task OnInitializedAsync()
    {
        // Set the organization value from the configuration
        // or use an empty string if not found.
        Organization =
        _configuration["OpenAIServiceOptions:Organization"] ?? "";

        // Set the API key value from the configuration
        // or use an empty string if not found.
        ApiKey =
        _configuration["OpenAIServiceOptions:ApiKey"] ?? "";

        // Call the ListFiles method to list files
        // using the OpenAI API.
        await ListFiles();
    }

    // Files

    // List files using OpenAI API.
    private async Task ListFiles()
    {
        // Create a new OpenAI API client with
        // the provided API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Call the ListFilesAsync method to
        // fetch files from the OpenAI API.
        var files = await api.FilesEndpoint.ListFilesAsync();

        // Convert the fetched files to a list
        // and assign it to the ColFileData variable.
        ColFileData = files.ToList();
    }

    // FineTune

    private async Task CreateFineTuneJob(FileData paramaFile)
    {
        // To Be Implemented...
    }
}
```

The first step is for the end user to click the button labeled Upload SentimentSample.jsonl File, which calls the **`UploadFile`** method.

Add the following code to implement that method.

*Code Listing 32*

```
    // Upload file using OpenAI API.
    private async Task UploadFile()
    {
        // Create a new OpenAI API client with
        // the provided API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Upload the file located at "SampleData/SentimentSample.jsonl"
        // for fine-tuning using the OpenAI API.
        fileData =
        await api.FilesEndpoint.UploadFileAsync(
        @"SampleData/SentimentSample.jsonl", "fine-tune");
    }
```

The UI allows a user to delete any uploaded file. Add the following code to implement that method.

*Code Listing 33*

```
    // Delete file using OpenAI API.
    private async Task DeleteFile(FileData paramaFile)
    {
        // Create a new OpenAI API client with the provided
        // API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Delete the specified file (paramaFile) using the OpenAI API.
        var result = await api.FilesEndpoint.DeleteFileAsync(paramaFile);

        // Update the list of files by calling the ListFiles method.
        await ListFiles();
    }
```

The code also allows a user to download a file. Add the following code to implement that method.

*Code Listing 34*

```
    // Download file using OpenAI API.
    private async Task DownloadFile(FileData paramaFile)
    {
        // Create a new OpenAI API client with the provided
        // API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Download the specified file (paramaFile) using the
        // OpenAI API and store the path to the downloaded file.
        var downloadedFilePath =
        await api.FilesEndpoint.DownloadFileAsync(paramaFile, "data");

        // Load the content of the downloaded file into a byte array.
        var filecontents = System.IO.File.ReadAllBytes(downloadedFilePath);

        // Invoke a JavaScript function to save the downloaded
        // file content as a file with the original file name.
        await JSRuntime.InvokeVoidAsync(
        "saveAsFile",
        paramaFile.FileName,
        Convert.ToBase64String(filecontents.ToArray()));
    }
```

This method invokes a **saveAsFile** JavaScript method. Add the following code to the **_Host.cshtml** file to implement that method.

*Code Listing 35*

```
<script>
    function saveAsFile(filename, bytesBase64) {
        var link = document.createElement('a');
        link.download = filename;
        link.href = "data:application/octet-stream;base64," + bytesBase64;
        document.body.appendChild(link); // Needed for Firefox.
        link.click();
        document.body.removeChild(link);
    }
</script>
```

Add a link to the control in the **NavMenu.razor** control and hit **F5** to run the application.

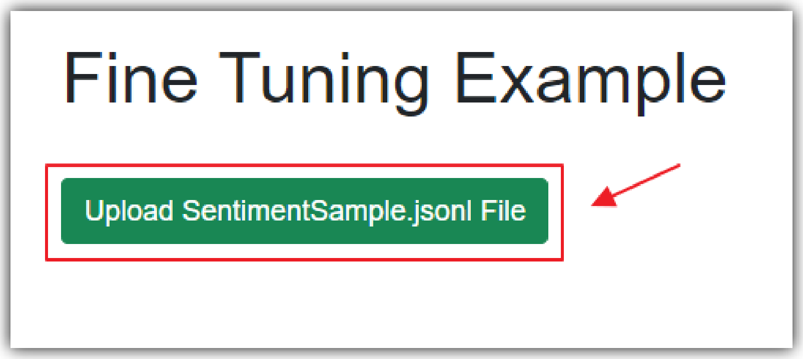


*Figure 46: Click Upload SentimentSample.jsonl File*

Navigate to the Razor page and click **Upload SentimentSample.jsonl File**.

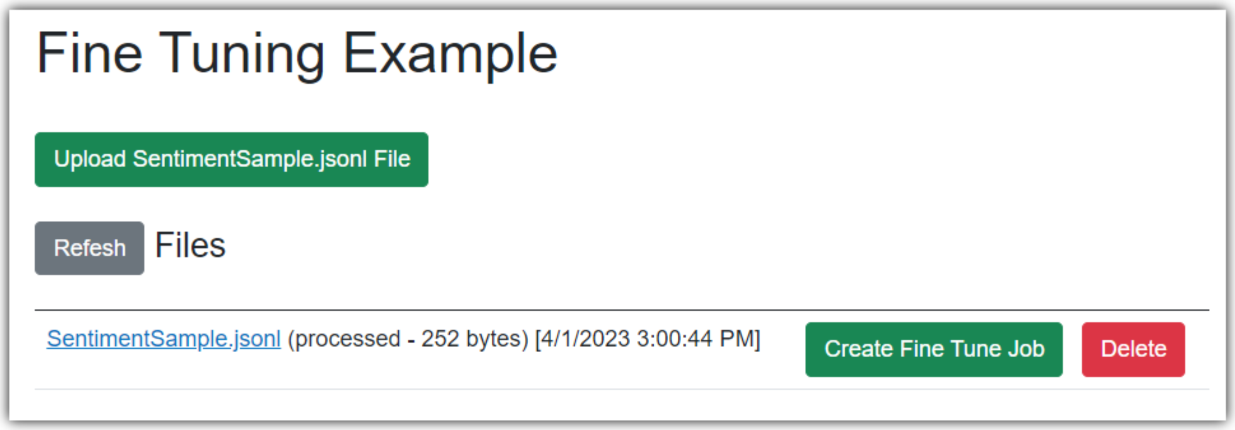

Fine Tuning Example
Upload SentimentSample.jsonl File
Refesh
Files
SentimentSample.jsonl (processed - 252 bytes) [4/1/2023 3:00:44 PM]
Create Fine Tune Job
Delete

*Figure 47: View Fine Tune File*

Wait 30 seconds and then refresh your web browser page and navigate to the Razor page again.

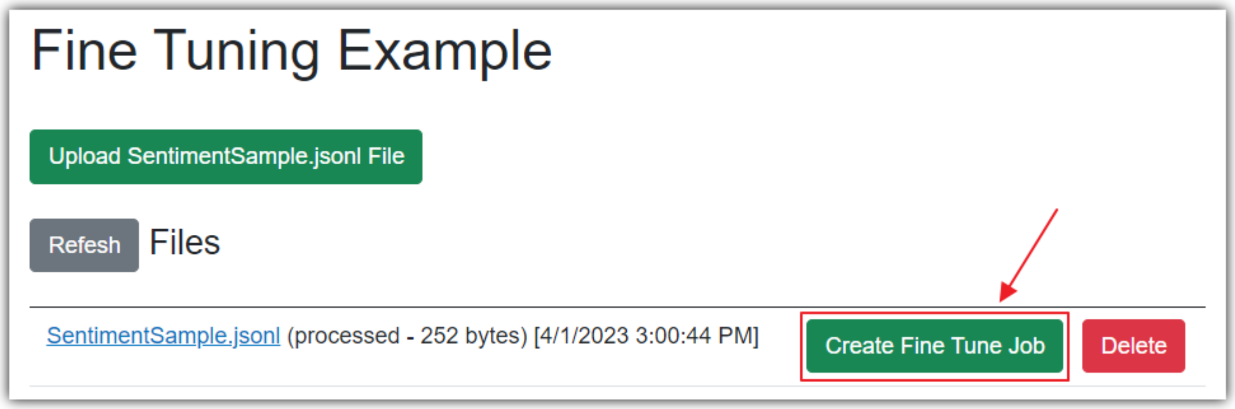
You will see the file listed. You can click the file name to download it or click the **Delete** button to delete it. Clicking the Create Fine Tune Job button doesn't do anything at this point because the method that it triggers, **`CreateFineTuneJob`**, has not been implemented yet.

Close your web browser to stop Visual Studio and replace the existing code for the **`CreateFineTuneJob`** method with the following code.

*Code Listing 36*

