```
import os
from typing import Any, Callable, Dict, List, Optional
import time
from pydantic import BaseModel, Field
from concurrent.futures import ThreadPoolExecutor, as_completed
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="tool_registry")


class ToolMetadata(BaseModel):
    name: str
    documentation: Optional[str] = None
    time_created: str = Field(
        time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime()),
        description="Time when the tool was added to the registry.",
    )


class ToolStorageSchema(BaseModel):
    name: str
    description: str
    tools: List[ToolMetadata]
    time_created: str = Field(
        time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime()),
        description="Time when the registry was created.",
    )



class ToolStorage:
    """
    A class that represents a storage for tools.

    Attributes:
        verbose (bool): A flag to enable verbose logging.
        tools (List[Callable]): A list of tool functions.
        _tools (Dict[str, Callable]): A dictionary that stores the tools, where the key is the tool name and
the value is the tool function.
        _settings (Dict[str, Any]): A dictionary that stores the settings, where the key is the setting name
and the value is the setting value.
    """

    def __init__(
        self,
        name: str = None,
        description: str = None,
        verbose: bool = None,
        tools: List[Callable] = None,
        *args,
        **kwargs,
    ) -> None:
        self.name = name
        self.description = description
        self.verbose = verbose
        self.tools = tools
        # self.tool_storage_schema = tool_storage_schema
        self._tools: Dict[str, Callable] = {}
        self._settings: Dict[str, Any] = {}
        self.tool_storage_schema = ToolStorageSchema(
            name=name,
            description=description,
            tools=[],
        )

        # Pool
        self.pool = ThreadPoolExecutor(max_workers=os.cpu_count())

    def add_tool(self, func: Callable) -> None:
        """
        Adds a tool to the storage.

        Args:
            func (Callable): The tool function to be added.

        Raises:
            ValueError: If a tool with the same name already exists.
        """
        try:
                name = func.__name__
                docs = func.__doc__

                self.add_tool_to_log(name, docs)

                logger.info(f"Adding tool: {name}")
                if name in self._tools:
                    raise ValueError(
                        f"Tool with name {name} already exists."
                    )
                self._tools[name] = func
                logger.info(f"Added tool: {name}")
            except ValueError as e:
                logger.error(e)
                raise

        def add_many_tools(self, funcs: List[Callable]) -> None:
            """
            Adds multiple tools to the storage.

            Args:
                funcs (List[Callable]): The list of tool functions to be added.
            """
            # Upload many tools
            with ThreadPoolExecutor(
                max_workers=os.cpu_count()
            ) as executor:
                futures = [
                    executor.submit(self.add_tool, func) for func in funcs
                ]
                for future in as_completed(futures):
                    try:
                        future.result()
                    except Exception as e:
                        logger.error(f"Error adding tool: {e}")

        def get_tool(self, name: str) -> Callable:
            """
            Retrieves a tool by its name.


            Args:
                name (str): The name of the tool to retrieve.


            Returns:
                Callable: The tool function.


            Raises:
                ValueError: If no tool with the given name is found.
            """
            try:
                logger.info(f"Getting tool: {name}")
                if name not in self._tools:
                    raise ValueError(f"No tool found with name: {name}")
                return self._tools[name]
            except ValueError as e:
                logger.error(e)
                raise

        def set_setting(self, key: str, value: Any) -> None:
            """
            Sets a setting in the storage.

            Args:
                key (str): The key for the setting.
                value (Any): The value for the setting.
            """
            self._settings[key] = value
            logger.info(f"Setting {key} set to {value}")

        def get_setting(self, key: str) -> Any:
            """
            Gets a setting from the storage.

            Args:
                key (str): The key for the setting.

            Returns:
                Any: The value of the setting.
            Raises:
                KeyError: If the setting is not found.
            """
            try:
                return self._settings[key]
            except KeyError as e:
                logger.error(f"Setting {key} not found error: {e}")
                raise

        def list_tools(self) -> List[str]:
            """
            Lists all registered tools.

            Returns:
                List[str]: A list of tool names.
            """
            # return list(self._tools.keys())
            return self.tool_storage_schema.model_dump_json(indent=4)

        def add_tool_to_log(self, name: str, docs: str, *args, **kwargs):
            log = ToolMetadata(
                name=name,
                documentation=docs,
            )
```

```
        self.tool_storage_schema.tools.append(log)


    def add_multiple_tools_to_log(
        self,
        names: List[str],
        docs: List[str],
        *args,
        **kwargs,
    ):
        for name, docs in zip(names, docs):
            self.add_tool_to_log(name, docs)



# Decorator
def tool_registry(storage: ToolStorage = None) -> Callable:
    """
    A decorator that registers a function as a tool in the storage.


    Args:
        storage (ToolStorage): The storage instance to register the tool in.


    Returns:
        Callable: The decorator function.
    """


    def decorator(func: Callable) -> Callable:
        name = func.__name__

        logger.info(f"Registering tool: {name}")
        storage.add_tool(func)

        def wrapper(*args, **kwargs):
            try:
                result = func(*args, **kwargs)
                logger.info(f"Tool {name} executed successfully")
                return result
            except Exception as e:
                logger.error(f"Error executing tool {name}: {e}")
                raise

        logger.info(f"Registered tool: {name}")
        return wrapper

    return decorator


# storage = ToolStorage(
#     name="Tool Storage",
#     description="A storage for tools.",
# )
# # Tools

# @tool_registry(storage)

# def example_tool(a: int, b: int) -> int:

#     """

#     An example tool that adds two numbers.


#     Args:

#         a (int): The first number.

#         b (int): The second number.


#     Returns:

#         int: The sum of the two numbers.

#     """

#     return a + b



# def sample_api_tool(a: int, b: int) -> int:

#     """

#     An example tool that adds two numbers.


#     Args:

#         a (int): The first number.

#         b (int): The second number.


#     Returns:

#         int: The sum of the two numbers.
#     """
#     return a + b


# def use_example_tool(a: int, b: int) -> int:
#     """
#     A function that uses the example tool.

#     Args:
#         a (int): The first number.
#         b (int): The second number.

#     Returns:
#         int: The result of the example tool.
#     """
#     tool = storage.get_tool("example_tool")
#     return tool(a, b)

# # Test the storage and querying
# if __name__ == "__main__":
#     # print(storage.list_tools())  # Should print ['example_tool']
#     storage.add_many_tools(
#         [
#             example_tool,
#             sample_api_tool,
#             use_example_tool
#         ]
#     )
#     # print(use_example_tool(2, 3))  # Should print 5
#     storage.set_setting("example_setting", 42)
#     print(storage.get_setting("example_setting"))  # Should print 42
#     print(storage.list_tools())  # Should print ['example_tool', 'sample_api_tool']
```