Output parsers

Intro

Alright, let's dive into the nitty-gritty of output parsers in Langchain.

Why do we need them? Well, when working with LLMs you will realize the response may not be the same if we ask the same question twice and the format for the output too.

For example, if we ask the recipe for a dish, sometimes the LLM will return with Ingredients as the first step, sometimes with Preparations. It is very normal as LLM is generative not deterministic. This maybe fine if it is some human that read those responses, but it will become a problem if we want some machine to process those responses.

And try to see what happens when you change the LLM type.

Think of output parsers as your formatting sidekicks, working their magic in two main ways:

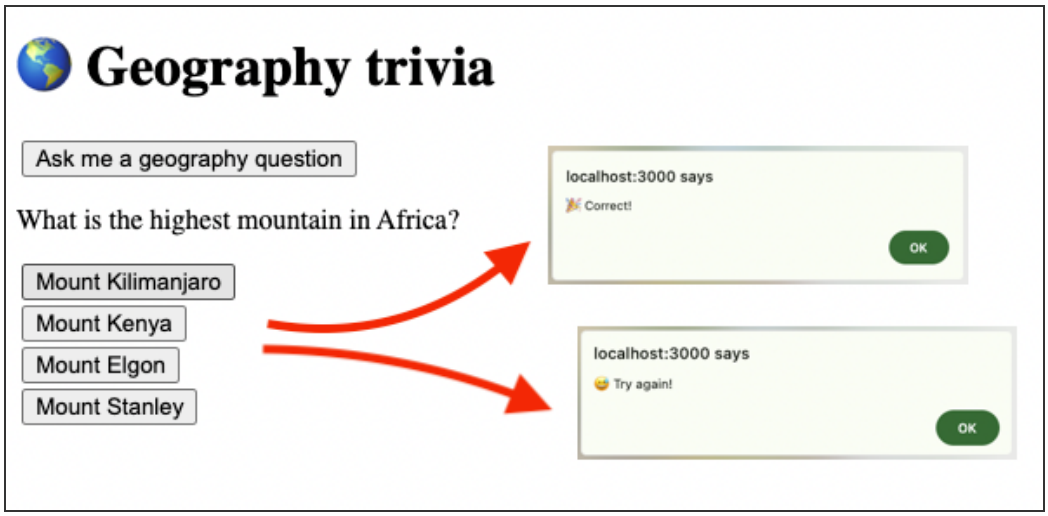
1. they can tweak your prompt by adding things such as prefixes or suffixes to steer the output in a certain direction. Want a Javascript array? Want a simple string or JSON nested structure? Langchain's got you covered!
2. After the output pops out, a Javascript function steps in to tidy things up. This function might even tap into LLMs to polish the output into your dream format.

Example setup

Langchain has several pre-defined output parsers that can be used directly. We will some of them in the following example.

You were tasked to make a trivia game using ChatGPT. The user will ask for a trivia question with 4 possible answers. Only one of these answers is correct. Once a possible answer is clicked, a popup indicating if the answer is correct or incorrect will be displayed. At any time the user will be able to ask for a new question.

This is how the final version of the app will look like:



We will start from a very simple state.

On the backend, we will build an OpenAI model and ask it for a trivia question:

```
// code/trivia-game/src/app/api/route.js

import { ChatOpenAI } from "@langchain/openai"
import { PromptTemplate } from "@langchain/core/prompts"
import { LLMChain } from "langchain/chains"

const model = new ChatOpenAI({
    openAIApiKey: process.env.OPENAI_API_KEY,
})

const prompt = PromptTemplate.fromTemplate(
    `Ask me a trivia question about geography.`
)

const chain = new LLMChain({
    llm: model,
    prompt
})

export async function GET() {
    const gptResponse = await chain.call()
    return Response.json({question: gptResponse.text})
}
```

Notice that this time we are using a GET request, given that we don't need to send any request body. We just need to get a question back.

