JS-Craft Academy

Daniel Nastase

LangChain

for JavaScript developers

Integrate LLMs into web apps
a beginner's guide

Build 5 apps with
LangChain, Next.js and React

1. Introduction

1.1. Getting Book Updates and Code Examples

To receive the latest versions of the book, subscribe to our mailing list by emailing daniel@js-craft.io. I strive to keep my book with the latest version of LangChain and other libraries. I regularly update the code and content so make sure to subscribe to stay informed about these updates.

In order to access the full source code for all the completed projects, or the PDF version of the book just write me at daniel@js-craft.io.

1.2. About the author

Hi there, friend! I'm Daniel, a software developer and educator.

I like computers. I try to make them like me back. More than computers I like humans. I think every person has value, intelligence, and good inside. I believe that education is the key to building a better, stable, and richer world.

I used to work at companies such as Skillshare, Tradeshift and ING, where I had a chance to be exposed to completely different types of frontend development in various teams.

Over the past five years, I've been writing articles on js-craft.io about JavaScript, CSS, and other software development topics.

I've always enjoyed teaching, holding both in-class and online classes, and being involved in tech education startups.

You can always reach me at daniel@js-craft.io, and read more about me at https://www.js-craft.io/about/.

You can find me also on:

- GitHub: github.com/daniel-jscraft
- Twitter: @js_craft_hq
- Mastdotodn: @daniel_js_craft
- YouTube: @js-craftacademy6740

1.3. How this book came to be

The first time I wrote a computer program was 26 years ago, in Turbo Pascal. To someone who did not know how to code it felt like casting spells. But I understood how spells were made. If - then - else statements, functions, variables. This was how you make spells.

The only time I've had again this feeling was in 2022 when a friend showed me ChatGPT. It felt like some mystical magic. But this time I was not the one who was writing the spells. I did not truly understand how that 'AI thing' was made; you couldn't create it with just the programming I was used to.

So, I went down to the "learn how AI works" rabbit hole. And oh boy, this was a deep one. I was trying to understand this thing from the ground up.

I was trying to build and train my models from scratch. After some time, my only practical project was a pure JavaScript neuronal network. The model was written from scratch and could be trained to detect if a hand-drawn shape was a digit.

I gave up after some time. With all the different architectures, bias neurons, training data sets, backpropagation, or gradient descent. It was too much for my mediocre brain. The AI thing was a bit more logical and less mystical but I was far away from being able to use the new learnings practically.

Therefore I went back to my normal job as a web developer. But in the background of my mind was still the feeling that I should still pay attention to AI. There was something there.

After a while, I was listening to an episode of the excellent Latent Space podcast. I was following Swan Swyx Wang, the co-founder of this podcast from the time he has speaking about React (the JavaScript framework not ReAct prompting).

In that episode, Swyx mentioned something about LangChain. This LangChain framework was made for integrating AI models with "traditional" apps. I realized that I did not fully understand all the inner mechanics of databases, or operating systems, but I was using them. Therefore decided to give AI another shot. But this time from a more practical angle.

One book later I can now say that I love LangChain for all it taught me about how Large Language Models (LLMs for short) work. The abstractions, the mental models, and the use cases of this framework will teach you a lot about AI models and how to use them in conjunction with JavaScript-powered apps.

LangChain can be seen as the orchestrator that connects nearly everything in the AI-Webapp integration system. This makes it an excellent gateway for understanding how all the components work together.

So, let's learn!

1.4. What is LangChain

LangChain is a framework designed to simplify the creation of applications that integrate Large Language Models (LLMs).

LangChain provides all the AI integration building blocks in one framework. It offers a modular, flexible, and scalable architecture that is easy to maintain.

You can see LangChain as the glue layer for almost everything in the AI ecosystem.

By learning LangChain, you will gain a deep understanding of the structure, workflows, and practices of AI Engineering.

Some use cases for LangChain include:

- LLMs are trained on human-written unstructured text. While this works well for human interactions, it may not work well for sending unformatted text to an API. APIs instead of structured data like JSON. LangChain can format the input and output in LLM interactions.
- LLMs are stateless. They don't remember who you are or what you said a few seconds ago. LangChain provides support for both long-term and short-term memory.
- LLMs can be slow. LangChain can stream responses as they are generated, providing fast feedback to the user.
- LLMs have knowledge cutoff dates. For example, GPT 3.5 does not have training data after 2021. And GPT-4o does not know anything after Oct 2023. With LangChain, we can create and manage AI Agents that go online, search for information, and use tools to parse that information.
- We can chain multiple LLMs together. For example, we can generate an article with Google's Gemini model and then pass the text to Midjourney to generate images for that article.

- LLMs lack access to internal organization documents. We can use Retrieval-Augmented Generation (RAG) to provide extra context to a model, allowing it to interact with users based on an organization's rules and knowledge. LangChain has a great toolset for RAG operations.

LangChain standardizes operations like the ones above, making it easy to swap components as needed. Want to use a different LLM because it's better or cheaper? Want to change the vector database to store the RAG embeddings? No problem, LangChain has you covered.

LangChain abstracts and standardizes common patterns when working with LLMs. Remember the old days with browser incompatibilities? Then jQuery appeared. This is what LangChain aims to do for LLM integration.

While it is possible to create any app without LangChain, it simplifies the process to a great extent and is much better than manual prompting.

If you want to delve deeper into the subject, I recommend listening to the episode with Harrison Chase, the founder of LangChain from the excellent Latent Space podcast.

1.5. Overview of this book

In this book, we email on a fun, hands-on, and pragmatic journey to learning LangChain. Every main chapter has an associated example application that gradually evolves during that chapter. Each step introduces a few new LangChain core concepts in a manageable way without overwhelming you.

The book is purposefully broken down into short chapters, each focusing on different topics.

Here is how the main chapters and example applications are structured:

- **Chapter 1**: We will use a Story Generator for Kids app to introduce the basics of LangChain. We will connect to the first LLM, and use prompt templates and chains. We will delve deeper into partial templates, template composition, and more.
- **Chapter 2**: After the basic setup in Chapter 1, this chapter will introduce the concept of streaming. Using the same app, Story Generator for Kids, we will stream long responses from the LLMs to improve the user experience.
- **Chapter 3**: One challenge with LLMs is that they give responses in an unstructured format, as they were trained on human-intelligible text. In this chapter, we will study output parsers by making a full Trivia Game from scratch using ChatGPT.
- **Chapter 4**: The memory module enables LLM memory for LangChain apps. We can remember past conversations and responses in the current session similar to how ChatGPT does it. In this chapter, we will build a Tea Facts Wiki application.
- **Chapter 5**: One of the key features of LangChain is its support for RAG - Retrieval Augmented Generation. In this chapter, we will build a Chat Bot app that answers questions based on external context. We will see how to provide data from sources like PDFs or online documents, what embeddings are, and how vector data stores work.

- **Chapter 6**: More flexible and versatile compared to chains, AI agents help you build complex solutions and enable access to third-party tools such as Google search and math calculators. They can make decisions on the appropriate tool to use in a given situation.

From building a Story Generator to creating a Trivia Game and a real Chat Bot application, each chapter gradually expands your understanding. By the end, we will have explored streaming, output parsing, memory modules, AI agents, and RAG capabilities, empowering you to develop complex solutions.

Get ready to embark on an enriching journey into the world of LangChain development.

1.6. Requirements and how to use this book

LangChain offers implementations in both JavaScript and Python. Throughout this book, our focus will be on JavaScript.

You'll find that you learn best by coding along with the examples provided in the book. My suggestion is to initially go through each chapter, absorbing the content to grasp the concepts. Then, during a second pass, code along as you progress.

We'll utilize a combination of LangChain, React, and Next.js. While a basic understanding of React is expected, you need not be an expert.

For setting up Next.js, the simplest approach is to utilize the create-next-app utility.

In most instances, our work will be confined to just 2 files:

- `src/app/page.js` - handling the frontend; its primary function is to display information and data.
- `src/app/api/route.js` - managing the backend; this file will handle the API interactions with the LLM.

The examples will progress in a step-by-step iterative manner. Each modification will be denoted by the 🟠 sign, with the path of the modified file provided for reference.

For instance:

```
// src/app/page.js - path of the updated file

//🟠 add the PromptTemplate to avoid repetition - file change
import { PromptTemplate } from "@langchain/core/prompts"

//🟠 make a new ChatGPT model - file change
const model = new ChatGPT({
  openAIApiKey: process.env.OPENAI_API_KEY,
  temperature: 0.9
})
```

Please note that LangChain is currently undergoing development and is chaining rapidly. While I'll strive to update the book frequently, there might be brief periods where method names or import statements are not aligned with the latest version.

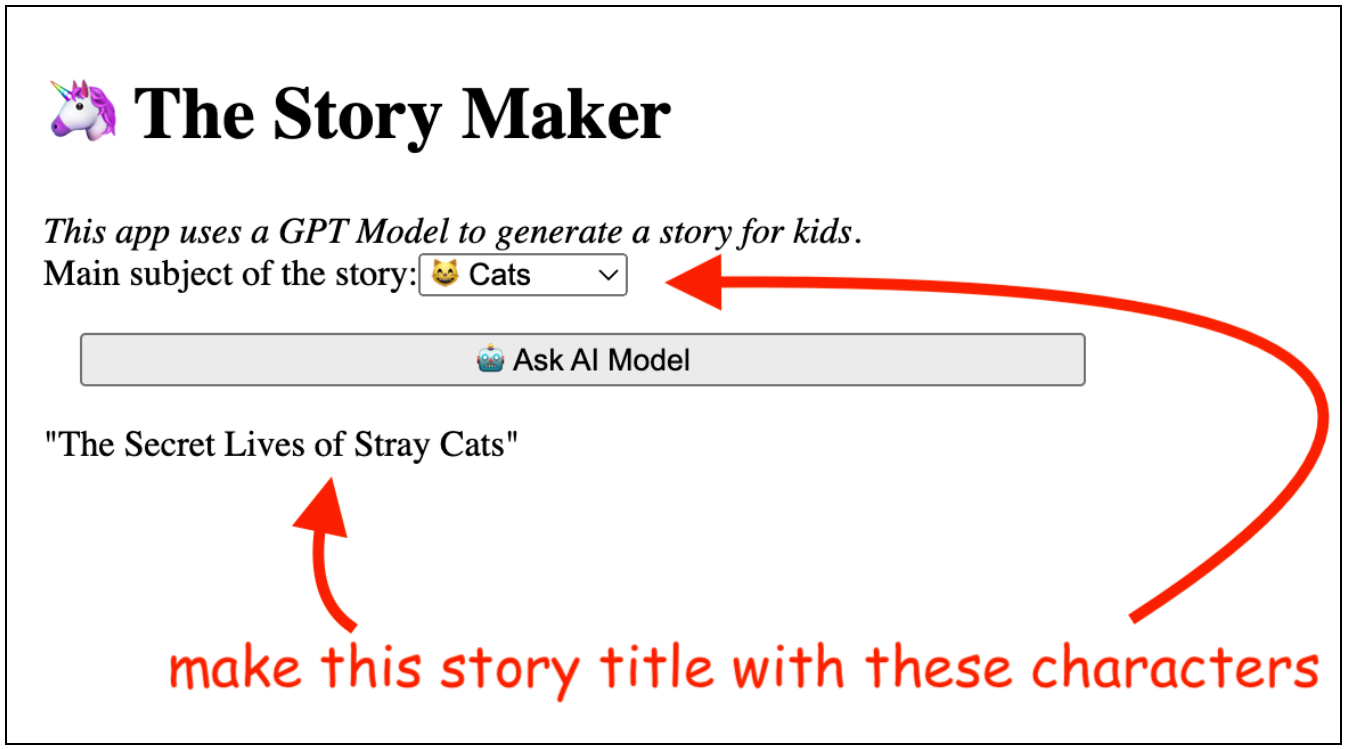
For support, feel free to reach out via email at daniel@js-craft.io.

2. Models, Prompts and Chains

2.1. Introduction

It's now time to create our first app using LangChain and JavaScript.

You have been assigned to create an app called "The Story Maker". This app will generate titles for children's stories. Here's what the final UI of the app will look like:



For instance, if we specify that we want a story with `Unicorns` as the main characters, the app will generate a story title like this:

```
"The Magical Adventures of Stardust the Unicorn"
```

2.2. Getting Started

The easiest way to get started is by using the create-next-app utility. Once you have it installed, create a new project named `story-generator`:

```
yarn create next-app
```

Alternatively, you can use the starting code from the code examples folder. Just remember to run `npm install` from within the folder of the example.

We'll need to install a couple of npm dependencies: LangChain and OpenAI.

Go to the root folder of the project and, in your terminal, execute the following command:

```
npm i langchain @langchain/openai
```

At the time of writing this book, these are the current versions of the installed modules:

```
"@langchain/openai": "^0.0.14",
"langchain": "^0.1.19",
```

Please note that LangChain is currently undergoing rapid development, and as it progresses, certain parts of the code may become obsolete. I'm striving to keep the book updated, but if, for any reason, something breaks, it might be wise to revert to these version numbers.

Once the dependencies are set up, we'll proceed to create the frontend of our app using React.

Insert the following code into `src\App.js`:

```
// code/story-generator/src/app/page.js

export default function Home() {
  const onSubmitHandler = async (event)=> {
    event.preventDefault()
```

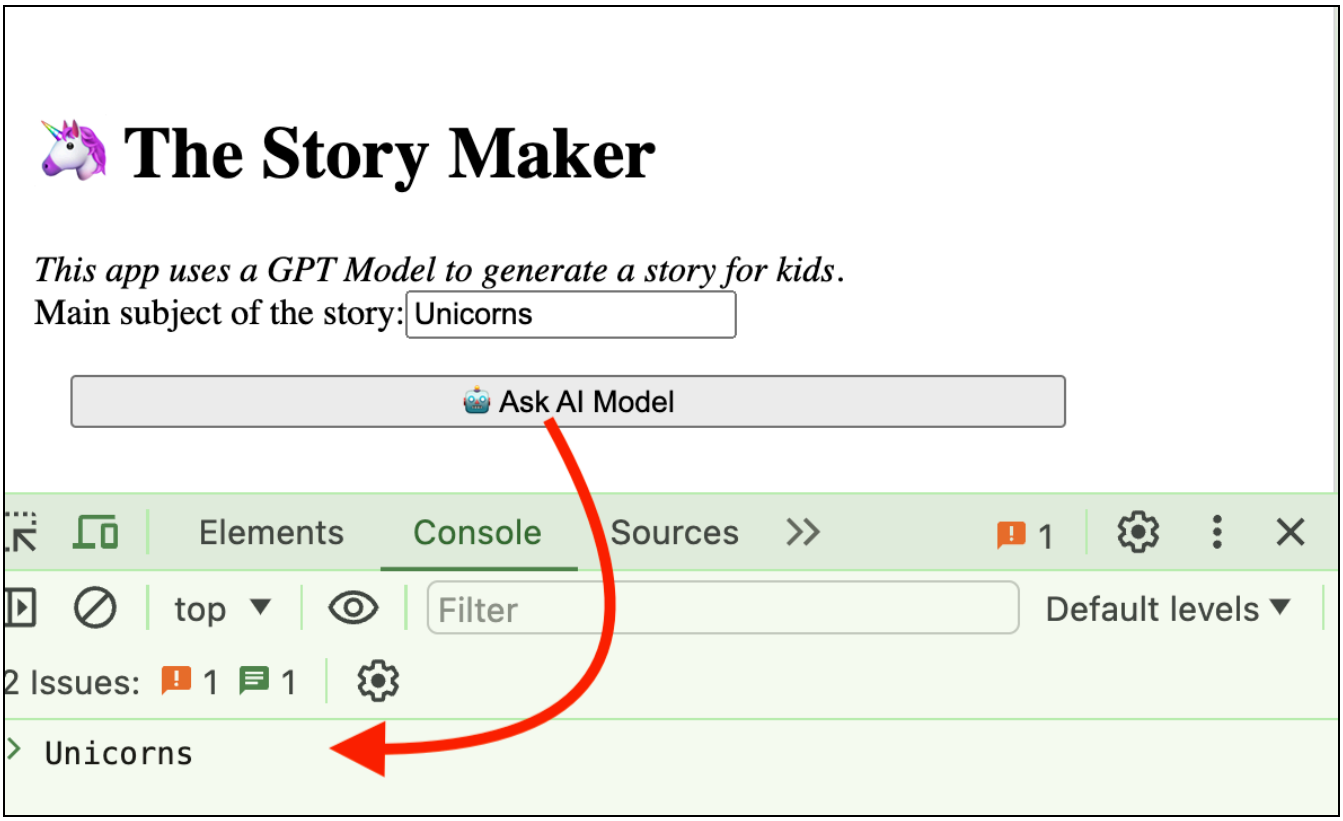
```
    const subject = event.target.subject.value
    console.log(subject)
  }

  return (
    <>
      <h1>🦄 The Story Maker</h1>
      <em>This app uses a GPT Model to generate a story for
kids.</em>
      <form onSubmit={onSubmitHandler}>
        <label htmlFor="subject">Main subject of the story:
</label>
        <input name='subject' placeholder='subject...' />
        <button>🤖 Ask AI Model</button>
      </form>
    </>
  )
}
```

Nothing particularly remarkable at this stage. It's just a React UI that fetches the subject of the story from a text input and logs it to the console.

We'll launch our app by executing the following command in the terminal:

```
npm run dev
```

This will generate the following output:



Currently, if the user enters a subject, no story will be generated as we haven't yet integrated the app with OpenAI. We'll address this in the next chapter.

2.3. The API key for the OpenAI ChatGPT model

To use OpenAI's ChatGPT model, you will need an API key. The API is subscription-based and the key can be generated from the following URL:

```
https://platform.openai.com/api-keys
```

You can top up your account with the minimum value. For example, writing all the examples in this book cost me under 2 USD.

Once you've generated your API key, return to the Next.js project and paste it into the `.env` file:

```
// code/story-generator/.env

OPENAI_API_KEY='sk-1234567890B2tT4HT3BlbkFJXkATN1arB9DXABkRQ1uj'
```

This key will be used only in the backend part of our app. To access the API key in the `code/story-generator/src/app/api/route.js` file we can write the following:

```
// code/story-generator/src/app/api/route.js

console.log( process.env.OPENAI_API_KEY )
```

The key will be later passed to the LangChain model object constructor.

This step will be the default procedure for each of the code examples in the following chapters of this book.
If you don’t wish to use an API key, you can check out this list](https://js.langchain.com/docs/modules/model_io/models/) with the LLMs supported by LangChain and pick up a free model. You can even use a free HuggingFace token to avoid the OpenAI API.

2.4. Using the ChatOpenAI LangChain model

Let's move forward by establishing a connection to OpenAI's LLM. Add the following code to the backend of the app, in the `src/app/api/route.js` file:

```
// code/story-generator/src/app/api/route.js
import { ChatOpenAI } from "@langchain/openai"

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

export async function POST(req) {
    const { subject } = await req.json()
    const gptResponse = await model.invoke(subject)
    return Response.json({data: gptResponse})
}
```

To initiate a connection with OpenAI, we create a new instance of the ChatOpenAI object. Within its constructor, we define the following:

- a temperature parameter, set to 0.9
- the API key obtained from the preceding subchapter

You might be curious about this temperature parameter, right?

