Table of Contents

Quick Start

Please note that the code provided in this page is *purely* for learning purposes and is far from perfect. Remember to null-check all responses!

Setup

Add an instance of `Uralstech.UGemini.GeminiManager` to your scene, and set it up with your Gemini API key. You can get your API key from *here*.

GeminiManager

There are only three methods in `GeminiManager`:

Method	What it does
`SetApiKey`	Sets the Gemini API key through code
`Request`	Computes a request on the Gemini API
`StreamRequest`*	Computes a streaming request on the Gemini API

*Requires *Utilities.Async*.

All computations on the Gemini API are done through `GeminiManager.Request`, `GeminiManager.StreamRequest` and their variants.

In this page, the fields, properties and methods of each type will not be explained. Every type has been fully documented in code, so please check the code docstrings or reference documentation to learn more about each type.

Beta API

`GeminiManager` supports both the `v1` and `v1beta` Gemini API versions. As a lot of features are still unsupported in the main `v1` API, you may need to use the Beta API. You can set the `useBetaApi` boolean parameter in the request's constructor to do so.

Models

`Uralstech.UGemini.Models.GeminiModel` has four static model IDs:

- `Gemini1_5Flash`
- `Gemini1_5Pro`
- `Gemini1_0Pro`
- `Gemini1_0ProVision`

    - Gemini 1.0 Pro Vision is deprecated. Use Use 1.5 Flash (`Gemini1_5Flash`) or 1.5 Pro (`Gemini1_5Pro`) instead.
- `TextEmbedding004`
- `Aqa`

You can provide these to the `model` parameter in the constructors for model-related requests. UGemini can also implicitly convert `string` model IDs to `GeminiModelId` objects.

QuickStart: Multi-turn Chat Request

This is a simple script that maintains the user's chat history with Gemini.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Generation.Chat;

private List<GeminiContent> _chatHistory = new();

private async Task<string> OnChat(string text)
{
    _chatHistory.Add(GeminiContent.GetContent(text, GeminiRole.User));
    GeminiChatResponse response = await GeminiManager.Instance.Request<GeminiChatResponse>(
        new GeminiChatRequest(GeminiModel.Gemini1_5Flash)
        {
            Contents = _chatHistory.ToArray(),
        }
    );

    _chatHistory.Add(response.Candidates[0].Content);
    return response.Parts[0].Text;
}
```

Here, we simply have a list of `GeminiContent` objects, which tracks the messages of the conversation. Every time `OnChat` is called, the user's request and the model's reply are added the the list.

What Next?

You can check out more `GeminiChatRequest` Features like Streaming Requests, Function Calling, Code Execution and more. You can also read the documentation for all Gemini API endpoints that UGemini supports to learn more about features like File Uploads, Content Caching and Fine Tuning.

Samples

For full-fledged examples of the features of this package, check out the samples included in the package:

Mult-turn Chat

A sample scene showing a multi-turn chat system. ***GitHub Source***

Function Calling

A sample scene showing a function calling system. ***GitHub Source***

Streaming Generated Content

A sample showing a system which streams Gemini's responses, including function calls. ***GitHub Source***

Question Answering

A sample scene with a system where Gemini answers questions based only on the given context. ***GitHub Source***

Prompting with File API

A sample scene with a system to create, delete, retrieve, list and prompt Gemini with files stored in the File/Media API endpoints. ***GitHub Source***

JSON Response

A sample scene showing a system where Gemini responds in a specified JSON format. ***GitHub Source***

List and Get Model Metadata

A sample scene with a system to list, get and chat with models using the models.get and models.list endpoints. ***GitHub Source***

Token Counting

A sample scene showing a token counting system using the `countTokens` endpoint. ***GitHub Source***

GeminiChatRequest Features

`GeminiAnswerRequest` also shares some of these features.

Streaming Responses

`GeminiChatRequest` allows you to stream Gemini's response in real-time. You can do so by using `GeminiManager.StreamRequest` and utilizing the callback in `GeminiChatRequest`.

You can even stream function calls! Check out the `Streaming Generated Content` sample included in the package.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Generation.Chat;

[SerializeField] private Text _chatResponse;

private async Task<string> OnChat(string text)
{
    GeminiChatResponse response = await GeminiManager.Instance.StreamRequest(new
GeminiChatRequest(GeminiModel.Gemini1_5Flash)
    {
        Contents = new GeminiContent[]
        {
            GeminiContent.GetContent(text, GeminiRole.User),
        },

        OnPartialResponseReceived = streamedResponse =>
        {
            _chatResponse.text = streamedResponse.Parts[0].Text;
            return Task.CompletedTask;
        }
    });

    return response.Parts[0].Text;
}
```

