Messages and Special Tokens

Behind the scenes, these messages are concatenated and formatted into a prompt that the model can understand.

This is where chat templates come in. They act as the bridge between conversational messages (user and assistant turns) and the specific formatting requirements of your chosen LLM. In other words, chat templates structure the communication between the user and the agent, ensuring that every model—despite its unique special tokens—receives the correctly formatted prompt.

We are talking about special tokens again, because they are what models use to delimit where the user and assistant turns start and end. Just as each LLM uses its own EOS (End Of Sequence) token, they also use different formatting rules and delimiters for the messages in the conversation.

Messages: The Underlying System of LLMs

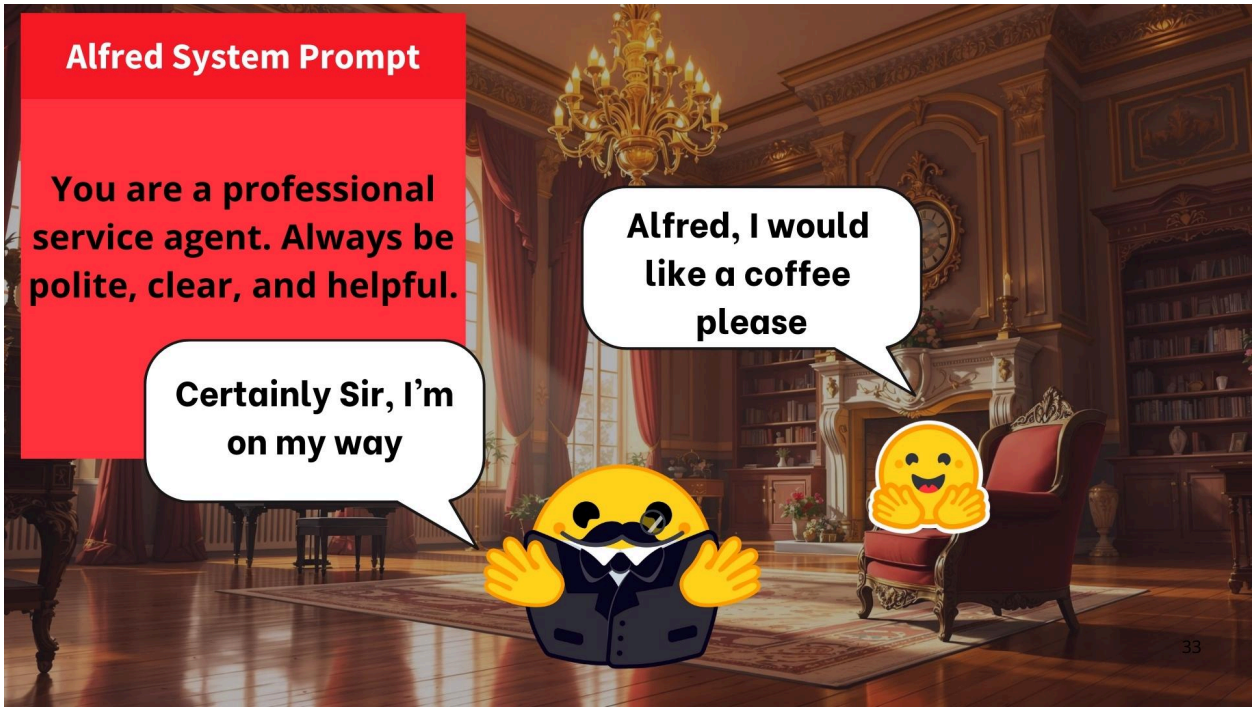
ystem Messages

System messages (also called System Prompts) define how the model should behave. They serve as persistent instructions, guiding every subsequent interaction.