The temperature parameter affects how imaginative our model gets. Increasing it results in more creative outputs while lowering it keeps responses more grounded and factual. Sticking to facts reduces potential inaccuracies or hallucinations from the model.

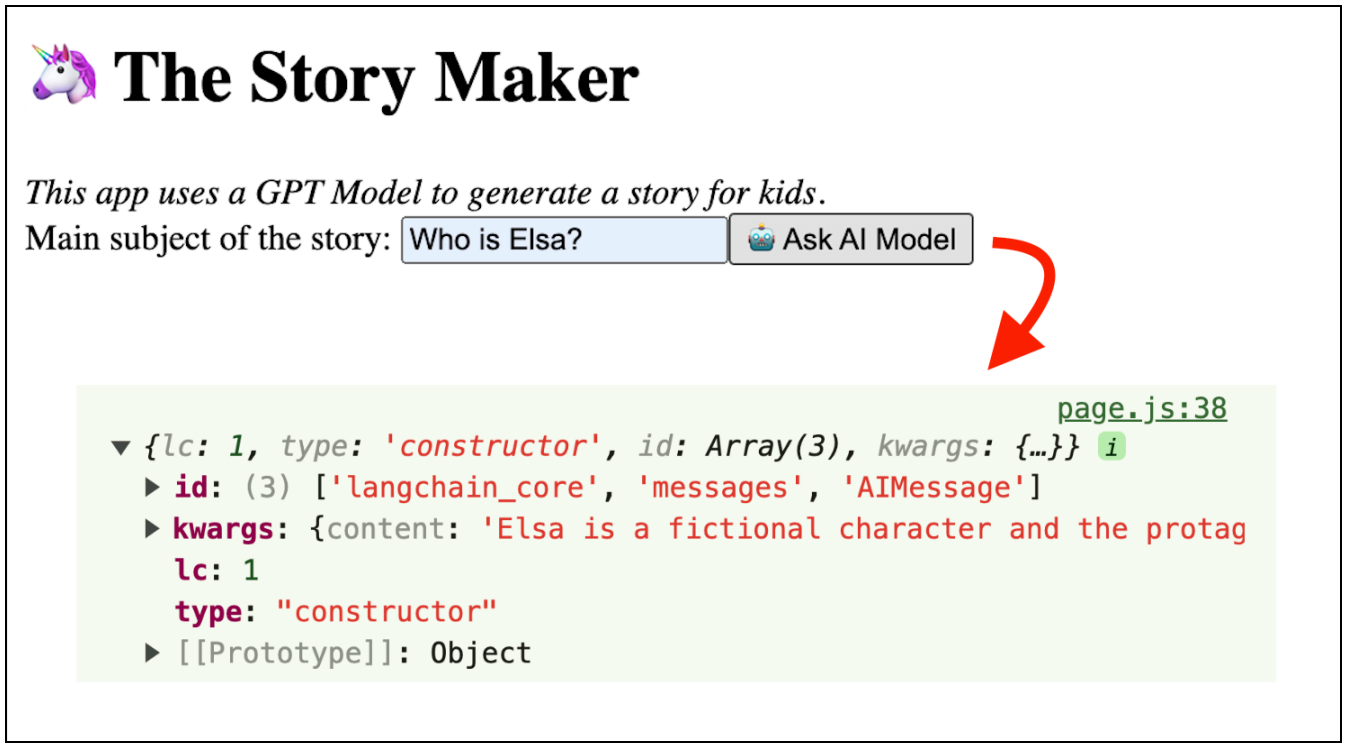
Choosing the right temperature depends on the task. For brainstorming catchy story titles, a higher temperature is preferable. For tasks where accuracy is key like summarizing a legal document, a lower temperature is better.

With our temperature set, let's now update the frontend of our app:

```
// code/story-generator/src/app/page.js
export default function Home() {
  const onSubmitHandler = async (event) => {
    event.preventDefault()
    const subject = event.target.subject.value
    //🟠 call the LLM with the subject as the main prompt
    const response = await fetch('api', {
      method: "POST",
      body: JSON.stringify({ subject })
    })
    //🟠 destructure and print out the response's data
    const {data} = await response.json()
    console.log(response)
  }

  // the rest remains the same
  // ...
}
```

To test, let's input a generic question like `Who is Elsa?` As an answer, we will receive:

```
"Elsa is a fictional character and the protagonist of Disney
animated film, Frozen. She is the queen of Arendelle and
possesses the magical ability to create ice and snow."
```



This confirms our successful connection to OpenAI.

Please note that your answer may be a bit different. Opposite to the classic way of programming, an LLM can give a different output to the same input. LLMs work by estimating stuff, it's all probabilistic.

Another test could involve requesting story a title such as:

```
"Tell me a bedside story title about unicorns."
```

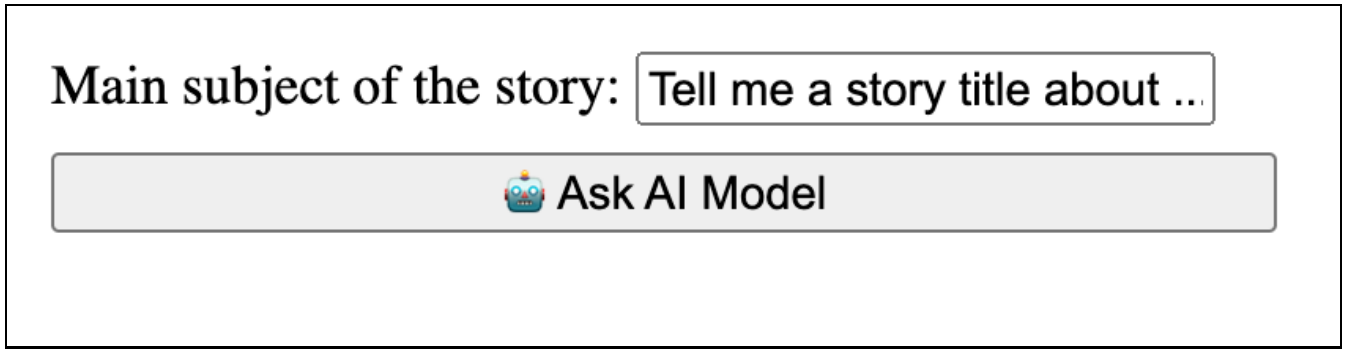
Or:

```
"Tell me a bedside story title about cats."
```

Give it a shot and see how it goes.

2.5. Prompt templates

You'll soon notice that the prompt requires repetitive input, such as `Tell me a bedside story title about ...` followed by the main subject.



This redundancy can be addressed using prompt templates, which we'll explore next.

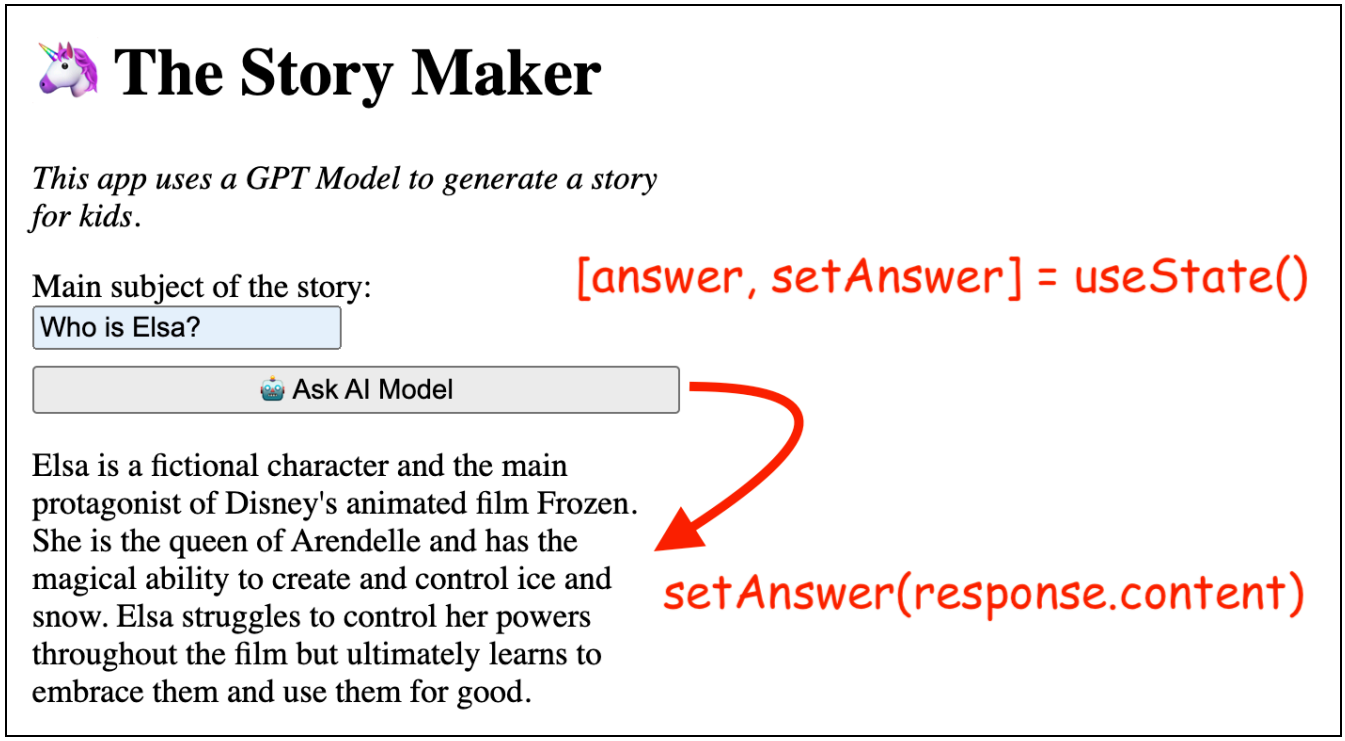
Before we move to the next chapter let's add a small tweak to our app. Will display the story title given by the LLM in the UI using a React state variable.

This is the full code with the highlighted additions:

```
//code/story-generator/src/app/page.js

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

export default function Home() {
  //🟠 adding the React state variable
  const [storyTitle, setStoryTitle] = useState()

  const onSubmitHandler = async (e)=> {
    e.preventDefault()
    const subject = e.target.subject.value
    const response = await fetch('api', {
      method: "POST",
      body: JSON.stringify({ subject })
    })
    const { data } = await response.json()
    //🟠 update the React state when we have the LLM's answer
    setStoryTitle(data)
```

```
  }

  return (
    <>
      <h1>🦄 The Story Maker</h1>
      <em>This app uses a GPT Model to generate a story for
kids.</em>
      <form onSubmit={onSubmitHandler}>
        <label htmlFor="subject">Main subject of the story:
</label>
        <input name='subject' placeholder='subject...' />
        <button>🤖 Ask AI Model</button>
      </form>
      {/*🟠 display the story title in the UI */}
      <p>{ storyTitle }</p>
    </>
  )
}
```

And this is how the UI will look after the change:



Let's now get back to the main content of this chapter: prompt templates.

Language models use text as input. That text is called a prompt. It's not just a fixed string, though. Usually, it's a mix of a template, examples, and what the user adds.

With templates, we can streamline the prompting process, making sure that users don't have to repetitively type the same prefix for each request.

For our particular use case, the prompt template will look like so:

```
`Tell me a story title about {subject}`
```

All the chances we need to make are in the backend:

```
//code/story-generator/src/app/api/route.js

import { ChatOpenAI } from "@langchain/openai"
//🟠 we will use the PromptTemplate to avoid repetition
import { PromptTemplate } from "@langchain/core/prompts"

const model = new ChatOpenAI({
  openAIApiKey: process.env.OPENAI_API_KEY,
  temperature: 0.9
})
//🟠 establishing the general format of the template
const prompt = new PromptTemplate({
    inputVariables: [ "subject"],
    template: "Tell me a story title about {subject}"
})

export async function POST(req) {
    const { subject } = await req.json()
    //🟠 passing down the { subject } parameter
    const formattedPrompt = await prompt.format({
        subject
    })
    const gptResponse = await model.invoke(formattedPrompt)
    return Response.json({data: gptResponse.content})
}
```

The default format of writing prompt templates is the F-String format, popular in Python. If you want to use the more familiar double curly braces JavaScript Mustache format you can have:

```
const prompt = PromptTemplate.fromTemplate(
  inputVariables: [ "subject"],
  template: "Tell me a story title about {{subject}}",
  {
    templateFormat: "mustache"
  }
)
```

If you want to read more you can take a look at the quick start guide from the documentation.

2.6. Chains

Chains refer to sequences of calls - whether to an LLM, a tool, or a data preprocessing step.

They are a vital core concept of LangChain and can be seen as short programs written around LLMs to execute complex tasks.

Speaking of chains, here's a fun fact about LangChain's logo: The parrot and a chain link were inspired by people calling LLMs "stochastic parrots". These models can mimic human writing but don't truly understand what they're saying. This library aims to "chain" these "parrots" together to produce more meaningful and useful outputs.

The output of one link from the chain becomes the input for the next link.

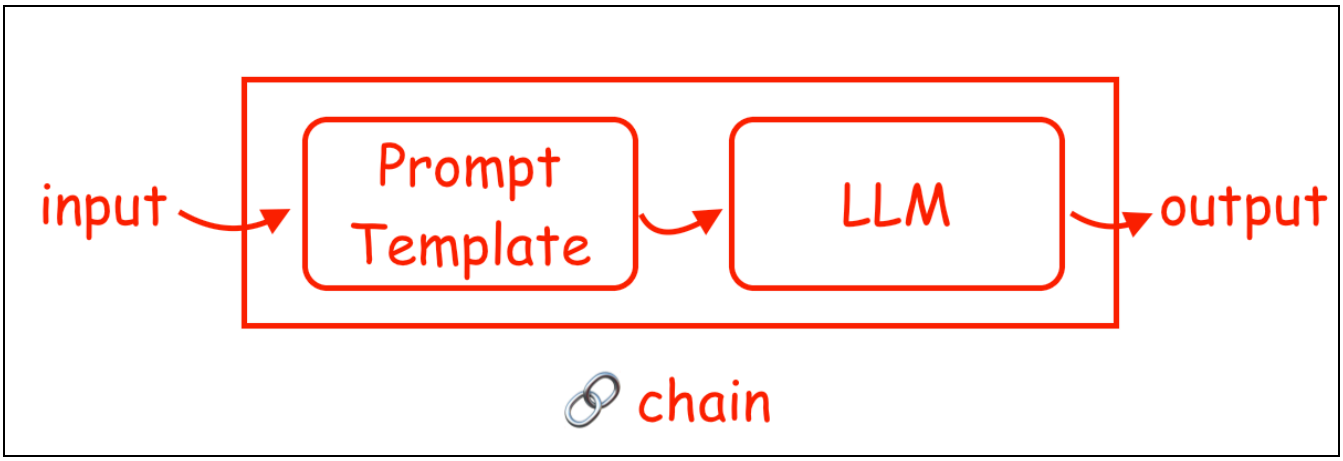
Let's say we need to create an article for a magazine. We will follow these steps:

1. Use ChatGPT to generate the article title.
2. Pass this title to a local LLM to compose the actual article, utilizing internal documents.

3. Pass the article to Midjourney, another LLM, to generate images for the article.

Let's see how to add a chain to our backend code:

```
import { ChatOpenAI } from "@langchain/openai"
import { PromptTemplate } from "@langchain/core/prompts"
//🟠 using the LLMChain; maybe the most straightforward chain
import { LLMChain } from "langchain/chains"

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

const prompt = new PromptTemplate({
    inputVariables: [ "subject"],
    template: "Tell me a story title about {subject}"
})

//🟠 a chain that links the model, the prompt and the verbose
option
const chain = new LLMChain({
    llm: model,
    prompt,
    //🟠 use verbose to debug the chain
    verbose: true
})

export async function POST(req) {
    const { subject } = await req.json()
    //🟠 invoking the chain
    const gptResponse = await chain.invoke({subject})
    return Response.json({data: gptResponse.text})
}
```



You may encounter some examples online that utilize the `chain.call()` method instead of `chain.invoke()`. Please note that `.call()` is deprecated and will be removed.

The `verbose: true` option is optional. If enabled, it will provide additional details about the chain in the backend console. This information includes:

- the input and output for each step of the chain
- how long each step took
- the `tokenUsage` variable

This last point is important because we pay to use the LLM's API with tokens. On average, 100 words are roughly 130 tokens. A helpful tool for managing tokens is the OpenAI Tokenizer. The `tokenUsage` is divided into how much the input prompt used and how much the actual response used:

```
"llmOutput": {
  "tokenUsage": {
    "completionTokens": 14,
    "promptTokens": 14,
    "totalTokens": 28
  }
}
```

In the frontend code, we will simply replace the text input with a select box to facilitate data entry:

```
export default function Home() {
```

```
  // the rest remains the same
  // ...

  return (
    <>
      <h1>🦄 The Story Maker</h1>
      <em>This app uses a GPT Model to generate a story for
kids.</em>
      {/*🟠 adding a select box to make the data entry faster
*/}
      <form onSubmit={onSubmitHandler}>
        Main subject of the story:
        <select name="subject">
            <option value="cats">😸 Cats</option>
            <option value="unicorns">🦄 Unicorns</option>
            <option value="elfs">👨‍♀️ Elfs</option>
        </select>
        <button>🤖 Ask AI Model</button>
      </form>
      <p>{ storyTitle }</p>
    </>
  )
}
```

2.7. LCEL - LangChain Expression Language

The LangChain Expression Language, or LCEL, provides a simpler way to chain multiple custom components.

In short, using LCEL, we can replace other classes, such as `LLMChain`, with the `.pipe()` method to form a chain.