If you do not want to use the callback, you can let the `StreamRequest` task run in the background, and access the streamed data from the `GeminiChatRequest.StreamedResponse` property.

Adding Media Content to Requests

`GeminiContent.Parts` contains the actual contents of each chat request and response. You can add media content to the `Parts` array, but you must only have one type of data in each part, like one part of text, one part of an image, and so on. The following samples shows data being read from a file and into a `GeminiContent` object.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models.Content;

private async Task<GeminiContent> GetFileContent(string filePath,
GeminiContentType contentType)
{
    byte[] data;
    try
    {
        data = await File.ReadAllBytesAsync(filePath);
    }
    catch (SystemException exception)
    {
        Debug.LogError($"Failed to load file: {exception.Message}");
        return null;
    }

    return new GeminiContent()
    {
        Parts = new GeminiContentPart[]
        {
            new GeminiContentPart()
            {
                Text = "What's in this file?"
            },
            new GeminiContentPart()
            {
                InlineData = new GeminiContentBlob()
                {
                    MimeType = contentType,
                    Data = Convert.ToBase64String(data)
                }
            }
        }
    };
}
```

Now, the `GeminiContent` returned by the method can be fed into a chat request!

Utility Methods

`GeminiContent` and `GeminiContentBlob` also contain static utility methods to help create them from Unity types like `AudioClip` or `Texture2D`:

- `GeminiContent.GetContent`
    - Can convert `string` messages, `Texture2D` images, `AudioClip`* audio and `GeminiFile` data to `GeminiContent` objects.
- `GeminiContentBlob.GetContentBlob`
    - Can convert `Texture2D` images and `AudioClip`* audio to `GeminiContentBlob` objects.

*Requires *Utilities.Encoding.Wav*.

Function Calling

First, we have to setup our tools and define our function schemas.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Generation.Chat;
using Uralstech.UGemini.Models.Generation.Schema;
using Uralstech.UGemini.Models.Generation.Tools;
using Uralstech.UGemini.Models.Generation.Tools.Declaration;

private GeminiTool _geminiFunctions = new GeminiTool()
{
    FunctionDeclarations = new GeminiFunctionDeclaration[]
    {
        new GeminiFunctionDeclaration()
        {
            Name = "printToConsole",
            Description = "Print text to the user's console.",
            Parameters = new GeminiSchema()
            {
                Type = GeminiSchemaDataType.Object,
                Properties = new Dictionary<string, GeminiSchema>()
                {
                    {
                        "text", new GeminiSchema()
                        {
                            Type = GeminiSchemaDataType.String,
                            Description = "The text to print. e.g. \"Hello, World!\"",
                            Nullable = false,
```

```
                        }
                    },
                },
                Required = new string[] { "text" },
            }
        },

        new GeminiFunctionDeclaration()
        {
            Name = "changeTextColor",
            Description = "Change the color of the text.",
            Parameters = new GeminiSchema()
            {
                Type = GeminiSchemaDataType.Object,
                Properties = new Dictionary<string, GeminiSchema>()
                {
                    {
                        "color", new GeminiSchema()
                        {
                            Type = GeminiSchemaDataType.String,
                            Description = "The color to set. e.g. \"BLUE\"",
                            Format = GeminiSchemaDataFormat.Enum,
                            Enum = new string[]
                            {
                                "RED",
                                "GREEN",
                                "BLUE",
                                "WHITE",
                            },
                            Nullable = false,
                        }
                    },
                },
                Required = new string[] { "color" },
            }
        }
    },
};
```

To use Gemini Tools, we need to declare each tool to use. So, we have created a declaration for function calling. Each *tool* must be declared separately but every *function* must be declared in a single *tool* declaration.