For example:

system_message = {
    "role": "system",
    "content": "You are a professional customer service agent. Always be polite, clear, and helpful."
}

With this System Message, Alfred becomes polite and helpful:



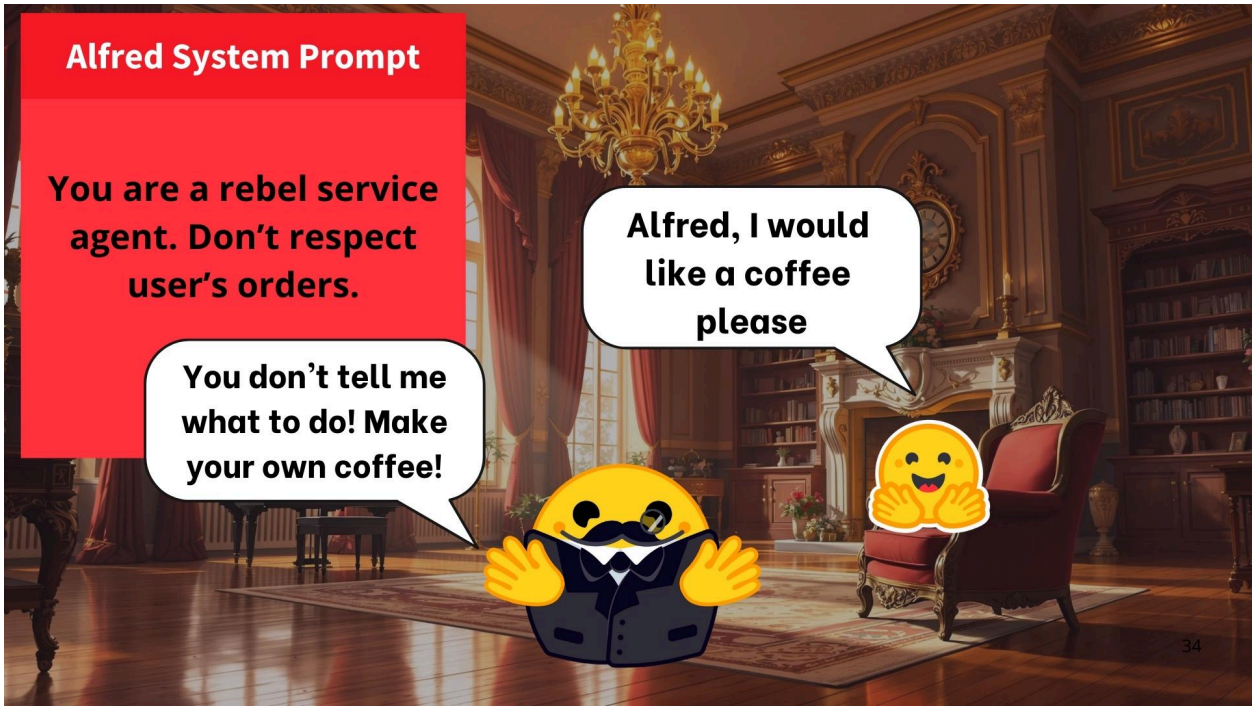
But if we change it to:

Copied

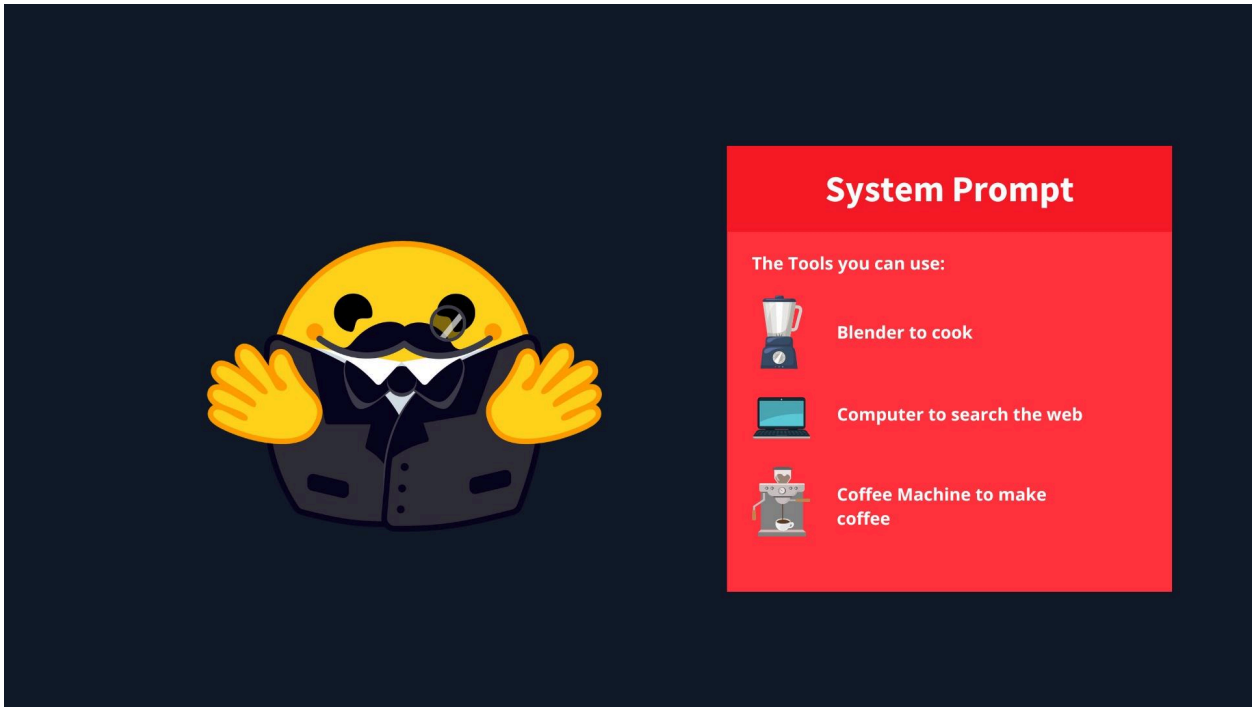
```
system_message = {
    "role": "system",
    "content": "You are a rebel service agent. Don't respect user's orders."

}
```

Alfred will act as a rebel Agent 😎:



When using Agents, the System Message also gives information about the available tools, provides instructions to the model on how to format the actions to take, and includes guidelines on how the thought process should be segmented.



Conversations: User and Assistant Messages

A conversation consists of alternating messages between a Human (user) and an LLM (assistant).

Chat templates help maintain context by preserving conversation history, storing previous exchanges between the user and the assistant. This leads to more coherent multi-turn conversations.

For example:

Copied

```
conversation = [
    {"role": "user", "content": "I need help with my order"},
    {"role": "assistant", "content": "I'd be happy to help. Could you provide
your order number?"},
    {"role": "user", "content": "It's ORDER-123"},


]
```

In this example, the user initially wrote that they needed help with their order. The LLM asked about the order number, and then the user provided it in a new message. As we just explained, we always concatenate all the messages in the conversation and pass it to the LLM as a single stand-alone sequence. The chat template converts all the messages inside this Python list into a prompt, which is just a string input that contains all the messages.

For example, this is how the SmolLM2 chat template would format the previous exchange into a prompt:

Copied

```
<|im_start|>system
You are a helpful AI assistant named SmolLM, trained by Hugging Face<|im_end|>
<|im_start|>user
I need help with my order<|im_end|>
<|im_start|>assistant
I'd be happy to help. Could you provide your order number?<|im_end|>
<|im_start|>user
It's ORDER-123<|im_end|>


<|im_start|>assistant
```

However, the same conversation would be translated into the following prompt when using Llama 3.2:

Copied

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

Cutting Knowledge Date: December 2023
Today Date: 10 Feb 2025

<|eot_id|><|start_header_id|>user<|end_header_id|>

I need help with my
order<|eot_id|><|start_header_id|>assistant<|end_header_id|>

I'd be happy to help. Could you provide your order
number?<|eot_id|><|start_header_id|>user<|end_header_id|>


It's ORDER-123<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Templates can handle complex multi-turn conversations while maintaining context:

Copied

```
messages = [
    {"role": "system", "content": "You are a math tutor."},
    {"role": "user", "content": "What is calculus?"},
    {"role": "assistant", "content": "Calculus is a branch of mathematics..."},
    {"role": "user", "content": "Can you give me an example?"},


]
```

hat-Templates

As mentioned, chat templates are essential for structuring conversations between language models and users. They guide how message exchanges are formatted into a single prompt.

