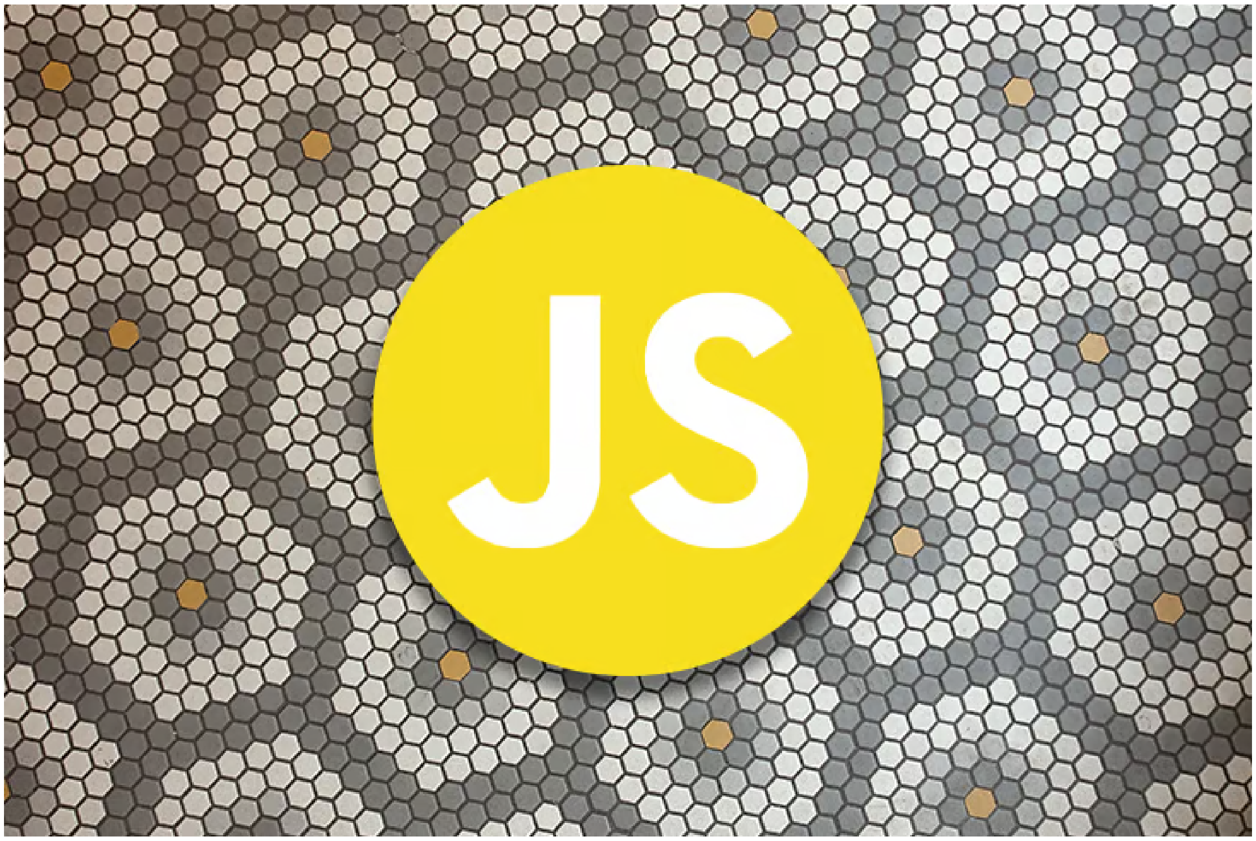
Since the release of ChatGPT, AI has become a popular addition to many applications. However, it can be difficult to leverage AI's power unless you know how to interact with large language models (LLMs).



LangChain is a JavaScript library that makes it easy to interact with LLMs. Its powerful abstractions allow developers to quickly and efficiently build AI-powered applications.

In this guide, we will learn the fundamental concepts of LLMs and explore how LangChain can simplify interacting with large language models. Finally, for a practical demonstration, we'll build our own custom chatbot using LangChain. You can follow along in the project's GitHub repository.

***Jump ahead*:**

What is LangChain?

LangChain is an open source framework that lets software developers work with artificial intelligence. LangChainJS is the JavaScript version of LangChain, offering the following features:

Custom prompt chatbots: You can create a custom prompt chatbot using LangChainJS. You can use the library's available LLMs as a base to create your custom chatbotCustom knowledge chatbots: You can use your own data to train these LLMs to create a custom knowledge chatbot. LangChain supports many

forms of input data, including JSON, CSV, TXT, etc.Document search engine: Use LangChainJS to implement semantic search for your data

Foundational AI concepts

Before we start using LangChain, let’s familiarize ourselves with some core AI concepts. These are not always specific to LangChain, but they are crucial for understanding how everything works together.