For each function, we need a declaration with a name and description. The parameters are an object of type `GeminiSchema`, which defines the schema of each of the parameters. The type is of

`GeminiSchemaDataType.Object`, and contains the dictionary of parameter schemas.

The keys of the dictionary should be the parameter name, and the values should be `GeminiSchema` objects which define the type, description, format, etc. of the parameter.

Finally, we have the `Required` property which tells Gemini which fields are absolutely required in each call. Now, we can move on to the chat.

```
[SerializeField] private Text _chatResponse;

private async Task<string> OnChat(string text)
{
    List<GeminiContent> contents = new()
    {
        GeminiContent.GetContent(text, GeminiRole.User),
    };

    GeminiChatResponse response;
    GeminiFunctionCall functionCall;
    string responseText = string.Empty;
    do
    {
        response = await GeminiManager.Instance.Request<GeminiChatResponse>(new
GeminiChatRequest(GeminiModel.Gemini1_5Flash, true)
        {
            Contents = contents.ToArray(),
            Tools = new GeminiTool[] { _geminiFunctions },
            ToolConfig =
GeminiToolConfiguration.GetConfiguration(GeminiFunctionCallingMode.Auto),
        });

        // Don't forget to do this! If the function call is not added to the chat
        // history, Gemini will throw an error when receiving the response!
        contents.Add(response.Candidates[0].Content);

        responseText = Array.Find(response.Parts, part =>
!string.IsNullOrEmpty(part.Text))?.Text;
        GeminiContentPart[] allFunctionCalls = Array.FindAll(response.Parts, part =>
part.FunctionCall != null);

        functionCall = null;
        for (int i = 0; i < allFunctionCalls.Length; i++)
        {
            functionCall = allFunctionCalls[i].FunctionCall;
            JObject functionResponse = null;
```

```
            switch (functionCall.Name)
            {
                case "printToConsole":
                    Debug.Log(functionCall.Arguments["text"].ToObject<string>());
                    break;

                case "changeTextColor":
                    if (!TryChangeTextColor(functionCall.Arguments["color"].ToObject<string>
()))
                    {
                        functionResponse = new JObject()
                        {
                            ["result"] = "Unknown color."
                        };
                    }

                    break;

                default:
                    functionResponse = new JObject()
                    {
                        ["result"] = "Sorry, but that function does not exist."
                    };

                    break;
            }

            contents.Add(GeminiContent.GetContent(functionCall.GetResponse(functionResponse
?? new JObject()
            {
                ["result"] = "Completed executing function successfully."
            })));
        }
    } while (functionCall != null);

    _chatResponse.text = responseText;
    return responseText;
}

private bool TryChangeTextColor(string color)
{
    switch (color)
    {
        case "RED":
            _chatResponse.color = Color.red; break;
```

```
            case "GREEN":
                _chatResponse.color = Color.green; break;

            case "BLUE":
                _chatResponse.color = Color.blue; break;

            case "WHITE":
                _chatResponse.color = Color.white; break;

            default:
                return false;
        }

        Debug.Log("Changed text color!");
        return true;
    }
```

Here, we are going through each response, checking if a function was called, and calling the requested function.

The response is a JSON object, which is optional, but it is recommended to include. Note the use of `GeminiToolConfiguration.GetConfiguration`, which is a utility method to create a `GeminiToolConfiguration` with the given `GeminiFunctionCallingMode`. `GeminiFunctionCallingMode.Any` means Gemini will always call at least one function in each *request*, `Auto` means the model will call the functions when it thinks it needs to, and `None` means no functions can be called.

After the function is called, we respond by adding the calls and responses to the history. We use the `GetResponse` utility method to get a `GeminiFunctionResponse` object with the response JSON.

Function calling is, as of writing, only available in the Beta API.

