Callbacks | 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/ORetrievalChainsMemoryAgentsCallbacksHow-toModulesSecurityGuidesEcosystemModulesCallbacksOn this page

Callbacks

LangChain provides a callbacks system that allows you to hook into the various stages of your LLM application. This is useful for logging, monitoring, streaming, and other tasks.

You can subscribe to these events by using the callbacks argument available throughout the API. This method accepts a list of handler objects, which are expected to implement one or more of the methods described in the API docs.

How to use callbacks

The callbacks argument is available on most objects throughout the API (Chains, Language Models, Tools, Agents, etc.) in two different places.

Constructor callbacks

Defined in the constructor, eg. new LLMChain({ callbacks: [handler] }), which will be used for all calls made on that object, and will be scoped to that object only, eg. if you pass a handler to the LLMChain constructor, it will not be used by the Model attached to that chain.import { ConsoleCallbackHandler } from "langchain/callbacks";import { OpenAI } from "langchain/llms/openai";const llm = new OpenAI({ temperature: 0, // These tags will be attached to all calls made with this LLM. tags: ["example", "callbacks", "constructor"], // This handler will be used for all calls made with this LLM. callbacks: [new ConsoleCallbackHandler()],});API Reference:ConsoleCallbackHandler from langchain/callbacksOpenAI from langchain/llms/openaiRequest callbacksDefined in the call()/run()/apply() methods used for issuing a request, eg. chain.call({ input: '...' }, [handler]), which will be used for that specific request only, and all sub-requests that it contains (eg. a call to an LLMChain triggers a call to a Model, which uses the same handler passed in the call() method).import { ConsoleCallbackHandler } from "langchain/callbacks";import { OpenAI } from "langchain/llms/openai";const llm = new OpenAI({ temperature: 0,});const response = await llm.call("1 + 1 =", { // These tags will be attached only to this call to the LLM. tags: ["example", "callbacks", "request"], // This handler will be used only for this call. callbacks: [new ConsoleCallbackHandler()],});API Reference:ConsoleCallbackHandler from langchain/callbacksOpenAI from langchain/llms/openaiVerbose modeThe verbose argument is available on most objects throughout the API (Chains, Models, Tools, Agents, etc.) as a constructor argument, eg. new LLMChain({ verbose: true }), and it is equivalent to passing a ConsoleCallbackHandler to the callbacks argument of that object and all child objects. This is useful for debugging, as it will log all events to the console. You can also enable verbose mode for the entire application by setting the environment variable LANGCHAIN_VERBOSE=true.import { PromptTemplate } from "langchain/prompts";import { LLMChain } from "langchain/chains";import { OpenAI } from "langchain/llms/openai";const chain = new LLMChain({ llm: new OpenAI({ temperature: 0 }), prompt: PromptTemplate.fromTemplate("Hello, world!"), // This will enable logging of all Chain *and* LLM events to the console. verbose: true,});API Reference:PromptTemplate from langchain/promptsLLMChain from langchain/chainsOpenAI from langchain/llms/openaiWhen do you want to use each of these?Constructor callbacks are most useful for use cases such as logging, monitoring, etc., which are not specific to a single request, but rather to the entire chain. For example, if you want to log all the requests made to an LLMChain, you would pass a handler to the constructor.Request callbacks are most useful for use cases such as streaming, where you want to stream the output of a single request to a specific websocket connection, or other similar use cases. For example, if you want to stream the output of a single request to a websocket, you would pass a handler to the call() methodUsage examplesBuilt-in handlersLangChain provides a few built-in handlers that you can use to get started. These are available in the langchain/callbacks module. The most basic handler is the ConsoleCallbackHandler, which simply logs all events to the console. In the future we will add more default handlers to the library. Note that when the verbose flag on the object is set to true, the ConsoleCallbackHandler will be invoked even without being explicitly passed in.import { ConsoleCallbackHandler } from "langchain/callbacks";import { LLMChain } from "langchain/chains";import { OpenAI } from "langchain/llms/openai";import { PromptTemplate } from "langchain/prompts";export const run = async () => { const handler = new ConsoleCallbackHandler(); const llm = new OpenAI({ temperature: 0, callbacks: [handler] }); const prompt = PromptTemplate.fromTemplate("1 + {number} ="); const chain = new LLMChain({ prompt, llm, callbacks: [handler] }); const output = await chain.call({ number: 2 }); /* Entering new llm_chain chain... Finished chain. */ console.log(output); /* { text: ' 3\n\n3 - 1 = 2' }   */ // The non-enumerable key `__run` contains the runId. console.log(output.__run); /* { runId: '90e1f42c-7cb4-484c-bf7a-70b73ef8e64b' }   */};API Reference:ConsoleCallbackHandler from langchain/callbacksLLMChain from langchain/chainsOpenAI from langchain/llms/openaiPromptTemplate from langchain/promptsOne-off handlersYou can create a one-off handler inline by passing a plain object to the callbacks argument. This object should implement the CallbackHandlerMethods interface. This is useful if eg. you need to create a handler that you will use only for a single request, eg to stream the output of an LLM/Agent/etc to a websocket.import { OpenAI } from "langchain/llms/openai";// To enable streaming, we pass in `streaming: true` to the LLM constructor.// Additionally, we pass in a handler for the `handleLLMNewToken` event.const model = new OpenAI({   maxTokens: 25,   streaming: true,});const response = await model.call("Tell me a joke.", {   callbacks: [{   handleLLMNewToken(token: string) {   console.log({ token });   },   },   ],});console.log(response);/*{ token: '\n' }{ token: '\n' }{ token: 'Q' }{ token: ':' }{ token: ' Why' }{ token: ' did' }{ token: ' the' }{ token: ' chicken' }{ token: ' cross' }{ token: ' the' }{ token: ' playground' }{ token: '?' }{ token: '\n' }{ token: 'A' }{ token: ':' }{ token: ' To' }{ token: ' get' }{ token: ' to' }{ token: ' the' }{ token: ' other' }{ token: ' slide' }{ token: '.' }Q: Why did the chicken cross the playground?A: To get to the other slide.*/API Reference:OpenAI from langchain/llms/openaiMultiple handlersWe offer a method on the CallbackManager class that allows you to create a one-off handler. This is useful if eg. you need to create a handler that you will use only for a single request, eg to stream the output of an LLM/Agent/etc to a websocket.This is a more complete example that passes a CallbackManager to a ChatModel, and LLMChain, a Tool, and an Agent.import { LLMChain } from "langchain/chains";import { AgentExecutor, ZeroShotAgent } from "langchain/agents";import { BaseCallbackHandler } from "langchain/callbacks";import { ChatOpenAI } from "langchain/chat_models/openai";import { Calculator } from "langchain/tools/calculator";import { AgentAction } from "langchain/schema";import { Serialized } from "langchain/load/serializable";export const run = async () => {   // You can implement your own callback handler by extending BaseCallbackHandler   class CustomHandler extends BaseCallbackHandler {   name = "custom_handler";   handleLLMNewToken(token: string) {   console.log("token", { token });   }   handleLLMStart(llm: Serialized, _prompts: string[]) {   console.log("handleLLMStart", { llm });   }   handleChainStart(chain: Serialized) {   console.log("handleChainStart", { chain });   }   handleAgentAction(action: AgentAction) {   console.log("handleAgentAction", action);   }   handleToolStart(tool: Serialized) {   console.log("handleToolStart", { tool });   }   }   const handler1 = new CustomHandler(); // Additionally, you can use the `fromMethods` method to create a callback handler const handler2 = BaseCallbackHandler.fromMethods({ handleLLMStart(llm, _prompts: string[]) { console.log("handleLLMStart: I'm the second handler!!", { llm }); }, handleChainStart(chain) { console.log("handleChainStart: I'm the second handler!!", { chain }); }, handleAgentAction(action) { console.log("handleAgentAction", action); }, handleToolStart(tool) { console.log("handleToolStart", { tool }); }, }); // You can restrict callbacks to a particular object by passing it upon creation const model = new ChatOpenAI({ temperature: 0, callbacks: [handler2], // this will issue handler2 callbacks related to this model streaming: true, // needed to enable streaming, which enables handleLLMNewToken }); const tools = [new Calculator()]; const agentPrompt = ZeroShotAgent.createPrompt(tools); const llmChain = new LLMChain({ llm: model, prompt: agentPrompt, callbacks: [handler2], // this will issue handler2 callbacks related to this chain }); const agent = new ZeroShotAgent({ llmChain, allowedTools: ["search"], }); const agentExecutor = AgentExecutor.fromAgentAndTools({ agent, tools, }); /* * When we pass the callback handler to the agent executor, it will be used for all * callbacks related to the agent and all the objects involved in the agent's * execution, in this case, the Tool, LLMChain, and LLM. * * The `handler2` callback handler will only be used for callbacks related to the * LLMChain and LLM, since we passed it to the LLMChain and LLM objects upon creation. */ const result = await agentExecutor.invoke({ input: "What is 2 to the power of 8", }, { callbacks: [handler1] }); // this is needed to see handleAgentAction /* handleChainStart { chain: { name: 'agent_executor' } } handleChainStart { chain: { name: 'llm_chain' } } handleChainStart: I'm the second handler!! { chain: { name: 'llm_chain' } } handleLLMStart { llm: { name: 'openai' } } handleLLMStart: I'm the second handler!! { llm: { name: 'openai' } } token { token: '' } token { token: 'I' } token { token: ' can' } token { token: ' use' } token { token: ' the' } token { token: ' calculator' } token { token: ' tool' } token { token: ' to' } token { token: ' solve' } token { token: ' this' } token { token: '.\n' } token {

