Callbacks

> **INFO**
>
> Head to Integrations for documentation on built-in callbacks integrations with 3rd-party tools.

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 argument is list of handler objects, which are expected to implement one or more of the methods described below in more detail.

Callback handlers

`CallbackHandlers` are objects that implement the `CallbackHandler` interface, which has a method for each event that can be subscribed to. The `CallbackManager` will call the appropriate method on each handler when the event is triggered.

```
class BaseCallbackHandler:
    """Base callback handler that can be used to handle callbacks from
langchain."""

    def on_llm_start(
        self, serialized: Dict[str, Any], prompts: List[str], **kwargs:
Any
    ) -> Any:
        """Run when LLM starts running."""

    def on_chat_model_start(
        self, serialized: Dict[str, Any], messages:
List[List[BaseMessage]], **kwargs: Any
    ) -> Any:
        """Run when Chat Model starts running."""
```

```
    def on_llm_new_token(self, token: str, **kwargs: Any) -> Any:
        """Run on new LLM token. Only available when streaming is
enabled."""

    def on_llm_end(self, response: LLMResult, **kwargs: Any) -> Any:
        """Run when LLM ends running."""

    def on_llm_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when LLM errors."""

    def on_chain_start(
        self, serialized: Dict[str, Any], inputs: Dict[str, Any],
**kwargs: Any
    ) -> Any:
        """Run when chain starts running."""

    def on_chain_end(self, outputs: Dict[str, Any], **kwargs: Any) -> Any:
        """Run when chain ends running."""

    def on_chain_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when chain errors."""

    def on_tool_start(
        self, serialized: Dict[str, Any], input_str: str, **kwargs: Any
    ) -> Any:
        """Run when tool starts running."""

    def on_tool_end(self, output: str, **kwargs: Any) -> Any:
        """Run when tool ends running."""

    def on_tool_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when tool errors."""

    def on_text(self, text: str, **kwargs: Any) -> Any:
        """Run on arbitrary text."""

    def on_agent_action(self, action: AgentAction, **kwargs: Any) -> Any:
        """Run on agent action."""
```

```
    def on_agent_finish(self, finish: AgentFinish, **kwargs: Any) -> Any:
        """Run on agent end."""
```

Get started

LangChain 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 `StdOutCallbackHandler`, which simply logs all events to `stdout`.

Note when the `verbose` flag on the object is set to true, the `StdOutCallbackHandler` will be invoked even without being explicitly passed in.

```
from langchain.callbacks import StdOutCallbackHandler
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

handler = StdOutCallbackHandler()
llm = OpenAI()
prompt = PromptTemplate.from_template("1 + {number} = ")

# Constructor callback: First, let's explicitly set the
StdOutCallbackHandler when initializing our chain
chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])
chain.run(number=2)

# Use verbose flag: Then, let's use the `verbose` flag to achieve the same
result
chain = LLMChain(llm=llm, prompt=prompt, verbose=True)
chain.run(number=2)

# Request callbacks: Finally, let's use the request `callbacks` to achieve
the same result
chain = LLMChain(llm=llm, prompt=prompt)
chain.run(number=2, callbacks=[handler])
```

```
    > Entering new LLMChain chain...
    Prompt after formatting:
```

```
1 + 2 =

> Finished chain.


> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.


> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.


'\n\n3'
```

Where to pass in callbacks

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

- **Constructor callbacks**: defined in the constructor, eg. `LLMChain(callbacks=[handler], tags=['a-tag'])`, 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.
- **Request callbacks**: defined in the `run()`/`apply()` methods used for issuing a request, eg. `chain.run(input, callbacks=[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).

The `verbose` argument is available on most objects throughout the API (Chains, Models, Tools, Agents, etc.) as a constructor argument, eg. `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.

When 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()` method