On the front end, we have a simple button that makes the call to the back end. Using a React state variable, the question is displayed once we get the response:

```
// code/trivia-game/src/app/page.js

import { ChatOpenAI } from "@langchain/openai"
```

```
import { PromptTemplate } from "@langchain/core/prompts"
import { LLMChain } from "langchain/chains"

const model = new ChatOpenAI({
    openAIApiKey: process.env.OPENAI_API_KEY
})

const prompt = PromptTemplate.fromTemplate(
    `Ask me a trivia question about geography.`
)

const chain = new LLMChain({
    llm: model,
    prompt
})

export async function GET() {
    const gptResponse = await chain.call()
    return Response.json({question: gptResponse.text})
}
```

This is how the project will look like in its first phase.

The StringOutputParser

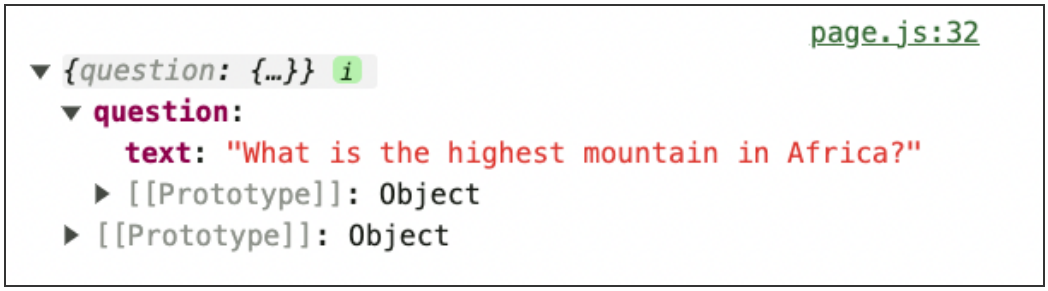
The output parsers from Langchain.js are meant to convert the AI responses into common structures, like JSON, Arrays, String, and more.

The StringOutputParser takes the model's output and converts it into a simple string. This one is maybe the easiest-to-use parser.

Let's take a look at what an actual response from a ChatGPT call looks like:

```
const prompt = PromptTemplate.fromTemplate(
    `Ask me a trivia question about geography.`
)

const chain = new LLMChain({
    llm: model,
    prompt
})

const gptResponse = await chain.call()

console.log(gptResponse)
```

If we print the `gptResponse` constant it will look like so:



This is the structure that comes from a ChatGPT model. No added output parsers.

But if we use another type of model such as a Llama (made by Meta) or Claude (made by Anthropic) then this structure can be different. Each model may provide the answer wrapped into a different structure.

This means that the line that extracts the final string response will fail, because the `text` property may not exist:

```
question: gptResponse.text
```

In the end, we need just a simple string containing the question.

This is where the `StringOutputParser` comes into play. You can add the parser to the chain, and it will make sure the various response types that come from different LLM types, are transformed into just a simple string.

Let's see how this will look like in code:

```
// code/trivia-game/src/app/api/route.js

import { ChatOpenAI } from "@langchain/openai"
import { PromptTemplate } from "@langchain/core/prompts"
//🟠 importing the parser
import { StringOutputParser} from "@langchain/core/output_parsers"

const model = new ChatOpenAI({
    openAIApiKey: process.env.OPENAI_API_KEY
})

//🟠 making the prompt and invoking the chain
```

```
const makeQuestion = async ()=> {
    const prompt = PromptTemplate.fromTemplate(
        `Ask me a trivia question about geography.`
    )
    //🟠 adding the parser to the chain so that we get
    //🟠 just a simple string output instead of an object
    const chain = prompt
                    .pipe(model)
                    .pipe(new StringOutputParser())
    return await chain.invoke()
}

export async function GET() {
    //🟠 extracted the make question part into its method
    const question = await makeQuestion()
    return Response.json({question})
}
```

Remember that in a chain we can set up multiple consecutive operations, including output parsers.

Notice that after we have added an output parser to the chain we don't need to do the manual `.text` parsing:

```
// before using the parser
return Response.json({question: gptResponse.text})

// after adding the output parser to the chain
return Response.json({question})
```

And this will work even if we change the model type.

Next let's see how we can use another type of parser, to create an array of responses.

