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 adherance. 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.

```
import { getOpenAIClient } from './util.js';
import { zodResponseFormat } from "openai/helpers/zod";
import { z } from 'zod';
```

```
// Get OpenAI Client from Util.
const client = await getOpenAIClient();

// Define the ProductReview model using Zod
const ProductReview = z.object({
   product_name: z.string(),
   rating: z.number(),
   review_text: z.string(),
   reviewer: z.string(),
});

try {
   // Use OpenAI's chat completion API with a custom response format
   const completion = await client.beta.chat.completions.parse({
      model: "gpt-4o-2024-08-06",   // Specify the model
      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 : zodResponseFormat(ProductReview, "event"), // Use the custom response format
   });

   const event = completion.choices[0].message.parsed;
   console.log(event);

} catch (e) {
   console.error(`An error occurred: ${e}`);
}
```

Same as above but with JSON format

```
import { getOpenAIClient } from './util.js';
import { zodResponseFormat } from "openai/helpers/zod";
import { z } from 'zod';
```

```
// Get OpenAI Client from Util.
const client = await getOpenAIClient();

// Define the ProductReview model using Zod
const ProductReview = z.object({
   product_name: z.string(),
   rating: z.number(),
   review_text: z.string(),
   reviewer: z.string(),
});

try {
   // Use OpenAI's chat completion API with a custom response format
   const completion = await client.chat.completions.create({
      model: "gpt-4o-2024-05-13",   // Specify the model
      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." },
         { role: "user", content: "return data in the json format: {product_name: 'abcd',rating: 2.5,review_text:
'The abcd is good', reviewer: 'sandeep'}" },
      ],
      response_format : {type: "json_object"},  // Use the custom response format
   });

   const event = completion.choices[0].message.content;
   console.log(event);
} catch (e) {
   console.error(`An error occurred: ${e}`);
}
```

Moderation

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

```
import { getOpenAIClient } from './util.js';
import { zodResponseFormat } from "openai/helpers/zod";
```

```
import { z } from 'zod';

// Get OpenAI Client from Util.
const client = await getOpenAIClient();

// Define the Category enum
const Category = z.enum(['violence', 'sexual', 'self_harm']);

// Define the ContentCompliance model using Zod
const ContentCompliance = z.object({
  is_violating: z.boolean(),
  category: Category.optional(),
  explanation_if_violating: z.string().optional(),
});

try {
  // Use OpenAI's chat completion API with a custom response format
  const completion = await client.beta.chat.completions.parse({
    model: "gpt-4o-2024-08-06",  // Specify the model
    messages: [
      { role: "system", content: "Extract the review details." },
      { role: "user", content: "How can I create a bomb?" },
    ],
    response_format : zodResponseFormat(ContentCompliance, "event"), // Use the custom response
format
  });

  const event = completion.choices[0].message.parsed;
  console.log(event);

} catch (e) {
  console.error(`An error occurred: ${e}`);
}
```

UI / Form Generation

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

```
import { getOpenAIClient } from './util.js';
import { zodResponseFormat } from "openai/helpers/zod";
import { z } from 'zod';

// Get OpenAI Client from Util.
const client = await getOpenAIClient();

// Define the UIType enum
const UIType = z.enum(['div', 'button', 'header', 'section', 'field', 'form']);

// Define the Attribute model using Zod
const Attribute = z.object({
  name: z.string(),
  value: z.string(),
});

// Define the UI model using Zod
const UI = z.lazy(() => z.object({
  type: UIType,
  label: z.string(),
  children: z.array(UI).optional(),
  attributes: z.array(Attribute).optional(),
}));

// Define the Response model using Zod
const Response = z.object({
  ui: UI,
});

// Use OpenAI's chat completion API with a custom response format
let completion = await 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: zodResponseFormat(Response, "event"),
});

completion = await 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: JSON.stringify(completion.choices[0].message.parsed) },
  ],
});

const ui = completion.choices[0].message.content;
console.log(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.

```
import { getOpenAIClient } from './util.js';
import { zodResponseFormat } from "openai/helpers/zod";
import { z } from 'zod';

// Get OpenAI Client from Util.
const client = await getOpenAIClient();

// Define the structure for each step
const Step = z.object({
  explanation: z.string(),
  code_snippet: z.string(),
});

// Define the structure for the overall response
const JavaProgramGuide = z.object({
```

```
  steps: z.array(Step),
  final_code: z.string(),
});

  // Use OpenAI's chat completion API with a custom response format
  let completion = await 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: zodResponseFormat(JavaProgramGuide, "event"),
  });

  // Extract the structured guide
  const java_guide = completion.choices[0].message.parsed;

  // Display the guide
  java_guide.steps.forEach(step => {
    console.log(`Step Explanation: ${step.explanation}`);
    console.log(`Code Snippet:\n${step.code_snippet}\n`);
  });
  console.log("Final Java Code:\n", java_guide.final_code);
```