Conversational Retrieval QA

> **INFO**
>
> Looking for the LCEL version? Click here.

The ConversationalRetrievalQA chain builds on RetrievalQAChain to provide a chat history component.

It first combines the chat history (either explicitly passed in or retrieved from the provided memory) and the question into a standalone question, then looks up relevant documents from the retriever, and finally passes those documents and the question to a question answering chain to return a response.

To create one, you will need a retriever. In the below example, we will create one from a vector store, which can be created from embeddings.

```
import { ChatOpenAI, OpenAIEmbeddings } from "@langchain/openai";
import { ConversationalRetrievalQAChain } from "langchain/chains";
import { HNSWLib } from "@langchain/community/vectorstores/hnswlib";
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";
import { BufferMemory } from "langchain/memory";
import * as fs from "fs";

export const run = async () => {
  /* Initialize the LLM to use to answer the question */
  const model = new ChatOpenAI({});
  /* Load in the file we want to do question answering over */
  const text = fs.readFileSync("state_of_the_union.txt", "utf8");
  /* Split the text into chunks */
  const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize:
1000 });
  const docs = await textSplitter.createDocuments([text]);
```

```
  /* Create the vectorstore */
  const vectorStore = await HNSWLib.fromDocuments(docs, new
OpenAIEmbeddings());
  /* Create the chain */
  const chain = ConversationalRetrievalQAChain.fromLLM(
    model,
    vectorStore.asRetriever(),
    {
      memory: new BufferMemory({
        memoryKey: "chat_history", // Must be set to "chat_history"
      }),
    }
  );
  /* Ask it a question */
  const question = "What did the president say about Justice Breyer?";
  const res = await chain.invoke({ question });
  console.log(res);
  /* Ask it a follow up question */
  const followUpRes = await chain.invoke({
    question: "Was that nice?",
  });
  console.log(followUpRes);
};
```

API Reference:

- ChatOpenAI from `@langchain/openai`
- OpenAIEmbeddings from `@langchain/openai`
- ConversationalRetrievalQAChain from `langchain/chains`
- HNSWLib from `@langchain/community/vectorstores/hnswlib`
- RecursiveCharacterTextSplitter from `langchain/text_splitter`
- BufferMemory from `langchain/memory`

In the above code snippet, the fromLLM method of the `ConversationalRetrievalQAChain` class has the following signature:

```
static fromLLM(
  llm: BaseLanguageModelInterface,
  retriever: BaseRetrieverInterface,
  options?: {
```

```
    questionGeneratorChainOptions?: {
      llm?: BaseLanguageModelInterface;
      template?: string;
    };
    qaChainOptions?: QAChainParams;
    returnSourceDocuments?: boolean;
  }
): ConversationalRetrievalQAChain
```

Here's an explanation of each of the attributes of the options object:

- `questionGeneratorChainOptions`: An object that allows you to pass a custom template and LLM to the underlying question generation chain.
  - If the template is provided, the `ConversationalRetrievalQAChain` will use this template to generate a question from the conversation context instead of using the question provided in the question parameter.
  - Passing in a separate LLM (`llm`) here allows you to use a cheaper/faster model to create the condensed question while using a more powerful model for the final response, and can reduce unnecessary latency.
- `qaChainOptions`: Options that allow you to customize the specific QA chain used in the final step. The default is the `StuffDocumentsChain`, but you can customize which chain is used by passing in a `type` parameter. **Passing specific options here is completely optional**, but can be useful if you want to customize the way the response is presented to the end user, or if you have too many documents for the default `StuffDocumentsChain`. You can see the API reference of the usable fields here. In case you want to make chat_history available to the final answering `qaChain`, which ultimately answers the user question, you HAVE to pass a custom qaTemplate with chat_history as input, as it is not present in the default Template, which only gets passed `context` documents and generated `question`.
- `returnSourceDocuments`: A boolean value that indicates whether the `ConversationalRetrievalQAChain` should return the source documents that were used to retrieve the answer. If set to true, the documents will be included in the result returned by the call() method. This can be useful if you want to allow the user to see the sources used to generate the answer. If not set, the default value will be false.
  - If you are using this option and passing in a memory instance, set `inputKey` and `outputKey` on the memory instance to the same values as the chain input and final conversational chain output. These default to `"question"` and `"text"` respectively, and specify the values that the memory should store.

Built-in Memory

Here's a customization example using a faster LLM to generate questions and a slower, more comprehensive LLM for the final answer. It uses a built-in memory object and returns the referenced source documents. Because we have `returnSourceDocuments` set and are thus returning multiple values from the chain, we must set `inputKey` and `outputKey` on the memory instance to let it know which values to store.

