from typing import Optional

from pydantic import BaseModel

from langgraph.graph import StateGraph, END

from langchain\_core.runnables import RunnableLambda

# Define a simple state

class SimpleState(BaseModel):

should\_end: bool = False

message: str = ""

# Define a node that might set the 'should\_end' flag

def check\_condition\_agent():

def \_run(state: SimpleState) -> SimpleState:

print(f"--- Inside check\_condition\_agent ---")

print(f"State on entry: {state}")

# Example condition: if the message contains "stop", set should\_end to True

if "stop" in state.message.lower():

state.should\_end = True

print("Condition met: Setting should\_end to True.")

else:

print("Condition not met.")

print(f"State before returning from check\_condition\_agent: {state}")

print(f"--- Exiting check\_condition\_agent ---")

return state

return \_run

# Define a node that runs if the graph doesn't end

def continue\_processing\_agent():

def \_run(state: SimpleState) -> SimpleState:

print(f"--- Inside continue\_processing\_agent ---")

print(f"State on entry: {state}")

state.message += " ... continued processing."

print(f"State before returning from continue\_processing\_agent: {state}")

print(f"--- Exiting continue\_processing\_agent ---")

return state

return \_run

# Build the graph

def build\_conditional\_end\_graph():

graph = StateGraph(SimpleState)

# Add nodes

graph.add\_node("check\_condition", RunnableLambda(check\_condition\_agent()))

graph.add\_node("continue\_processing", RunnableLambda(continue\_processing\_agent()))

# Set entry point

graph.set\_entry\_point("check\_condition")

# Define a conditional edge from check\_condition using the correct syntax

# The callable (lambda) determines the next step based on the state

# The dictionary maps the return values of the callable to nodes or END

graph.add\_conditional\_edges(

"check\_condition", # Start node

lambda state: "end" if state.should\_end else "continue", # Callable returning a string key

{

"end": END, # Map the string key "end" to END

"continue": "continue\_processing" # Map the string key "continue" to the node name

}

)

# Define edge from continue\_processing to END (if it gets this far)

graph.add\_edge("continue\_processing", END)

# Note: You don't necessarily need a set\_finish\_point if all paths lead to END.

# But it can be useful for clarity or if some paths don't explicitly end.

# Here, both paths lead to END.

return graph.compile()

# Build and test the graph

print("Building conditional end graph...")

conditional\_end\_graph = build\_conditional\_end\_graph()

print("Conditional end graph built successfully.")

# Test case 1: Condition met, graph ends early

test\_input\_stop = SimpleState(message="Please stop processing.")

print(f"\n--- Invoking graph with message: '{test\_input\_stop.message}' (Expecting early END) ---")

try:

result\_stop = conditional\_end\_graph.invoke(test\_input\_stop)

print("\n--- Result from graph (Early END) ---")

print(result\_stop) # The state at the point of ending

except Exception as e:

print(f"\nAn error occurred: {e}")

# Test case 2: Condition not met, graph continues

test\_input\_continue = SimpleState(message="Keep going.")

print(f"\n--- Invoking graph with message: '{test\_input\_continue.message}' (Expecting continue) ---")

try:

result\_continue = conditional\_end\_graph.invoke(test\_input\_continue)

print("\n--- Result from graph (Continue to END) ---")

print(result\_continue) # The state at the final END node

except Exception as e:

print(f"\nAn error occurred: {e}")