
LlamaIndex.TS

downloads 70k/month downloads 70k/month downloads 70k/month chat 1696 online

LlamaIndex is a data framework for your LLM application.

Use your own data with large language models (LLMs, OpenAI ChatGPT and others) in Typescript and Javascript.

Documentation: <https://ts.llamaindex.ai/>

Try examples online:



What is LlamaIndex.TS?

LlamaIndex.TS aims to be a lightweight, easy to use set of libraries to help you integrate large language models into your applications with your own data.

Getting started with an example:

LlamaIndex.TS requires Node v18 or higher. You can download it from <https://nodejs.org> or use <https://nvm.sh> (our preferred option).

In a new folder:

```
1 export OPENAI_API_KEY="sk-....." # Replace with your key from https://
  platform.openai.com/account/api-keys
2 pnpm init
3 pnpm install typescript
4 pnpm exec tsc --init # if needed
5 pnpm install llamaindex
6 pnpm install @types/node
```

Create the file example.ts

```
1 // example.ts
2 import fs from "fs/promises";
3 import { Document, VectorStoreIndex } from "llamaindex";
4
5 async function main() {
6   // Load essay from abramov.txt in Node
7   const essay = await fs.readFile(
8     "node_modules/llamaindex/examples/abramov.txt",
9     "utf-8",
```

```
10  );
11
12  // Create Document object with essay
13  const document = new Document({ text: essay });
14
15  // Split text and create embeddings. Store them in a VectorStoreIndex
16  const index = await VectorStoreIndex.fromDocuments([document]);
17
18  // Query the index
19  const queryEngine = index.asQueryEngine();
20  const response = await queryEngine.query({
21    query: "What did the author do in college?",
22  });
23
24  // Output response
25  console.log(response.toString());
26 }
27
28 main();
```

Then you can run it using

```
1  npm dlx ts-node example.ts
```

Playground

Check out our NextJS playground at <https://llama-playground.vercel.app/>. The source is available at <https://github.com/run-llama/ts-playground>

Core concepts for getting started:

- Document: A document represents a text file, PDF file or other contiguous piece of data.
- Node: The basic data building block. Most commonly, these are parts of the document split into manageable pieces that are small enough to be fed into an embedding model and LLM.
- Embedding: Embeddings are sets of floating point numbers which represent the data in a Node. By comparing the similarity of embeddings, we can derive an understanding of the similarity of two pieces of data. One use case is to compare the embedding of a question with the embeddings of our Nodes to see which Nodes may contain the data needed to answer that question. Because the default service context is OpenAI, the default embedding is [OpenAIEmbedding](#). If using different models, say through Ollama, use this Embedding (see all here).
- Indices: Indices store the Nodes and the embeddings of those nodes. QueryEngines retrieve Nodes from these Indices using embedding similarity.

-
- **QueryEngine:** Query engines are what generate the query you put in and give you back the result. Query engines generally combine a pre-built prompt with selected Nodes from your Index to give the LLM the context it needs to answer your query. To build a query engine from your Index (recommended), use the `asQueryEngine` method on your Index. See all query engines [here](#).
 - **ChatEngine:** A ChatEngine helps you build a chatbot that will interact with your Indices. See all chat engines [here](#).
 - **SimplePrompt:** A simple standardized function call definition that takes in inputs and formats them in a template literal. SimplePrompts can be specialized using currying and combined using other SimplePrompt functions.

Using NextJS

If you're using the NextJS App Router, you can choose between the Node.js and the Edge runtime.

With NextJS 13 and 14, using the Node.js runtime is the default. You can explicitly set the Edge runtime in your router handler by adding this line:

```
1 export const runtime = "edge";
```

The following sections explain further differences in using the Node.js or Edge runtime.

Using the Node.js runtime

Add the following config to your `next.config.js` to ignore specific packages in the server-side bundling:

```
1 // next.config.js
2 /** @type {import('next').NextConfig} */
3 const nextConfig = {
4   experimental: {
5     serverComponentsExternalPackages: [
6       "pdf2json",
7       "@zilliz/milvus2-sdk-node",
8       "sharp",
9       "onnxruntime-node",
10    ],
11  },
12  webpack: (config) => {
13    config.externals.push({
14      pdf2json: "commonjs pdf2json",
15      "@zilliz/milvus2-sdk-node": "commonjs @zilliz/milvus2-sdk-node",
16      sharp: "commonjs sharp",
```

```
17     "onnxruntime-node": "commonjs onnxruntime-node",
18   });
19
20   return config;
21 },
22 };
23
24 module.exports = nextConfig;
```

Using the Edge runtime

We publish a dedicated package (`@llamaindex/edge` instead of `llamaindex`) for using the Edge runtime. To use it, first install the package:

```
1 pnpm install @llamaindex/edge
```

Note: Ensure that your `package.json` doesn't include the `llamaindex` package if you're using `@llamaindex/edge`.

Then make sure to use the correct import statement in your code:

```
1 // replace 'llamaindex' with '@llamaindex/edge'
2 import {} from "@llamaindex/edge";
```

A further difference is that the `@llamaindex/edge` package doesn't export classes from the `readers` or `storage` folders. The reason is that most of these classes are not compatible with the Edge runtime.

If you need any of those classes, you have to import them instead directly. Here's an example for importing the `PineconeVectorStore` class:

```
1 import { PineconeVectorStore } from "@llamaindex/edge/storage/
  vectorStore/PineconeVectorStore";
```

As the `PDFReader` is not working with the Edge runtime, here's how to use the `SimpleDirectoryReader` with the `LlamaParseReader` to load PDFs:

```
1 import { SimpleDirectoryReader } from "@llamaindex/edge/readers/
  SimpleDirectoryReader";
2 import { LlamaParseReader } from "@llamaindex/edge/readers/
  LlamaParseReader";
3
4 export const DATA_DIR = "./data";
5
6 export async function getDocuments() {
7   const reader = new SimpleDirectoryReader();
```

```
8 // Load PDFs using LlamaParseReader
9 return await reader.loadData({
10   directoryPath: DATA_DIR,
11   fileExtToReader: {
12     pdf: new LlamaParseReader({ resultType: "markdown" }),
13   },
14 });
15 }
```

Note: Reader classes have to be added explicitly to the `fileExtToReader` map in the Edge version of the `SimpleDirectoryReader`.

You'll find a complete example of using the Edge runtime with LlamaIndexTS here: https://github.com/run-llama/create_llama_projects/tree/main/nextjs-edge-llamaparse

Supported LLMs:

- OpenAI GPT-3.5-turbo and GPT-4
- Anthropic Claude 3 (Opus, Sonnet, and Haiku) and the legacy models (Claude 2 and Instant)
- Groq LLMs
- Llama2/3 Chat LLMs (70B, 13B, and 7B parameters)
- MistralAI Chat LLMs
- Fireworks Chat LLMs

Contributing:

We are in the very early days of LlamaIndex.TS. If you're interested in hacking on it with us check out our contributing guide

Bugs? Questions?

Please join our Discord! <https://discord.com/invite/eN6D2HQ4aX>