> **TIP**
>
> See this section for general instructions on installing integration packages.

npm | Yarn | pnpm

```
npm install @langchain/openai @langchain/community
```

```
import { ChatOpenAI, OpenAIEmbeddings } from "@langchain/openai";
import { ConversationalRetrievalQAChain } from "langchain/chains";
import { HNSWLib } from "@langchain/community/vectorstores/hnswlib";
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";
import { BufferMemory } from "langchain/memory";

import * as fs from "fs";

export const run = async () => {
  const text = fs.readFileSync("state_of_the_union.txt", "utf8");
  const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize:
1000 });
  const docs = await textSplitter.createDocuments([text]);
  const vectorStore = await HNSWLib.fromDocuments(docs, new
OpenAIEmbeddings());
  const fasterModel = new ChatOpenAI({
    model: "gpt-3.5-turbo",
  });
  const slowerModel = new ChatOpenAI({
    model: "gpt-4",
  });
```

```
  const chain = ConversationalRetrievalQAChain.fromLLM(
    slowerModel,
    vectorStore.asRetriever(),
    {
      returnSourceDocuments: true,
      memory: new BufferMemory({
        memoryKey: "chat_history",
        inputKey: "question", // The key for the input to the chain
        outputKey: "text", // The key for the final conversational output
of the chain
        returnMessages: true, // If using with a chat model (e.g. gpt-3.5
or gpt-4)
      }),
      questionGeneratorChainOptions: {
        llm: fasterModel,
      },
    }
  );
  /* Ask it a question */
  const question = "What did the president say about Justice Breyer?";
  const res = await chain.invoke({ question });
  console.log(res);

  const followUpRes = await chain.invoke({ question: "Was that nice?" });
  console.log(followUpRes);
};
```

API Reference:

- ChatOpenAI from `@langchain/openai`
- OpenAIEmbeddings from `@langchain/openai`
- ConversationalRetrievalQAChain from `langchain/chains`
- HNSWLib from `@langchain/community/vectorstores/hnswlib`
- RecursiveCharacterTextSplitter from `langchain/text_splitter`
- BufferMemory from `langchain/memory`

Streaming

You can also use the above concept of using two different LLMs to stream only the final response from the chain, and not output from the intermediate standalone question generation step. Here's an example:

```
import { ChatOpenAI, OpenAIEmbeddings } from "@langchain/openai";
import { ConversationalRetrievalQAChain } from "langchain/chains";
import { HNSWLib } from "@langchain/community/vectorstores/hnswlib";
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";
import { BufferMemory } from "langchain/memory";

import * as fs from "fs";

export const run = async () => {
  const text = fs.readFileSync("state_of_the_union.txt", "utf8");
  const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize:
1000 });
  const docs = await textSplitter.createDocuments([text]);
  const vectorStore = await HNSWLib.fromDocuments(docs, new
OpenAIEmbeddings());
  let streamedResponse = "";
  const streamingModel = new ChatOpenAI({
    streaming: true,
    callbacks: [
      {
        handleLLMNewToken(token) {
          streamedResponse += token;
        },
      },
    ],
  });
  const nonStreamingModel = new ChatOpenAI({});
  const chain = ConversationalRetrievalQAChain.fromLLM(
    streamingModel,
    vectorStore.asRetriever(),
    {
      returnSourceDocuments: true,
      memory: new BufferMemory({
        memoryKey: "chat_history",
        inputKey: "question", // The key for the input to the chain
        outputKey: "text", // The key for the final conversational output
of the chain
        returnMessages: true, // If using with a chat model
      }),
```

```
      questionGeneratorChainOptions: {
        llm: nonStreamingModel,
      },
    }
  );
  /* Ask it a question */
  const question = "What did the president say about Justice Breyer?";
  const res = await chain.invoke({ question });
  console.log({ streamedResponse });
  /*
    {
      streamedResponse: 'President Biden thanked Justice Breyer for his
service, and honored him as an Army veteran, Constitutional scholar and
retiring Justice of the United States Supreme Court.'
    }
  */
};
```

API Reference:

- ChatOpenAI from `@langchain/openai`
- OpenAIEmbeddings from `@langchain/openai`
- ConversationalRetrievalQAChain from `langchain/chains`
- HNSWLib from `@langchain/community/vectorstores/hnswlib`
- RecursiveCharacterTextSplitter from `langchain/text_splitter`
- BufferMemory from `langchain/memory`

Externally-Managed Memory

For this chain, if you'd like to format the chat history in a custom way (or pass in chat messages directly for convenience), you can also pass the chat history in explicitly by omitting the `memory` option and supplying a `chat_history` string or array of HumanMessages and AIMessages directly into the `chain.call` method:

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

```
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";
import * as fs from "fs";

/* Initialize the LLM to use to answer the question */
const model = new OpenAI({});
/* Load in the file we want to do question answering over */
const text = fs.readFileSync("state_of_the_union.txt", "utf8");
/* Split the text into chunks */
const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize: 1000
});
const docs = await textSplitter.createDocuments([text]);
/* Create the vectorstore */
const vectorStore = await HNSWLib.fromDocuments(docs, new
OpenAIEmbeddings());
/* Create the chain */
const chain = ConversationalRetrievalQAChain.fromLLM(
  model,
  vectorStore.asRetriever()
);
/* Ask it a question */
const question = "What did the president say about Justice Breyer?";
/* Can be a string or an array of chat messages */
const res = await chain.invoke({ question, chat_history: "" });
console.log(res);
/* Ask it a follow up question */
const chatHistory = `${question}\n${res.text}`;
const followUpRes = await chain.invoke({
  question: "Was that nice?",
  chat_history: chatHistory,
});
console.log(followUpRes);
```

API Reference:

- OpenAI from `@langchain/openai`
- OpenAIEmbeddings from `@langchain/openai`
- ConversationalRetrievalQAChain from `langchain/chains`
- HNSWLib from `@langchain/community/vectorstores/hnswlib`
- RecursiveCharacterTextSplitter from `langchain/text_splitter`

Prompt Customization

If you want to further change the chain's behavior, you can change the prompts for both the underlying question generation chain and the QA chain.

One case where you might want to do this is to improve the chain's ability to answer meta questions about the chat history. By default, the only input to the QA chain is the standalone question generated from the question generation chain. This poses a challenge when asking meta questions about information in previous interactions from the chat history.

For example, if you introduce a friend Bob and mention his age as 28, the chain is unable to provide his age upon asking a question like "How old is Bob?". This limitation occurs because the bot searches for Bob in the vector store, rather than considering the message history.

You can pass an alternative prompt for the question generation chain that also returns parts of the chat history relevant to the answer, allowing the QA chain to answer meta questions with the additional context:

```
import { ChatOpenAI, OpenAIEmbeddings } from "@langchain/openai";
import { ConversationalRetrievalQAChain } from "langchain/chains";
import { HNSWLib } from "@langchain/community/vectorstores/hnswlib";
import { BufferMemory } from "langchain/memory";

const CUSTOM_QUESTION_GENERATOR_CHAIN_PROMPT = `Given the following
conversation and a follow up question, return the conversation history
excerpt that includes any relevant context to the question if it exists
and rephrase the follow up question to be a standalone question.
Chat History:
{chat_history}
Follow Up Input: {question}
Your answer should follow the following format:
\`\`\`
Use the following pieces of context to answer the users question.
If you don't know the answer, just say that you don't know, don't try to
make up an answer.
----------------
<Relevant chat history excerpt as context here>
Standalone question: <Rephrased question here>
\`\`\`
Your answer:`;
```

```
const model = new ChatOpenAI({
  model: "gpt-3.5-turbo",
  temperature: 0,
});

const vectorStore = await HNSWLib.fromTexts(
  [
    "Mitochondria are the powerhouse of the cell",
    "Foo is red",
    "Bar is red",
    "Buildings are made out of brick",
    "Mitochondria are made of lipids",
  ],
  [{ id: 2 }, { id: 1 }, { id: 3 }, { id: 4 }, { id: 5 }],
  new OpenAIEmbeddings()
);

const chain = ConversationalRetrievalQAChain.fromLLM(
  model,
  vectorStore.asRetriever(),
  {
    memory: new BufferMemory({
      memoryKey: "chat_history",
      returnMessages: true,
    }),
    questionGeneratorChainOptions: {
      template: CUSTOM_QUESTION_GENERATOR_CHAIN_PROMPT,
    },
  }
);

const res = await chain.invoke({
  question:
    "I have a friend called Bob. He's 28 years old. He'd like to know what
the powerhouse of the cell is?",
});

console.log(res);
/*
  {
    text: "The powerhouse of the cell is the mitochondria."
  }
*/
```

```
const res2 = await chain.invoke({
  question: "How old is Bob?",
});

console.log(res2); // Bob is 28 years old.

/*
  {
    text: "Bob is 28 years old."
  }
*/
```

API Reference:

- ChatOpenAI from `@langchain/openai`
- OpenAIEmbeddings from `@langchain/openai`
- ConversationalRetrievalQAChain from `langchain/chains`
- HNSWLib from `@langchain/community/vectorstores/hnswlib`
- BufferMemory from `langchain/memory`

Keep in mind that adding more context to the prompt in this way may distract the LLM from other relevant retrieved information.

Help us out by providing feedback on this documentation page: