Memory | Langchain

Skip to main content LangChainDocsUse casesIntegrationsAPIMoreCommunityTutorialsContributingAlso by LangChainChat our docsLangSmithLangChain HubLangServePython DocsSearchCTRLKGet startedIntroductionInstallationQuickstartLangChain Expression LanguageInterfaceHow toCookbookWhy use LCEL?LangChain Expression Language (LCEL)ModulesModel I/ORetrievalChainsMemoryHow-toAgentsCallbacksModulesSecurityGuidesEcosystemModulesMemoryOn this pageMemoryinfoHead to Integrations for documentation on built-in integrations with memory providers. Docs under construction By default, Chains and Agents are stateless, meaning that they treat each incoming query independently (like the underlying LLMs and chat models themselves).

In some applications, like chatbots, it is essential to remember previous interactions, both in the short and long-term.

The Memory class does exactly that.LangChain provides memory components in two forms.

First, LangChain provides helper utilities for managing and manipulating previous chat messages.

These are designed to be modular and useful regardless of how they are used.

Secondly, LangChain provides easy ways to incorporate these utilities into chains.Get startedMemory involves keeping a concept of state around throughout a user's interactions with a language model. A user's interactions with a language model are captured in the concept of ChatMessages, so this boils down to ingesting, capturing, transforming and extracting knowledge from a sequence of chat messages. There are many different ways to do this, each of which exists as its own memory type.In general, for each type of memory there are two ways to understanding using memory. These are the standalone functions which extract information from a sequence of messages, and then there is the way you can use this type of memory in a chain.Memory can return multiple pieces of information (for example, the most recent N messages and a summary of all previous messages). The returned information can either be a string or a list of messages.We will walk through the simplest form of memory: "buffer" memory, which just involves keeping a buffer of all prior messages. We will show how to use the modular utility functions here, then show how it can be used in a chain (both returning a string as well as a list of messages).ChatMessageHistoryOne of the core utility classes underpinning most (if not all) memory modules is the ChatMessageHistory class. This is a super lightweight wrapper which exposes convenience methods for saving Human messages, AI messages, and then fetching them all.Subclassing this class allows you to use different storage solutions, such as Redis, to keep persistent chat message histories.import { ChatMessageHistory } from "langchain/memory";const history = new ChatMessageHistory();await history.addUserMessage("Hi!");await history.addAIChatMessage("What's up?");const messages = await history.getMessages();console.log(messages);/* [HumanMessage { content: 'Hi!', }, AIMessage { content: "What's up?", }]*/You can also load messages into memory instances by creating and passing in a ChatHistory object.

This lets you easily pick up state from past conversations. In addition to the above technique, you can do:import { BufferMemory, ChatMessageHistory } from "langchain/memory";import {

HumanChatMessage, AIChatMessage } from "langchain/schema";const pastMessages = [new HumanMessage("My name's Jonas"), new AIMessage("Nice to meet you, Jonas!"),];const memory = new BufferMemory({ chatHistory: new ChatMessageHistory(pastMessages),});noteDo not share the same history or memory instance between two different chains, a memory instance represents the history of a single conversationnoteIf you deploy your LangChain app on a serverless environment do not store memory instances in a variable, as your hosting provider may have reset it by the next time the function is called.BufferMemoryWe now show how to use this simple concept in a chain. We first showcase BufferMemory, a wrapper around ChatMessageHistory that extracts the messages into an input variable.import { OpenAI } from "langchain/llms/openai";import { BufferMemory } from "langchain/memory";import { ConversationChain } from "langchain/chains";const model = new OpenAI({});const memory = new BufferMemory();// This chain is preconfigured with a default promptconst chain = new ConversationChain({ llm: model, memory: memory });const res1 = await chain.call({ input: "Hi! I'm Jim." });console.log({ res1 });{response: " Hi Jim! It's nice to meet you. My name is AI. What would you like to talk about?"}const res2 = await chain.call({ input: "What's my name?" });console.log({ res2 });{response: ' You said your name is Jim. Is there anything else you would like to talk about?'}There are plenty of different types of memory, check out our examples to see more!Creating your own memory classThe BaseMemory interface has two methods:export type InputValues = Record<string, any>;export type OutputValues = Record<string, any>;interface BaseMemory { loadMemoryVariables(values: InputValues): Promise<MemoryVariables>; saveContext(inputValues: InputValues, outputValues: OutputValues): Promise<void>;}To implement your own memory class you have two options:Subclassing BaseChatMemoryThis is the easiest way to implement your own memory class. You can subclass BaseChatMemory, which takes care of saveContext by saving inputs and outputs as Chat Messages, and implement only the loadMemoryVariables method. This method is responsible for returning the memory variables that are relevant for the current input values.abstract class BaseChatMemory extends

BaseMemory { chatHistory: ChatMessageHistory; abstract loadMemoryVariables(values: InputValues): Promise<MemoryVariables>;}Subclassing BaseMemoryIf you want to implement a more custom memory class, you can subclass BaseMemory and implement both loadMemoryVariables and saveContext methods. The saveContext method is responsible for storing the input and output values in memory. The loadMemoryVariables method is responsible for returning the memory variables that are relevant for the current input values.abstract class BaseMemory { abstract loadMemoryVariables(values: InputValues): Promise<MemoryVariables>; abstract saveContext(inputValues: InputValues, outputValues: OutputValues): Promise<void>;}PreviousDynamically selecting from multiple retrieversNextConversation buffer memoryGet startedChatMessageHistoryBufferMemoryCreating your own memory classSubclassing BaseChatMemorySubclassing BaseMemoryCommunityDiscordTwitterGitHubPythonJS/TSMoreHomepageBlog