

Quickstart | 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/ORetrievalChainsMemoryAgentsCallbacksModulesSecurityGuidesEcosystemGet startedQuickstartOn this pageQuickstartInstallationTo install LangChain run:`npm install -S langchain``yarn add langchain``npm add langchain`For more details, see our Installation guide.Environment setupUsing LangChain will usually require integrations with one or more model providers, data stores, APIs, etc. For this example, we'll use OpenAI's model APIs.Accessing their API requires an API key, which you can get by creating an account and heading here. Once we have a key we'll want to set it as an environment variable by running:`export OPENAI_API_KEY="..."`If you'd prefer not to set an environment variable you can pass the key in directly via the `openAIApiKey` parameter when initializing the OpenAI LLM class:`import { OpenAI }`

```
from "langchain/llms/openai";const llm = new OpenAI({ openAIApiKey:
"YOUR_KEY_HERE",});
```

Building an application

Now we can start building our language model application. LangChain provides many modules that can be used to build language model applications. Modules can be used as stand-alones in simple applications and they can be combined for more complex use cases.

The most common and most important chain that LangChain helps create contains three things:

- LLM:** The language model is the core reasoning engine here. In order to work with LangChain, you need to understand the different types of language models and how to work with them.
- Prompt Templates:** This provides instructions to the language model. This controls what the language model outputs, so understanding how to construct prompts and different prompting strategies is crucial.
- Output Parsers:** These translate the raw response from the LLM to a more workable format, making it easy to use the output downstream.

In this getting started guide we will cover those three components by themselves, and then go over how to combine all of them. Understanding these concepts will set you up well for being able to use and customize LangChain applications. Most LangChain applications allow you to configure the LLM and/or the prompt used, so knowing how to take advantage of this will be a big enabler.

LLMs

There are two types of language models, which in LangChain are called:

- LLMs:** this is a language model which takes a string as input and returns a string
- ChatModels:** this is a language model which takes a list of messages as input and returns a message

The input/output for LLMs is simple and easy to understand - a string. But what about ChatModels? The input there is a list of ChatMessages, and the output is a single ChatMessage. A ChatMessage has two required components:

- content:** This is the content of the message.
- role:** This is the role of the entity from which the ChatMessage is coming from.

LangChain provides several objects to easily distinguish between different roles:

- HumanMessage:** A ChatMessage coming from a human/user.
- AIMessage:** A ChatMessage coming from an AI/assistant.
- SystemMessage:** A ChatMessage coming from the system.
- FunctionMessage:** A ChatMessage coming from a function call.

If none of those roles sound right, there is also a ChatMessage class where you can specify

the role manually. For more information on how to use these different messages most effectively, see our prompting guide. LangChain provides a standard interface for both, but it's useful to understand this difference in order to construct prompts for a given language model. The standard interface that LangChain provides has two methods:

- `predict`: Takes in a string, returns a string
- `predictMessages`: Takes in a list of messages, returns a message.

Let's see how to work with these different types of models and these different types of inputs. First, let's import an LLM and a ChatModel and call `predict`.

```
import { OpenAI } from "langchain/llms/openai";
import { ChatOpenAI } from "langchain/chat_models/openai";
const llm = new OpenAI({ temperature: 0.9, });
const chatModel = new ChatOpenAI();
const text = "What would be a good company name for a company that makes colorful socks?";
const llmResult = await llm.predict(text); /* "Feetful of Fun" */
const chatModelResult = await chatModel.predict(text); /* "Socks O'Color" */
```

The OpenAI and ChatOpenAI objects are basically just configuration objects. You can initialize them with parameters like temperature and others, and pass them around. Next, let's use the `predictMessages` method to run over a list of messages.

```
import { HumanMessage } from "langchain/schema";
const text = "What would be a good company name for a company that makes colorful socks?";
const messages = [new HumanMessage({ content: text })];
const llmResult = await llm.predictMessages(messages); /* AIMessage { content: "Feetful of Fun" } */
const chatModelResult = await chatModel.predictMessages(messages); /* AIMessage { content: "Socks O'Color" } */
```

For both these methods, you can also pass in parameters as keyword arguments. For example, you could pass in `temperature: 0` to adjust the temperature that is used from what the object was configured with. Whatever values are passed in during run time will always override what the object was configured with.