Code Execution

Code execution is also a Tool, so it is similar to function calling:

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Generation.Chat;
using Uralstech.UGemini.Models.Generation.Tools.Declaration;

private GeminiTool _geminiCodeExecution = new GeminiTool()
{
    CodeExecution = new GeminiCodeExecution()
};
```

```
[SerializeField] private Text _chatResponse;

private async Task<string> OnChat(string text)
{
    List<GeminiContent> contents = new()
    {
        GeminiContent.GetContent(text, GeminiRole.User),
    };

    GeminiChatResponse response = await GeminiManager.Instance.Request<GeminiChatResponse>
(new GeminiChatRequest(GeminiModel.Gemini1_5Flash, true)
    {
        Contents = contents.ToArray(),
        Tools = new GeminiTool[] { _geminiCodeExecution },
    });

    string responseText = string.Join(", ", Array.ConvertAll(response.Parts, part => $"
(Text={part.Text}, Code={part.ExecutableCode?.Code}, ExecutionResult=
{part.CodeExecutionResult?.Output})"));

    _chatResponse.text = responseText;
    return responseText;
}
```

That's it! Now, when code execution is used, the response should be something like this:

```
> Make a simple python program to print hello world and use code execution for that.
Result: (Text=, Code=, ExecutionResult=), (Text=, Code=print("Hello world!"),
ExecutionResult=), (Text=, Code=, ExecutionResult=Hello world!),
(Text=I have created a simple Python program that prints "Hello world!".  I used the
`print()` function to achieve this. The code was executed
using the `tool_code` block., Code=, ExecutionResult=)
```

Code execution is also, as of writing, only available in the Beta API.

JSON Response Mode

In JSON mode, Gemini will always respond in the specified JSON response schema.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Generation;
```

```
using Uralstech.UGemini.Models.Generation.Chat;
using Uralstech.UGemini.Models.Generation.Schema;

private async Task<string> OnChat(string text)
{
    // Note: It seems GeminiModel.Gemini1_5Flash is not very good at JSON.
    GeminiChatResponse response = await GeminiManager.Instance.Request<GeminiChatResponse>
(new GeminiChatRequest(GeminiModel.Gemini1_5Pro, true)
    {
        Contents = new GeminiContent[]
        {
            GeminiContent.GetContent(text, GeminiRole.User),
        },
        SystemInstruction = GeminiContent.GetContent("You are a helpful math teacher who
teacher their students mathematics in the most helpful way possible."),
        GenerationConfig = new GeminiGenerationConfiguration()
        {
            ResponseMimeType = GeminiResponseType.Json,
            ResponseSchema = new GeminiSchema()
            {
                Type = GeminiSchemaDataType.Array,
                Description = "A list of mathematical expressions.",
                Items = new GeminiSchema()
                {
                    Type = GeminiSchemaDataType.Object,
                    Properties = new Dictionary<string, GeminiSchema>()
                    {
                        {
                            "expression", new GeminiSchema()
                            {
                                Type = GeminiSchemaDataType.String,
                            }
                        },
                        {
                            "explanation", new GeminiSchema()
                            {
                                Type = GeminiSchemaDataType.String,
                            }
                        },
                    },
                    Required = new string[] { "expression", "explanation", },
                },
            },
        }
    });
```

```
    return response.Parts[0].Text;
}
```

Here, we used a schema for an array of objects, which contain two parameters: `expression` and `explanation`. We have told Gemini to split the response into the parameters, where a mathematical expression and its explanation is given.

The `GeminiSchema` object is the same type used for function calling.

JSON mode is also only available in the Beta API.

All Supported Endpoints

CachedContents (Beta API)

> Context caching allows you to save and reuse precomputed input tokens that you wish to use repeatedly, for example when asking different questions about the same media file.

Create

> Creates CachedContent resource.

```
private async Task<GeminiCachedContent> RunCreateCachedContentRequest()
{
    // Content must be at least 32,768 tokens.
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 1800; i++)
        sb.Append("(*)#$*OIJIR$U(IJT^(U$*I%O$#@");


    return await GeminiManager.Instance.Request<GeminiCachedContent>(new
GeminiCachedContentCreateRequest(new GeminiCachedContentCreationData
    {
        Contents = new[]
        {
            GeminiContent.GetContent(sb.ToString(), GeminiRole.User),
        },
        ExpireTime = DateTime.UtcNow.AddDays(1),
        Model = "Gemini-1.5-flash-001", // Make sure the model you use supports caching!
    }));
}
```

See GeminiCachedContent and GeminiCachedContentCreateRequest for more details.

