Documentation

Text

Chat Completion Models

The Groq Chat Completions API processes a series of messages and generates output responses. These models can perform multi-turn discussions or tasks that require only one interaction.

For details about the parameters, visit the reference page.

JSON mode *(beta)*

JSON mode is a beta feature that guarantees all chat completions are valid JSON.
Usage:

1. Set `"response_format": {"type": "json_object"}` in your chat completion request
2. Add a description of the desired JSON structure within the system prompt (see below for example system prompts)

Recommendations for best beta results:

- Mixtral performs best at generating JSON, followed by Gemma, then Llama
- Use pretty-printed JSON instead of compact JSON
- Keep prompts concise

Beta Limitations:

- Does not support streaming
- Does not support stop sequences

Error Code:

- Groq will return a 400 error with an error code of `json_validate_failed` if JSON generation fails.

Example system prompts:

```
You are a legal advisor who summarizes documents in JSON
```

```
You are a data analyst API capable of sentiment analysis that responds in JSON.  The JSON schema should ir
{
  "sentiment_analysis": {
    "sentiment": "string (positive, negative, neutral)",
    "confidence_score": "number (0-1)"
    # Include additional fields as required
  }
}
```

Generating Chat Completions with groq SDK

Code Overview

Python JavaScript

```
pip install groq
```

Performing a basic Chat Completion

```
from groq import Groq

client = Groq()

chat_completion = client.chat.completions.create(
    #
    # Required parameters
    #
    messages=[
        # Set an optional system message. This sets the behavior of the
        # assistant and can be used to provide specific instructions for
        # how it should behave throughout the conversation.
        {
            "role": "system",
            "content": "you are a helpful assistant."
        },
        # Set a user message for the assistant to respond to.
        {
            "role": "user",
            "content": "Explain the importance of fast language models",
        }
    ],

    # The language model which will generate the completion.
    model="llama-3.3-70b-versatile",

    #
    # Optional parameters
    #

    # Controls randomness: lowering results in less random completions.
    # As the temperature approaches zero, the model will become deterministic
    # and repetitive.
    temperature=0.5,

    # The maximum number of tokens to generate. Requests can use up to
    # 32,768 tokens shared between prompt and completion.
    max_completion_tokens=1024,

    # Controls diversity via nucleus sampling: 0.5 means half of all
    # likelihood-weighted options are considered.
    top_p=1,

    # A stop sequence is a predefined or user-specified text string that
    # signals an AI to stop generating content, ensuring its responses
    # remain focused and concise. Examples include punctuation marks and
    # markers like "[end]".
    stop=None,

    # If set, partial message will be sent.
    stream=False,
)
```

Streaming a Chat Completion

To stream a completion, simply set the parameter `stream=True`. Then the completion function will return an iterator of completion deltas rather than a single, full completion.

```
from groq import Groq

client = Groq()

stream = client.chat.completions.create(
    #
    # Required parameters
    #
    messages=[
        # Set an optional system message. This sets the behavior of the
        # assistant and can be used to provide specific instructions for
        # how it should behave throughout the conversation.
        {
            "role": "system",
            "content": "you are a helpful assistant."
        },
        # Set a user message for the assistant to respond to.
        {
            "role": "user",
            "content": "Explain the importance of fast language models",
        }
    ],

    # The language model which will generate the completion.
    model="llama-3.3-70b-versatile",

    #
    # Optional parameters
```

Performing a Chat Completion with a stop sequence

```
from groq import Groq

client = Groq()

chat_completion = client.chat.completions.create(
# Print the completion returned by the LLM.
print(chat_completion.choices[0].message.content)
    # Controls randomness: lowering results in less random completions.
    # As the temperature approaches zero, the model will become deterministic
    # and repetitive.
    temperature=0.5,
    # The maximum number of tokens to generate. Requests can use up to
    # 2048 tokens shared between prompt and completion.
    max_completion_tokens=1024,
    # Controls diversity via nucleus sampling: 0.5 means half of all
    # likelihood-weighted options are considered.
    top_p=1,
            "content": "Count to 10.  Your response must begin with \"1, \".  example: 1, 2, 3, ...",
    # A stop sequence is a predefined or user-specified text string that
    # signals an AI to stop generating content, ensuring its responses
    # remain focused and concise. Examples include punctuation marks and
    # markers like "[end]".
    stop=None,
    # If set, partial message deltas will be sent.
    stream=False,
)
```