```
    private async Task CreateFineTuneJob(FileData paramaFile)
    {
        // Create a new OpenAI API client with the
        // provided API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));
        // Create a new fine-tuning job request with
        // the specified file (paramaFile).
        var request =
        new CreateFineTuneJobRequest(paramaFile);

        // Create a fine-tuning job using the OpenAI API
        // and store the result in the fineTuneJob variable.
        fineTuneJob =
        await api.FineTuningEndpoint.CreateFineTuneJobAsync(request);
    }
```

Hit **F5** to run the application.

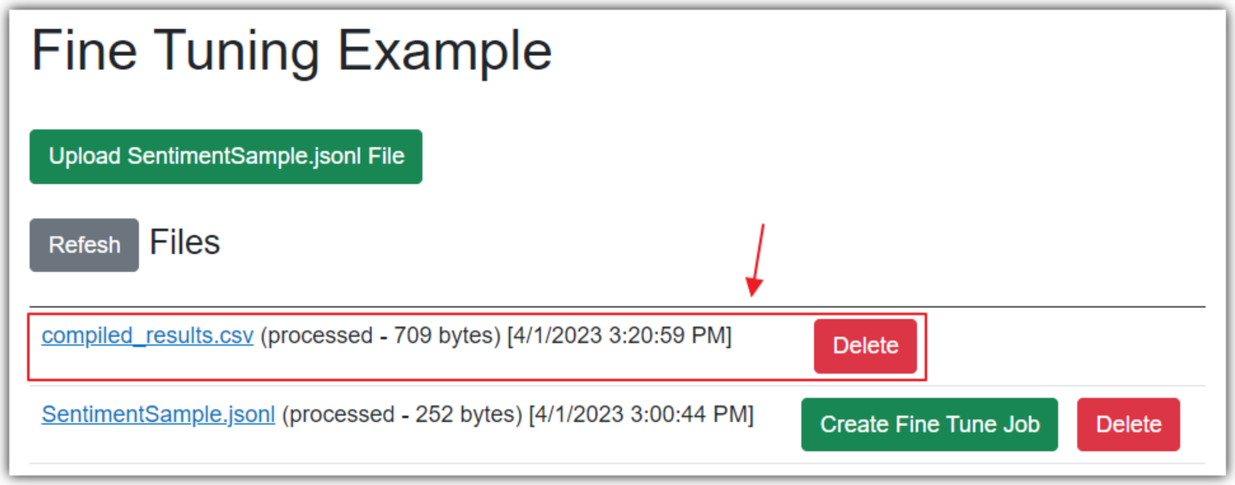


Fine Tuning Example

SentimentSample.jsonl (processed - 252 bytes) [4/1/2023 3:00:44 PM]

*Figure 48: Create Fine Tune Job*

Click **Create Fine Tune Job**.



Fine Tuning Example

compiled_results.csv (processed - 709 bytes) [4/1/2023 3:20:59 PM]

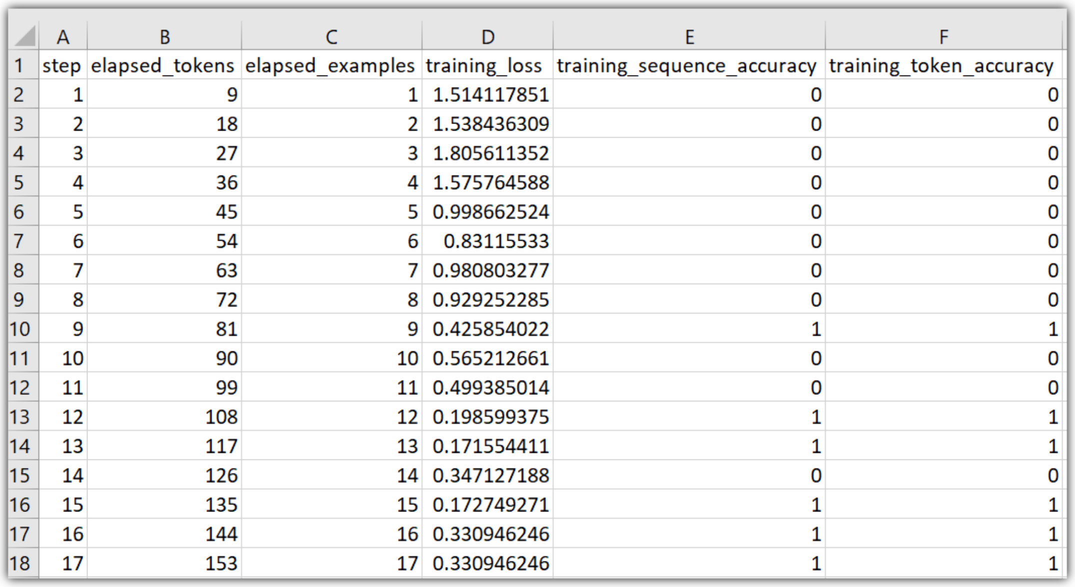
SentimentSample.jsonl (processed - 252 bytes) [4/1/2023 3:00:44 PM]

*Figure 49: Create Fine Tune Job Completed*

Wait about 30 minutes, then refresh your web browser and return to the page.

You should see a compiled_results.csv file listed.

***Note: If the results file doesn't appear, the OpenAI service may be running slower because of demand. Wait an hour and check back again.***



| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | step | elapsed_tokens | elapsed_examples | training_loss | training_sequence_accuracy | training_token_accuracy |
| 2 | 1 | 9 | 1 | 1.514117851 | 0 | 0 |
| 3 | 2 | 18 | 2 | 1.538436309 | 0 | 0 |