Later in this article, we will see how to combine these concepts to create our custom, knowledge-based chatbot.

Large language models (LLM)

Large language models are models that can generate text based on an inputted dataset and learned patterns. Some of the most common and powerful LLMs are GPT 4 by OpenAI, which is the underlying model behind ChatGPT, and LLama 2 by Meta.

In practice, each of these models has a different API interface. LangChainJS simplifies working with them by creating a common interface.

Vector embeddings

Vector embeddings are numerical representations of objects, such as text. They are useful because machines don’t understand text — but they do understand numbers.

For an example, let’s say we want to know how similar the following sentences are:

This is a cat.This cat is beautiful.Today is Friday.

Clearly, the first two sentences are related, but the third isn’t. But a machine doesn’t necessarily know that. So, let’s assign a numerical values to each of the sentences:

Sentence	Score
This is a cat.	5
This cat is beautiful.	6
Today is Friday.	10

Now, a machine will understand that five and six are closer than 10. This is why we need to use vector embeddings to convert texts to numbers.

Exploring LangChain modules

Now that we have a better understanding of some AI concepts, let's explore different LangChainJS modules to understand the library's power.

Loading data with document loaders

All LLMs are trained on a huge amount of data. ChatGPT is not inventing anything new; it's just recognizing patterns based on the data it was trained on. So, if you want these LLMs to understand your business, you need to feed a lot of data into them.

Over 200k developers use LogRocket to create better digital experiencesLearn more →

Francisco Quintero
@Fran_Quintero · Follow

I need to give a special shout-out to @LogRocket. Drastically cut our debugging time in an issue that got reported this morning. Top 5 tools in our startup arsenal without a doubt. #startuplife

10:51 AM · Jan 8, 2022

Whether you have your data in a webpage, Excel sheet, or a bunch of text files, LangChain will be able to collect and process all of these data sources using document loaders.

Document loaders are utility functions that help extract data from different sources. There are two kinds of loaders, which we'll explore below.

File loaders

File loaders can import data from files or blob objects, accommodating a variety of formats including TXT, CSV, PDF, JSON, Docs, and more.

The following is a simple example of loading data from a text file:

```
import { TextLoader } from "langchain/document_loaders/fs/tex
t";

const loader = new TextLoader("./example.txt");
const docs = await loader.load();
```

This is just a function call, and LangChainJS takes care of the rest. You can use `docs` to feed your LLM or use it to generate vector embeddings.

Web loaders

Web loaders can process data from web sources. If your data lives on a webpage, such as your portfolio, company website, or maybe your company's Notion docs, and you want to feed this data into AI, you would use web loaders.

LangChain uses different headless browsers or web crawlers like Puppeteer, Cheerio, etc., to read documents online. The code below is an example of LangChain using Puppeteer to scrape data from a webpage:

```
import { PuppeteerWebBaseLoader } from "langchain/document_lo
aders/web/puppeteer";

const loader = new PuppeteerWebBaseLoader("https://www.mohamm
adfaisal.dev/about-me");

const docs = await loader.load();
```

How simple is that? It abstracts away all the complexity of writing your own web crawler. You can explore more options for document loaders in the LangChain

documentation.

Transforming data with document transformers

Loading your data is just half of the story. Now, you need to clean it and arrange it in the proper format.

If you have scraped data from some web source using Puppeteer, the data is probably in HTML format, and not very useful. LangChain gives you instructions on how to get clean data from HTML. But if your data is too big, the LLMs can't handle it because they have a token input limit.

To solve this, you need to split your data in a logical manner, which you can do using document transformers. Document transformers allow you to:

Split data/text into small and semantically meaningful chunksCombine small chunks of data into large, logical chunks

You have full control over how to split your data. Let's say you have a large piece of text and you want to split it into small chunks. The following example is a way to do that using the character text splitter:

```
 import { Document } from "langchain/document";
import { CharacterTextSplitter } from "langchain/text_splitte
r";

const text = "foo bar baz 123";
const splitter = new CharacterTextSplitter({
 separator: ", "
 chunkSize: 7,
 chunkOverlap: 3,
});
const output = await splitter.createDocuments([text]);
```

Explore more options for text splitters in the documentation.