Performing an Async Chat Completion

Simply use the Async client to enable asyncio

```
import asyncio

from groq import AsyncGroq


async def main():
    client = AsyncGroq()

    chat_completion = await client.chat.completions.create(
        #
        # Required parameters
        #
        messages=[
            # Set an optional system message. This sets the behavior of the
            # assistant and can be used to provide specific instructions for
            # how it should behave throughout the conversation.
            {
                "role": "system",
                "content": "you are a helpful assistant."
            },
            # Set a user message for the assistant to respond to.
            {
                "role": "user",
                "content": "Explain the importance of fast language models",
            }
        ],

        # The language model which will generate the completion.
        model="llama-3.3-70b-versatile",

        #
        # Optional parameters
        #

        # Controls randomness: lowering results in less random completions.
        # As the temperature approaches zero, the model will become deterministic
        # and repetitive.
        temperature=0.5,

        # The maximum number of tokens to generate. Requests can use up to
        # 2048 tokens shared between prompt and completion.
        max_completion_tokens=1024,

        # Controls diversity via nucleus sampling: 0.5 means half of all
        # likelihood-weighted options are considered.
        top_p=1,

        # A stop sequence is a predefined or user-specified text string that
        # signals an AI to stop generating content, ensuring its responses
        # remain focused and concise. Examples include punctuation marks and
        # markers like "[end]".
        stop=None,

        # If set, partial message deltas will be sent.
        stream=False,
    )

    # Print the completion returned by the LLM.
    print(chat_completion.choices[0].message.content)
```

Streaming an Async Chat Completion

```
import asyncio

from groq import AsyncGroq


async def main():
    client = AsyncGroq()

    stream = await client.chat.completions.create(
        #
        # Required parameters
        #
        messages=[
            # Set an optional system message. This sets the behavior of the
            # assistant and can be used to provide specific instructions for
            # how it should behave throughout the conversation.
            {
                "role": "system",
                "content": "you are a helpful assistant."
            },
            # Set a user message for the assistant to respond to.
            {
                "role": "user",
```

```
    # For this example, we will use ", 6" so that the llm stops counting at 5.
    # If multiple stop values are needed, an array of string may be passed,
    # stop=[", 6", ", six", ", Six"]
    stop=", 6",
# Print the incremental deltas returned by the LLM.
for chunk in stream:
    print(chunk.choices[0].delta.content, end="")
```

JSON Mode

```
        # Controls diversity via nucleus sampling: 0.5 means half of all
        # likelihood-weighted options are considered.
```

```python
        top_p=1,

        # A stop sequence is a predefined or user-specified text string that
        # signals an AI to stop generating content, ensuring its responses
        # remain focused and concise. Examples include punctuation marks and
        # markers like "[end]".
        stop=None,

        # If set, partial message deltas will be sent.
        stream=False,
    )

    # Print the completion returned by the LLM.
    print(chat_completion.choices[0].message.content)

asyncio.run(main())
```

```python
                "content": "Explain the importance of fast language models",
            }
        ],

        # The language model which will generate the completion.
        model="llama-3.3-70b-versatile",

        #
        # Optional parameters
        #

        # Controls randomness: lowering results in less random completions.
        # As the temperature approaches zero, the model will become
        # deterministic and repetitive.
        temperature=0.5,

        # The maximum number of tokens to generate. Requests can use up to
        # 2048 tokens shared between prompt and completion.
        max_completion_tokens=1024,

        # Controls diversity via nucleus sampling: 0.5 means half of all
        # likelihood-weighted options are considered.
        top_p=1,

        # A stop sequence is a predefined or user-specified text string that
        # signals an AI to stop generating content, ensuring its responses
        # remain focused and concise. Examples include punctuation marks and
        # markers like "[end]".
        stop=None,

        # If set, partial message deltas will be sent.
        stream=True,
    )

    # Print the incremental deltas returned by the LLM.
    async for chunk in stream:
        print(chunk.choices[0].delta.content, end="")

asyncio.run(main())
```

```python
from typing import List, Optional
import json

from pydantic import BaseModel
from groq import Groq

groq = Groq()


# Data model for LLM to generate
class Ingredient(BaseModel):
    name: str
    quantity: str
    quantity_unit: Optional[str]


class Recipe(BaseModel):
    recipe_name: str
    ingredients: List[Ingredient]
    directions: List[str]


def get_recipe(recipe_name: str) -> Recipe:
    chat_completion = groq.chat.completions.create(
        messages=[
            {
                "role": "system",
                "content": "You are a recipe database that outputs recipes in JSON.\n"
                # Pass the json schema to the model. Pretty printing improves results.
                f" The JSON object must use the schema: {json.dumps(Recipe.model_json_schema(), indent
            },
            {
                "role": "user",
                "content": f"Fetch a recipe for {recipe_name}",
            },
        ],
        model="llama3-70b-8192",
        temperature=0,
        # Streaming is not supported in JSON mode
        stream=False,
        # Enable JSON mode by setting the response format
        response_format={"type": "json_object"},
    )
    return Recipe.model_validate_json(chat_completion.choices[0].message.content)


def print_recipe(recipe: Recipe):
    print("Recipe:", recipe.recipe_name)

    print("\nIngredients:")
    for ingredient in recipe.ingredients:
        print(
            f"- {ingredient.name}: {ingredient.quantity} {ingredient.quantity_unit or ''}"
        )
    print("\nDirections:")
    for step, direction in enumerate(recipe.directions, start=1):
        print(f"{step}. {direction}")


recipe = get_recipe("apple pie")
print_recipe(recipe)
```