Let's take another look at the chain for the above example:

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

const prompt = new PromptTemplate({
    inputVariables: ["subject"],
    template: "Tell me a story title about {subject}"
})

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

await chain.invoke({subject})
```

We can replace the `LLMChain` wrapper by using the `.pipe()` method. The rest of the code remains the same:

```
// BEFORE
const chain = new LLMChain({
    llm: model,
    prompt,
})
await chain.invoke({subject})

// AFTER
const chain = prompt.pipe(model)
await chain.invoke({subject})
```

Later, we will see that we can build more complex chains by adding other types of nodes, such as output parsers.

The output of a node in the chain becomes the input of the next node. The key element of LCEL is the RunnableInterface. It ensures that the nodes in a chain have a common communication pattern.

LCEL standardizes communication and enables easy swapping of components within the LangChain framework. All the components are modular. If something needs to be changed, only that part of the code has to be replaced rather than the entire pipeline.

It also makes the code cleaner.

Many of the examples in this book will be made using LCEL chains.

If you want to explore more on this topic, be sure to check out the official LCEL docs and I've also written a few articles on my blog:

- Nested Chains in LangChain JS – Passing the Output from One Chain to Another
- Custom Functions with RunnableLambda in LangChain JS
- RunnableMap and RunnableParallel in LangChain JS
- Routing in LangChain JS – Use LLM Classifiers to Call Different Prompts
- Using RunnableBranch to Route Execution to Different Prompts in LangChain.js

This concludes our first small app with JavaScript, LangChain, and ChatGPT. We have learned how to:

- bootstrap and connect a web app app to an LLM.
- utilize LangChain's prompt templates to prevent repetition
- and create our first chain.

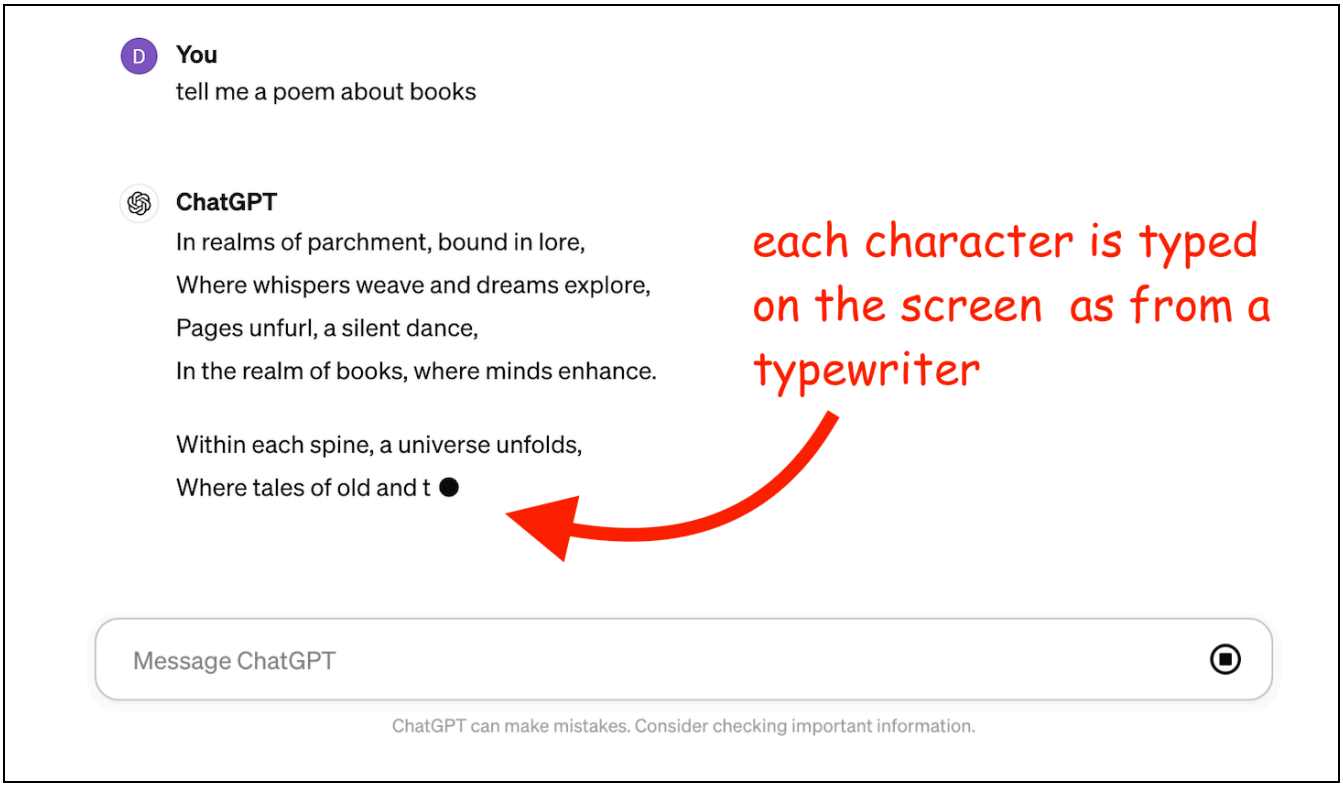
In the next chapter, we will expand this application to generate the actual story body and integrate data streams.

3. Streams in LangChain

3.1. Introduction and project setup

Some answers may take a lot of time to be generated by an LLM. For instance, if you request the model to craft a complex response like a poem or summarize a document.

Given the way LLMs work, we don't have to wait for the full answer to be complete. Have you noticed that when you ask ChatGPT something you will see that typewriter effect, wherein each word appears gradually on the screen?



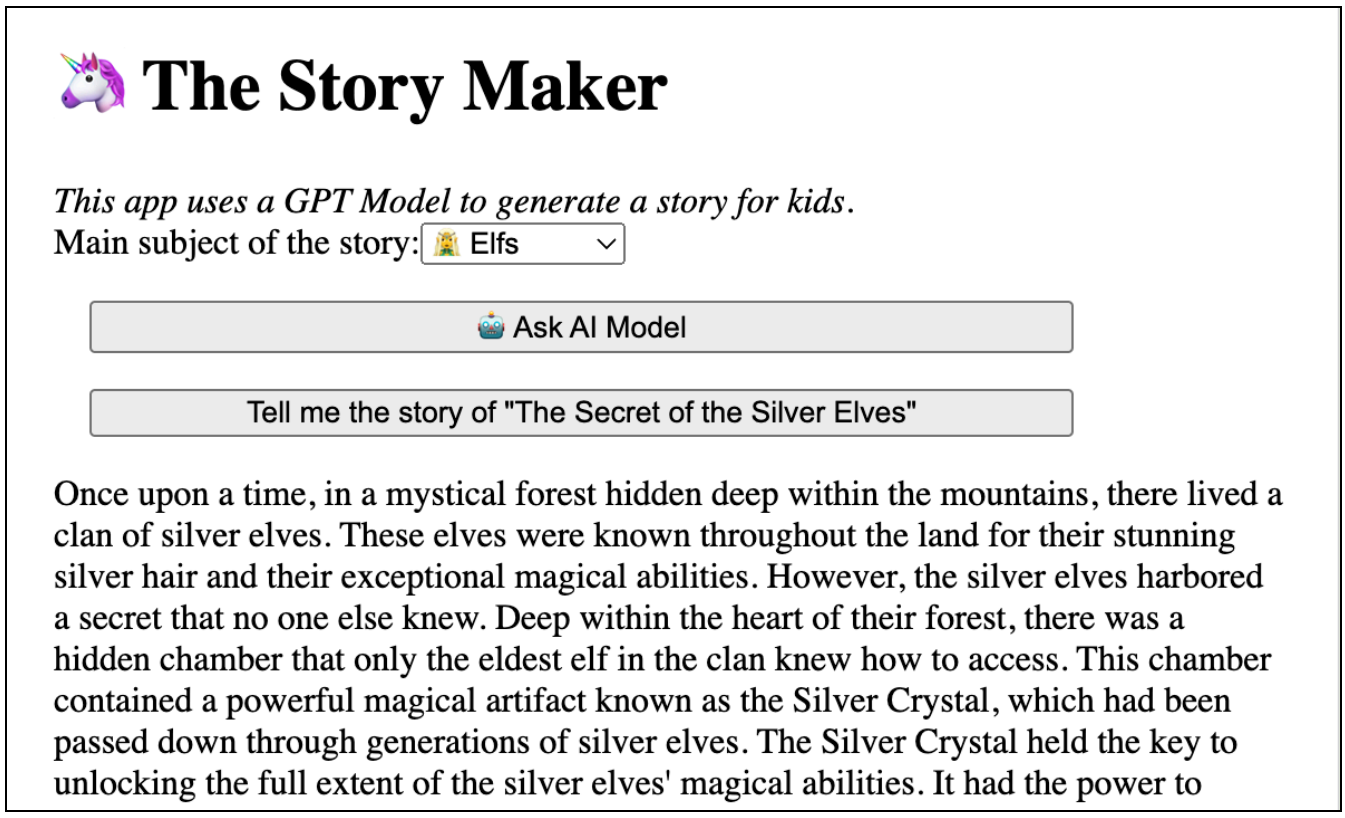
A Large Language Model does not reason like a human. It does not first build the full response and then explain it. Nor does it map the answer to a knowledge graph or something similar. It just adds one token at a time to the answer.

This is great because we can use this streaming behavior to update the UI as soon as we have new info to show to the user.

Returning to the story generator example, let's say we want to make a story named `The Secret of the Silver Elves`. Instead of waiting for the full story to be generated, we can stream the story while it is written:

```
Once upon a time
...
...
Once upon a time, in a mystical forest
...
...
Once upon a time, in a mystical forest hidden deep within the
mountains
...
...
Once upon a time, in a mystical forest hidden deep within the
mountains, there lived a clan of silver elves.
```

In this chapter, we will implement the streaming feature. This is how the UI of the app will look like at the end:

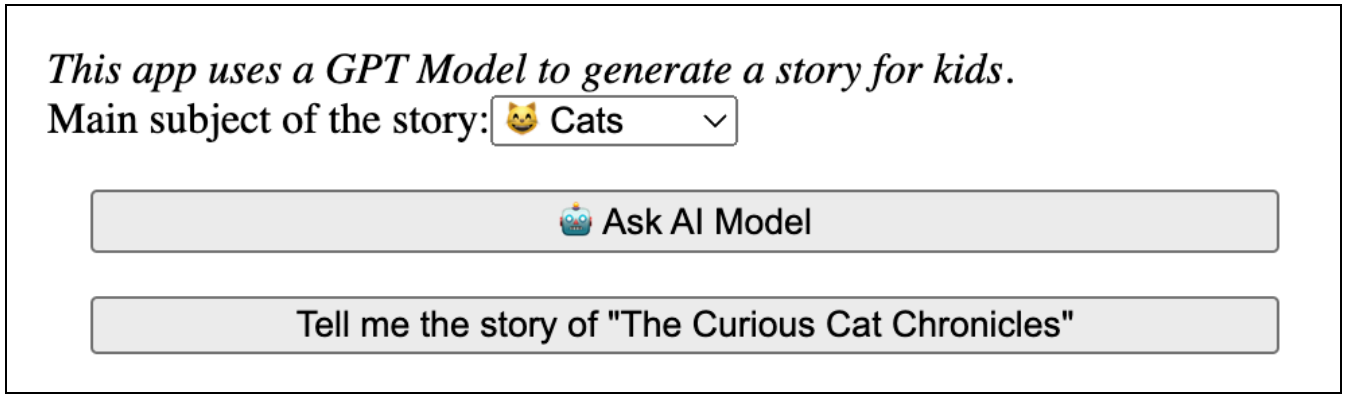


3.2. Project refactor

Before we move forward, we first need to do a bit of refactoring to our project.

On the frontend, we will do the following:

1. once we have a title for the story, we will display it as a button with the text `Tell me the story of <<story name here>>`
2. when this button is pressed, we will make a call to the backend, pass in the story name and print the response



This is how the code will look like:

```
// code/streaming/src/app/page.js

'use client'
import { useState } from "react"
export default function Home() {
  const [storyTitle, setStoryTitle] = useState()
  //🟠 a new state variable for the storyBody
  const [storyBody, setStoryBody] = useState()

  const onSubmitHandler = async (e)=> {
    e.preventDefault()
    const subject = e.target.subject.value
    const response = await fetch('api', {
      method: "POST",
      body: JSON.stringify({ subject })
    })
```

```
    const { data } = await response.json()
    setStoryTitle(data)
  }

  //🟠 call this method when the start story button is pressed
  const startStoryStream = async ()=> {
    //🟠 reset the story body, call the backend
    // and display the response in the story body
    setStoryBody('')
    const response = await fetch('api', {
      method: "POST",
      body: JSON.stringify({ storyTitle })
    })
    const { data } = await response.json()
    setStoryBody(data)
  }

  return (
    <>
      <h1>🦄 The Story Maker</h1>
      <em>This app uses a GPT Model to generate a story for
kids.</em>
      <form onSubmit={onSubmitHandler}>
        Main subject of the story:
        <select name="subject">
            <option value="cats">😸 Cats</option>
            <option value="unicorns">🦄 Unicorns</option>
            <option value="elfs">👨‍♀️ Elfs</option>
        </select>
        <button>🤖 Ask AI Model</button>
      </form>
      {/* 🟠 if we have the story title show
      the start story button and the story body */}
      {storyTitle && <div>
        <button onClick={startStoryStream}>
          Tell me the story of {storyTitle}
        </button>
        <p>{storyBody}</p>
```

```
      </div>}
    </>
  )
}
```

On the backend, we will add these changes:

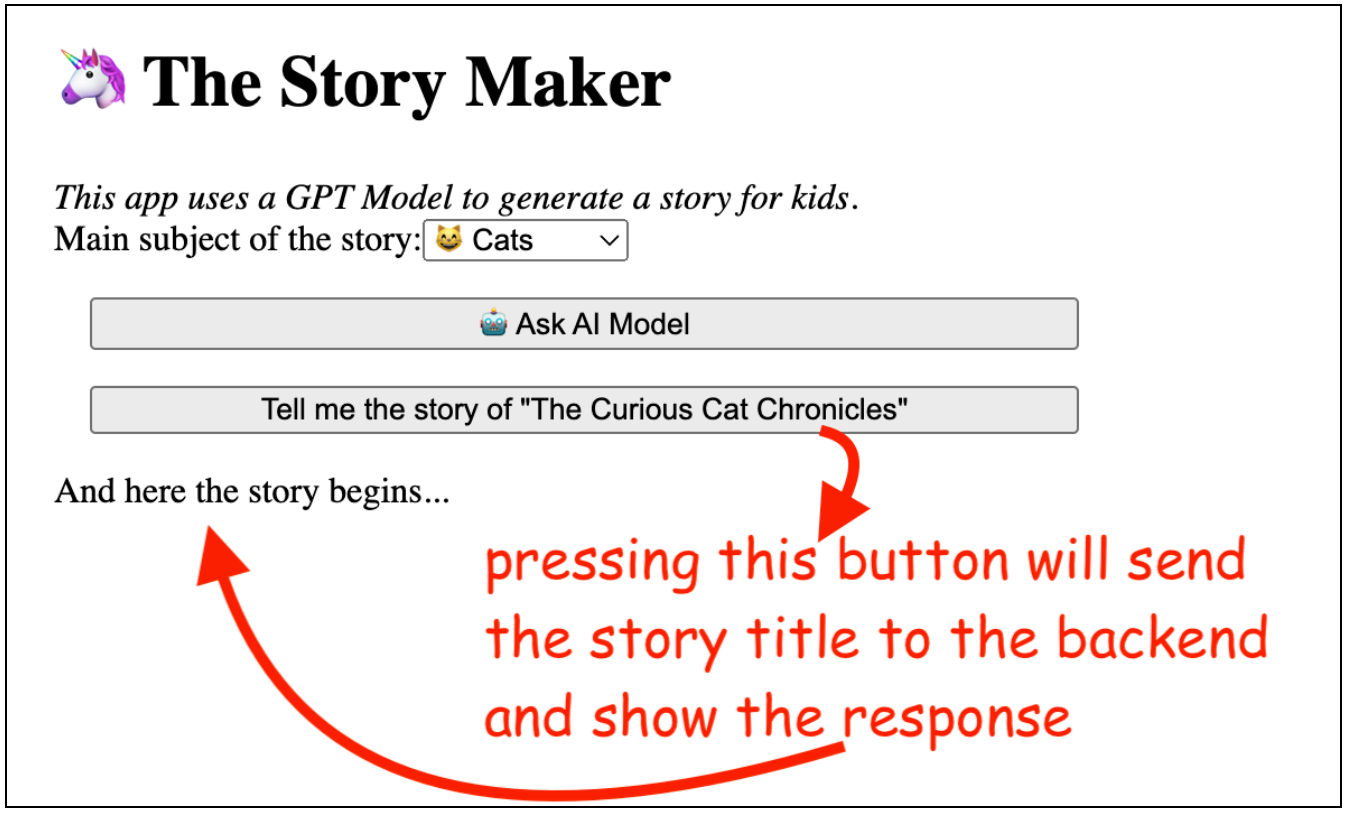
- add a `makeStoryTitle()` method that will contain all the previous logic
- add a `streamStory()` method. We will later use this method to implement the streaming mechanism. For the time being, it will return some static text
- check if the request from the frontend is asking for a story title or for the story body. Call one of the above methods.

Here is how the backend code will look:

```
import { ChatOpenAI } from "@langchain/openai"
import { PromptTemplate } from "@langchain/core/prompts"
//🟠 we will use the NextResponse to send the story body
import { NextResponse } from "next/server"

//🟠 all the previous logic goes in here
const makeStoryTitle = async (subject)=> {
    const model = new ChatOpenAI({
        openAIApiKey: process.env.OPENAI_API_KEY,
        temperature: 0.9
    })
    const prompt = new PromptTemplate({
        inputVariables: [ "subject"],
        template: "Tell me a story title about {subject}",
    })
    const chain = prompt.pipe(model)
    return await chain.invoke({subject})
}

//🟠 we will later update this to add the streaming
const streamStory = async (storyTitle)=> {
```

```
    return new NextResponse(
        JSON.stringify( { data: "And here the story begins..."}
),
        { headers: { 'content-type': 'application/json' },
    })
}

export async function POST(req) {
    const { subject, storyTitle } = await req.json()
    //🟠 if the request has a storyTitle then return the
streamStory()
    if (storyTitle) {
        return streamStory(storyTitle)
    }
    //🟠 if we don't have a storyTitle the return one
    const gptResponse = await makeStoryTitle(subject)
    return Response.json({data: gptResponse.text})
}
```

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



3.3. Streaming the response

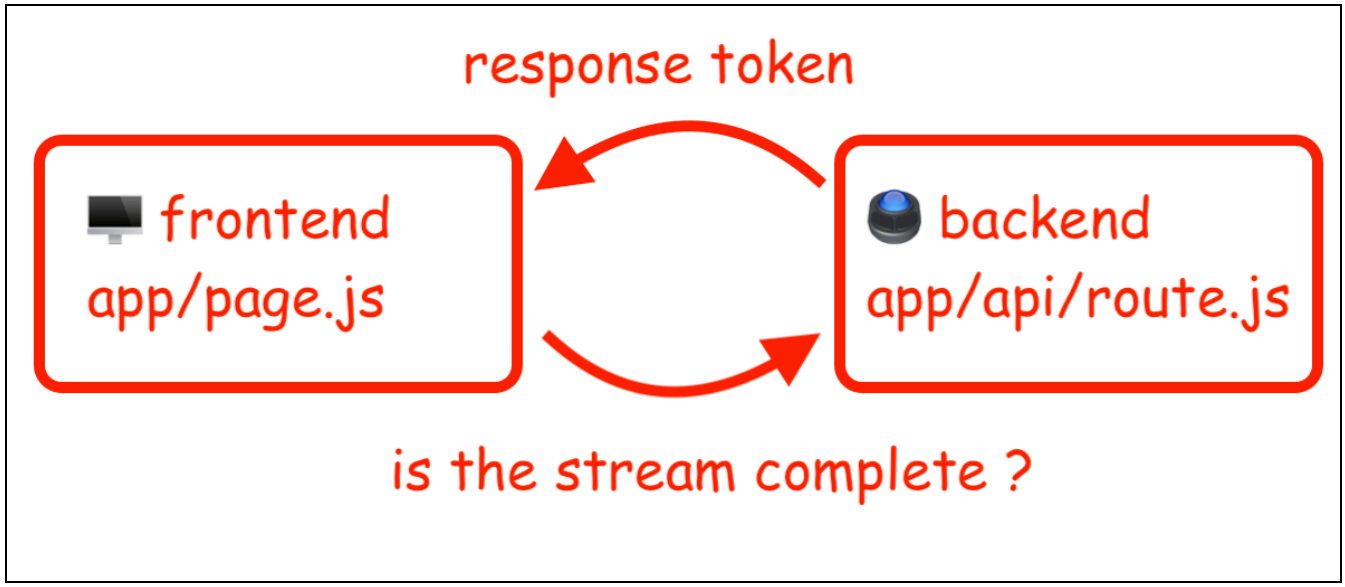
In some cases, AI models can take several seconds to generate a complete response to a prompt. This is slower than the ~200-300 ms threshold at which an app feels responsive to the end user.

To make the app feel more responsive we need to show intermediate progress and stream the final output in chunks, yielding each chunk as soon as it is available.

Not all models support streaming, so be sure to check if the model you use supports this feature.

It's time to add the streaming feature to our story generator app.

Note that this will not be a single request-response interaction. It will be more like an ongoing chat. The frontend will keep requesting new LLM response chunks and update the ongoing result, while the backend sends back response chunks or indicates when the stream is finished.



We will start with the backend. All the changes will be made in the `streamStory()` method we have defined during the previous step:

```
// code/streaming/src/app/api/route.js

// the rest remains the same
```

```
// ...

const streamStory = async (storyTitle)=> {
    const encoder = new TextEncoder()
    const stream = new TransformStream()
    const writer = stream.writable.getWriter()
    const model = new ChatOpenAI({
        openAIApiKey: process.env.OPENAI_API_KEY,
        temperature: 0.9,
        streaming: true,
        callbacks: [{
            handleLLMNewToken: async (token) => {
              await writer.ready
              await writer.write(encoder.encode(`${token}`))
            },
            handleLLMEnd: async () => {
              await writer.ready
              await writer.close()
            }
        }]
    })
    const prompt = new PromptTemplate({
        inputVariables: [ "storyTitle" ],
        template: "Tell me story titled {storyTitle}"
    })
    const chain = prompt.pipe(model)
    chain.invoke({storyTitle})
    return new NextResponse(stream.readable, {
        headers: { "Content-Type": "text/event-stream" }
    })
}

// the rest remains the same
// ...
```

First, we will need to create a new chain that generates the story body.

The model instance passed to this chain has two updates:

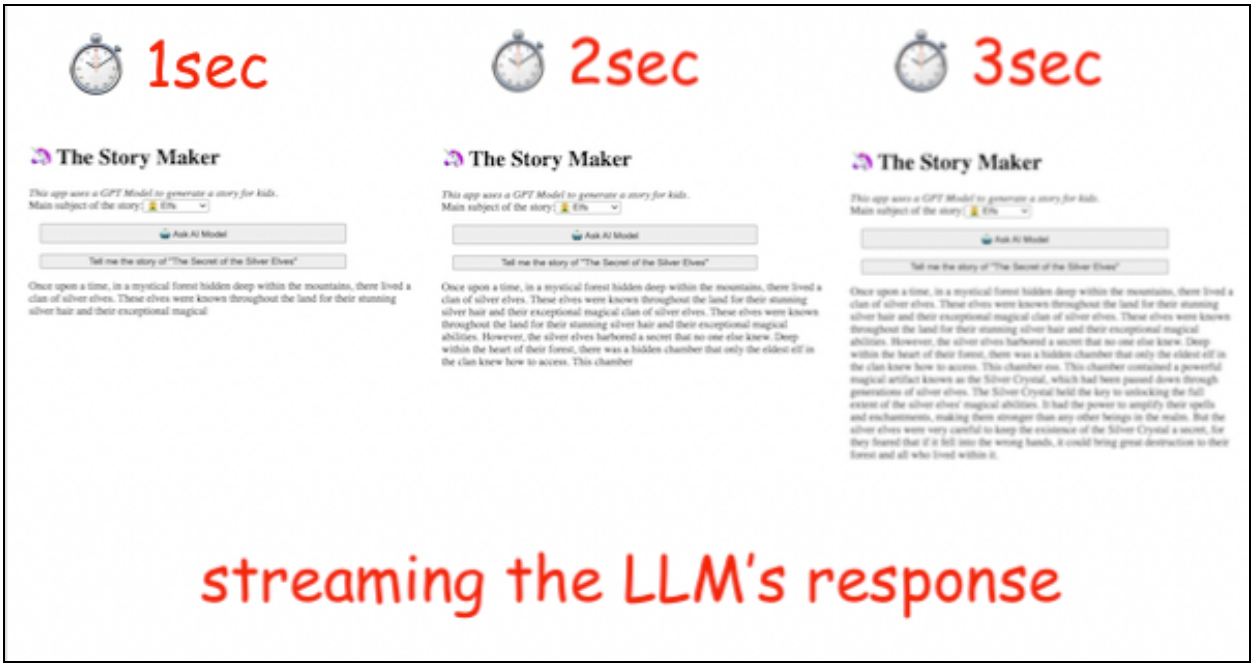
- the `streaming` flag is enabled
- the `handleLLMNewToken()` and `handleLLMEnd()` callbacks implemented.

To keep the communication channel open after the initial request, the returned response content type is set to `text/event-stream` instead of the standard `application/json`.

Additionally, on the frontend, we will modify a single method: `startStoryStream()`:

```
// code/streaming/src/app/page.js

// the rest remains the same
// ...

const startStoryStream = async ()=> {
  setStoryBody('')
  const response = await fetch('api', {
    method: "POST",
    body: JSON.stringify({ storyTitle })
  })
  //🟠 the reader will act as a data communication "pipe"
  const reader = response.body.getReader()
  const decoder = new TextDecoder()
  //🟠 keep the connection while we keep receiving new response
chunks
  while (true) {
    const { value, done } = await reader.read()
    if (done) break
    const chunkValue = decoder.decode(value)
    //🟠 the way to append text to a React state string value
    setStoryBody(prev => prev + chunkValue)
  }
}

// the rest remains the same
// ...
```

With these final UI changes, we can watch the story unfold as it's generated by the LLM:



If you want to take a more in-depth look at this topic the LangChain documentation has a nice guide about streaming, including this part on how to stop the streaming process if the response is not as you intended.

4. Output parsers

4.1. Introduction

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 you ask the same question twice. Also, the format of the output can vary as well.

For example, if we ask for the recipe for a dish, sometimes the LLM will return Ingredients as the first step, sometimes Preparations. This is normal as LLMs are generative, not deterministic. This variability may be fine for a human reader, but it becomes problematic if we want an API endpoint to process those responses.

And when you change the type of underlying LLM model it will only get worse.

Think of output parsers as your formatting sidekicks. They can:

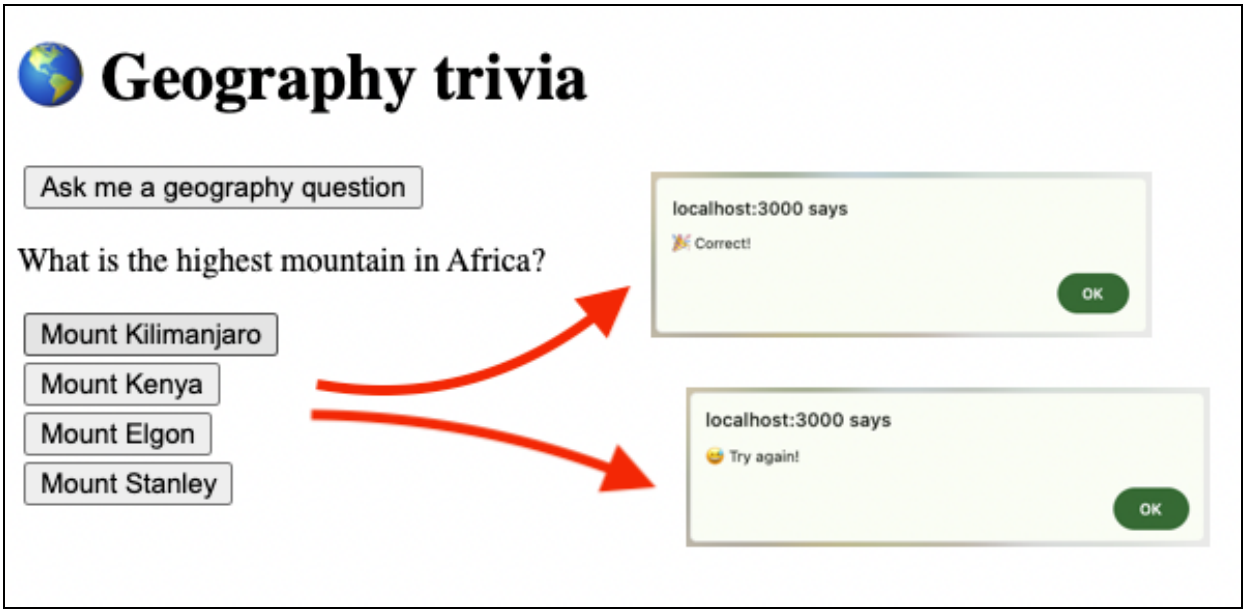
1. tweak your prompt by adding things such as prefixes or suffixes to steer the output in a certain direction. Want a JavaScript array? Want a simple string or JSON nested structure? LangChain's output parsers got you covered;
2. after the output is ready, a JavaScript function may step in to tidy things up. This function might even use other LLMs to polish the output into the perfect format.

4.2. Example setup

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

You were tasked with making a trivia game using ChatGPT. The user will ask for a trivia question with four possible answers. Only one of these answers is correct. Once a possible answer is clicked, a popup indicating whether 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:



We will start from a very simple state.

On the backend, we will make 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"

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

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

const chain = prompt.pipe(model)

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

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

On the frontend, we have a simple button that makes the call to the backend. 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"

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

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

const chain = new prompt.pipe(model)

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

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

4.3. StringOutputParser

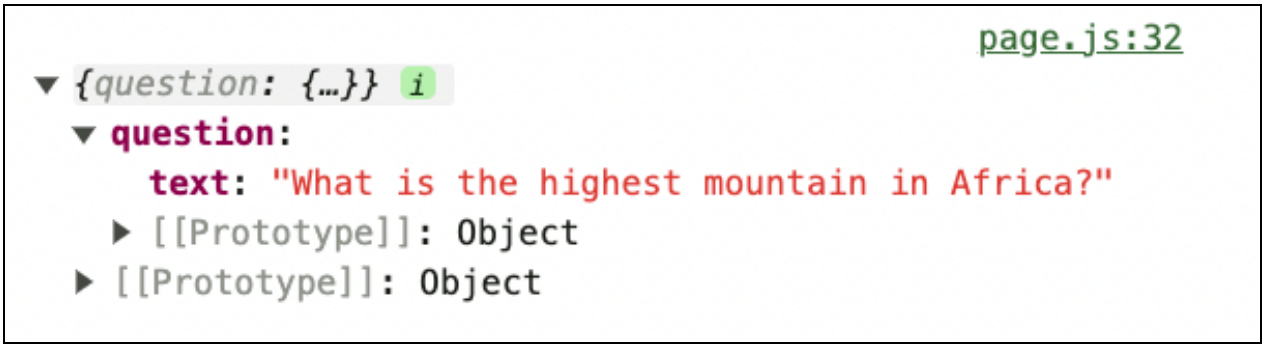
The output parsers from LangChain.js are meant to convert the AI responses into common structures, like JSON, Arrays, String, and more.

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 = prompt.pipe(model)

const gptResponse = await chain.invoke()

console.log(gptResponse)
```

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



This is the structure that comes from a ChatGPT model without any added output parsers.

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

This means the line that extracts the final string response may fail because the `text` property might 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 ensure the various response types from different LLMs are transformed into a simple string.

Let's see how this looks in the 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 LCEL 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 add an output parser to the chain, we don't need to manually parse the `.text` property:

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

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

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.

4.4. 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:

```
`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 from ChatGPT will look like this:

```
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 into more detail about the `AIMessage` class during the chat memory chapter.

For situations like these, LangChain can help us with the CommaSeparatedListOutputParser.

The `CommaSeparatedListOutputParser` takes a complex answer given by an LLM and reformats 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 after we add this `CommaSeparatedListOutputParser` to handle 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 use 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 AI models.

At this point, we are using two output parsers:

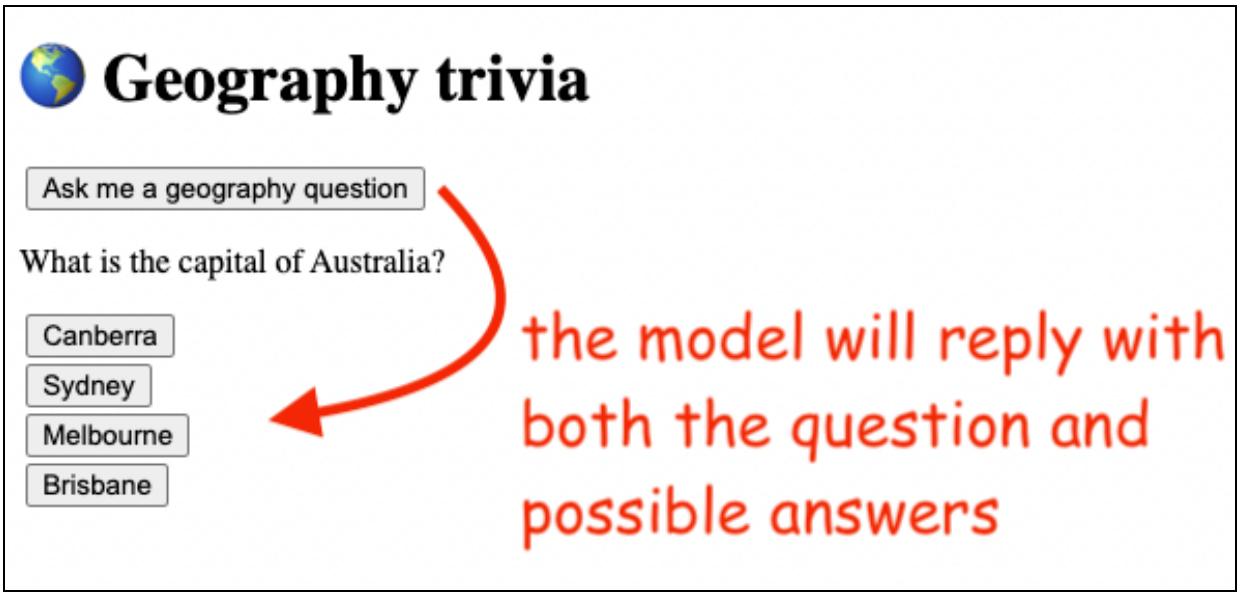
- the `StringOutputParser` for the main trivia question
- the `CommaSeparatedListOutputParser` to parse the list of possible answers

On the frontend, we will read the possible answers and 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>
      )}
    </>)
}
```

This is how the UI of the app will look 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 determine if the user clicked the correct answer.

4.5. The getFormatInstructions() method and custom parsers

Before we continue, let's make a short detour to 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 the `getFormatInstructions()` method in common.

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.

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

```
const commaListOutputParser = 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:
commaListOutputParser.getFormatInstructions()
})
```

If we print `commaListOutputParser.getFormatInstructions()` we will get the following:

```
Your response should be a list of comma separated values, eg:
`foo, bar, baz`
```

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

This will do the same as adding the parser directly to the chain, and 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 a specific structure.

4.6. StructuredOutputParser

Large Language Models like to have 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 other types of parsers such as StructuredOutputParser or JsonOutputParser.

For 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 though 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 are returning a single JSON structure that contains all the info our app needs.

Also, the `makeQuestion()` and `makePossibleAnswers()` methods are now merged into a single call, making the app a bit 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:

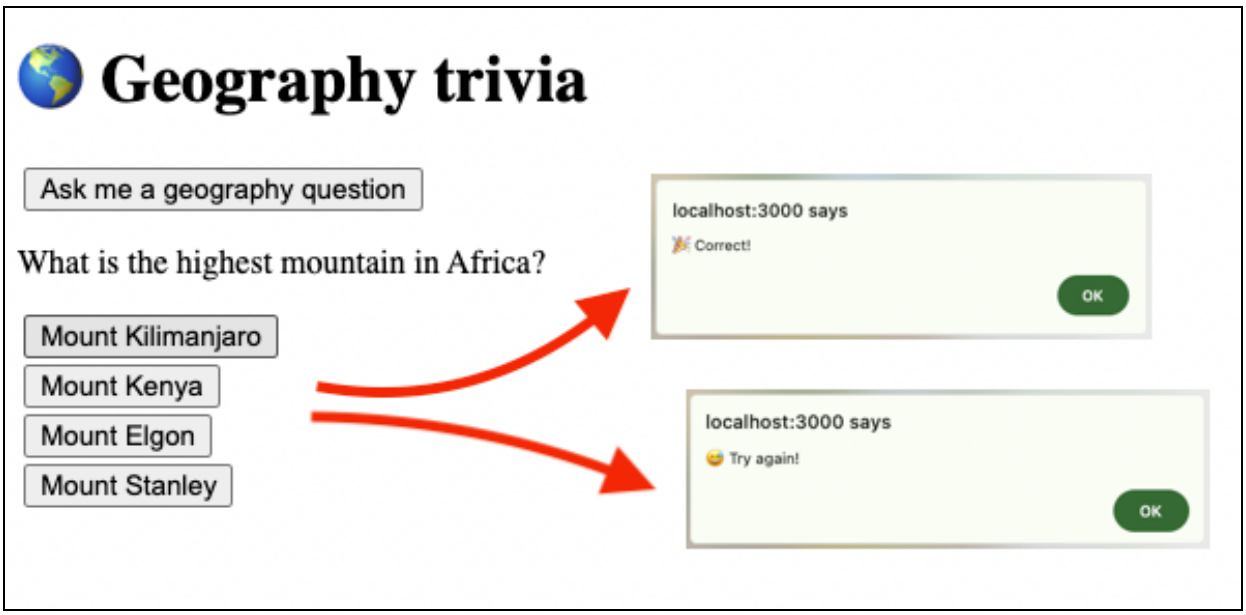
```
// code/trivia-game/src/app/page.js
```

```
'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 😎!

5. Chat memory

5.1. Introduction and project setup

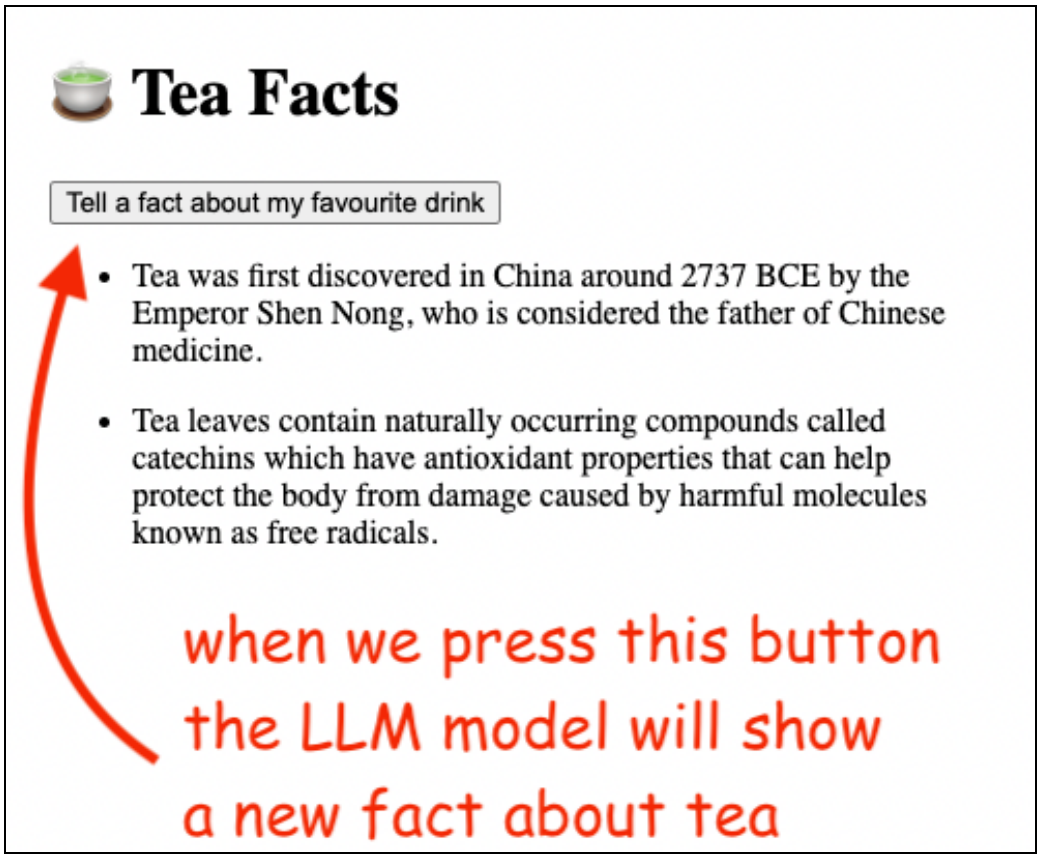
The chat memory feature allows a Large Language Model (LLM) to remember previous interactions with the user.