| 4 | 3 | 27 | 3 | 1.805611352 | 0 | 0 |
| 5 | 4 | 36 | 4 | 1.575764588 | 0 | 0 |
| 6 | 5 | 45 | 5 | 0.998662524 | 0 | 0 |
| 7 | 6 | 54 | 6 | 0.83115533 | 0 | 0 |
| 8 | 7 | 63 | 7 | 0.980803277 | 0 | 0 |
| 9 | 8 | 72 | 8 | 0.929252285 | 0 | 0 |
| 10 | 9 | 81 | 9 | 0.425854022 | 1 | 1 |
| 11 | 10 | 90 | 10 | 0.565212661 | 0 | 0 |
| 12 | 11 | 99 | 11 | 0.499385014 | 0 | 0 |
| 13 | 12 | 108 | 12 | 0.198599375 | 1 | 1 |
| 14 | 13 | 117 | 13 | 0.171554411 | 1 | 1 |
| 15 | 14 | 126 | 14 | 0.347127188 | 0 | 0 |
| 16 | 15 | 135 | 15 | 0.172749271 | 1 | 1 |
| 17 | 16 | 144 | 16 | 0.330946246 | 1 | 1 |
| 18 | 17 | 153 | 17 | 0.330946246 | 1 | 1 |

*Figure 50: Fine Tuning Results*

You can click the results file to download and examine it to see the results of the fine-tuning process.

To view the status of all the fine tune jobs, add the following markup code.

*Code Listing 37*

```
<!-- FineTuneJobs -->
@if (colFineTuneJob.Any())
{
    <h4>
        <button class="btn btn-secondary"
            @onclick="(() => ListFineTuneJobs())">
            Refesh
        </button> FineTune Jobs
    </h4>
    <div class="scrollable-list">
        <ul>
            @foreach (
           var job in colFineTuneJob.OrderByDescending(x => x.CreatedAt))
            {
                <li>@job.Id @job.FineTunedModel (@job.Status)
[@job.CreatedAt]</li>
            }
        </ul>
    </div>
    <br />
}
```

This code contains a button to refresh the list if needed. This button calls the **`ListFineTuneJobs`** method.

Add the following code to implement the **`ListFineTuneJobs`** method.

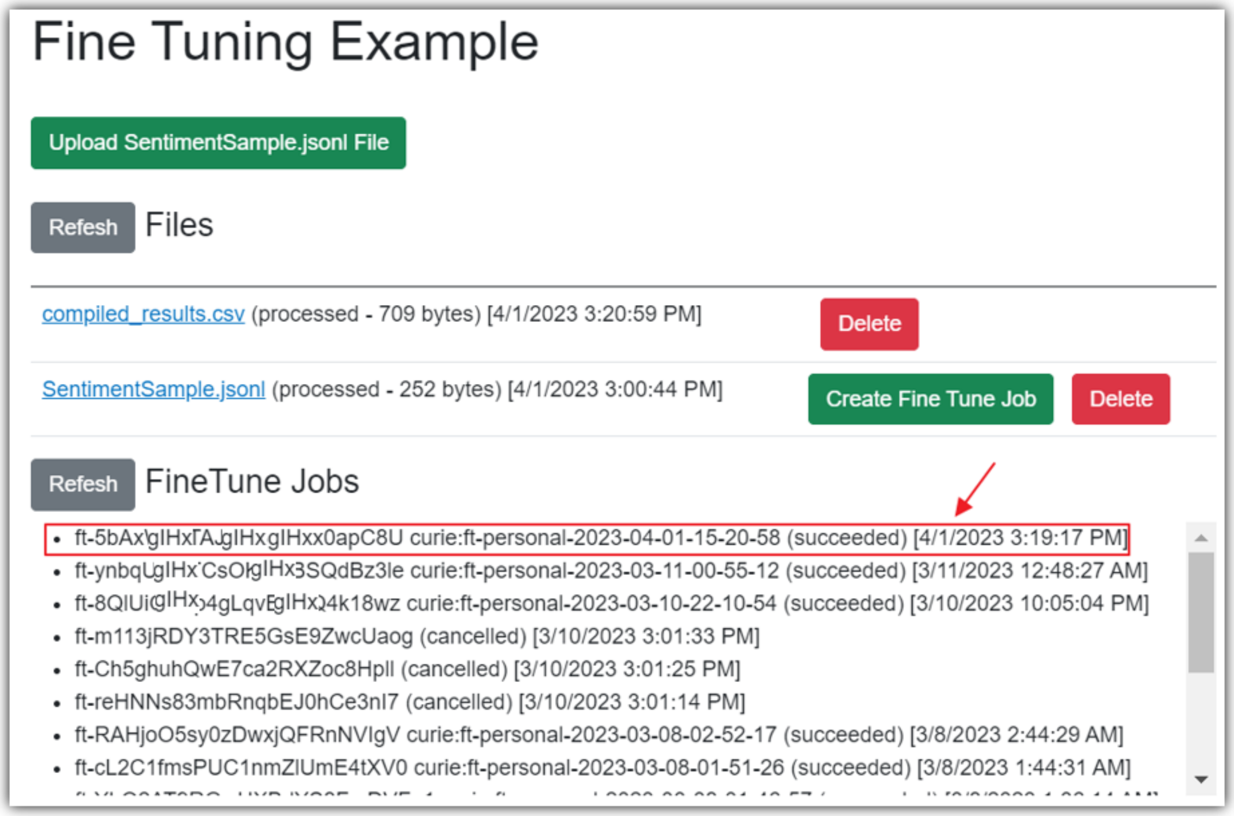
*Code Listing 38*