The CommaSeparatedListOutputParser

There are cases where we need to get multiple items from the response of an LLM.

Let's ask ChatGPT to also help us with some possible answers for the Trivia question. This is how the prompt will look like:

```
`Give 4 possible answers for {question}, separated by commas,
3 false and 1 correct, in a random order.`
```

Without any output parsers, the standard response that will come from ChatGPT will look like so:

```
AIMessage {
  lc_serializable: true,
  lc_kwargs: {
    content: 'North America, Africa, Australia, Asia',
```

```
    additional_kwargs: { function_call: undefined, tool_calls: undefined },
    response_metadata: {}
  },
  lc_namespace: [ 'langchain_core', 'messages' ],
  content: 'North America, Africa, Australia, Asia',
  name: undefined,
  additional_kwargs: { function_call: undefined, tool_calls: undefined },
  response_metadata: {
    tokenUsage: { completionTokens: 8, promptTokens: 47, totalTokens: 55 },
    finish_reason: 'stop'
  }
}
```

A long complex JavaScript object that we will need to manually parse.

By the way, we will go more into detail about the `AIMessage` class during the chat memory chapter.

For situations like these Langchain can help us via the parser CommaSeparatedListOutputParser.

The `CommaSeparatedListOutputParser` will take a complex answer given by an LLM and it will reformat it as a standard array of strings. For example, the `AIMessage` from the above example will become:

```
[
  'North America',
  'Africa',
  'Australia',
  'Asia'
]
```

Much much cleaner!

Let's see how our backend code will look like after we add this `CommaSeparatedListOutputParser` to the possible answers for the question:

```
// code/trivia-game/src/app/api/route.js

import { ChatOpenAI } from "@langchain/openai"
import { PromptTemplate } from "@langchain/core/prompts"
import { StringOutputParser } from "@langchain/core/output_parsers"
//🟠 import the CommaSeparatedListOutputParser
import { CommaSeparatedListOutputParser } from "@langchain/core/output_parsers"

const model = new ChatOpenAI({
    openAIApiKey: process.env.OPENAI_API_KEY,
    //🟠 increase the model temperature
    temperature: 0.9
})

 //🟠 ask the model for possible answers
const makePossibileAnswers = async (question) => {
    const prompt = PromptTemplate.fromTemplate(
```

```
        `Give 4 possible answers for {question}, separated by commas,
        3 false and 1 correct, in a random order.`
    )
     //🟠 apply the CommaSeparatedListOutputParser to the chain
    const chain = prompt
                    .pipe(model)
                    .pipe(new CommaSeparatedListOutputParser())
    return await chain.invoke({question: question})
}

const makeQuestion = async ()=> {
    const prompt = PromptTemplate.fromTemplate(
        `Ask me a trivia question about geography.`
    )
    const chain = prompt
                    .pipe(model)
                    .pipe(new StringOutputParser())
    return await chain.invoke()
}

export async function GET() {
    const question = await makeQuestion()
     //🟠 once we have the question fetch some possible answers
    const answers = await makePossibileAnswersmakePossibileAnswers(question)
    return Response.json({question, answers})
}
```

Note that we have increased the temperature of the model, given that it will need to be creative when generating the possible answers.

While you may be tempted to say that we can do this with something like the Javascript string split() method think again! What will happen if we change the LLM and the structure of the response changes !? This is why Langchain is so powerful! It standardizes the way we interact with the AI models.

At this point, we are using two output parsers:

