Jacob Orshalick
https://linkedin.com/in/jorshalick

The Busy **Developers** **Guide** to **GEN AI**

JavaScript Code Tutorials

LangChain

express

OpenAI

Preface

This mini-book is here to jumpstart your AI learning journey. You won't need to dust off your Python spellbook, train neural networks, or get a PhD in statistics. All you need is some basic JavaScript knowledge and a willingness to learn.

We'll skip all the data science wizardry by using pre-trained generative AI models (specifically LLMs). APIs and SDKs have made these models accessible to developers in the languages they're already building applications with.

This book will guide you through combining your own data with the power of LLMs using a technique called RAG (Retrieval Augmented Generation). During this journey, you'll learn important concepts while gaining practical experience with some key JavaScript libraries: Node.js, Express, and LangChain.js.

If you want to see the source code right away you can download it here:

https://github.com/jorshali/ai-for-developers

So come on. Let's get your AI learning journey started. It's never been easier.

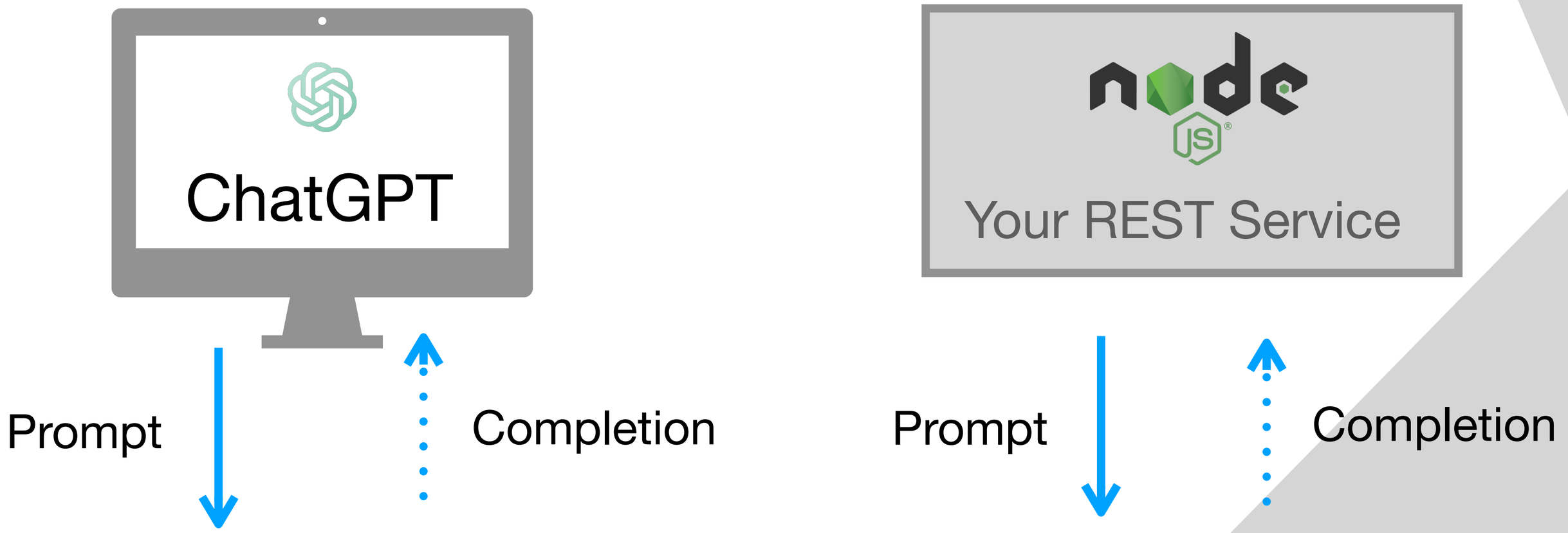
Part 1

Build a REST Service that calls an LLM

ChatGPT vaulted generative AI into mainstream culture. But, it's really just a user interface, powered by the true marvel that lies beneath—the large language model (LLM).

More precisely, LLMs are very large deep learning models that are pre-trained on vast amounts of data. The keyword there is pre-trained.

All we have to do to make use of these same models is send them a prompt telling it what we want. We can do that by calling the OpenAI APIs.



We'll get started by setting up a new project for our REST service.

1. Install Node

- Download and install: https://nodejs.org
- Verify the install in your terminal:

```
~ % node -v
```

- If the installation succeeded, the version will print

2. Initialize your project

- Create a new directory for your project.
- Navigate to it in your terminal.
- Run the following command:

```
~/ai-for-devs % npm init -y
```

- Creates a new package.json file, initializing the project.

3. Install Node modules

- The node modules we'll be using:
  - `express`: which makes server creation quick and easy
  - `langchain`: which provides a framework for building Apps with LLMs
  - `@langchain/openai`: which provides OpenAI integrations through their SDK
  - `cors`: Express middleware to enable CORS
- In the same terminal, run the following command:

```
~/ai-for-devs % npm install express \
  langchain @langchain/openai cors
```

- This command will download the necessary modules into your project.