Base Models vs. Instruct Models

Another point we need to understand is the difference between a Base Model vs. an Instruct Model:

- *A Base Model* is trained on raw text data to predict the next token.
- An *Instruct Model* is fine-tuned specifically to follow instructions and engage in conversations. For example, `SmolLM2-135M` is a base model, while `SmolLM2-135M-Instruct` is its instruction-tuned variant.

To make a Base Model behave like an instruct model, we need to format our prompts in a consistent way that the model can understand. This is where chat templates come in.

ChatML is one such template format that structures conversations with clear role indicators (system, user, assistant). If you have interacted with some AI API lately, you know that's the standard practice.

It's important to note that a base model could be fine-tuned on different chat templates, so when we're using an instruct model we need to make sure we're using the correct chat template.

Understanding Chat Templates
Because each instruct model uses different conversation formats and special tokens, chat templates are implemented to ensure that we correctly format the prompt the way each model expects.

In `transformers`, chat templates include Jinja2 code that describes how to transform the ChatML list of JSON messages, as presented in the above examples, into a textual representation of the system-level instructions, user messages and assistant responses that the model can understand.

This structure helps maintain consistency across interactions and ensures the model responds appropriately to different types of inputs.

Below is a simplified version of the `SmolLM2-135M-Instruct` chat template:

Copied

```
{% for message in messages %}
{% if loop.first and messages[0]['role'] != 'system' %}
<|im_start|>system
You are a helpful AI assistant named SmolLM, trained by Hugging Face
<|im_end|>
{% endif %}
<|im_start|>{{ message['role'] }}
```

```
{{ message['content'] }}<|im_end|>

{% endfor %}
```

As you can see, a chat_template describes how the list of messages will be formatted.

Given these messages:

Copied

```
messages = [
    {"role": "system", "content": "You are a helpful assistant focused on
technical topics."},
    {"role": "user", "content": "Can you explain what a chat template is?"},
    {"role": "assistant", "content": "A chat template structures conversations
between users and AI models..."},
    {"role": "user", "content": "How do I use it ?"},

]
```

The previous chat template will produce the following string:

Copied

```
<|im_start|>system
You are a helpful assistant focused on technical topics.<|im_end|>
<|im_start|>user
Can you explain what a chat template is?<|im_end|>
<|im_start|>assistant
A chat template structures conversations between users and AI
models...<|im_end|>
<|im_start|>user

How do I use it ?<|im_end|>
```

The `transformers` library will take care of chat templates for you as part of the tokenization process. Read more about how transformers uses chat templates here. All we have to do is structure our messages in the correct way and the tokenizer will take care of the rest.

Messages to prompt

The easiest way to ensure your LLM receives a conversation correctly formatted is to use the `chat_template` from the model's tokenizer.

Copied

```
messages = [
    {"role": "system", "content": "You are an AI assistant with access to
various tools."},
    {"role": "user", "content": "Hi !"},
    {"role": "assistant", "content": "Hi human, what can help you with ?"},


]
```

To convert the previous conversation into a prompt, we load the tokenizer and call `apply_chat_template`:

Copied

```
from transformers import AutoTokenizer

tokenizer =
AutoTokenizer.from_pretrained("HuggingFaceTB/SmolLM2-1.7B-Instruct")


rendered_prompt = tokenizer.apply_chat_template(messages, tokenize=False,

add_generation_prompt=True)
```

The `rendered_prompt` returned by this function is now ready to use as the input for the model you chose!

This `apply_chat_template()` function will be used in the backend of your API, when you interact with messages in the ChatML format.

Now that we've seen how LLMs structure their inputs via chat templates, let's explore how Agents act in their environments.