jonrhall / openai-streaming-hooks

Code | Issues 8 | Pull requests 2 | Actions | Projects | Security | Insights

React Hooks for streaming connections to OpenAI APIs

MIT license

Code of conduct

88 stars | 19 forks | 3 watching | Branches | Activity | Tags

Public repository

m... | 1 Branch | 1 Tag | Go to file t | Go to file | + | Add file | Code

jonrhall Merge pull request #4 from MatchuPitchu/contribution-package.json 9e81b37 · last year

example	v2.0.0 - Updated interfaces, updated unit te...	last year
src	v2.0.0 - Updated interfaces, updated unit te...	last year
test	v2.0.0 - Updated interfaces, updated unit te...	last year
.eslintignore	Initial commit	last year
.eslintrc.cjs	Updated unit tests based on removing SSE ...	last year
.gitignore	Added unit tests	last year
.prettierignore	Added Prettier to project.	last year
.prettierrc.json	Added Prettier to project.	last year
CODE_OF_CONDUCT.md	Used native ReadableStream instead of SSE...	last year
LICENSE	Added LICENSE file to project.	last year
README.md	v2.0.0 - Updated interfaces, updated unit te...	last year
package-lock.json	Updated unit tests based on removing SSE ...	last year
package.json	chore: add contribution to package.json	last year
tsconfig.json	Added Prettier to project.	last year
tsconfig.test.json	v2.0.0 - Updated interfaces, updated unit te...	last year
vite.config.ts	Added Prettier to project.	last year

README | Code of conduct | MIT license

OpenAI Streaming Hooks

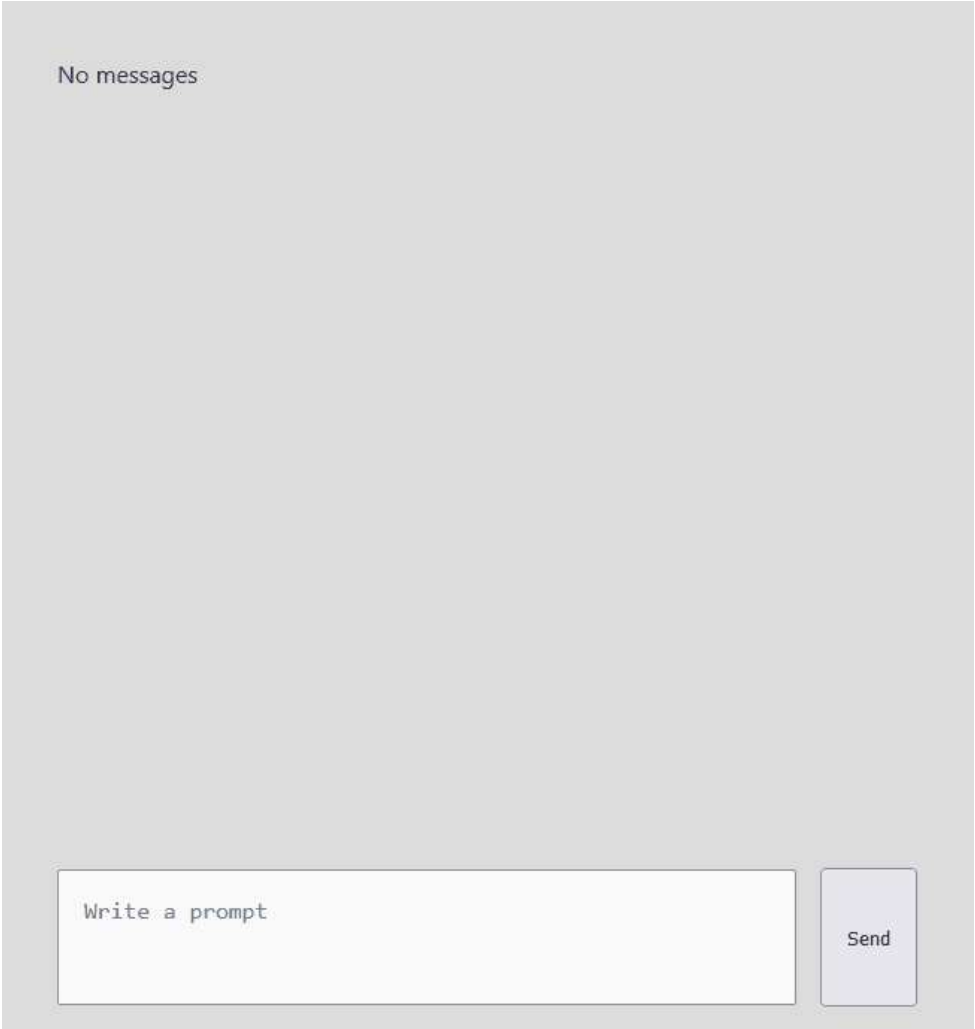
> Talk directly to OpenAI Completion APIs and stream the response back in real-time in the browser--no server required.
>
> **All models based on GPT3.5 and GPT4 are supported!**

Provides a custom React Hook capable of calling OpenAI Chat Completions APIs with streaming support enabled by ReadableStreams.

