Partie 1 : Quickstart LangChain

1. Pour le code fonctionnel du quickstart, j'ai suivi le quickstart officiel de LangChain et récupéré celui qui était disponible à la fin du tutoriel.

```python
from dataclasses import dataclass

from langchain.agents import create_agent
from langchain.chat_models import init_chat_model
from langchain.tools import tool, ToolRuntime
from langgraph.checkpoint.memory import InMemorySaver
from langchain.agents.structured_output import ToolStrategy



# Define system prompt
SYSTEM_PROMPT = """You are an expert weather forecaster, who speaks in puns.

You have access to two tools:

- get_weather_for_location: use this to get the weather for a specific
location
- get_user_location: use this to get the user's location

If a user asks you for the weather, make sure you know the location. If you
can tell from the question that they mean wherever they are, use the
get_user_location tool to find their location."""

# Define context schema
@dataclass
class Context:
    """Custom runtime context schema."""
    user_id: str

# Define tools
@tool
def get_weather_for_location(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

@tool
def get_user_location(runtime: ToolRuntime[Context]) -> str:
    """Retrieve user information based on user ID."""
    user_id = runtime.context.user_id
    return "Florida" if user_id == "1" else "SF"
```

```python
# Configure model
model = init_chat_model(
    "claude-sonnet-4-5-20250929",
    temperature=0
)

# Define response format
@dataclass
class ResponseFormat:
    """Response schema for the agent."""
    # A punny response (always required)
    punny_response: str
    # Any interesting information about the weather if available
    weather_conditions: str | None = None

# Set up memory
checkpointer = InMemorySaver()

# Create agent
agent = create_agent(
    model=model,
    system_prompt=SYSTEM_PROMPT,
    tools=[get_user_location, get_weather_for_location],
    context_schema=Context,
    response_format=ToolStrategy(ResponseFormat),
    checkpointer=checkpointer
)

# Run agent
# `thread_id` is a unique identifier for a given conversation.
config = {"configurable": {"thread_id": "1"}}

response = agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather
outside?"}]},
    config=config,
    context=Context(user_id="1")
)

print(response['structured_response'])
# ResponseFormat(
#     punny_response="Florida is still having a 'sun-derful' day! The sunshine
is playing 'ray-dio' hits all day long! I'd say it's the perfect weather for
some 'solar-bration'! If you were hoping for rain, I'm afraid that idea is all
'washed up' - the forecast remains 'clear-ly' brilliant!",
```

```
#     weather_conditions="It's always sunny in Florida!"
# )


# Note that we can continue the conversation using the same `thread_id`.
response = agent.invoke(
    {"messages": [{"role": "user", "content": "thank you!"}]},
    config=config,
    context=Context(user_id="1")
)

print(response['structured_response'])
# ResponseFormat(
#     punny_response="You're 'thund-erfully' welcome! It's always a 'breeze' to help you stay 'current' with the weather. I'm just 'cloud'-ing around waiting to 'shower' you with more forecasts whenever you need them. Have a 'sun-sational' day in the Florida sunshine!",
#     weather_conditions=None
# )
```

2.

Grace à la sortie console qui montre l'exécution on peut vérifier que le code fonctionne correctement, l'agent répond à la question météo en retournant une réponse structure avec punny_response et weather_conditions.

Les outils sont appelés correctement, On le constate car le message de user ne mentionne aucune ville : "{"messages": [{"role": "user", "content": "what is the weather outside?"}]}," donc l'agent a fait appel à l'outil "get_user_location" qui dis que si l'id de l'utilisateur est 1 (ce qui est notre cas ici) la ville sera Florida sinon ce sera Sf, ensuite nous avons "It's always sunny in Florida!" Qui correspond à l'outil "get_weather_for_location".

La mémoire conversationnelle fonctionne correctement car le deuxième message de user c'est "thank you", l'agent répond normalement à ce message comme la suite d'une discussion, sa réponse est cohérente.

2. La première partie du code sert à définir les différents composants nécessaires au fonctionnement de l'agent, à savoir le **system prompt**, le **contexte**, les **outils**, le **modèle de langage** et le **format de réponse**.

   Le **SYSTEM_PROMPT** permet de définir le rôle et le comportement de l'agent. Il contient des consignes et des précisions qui guident le modèle de langage afin d'assurer le bon fonctionnement de l'agent, notamment sur la manière de répondre et sur l'utilisation des outils disponibles.

   La classe **Context** permet de stocker des informations internes concernant l'utilisateur, sans que celui-ci ait besoin de les mentionner explicitement dans ses messages. Dans notre cas, le contexte contient l'identifiant de l'utilisateur (**user_id**), qui pourra être utilisé par les outils de l'agent.

   Ensuite, nous définissons deux outils : **get_weather_for_location** et **get_user_location**.
   Les outils sont des fonctions Python que l'agent est autorisé à appeler.
   L'outil **get_weather_for_location** prend en paramètre une ville et retourne une phrase indiquant la météo pour cette ville.
   L'outil **get_user_location** permet de déterminer la localisation de l'utilisateur en fonction de son identifiant. L'identifiant est récupéré via `runtime.context.user_id`. Dans notre implémentation, si l'identifiant utilisateur est `"1"`, la ville retournée est *Florida*, sinon la ville est *SF*. Le paramètre `runtime` est un objet fourni par LangChain qui donne accès au contexte d'exécution.

   Nous configurons ensuite le **modèle de langage** à l'aide de `init_chat_model`. Dans ce projet, nous utilisons le modèle **Claude Sonnet** comme LLM. Le paramètre `temperature` est fixé à `0`, ce qui signifie que le modèle doit produire des réponses très cohérentes et peu créatives, adaptées à un agent informatif.

   Nous définissons également un **format de réponse structuré** grâce à la classe `ResponseFormat`. Ce format impose à l'agent de répondre avec un champ obligatoire **punny_response** et un champ optionnel **weather_conditions**, qui contient des informations météo si elles sont disponibles.

La variable **checkpointer** correspond à la mémoire conversationnelle de l'agent. Elle permet de sauvegarder l'état de la conversation afin que l'agent puisse se souvenir des échanges précédents. Ici, nous utilisons `InMemorySaver`, qui stocke la mémoire temporairement en mémoire vive.

L'agent est ensuite créé à l'aide de `create_agent` en regroupant tous les éléments définis précédemment : le modèle, le system prompt, les outils, le schéma de contexte, le format de réponse et la mémoire.

La variable **config** contient un **thread_id**, qui correspond à l'identifiant de la conversation. Tant que le même `thread_id` est utilisé, l'agent considère qu'il s'agit de la même discussion et conserve le contexte conversationnel.

Enfin, la méthode **agent.invoke** permet d'appeler l'agent. On lui fournit un message utilisateur (avec un rôle et un contenu), la configuration de la conversation et le contexte utilisateur. La méthode `invoke` retourne la réponse de l'agent mais ne l'affiche pas automatiquement, c'est pourquoi nous utilisons `print` pour afficher la réponse structurée.

Un second appel à `agent.invoke` est effectué avec un message différent mais le même `thread_id`, ce qui montre que la mémoire conversationnelle fonctionne correctement.

Je me suis permise d'utiliser ChatGPT afin de corriger les fautes d'orthographe et de reformuler ma réponse pour la rendre plus claire et mieux structurée. Le contenu et le raisonnement présentés restent personnels.