```
from typing import List, Callable

from swarm_models.fuyu import Fuyu  # noqa: E402

from swarm_models.gpt4_vision_api import GPT4VisionAPI  # noqa: E402

from swarm_models.huggingface import HuggingfaceLLM  # noqa: E402

from swarm_models.idefics import Idefics  # noqa: E402

from swarm_models.kosmos_two import Kosmos  # noqa: E402

from swarm_models.layoutlm_document_qa import LayoutLMDocumentQA

from swarm_models.llama3_hosted import llama3Hosted

from swarm_models.llava import LavaMultiModal  # noqa: E402

from swarm_models.nougat import Nougat  # noqa: E402

from swarm_models.openai_embeddings import OpenAIEmbeddings

from swarm_models.openai_tts import OpenAITTS  # noqa: E402

from swarm_models.palm import GooglePalm as Palm  # noqa: E402

from swarm_models.popular_llms import Anthropic as Anthropic

from swarm_models.popular_llms import (

    AzureOpenAILLM as AzureOpenAI,

)

from swarm_models.popular_llms import (

    CohereChat as Cohere,

)

from swarm_models.popular_llms import FireWorksAI, OctoAIChat

from swarm_models.popular_llms import (

    OpenAIChatLLM as OpenAIChat,

)

from swarm_models.popular_llms import (

    OpenAILLM as OpenAI,
)
from swarm_models.popular_llms import ReplicateChat as Replicate
from swarm_models.qwen import QwenVLMultiModal  # noqa: E402
from swarm_models.sampling_params import SamplingParams
from swarm_models.together import TogetherLLM  # noqa: E402
from swarm_models.vilt import Vilt  # noqa: E402
from loguru import logger


# # New type BaseLLM and BaseEmbeddingModel and BaseMultimodalModel
# omni_model_type = Union[
#     BaseLLM, BaseEmbeddingModel, BaseMultiModalModel, callable
# ]
# list_of_Callable = List[Callable]



models = [
    Fuyu,
    GPT4VisionAPI,
    HuggingfaceLLM,
    Idefics,
    Kosmos,
    LayoutLMDocumentQA,
    llama3Hosted,
    LavaMultiModal,
    Nougat,
    OpenAIEmbeddings,
    OpenAITTS,
    Palm,
    Anthropic,
    AzureOpenAI,
    Cohere,
    OctoAIChat,
    OpenAIChat,
    OpenAI,
    Replicate,
    QwenVLMultiModal,
    SamplingParams,
    TogetherLLM,
    Vilt,
    FireWorksAI,
    # OpenAIFunctionCaller,
]


class ModelRouter:
    """
    A router for managing multiple models.

    Attributes:
        model_router_id (str): The ID of the model router.
        model_router_description (str): The description of the model router.
        model_pool (List[Callable]): The list of models in the model pool.
    Methods:
        check_for_models(): Checks if there are any models in the model pool.
        add_model(model: Callable): Adds a model to the model pool.
        add_models(models: List[Callable]): Adds multiple models to the model pool.
         get_model_by_name(model_name: str) -> Callable: Retrieves a model from the model pool by
its name.
            get_multiple_models_by_name(model_names: List[str]) -> List[Callable]: Retrieves multiple
models from the model pool by their names.
        get_model_pool() -> List[Callable]: Retrieves the entire model pool.
           get_model_by_index(index: int) -> Callable: Retrieves a model from the model pool by its
index.
        get_model_by_id(model_id: str) -> Callable: Retrieves a model from the model pool by its ID.
        dict() -> dict: Returns a dictionary representation of the model router.

    """

    def __init__(
        self,
        model_router_id: str = "model_router",
        model_router_description: str = "A router for managing multiple models.",
        model_pool: List[Callable] = models,
        verbose: bool = False,
        *args,
        **kwargs,
    ):
```

```
    super().__init__(*args, **kwargs)
    self.model_router_id = model_router_id
    self.model_router_description = model_router_description
    self.model_pool = model_pool
    self.verbose = verbose

    self.check_for_models()
    # self.refactor_model_class_if_invoke()

def check_for_models(self):
    """
    Checks if there are any models in the model pool.

