Jacob Orshalick

https://linkedin.com/in/jorshalick

Use JavaScript to

STREAM LLM RESPONSES

From a REST Service express

AI for Developers, not Data Scientists

Part 2

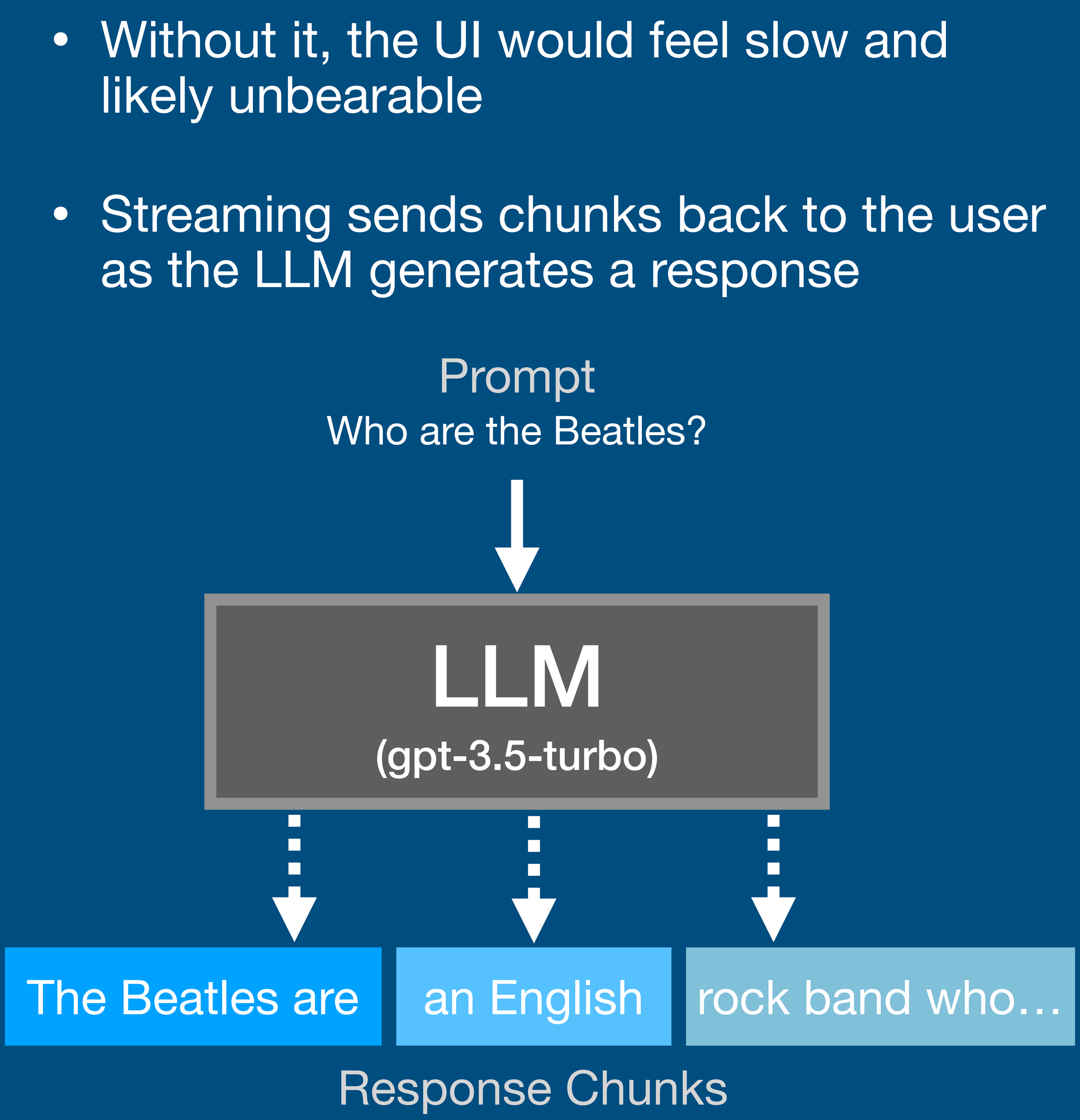
LangChain

node JS

OpenAI

What is streaming?

- Streaming is what powers that cool ChatGPT typing effect
- Without it, the UI would feel slow and likely unbearable
- Streaming sends chunks back to the user as the LLM generates a response



1 Streaming 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

- The following `server.mjs` file streams 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`

- Now in your UI, you want to render the response chunks as they are returned

- 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

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

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

5 Decode the response chunks and render to the UI

- The chunks we 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

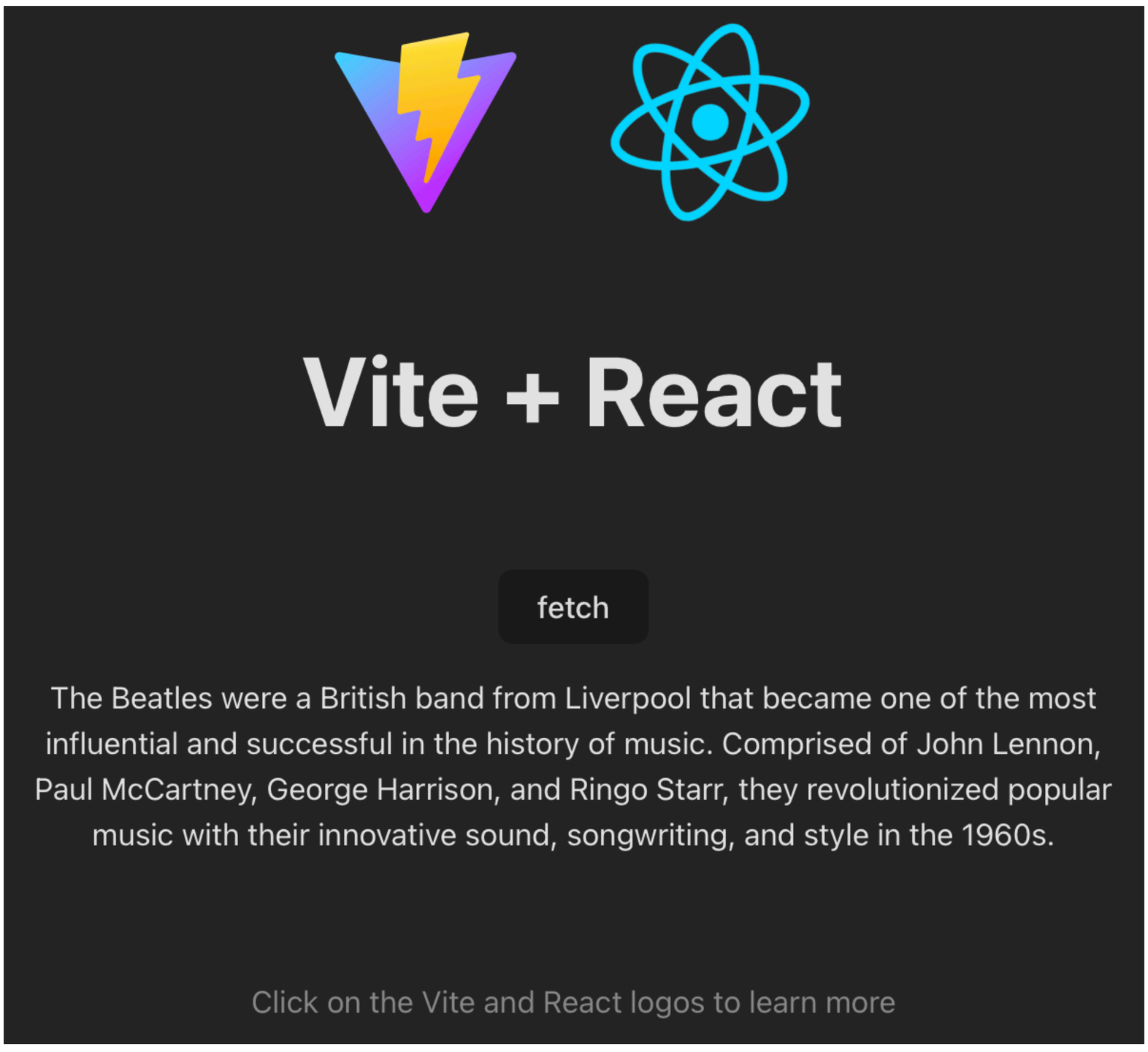
Try out the example code

- You can find the source code for this tutorial 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 can now stream LLM responses!
- Quick Tips:
  - See how many classes extend `Runnable`
  - Research the `name` attribute on `BaseMessageChunk`
  - Try to make the prompt dynamic by sending it from the UI

Jacob Orshalick

https://linkedin.com/in/jorshalick

focus.dev

IT PRO search