Conversational memory is how a chatbot can respond to multiple queries in a chat-like manner. Without it, every query would be treated as a single independent input without considering past interactions.

Follow-up questions only make sense when they reference the previous questions and answers. For example, if I just say to an LLM `Explain in detail` it will ask for more context. But if I first tell the same LLM `How the speed of light was calculated` followed by `Explain in detail` I will get a coherent answer.

By default, LLMs are stateless. Each incoming query is processed independently of other interactions. The only thing that exists for a stateless entity is the current input and nothing else.

You are working with a tea shop and have been tasked to make a GPT-powered app that provides users with interesting facts about tea. Below is a picture of what the final version of the app should look like:



We will start with the following setup.

On the frontend, we have a button that makes a POST request to the API and prints out the answer:

```
// code/tea-wiki/src/app/page.js
'use client'
import { useState } from "react"

export default function Home() {
  const [answer, setAnswer] = useState()
  const tellFact = async ()=> {
    const response = await fetch('api', {
      method: "POST"
    })
    const { data } = await response.json()
    setAnswer(data)
  }

  return (
    <>
```

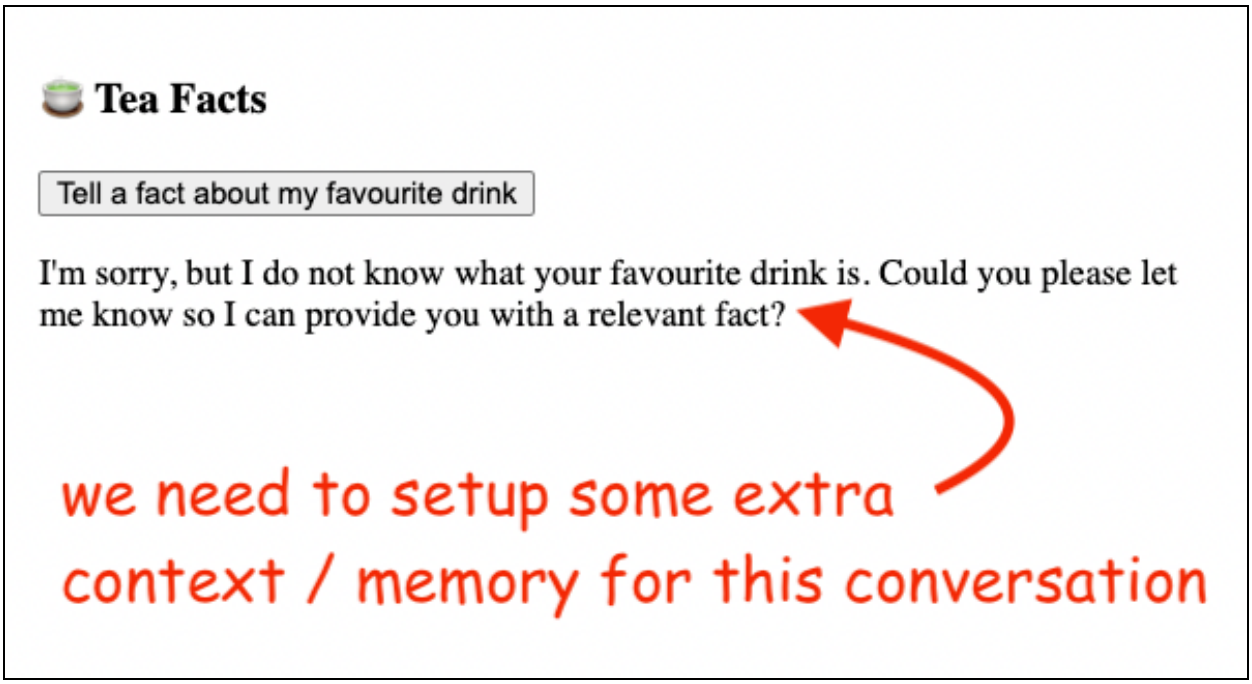
```
      <h1>🍵 Tea Facts</h1>
      <button onClick={tellFact}>
        Tell a fact about my favourite drink
      </button>
      <div>{answer}</div>
    </>
  )
}
```

While on the backend, we return the answer given by a GPT model to the prompt `Tell me a fact about my favorite drink`:

```
// code/tea-wiki/src/app/api/route.js
import { ChatOpenAI } from "@langchain/openai"
import { ChatPromptTemplate } from "@langchain/core/prompts"
import { StringOutputParser } from
"@langchain/core/output_parsers"

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

const prompt = ChatPromptTemplate.fromMessages([
    ["human", "{input}"]
])

const outputParser = new StringOutputParser()

const chain = prompt.pipe(model).pipe(outputParser)

export async function POST() {
    const question = `Tell me a fact about my favorite drink.`
    const fact = await chain.invoke({
        input: question,
    })
    return Response.json({data: fact})
}
```

5.2. Injecting messages into the conversation memory

Right from the gecko it's clear that we have a problem:



When invoking a chain we can use the `chat_history` property to set up a memory context for the conversation at hand.

Let's see how we can do this!

Below is the updated code for the backend:

```
// code/tea-wiki/src/app/api/route.js
import { ChatOpenAI } from "@langchain/openai"
import { ChatPromptTemplate } from "@langchain/core/prompts"
import { StringOutputParser } from
"@langchain/core/output_parsers"
//🟠 add the HumanMessage, and MessagesPlaceholder imports
import { HumanMessage } from "@langchain/core/messages"
import { MessagesPlaceholder } from "@langchain/core/prompts"

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

```
//🟠 the initial conversation memory context
const chatHistory = [
    new HumanMessage(`My favorite drink is tea.`)
]

const prompt = ChatPromptTemplate.fromMessages([
    //🟠 we need to tell to the promt that it will have a memory
    new MessagesPlaceholder("chat_history"),
    ["human", "{input}"]
])

const outputParser = new StringOutputParser()

const chain = prompt.pipe(model).pipe(outputParser)

export async function POST() {
    const question = `Tell me a fact about my favorite drink.`
    const fact = await chain.invoke({
        input: question,
        //🟠 link the chatHistory array with the prompt
        chat_history: chatHistory
    })
    return Response.json({data: fact})
}
```

The key here is the `MessagesPlaceholder` object. We can set up different message placeholders for a prompt.

For example, we can use the agent scratchpad message placeholders to store the "thoughts" of an AI agent. Don't worry, we will talk more about AI agents in the next chapters.

In this case, we will use the `MessagesPlaceholder` with a `chat_history` to indicate to the prompt that it's not stateless and has a previous history.

The `chatHistory` is just a simple array where we can store the conversation history with `HumanMessage` and `AIMessage` objects. In the end, the content of these objects is made of just simple strings with some extra formatting.

With these changes, the model will now know that our favorite drink is tea:

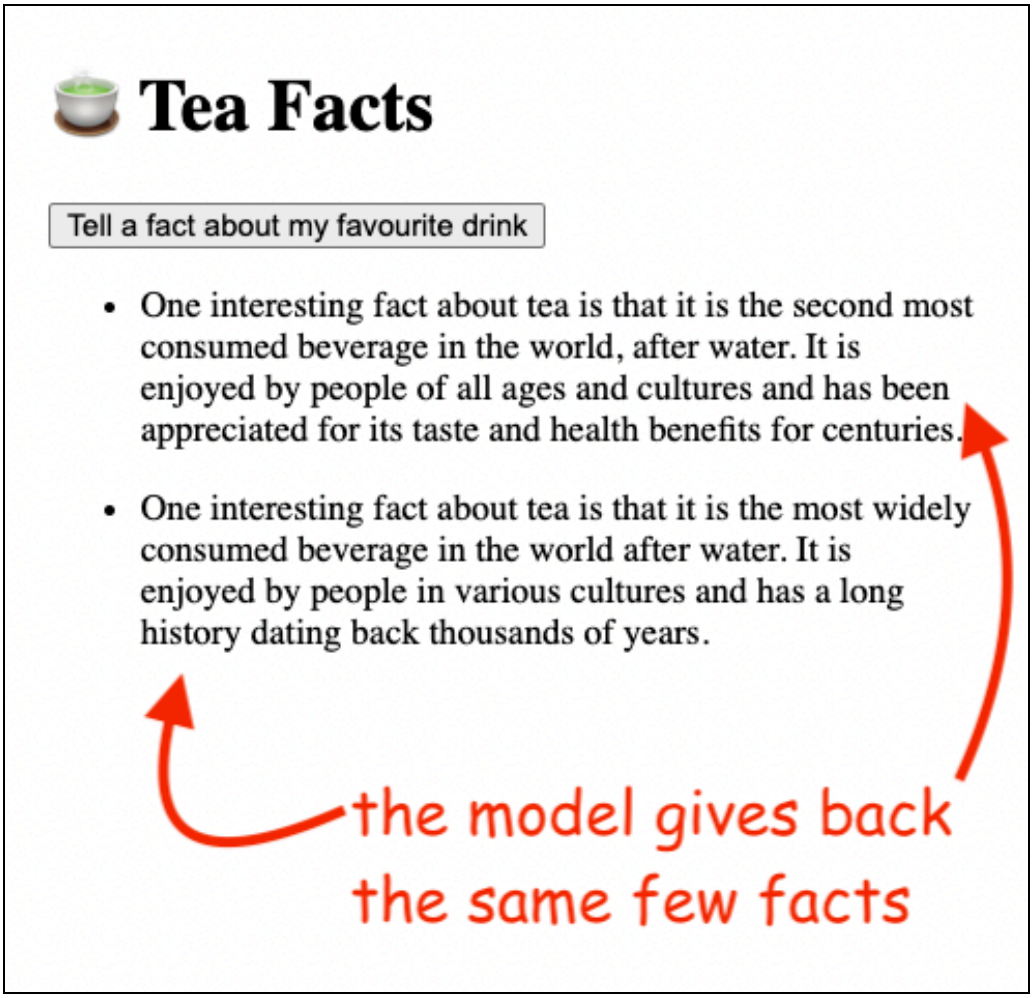
Great! One step forward!

However, we now face another issue. Let's see what happens after we request multiple facts.

To implement storing the previous facts we only change the frontend. We will track and show all the previous responses given by the model:

```
// code/tea-wiki/src/app/page.js
'use client'
import { useState } from "react"

export default function Home() {
  //🟠 replace the answer with an array of facts
  const [facts, setFacts] = useState([])
  const tellFact = async ()=> {
    const response = await fetch('api', {
      method: "POST"
    })
    const { data } = await response.json()
    //🟠 add the new fact to the array
    setFacts([data, ...facts])
  }

  return (
    <>
```

```
      <h1>🍵 Tea Facts</h1>
      <button onClick={tellFact}>
        Tell a fact about my favorite drink
      </button>
      <ul>
        {
          //🟠 display the array
          facts.map( (fact,i) => <li key={i}>{fact}</li>)
        }
      </ul>
    </>
  )
}
```



The model keeps giving back the same old few facts, like an old grandpa who keeps repeating the same stories over and over again.

We can try to update the prompt, but it will not help:

```
//🟠 changing the promt will not fix the duplication
const response = await chain.invoke({
  input: `Tell me a fact about my favorite drink.
    Do not repeat any of the previous facts.`,
  chat_history: chatHistory
})
```

Remember that the LLMs are stateless — meaning each incoming query is processed independently of other interactions. For a stateless model, the only thing that exists is the current input. Nothing else.

This happens because we only provided the initial message (`My favorite drink is tea.`). We also need to update the message history as the conversation evolves.

5.3. Full conversation history

If you take a look at how we store the conversation history, it's just a plain JavaScript array.

```
const chatHistory = [
  new HumanMessage(`My favorite drink is tea.`)
]
```

This means that we can easily add messages by pushing new items to the array:

- messages from the user side, represented with the `HumanMessage` class
- and responses given by the AI, represented by the `AIMessage` class

As we receive more facts from the model, the conversation will look something like this:

```
[
// starting state
new HumanMessage(`My favorite drink is tea.`),
// user clicks the button
new HumanMessage(`Tell me a fact about my favorite drink. Do not
repeat any of the previous facts.`),
// the LLM replies
new AIMessage(`Is the second most consumed beverage in the
world, after water...`),
// user clicks again the button
new HumanMessage(`Tell me a fact about my favorite drink. Do not
repeat any of the previous facts.`),
// the LLM replies with another fact
new AIMessage(`Is the second most consumed beverage in the
world, after water...`),
// and so on
// ...
// ...
]
```

For this step, there will be no changes to the frontend.

Here is how the full code of the backend will look:

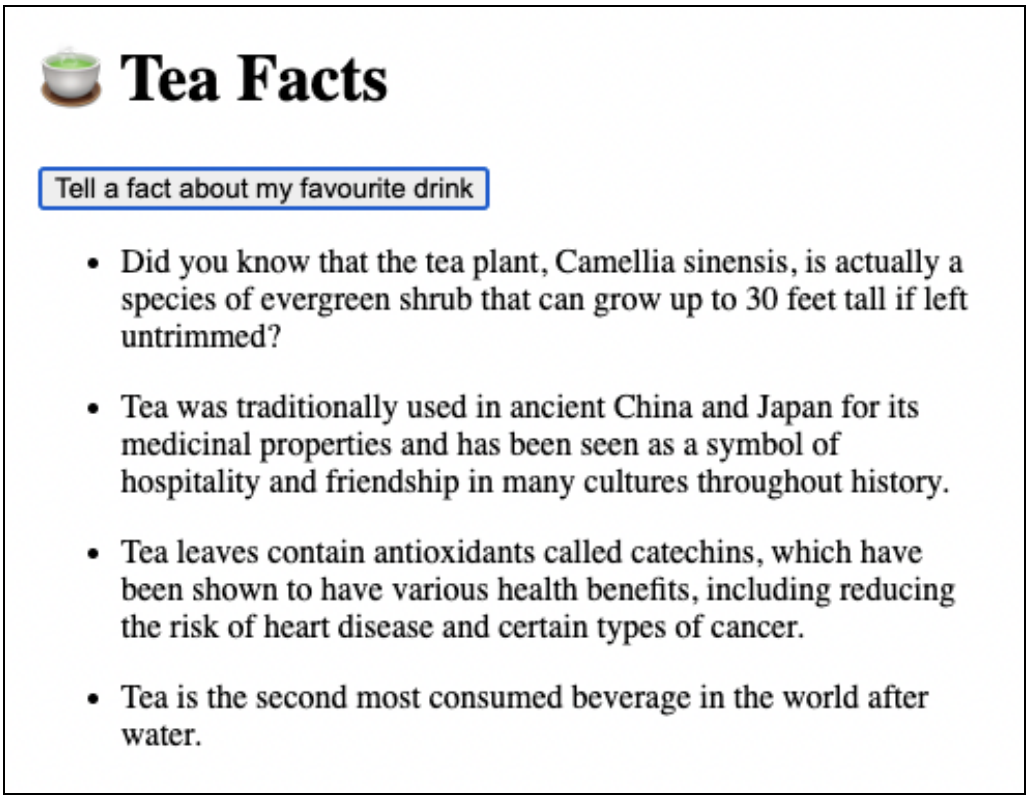
```
import { ChatOpenAI } from "@langchain/openai"
import { ChatPromptTemplate } from "@langchain/core/prompts"
import { StringOutputParser } from
"@langchain/core/output_parsers"
//🟠 import the AIMessage class to store the responses given by
the LLM
import { HumanMessage, AIMessage } from
"@langchain/core/messages"
import { MessagesPlaceholder } from "@langchain/core/prompts"

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

const chatHistory = [
    new HumanMessage(`My favorite drink is tea.`)
]

const prompt = ChatPromptTemplate.fromMessages([
    new MessagesPlaceholder("chat_history"),
    ["human", "{input}"]
])

const outputParser = new StringOutputParser()

const chain = prompt.pipe(model).pipe(outputParser)

export async function POST() {
    //🟠 update the prompt
    const question = `Tell me a fact about my favorite drink.
                      Do not repeat any of the previous
facts.`
```

```
    //🟠 add in the chat history the conversations sent by the
user
    chatHistory.push(new HumanMessage(question))
    const fact = await chain.invoke({
        input: question,
        chat_history: chatHistory
    })
    //🟠 store each fact so that the LLM knows what not to
repeat
    chatHistory.push(new AIMessage(fact))
    return Response.json({data: fact})
}
```

Each time a new question comes from the user, we will push it wrapped in a `HumanMessage` object.

More importantly, we will push each fact about tea in an `AIMessage` object. This will allow the LLM to avoid repeating itself.

And this is how the final version of our app will look at the end:

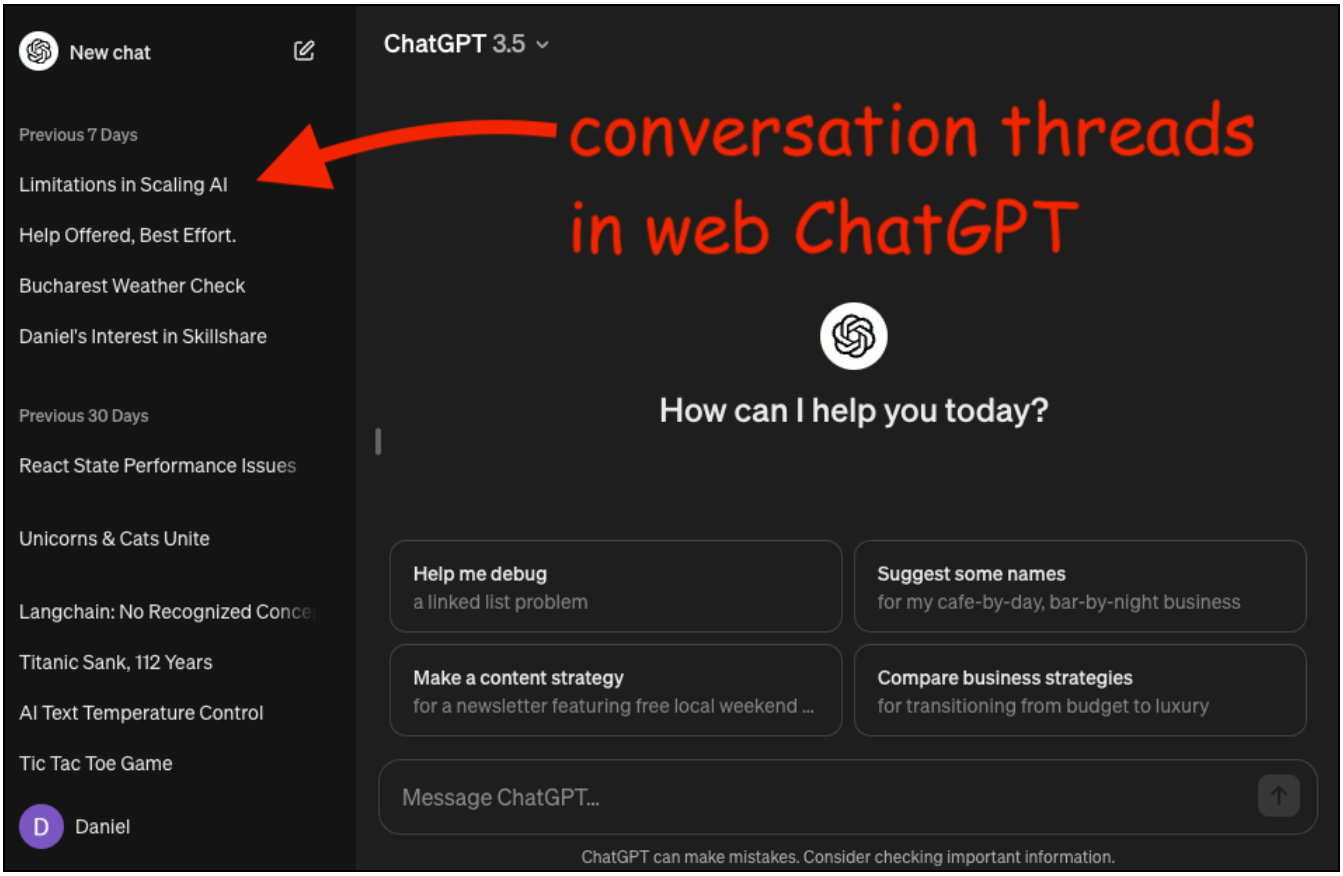


No more repetition here, thanks to the updated version of the conversation history.

As a final thought keep in mind that we can use a `sessionId` configuration parameter in the `invoke()` method to manage multiple conversation threads on different topics:

```
{
  configurable: {
    sessionId: "id_value_here"
  }
}
```

The way how the OpenAI ChatGPT web app does it:



You can read more details about this here:
https://js.langchain.com/docs/use_cases/chatbots/memory_management/

6. RAG / Chating with documents

6.1. Introduction

Let's talk about a very important topic in working with LLMs: RAG, which stands for Retrieval Augmented Generation.

RAG is the process of optimizing the output of an LLM by referencing a knowledge base outside of its training data sources.

For example, the knowledge cutoff date for ChatGPT 3.5 Turbo is September 2021. This means that, without external help, the model will not know what happened after this date.

But what if we need to work with information about LangSmith, a tool developed by the same company as LangChain? This new tool was released in 2023, after the knowledge cutoff date.

Or what if we need to make a chatbot that references some internal documents of an organization to provide customer support?

This is where RAG comes in. Using RAG you can connect external data sources like CSV files, videos, PDFs, etc. to an LLM and provide external context.

A few considerations about RAG:

- RAG does not retrain the model. It does not update the weights of the neuronal network. It just provides more information for the model to reference.
- in the next chapter, we will talk about AI Agents. Although AI Agents can search for external info, they are slower, more expensive, and can provide unreliable outputs. AI Agents are good for complex situations and if-then-else nonlinear cases that can be solved using various tools. With RAG, you can perform the knowledge retrieval process upfront and deliver fast solutions for scenarios such as Q&A.

But why is RAG so important?

LLM training costs a lot. Like a LOT! For example, the Llama 2 model from Meta, was trained using 6000 GPUs, running for 12 days. The training had a cost of about 2 million USD. In the vast majority of cases, you don't want to train your model from scratch.

However, with every passing minute, there is some information/knowledge that is getting generated. Keeping a model updated with new information is a constant race.

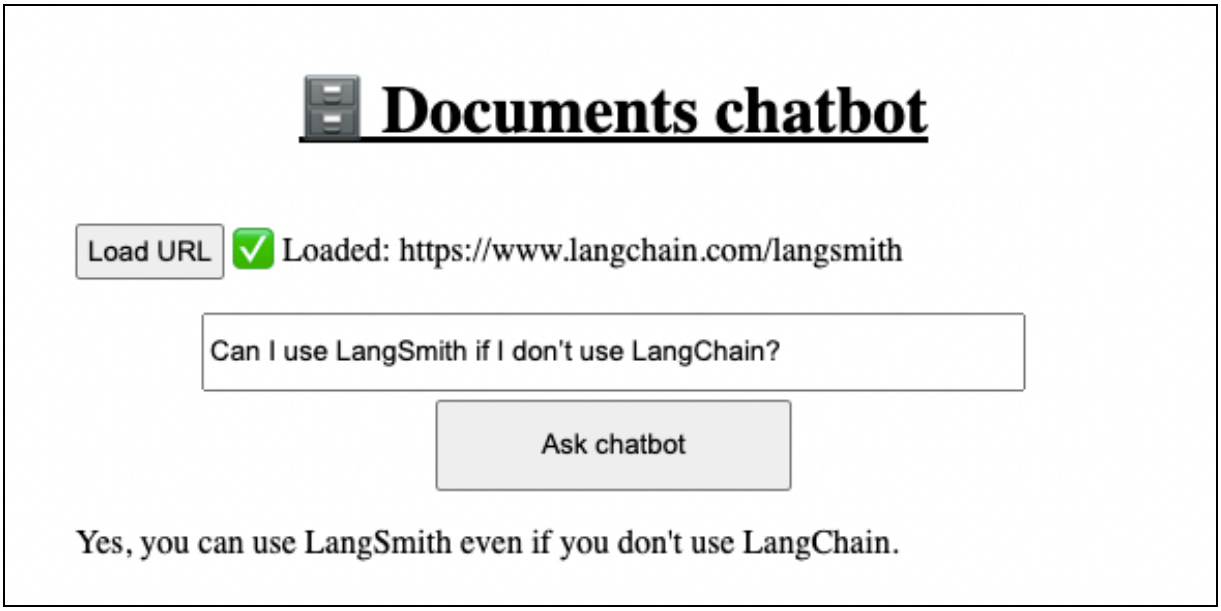
Even fine-tuning takes a lot of time.

What’s the other option? Use RAG to provide just the slice of knowledge we need for our LLM.

RAG saves both time and money, as no training is required.

6.2. Example setup

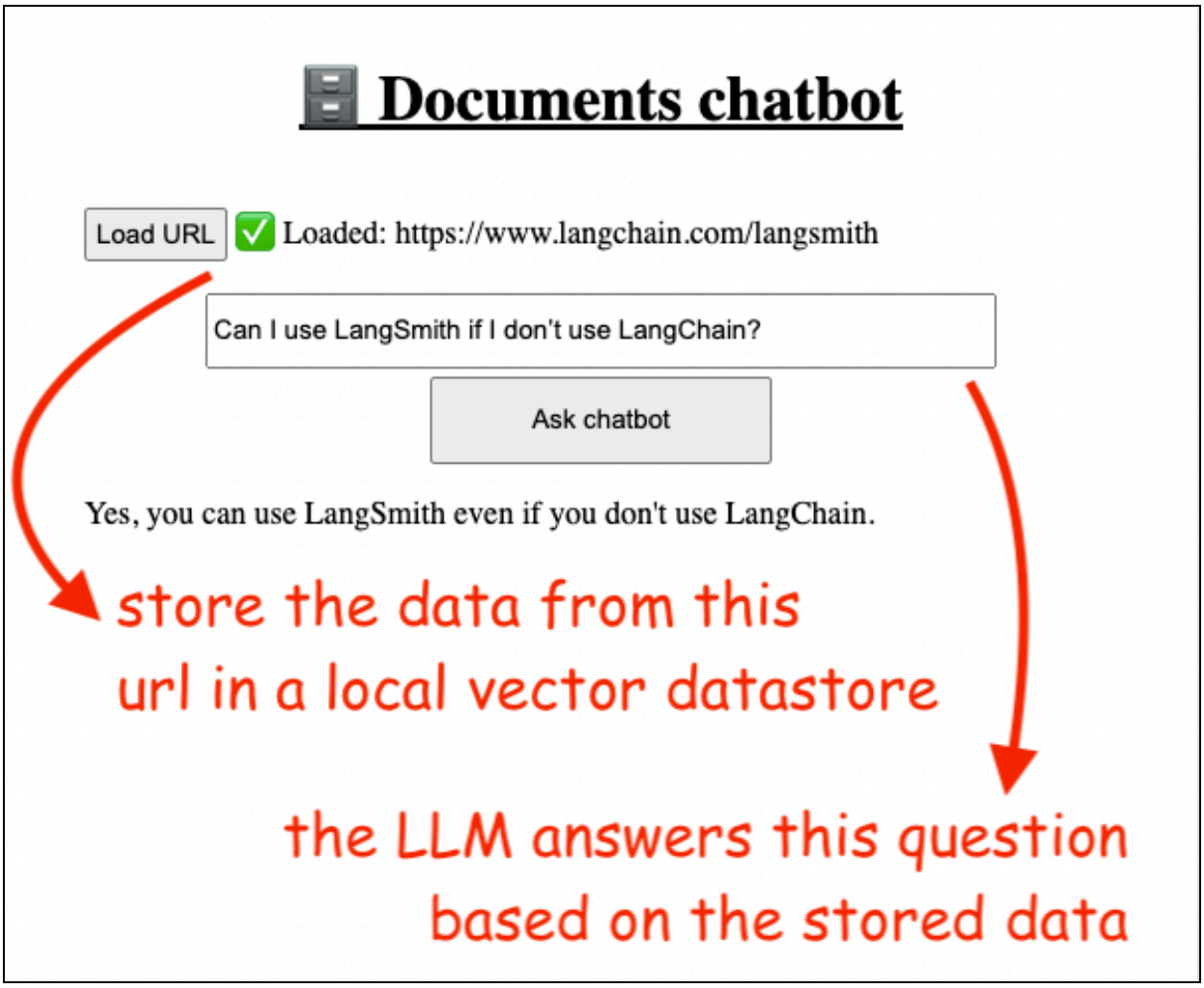
We will explore how to connect an LLM with external documents by building the following example:



In its final version, our app will know to do the following:

- read an external data source
- extract the embeddings from the text
- store the embeddings from the previous step in a local vector databases
- answer questions based on this information

This will be the data flow of the app:



Below is the initial frontend setup. We send a question and print out the response:

```
// code/rag/src/app/page.js

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

export default function Home() {
  const [answer, setAnswer] = useState()

  const askQuestion = async (e)=> {
    e.preventDefault()
    const question = e.target.question.value
    const response = await fetch('api', {
      method: "POST",
      body: JSON.stringify({question})
    })
```

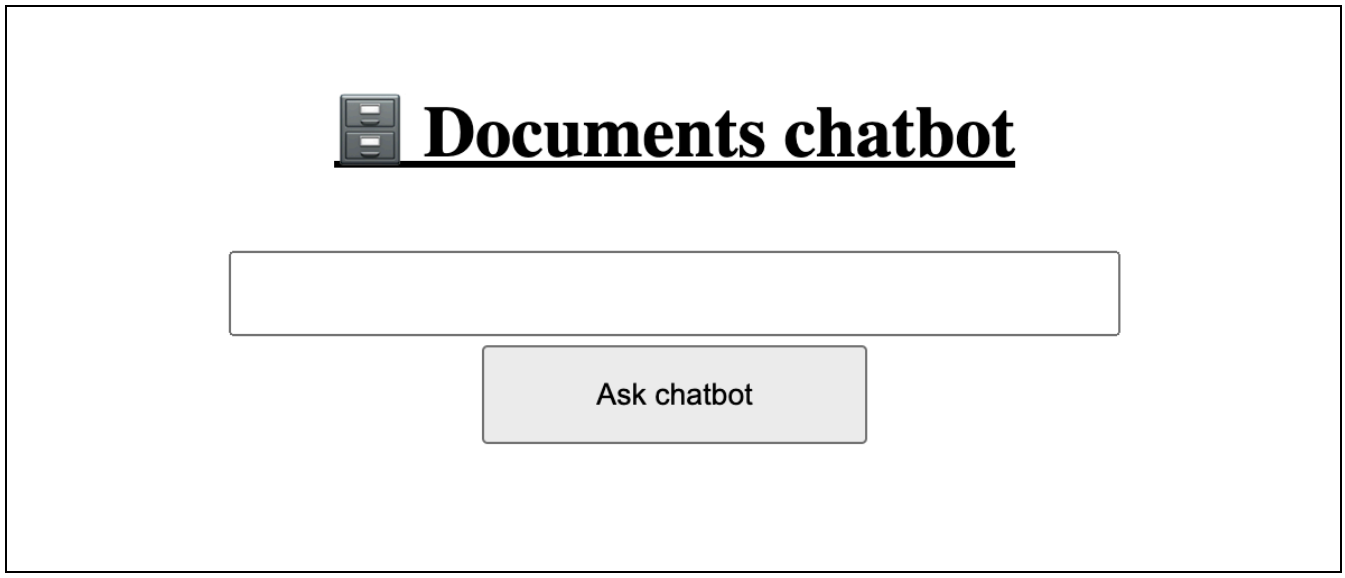
```
    const {data} = await response.json()
    setAnswer(data)
  }

  return (<>
    <h1>🗄️ Documents chatbot</h1>
    <form onSubmit={askQuestion} >
      <input name="question" />
      <button>Ask chatbot</button>
    </form>
    <p>{answer}</p>
  </>)
}
```

While on the backend we use a GPT model to answer the question:

```
// code/rag/src/app/api/route.js
import { ChatOpenAI } from "@langchain/openai"
import { ChatPromptTemplate } from "@langchain/core/prompts"
import { LLMChain } from "langchain/chains"

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

const prompt = ChatPromptTemplate.fromTemplate(
    `Answer the user's question:
    Question: {input}`
)

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

export async function POST(req) {
    const { question } = await req.json()
    const data = await chain.invoke({
```

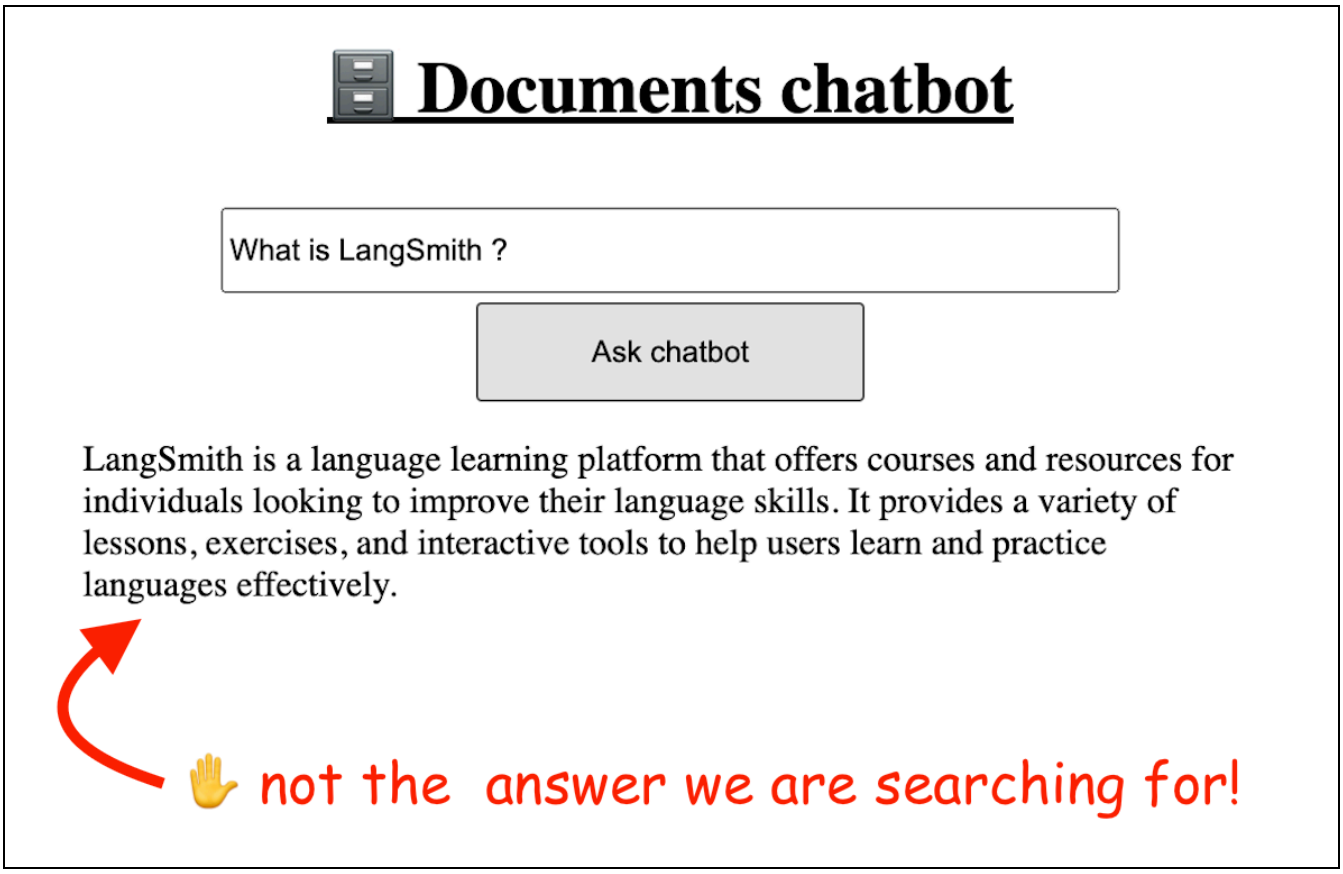
```
        input: question
    })
    return Response.json({data: data.text})
}
```

This is how the initial setup of the app looks:



While this works as expected for simple questions, the limits of the model become visible when we ask about topics outside its training data.

For example, if we ask about LangSmith, a platform for deploying and monitoring LLM applications, it will give general answers like the one below:



You may say that we could try to use a newer model. For example, we could try to use a gpt-4o model, which has its training data up to October 2023. Most likely, it will know something about LangSmith. However, the core problem will remain: we will always play catch-up with the cutoff date of the models.

Therefore, it's time to bring in the Retrieval Augmented Generation!

We can use RAG to add new information to the model and overcome its limitations. This information can be:

- after the cut-off date of the model training
- outside the domain of the initial training data

So, let's explain to our model what LangSmith is.

6.3. Using local documents

One of the easiest ways to expand the context of an LLM with LangChain is by using local documents.

We can create small pieces of information and feed them to the LLM:

```
//🟠 manually create static documents
const documentA = new Document({
  pageContent:
    `LangSmith is a unified DevOps platform for developing,
    collaborating, testing, deploying, and monitoring
    LLM applications.`
})

const documentB = new Document({
  pageContent: `LangSmith was first launched in closed beta in
July 2023`
})
```

In this example, we have used simple text pieces, but we can also have locally stored PDFs, CSVs, and other types of files. We can even include videos and audio, which can contain a larger quantity of information.

Let's see how all of this comes together in backend code:

```
// code/rag/src/app/api/route.js

import { ChatOpenAI } from "@langchain/openai"
import { ChatPromptTemplate } from "@langchain/core/prompts"
//🟠 we need the this one to create local documents
import { Document } from "@langchain/core/documents"
//🟠 createStuffDocumentsChain replaces LLMChain
import { createStuffDocumentsChain } from
"langchain/chains/combine_documents"

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

```
    //🟠 zero temperature, no extra creativity
    temperature: 0
})

//🟠 adding the extra context to the prompt
const prompt = ChatPromptTemplate.fromTemplate(
    `Answer the user's question from the following context:
    {context}
    Question: {input}`
)

//🟠 manually create static documents
const documentA = new Document({
  pageContent:
    `LangSmith is a unified DevOps platform for developing,
    collaborating, testing, deploying, and monitoring
    LLM applications.`
})

const documentB = new Document({
  pageContent: `LangSmith was first launched in closed beta in
July 2023`
})

//🟠 updating the chain type
const chain = await createStuffDocumentsChain({
    llm: model,
    prompt,
})

export async function POST(req) {
    const { question } = await req.json()
    //🟠 pass the documents array as the context
    const data = await chain.invoke({
        input: question,
        context: [documentA, documentB]
    })
    return Response.json({data})
```

```
}
```

Note that a chain made with `createStuffDocumentsChain()` expects to receive a `context` variable when you are calling its `invoke()` method.

