```
import random
from swarms.structs.base_swarm import BaseSwarm
from typing import List
from swarms.structs.agent import Agent
from pydantic import BaseModel, Field
from typing import Optional
from datetime import datetime
from swarms.schemas.agent_step_schemas import ManySteps
import tenacity
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger("round-robin")

datetime_stamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")


class MetadataSchema(BaseModel):
    swarm_id: Optional[str] = Field(
        ..., description="Unique ID for the run"
    )
    name: Optional[str] = Field(
        "RoundRobinSwarm", description="Name of the swarm"
    )
    task: Optional[str] = Field(
        ..., description="Task or query given to all agents"
    )
    description: Optional[str] = Field(
        "Concurrent execution of multiple agents",
        description="Description of the workflow",
    )
    agent_outputs: Optional[List[ManySteps]] = Field(
        ..., description="List of agent outputs and metadata"
    )
    timestamp: Optional[str] = Field(
        default_factory=datetime.now,
        description="Timestamp of the workflow execution",
    )
    max_loops: Optional[int] = Field(
        1, description="Maximum number of loops to run"
    )


class RoundRobinSwarm(BaseSwarm):
    """
    A swarm implementation that executes tasks in a round-robin fashion.

    Args:
        agents (List[Agent], optional): List of agents in the swarm. Defaults to None.
        verbose (bool, optional): Flag to enable verbose mode. Defaults to False.
        max_loops (int, optional): Maximum number of loops to run. Defaults to 1.
        callback (callable, optional): Callback function to be called after each loop. Defaults to None.
            return_json_on (bool, optional): Flag to return the metadata as a JSON object. Defaults to
```

```
False.
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.

    Attributes:
        agents (List[Agent]): List of agents in the swarm.
        verbose (bool): Flag to enable verbose mode.
        max_loops (int): Maximum number of loops to run.
        index (int): Current index of the agent being executed.

    Methods:
        run(task: str, *args, **kwargs) -> Any: Executes the given task on the agents in a round-robin
fashion.

    """

    def __init__(
        self,
        name: str = "RoundRobinSwarm",
        description: str = "A swarm implementation that executes tasks in a round-robin fashion.",
        agents: List[Agent] = None,
        verbose: bool = False,
        max_loops: int = 1,
        callback: callable = None,
        return_json_on: bool = False,
        max_retries: int = 3,
    *args,
    **kwargs,
):
    try:
        super().__init__(
            name=name,
            description=description,
            agents=agents,
            *args,
            **kwargs,
        )
        self.name = name
        self.description = description
        self.agents = agents or []
        self.verbose = verbose
        self.max_loops = max_loops
        self.callback = callback
        self.return_json_on = return_json_on
        self.index = 0
        self.max_retries = max_retries

        # Store the metadata for the run
        self.output_schema = MetadataSchema(
            name=self.name,
            swarm_id=datetime_stamp,
            task="",
                description=self.description,
                agent_outputs=[],
                timestamp=datetime_stamp,
                max_loops=self.max_loops,
            )

            # Set the max loops for every agent
            if self.agents:
                for agent in self.agents:
                    agent.max_loops = random.randint(1, 5)

            logger.info(
                f"Successfully initialized {self.name} with {len(self.agents)} agents"
            )

        except Exception as e:
            logger.error(
                f"Failed to initialize {self.name}: {str(e)}"
            )
            raise

    @tenacity.retry(
        stop=tenacity.stop_after_attempt(3),
        wait=tenacity.wait_exponential(multiplier=1, min=4, max=10),
        retry=tenacity.retry_if_exception_type(Exception),
        before_sleep=lambda retry_state: logger.info(
            f"Retrying in {retry_state.next_action.sleep} seconds..."
        ),
    )
    def _execute_agent(
        self, agent: Agent, task: str, *args, **kwargs
    ) -> str:
        """Execute a single agent with retries and error handling"""
        try:
            logger.info(
                f"Running Agent {agent.agent_name} on task: {task}"
            )
            result = agent.run(task, *args, **kwargs)
            self.output_schema.agent_outputs.append(
                agent.agent_output
            )
            return result
        except Exception as e:
            logger.error(
                f"Error executing agent {agent.agent_name}: {str(e)}"
            )
            raise

    def run(self, task: str, *args, **kwargs):
        """
        Executes the given task on the agents in a round-robin fashion.
```

```
Args:

    task (str): The task to be executed.

    *args: Variable length argument list.

    **kwargs: Arbitrary keyword arguments.


Returns:

    Any: The result of the task execution.


Raises:

    ValueError: If no agents are configured

    Exception: If an exception occurs during task execution.

"""

if not self.agents:

    logger.error("No agents configured for the swarm")

    raise ValueError("No agents configured for the swarm")


try:

    result = task

    self.output_schema.task = task

    n = len(self.agents)

    logger.info(

        f"Starting round-robin execution with task '{task}' on {n} agents"

    )


    for loop in range(self.max_loops):

        logger.debug(
        f"Starting loop {loop + 1}/{self.max_loops}"
    )

    for _ in range(n):
        current_agent = self.agents[self.index]
        try:
            result = self._execute_agent(
                current_agent, result, *args, **kwargs
            )
        finally:
            self.index = (self.index + 1) % n

    if self.callback:
        logger.debug(
            f"Executing callback for loop {loop + 1}"
        )
        try:
            self.callback(loop, result)
        except Exception as e:
            logger.error(
                f"Callback execution failed: {str(e)}"
            )

logger.success(
    f"Successfully completed {self.max_loops} loops of round-robin execution"
)
```

```
        if self.return_json_on:
            return self.export_metadata()
        return result

    except Exception as e:
        logger.error(f"Round-robin execution failed: {str(e)}")
        raise

def export_metadata(self):
    """Export the execution metadata as JSON"""
    try:
        return self.output_schema.model_dump_json(indent=4)
    except Exception as e:
        logger.error(f"Failed to export metadata: {str(e)}")
        raise
```