Quickstart

In this quickstart we'll show you how to:

- Get setup with LangChain, LangSmith and LangServe
- Use the most basic and common components of LangChain: prompt templates, models, and output parsers
- Use LangChain Expression Language, the protocol that LangChain is built on and which facilitates component chaining
- Build a simple application with LangChain
- Trace your application with LangSmith
- Serve your application with LangServe

That's a fair amount to cover! Let's dive in.

Setup

Jupyter Notebook

This guide (and most of the other guides in the documentation) use Jupyter notebooks and assume the reader is as well.

Jupyter notebooks are perfect for learning how to work with LLM systems because often times things can go wrong (unexpected output, API down, etc) and going through guides in an interactive environment is a great way to better understand them.

You do not NEED to go through the guide in a Jupyter Notebook, but it is recommended. See here for instructions on how to install.

Installation

To install LangChain run:

Pip Conda

```
pip install langchain
```

For more details, see our Installation guide.

LangSmith

Many of the applications you build with LangChain will contain multiple steps with multiple invocations of LLM calls. As these applications get more and more complex, it becomes crucial to be able to inspect what exactly is going on inside your chain or agent. The best way to do this is with LangSmith.

Note that LangSmith is not needed, but it is helpful. If you do want to use LangSmith, after you sign up at the link above, make sure to set your environment variables to start logging traces:

```
export LANGCHAIN_TRACING_V2="true"
export LANGCHAIN_API_KEY="..."
```

Building with LangChain

LangChain enables building application that connect external sources of data and computation to LLMs. In this quickstart, we will walk through a few different ways of doing that. We will start with a simple LLM chain, which just relies on information in the prompt template to respond. Next, we will build a retrieval chain, which fetches data from a separate database and passes that into the prompt template. We will then add in chat history, to create a conversation retrieval chain. This allows you interact in a chat manner with this LLM, so it remembers previous questions. Finally, we will build an agent - which utilizes an LLM to determine whether or not it needs to fetch data to answer questions. We will cover these at a high level, but there are lot of details to all of these! We will link to relevant docs.

LLM Chain

For this getting started guide, we will provide two options: using OpenAI (a popular model available via API) or using a local open source model.

OpenAI **Local**

First we'll need to import the LangChain x OpenAI integration package.

```
pip install langchain-openai
```

Accessing the 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="..."
```

We can then initialize the model:

```python
from langchain_openai import ChatOpenAI

llm = ChatOpenAI()
```

If you'd prefer not to set an environment variable you can pass the key in directly via the `openai_api_key` named parameter when initiating the OpenAI LLM class:

```python
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(openai_api_key="...")
```

Once you've installed and initialized the LLM of your choice, we can try using it! Let's ask it what LangSmith is - this is something that wasn't present in the training data so it shouldn't have a very good response.

```python
llm.invoke("how can langsmith help with
testing?")
```

We can also guide it's response with a prompt template. Prompt templates are used to convert raw user input to a better input to the LLM.

```
from langchain_core.prompts import
ChatPromptTemplate
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are world class technical
documentation writer."),
    ("user", "{input}")
])
```

We can now combine these into a simple LLM chain:

```
chain = prompt | llm
```

We can now invoke it and ask the same question. It still won't know the answer, but it should respond in a more proper tone for a technical writer!

```
chain.invoke({"input": "how can langsmith
help with testing?"})
```

The output of a ChatModel (and therefore, of this chain) is a message. However, it's often much more convenient to work with strings. Let's add a simple output parser to convert the chat message to a string.

```
from langchain_core.output_parsers import
StrOutputParser

output_parser = StrOutputParser()
```

We can now add this to the previous chain:

```
chain = prompt | llm | output_parser
```

We can now invoke it and ask the same question. The answer will now be a string (rather than a ChatMessage).