Prompt templates

Most LLM applications do not pass user input directly into an LLM. Usually they will add the user input to a larger piece of text, called a prompt template, that provides additional context on the specific task at hand. In the previous example, the text we passed to the model contained instructions to generate a company name. For our application, it'd be great if the user only had to provide the

description of a company/product, without having to worry about giving the model instructions. PromptTemplates help with exactly this! They bundle up all the logic for going from user input into a fully formatted prompt. This can start off very simple - for example, a prompt to produce the above string would just be:

```
import { PromptTemplate } from "langchain/prompts";const prompt = PromptTemplate.fromTemplate( "What is a good name for a company that makes {product}?");const formattedPrompt = await prompt.format({ product: "colorful socks",});/* "What is a good name for a company that makes colorful socks?"*/
```

There are several advantages to using these over raw string formatting. You can "partial" out variables - e.g. you can format only some of the variables at a time. You can compose them together, easily combining different templates into a single prompt. For explanations of these functionalities, see the section on prompts for more detail. PromptTemplates can also be used to produce a list of messages. In this case, the prompt not only contains information about the content, but also each message (its role, its position in the list, etc). Here, what happens most often is a ChatPromptTemplate is a list of ChatMessageTemplates. Each ChatMessageTemplate contains instructions for how to format that ChatMessage - its role, and then also its content. Let's take a look at this below:

```
import { ChatPromptTemplate } from "langchain/prompts";const template = "You are a helpful assistant that translates {input_language} into {output_language}.";const humanTemplate = "{text}";const chatPrompt = ChatPromptTemplate.fromMessages([["system", template], ["human", humanTemplate],]);const formattedChatPrompt = await chatPrompt.formatMessages({ input_language: "English", output_language: "French", text: "I love programming.",});/* [ SystemMessage { content: 'You are a helpful assistant that translates English into French.' }, HumanMessage { content: 'I love programming.' } ]*/
```

ChatPromptTemplates can also be constructed in other ways - see the section on prompts for more detail. Output parsers OutputParsers convert the raw output of an LLM into a format that can be used downstream. There are few main type of OutputParsers, including:

- Convert text from LLM -> structured information (e.g. JSON)
- Convert a ChatMessage into just a string
- Convert the extra

information returned from a call besides the message (like OpenAI function invocation) into a string. For more information, see the section on output parsers. In this getting started guide, we will write our own output parser - one that converts a comma separated list into a list.

```
import {
  BaseOutputParser
} from "langchain/schema/output_parser";/** * Parse the output of an LLM call
to a comma-separated list. */class CommaSeparatedListOutputParser extends
BaseOutputParser<string[]> {  async parse(text: string): Promise<string[]> {    return
text.split(",").map((item) => item.trim());  }}const parser = new
CommaSeparatedListOutputParser();const result = await parser.parse("hi, bye");/* ['hi',
'bye']*/
```

PromptTemplate + LLM + OutputParser

We can now combine all these into one chain. This chain will take input variables, pass those to a prompt template to create a prompt, pass the prompt to a language model, and then pass the output through an (optional) output parser. This is a convenient way to bundle up a modular piece of logic. Let's see it in action!

```
import {
  ChatOpenAI
} from "langchain/chat_models/openai";import { ChatPromptTemplate } from
"langchain/prompts";import { BaseOutputParser } from "langchain/schema/output_parser";/** *
Parse the output of an LLM call to a comma-separated list. */class
CommaSeparatedListOutputParser extends BaseOutputParser<string[]> {  async parse(text:
string): Promise<string[]> {    return text.split(",").map((item) => item.trim());  }}const
template = `You are a helpful assistant who generates comma separated lists. A user will pass
in a category, and you should generate 5 objects in that category in a comma separated list. ONLY
return a comma separated list, and nothing more.`;const humanTemplate = "{text}";/** * Chat
prompt for generating comma-separated lists. It combines the system * template and the human
template. */const chatPrompt = ChatPromptTemplate.fromMessages([ ["system", template],
["human", humanTemplate],]);const model = new ChatOpenAI({});const parser = new
CommaSeparatedListOutputParser();const chain = chatPrompt.pipe(model).pipe(parser);const
result = await chain.invoke({  text: "colors",});/* ["red", "blue", "green", "yellow",
"orange"]*/
```

Note that we are using the .pipe() method to join these components together. This

.pipe() method is part of the LangChain Expression Language. To learn more about this syntax, read the documentation here.

Next steps

And that's it for the quickstart! We've now gone over how to create the core building block of LangChain applications. There is a lot more nuance in all these components (LLMs, prompts, output parsers) and a lot more different components to learn about as well. To continue on your journey:

Continue learning with this free interactive course by our friends at Scrimba.

Dive deeper into LLMs, prompts, and output parsers

Learn the other key components

Read up on LangChain Expression Language to learn how to chain these components together

Check out our helpful guides for detailed walkthroughs on particular topics

Explore end-to-end use cases

[Previous](#) [Installation](#) [Next](#)

[LangChain Expression Language \(LCEL\)](#) [Installation](#) [Environment setup](#) [Building an application](#) [LLMs](#) [Prompt templates](#) [Output parsers](#) [PromptTemplate](#) + [LLM](#) + [OutputParser](#) [Next steps](#)

[Community](#) [Discord](#) [Twitter](#) [GitHub](#) [Python](#) [JS/TS](#) [More](#) [Homepage](#) [Blog](#) [Copyright](#) © 2023

LangChain, Inc.