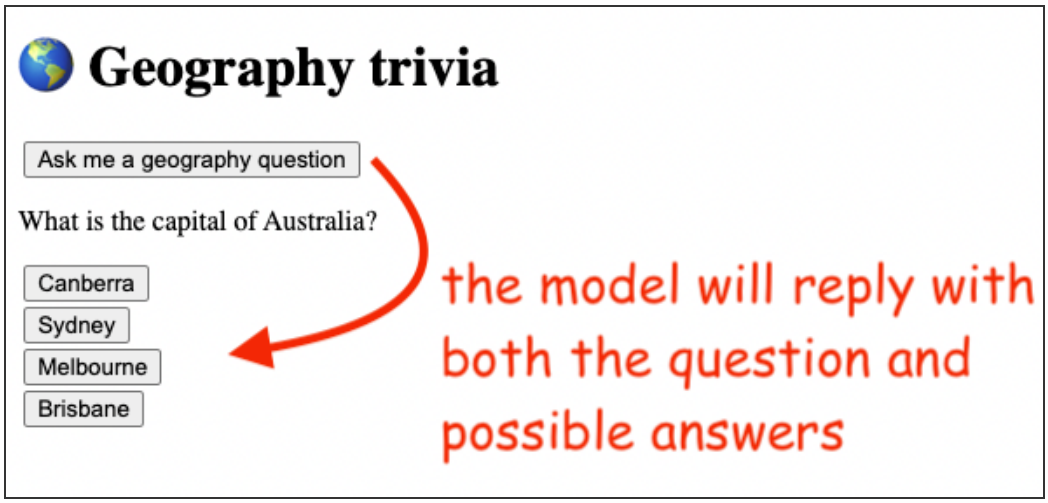
- the `StringOutputParser` for the main trivia question
- the `CommaSeparatedListOutputParser` to parse the list of possible answers

On the frontend, we will read the possible answers, and we will display them as buttons:

```
// code/trivia-game/src/app/page.js

'use client'
import { useState } from "react"

export default function Home() {
  const [question, setQuestion] = useState()
   //🟠 adding an empty array state variable to store the answers
  const [answers, setAnswers] = useState([])
```

```
  const getTriviaQuestion = async ()=> {
    const response = await fetch('api')
    const data = await response.json()
    console.log(data)
    setQuestion(data.question)
    //🟠 once we have the answers update the state
    setAnswers(data.answers)
  }

  return (<>
    <h1>🌎 Geography trivia</h1>
    <button onClick={getTriviaQuestion}>
      Ask me a geography question
    </button>
    <p>{question}</p>
    {/*🟠 loop through and display each answer as a button */}
    {answers.map((answ, i) =>
      <button key={i}>
        {answ}
      </button>
    )}
  </>)
}
```

And this is how the UI of the app will look like at this stage:



By the way, if the list is not separated by a comma character you can use the CustomListOutputParser.

A fun extra project will be to add memory to this app so that it will not repeat its questions. We will see in the next chapters how to use the chat memory features of LangChain.

The next step will be to decide if the user clicked the correct answer.

The getFormatInstructions() method and custom parsers

Before we continue let's make a short detour and talk about the `getFormatInstructions()` method, and how output parsers work under the hood.

There are many output parsers in Langchain. The documentation provides this table to summarize the main types:

Name	Supports Streaming	Has Format Instructions	Calls LLM	Input Type	Output Type	Description
String	✅			`string` or `Message`	`string`	Takes language model output (either an entire response or as a stream) and converts it into a string. This is useful for standardizing chat model and LLM output and makes working with chat model outputs much more convenient.
HTTPResponse	✅			`string` or `Message`	`binary`	Allows you to stream LLM output properly formatted bytes a web HTTP response for a variety of content types.
OpenAIFunctions	✅	(Passes `functions` to model)		`Message` (with `function_call`)	JSON object	Allows you to use OpenAI function calling to structure the return output. If you are using a model that supports function calling, this is generally the most reliable method.
CSV		✅		`string` or `Message`	`string[]`	Returns a list of comma separated values.
OutputFixing			✅	`string` or `Message`		Wraps another output parser. If that output parser errors, then this will pass the error message and the bad output to an LLM and ask it to fix the output.
Structured		✅		`string` or `Message`	`Record<string, string>`	An output parser that returns structured information. It is less powerful than other output parsers since it only allows for fields to be strings. This can be useful when you are working with smaller LLMs.
Datetime		✅		`string` or `Message`	`Date`	Parses response into a JavaScript date.

All of these parsers have in common is the `getFormatInstructions()` method.