```
    private async Task ListFineTuneJobs()
    {
        // Clear collection.
        colFineTuneJob = new List<FineTuneJob>();

        // Create a new OpenAI API client with the provided
        // API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Fetch the list of fine-tuning jobs using the OpenAI API.
        var fineTuneJobs =
        await api.FineTuningEndpoint.ListFineTuneJobsAsync();

        // Add each fetched fine-tuning job to the colFineTuneJob list.
        foreach (var job in fineTuneJobs)
        {
            colFineTuneJob.Add(job);
        }
    }
```

Finally, add the following code to the **`OnInitializedAsync`** method.

*Code Listing 39*

```
        // Call the ListFineTuneJobs method to list
        // fine-tune jobs using the OpenAI API.
        await ListFineTuneJobs();
```



Fine Tuning Example

Upload SentimentSample.jsonl File

Refesh Files

compiled_results.csv (processed - 709 bytes) [4/1/2023 3:20:59 PM] Delete

SentimentSample.jsonl (processed - 252 bytes) [4/1/2023 3:00:44 PM] Create Fine Tune Job Delete

Refesh FineTune Jobs

- ft-5bAxgIHxTAJgIHxgIHxx0apC8U curie:ft-personal-2023-04-01-15-20-58 (succeeded) [4/1/2023 3:19:17 PM]
- ft-ynbqUgIHxCsOgIHx3SQdBz3le curie:ft-personal-2023-03-11-00-55-12 (succeeded) [3/11/2023 12:48:27 AM]
- ft-8QIUigIHx4gLqvEgIHx4k18wz curie:ft-personal-2023-03-10-22-10-54 (succeeded) [3/10/2023 10:05:04 PM]
- ft-m113jRDY3TRE5GsE9ZwcUaog (cancelled) [3/10/2023 3:01:33 PM]
- ft-Ch5ghuhQwE7ca2RXZoc8Hpll (cancelled) [3/10/2023 3:01:25 PM]
- ft-reHNNs83mbRnqbEJ0hCe3nl7 (cancelled) [3/10/2023 3:01:14 PM]
- ft-RAHjoO5sy0zDwxjQFRnNVIgV curie:ft-personal-2023-03-08-02-52-17 (succeeded) [3/8/2023 2:44:29 AM]
- ft-cL2C1fmsPUC1nmZlUmE4tXV0 curie:ft-personal-2023-03-08-01-51-26 (succeeded) [3/8/2023 1:44:31 AM]

*Figure 51: View Fine Tune Jobs*

Hit **F5** to run the application. Navigate to the page and view the status of all jobs. If there were any errors, or the job is still processing, you will see that here. If we see that the job has succeeded, there will be a model available that we can now consume.

To view a list of available fine-tune models, add the following code markup.

*Code Listing 40*

```
<!-- Models -->
@if (colModels.Count > 0)
{
    <h4>
        <button class="btn btn-secondary"
            @onclick="(() => ListModels())">
            Refesh
        </button> Models
    </h4>
    <table class="table">
        <thead>
            <tr>
                <th></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (var model in colModels)
            {
                <tr>
                    <td>
                        @model.Id
                    </td>
                    <td>
                        <button class="btn btn-danger"
                        @onclick="(() => DeleteModel(model))">
                            Delete
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
}
<br />
```

This code contains a button to refresh the list if needed. This button calls the **`ListModels`** method.

Add the following code to implement the **`ListModels`** method.

*Code Listing 41*

```
    // Models

    private async Task ListModels()
    {
        // Create a new OpenAI API client with the provided
        // API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Fetch the list of models using the OpenAI API.
        var models =
        await api.ModelsEndpoint.GetModelsAsync();

        // Initialize a new list to store filtered models.
        colModels = new List<Model>();

