

LangChain: Prompt Templates (Part 2)

Welcome back! In this notebook, we compare **PromptTemplate** and **ChatPromptTemplate** side-by-side.

The key idea: PromptTemplate builds **one string**, while ChatPromptTemplate builds a **structured list of messages**. That structure is why ChatPromptTemplate is the preferred approach for chat, history, and tools.

Why this matters:

- Chat models expect role-based messages under the hood
- History stays clean and typed (no fake USER/ASSISTANT labels)
- It scales to tools/agents without changing the pattern

We will show the same task both ways so you can see where string prompts become brittle.

Setup: model and environment

We will reuse a lightweight model and load `.env` for the API key.

```
# Import the dotenv loader
from dotenv import load_dotenv
# Import the chat model initializer
from langchain.chat_models import init_chat_model

# Load variables from .env into the process environment
load_dotenv()

# Create a lightweight chat model for this demo
model = init_chat_model(
    # Choose a small, fast OpenAI model
    model="gpt-4o-mini"
)
```

Approach A: PromptTemplate (history as text)

We will do the string version first. With PromptTemplate, history is just a **string blob**. You must invent labels like USER/ASSISTANT and keep the formatting consistent yourself.

```
# Import the string prompt template class
from langchain_core.prompts import PromptTemplate

# Create a plain string prompt that includes conversation history
string_prompt = PromptTemplate.from_template(
    # The history and question are injected as text
    "You are a concise tutor.\n\n"
    "Conversation so far:\n"
    "{history}\n\n"
    "Next user question:\n"
    "{question}\n\n"
```

```
"Answer in one sentence"
```

```
)
```

We build the string history and the next question.

Notice how the roles are just text labels, not real message objects.

```
# Build a manual text history (note the fake USER/ASSISTANT labels)
string_history = "USER: What is overfitting?\nASSISTANT: Overfitting is when a model learns r

# Define the next user question
question = "Give me one quick way to reduce it."
```

We fill the template to create the final prompt string.

```
# Fill the string template with history and the next question
final_string_prompt = string_prompt.invoke(
    {
        "history": string_history,
        "question": question
    }
)
```

Now we send the string prompt to the model.

```
# Send the final string prompt to the model
response = model.invoke(final_string_prompt)

# Print the assistant response
print("Assistant:", response.content)
```

Assistant: One quick way to reduce overfitting is to use regularization techniques, such as L1 or L2 regularization.

Output note: One sentence back - nice. But notice we're still in string-land.

To continue the chat, we manually append the new turn back into the history string.

```
# Manually append the new turn back into the history text
string_history += f"\nUSER: {question}\nASSISTANT: {response.content}"

# Print the updated history string
print(string_history)
```

USER: What is overfitting?
ASSISTANT: Overfitting is when a model learns noise and performs poorly on new data.
USER: Give me one quick way to reduce it.
ASSISTANT: One quick way to reduce overfitting is to use regularization techniques, such as L1 or L2 regularization.

Output note: The history grew, but it's just a text blob you have to maintain by hand.

Approach B: ChatPromptTemplate (history as messages)

Now we switch to the message-based version. Here, history stays **typed** as message objects, which is safer and scales to tools and multi-turn chat.

```
# Import chat prompt tools and message classes
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.prompts.chat import MessagesPlaceholder
from langchain.messages import HumanMessage, AIMessage
```

We define a chat prompt with a `MessagesPlaceholder` for history.

```
# Create a chat prompt with a history placeholder
chat_prompt = ChatPromptTemplate.from_messages([
    # System rule for the assistant
    ("system", "You are a concise tutor. Answer in one sentence."),
    # Insert past messages here
    MessagesPlaceholder("history"),
    # Next user question
    ("human", "{question}")
])
```

We create real `HumanMessage` and `AIMessage` objects for history.

```
# Create structured history as real messages
chat_history = [
    HumanMessage(
        content="What is overfitting?"
    ),
    AIMessage(
        content="Overfitting is when a model learns noise and performs poorly on new data."
    )
]

# Reuse the same question
question = "Give me one quick way to reduce it."
```

Finally, we format the template into a message list that is ready to send to the model.

```
# Fill the chat template with history and the next question
chat_prompt_value = chat_prompt.invoke(
    {
        "history": chat_history,
        "question": question
    }
)

# Convert the prompt value into messages ready for the model
messages_to_send = chat_prompt_value.to_messages()
```

Now we send the **message list** to the model.

```
response = model.invoke(messages_to_send)

print("Assistant:", response.content)
```

Assistant: One quick way to reduce overfitting is to use regularization techniques, such as L1 or L2 regularization.

Output note: One sentence again, but now it comes from structured messages.

Here we append the new user + assistant messages back into history.

```
chat_history.append(HumanMessage(content=question))
chat_history.append(AIMessage(content=response.content))

display(chat_history)
```

```
[HumanMessage(content='What is overfitting?', additional_kwargs={}, response_metadata={}),
 AIMessage(content='Overfitting is when a model learns noise and performs poorly on new data.', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[]),
 HumanMessage(content='Give me one quick way to reduce it.', additional_kwargs={}, response_metadata={}),
 AIMessage(content='One quick way to reduce overfitting is to use regularization techniques, such as L1 or L2 regularization.', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[]),
 HumanMessage(content='Give me one quick way to reduce it.', additional_kwargs={}, response_metadata={}),
 AIMessage(content='One quick way to reduce overfitting is to use regularization techniques, such as L1 or L2 regularization.', additional_kwargs={}, response_metadata={}, tool_calls=[], invalid_tool_calls=[])]
```

Output note: You should see a list of HumanMessage and AIMessage objects with the new turn appended.

Key takeaways

- PromptTemplate -> outputs **one string**; you must manage history formatting yourself.
- ChatPromptTemplate -> outputs **messages**; history stays structured and reusable.
- For real chat apps (memory, tools, multi-turn), **ChatPromptTemplate is the safer default**.