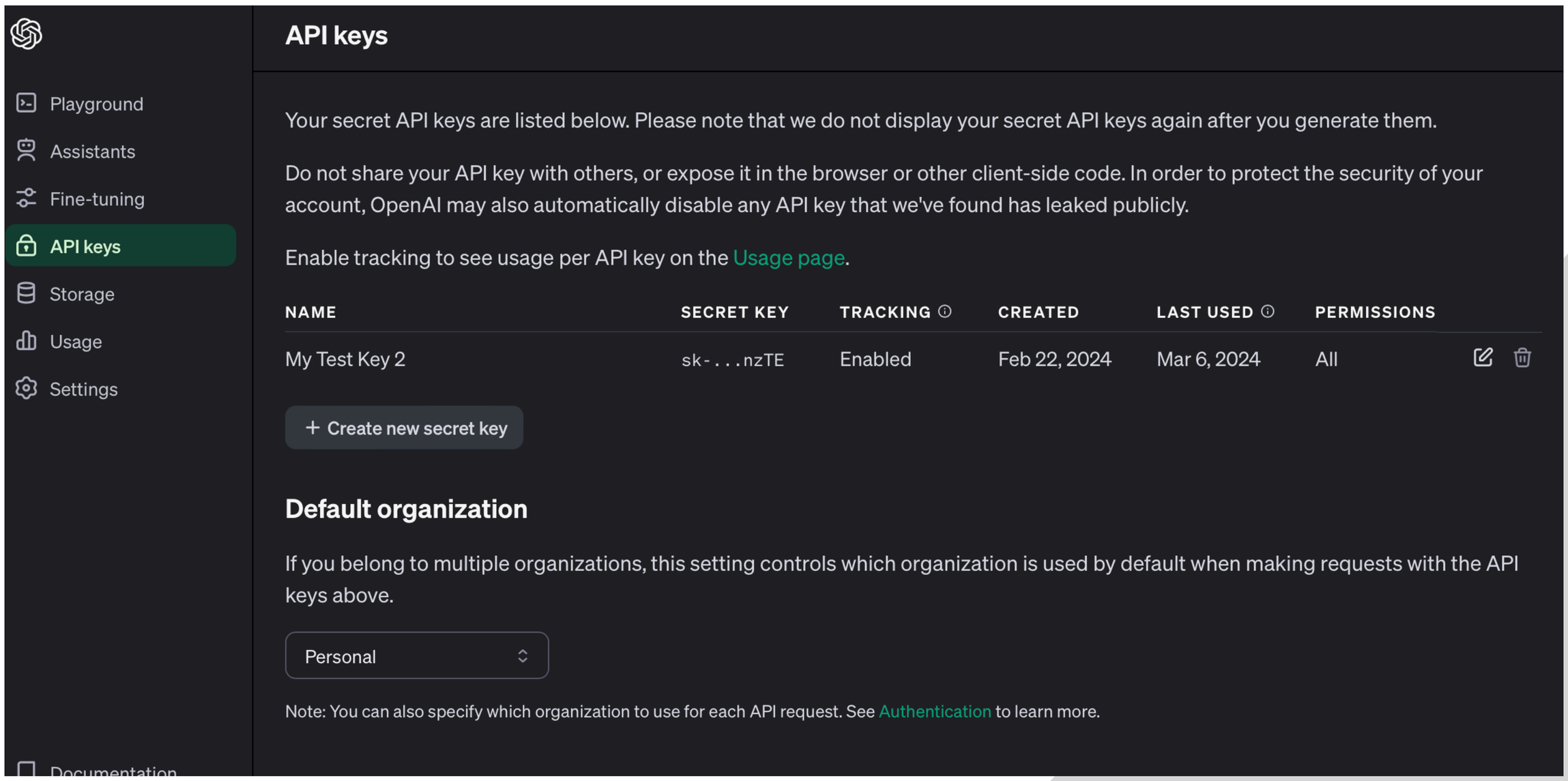
4. Create the server file

- Create a file called server.mjs in the project directory.
- Open it in a text editor and add the following lines of code:

```
import express from "express";

const app = express();

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

const chatModel = new ChatOpenAI({});

app.get('/', async (req, res) => {
  const response =
    await chatModel.invoke(
      "Can you simply say 'test'?");

  res.send(response.content);
});

app.listen(3000, () => {
  console.log(`Server is running on port 3000`);
});
```

5. Create an OpenAI account

- Register here: https://platform.openai.com
- Obtain an API key:
  - Simply select ‘API keys’ in the upper left navigation
  - Select ‘+ Create new secret key’
  - Copy the key somewhere safe for now



6. Set an environment variable

- In the same terminal, run the following command with your key value:

```
~/ai-for-devs % export \
  OPENAI_API_KEY=<YOUR_KEY_VALUE>
```

- *Optionally* add this command to your bash profile: `~/.zshrc`

7. Launch your server

- Back in the terminal, run the following command:

```
~/node-openai % node server.mjs
```

- Open your web browser and visit: http://localhost:3000
- You'll see the response from the OpenAI model: test

Congratulations!