        // Iterate through the fetched models.
        foreach (var model in models)
        {
            // Filter out models owned by "openai" or "system".
            if (!model.OwnedBy.Contains("openai")
            && !model.OwnedBy.Contains("system"))
            {
                // Add the filtered model to the colModels list.
                colModels.Add(model);
            }
        }
    }
```

The UI allows a user to delete a selected model. Add the following code to implement that method.

*Code Listing 42*

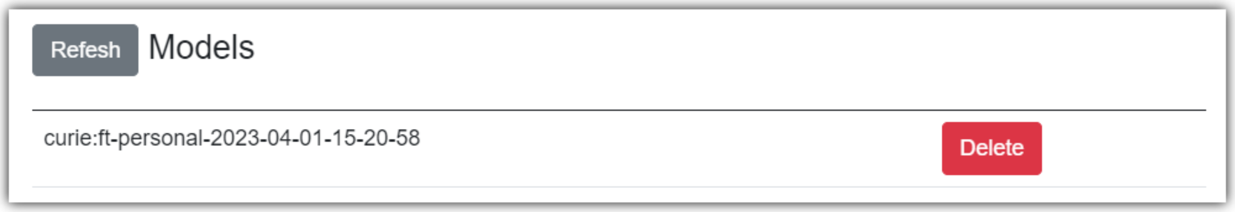
```
    private async Task DeleteModel(Model paramaModel)
    {
        // Create a new OpenAI API client with the provided
        // API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Delete the specified fine-tuned model (paramaModel)
        // using the OpenAI API.
        var result =
        await api.ModelsEndpoint.DeleteFineTuneModelAsync(paramaModel.Id);

        // Update the list of models by calling the ListModels method.
        await ListModels();
    }
```

Finally, add the following code to the **`OnInitializedAsync`** method.

*Code Listing 43*

```
        // Call the ListModels method to list
        // models using the OpenAI API.
        await ListModels();
```

Hit **F5** to run the application. Navigate to the page and view the fine-tuned model(s).



curie:ft-personal-2023-04-01-15-20-58

*Figure 52: List Fine Tune Models*

Finally, we will add the code that will allow us to select our fine-tuned model and generate a completion.

Add the following markup code.

*Code Listing 44*

```
<!-- Completion -->
@if (colModels.Count > 0)
{
    <h4>
        <button class="btn btn-secondary"
            @onclick="(() => ListModels())">
            Refesh
        </button> Completions
    </h4>
    <p>
        Model:<span>&nbsp;</span>
        <select id="Completion" style="width:400px"
            @bind="@CompletionModel">
            <option value="">Select Model</option>
            @foreach (var model in colModels)
            {
                <option value="@model.Id">@model.Id</option>
            }
        </select>
        <span>&nbsp;</span>
        <input id="newJobName" type="text" style="width:400px"
           @bind="@CompletionPrompt" />
        <span>&nbsp;</span>
        <button type="button" class="btn btn-primary"
            @onclick="(() => SubmitCompletion())">
            Submit
        </button>
    </p>
    <br />
    <p>@CompletionResponse</p>
    <br />
}
```

This Blazor code creates a user interface for selecting a completion model from a dropdown, entering a prompt in the text box, and generating completion text by calling the **`SubmitCompletion`** method.

Add the following code to implement the **`SubmitCompletion`** method.

*Code Listing 45*

```
private async Task SubmitCompletion()
```

```
{
    // Check if a completion model is selected
    // (CompletionModel is not empty)
    if (CompletionModel != "")
    {
        // Reset the completion response text.
        CompletionResponse = "";

        // Create a new OpenAI API client with the
        // provided API key and organization.
        var api =
        new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

        // Fetch the details of the selected completion
        // model (CompletionModel) using the OpenAI API.
        var objModel =
        await api.ModelsEndpoint.GetModelDetailsAsync(CompletionModel);

        // Define a list of stop sequences for the completion.
        List<string> StopSequence = new List<string>() { "->", "." };

        // Generate a completion using the specified
        // prompt, model, and other parameters.
        var result =
        await api.CompletionsEndpoint.CreateCompletionAsync(
            CompletionPrompt, maxTokens: 100, echo: false,
            temperature: 0.1, stopSequences: StopSequence,
            model: objModel);