```
chain.invoke({"input": "how can langsmith
help with testing?"})
```

Diving Deeper

We've now successfully set up a basic LLM chain. We only touched on the basics of prompts, models, and output parsers - for a deeper dive into everything mentioned here, see this section of documentation.

Retrieval Chain

In order to properly answer the original question ("how can langsmith help with testing?"), we need to provide additional context to the LLM. We can do this via *retrieval*. Retrieval is useful when you have **too much data** to pass to the LLM directly. You can then use a retriever to fetch only the most relevant pieces and pass those in.

In this process, we will look up relevant documents from a *Retriever* and then pass them into the prompt. A Retriever can be backed by anything - a SQL table, the internet, etc - but in this instance we will populate a vector store and use that as a retriever. For more information on vectorstores, see this documentation.

First, we need to load the data that we want to index. In order to do this, we will use the WebBaseLoader. This requires installing BeautifulSoup:

```
pip install beautifulsoup4
```

After that, we can import and use WebBaseLoader.

```
from langchain_community.document_loaders impor
WebBaseLoader
loader =
WebBaseLoader("https://docs.smith.langchain.com
docs = loader.load()
```

Next, we need to index it into a vectorstore. This requires a few components, namely an embedding model and a vectorstore.

For embedding models, we once again provide examples for accessing via OpenAI or via local models.

OpenAI **Local**

Make sure you have the `langchain_openai` package installed an the appropriate environment variables set (these are the same as needed for the LLM).

```
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
```

Now, we can use this embedding model to ingest documents into a vectorstore. We will use a simple local vectorstore, FAISS, for simplicity's sake.

First we need to install the required packages for that:

```
pip install faiss-cpu
```

Then we can build our index:

```
from langchain_community.vectorstores import
FAISS
from langchain.text_splitter import
RecursiveCharacterTextSplitter


text_splitter =
RecursiveCharacterTextSplitter()
documents =
text_splitter.split_documents(docs)
vector = FAISS.from_documents(documents,
embeddings)
```

Now that we have this data indexed in a vectorstore, we will create a retrieval chain. This chain will take an incoming question, look up relevant documents, then pass those documents along with the original question into an LLM and ask it to answer the original question.

First, let's set up the chain that takes a question and the retrieved documents and generates an answer.

```
from langchain.chains.combine_documents
import create_stuff_documents_chain

prompt =
ChatPromptTemplate.from_template("""Answer
the following question based only on the
provided context:

<context>
{context}
</context>

Question: {input}""")

document_chain =
create_stuff_documents_chain(llm, prompt)
```

If we wanted to, we could run this ourselves by passing in documents directly:

```
from langchain_core.documents import Document

document_chain.invoke({
    "input": "how can langsmith help with
testing?",
    "context":
[Document(page_content="langsmith can let you
visualize test results")]
})
```

However, we want the documents to first come from the retriever we just set up. That way, for a given question we can use the retriever to dynamically select the most relevant documents and pass those in.

```
from langchain.chains import
create_retrieval_chain

retriever = vector.as_retriever()
retrieval_chain =
create_retrieval_chain(retriever,
document_chain)
```

We can now invoke this chain. This returns a dictionary - the response from the LLM is in the `answer` key

```
response = retrieval_chain.invoke({"input":
"how can langsmith help with testing?"})
print(response["answer"])

# LangSmith offers several features that can
help with testing:...
```

This answer should be much more accurate!

Diving Deeper

We've now successfully set up a basic retrieval chain. We only touched on the basics of retrieval - for a deeper dive into everything mentioned here, see this section of documentation.

Conversation Retrieval Chain

The chain we've created so far can only answer single questions. One of the main types of LLM applications that people are building are chat bots. So how do we turn this chain into one that can answer follow up questions?

We can still use the `create_retrieval_chain` function, but we need to change two things:

1. The retrieval method should now not just work on the most recent input, but rather should take the whole history into account.
2. The final LLM chain should likewise take the whole history into account

Updating Retrieval

In order to update retrieval, we will create a new chain. This chain will take in the most recent input (`input`) and the conversation history (`chat_history`) and use an LLM to generate a search query.

```
from langchain.chains import
create_history_aware_retriever
from langchain_core.prompts import
MessagesPlaceholder

# First we need a prompt that we can pass into
LLM to generate this search query

prompt = ChatPromptTemplate.from_messages([

MessagesPlaceholder(variable_name="chat_history
    ("user", "{input}"),
    ("user", "Given the above conversation,
generate a search query to look up in order to
information relevant to the conversation")
])
retriever_chain =
create_history_aware_retriever(llm, retriever,
prompt)
```

We can test this out by passing in an instance where the user is asking a follow up question.