You've successfully established a functional REST service. Beyond its ability to prompt an AI and generate responses, it forms the foundation for the upcoming sections of this book. In Part 2, we'll explore the process of streaming longer responses so our users don’t have to wait.

Here are some things you can try on your own:

- Try passing a query parameter to your REST service for the prompt. Simply run a web search (or use ChatGPT) to find out how you can read query parameters with Express.

- Look at LangChain’s `ChatPromptTemplate`. See if you can create a template and pass in what you want the AI to say.

Quick Tip

Pay attention to your usage in the OpenAI UI. You’ll notice that calling the default model (currently GPT-3.5-turbo) is inexpensive, but does build up charges over time. You can view your usage here: https://platform.openai.com/usage

https://github.com/jorshali/ai-for-developers/part1

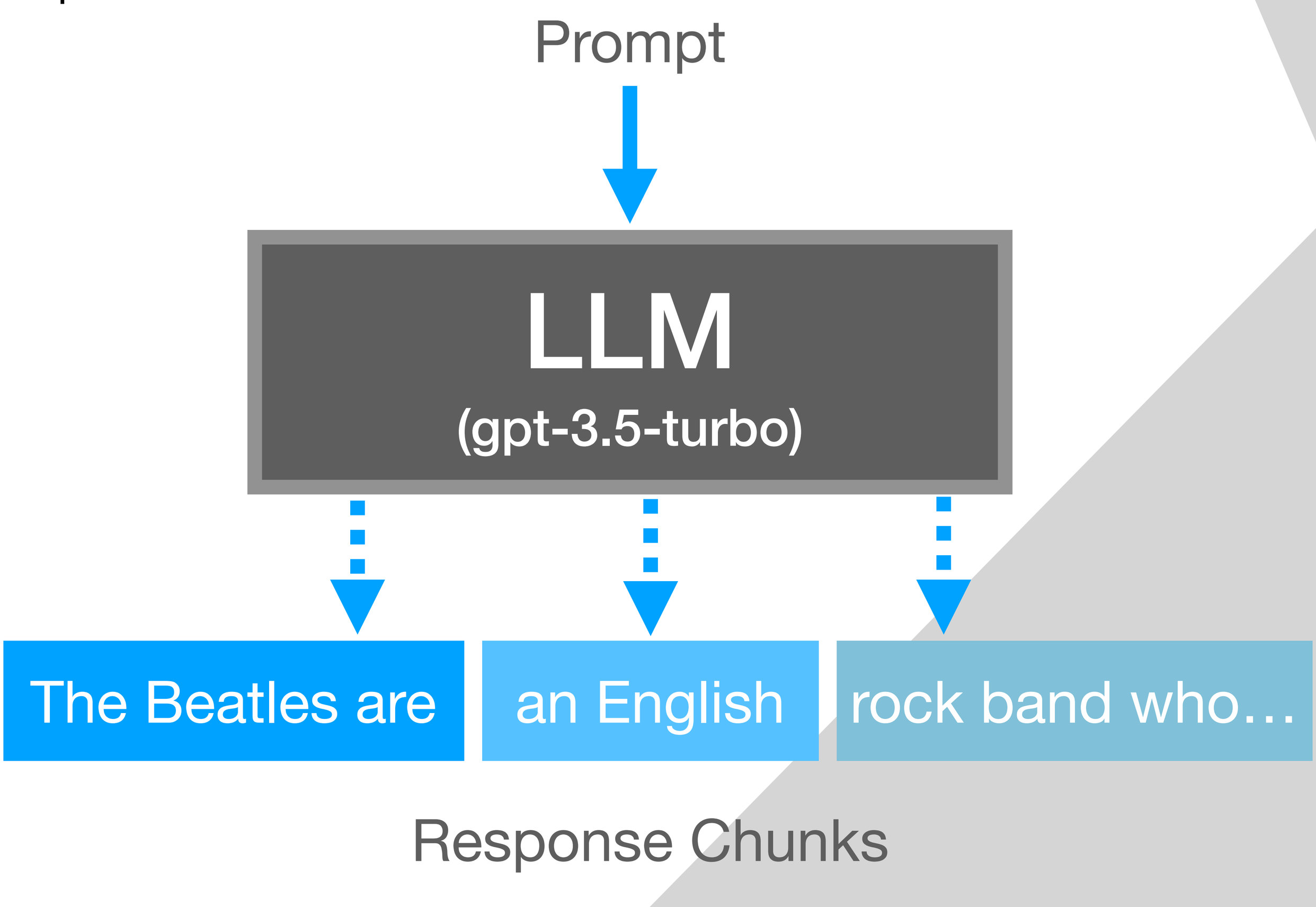
Part 2

Stream your LLM Responses

The cool typing effect you see in ChatGPT makes it feel responsive. Without it, the UI would feel slow and often unbearable.

OpenAI ingeniously transformed a drawback of Large Language Models (LLMs) — their text generation speed — into a captivating UI effect. Achieving a similar level of responsiveness is possible by employing LLM response streaming.