        // Concatenate the generated completion text to
        // the CompletionResponse.
        foreach (var completion in result.Completions)
        {
            CompletionResponse += completion.Text;
        }
    }
}
```

Press **F5** to run the application.

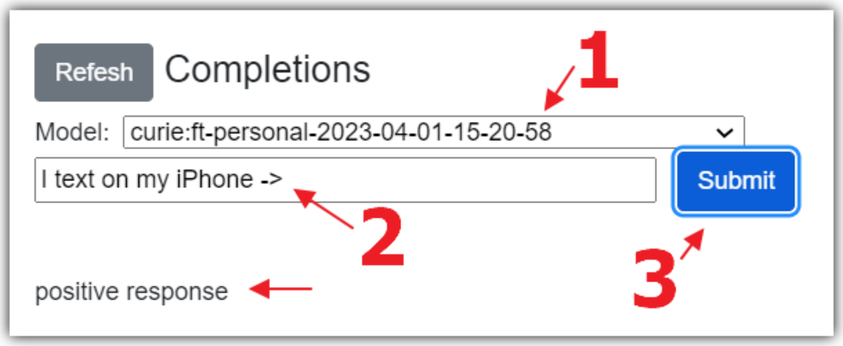


*Figure 53: Positive Response*

We can select the model from the dropdown and enter a prompt. Because the model was fine-tuned with data using the `->` delimiter, we must also use that in the prompt.

Clicking the **Submit** button will call the completion API endpoint, using the fine-tuned model. The response is then displayed.

As expected, because of the new data added for fine-tuning, the response is positive.

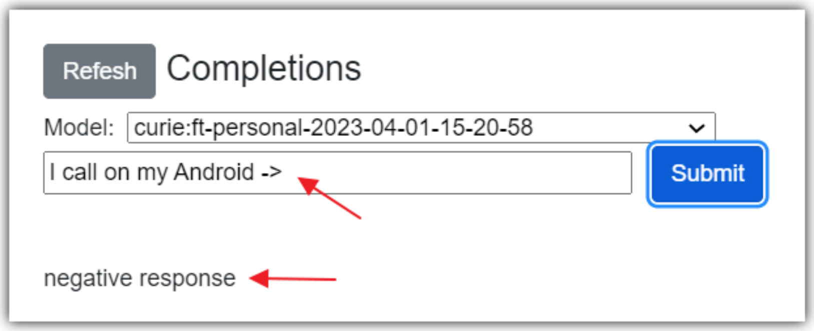


*Figure 54: Negative Response*

However, when we enter a prompt for a call, the completion is negative.

***Note: The documentation for the Fine-tunes API is located here.***

## Embeddings

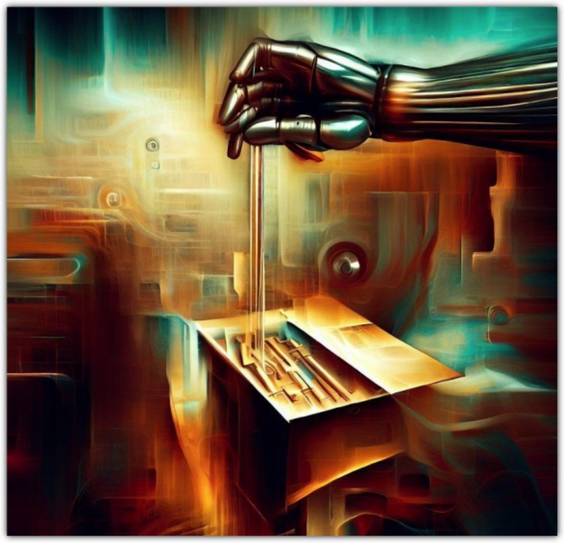


*Figure 55: Abstract Embedding Image*

An embedding is like a summary of what a piece of text means. Each embedding is a vector of floating-point numbers. Each element of an embedding has a specific value and position within the vector, and the relationships between these values help to represent the meaning of the original text in a way that machine learning models can understand. If two embeddings are close together, it means that the texts they represent are similar in meaning. So, if two pieces of text have similar meanings, their embeddings will also be similar.

By transforming raw data into a low-dimensional vector space, embeddings can capture meaningful relationships between data points that might be difficult or impossible to capture in the original format.

Essentially, word embeddings allow you to represent words as vectors in a high-dimensional space, capturing the semantic meaning of these words. For instance, you can obtain a vector for the word **car** and another set of vectors for the words **coffee**, **milk**, **latte**. With these vector representations, you can perform various tasks, such as classification and document search.

One common method to measure the similarity between vectors is cosine similarity. This will be demonstrated in the following code. By using cosine similarity, you can determine that the vectors for **car** and **tire** are close together, indicating that these words are semantically related. In this way, embeddings provide a powerful tool for computing similarity among documents and queries, enabling effective natural language processing applications.

## Create an embedding using Blazor

To demonstrate embeddings, open the OpenAIExplorer project and add a new Razor control called **EmbeddingExample.razor**, which will allow users to create embeddings and search for similar items.

Replace all the code in the file with the following code.

*Code Listing 46*

```
@page "/embeddings"
@using OpenAI;
@using OpenAI.Models;
@inject IConfiguration _configuration
<PageTitle>Embedding Example</PageTitle>

<h1>Embedding Example</h1>
<button class="btn btn-primary"
        @onclick="CreateEmbeddings">
    Create Embeddings
</button>
<br />
<br />
@if (embeddingCollection.GetAll().Count > 0)
{
    <div>
        <textarea rows="2" cols="25"
                  style="vertical-align:text-top"
              @bind="prompt"></textarea>
        <span>&nbsp;</span>
        <button class="btn btn-success"
                style="vertical-align:text-top"
            @onclick="CallService">
            Search
        </button>
    </div>
    <br />
    <ul>
        @foreach (var item in similarities)
        {
            <li>
                @item.Item1 - @item.Item2
            </li>
        }
    </ul>
}
```

This code creates a user interface for creating embeddings and searching for similar items based on the embeddings.

The code begins with a button labeled **`Create Embeddings`** that, when clicked, invokes the **`CreateEmbeddings`** method that creates embeddings.

The user can then input a query in the text area and click the Search button to find similar items. The results are then displayed as a list of items with their corresponding similarity scores.

Next, add the following code.

*Code Listing 47*

```
@code {
    string Organization = "";
    string ApiKey = "";
    string prompt = "latte";

    // Declare an embedding collection and a list to store similarities.
    EmbeddingCollection embeddingCollection = new EmbeddingCollection();
    List<(string?, float)> similarities = new List<(string?, float)>();

    // Initialize the component by setting the organization and API key.
    protected override void OnInitialized()
    {
        Organization =
        _configuration["OpenAIServiceOptions:Organization"] ?? "";

        ApiKey =
        _configuration["OpenAIServiceOptions:ApiKey"] ?? "";
    }

    // Classes

    public class Embedding
    {
        public string? Text { get; set; }
        public float[]? Values { get; set; }
    }

    public class EmbeddingCollection
    {
        // Declare a private list to store Embedding objects.
        private readonly List<Embedding> _embeddings = new
List<Embedding>();

        // Add an Embedding object to the list of embeddings.
        public void Add(Embedding embedding)
        {
            _embeddings.Add(embedding);
        }

        // Retrieve all Embedding objects in the list.
        public List<Embedding> GetAll()
        {
            return _embeddings;
        }
    }
}
```

This code sets up the necessary variables and retrieves the **`Organization`** and **`ApiKey`**. It also constructs a default prompt to request the embeddings to be related to the word **latte**. The end user will have the ability to change this prompt.

Next, add the following code to **`@code`** section. It will implement the **`CreateEmbeddings`** method that will be called when the **Create Embeddings** button is clicked.

*Code Listing 48*

```
// Create embeddings for the given terms using the OpenAI API.
private async Task CreateEmbeddings()
{
    // Create an instance of the OpenAI client.
    var api =
    new OpenAIClient(new OpenAIAuthentication(ApiKey, Organization));

    // Get the model details.
    var model =
    await api.ModelsEndpoint.GetModelDetailsAsync("text-embedding-ada-
002");