Failing to do so will result in the following error:

```
Error: Prompt must include a context variable
```

The `createStuffDocumentsChain()` takes a list of documents and formats them all into a prompt, then passes that prompt to an LLM. It passes ALL documents, so you should make sure it fits within the context length of the LLM you are using.

Context length refers to the maximum number of words a model can handle at once. Think of it as the model's memory or attention span. This is a predetermined feature in transformer-based models such as ChatGPT and Llama.

Below are the default context lengths of some popular LLMs:

Model	Context Size	Number of pages*
🤖 GPT 3.5	4,096	6
🤖 GPT 4	8,192	12
🤖 GPT 4-32k	32,768	49
🦙 Llama 1	2,048	3
🦙 Llama 2	4,096	6
🦙 Llama 3	8,192	12
** 500 words per page*		

Expanding the context does not retrain the model; the weights remain unchanged in this process.

The frontend will remain the same. We simply pass in the question and display the answer.

Let's test it out. Will ask it again `What is LangSmith?` and will now get an answer from the local documents we have just created:

```
const documentA = new Document({
  pageContent:
    `LangSmith is a unified DevOps platform for developing,
    collaborating, testing, deploying, and monitoring
    LLM applications.`
})
```

6.4. Components of the RAG process

Let's take a look at the different component types involved in the RAG process:

1. Document Loaders - think of these as your data gatherers. They pull in external info and load it up, whether it's plain old .txt files, web page text, or even YouTube video transcripts.
2. Transformers - you've got your docs loaded, but sometimes they need a little makeover to fit your app just right. LangChain's pre-made transformers help you tweak the docs. A basic example can be to use a transformer to break down a big document into smaller segments that fit within your model's contextual window.
3. Embeddings - this is all about turning data into vectors. The Embeddings class is your go-to for working with embedding models like the ones from OpenAI, Meta, or those on Hugging Face. Think of text embeddings as a way to represent text or data in a vector space, making stuff like semantic search faster.
4. Vector Databases - time to stash the text embeddings we have generated in the previous step. These specialized databases are built to store and fetch vector data efficiently. They make semantic searches easy and quick. We can pull up the most relevant text in a snap. We'll dive deeper into vector databases later on.
5. Retrievers - last but not least, meet your info-fetching sidekick: the retrievers. Unlike vector stores that just hold onto data, a retriever goes out and fetches the docs you need. While some retrievers might use vector stores, there are all kinds of retrievers tailored for specific tasks.

6.5. Vectors, Embeddings and Vector databases

Before we continue, we need to make a short detour and discuss some fundamental concepts involved in the RAG process.

We will start with vectors. A vector is simply an array of numbers.

```
// this is how a vector looks like
let vec = [2.5, 1.3, 7.0, 9.2]
```

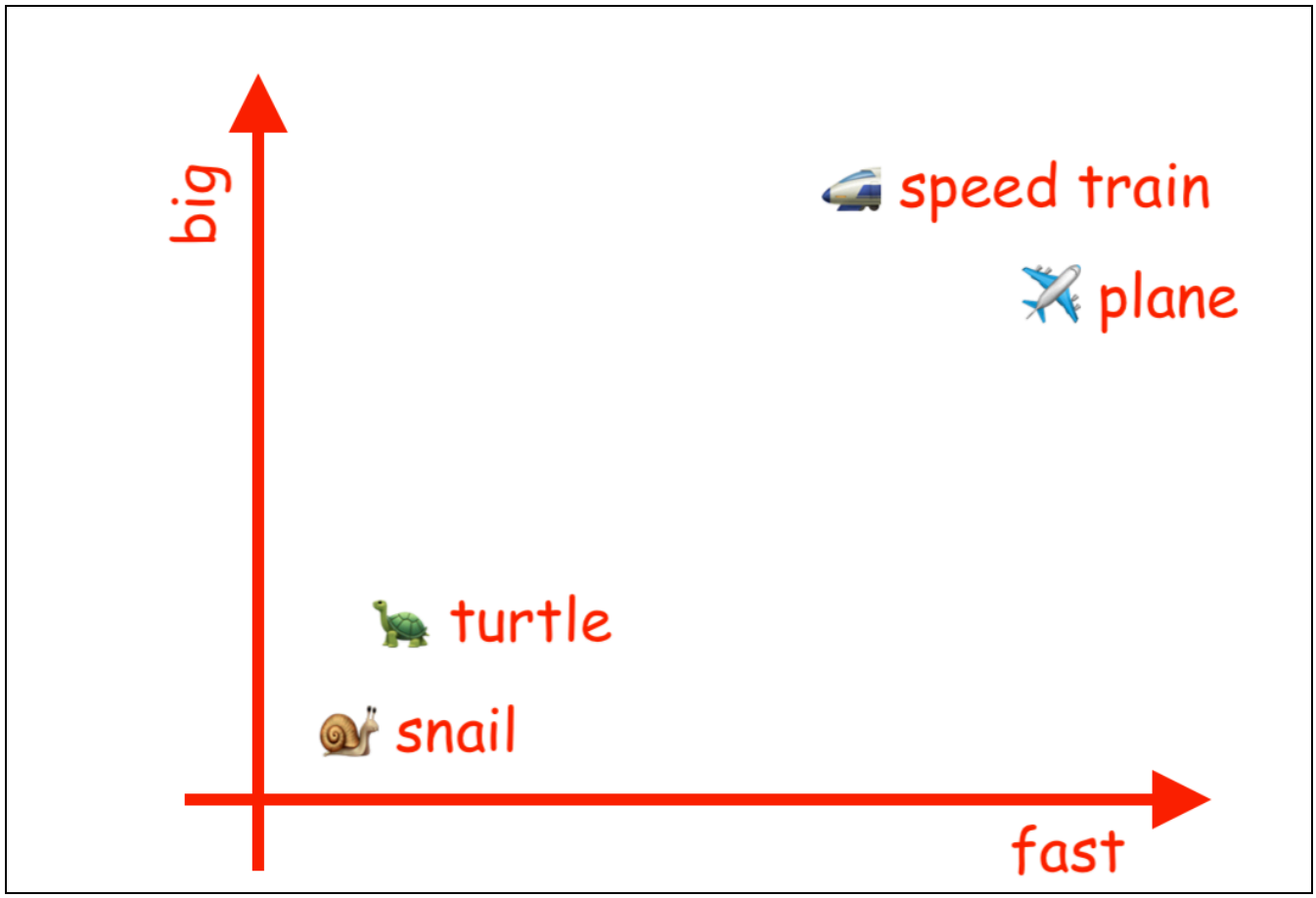
What's cool about vectors is that they can represent more complex objects like words, sentences, audio files, or even images in a high-dimensional space called an embedding.

For example, let's take the following entries: `snail`, `plane`, `speed train` and `turtle`.

We can store these entries in a two-dimensional embedding space based on:

- how big they are
- how fast they are

A visual representation might look like this:



Each point is a vector with the following format:

```
[
  x, // how big is the object
  y  // how fast is the object
]
```

Each axis of an embedding space is also known as a dimension or a feature.

Keep in mind that this representation is very simplified. In general, each entry has many more dimensions, ranging from tens to thousands, depending on the complexity and granularity of the data.

Given a set of dimensions, similar objects are grouped together, allowing us to map the semantic meaning of words or similar features in virtually any other data type.

Using these embeddings we can do stuff such as recommendation systems, search engines and even text generation like the one in Chat GPT.

But once we have our vector entries mapped to the dimensions of the embeddings, where do we store them? And how can we query them quickly?

Well, that's where vector databases come in. In a classic relational database, we have rows and columns; in a vector database, we have arrays of numbers (vectors) grouped together based on similarity.

Organizing data this way allows us to make similar queries with ultra-low latency.

We can easily retrieve the distance between these entries. This makes semantic search faster as we can easily relate between different concepts based on one or multiple dimensions.

Some of the main vector databases we can use today:

- Chroma
- Pinecone
- Weaviate
- Redis Vector Database

6.6. Using CheerioWebBaseLoader and MemoryVectorStore

In the previous example, we used just the document loaders and the retriever. Now, it's time to expand our example and put the other RAG components to work.

We want to add the following features to the app:

- read a given URL provided by the user
- load the content of the page from the URL
- create embeddings and store the results in the local vector database
- retrive and answer questions from the new data

This is how the final example will look like:

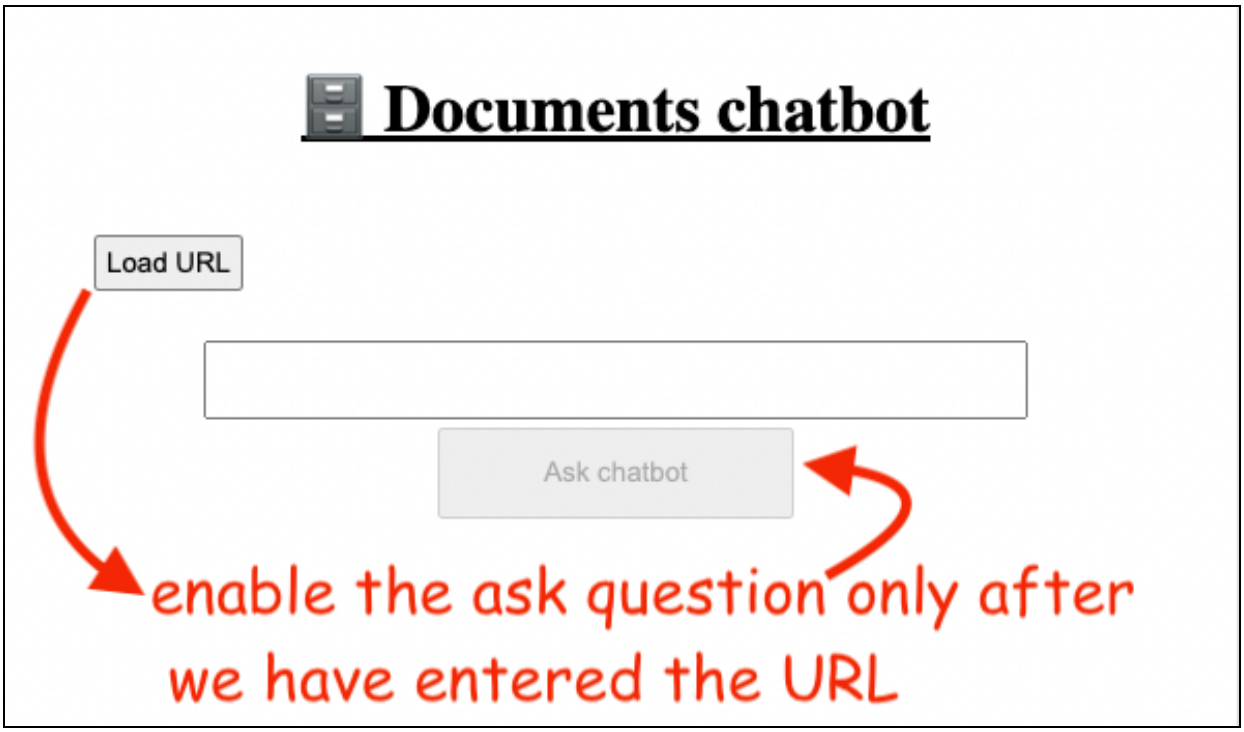
We'll start with the frontend. We will add a new button that asks for a URL and sends it to the backend:

```
// code/rag/src/app/page.js

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

```
export default function Home() {
  const [answer, setAnswer] = useState()
  //🟠 add a new state variable
  const [docStatus, setDocStatus] = useState(false)

  const askQuestion = async (e)=> {
    e.preventDefault()
    const question = e.target.question.value
    const response = await fetch('api', {
      method: "POST",
      body: JSON.stringify({question})
    })
    const {data} = await response.json()
    setAnswer(data)
  }

  //🟠 ask for the URL, pass it to the backend, update the
docStatus
  const loadUrl = async (e)=> {
    const url = prompt("Please enter the URL for the document
context:")
    const response = await fetch('api', {
      method: "POST",
      body: JSON.stringify({url})
    })
    const { loaded } = await response.json()
    setDocStatus(loaded ? `✅ Loaded: ${url}` : false)
  }


  return (<>
    <h1>🗄️ Documents chatbot</h1>
    {/*🟠 ask for the URL and show the docStatus */}
    <p>
      <button onClick={loadUrl}>Load URL</button>
      {docStatus}
    </p>
```

```
    <form onSubmit={askQuestion} >
      <input name="question" />
      {/*🟠 disabled until we have a doc status*/}
      <button disabled={!docStatus}>
        Ask chatbot
      </button>
    </form>
    <p>{answer}</p>
  </>)
}
```

By default, the ask-question form will be disabled. Once the backend finishes loading the data from the URL it will enable the question form:



The main difference is in the backend, where the model answers the question.

First, we will need to install the `CheerioWebBaseLoader`:

```
npm install --save cheerio
```

Now, let's first see the complete code, and we'll delve into further explanations afterward:

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

```
import { ChatPromptTemplate } from "@langchain/core/prompts"
import { createStuffDocumentsChain } from
"langchain/chains/combine_documents"
//🟠 document loader that can retrive the content of a web page
import { CheerioWebBaseLoader } from
"langchain/document_loaders/web/cheerio"
//🟠 tool for making the vectors and embeddings
import { RecursiveCharacterTextSplitter } from
"langchain/text_splitter"
import { OpenAIEmbeddings } from "@langchain/openai"
import { MemoryVectorStore } from
"langchain/vectorstores/memory"
//🟠 retrieval tool
import { createRetrievalChain } from
"langchain/chains/retrieval"

const model = new ChatOpenAI({
    openAIApiKey: process.env.OPENAI_API_KEY,
    temperature: 0
})

const prompt = ChatPromptTemplate.fromTemplate(
    `Answer the user's question from the following context:
    {context}
    Question: {input}`
)

let retrievalChain, splitDocs

//🟠 the RAG process
async function loadDocumentsFromUrl(url) {
  //🟠 document loaders
  const loader = new CheerioWebBaseLoader(url)
  const docs = await loader.load()

  //🟠 document transformers
  const splitter = new RecursiveCharacterTextSplitter({
      chunkSize: 100,
```

```
        chunkOverlap: 20,
    })

    splitDocs = await splitter.splitDocuments(docs)

    //🟠 setting up the embeddings
    const embeddings = new OpenAIEmbeddings()

    //🟠 making a local vector DB
    const vectorstore = await MemoryVectorStore.fromDocuments(
        splitDocs,
        embeddings
    )

    //🟠 what we use to fetch data from the vector DB
    const retriever = vectorstore.asRetriever()

    const chain = await createStuffDocumentsChain({
        llm: model,
        prompt
    })

    retrievalChain = await createRetrievalChain({
        combineDocsChain: chain,
        retriever
    })
}

export async function POST(req) {
    const { question, url } = await req.json()
    //🟠 if the user sends an url setup RAG and return
    if (url) {
        await loadDocumentsFromUrl(url);
        return Response.json({ loaded: true });
    }
    const data = await retrievalChain.invoke({
        input: question,
        context: splitDocs
```

```
    })
    return Response.json({data: data.answer})
}
```

In this code, we see in action the RAG components described in the previous chapter.

First, we load the raw text from the URL provided by the user:

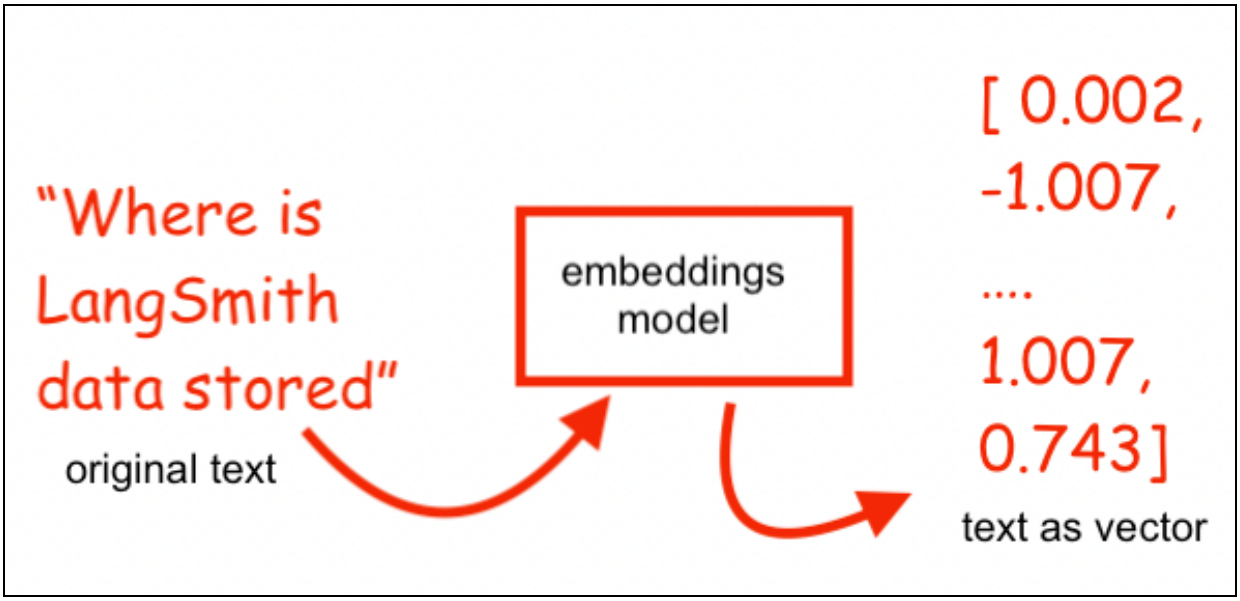
```
const loader = new CheerioWebBaseLoader(url)
const docs = await loader.load()
```

This is just one type of loader LangChain provides. You can see the full list of loaders here.

We have access to two types of loaders:

- file loaders, such as CVSs, PDFs, EPUBs etc
- and web loaders, such as Figma, Amazon S3, Youtube transcripts etc

Once the raw text is loaded we start the process of transforming the text (understandable by humans) into number vectors (understandable by AI):



This splitter is tailored for processing general text. It attempts to segment the text at these characters sequentially until the resulting chunks reach an optimal size.

By default, it employs a list comprising ["\n\n", "\n", " ", ""]. This configuration aims to maintain the integrity of paragraphs first, then sentences, and finally words, as these segments typically exhibit the highest semantic cohesion.

The text is split by the list of characters, and the chunk size is measured by the number of characters:

```
const splitter = new RecursiveCharacterTextSplitter({
    chunkSize: 100,
    chunkOverlap: 20,
})
splitDocs = await splitter.splitDocuments(docs)
const embeddings = new OpenAIEmbeddings()
```

Next, we store all of these embeddings in a vector database. For this small example, we used a local database. For production, we can use a real vector DB service such as Chroma or Pinecone.