When you prompt an LLM, it generates the response in chunks. Streaming allows you to send these chunks back to the user when they are ready. As your prompts and chains grow in size and complexity, streaming becomes crucial to performance.



Now that we have our REST service, we can build upon it to stream the LLM responses.

1. Stream an LLM response with LangChain

- LangChain.js provides the `stream` method through its `Runnable` protocol
- LLMs, parsers, prompts, retrievers, and agents are all `Runnable`
- This allows you to simply call `stream` on `ChatOpenAI` to receive the response in chunks

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

const chatModel = new ChatOpenAI({});

const stream = await chatModel.stream(
  "Tell me about the Beatles " +
  "in 50 words or less.");
```

2. Streaming a REST response with Express

- Express allows you to stream a response to a client
- When the stream yields a response chunk, we can immediately send it to the client
- The Response object provides the write method for this:

```
const chatModel = new ChatOpenAI({});

app.get('/', async (request, response) => {
  const stream = await chatModel.stream(
    "Tell me about the Beatles " +
    "in 50 words or less.");

  for await (const chunk of stream) {
    response.write(chunk.content);
  }

  response.end();
});
```

3. Create the entire server file with Express

- You can replace your `server.mjs` file with the following to stream the LLM response to a client

```
import express from "express";
import { ChatOpenAI } from "@langchain/openai";
import cors from 'cors';

const app = express();

const chatModel = new ChatOpenAI({});

app.use(cors());

app.get('/', async (req, res) => {
  const stream = await chatModel.stream(
    "Tell me about the Beatles " +
    "in 50 words or less.");

  for await (const chunk of stream) {
    res.write(chunk.content);
  }

  res.end();
});

app.listen(3000, () => {
  console.log(`Server is running on port 3000`);
});
```

4. Streaming to the UI with `fetch`

- If you wanted to use JavaScript to render the response chunks as they are returned to UI, you can use `fetch`
- The `fetch` API allows you to call your server and read the server response as a `ReadableStream`
- You simply call `getReader()` on the body of the response so you can read each chunk as it's returned
- The example source code includes a simple `client` that uses fetch to stream the response

```
const url = 'http://localhost:3000';
const response = await fetch(url);

const reader = response.body.getReader();
```

5. Decode the response chunks and render

- The response chunks you receive are returned as a stream of bytes
- To get the actual text, pipe the response through `TextDecoderStream`
- The `fetchData` function below renders the chunks of text as they are returned

```
const fetchData = async () => {
  const url = 'http://localhost:3000';
  const response = await fetch(url);

  const reader = response.body
    .pipeThrough(new TextDecoderStream())
    .getReader();

  while (true) {
    const { done, value } = await reader.read();

    renderResponseChunk(value);