```
from langchain_core.messages import
HumanMessage, AIMessage
```

```
chat_history = [HumanMessage(content="Can
LangSmith help test my LLM applications?"),
AIMessage(content="Yes!")]
retriever_chain.invoke({
    "chat_history": chat_history,
    "input": "Tell me how"
})
```

You should see that this returns documents about testing in LangSmith. This is because the LLM generated a new query, combining the chat history with the follow up question.

Now that we have this new retriever, we can create a new chain to continue the conversation with these retrieved documents in mind.

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Answer the user's questions bas
on the below context:\n\n{context}"),

MessagesPlaceholder(variable_name="chat_history
    ("user", "{input}"),
])
document_chain = create_stuff_documents_chain(l
prompt)

retrieval_chain =
create_retrieval_chain(retriever_chain,
document_chain)
```

We can now test this out end-to-end:

```
chat_history = [HumanMessage(content="Can
LangSmith help test my LLM applications?"),
AIMessage(content="Yes!")]
retrieval_chain.invoke({
    "chat_history": chat_history,
    "input": "Tell me how"
})
```

We can see that this gives a coherent answer - we've successfully turned our retrieval chain into a chatbot!

Agent

We've so far create examples of chains - where each step is known ahead of time. The final thing we will create is an agent - where the LLM decides what steps to take.

NOTE: for this example we will only show how to create an agent using OpenAI models, as local models are not reliable enough yet.

One of the first things to do when building an agent is to decide what tools it should have access to. For this example, we will give the agent access two tools:

1. The retriever we just created. This will let it easily answer questions about LangSmith
2. A search tool. This will let it easily answer questions that require up to date information.

First, let's set up a tool for the retriever we just created:

```
from langchain.tools.retriever import
create_retriever_tool

retriever_tool = create_retriever_tool(
    retriever,
    "langsmith_search",
    "Search for information about LangSmith.
For any questions about LangSmith, you must
use this tool!",
)
```

The search tool that we will use is Tavily. This will require an API key (they have generous free tier). After creating it on their platform, you need to set it as an environment variable:

```
export TAVILY_API_KEY=...
```

If you do not want to set up an API key, you can skip creating this tool.

```
from langchain_community.tools.tavily_search
import TavilySearchResults

search = TavilySearchResults()
```

We can now create a list of the tools we want to work with:

```
tools = [retriever_tool, search]
```

Now that we have the tools, we can create an agent to use them. We will go over this pretty quickly - for a deeper dive into what exactly is going on, check out the Agent's Getting Started documentation

Install langchain hub first

```
pip install langchainhub
```

Now we can use it to get a predefined prompt

```
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain.agents import
create_openai_functions_agent
from langchain.agents import AgentExecutor

# Get the prompt to use - you can modify
this!
prompt = hub.pull("hwchase17/openai-
functions-agent")
llm = ChatOpenAI(model="gpt-3.5-turbo",
temperature=0)
agent = create_openai_functions_agent(llm,
tools, prompt)
agent_executor = AgentExecutor(agent=agent,
tools=tools, verbose=True)
```

We can now invoke the agent and see how it responds! We can ask it questions about LangSmith:

```
agent_executor.invoke({"input": "how can
langsmith help with testing?"})
```

We can ask it about the weather:

```
agent_executor.invoke({"input": "what is the
weather in SF?"})
```

We can have conversations with it:

```
chat_history = [HumanMessage(content="Can
LangSmith help test my LLM applications?"),
AIMessage(content="Yes!")]
agent_executor.invoke({
    "chat_history": chat_history,
    "input": "Tell me how"
})
```

Diving Deeper

We've now successfully set up a basic agent. We only touched on the basics of agents - for a deeper dive into everything mentioned here, see this section of documentation.

Serving with LangServe

Now that we've built an application, we need to serve it. That's where LangServe comes in. LangServe helps developers deploy LangChain chains as a REST API. You do not need to use LangServe to use LangChain, but in this guide we'll show how you can deploy your app with LangServe.