Let's create a `CommaSeparatedListOutputParser` and call this method:

```
const parser = new CommaSeparatedListOutputParser()
console.log(parser.getFormatInstructions())

// PRINTS:
// the result will be just a simple string:
// Your response should be a list of comma separated values, eg: `foo, bar, baz`
```

Basically, this is just an extra part we can add to the prompt.

It's kind of funny if we think about it. We are using an LLM to parse the output of ... an LLM 😅.

For example, we can just make a `RunnableSequence` and pass in the result of the `getFormatInstructions()` method:

```
const parser = new CommaSeparatedListOutputParser()

const chain = RunnableSequence.from([
  PromptTemplate.fromTemplate(`Give me 3 {topic}.\n{formatInstructions}`),
  new OpenAI({}),
  parser,
])

return await chain.invoke({
  topic: 'car brands',
  formatInstructions: parser.getFormatInstructions()
})
```

This will do the same as adding the parser directly to the chain. Will output something similar to this:

```
[
  'Audi',
  'Ford',
  'Doge'
]
```

We can even make our own custom output parsers. Documentation link here.

There are two main methods an output parser must implement:

- `getFormatInstructions()` returns a string containing instructions for how the output of a language model should be formatted. You can inject this into your prompt if necessary.
- `parse()`: takes in a string (assumed to be the response from a language model) and parses it into some structure

The StructuredOutputParser

Large Language Models like structure! Their output is much more accurate when we define a clear expected structure for both the input and the output. More info about this here.

While `CommaSeparatedListOutputParser` or `StringOutputParser` are great tools for simpler direct responses, when we have to deal with more complex outputs we will need to reach for other types of parsers such as StructuredOutputParser or JsonOutputParser.

In the case of our small trivia app, a big improvement will be to receive the output in the following format:

```
{
  question: 'question text here',
  answers: [
    'foo',
    'bar',
    'qux',
    'baz',
  ],
  correct: 2 //  'qux' is correct
}
```

To achieve this we will use a `StructuredOutputParser` with a Zod Schema. The reason for this is that `StructuredOutputParser` is more versatile and covers more use cases than the `JsonOutputParser`.

Zod is a schema validation library for defining complex data structures.
Let's see how adding a `StructuredOutputParser` will change our backend code:

```
// code/trivia-game/src/app/api/route.js

import { ChatOpenAI } from "@langchain/openai"
import { PromptTemplate } from "@langchain/core/prompts"
//🟠 remove imports of StringOutputParser & CommaSeparatedListOutputParser
//🟠 we will need this one to build a new chain structure
import { RunnableSequence } from "@langchain/core/runnables"
//🟠 adding the StructuredOutputParser to replace the other parsers
import { StructuredOutputParser } from "langchain/output_parsers"
//🟠 we will use the Zod schema to define the types of returned data
import { z } from "zod"

const model = new ChatOpenAI({
    openAIApiKey: process.env.OPENAI_API_KEY,
    temperature: 0.9
})

//🟠 Zod is used to define if a field is a string, number, array etc
const parser = StructuredOutputParser.fromZodSchema(
    z.object({
        question: z.string().describe(
            `tell me a random geography trivia question`
        ),
        answers: z
            .array(z.string())
            .describe(`
                give 4 possible answers, in a random order,
                out of which only one is true.`
            ),
        correctIndex: z.number().describe(
            `the number of the correct answer, zero indexed`
        ),
    })
)
```

```
//🟠 define a new chain with RunnableSequence
const chain = RunnableSequence.from([
    PromptTemplate.fromTemplate(
        `Answer the user question as best as possible.\n
        {format_instructions}`
    ),
    model,
    parser,
])

//🟠 using the StructuredOutputParser we can now wrap all the
//🟠 data into one single structure
const makeQuestionAndAnswers = async () => {
    //🟠 returning a JSON the defined structure
    return await chain.invoke({
        format_instructions: parser.getFormatInstructions(),
    })
}

export async function GET() {
    //🟠 makeQuestion() & makePossbileAnswers() are merged in one function
    const data = await makeQuestionAndAnswers()
    return Response.json({data})
}
```

Even if it looks like we have made a lot of changes if we remove the new line comments you will notice that our code is simpler now.