    if (done) {
      return;
    }
  }
};
```

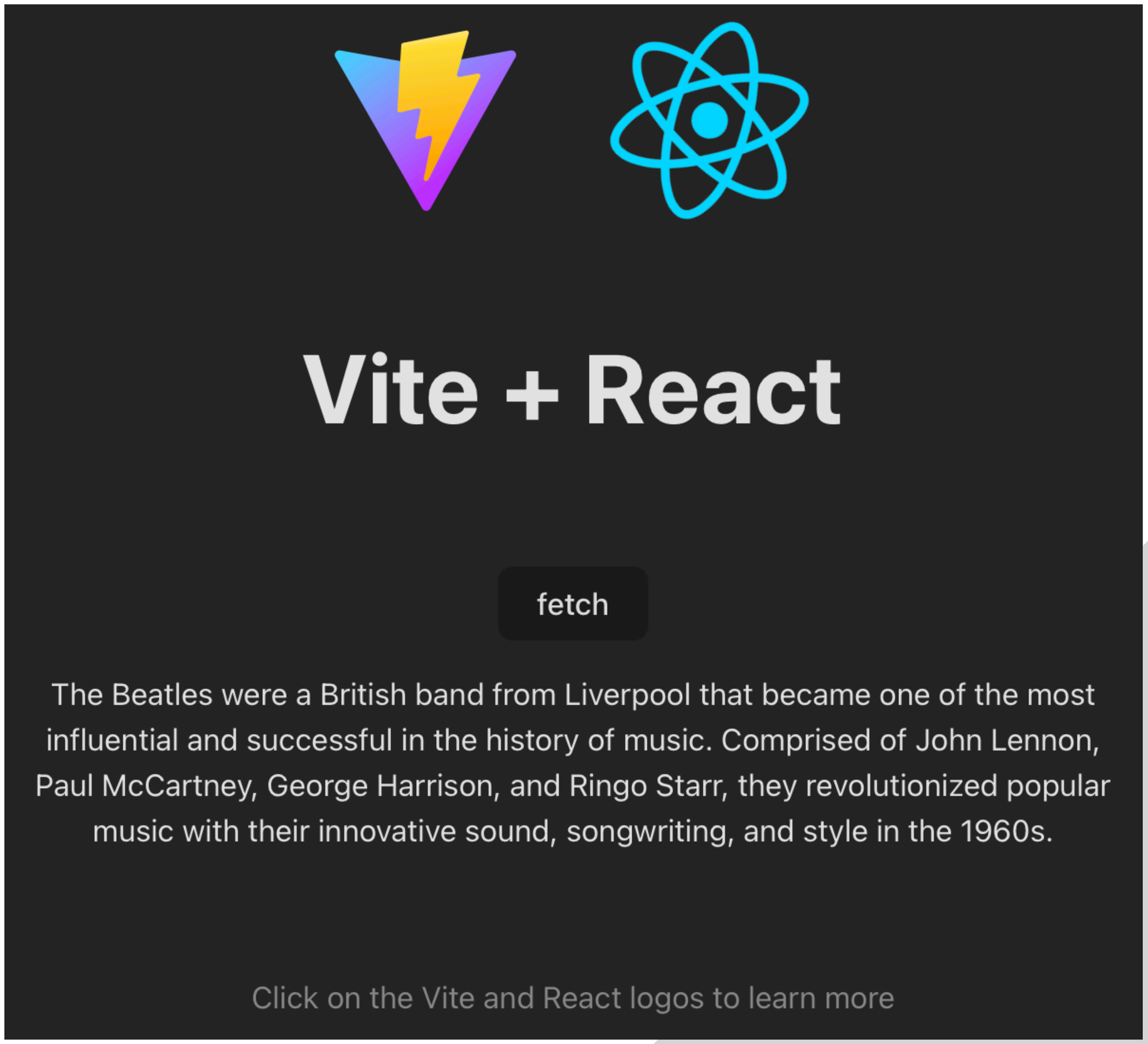
6. Try out the client in the example code

- You can find the source code for part 2 on GitHub:

https://github.com/jorshali/ai-for-developers/part2

- The README file provides the instructions to get the example up and running

- Once your up-and-running you can see the LLM streaming in action



Congratulations!

You’re now streaming LLM responses from a server to a client. In part 3, we'll delve into more intriguing territory as we begin the process of integrating our own data with the LLM.

Here are some things you can try on your own:

- See just how many classes extend `Runnable` in LangChain. It shows just how much streaming can be used throughout chains to enhance performance.
- If you tried passing the prompt as a parameter to your REST service, see if you can modify the example client to add a prompt input.

https://github.com/jorshali/ai-for-developers/part2

Part 3

Create a Vector Search with custom data

LLMs become much more interesting when we use them with our data. The problem is, an LLM is limited to the data it was trained on. At the same time, companies don't want to risk their proprietary data being exposed through something like ChatGPT. Samsung learned this the hard way…

The game-changer here is RAG, which stands for Retrieval Augmented Generation. In the most simple terms, RAG is how you get *your data* to an LLM so it can use it when generating responses.

This involves:

- Efficiently fetching the data you need
- Serving it up to an LLM
- Giving the LLM instruction on how to make the most of it
- Letting users ask their questions about it

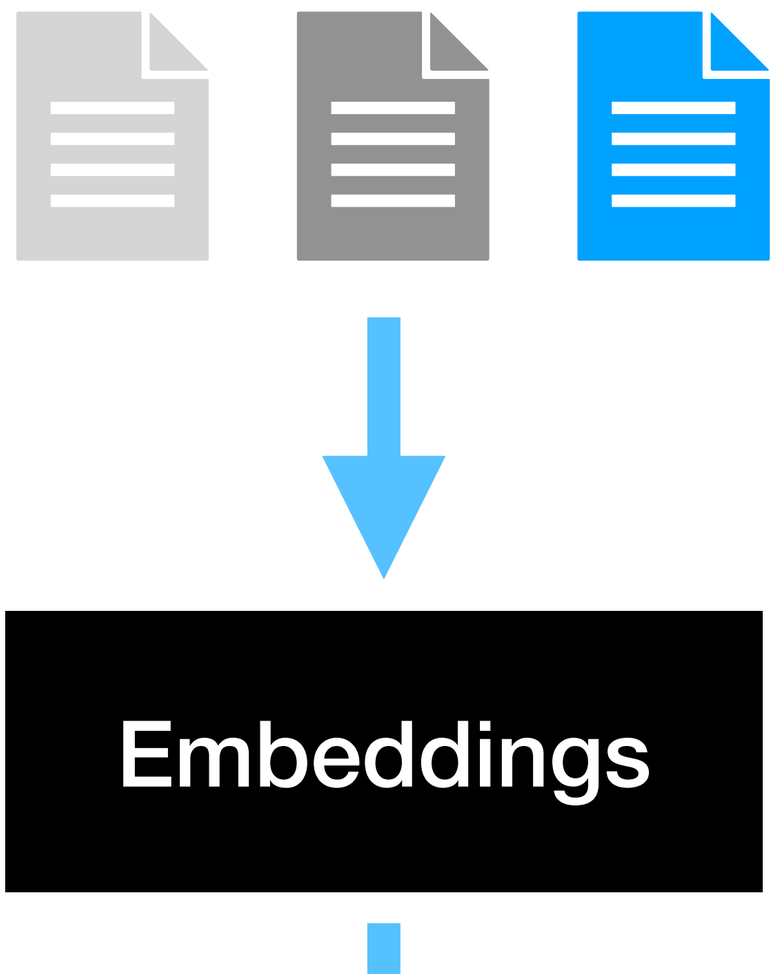
But that's not all – RAG means you don't need to "train" an AI on your data. No need to summon AI researchers like wizards; your organization can dive into the AI game right now.

So, how's this magic done? The first step is finding the right data and Vector Search is a great solution.

Things to keep in mind:

- prompts have a token limit
- LLM providers often charge by the token
- including irrelevant data may confuse the LLM response

Vector Search allows you to search your data based on the meaning of text instead of keywords. Your documents and search queries get converted to numeric vectors by an embedding provider. The Vector Store applies an algorithm to find the most similar vector during search.



Now let's setup a Vector Store with custom data.

1. Install Node dependencies

- We'll be using LangChain.js with `HNSWLib` to build our Vector Search:
  - `HNSWLib` provides a Vector Store that can be saved and loaded from disk
  - The `@langchain/community` module provides a simple API for interacting with `HNSWLib`
- Install the dependencies in a Node project by running the following in your terminal:

```
~/ai-for-developers/part3 % npm install \
  hnswlib-node @langchain/community
```

Quick Tip

The `@langchain/community` module contains third-party integrations for many more tools than `HNSWLib`. Have a look at the LangChain documentation to learn more.

2. Load documents to search with LangChain

- LangChain provides a variety of ways to load your documents:
  - `DirectoryLoader`: loads documents in a directory and additional loaders are specified to handle file types
  - `TextLoader`: loads a single text document from a disk path

```
import { DirectoryLoader } from
  "langchain/document_loaders/fs/directory";
import { TextLoader } from
  "langchain/document_loaders/fs/text";

const loader = new DirectoryLoader(
  "companies",
  {
    ".txt": (path) => new TextLoader(path)
  }
);

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

3. Create a Vector Store with `HNSWLib`

- We simply call the `fromDocuments` method on `HNSWLib` with our loaded documents
- Notice we also pass `OpenAIEmbeddings`
- `OpenAIEmbeddings` is used to generate the vectors from our documents using Open AI APIs

```
import { HNSWLib } from
  "@langchain/community/vectorstores/hnswlib";
import { OpenAIEmbeddings } from
  "@langchain/openai";

const loader = new DirectoryLoader(
  "companies",
  {
    ".txt": (path) => new TextLoader(path)
  }
);

const docs = await loader.load();

const vectorStore = await HNSWLib.fromDocuments(
  docs, new OpenAIEmbeddings());
```

4. Search the Vector Store

- Now we can ask the `vectorStore` a question in natural language
- The `similaritySearch` is passed our question and a *k* value (*k*-nearest neighbor or said in a simpler way, the number of results we want)
- The `vectorStore` will once again use `OpenAIEmbeddings` to create a vector from our question via the Open AI APIs
- That vector will be compared to the document vectors to find the closest match (our result)

```
const vectorStore = await HNSWLib.fromDocuments(
  docs, new OpenAIEmbeddings());

const resultOne = await vectorStore.similaritySearch(
  "Can you tell me about Acme Corp?", 1);
```

5. Save and load the Vector Store

- Recreating the vectors every time you start your Vector Store would be costly
- The `save` method from `HNSWLib` allows to provide a directory to save your Vector Store on disk
- You can then `load` your Vector Store from that directory

```
const embeddings = new OpenAIEmbeddings();

const vectorStore = await HNSWLib.fromDocuments(
  docs, embeddings);

const dataDirectory = 'data';

await vectorStore.save(dataDirectory);

const loadedVectorStore = await HNSWLib.load(
  dataDirectory, embeddings);
```

6. Create a file that loads the Vector Store

- Now we can create a `loadData.mjs` file that simply loads and saves the Vector Store using `HNSWLib`

```
import { DirectoryLoader } from
  "langchain/document_loaders/fs/directory";
import { TextLoader } from
  "langchain/document_loaders/fs/text";
import { HNSWLib } from
  "@langchain/community/vectorstores/hnswlib";
import { OpenAIEmbeddings } from
  "@langchain/openai";

const loader = new DirectoryLoader(
  "companies",
  {
    ".txt": (path) => new TextLoader(path)
  }
);

const docs = await loader.load();
const embeddings = new OpenAIEmbeddings();

const vectorStore = await HNSWLib.fromDocuments(
  docs, embeddings);

await vectorStore.save('data');
```

7. Modify your server file to perform the search

- We can modify our `server.mjs` file to simply load the Vector Store from disk at startup

```
import { HNSWLib } from
  "@langchain/community/vectorstores/hnswlib";
import { OpenAIEmbeddings } from
  "@langchain/openai";

import express from "express";
import cors from 'cors';

const app = express();

app.use(cors());
app.use(express.json());

const embeddings = new OpenAIEmbeddings();

const loadedVectorStore = await HNSWLib.load(
  'data', embeddings);

app.get('/', async (request, response) => {
  const resultOne = await loadedVectorStore.similaritySearch(
    "Can you tell me about Acme Corp?", 1);

  response.send(resultOne[0].pageContent);
});

app.listen(3000, () => {
  console.log(`Server is running on port 3000`);
});
```

8. Now you can load your Vector Store

- Run the following terminal command to load the Vector Store:

```
~/ai-for-devs/part3 % node loadData.mjs
```

9. Launch your server

- Back in the terminal, run the following command:

```
~/ai-for-devs/part3 % node server.mjs
```

- Open your web browser and visit: http://localhost:3000
- You'll see the search result in your browser.