While the first part of this guide was intended to be run in a Jupyter Notebook, we will now move out of that. We will be creating a Python file and then interacting with it from the command line.

Install with:

```
pip install "langserve[all]"
```

Server

To create a server for our application we'll make a `serve.py` file. This will contain our logic for serving our application. It consists of three things:

1. The definition of our chain that we just built above
2. Our FastAPI app
3. A definition of a route from which to serve the chain, which is done with `langserve.add_routes`

```
#!/usr/bin/env python
from typing import List

from fastapi import FastAPI
from langchain_core.prompts import ChatPromptTe
from langchain_openai import ChatOpenAI
from langchain_community.document_loaders impor
WebBaseLoader
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FA
from langchain.text_splitter import
RecursiveCharacterTextSplitter
```

```
from langchain.tools.retriever import
create_retriever_tool
from langchain_community.tools.tavily_search im
TavilySearchResults
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain.agents import create_openai_func
from langchain.agents import AgentExecutor
from langchain.pydantic_v1 import BaseModel, Fi
from langchain_core.messages import BaseMessage
from langserve import add_routes

# 1. Load Retriever
loader =
WebBaseLoader("https://docs.smith.langchain.com
docs = loader.load()
text_splitter = RecursiveCharacterTextSplitter(
documents = text_splitter.split_documents(docs)
embeddings = OpenAIEmbeddings()
vector = FAISS.from_documents(documents, embedd
retriever = vector.as_retriever()

# 2. Create Tools
retriever_tool = create_retriever_tool(
    retriever,
    "langsmith_search",
    "Search for information about LangSmith. Fo
questions about LangSmith, you must use this to
)
search = TavilySearchResults()
tools = [retriever_tool, search]
```

```
# 3. Create Agent
prompt = hub.pull("hwchase17/openai-functions-a
llm = ChatOpenAI(model="gpt-3.5-turbo", tempera
agent = create_openai_functions_agent(llm, tool
agent_executor = AgentExecutor(agent=agent, too
verbose=True)


# 4. App definition
app = FastAPI(
  title="LangChain Server",
  version="1.0",
  description="A simple API server using LangCh
Runnable interfaces",
)

# 5. Adding chain route

# We need to add these input/output schemas bec
current AgentExecutor
# is lacking in schemas.

class Input(BaseModel):
    input: str
    chat_history: List[BaseMessage] = Field(
        ...,
        extra={"widget": {"type": "chat", "inpu
"location"}},
    )
```

```
class Output(BaseModel):
    output: str

add_routes(
    app,
    agent_executor.with_types(input_type=Input,
output_type=Output),
    path="/agent",
)

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app, host="localhost", port=800
```

And that's it! If we execute this file:

```
python serve.py
```

we should see our chain being served at localhost:8000.

Playground

Every LangServe service comes with a simple built-in UI for configuring and invoking the application with streaming output and visibility into intermediate steps. Head to http://localhost:8000/agent/playground/ to try it out! Pass in the same question as before - "how can langsmith help with testing?" - and it should respond same as before.

Client

Now let's set up a client for programmatically interacting with our service. We can easily do this with the `[langserve.RemoteRunnable]` `(/docs/langserve#client)`. Using this, we can interact with the served chain as if it were running client-side.

```
from langserve import RemoteRunnable

remote_chain =
RemoteRunnable("http://localhost:8000/agent/")
remote_chain.invoke({"input": "how can
langsmith help with testing?"})
```

To learn more about the many other features of LangServe head here.

Next steps

We've touched on how to build an application with LangChain, how to trace it with LangSmith, and how to serve it with

LangServe. There are a lot more features in all three of these than we can cover here. To continue on your journey, we recommend you read the following (in order):

- All of these features are backed by LangChain Expression Language (LCEL) - a way to chain these components together. Check out that documentation to better understand how to create custom chains.
- Model IO covers more details of prompts, LLMs, and output parsers.
- Retrieval covers more details of everything related to retrieval
- Agents covers details of everything related to agents
- Explore common end-to-end use cases and template applications
- Read up on LangSmith, the platform for debugging, testing, monitoring and more
- Learn more about serving your applications with LangServe