From the LLM we a returning a single JSON structure that will contain all the info our app needs.

Also, the `makeQuestion()` and the `makePossibileAnswers()` methods are now merged into a single call making the app a bit more faster.

With these changes added we can now indicate if the user clicked a correct or incorrect answer.

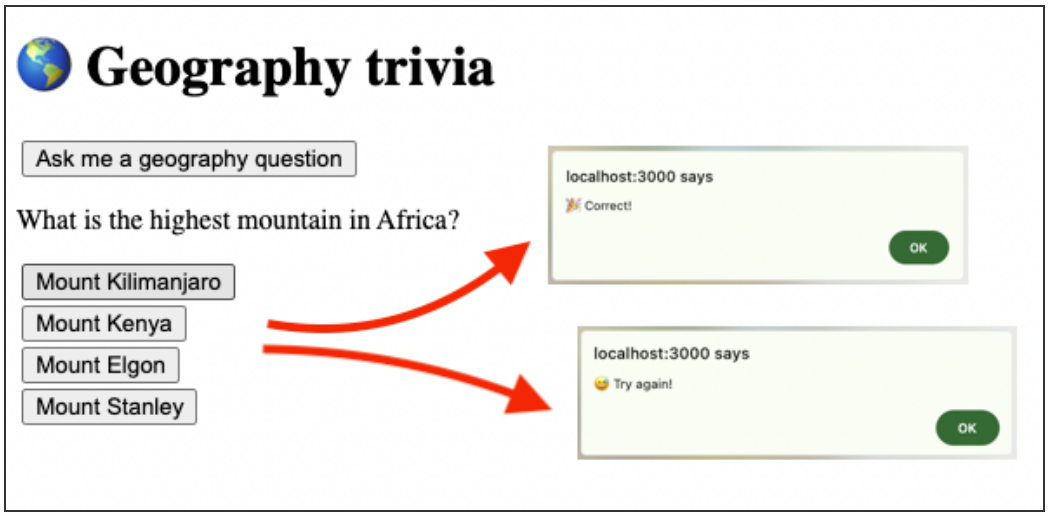
Let's see the changes in the frontend:

```
// code/trivia-game/src/app/page.js

'use client'
import { useState } from "react"

export default function Home() {
  const [question, setQuestion] = useState()
  const [answers, setAnswers] = useState([])
  //🟠 adding a third state var to store the correct answer
  const [correctIndex, setCorrectIndex] = useState()

  const getTriviaQuestion = async ()=> {
    const response = await fetch('api')
    const { data } = await response.json()
```

```
    setQuestion(data.question)
    setAnswers(data.answers)
    //🟠 once we have the correct answer index update the state
    setCorrectIndex(data.correctIndex)
  }

  //🟠 use the correct answer index to see if the right answer was picked
  const checkAnswer = async (selectedIndex) => {
    (correctIndex === selectedIndex) ?
      alert('🎉 Correct!'):
      alert('😅 Try again!')
  }

  return (<>
    <h1>🌎 Geography trivia</h1>
    <button onClick={getTriviaQuestion}>
      Ask me a geography question
    </button>
    <p>{question}</p>
    {/*🟠 onClick check if the user answered correctly */}
    {answers.map((answ, i) =>
      <button key={i}
        onClick={() => checkAnswer(i)}>
        {answ}
      </button>
    )}
  </>)
}
```

Nothing too special here. We have added a click listener to each answer button and we will check if the index of that clicked button matches the index of the correct answer.

And this is how the final version of the UI will look like:



Note that even if newer models such as ChatGpt 4 can do part of this data manipulation out of the box, they are more expensive to use. Therefore we can leverage output parsers to save costs and also define a very clear output format, thus reducing unexpected behavior.

Our final app is just a bit over 80 lines of code. Both backend and frontend. We're talking about a fully functional trivia game with an endless stream of questions. Plus, it's super flexible and a breeze to tweak. One more testament to how much power the new capabilities of LLMs provide us as web developers. Cool stuff 😎!