token: 'Action' } token { token: ':' } token { token: ' calculator' } token { token: '\n' } token { token: 'Action' } token { token: ' Input' } token { token: ':' } token { token: ' ' } token { token: '2' } token { token: '^' } token { token: '8' } token { token: '' } handleAgentAction { tool: 'calculator', toolInput: '2^8', log: 'I can use the calculator tool to solve this.\n' + 'Action: calculator\n' + 'Action Input: 2^8' } handleToolStart { tool: { name: 'calculator' } } handleChainStart { chain: { name: 'llm_chain' } } handleChainStart: I'm the second handler!! { chain: { name: 'llm_chain' } } handleLLMStart { llm: { name: 'openai' } } handleLLMStart: I'm the second handler!! { llm: { name: 'openai' } } token { token: '' } token { token: 'That' } token { token: ' was' } token { token: ' easy' } token { token: '!\n' } token { token: 'Final' } token { token: ' Answer' } token { token: ':' } token { token: ' ' } token { token: '256' } token { token: '' } */ console.log(result); /* { output: '256', __run: { runId: '26d481a6-4410-4f39-b74d-f9a4f572379a' } } */};API Reference:LLMChain from langchain/chainsAgentExecutor from langchain/agentsZeroShotAgent from langchain/agentsBaseCallbackHandler from langchain/callbacksChatOpenAI from langchain/chat_models/openaiCalculator from langchain/tools/calculatorAgentAction from langchain/schemaSerialized from langchain/load/serializablePreviousToolkitsNextBackgrounding callbacksHow to use callbacksConstructor callbacksRequest callbacksVerbose modeWhen do you want to use each of these?Usage examplesBuilt-in handlersOne-off handlersMultiple handlersCommunityDiscordTwitterGitHubPythonJS/TSMoreHomepageBlog