Chains | 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/ORetrievalChainsHow toFoundationalDocumentsPopularAdditionalMemoryAgentsCallbacksModulesSecurityGuidesEcosystemModulesChainsOn this pageChainsUsing an LLM in isolation is fine for simple applications, but more complex applications require chaining LLMs - either with each other or with other components.LangChain provides the Chain interface for such "chained" applications. We define a Chain very generically as a sequence of calls to components, which can include other chains. The base interface is simple:import { CallbackManagerForChainRun } from "langchain/callbacks";import { BaseMemory } from "langchain/memory";import { ChainValues } from "langchain/schema";abstract class BaseChain { memory?: BaseMemory; /** * Run the

core logic of this chain and return the output */ abstract _call(values: ChainValues, runManager?: CallbackManagerForChainRun): Promise<ChainValues>; /** * Return the string type key uniquely identifying this class of chain. */ abstract _chainType(): string; /** * Return the list of input keys this chain expects to receive when called. */ abstract get inputKeys(): string[]; /** * Return the list of output keys this chain will produce when called. */ abstract get outputKeys(): string[];}API Reference:CallbackManagerForChainRun from langchain/callbacksBaseMemory from langchain/memoryChainValues from langchain/schemaThis idea of composing components together in a chain is simple but powerful. It drastically simplifies and makes more modular the implementation of complex applications, which in turn makes it much easier to debug, maintain, and improve your applications.For more specifics check out:How-to for walkthroughs of different chain featuresFoundational to get acquainted with core building block chainsDocument to learn how to incorporate documents into chainsPopular chains for the most common use casesAdditional to see some of the more advanced chains and integrations that you can use out of the boxWhy do we need chains?Chains allow us to combine multiple components together to create a single, coherent application. For example, we can create a chain that takes user input, formats it with a PromptTemplate, and then passes the formatted response to an LLM. We can build more complex chains by combining multiple chains together, or by combining chains with other components.Get startedUsing LLMChainThe LLMChain is most basic building block chain. It takes in a prompt template, formats it with the user input and returns the response from an LLM.To use the LLMChain, first create a prompt template.import { OpenAI } from "langchain/llms/openai";import { PromptTemplate } from "langchain/prompts";import { LLMChain } from "langchain/chains";// We can construct an LLMChain from a PromptTemplate and an LLM.const model = new OpenAI({ temperature: 0 });const prompt = PromptTemplate.fromTemplate("What is a good name for a company that makes {product}?");We can now create a very simple chain that will take user input, format the prompt with it, and then send it to the LLM.const chain = new LLMChain({ llm: model, prompt

});// Since this LLMChain is a single-input, single-output chain, we can also `run` it.// This convenience method takes in a string and returns the value// of the output key field in the chain response. For LLMChains, this defaults to "text".const res = await chain.run("colorful socks");console.log({ res });// { res: "\n\nSocktastic!" }If there are multiple variables, you can input them all at once using a dictionary.

This will return the complete chain response.const prompt = PromptTemplate.fromTemplate("What is a good name for {company} that makes {product}?");const chain = new LLMChain({ llm: model, prompt });const res = await chain.call({ company: "a startup", product: "colorful socks",});console.log({ res });// { res: { text: '\n\Socktopia Colourful Creations.' } }You can use a chat model in an LLMChain as well:import { ChatPromptTemplate } from "langchain/prompts";import { LLMChain } from "langchain/chains";import { ChatOpenAI } from "langchain/chat_models/openai";// We can also construct an LLMChain from a ChatPromptTemplate and a chat model.const chat = new ChatOpenAI({ temperature: 0 });const chatPrompt = ChatPromptTemplate.fromMessages([["system", "You are a helpful assistant that translates {input_language} to {output_language}.",], ["human", "{text}"],]);const chainB = new LLMChain({ prompt: chatPrompt, llm: chat,});const resB = await chainB.call({ input_language: "English", output_language: "French", text: "I love programming.",});console.log({ resB });// { resB: { text: "J'adore la programmation." } }API Reference:ChatPromptTemplate from langchain/promptsLLMChain from langchain/chainsChatOpenAI from langchain/chat_models/openaiPreviousNeo4jNextHow toWhy do we need chains?Get startedUsing LLMChainCommunityDiscordTwitterGitHubPythonJS/TSMoreHomepageBlog