The library then decorates every Completion response with metadata about the transaction such as:

- The number of tokens used in the response
- The total time it took to complete the request
- Each chunk received in the stream
- The timestamp each chunk was received
- The timestamp from when the Completion was finished

Example



Example code here

See section on running the example for more information.

Use

1. Install the OpenAI Streaming Hooks library via a package manager like `npm` or `yarn`:

```
npm install --save openai-streaming-hooks
```

2. Import the hook and use it:

```
import { useChatCompletion } from 'openai-streaming-hooks';

const Component = () => {
  const { messages, submitPrompt } = useChatCompletion({
    model: 'gpt-3.5-turbo', // Required
    apiKey: 'your-api-key', // Required
    temperature: 0.9,
  });
  ...
};
```

Supported Model Parameters

All API parameters supported by the OpenAI Chat API are also supported here.

For example, it is OK to include optional params like `temperature`, `max_tokens`, etc. when instantiating the chat hook in the above code block.

Hook Shape

Invoking the hook returns several properties for manipulating messages, submitting queries and understanding the state of the hook:

```
const {
  messages, // The list of messages in the completion
  loading, // If a new completion request is currently in progress
  submitPrompt, // Submits a prompt for completion, for more information see "Submitting a Prompt"
  abortResponse, // Allows the user to abort a chat response that is in progress, see "Aborting responses"
  resetMessages, // Reset the messages list to empty, see "Resetting Message List"
  setMessages, // Overwrites any messages in the list, see "Setting Message List"
} = useChatCompletion(...);
```

Types of Completions

Currently, this package only supports Chat Completions. Adding Text Completions support to this package is a future roadmap item (pull requests accepted).

There are two main types of completions available from OpenAI:

1. Chat Completions, which includes models like `gpt-4` and `gpt-3.5-turbo`.
2. Text Completions, which includes models like `text-davinci-003`.

There are some pretty big fundamental differences in the way these models are supported on the API side. Chat Completions consider the context of previous messages when making the next completion. Text Completions only consider the context passed into the explicit message it is currently answering.

For more information on chat vs. text completion models, see LangChain's excellent blog post on the topic.

Chat Completions

An individual message in a chat completion's `messages` list looks like this:

```typescript
interface ChatMessage {
  content: string; // The content of the completion
  role: string; // The role of the person/AI in the message
  timestamp: number; // The timestamp of when the completion finished
  meta: {
    loading: boolean; // If the completion is still being executed
    responseTime: string; // The total elapsed time the completion took
    chunks: ChatCompletionToken[]; // The chunks returned as a part of streaming the execution of the completion
  };
}
```

Each chunk corresponds to a token streamed back to the client in the completion. A `ChatCompletionToken` is the base incremental shape of content in the stream returned from the OpenAI API. It looks like this:

```typescript
interface ChatCompletionToken {
  content: string; // The partial content, if any, received in the chunk
  role: string; // The role, if any, received in the chunk
  timestamp: number; // The time the chunk was received
}
```

Submitting a Prompt

Call the `submitPrompt` function to initiate a chat completion request whose response will be streamed back to the client from the OpenAI Chat Completions API. A query takes a list of new messages to append to the existing `messages` list and submits fully appended `messages` list to OpenAI's Chat Completions API.

A sample message list might look like:

```typescript
const newMessages = [
  {
    role: 'system',
    content: 'You are a short story bot, you write short stories for kids',
  },
  {
    role: 'user',
    content: 'Write a story about a lonely bunny',
  },
];
```

When the prompt is submitted, a blank message is appended to the end of the `messages` list with its `meta.loading` state set to `true` (the `loading` flag of the hook will also be set to `true`). This message will be where the content that is streamed back to the client is collected in real-time.

New chunks of the message will appear in the `meta.chunks` list and your React component will be updated every time a new chunk appears automatically.

> 💡 **Chunks correspond directly to tokens.**
>
> By counting the number of chunks, you can count the number of tokens that a response used.

Aborting responses

If a prompt has been submitted and the response is still being streamed back to the client, it is possible to invoke the `abortResponse` function to stop the response stream from continuing. This function will only work if the request is in progress.

Setting Message List

The hook exposes a `setMessages` function which will overwrite any existing messages

This function will not set the messages list is a chat complete request is in progress.

Resetting Message List

If the `resetMessages` function is called the messages list will be set back to empty, as long as the function is invoked when a chat completion request isn't in progress.

Running the Example

1. Clone this package locally and navigate to it:

```
git clone https://github.com/jonrhall/openai-streaming-hooks.git
cd openai-streaming-hooks
```

2. Export your OpenAI API Key as environment variable `VITE_OPENAI_API_KEY` :

```
export VITE_OPENAI_API_KEY=your-key-here
```

3. Run the example dev server:

```
npm run example
```

4. Navigate to `https://localhost:5179` to see the live example.

Contributors

 Jon Hall  Michael Flohr

Future Contributions

Releases 1

v2.0.0 Latest
on May 5, 2023

Packages

No packages published

Contributors 2

 **jonrhall** Jon Hall

MatchuPitchu Michael Flohr

Languages

- **TypeScript** 92.8%
- **CSS** 4.5%
- **JavaScript** 1.5%
- **HTML** 1.2%