Generating vector embeddings

Earlier, we introduced embeddings as a way to convert text into numbers with the help of LLMs. Unfortunately, each LLM has a different interface, which makes it difficult to work with different models.

This section will explore the process of generating vector embeddings and how LangChainJS makes it easy, enabling the utilization of different models.

Generating embeddings for your documents

The syntax to create embeddings for OpenAI's embedding generator is as follows. First, create the instance of `openai-embedding`:

```
import { OpenAIEmbeddings } from "langchain/embeddings/openai";

const embeddings = new OpenAIEmbeddings({
  openAIApiKey: "YOUR-API-KEY,"  // Replace the key with your own open API key,
  batchSize: 512, // Default value if omitted is 512. Max is
2048
});
```

Now, you can use the `embeddings` object to generate embeddings for your documents:

```
// Thesedocs can be the output from your document loaders or
transformers const docs = [ { pageContent: "The sun rises in
the east", metadata: { // this can be any thing you want // f
or example, you can store the url of the page here }, }, ]; c
onst embeddings = await embeddings.generate(docs);
```

Storing the generated embeddings

Now that you have generated the embeddings, it's time to store them in a database. But we can't just store them in any database; we need to store them in a database that supports vector embeddings.

There are many options available, such as Pinecone, Chroma, and HNSWLib. Here is a brief guide to help you pick the appropriate vector store:

Choose Pinecone if you want a production-ready databaseChoose Chroma if you want to host your own databaseChoose HNSWLib if you are doing some testing locally

Again, LangChainJS provides a simple interface to interact with all these vector stores, eliminating the need for you to learn each one individually.

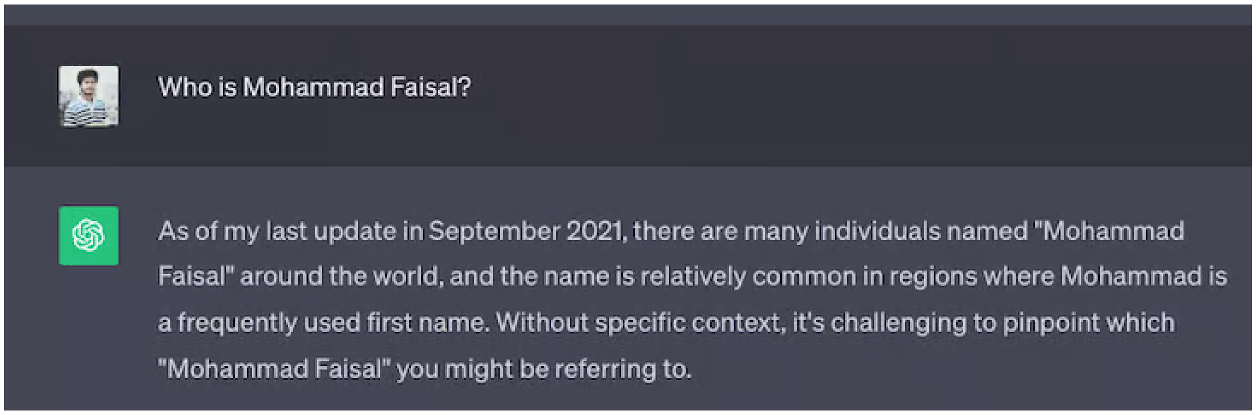
Below is an example of storing the embeddings in a local HNSWLib database:

```
import { HNSWLib } from "langchain/vectorstores/hnswlib";
import { OpenAIEmbeddings } from "langchain/embeddings/opena
i";

const vectorStore = await HNSWLib.fromTexts(
  ["Hello world", "Bye bye", "hello nice world"],
  [{ id: 2 }, { id: 1 }, { id: 3 }],
  new OpenAIEmbeddings(),
);

const resultOne = await vectorStore.similaritySearch("hello w
orld", 1);
console.log(resultOne);
```

Building a custom chatbot using LangChainJS

When using ChatGPT, its responses are limited to the knowledge available in the model. You can only ask questions about the data it has been trained on. If the relevant data doesn't exist, ChatGPT is unable to assist.

For example, it doesn't know about anyone named Mohammad Faisal!



Who is Mohammad Faisal?

As of my last update in September 2021, there are many individuals named "Mohammad Faisal" around the world, and the name is relatively common in regions where Mohammad is a frequently used first name. Without specific context, it's challenging to pinpoint which "Mohammad Faisal" you might be referring to.