Congratulations!

You've created your own Vector Search! In part 4 we will complete the RAG solution by adding back our call to the LLM along with our Vector Search results.

Here are some things you can try on your own:

- Try your own data with the Vector Store. You can change the `companies/company*.txt` files to any data files you want. You can then send in prompts to see if the data you expect is returned.
- Try changing the *k* value and modify the prompt to see how many results are returned.

Quick Tip

Have another look at your usage in the OpenAI UI. You'll now notice embedding model usage along with your GPT model usage. By saving your Vector Store to disk you only limit the cost of embedding model usage. You can view your usage here: https://platform.openai.com/usage

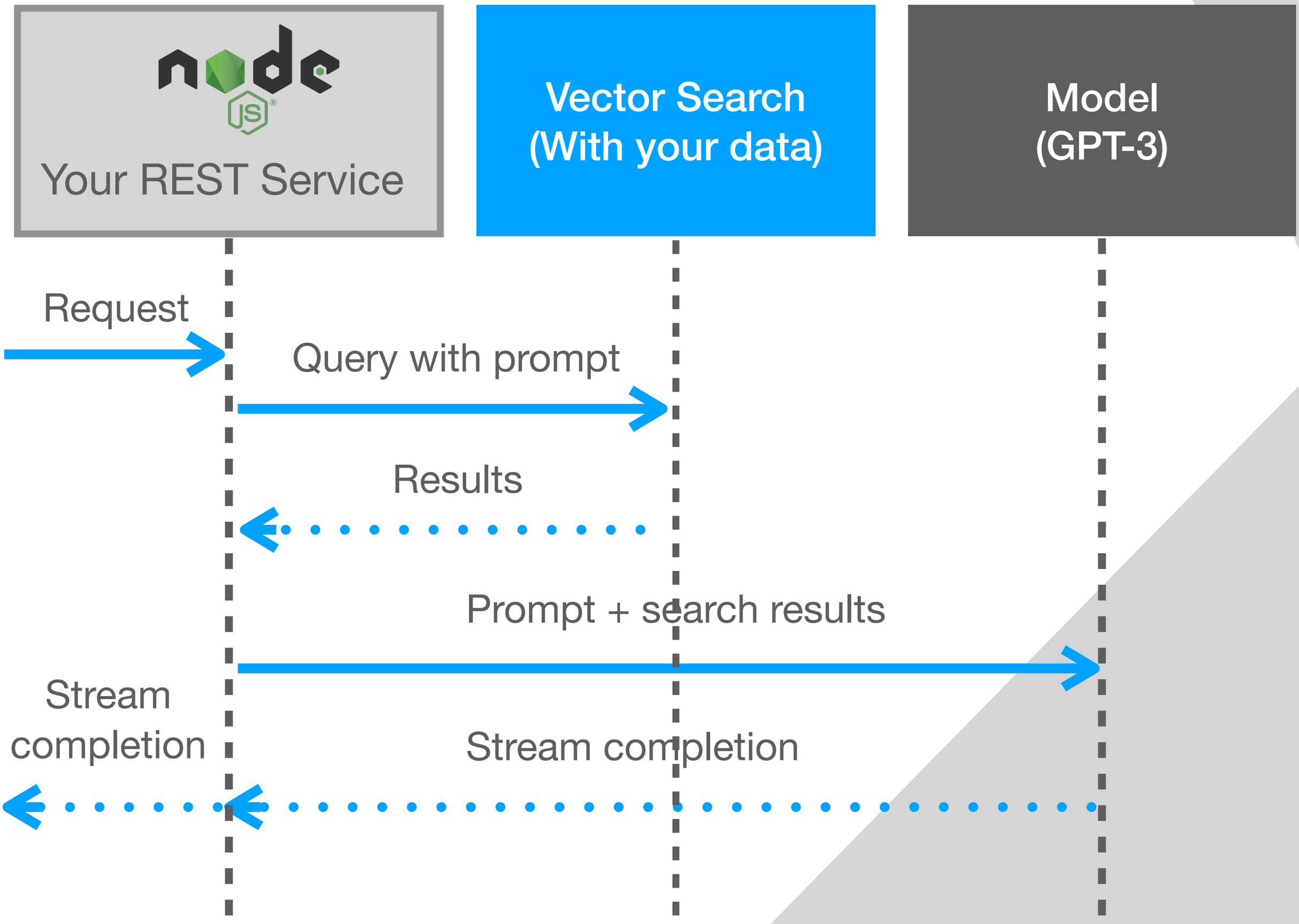
https://github.com/jorshali/ai-for-developers/part3

Part 4

Supercharge your data with RAG

We're almost there. We have all the necessary pieces in place, it's now just a matter of combining them to build a complete RAG solution.