```
const vectorstore = await MemoryVectorStore.fromDocuments(
    splitDocs,
    embeddings
)
```

Do we have to recreate embeddings each time we run a RAG query? This might work for small files, but for larger ones, the embedding creation process could be very time-consuming.

For example, we can set a parameter during the creation of a Chroma DB object, like `persist_directory`, specifying a valid local path. This way, when you create your Vector DB using a file, it will be stored in the specified location. Subsequently, it will be loaded from that location the next time, significantly reducing processing time.

The `createRetrievalChain()` takes a user inquiry, which is then passed to the retriever to fetch relevant documents. Those documents (and original inputs) are then passed to an LLM to generate a response.

Finally, we will build a retriever that can be used by a chain to fetch information from the vector database.

```
const retriever = vectorstore.asRetriever()
const chain = await createStuffDocumentsChain({
    llm: model,
    prompt
})
retrievalChain = await createRetrievalChain({
    combineDocsChain: chain,
    retriever
})
```

As a side note, Google's Gemini 1.5 model has a million token contexts and is researching 10 million tokens. It's so large that you can include an entire knowledge base, books, or films without worrying about context limits. Some people think this will make RAG obsolete. However, this is unlikely due to:

1. 💸 Costs - you still pay per token. Google is like perfectly happy to let you pay a million tokens every single time you make an API call. So, good luck having a $100 API call.
2. 🐢 Slow - the API call will be huuuuge, and the response will be slow as well.
3. 🔎 Troubleshooting - a very big context is not debuggable. If something goes wrong, you can't do the RAG's decomposition to see the source of the error.

One fun experiment could be to take a book like this use one, and using RAG feed the content to an LLM model. A user can later come and interact with that LLM to learn about LangChain using the approach from this book. Or just ask the model for recap questions.

Maybe this will be the future of future of how we consume technical books.

7. AI Agents

7.1. Introduction and project setup

Up until now, we have looked at creating chains. Chains follow a predetermined sequence specified by us.

Agents are different in the sense that we can simply give them a goal, and they will dynamically figure out the best course of actions to take and the order of those actions to reach that final goal.

We will assign tools to an agent, and the agent will try to figure out when to use a tool to complete the task.

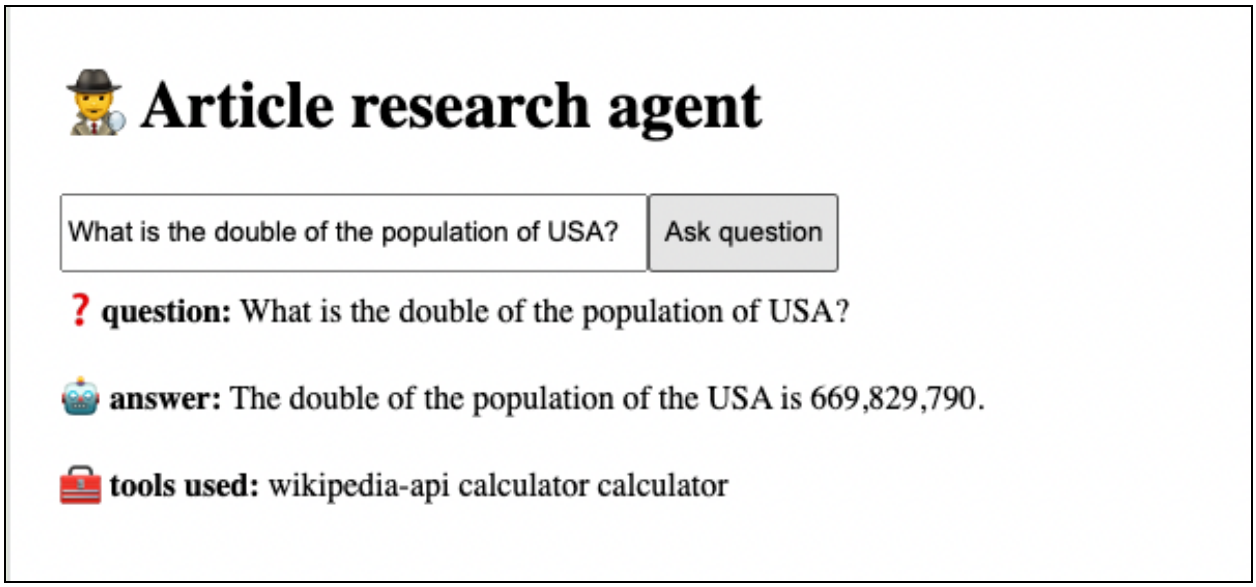
Until this moment, we have used the LLM primarily as an NLP (Natural Language Processing) tool. AI agents are closer to what we imagine "real AI" to be.

Having objectives, trying different approaches, interacting with the environment, and making decisions are more aligned with what we consider real intelligence.

We were tasked with building a tool to help journalists in their research for new articles. The tool needs to be able to search on Wikipedia and perform precise math calculations.

By the end of this chapter, we will build an app that will do the following:

- read a question from the user
- use an agent to answer the question
- the output will contain the answer to the question, and the tools used by the agent to find that answer.



We will start from the following state: on the frontend, we will let the user type a question and submit that question to the server. When the response arrives, we will display its text.

```
// code/agents/src/app/page.js

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

export default function Home() {
  const [data, setData] = useState();
  const onSubmitHandler = async (e) => {
    e.preventDefault();
    const question = e.target.question.value;
    if (question) {
      const response = await fetch("api", {
        method: "POST",
        body: JSON.stringify({
          question,
        }),
      });
      const { data } = await response.json();
      setData(data);
    }
  };
  return (
```

```
    <>
      <h1>🕵️‍♂️ Article research agent</h1>
      <form onSubmit={onSubmitHandler}>
        <input name="question" />
        <button>Ask question</button>
      </form>
      <p>{data?.text}</p>
    </>
  );
}
```

On the backend, we will ask a GPT-4 Omni model the question inputted by the user.

```
// code/agents/src/app/api/route.js
import { ChatOpenAI } from "@langchain/openai"
import { PromptTemplate } from "@langchain/core/prompts"
import { StringOutputParser } from
"@langchain/core/output_parsers"

const model = new ChatOpenAI({
    modelName: "gpt-4o",
    openAIApiKey: process.env.OPENAI_API_KEY,
})

const prompt = new PromptTemplate({
    inputVariables: ["question"],
    template: "Answer the user {question}",
})


const chain = prompt
                .pipe(model)
                .pipe(new StringOutputParser())

export async function POST(req) {
    const { question } = await req.json()
```

```
    const data = await chain.invoke({
        question
    })
    return Response.json({data})
}
```

The initial version of our app seems to work quite nicely. For example, if we ask it to tell us a joke, everything works as expected.

But what if we ask `Who won the  Super Bowl in 2024?` the limitations of the training data set start to emerge. Given that the model has been trained on data up to Oct 2023, it does not know the answer.

7.2. Making an agent

It's time to make our first agent. To overcome the limitations imposed by the training dataset, we will give the chatbot the option to conduct its own research online.

To do their job, agents require access to specific tools, such as Google or Wikipedia search capabilities. By combining the GPT models with these tools, agents determine the actions needed to achieve their goals.

If we check the Tools documentation we will see the full list of available options https://js.langchain.com/docs/integrations/tools/.
![](img/agents-docs-tools.png'>

While it might be tempting to load all available tools, it's essential to equip the agent only with the necessary ones. Providing too many options may lead to one or more of the following:

- the agent selected an inappropriate tool
- higher cost per API call; the agent will use more tokens
- increased network latency

For our use case, we will equip the agent with only 2 tools. A Wikipedia search tool and a math calculator.

```
const wikipediaQuery = new WikipediaQueryRun({topKResults: 1})
const calculator = new Calculator()
const tools = [wikipediaQuery, calculator]
```

Please note that for the `Calculator` tool, you will need to install the `@langchain/community` package:

```
npm install @langchain/community
```

Wait, what!? A math calculator for an AI model? Well, because neural networks work by estimating stuff, they are not great at math. There is a very small chance they will mess up a basic math operation. Therefore, it's safer and faster to use a basic math tool for these operations.

This toolset will be passed to an AI agent. Each agent type serves a unique purpose, as seen from the documentation: https://js.langchain.com/docs/modules/agents/agent_types/. For instance, a Structured Chat is optimized for multiple back-and-forth interactions.

We'll employ the ReAct Agent, the most versatile action agent. As a side note, the ReAct Agent has nothing to do with the JavaScript React framework.

This is how the full code of the backend will look like:

```
// code/agents/src/app/page.js
import { ChatOpenAI } from "@langchain/openai"
import { WikipediaQueryRun } from
"@langchain/community/tools/wikipedia_query_run"
import { createReactAgent, AgentExecutor } from
"langchain/agents"
import { Calculator } from
"@langchain/community/tools/calculator"
import { pull } from "langchain/hub"

const model = new ChatOpenAI({
  modelName: "gpt-4o",
  openAIApiKey: process.env.OPENAI_API_KEY
})

// 🟠 making the toolbox for the agent
// 🟠 remember to run 'npm install @langchain/community'
const wikipediaQuery = new WikipediaQueryRun({topKResults: 1})
const calculator = new Calculator()
const tools = [wikipediaQuery, calculator]

// 🟠 getting the agent rule set
const prompt = await pull("hwchase17/react")
```

```
// 🟠 define the Agent and the AgentExecutor
const agent = await createReactAgent({
  llm: model,
  tools,
  prompt
})

const agentExecutor = new AgentExecutor({
  agent,
  tools
})

export async function POST(req) {
  const { question } = await req.json()
  const response = await agentExecutor.invoke({
    input: question
  })
  return Response.json({data: response, prompt})
}
```

And this is the frontend code:

```
// code/agents/src/app/page.js
'use client'
import { useState } from "react"

export default function Home() {
  const [data, setData] = useState();

  const onSubmitHandler = async (e) => {
    e.preventDefault();
    const question = e.target.question.value;
    if (question) {
      const response = await fetch("api", {
        method: "POST",
        body: JSON.stringify({
          question,
```

```
        }),
      });
      const { data } = await response.json();
      setData(data);
    }
  };

  return (
    <>
      <h1>🕵️‍♂️ Article research agent</h1>
      <form onSubmit={onSubmitHandler}>
        <input name="question" />
        <button>Ask question</button>
      </form>
      {data && (
        <div>
          <p>
            <b>❓question:</b> {data.input}{" "}
          </p>
          <p>
            <b>🤖 answer:</b> {data.output}{" "}
          </p>
        </div>
      )}
    </>
  );
}
```

At this moment, if we ask the model again `Who won the Super Bowl in 2024?` it will be able to use the Wikipedia tool to search for an answer.
![](img/agents-wiki-answer.png'>

Oh boy, Taylor Swift must be happy!

7.3. Monitoring agents and performance considerations

Agents are fantastic, but they come at a cost. They can be:

- 🐢 slow to respond
- 💸 have high computational costs
- 🎲 sometimes can give random results

Luckily, we have a few ways to control the autonomous behavior of the agents.

Let's update the backend code with some monitoring and performance enhancements.

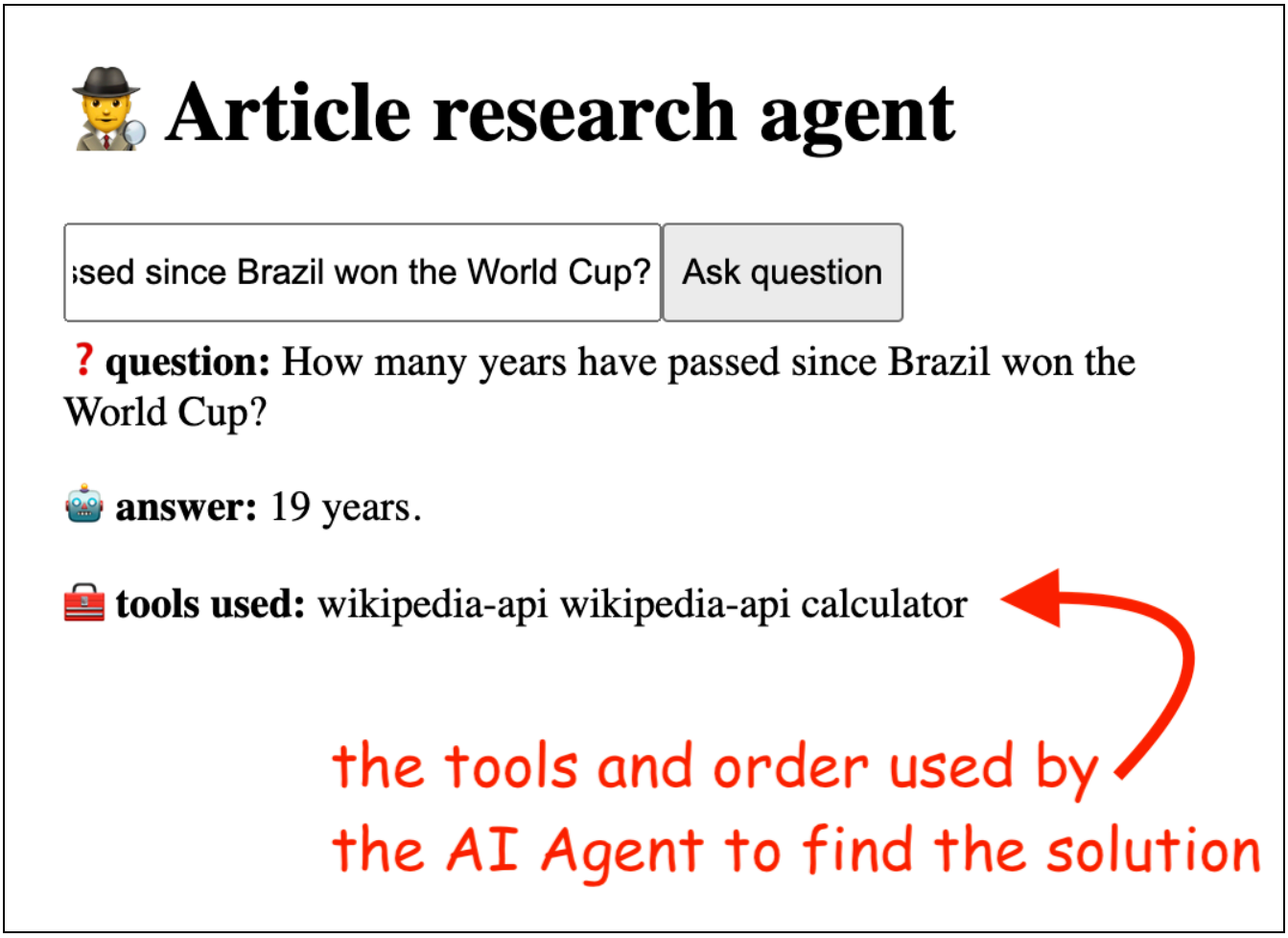
Add the following lines in the constructor of the `AgentExecutor`:

```
// code/agents/src/app/api/route.js

const agentExecutor = new AgentExecutor({
 agent,
 tools,
 verbose: true,
 returnIntermediateSteps:true,
 maxIterations: 4,
})
```

And this code in the returned HTML of the frontend:

```
// code/agents/src/app/page.js

data && (<div>
 <p><b>❓question:</b> {data.input} </p>
 <p><b>🤖 answer:</b> {data.output} </p>
 <p><b>🧰 tools used:</b> {
  data.intermediateSteps.map(step => step.action.tool)
 }</p>
</div>)
```

At this point, if we ask the agent, "How many years have passed since Brazil won the World Cup?" we will get the following result:



Let's talk a bit about the added chances.

First, the `maxIterations` option limits the number of attempts an agent has to find the solution. This is a good option to speed up the response and to ensure we do not run into an infinite loop.

As in the previous use case, the `verbose` option prints - in the backend console - the reasoning that leads to a given result. For example, when asked, "What is the double of PI?" it will print something like this:

```
[chain/start] [1:chain:AgentExecutor > 18:chain:ReactAgent >
22:prompt:PromptTemplate] Entering Chain run with input: {
  "input": "What is the double of PI?",
  "steps": [
    {
      "action": {
      "tool": "wikipedia-api",
      "toolInput": "PI",
      "log": "PI is a mathematical constant that can be found on
Wikipedia\nAction: wikipedia-api\nAction Input: PI"
    },
    {
      "action": {
      "tool": "calculator",
      "toolInput": "2*3.14159",
      "log": "I know that PI is approximately equal to 3.14159,
so I can calculate the double of it\nAction: calculator\nAction
Input: 2*3.14159"
    },
  ],
  "content": "I now know the final answer\nFinal Answer:
6.28318"
}
```

This is quite cool, isn't it?

And the third option, the `returnIntermediateSteps` (documentation link here) prints the tools used by the agent and their order. Agents are autonomous and can decide their own set of actions.

For example, the Wikipedia tool was enough to respond to the question, "Who won the Super Bowl in 2024?" On the other hand, the question, "How many years have passed since Brazil won the World Cup?" required the use of both Wikipedia and the Calculator tools. With this option, we can monitor

if the agent is using the correct tools for the job.

7.4. Recap

Agents and tool use in general, are key concepts of LangChain.

At a high level agents involve using a LLM and asking it to reason about which actions to take and then executing those actions. Often this is done in a loop that repeats until the LLM decides it has reached the objectives, or there are no other actions it can take.

The tools are representations of actions a language model can take, along with the functions that implement those actions.

Agents and tools are combined with an agent executor. This agent executor is essentially a loop that runs to call an LLM to figure out what tool the agent should try to use next unless it has reached its objective.

Agents are constructed based on prompts, as you may have noticed from the code:

```
const prompt = await pull("hwchase17/react")

const agent = await createReactAgent({
  llm: model,
  tools,
  prompt
})
```

There are many different agent types. You can see the five main types of agents available at this moment on this page:

LangChain

When to Use

Our commentary on when you should consider using this agent type.

Agent Type	Intended Model Type	Supports Chat History	Supports Multi-Input Tools	Supports Parallel Function Calling	Require Model Params
OpenAI Tools	Chat	✅	✅	✅	`tools`

Some are good for local models, some are good for the newest OpenAI models, some support parallel function calling, and some support conversational history. Others are great for using simple tools and single inputs because of their general prompting strategies.

When discussing AI agents, please note that there are many other topics, concepts, techniques, prompts, and features. What we have discussed in this chapter is just a very brief introduction to the subject.

7.5. Final Words

Hopefully, you have enjoyed this book!

I would love to get your feedback. Let me know what you liked and didn't like so I can improve this book.

I would love to get your feedback, and learn what you liked and didn't so I can improve this book.

Feel free to email me at daniel@js-craft.io to get updated versions of this book. If you think I should have covered more topics, please email me. I'd love to hear from you!

If you like the book, I would appreciate it if you could leave a review too.

Thanks, friend and keep coding!