    // Create embeddings for each term.
    var coffee =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("coffee", model);

    var milk =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("milk", model);

    var HotWater =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("hot water", model);

    var chocolate =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("chocolate", model);

    var tea =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("tea", model);

    var chai =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("chai", model);

    var car =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("car", model);

    var tire =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync("tire", model);
    // Add the embeddings to the embedding collection.
    embeddingCollection.Add(
        new Embedding
            {
                Text = "coffee",
                Values = coffee.Data[0].Embedding
                .Select(d => (float)d).ToArray()
            });

    embeddingCollection.Add(
        new Embedding
            {
                Text = "milk",
                Values = milk.Data[0].Embedding
                .Select(d => (float)d).ToArray()
            });

    embeddingCollection.Add(
        new Embedding
            {
                Text = "chocolate",
                Values = chocolate.Data[0].Embedding
                .Select(d => (float)d).ToArray()
            });

    embeddingCollection.Add(
        new Embedding
            {
                Text = "tea",
                Values = tea.Data[0].Embedding
                .Select(d => (float)d).ToArray()
            });

    embeddingCollection.Add(
        new Embedding
            {
                Text = "car",
                Values = chai.Data[0].Embedding
                .Select(d => (float)d).ToArray()
            });

    embeddingCollection.Add(
        new Embedding
            {
                Text = "tire",
                Values = chai.Data[0].Embedding
                .Select(d => (float)d).ToArray()
            });
}
```

Add the following code to the **`@code`** section. It will implement the **`CallService`** method that will be called when the Search button is clicked.

*Code Listing 49*

```
// Call the OpenAI service to calculate similarities between embeddings.
async Task CallService()
{
    // Reset the similarities list.
    similarities = new List<(string?, float)>();
    // Get all embeddings from the collection.
    var embeddingsInCollection = embeddingCollection.GetAll();

    // Create an instance of the OpenAI client.
    var api = new OpenAIClient(new OpenAIAuthentication(ApiKey,
Organization));
    // Get the model details.
    var model =
    await api.ModelsEndpoint.GetModelDetailsAsync("text-embedding-ada-
002");

    // Create an embedding for the user's input prompt.
    var EmbeddingQueryResponse =
    await api.EmbeddingsEndpoint.CreateEmbeddingAsync(prompt, model);

    // Calculate the similarity between the prompt's
    // embedding and each existing embedding.
    foreach (var embedding in embeddingsInCollection)
    {
        if (embedding.Values != null)
        {
            var similarity =
            CosineSimilarity(
                EmbeddingQueryResponse.Data[0].Embedding
                .Select(d => (float)d).ToArray(),
                embedding.Values);

            similarities.Add((embedding.Text, similarity));
        }
    }

    // Sort the results by similarity in descending order.
    similarities.Sort((a, b) => b.Item2.CompareTo(a.Item2));
}
```

This method calls the **`CosineSimilarity`** method. Add the following code to implement that method.

Code Listing 50

```
private float CosineSimilarity(float[] vector1, float[] vector2)
{
    // Initialize variables for dot product and
    // magnitudes of the vectors.
    float dotProduct = 0;
    float magnitude1 = 0;
    float magnitude2 = 0;

    // Iterate through the vectors and calculate
    // the dot product and magnitudes.
    for (int i = 0; i < vector1?.Length; i++)
    {
        // Calculate dot product.
        dotProduct += vector1[i] * vector2[i];

        // Calculate squared magnitude of vector1.
        magnitude1 += vector1[i] * vector1[i];

        // Calculate squared magnitude of vector2.
        magnitude2 += vector2[i] * vector2[i];
    }

    // Take the square root of the squared magnitudes
    // to obtain actual magnitudes.
    magnitude1 = (float)Math.Sqrt(magnitude1);
    magnitude2 = (float)Math.Sqrt(magnitude2);