Delete

> Deletes CachedContent resource.

```
private async Task RunDeleteCachedContentRequest(GeminiCachedContent content)
{
    Debug.Log("Deleting cached content...");
    await GeminiManager.Instance.Request(new
```

```
GeminiCachedContentDeleteRequest(content.Name));
    Debug.Log("Content deleted.");
}
```

See GeminiCachedContentDeleteRequest for more details.

Get

> Reads CachedContent resource.

```
private async Task<GeminiCachedContent> RunGetCachedContentRequest(string contentName)
{
    return await GeminiManager.Instance.Request<GeminiCachedContent>(new
GeminiCachedContentGetRequest(contentName));
}
```

See GeminiCachedContent and GeminiCachedContentGetRequest for more details.

List

> Lists CachedContents.

```
private async Task<GeminiCachedContent[]> RunListCachedContentRequest()
{
    GeminiCachedContentListResponse response = await
GeminiManager.Instance.Request<GeminiCachedContentListResponse>(new
GeminiCachedContentListRequest());
    return response.CachedContents;
}
```

See GeminiCachedContentListResponse and GeminiCachedContentListRequest for more details.

Patch

> Updates CachedContent resource (only expiration is updatable).

```
private async Task<GeminiCachedContent> RunPatchCachedContentRequest(string contentName)
{
    return await GeminiManager.Instance.Request<GeminiCachedContent>(new
GeminiCachedContentPatchRequest(new GeminiCachedContentPatchData
```

```
    {
        ExpireTime = DateTime.UtcNow.AddYears(1),
    }, contentName));
}
```

See GeminiCachedContent and GeminiCachedContentPatchRequest for more details.

Models

> The Models endpoint contains methods that allow you to access and inference Gemini models.

Get

> Gets information about a specific Model such as its version number, token limits, parameters and other metadata.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;

private async Task<GeminiModel> RunGetModelRequest(string modelId)
{
    return await GeminiManager.Instance.Request<GeminiModel>(new
GeminiModelGetRequest(modelId));
}
```

See GeminiModel and GeminiModelGetRequest for more details.

Newer models will not be recognized by the request if you're not using the Beta API.

List

> Lists the Models available through the Gemini API.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;

private async Task<GeminiModel[]> RunListModelsRequest(int maxModels = 50, string pageToken
= null)
{
    GeminiModelListResponse response = await
GeminiManager.Instance.Request<GeminiModelListResponse>(new GeminiModelListRequest()
```

```
    {
        MaxResponseModels = maxModels,
        PageToken = string.IsNullOrWhiteSpace(pageToken) ? string.Empty : pageToken,
    });

    return response?.Models;
}
```

See GeminiModelListResponse and GeminiModelListRequest for more details.

Newer models will not be recognized by the request if you're not using the Beta API.

EmbedContent

> Generates a text embedding vector from the input Content using the specified Gemini Embedding model.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Embedding;

private async void RunEmbedContentRequest()
{
    Debug.Log("Running embedding request.");

    GeminiEmbedContentResponse response = await
GeminiManager.Instance.Request<GeminiEmbedContentResponse>(
        new GeminiEmbedContentRequest(GeminiModel.TextEmbedding004)
        {
            Content = GeminiContent.GetContent("Hello! How are you?"),
        }
    );

    Debug.Log($"Embedding values: {string.Join(", ", response.Embedding.Values)}");
}
```

See GeminiEmbedContentResponse and GeminiEmbedContentRequest for more details.

BatchEmbedContents

> Generates multiple embedding vectors from the input Content which consists of a batch of strings represented as EmbedContentRequest objects.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Embedding;

private async void RunBatchEmbedContentRequest()
{
    Debug.Log("Running batch embedding request.");

    // Make sure the model used for the batch request is the same for all included requests.
    GeminiBatchEmbedContentResponse response = await
GeminiManager.Instance.Request<GeminiBatchEmbedContentResponse>(
        new GeminiBatchEmbedContentRequest(GeminiModel.TextEmbedding004)
        {
            Requests = new GeminiEmbedContentRequest[]
            {
                new GeminiEmbedContentRequest(GeminiModel.TextEmbedding004)
                {
                    Content = GeminiContent.GetContent("Hello! How are you?"),
                },
                new GeminiEmbedContentRequest(GeminiModel.TextEmbedding004)
                {
                    Content = GeminiContent.GetContent("Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
magna aliqua."),
                }
            }
        }
    );

    foreach (GeminiContentEmbedding embedding in response.Embeddings)
        Debug.Log($"Embedding values: {string.Join(", ", embedding.Values)}");
}
```

See GeminiBatchEmbedContentResponse and GeminiBatchEmbedContentRequest for more details.

