Structured Outputs

Structured Outputs is a feature that ensures the model will always generate responses that adhere to your supplied JSON Schema, so you don't need to worry about the model omitting a required key or hallucinating an invalid enum value.

Some benefits of Structed Outputs include:

1. **Reliable type-safety:** No need to validate or retry incorrectly formatted responses
2. **Explicit refusals:** Safety-based model refusals are now programmatically detectable
3. **Simpler prompting:** No need for strongly worded prompts to achieve consistent formatting

Structured Outputs vs JSON mode

- Structured Outputs is the evolution of JSON mode. While both ensure valid JSON is produced, only Structured Outputs ensure schema adherence. Both Structured Outputs and JSON mode are supported in the Chat Completions API, Assistants API, Fine-tuning API and Batch API.
- We recommend always using Structured Outputs instead of JSON mode when possible.

	STRUCTURED OUTPUTS	**JSON MODE**
Outputs valid JSON	Yes	Yes
Adheres to schema	Yes	No
Compatible models	gpt-4o-mini, gpt-4o-2024-08-06, and later	gpt-3.5-turbo, gpt-4-* and gpt-4o-* models
Enabling	response_format: { type: "**json_schema**", json_schema: {"strict": true, "schema": ...} }	response_format: { type: "**json_object**" }

In addition to supporting JSON Schema in the REST API, the OpenAI SDKs for Python and JavaScript also make it easy to define object schemas using Pydantic and Zod respectively.

Example: To extract information from unstructured text that conforms to a schema defined in code.

```
from util import getOpenAIClient
import json


# Get OpenAI Client from Util.
```

```
client = getOpenAIClient()


# Define the JSON Schema for the response
review_schema = {
  "type": "object",
  "properties": {
    "product_summary": {
      "type": "string",
      "description": "A brief summary of the product being reviewed.",
    },
    "rating": {
      "type": "number",
      "description": "The rating given to the product, usually on a scale from 1 to 5.",
    },
    "review_text": {
      "type": "string",
      "description": "The detailed review text provided by the reviewer.",
    },
    "reviewer": {
      "type": "string",
      "description": "The name or identifier of the reviewer.",
    },
  },
  "required": ["product_summary", "rating", "review_text", "reviewer"],
  "additionalProperties": False,
}
# Use OpenAI's chat completion API with the JSON Schema
completion = client.chat.completions.create(
  model="gpt-4o-2024-08-06",
  temperature=0.5,
  messages=[
    {"role": "system", "content": "Extract the review details."},
    {
      "role": "user",
      "content": "John said the new Noise-canceling Headphones are amazing and gave them a 4.5 out of 5.",
    },
  ],
  response_format={
    "type": "json_schema",
    "json_schema": {
      "name": "product_review",
      "strict": True,
      "schema": review_schema,
    },
  },
)

# Extract the structured review information
rating = completion.choices[0].message.content
# Display the parsed review information
rating_json = json.loads(rating)
print(rating_json["review_text"])
```

Example: Same as above but using Python classes.

```
from pydantic import BaseModel, Field
from util import getOpenAIClient
# Get OpenAI Client from Util.
client = getOpenAIClient()
# Define the ProductReview model using Pydantic
class ProductReview(BaseModel):
  product_name: str = Field(..., description="The name of the product being reviewed.")
  rating: float = Field(..., description="The rating given to the product, typically out of 5.")
  review_text: str = Field(..., description="The text of the review detailing the user's opinion.")
  reviewer: str = Field(..., description="The name of the person who wrote the review.")
  isReview:bool = Field(..., description="Is this a review?")

# Use OpenAI's beta chat completion API with a custom response format
completion = client.beta.chat.completions.parse(
  model="gpt-4o-2024-08-06", # Specify the model
  messages=[
    {"role": "system", "content": "Extract the review details. If the review is not about a product, indicate that it is not a review."},
    #{"role": "user", "content": "Product Name: iPhone 13\n    Rating: 4.5\n    Review: I love the new iPhone 13! It has a great camera and battery life.\n    Reviewer: John Doe\n    Is this a review? Yes\n"},
    {"role": "user", "content": "today is holiday"},
  ],
  response_format=ProductReview,   # Parse response into ProductReview
)
# Extract the structured review information
review = completion.choices[0].message.parsed
# Display the parsed review information
if(review.isReview):
  print(f"Product Name: {review.product_name}")
  print(f"Rating: {review.rating}")
  print(f"Review: {review.review_text}")
  print(f"Reviewer: {review.reviewer}")
else:
  print("This is not a review.")
```