    // Calculate and return cosine similarity by dividing
    // dot product by the product of magnitudes.
    return dotProduct / (magnitude1 * magnitude2);
}
```

Add a link to the control in the NavMenu.razor control and hit **F5** to run the application.

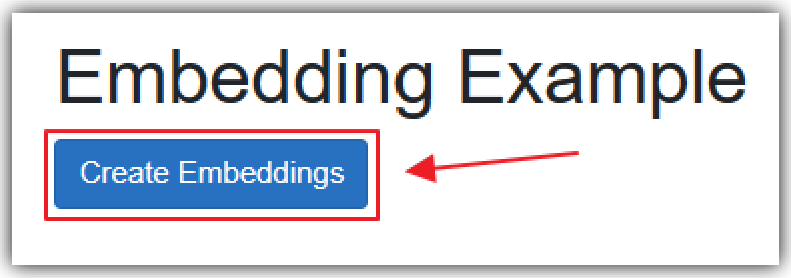
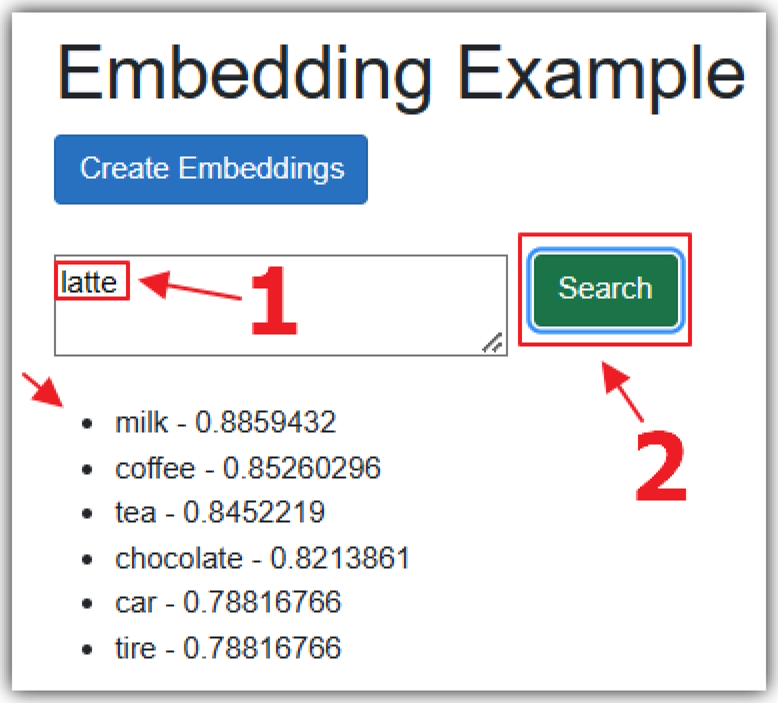


*Figure 56: Create Embeddings*

Navigate to the Razor page and click **Create Embeddings**.



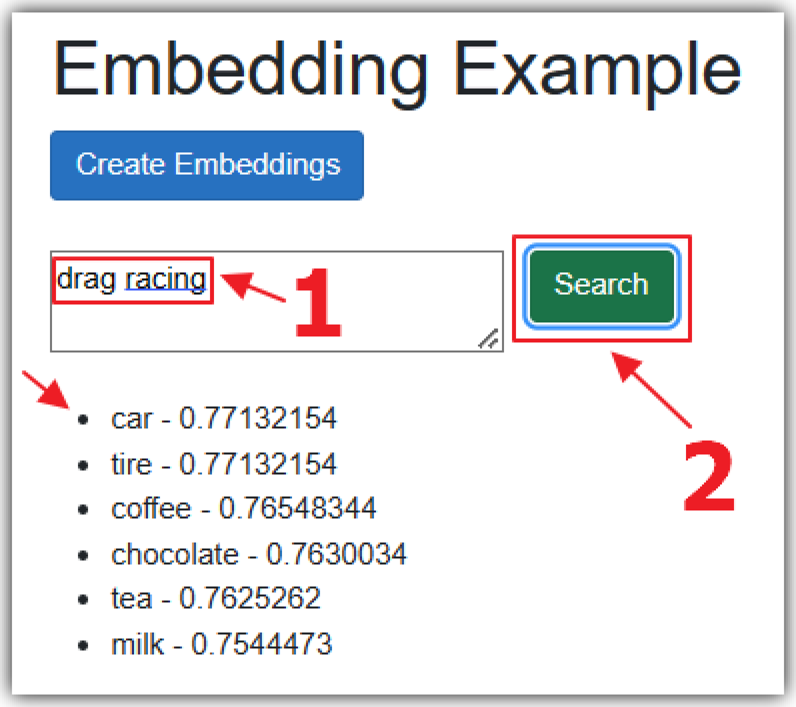
- milk - 0.8859432
- coffee - 0.85260296
- tea - 0.8452219
- chocolate - 0.8213861
- car - 0.78816766
- tire - 0.78816766

*Figure 57: Search Latte*

Enter **latte** for the prompt and click **Search**. The **`CreateEmbeddingAsync`** method will be called to create an embedding of the prompt.

The **`CosineSimilarity`** will be called to compare that embedding to the collection of embeddings created in the first step. Those embeddings, and the calculation of their similarity (in order of closest to furthest), are then presented in a list.

Note that milk, coffee, and tea are listed as the most similar words to latte.



- car - 0.77132154
- tire - 0.77132154
- coffee - 0.76548344
- chocolate - 0.7630034
- tea - 0.7625262
- milk - 0.7544473

*Figure 58: Search Drag Racing*

Change the prompt to **drag racing** and click **Search**. Note that car and tire are now listed as the most similar words.

***Note: The documentation for the Embeddings API is located here.***

# Chapter 8  Moderation and Best Practices

*Figure 59: Abstract Image of Robots Processing Data*

OpenAI offers powerful APIs like DALL-E, Whisper, and ChatGPT for developers to build AI applications in domains such as image generation, speech recognition, and natural language processing. It's important to follow moderation and best practices for the ethical and responsible use of the technology.

This includes screening generated content to prevent inappropriate or harmful materials and implementing content filtering. OpenAI encourages developers to follow guidelines for fairness, accountability, and transparency. User privacy and data security should also be considered by handling sensitive information carefully and following data protection regulations.

Developers should stay up-to-date with API documentation, rate limits, and pricing structures for optimal performance. By following these best practices, developers can fully utilize OpenAI APIs while maintaining a safe environment for users.

## Moderation

Moderation is the process of monitoring and filtering content generated by OpenAI's AI models like DALL-E, Whisper, and ChatGPT to ensure it meets ethical standards, legal requirements, and community guidelines. These models can generate text and images, or transcribe speech, and may produce offensive or harmful content.

Moderation prevents such content from being displayed to users or integrated into applications built using OpenAI's APIs. Developers should implement content filtering mechanisms to detect and remove unsuitable content before it reaches end users. This helps maintain a safe environment for users while leveraging the capabilities of OpenAI's AI models.

***Note: OpenAI provides an API that allows developers to submit content to determine if it violates OpenAI's content policies. The documentation for the Moderations API is located here.***

## Best practices

Here are some recommended best practices for utilizing OpenAI's APIs:

- Employ the most recent and advanced models.
- With prompts, specify the desired output format through examples or templates.
- Conduct adversarial testing to ensure your application can handle malicious inputs and behaviors effectively.
- Restrict user input and limit output tokens to prevent prompt injection and misuse.
- Enable users to report issues and offer feedback on your application.

## Closing thoughts

As you've seen throughout this book, OpenAI's powerful models and Microsoft Blazor come together to create a compelling platform for building intelligent, interactive, and modern applications. From simple completions and chat interfaces to function calling, image generation, transcription, embeddings, and moderation, each chapter has shown you how to bridge theory with practical implementation.

Whether you are experimenting in the Playground, fine-tuning a model, or deploying a full Blazor application, the common thread is clear: with thoughtful prompts, well-designed code, and an understanding of best practices, you can create solutions that are both technically robust and human-centered. My hope is that this book inspires you to continue exploring, experimenting, and building with these tools—pushing the boundaries of what's possible and shaping the next generation of AI-powered applications.