But won't it be cool if an AI model could answer questions about you or your business? In the following sections, we will build a custom knowledge chatbot to answer questions based on a custom data set.

Specifically, we will build:

A system to extract data from a data source (in this case, my portfolio)A vector database of our custom dataA chatbot that can answer the question (in this case, about me)

Prerequisites

To follow along with this tutorial, you need an OpenAI API key. You can get one for free here. Let's first understand the two major steps to this process:

Training the chatbot

◦

Preparing the data (loaders and transformers)

◦

Generating embeddings

◦

Storing the embeddings in a database (vector stores)Using the chatbot

◦

Creating embeddings for the question

◦

Searching the vector database for the most similar content based on the question

◦

Passing the question and its related content as context to the LLM in order to get the answer

We will see how LangChainJS makes it easy to do all these steps. For simplicity, we will use a simple text file as our data source (.txt file), a local database to store the embeddings (HNSWLib), and OpenAI's embedding generator to generate embeddings (OpenAIEmbeddings).

Setting up a Node.js project

First, let's start a simple Node.js project. Go to the terminal and run the following commands:

```
mkdir langchainjs-demo
cd langchainjs-demo
npm init -y
```

This will initialize an empty Node project for us. Now, let's install LangChain and `hnswlib-node` to store embeddings locally:

```
npm install langchain hnswlib-node
```

Then, create a file named `index.js` as an entry point to our Node application and another file called `training-data.txt` to act as our data source:

```
touch index.js training-data.txt
```

You can paste any data into your `training-data.txt` file as your custom knowledge base.

Training the model

The complete code prepares the data, creates embeddings, and stores them in a database. Each line of code is explained in the comments below:

```
const { OpenAIEmbeddings } = require("langchain/embeddings/openai");
const { RecursiveCharacterTextSplitter } = require("langchain/text_splitter");
const { HNSWLib } = require("langchain/vectorstores/hnswlib");
const fs = require("fs");

const OPENAI_API_KEY = "YOUR_API_KEY_GOES_HERE";

async function generateAndStoreEmbeddings() {

  // STEP 1: Load the data
  const trainingText = fs.readFileSync("training-data.txt", "utf8");

  // STEP 2: Split the data into chunks
  const textSplitter = new RecursiveCharacterTextSplitter({
    chunkSize: 1000,
  });

  // STEP 3: Create documents
  const docs = await textSplitter.createDocuments([trainingText]);

  // STEP 4: Generate embeddings from documents
  const vectorStore = await HNSWLib.fromDocuments(
    docs,
    new OpenAIEmbeddings({ openAIApiKey: OPENAI_API_KEY }),
  );

  // STEP 5: Save the vector store
  vectorStore.save("hnswlib");
}
```

Now, you have a local database containing your documents' embeddings.

Using the chatbot

To use the chatbot, we need a function that takes a question as its input and returns the answer. This is what the complete code looks like:

```
const { OpenAI } = require("langchain/llms/openai");
const { OpenAIEmbeddings } = require("langchain/embeddings/op
enai");
const { HNSWLib } = require("langchain/vectorstores/hnswli
b");
const { RetrievalQAChain, loadQARefineChain } = require("lang
chain/chains");

const OPENAI_API_KEY = "YOUR_API_KEY_HERE";
const model = new OpenAI({ openAIApiKey: OPENAI_API_KEY, temp
erature: 0.9 });

async function getAnswer(question) {

  // STEP 1: Load the vector store
  const vectorStore = await HNSWLib.load(
    "hnswlib",
    new OpenAIEmbeddings({ openAIApiKey: OPENAI_API_KEY }),
  );
  // STEP 2: Create the chain
  const chain = new RetrievalQAChain({
    combineDocumentsChain: loadQARefineChain(model),
    retriever: vectorStore.asRetriever(),
  });

  // STEP 3: Get the answer
  const result = await chain.call({
    query: question,
  });

  return result.output_text;
}
```

Now, you have a custom chatbot that can answer questions related to your business. By using the specified functions, you can also easily run your custom chatbot. To further simplify interactions with the chatbot, I've created a simple Express API using the above functions. You can call these functions to run your custom chatbot.

Finally, to run our chatbot, we need to train our data. Get your `OPENAI_API_KEY` and include it in the `.env` file. Then, open the `trainig-data.txt` file and add some details about you or your business. For this example, I just copied the contents of my about page.