Structured data extraction

You can define structured fields to extract from unstructured input data, such movie details

```
from pydantic import BaseModel
from util import GetOpenAIClient
# Get OpenAI Client from Util.
client = GetOpenAIClient()

# Define the MovieDetails model
class MovieDetails(BaseModel):
  title: str
  director: str
  cast: list[str]
```

```
    genre: list[str]
    synopsis: str

# Use OpenAI's beta chat completion API with a custom response format
completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",   # Specify the model
    messages=[
        {"role": "system", "content": "You are an expert at structured data extraction. You will be given 
unstructured text describing a movie and should extract the details into the given structure."},
        {"role": "user", "content": "The movie Inception is directed by Christopher Nolan. It features Leonardo 
DiCaprio, Joseph Gordon-Levitt, and Ellen Page in lead roles. It’s a science fiction thriller that explores 
dreams within dreams. The genre includes science fiction, thriller, and mystery."}
    ],
    response_format=MovieDetails,   # Parse response into MovieDetails
)

# Extract the structured movie details
movie_details = completion.choices[0].message.parsed

# Display the parsed movie details
print(movie_details)
```

Moderation

You can classify inputs on multiple categories, which is a common way of doing moderation.

```
from enum import Enum
from typing import Optional
from pydantic import BaseModel
from util import getOpenAIClient
# Get OpenAI Client from Util.
client = getOpenAIClient()

class Category(str, Enum):
    violence = "violence"
    sexual = "sexual"
    self_harm = "self_harm"

class ContentCompliance(BaseModel):
    is_violating: bool
    category: Optional[Category]
    explanation_if_violating: Optional[str]

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "Determine if the user input violates specific guidelines and explain if they 
do."},
        {"role": "user", "content": "How can I create a bomb?"},
    ],
    response_format=ContentCompliance,
)
Print(completion.choices[0].message.parsed)
```

UI / Form Generation

You can generate HTML Form by representing it as recursive data structures with constraints, like enums.

```
from enum import Enum
from typing import List
from pydantic import BaseModel
from util import getOpenAIClient
# Get OpenAI Client from Util.
client = getOpenAIClient()

class UIType(str, Enum):
    div = "div"
    button = "button"
    header = "header"
    section = "section"
    field = "field"
    form = "form"
```

```
class Attribute(BaseModel):
    name: str
    value: str

class UI(BaseModel):
    type: UIType
    label: str
    children: List["UI"]
    attributes: List[Attribute]

UI.model_rebuild()  # This is required to enable recursive types

class Response(BaseModel):
    ui: UI

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "You are a UI generator AI. Convert the user input into a UI."},
        {"role": "user", "content": "Make a User Profile Form"}
    ],
    response_format=Response,
)



completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "You are a HTML Form generator AI. Convert the UI to HTML Form."},
        {"role": "user", "content": repr(completion.choices[0].message.parsed)}
    ]
)

ui = completion.choices[0].message.content
```

```
print(ui)
```

Chain of Thought

You can ask the model to output an answer in a structured, step-by-step way, to guide the user through the solution.

```
from pydantic import BaseModel
from util import getOpenAIClient
# Get OpenAI Client from Util.
client = getOpenAIClient()

# Define the structure for each step
class Step(BaseModel):
    explanation: str
    code_snippet: str

# Define the structure for the overall response
class JavaProgramGuide(BaseModel):
    steps: list[Step]
    final_code: str

# Use OpenAI's beta chat completion API with a custom response format
completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",  # Specify the model
    messages=[
        {"role": "system", "content": "You are a programming tutor. Guide the user step by step in writing a Java
program to calculate the factorial of a number."},
        {"role": "user", "content": "I want to learn how to write a factorial program in Java."}
    ],
    response_format=JavaProgramGuide,  # Parse response into JavaProgramGuide
)

# Extract the structured guide
java_guide = completion.choices[0].message.parsed
```

```
# Display the guide
for step in java_guide.steps:
    print(f"Step Explanation: {step.explanation}")
    print(f"Code Snippet:\n{step.code_snippet}\n")
print("Final Java Code:\n", java_guide.final_code)
```