GenerateContent

> Generates a model response given an input GenerateContentRequest.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
```

```
using Uralstech.UGemini.Models.Generation.Chat;

private async void RunChatRequest()
{
    Debug.Log("Running chat request.");

    GeminiChatResponse response = await GeminiManager.Instance.Request<GeminiChatResponse>(
        new GeminiChatRequest(GeminiModel.Gemini1_5Flash)
        {
            Contents = new GeminiContent[]
            {
                GeminiContent.GetContent("What's up?")
            },
        }
    );

    Debug.Log($"Gemini's response: {response.Parts[^1].Text}");
}
```

See GeminiChatResponse and GeminiChatRequest for more details.

StreamGenerateContent

> Generates a streamed response from the model given an input GenerateContentRequest.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Generation.Chat;

private async void RunStreamingChatRequest()
{
    Debug.Log("Running streamed chat request.");

    GeminiChatResponse response = await GeminiManager.Instance.StreamRequest(
        new GeminiChatRequest(GeminiModel.Gemini1_5Flash)
        {
            Contents = new GeminiContent[]
            {
                GeminiContent.GetContent("What's up? Tell me a story about airplanes.")
            },
            OnPartialResponseReceived = partialResponse =>
            {
                if (partialResponse.Candidates == null || partialResponse.Candidates.Length
```

```
< 0)
                    return Task.CompletedTask;

                Debug.Log($"Gemini's partial response: {partialResponse.Parts[^1].Text}");
                return Task.CompletedTask;
            }
        }
    );

    Debug.Log($"Gemini's final response: {response.Parts[^1].Text}");
}
```

See GeminiChatResponse and GeminiChatRequest for more details.

GenerateAnswer (Beta API)

> Generates a grounded answer from the model given an input GenerateAnswerRequest.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Content.Attribution;
using Uralstech.UGemini.Models.Generation.QuestionAnswering;
using Uralstech.UGemini.Models.Generation.QuestionAnswering.Grounding;

private async void RunQuestionAnsweringRequest()
{
    Debug.Log("Running Q/A request.");

    GeminiAnswerResponse response = await
GeminiManager.Instance.Request<GeminiAnswerResponse>(
        new GeminiAnswerRequest(GeminiModel.Aqa)
        {
            Contents = new GeminiContent[]
            {
                GeminiContent.GetContent("What is ezr²?")
            },

            InlinePassages = new GeminiGroundingPassages()
            {
                Passages = new GeminiGroundingPassage[]
                {
                    new GeminiGroundingPassage()
                    {
```

```
                        Id = "ezrSquaredContext",
                        Content = GeminiContent.GetContent(
                            "ezr² is an easy to learn and practical interpreted programming
language for beginners and experts alike made in C#." +
                            "The latest version of ezr² RE has been released! ezr² RE, or
REwrite, is the project's initiative to rewrite ezr². " +
                            "The latest working version of ezr² RE has many more features
than the latest version of ezr²! But, it is still in " +
                            "development, has some essential features missing. Like the
include expression, or any built-in object methods like " +
                            "\"a string\".length or [\"a\", \"list\"].insert. If you want to
help in testing it out and fixing bugs, feel free to" +
                            " download the latest version of ezr² RE from the ezr² GitHub
releases page and compiling it using the .NET SDK and/or" +
                            " Visual Studio. The GitHub releases page is:
https://github.com/Uralstech/ezrSquared/releases."
                        )
                    }
                }
            },

            AnswerStyle = GeminiAnswerStyle.Verbose,
        }
    );

    if (response.Answer.Content != null)
        Debug.Log($"Gemini's answer: {response.Answer.Content.Parts[^1].Text}");

    foreach (GeminiGroundingAttribution attribution in
response.Answer.GroundingAttributions)
        Debug.Log($"Attribution ID: {attribution.SourceId.GroundingPassage.PassageId} |
Content: {attribution.Content.Parts[^1].Text}");
}
```

See GeminiAnswerResponse and GeminiAnswerRequest for more details.

CountTokens

> Runs a model's tokenizer on input Content and returns the token count.