Run the express API with the following commands:

```
npm install
```

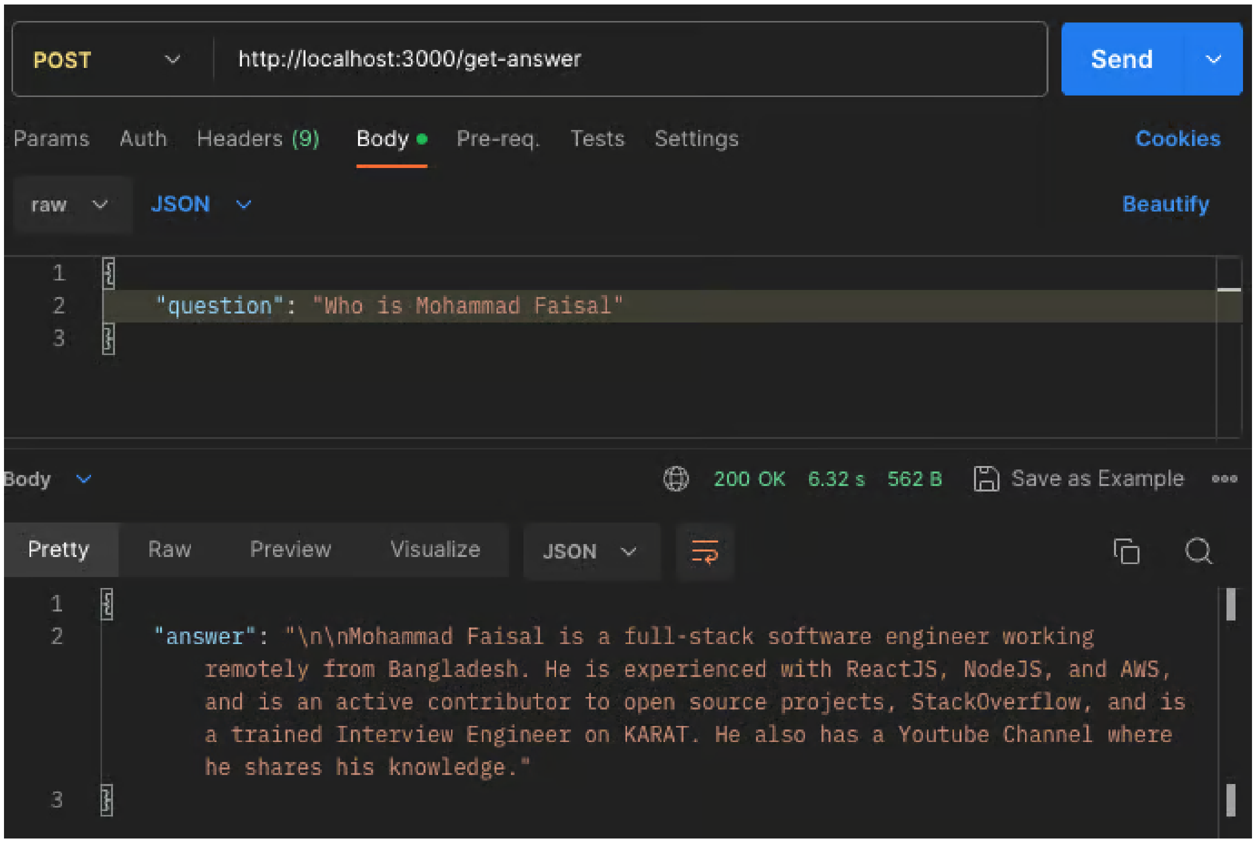
```
npm start
```

Now, hit the `http://localhost:3000/train-bot` endpoint to build the vector database based on your data:

`curl http://localhost:3000/train-bot`

And your custom knowledge base is ready!

Let's ask the chatbot a question. Hit the `http://localhost:3000/get-answer` endpoint with the question we asked ChatGPT earlier:



```
{
    "question": "Who is Mohammad Faisal"
}
```

```
{
    "answer": "\n\nMohammad Faisal is a full-stack software engineer working
        remotely from Bangladesh. He is experienced with ReactJS, NodeJS, and AWS,
        and is an active contributor to open source projects, StackOverflow, and is
        a trained Interview Engineer on KARAT. He also has a Youtube Channel where
        he shares his knowledge."
}
```

And sure enough, it will provide a detailed answer based on the data it was trained on. How cool is that!

Conclusion

Hopefully, you now have a better understanding of what LangChainJS is and how it can help us build AI-powered applications. Now it's your turn to create something new!

Thank you for reading this far. I hope you learned something new today. Have a wonderful rest of your day!

LogRocket