Our goal is simple really. When given a prompt, we'll first find any related data using our Vector Search. We'll then include that data with the prompt as *context* when calling the LLM. Finally, we'll stream the generated response back to the caller.



Now let's combine our prompts with vector search.

1. Create a prompt template

- We need a way to include the search results from our Vector Store in the prompt along with instructions
- `ChatPromptTemplate` from LangChain allows us to create flexible templated prompts
- Any bracketed `{}` prompt content that will be replaced by variables (e.g. `{context}` and `{question}`)

```
const prompt = ChatPromptTemplate.fromMessages([
  [
    'ai',
    'Answer the question with company information based ' +
    'on only the following context:\n\n' +
    '{context}'
  ],
  ['human', '{question}'],
]);
```

Quick Tip

Notice we are passing an array of messages to build the prompt. The `'ai'` role tells the LLM that the first message is instructive. The `'human'` role of the second message tells the AI that this is what it should be responding to.

2. Provide additional instruction

- We can further instruct the LLM on how it should behave by expanding our template
- In this case, we are explaining to the LLM how it should generate job postings.

```
const prompt = ChatPromptTemplate.fromMessages([
  [
    'ai',
    'Answer the question with company information based ' +
    'on only the following context:\n\n' +
    '{context}'
  ],
  [
    'ai',
    'You are an AI assistant that generates ' +
    'job postings for recruting software developers.  The ' +
    'job posting you generate should include the technical ' +
    'skills specified as well as any additional information ' +
    'requested.  The job post should include a job title, ' +
    'role summary, duties and responsibilities, technology ' +
    'experience required, and education requirements.'
  ],
  ['human', '{question}']
]);
```

3. Define how the templated variables are filled

- `RunnableMap` allows you to execute multiple `Runnables` in parallel:
  - `context` is populated with our Vector Store search logic using `RunnableLambda` (a function definition)
  - `question` is populated by `RunnablePassthrough` which just passes through `input` as-is
- The output of the `Runnables` is returned as a map.

```
const loadedVectorStore = await HNSWLib.load(
  'data', new OpenAIEmbeddings());

const retriever = loadedVectorStore.asRetriever(1);

const setupAndRetrieval = RunnableMap.from({
  context: new RunnableLambda({
    func: (input) =>
      retriever.invoke(input).then(
        (response) => response[0].pageContent),
  }).withConfig({ runName: 'contextRetriever' }),
  question: new RunnablePassthrough(),
});
```

4. Pipe the results with the prompt to the LLM

- We can now use `setupAndRetrieval` to combine our `prompt` and `model` by *chaining* them with the pipe method
- Calling the `stream` method on `chain` passes the prompt shown as the input to the `RunnableMap` we created
- The resulting map is used to populate the template variables in our `prompt`
- The fully populated prompt is then sent to our `model` to generate a response
- Finally, include a `StringOutputParser` at the end of our chain to simply return a string as the response

```
const model = new ChatOpenAI({});
const outputParser = new StringOutputParser();

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

const stream = await chain.stream(
  'Can you provide a job posting for Acme Corp for a ' +
  'senior developer with 5-7 years of experience?');
```

5. Modify your server file

- You can replace the contents of your `server.mjs` file with the code found in the `part4` example

6. Optionally, reload your Vector Store

- Run the following terminal command to load the Vector Store:

```
~/ai-for-devs/part4 % node loadData.mjs
```

7. Launch your server

- Back in the terminal, run the following command:

```
~/ai-for-devs/part4 % node server.mjs
```

- Open your web browser and visit: http://localhost:3000
- You'll see a job posting generated for Acme Corp.

Congratulations!

You've built a complete RAG solution. You can now use your knowledge and this project to build your own applications.

Here are some things you can try on your own:

- Try connecting the client from part 2 to the RAG service you completed built in part 4. See if you can display the response in a user-friendly way.

- Try other prompts and company names based on the data found in companies. See how well the Vector Store performs to different prompts.

If you found this book helpful, please share it with others. Always feel free to connect with me on LinkedIn:

https://linkedin.com/in/jorshalick

https://github.com/jorshali/ai-for-developers/part4

Appendix: Useful Links

If you'd like to jump ahead or if you just like to look at source code you can access it here:

https://github.com/jorshali/ai-for-developers

The `README` file provides the instructions to get the examples up and running.

You can also find more information in the documentation for the technologies we'll be using:

OpenAI API Reference

https://platform.openai.com/docs/api-reference

LangChain.js

https://js.langchain.com/docs/get_started/introduction

Node.js

https://nodejs.org/docs/latest/api/

Express

https://expressjs.com/en/4x/api.html

LangChain

OpenAI