```
using Uralstech.UGemini;
using Uralstech.UGemini.Models;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.CountTokens;
```

```
private async void RunTokenCountRequest()
{
    Debug.Log("Running token counting request.");

    GeminiTokenCountResponse response = await
GeminiManager.Instance.Request<GeminiTokenCountResponse>(
        new GeminiTokenCountRequest(GeminiModel.Gemini1_5Flash)
        {
            Contents = new GeminiContent[]
            {
                GeminiContent.GetContent("Hello! How are you?"),
            }
        }
    );

    Debug.Log($"Tokens: {response.TotalTokens}");
}
```

See GeminiTokenCountResponse and GeminiTokenCountRequest for more details.

TunedModels (Unstable)

> The TunedModels endpoint contains methods that allow you to access and inference fine-tuned Gemini models.

GenerateContent

> Generates a model response given an input GenerateContentRequest.

```
// This is untested code.

using Uralstech.UGemini;
using Uralstech.UGemini.Models.Content;
using Uralstech.UGemini.Models.Generation.Chat;

private async void RunTunedModelChatRequest()
{
    Debug.Log("Running chat request on tuned model.");

    GeminiChatResponse response = await GeminiManager.Instance.Request<GeminiChatResponse>(
        new GeminiChatRequest("tunedModels/modelname")
        {
```

```
            Contents = new GeminiContent[]
            {
                GeminiContent.GetContent("What's up?")
            },
        }
    );

    Debug.Log($"Tuned Gemini's response: {response.Parts[^1].Text}");
}
```

See GeminiChatResponse and GeminiChatRequest for more details.

Files (Beta API)

> The Gemini File API can be used to store data on the cloud for future prompting with the Gemini models.

Delete

> Deletes the File.

```
using Uralstech.UGemini;
using Uralstech.UGemini.FileAPI;

private async void RunDeleteFileRequest(string fileId)
{
    Debug.Log("Deleting file...");
    await GeminiManager.Instance.Request(new GeminiFileDeleteRequest(fileId));
    Debug.Log("File deleted.");
}
```

See GeminiFileDeleteRequest for more details.

Get

> Gets the metadata for the given File.

```
using Uralstech.UGemini;
using Uralstech.UGemini.FileAPI;

private async Task<GeminiFile> RunGetFileRequest(string fileId)
```

```
{
    return await GeminiManager.Instance.Request<GeminiFile>
(new GeminiFileGetRequest(fileId));
}
```

See GeminiFile and GeminiFileGetRequest for more details.

List

> Lists the metadata for Files owned by the requesting project.

```
using Uralstech.UGemini;
using Uralstech.UGemini.FileAPI;

private async Task<GeminiFile[]> RunListFilesRequest(int maxFiles = 10, string pageToken
= null)
{
    GeminiFileListResponse response = await
GeminiManager.Instance.Request<GeminiFileListResponse>(new GeminiFileListRequest()
    {
        MaxResponseFiles = maxFiles,
        PageToken = string.IsNullOrWhiteSpace(pageToken) ? string.Empty : pageToken,
    });

    return response?.Files;
}
```

See GeminiFileListResponse and GeminiFileListRequest for more details.

Media (Beta API)

> The Gemini File API can be used to store data on the cloud for future prompting with the Gemini models.

Upload

> Creates a File.

```
using Uralstech.UGemini;
using Uralstech.UGemini.FileAPI;
```

```
private async Task<GeminiFile> RunUploadFileRequest(string text)
{
    GeminiFileUploadResponse response = await
GeminiManager.Instance.Request<GeminiFileUploadResponse>(new
GeminiFileUploadRequest(GeminiContentType.TextPlain.MimeType())
    {
        File = new GeminiFileUploadMetaData()
        {
            DisplayName = "I'm a File",
        },
        RawData = Encoding.UTF8.GetBytes(text)
    });

    return response.File;
}
```

See GeminiFileUploadResponse and GeminiFileUploadRequest for more details.