    Returns:
        None

    Raises:
        ValueError: If no models are found in the model pool.
    """
    if len(self.model_pool) == 0:
        raise ValueError("No models found in model pool.")

def add_model(self, model: Callable):
    """
    Adds a model to the model pool.
    Args:
        model (Callable): The model to be added.

    Returns:
        str: A success message indicating that the model has been added to the model pool.
    """
    logger.info(f"Adding model {model.name} to model pool.")
    self.model_pool.append(model)
    return "Model successfully added to model pool."

def add_models(self, models: List[Callable]):
    """
    Adds multiple models to the model pool.

    Args:
        models (List[Callable]): The models to be added.

    Returns:
        str: A success message indicating that the models have been added to the model pool.
    """
    logger.info("Adding models to model pool.")
    self.model_pool.extend(models)
    return "Models successfully added to model pool."

def get_multiple_models_by_name(
    self, model_names: List[str]
) -> List[Callable]:
    """
    Retrieves multiple models from the model pool by their names.

    Args:
        model_names (List[str]): The names of the models.

    Returns:
        List[Callable]: The list of model objects.

    Raises:
        ValueError: If any of the models with the given names are not found in the model pool.
    """
    logger.info(
        f"Retrieving multiple models {model_names} from model pool."
    )
    models = []
    for model_name in model_names:
        models.append(self.get_model_by_name(model_name))
    return models

def get_model_pool(self) -> List[Callable]:
    """
    Retrieves the entire model pool.

    Returns:
        List[Callable]: The list of model objects in the model pool.
    """
    return self.model_pool

def get_model_by_index(self, index: int) -> Callable:
    """
    Retrieves a model from the model pool by its index.

    Args:
        index (int): The index of the model in the model pool.

    Returns:
        Callable: The model object.

    Raises:
        IndexError: If the index is out of range.
    """
    return self.model_pool[index]

def get_model_by_name(self, model_name: str) -> Callable:
    """
    Retrieves a model from the model pool by its name.

    Args:
        model_name (str): The name of the model.
```

```
Returns:
    Callable: The model object.


Raises:
    ValueError: If the model with the given name is not found in the model pool.
"""
logger.info(f"Retrieving model {model_name} from model pool.")
for model in self.model_pool:
    # Create a list of possible names to check
    model_names = []


    # Check for the existence of attributes before accessing them
    if hasattr(model, "name"):
        model_names.append(model.name)
    if hasattr(model, "model_id"):
        model_names.append(model.model_id)
    if hasattr(model, "model_name"):
        model_names.append(model.model_name)


    # Check if the model_name is in the list of model names
    if model_name in model_names:
        return model


return model


# raise ValueError(f"Model {model_name} not found in model pool.")
```

```
    def refactor_model_class_if_invoke(self, *args, **kwargs):
        """
        Refactors the model class if it has an 'invoke' method.

        Checks to see if the model pool has a model with an 'invoke' method and refactors it to have a
'run' method and '__call__' method.

        Returns:
            str: A success message indicating that the model classes have been refactored.
        """
        for model in self.model_pool:
            if hasattr(model, "invoke"):
                model.run = model.invoke(*args, **kwargs)
                model.__call__ = model.invoke(*args, **kwargs)
                logger.info(
                    f"Refactored model {model.name} to have run and __call__ methods."
                )

                # Update the model in the model pool
                self.model_pool[self.model_pool.index(model)] = model

            if hasattr(model, "generate"):
                model.run = model.invoke(*args, **kwargs)
                model.__call__ = model.invoke(*args, **kwargs)
                logger.info(
                f"Refactored model {model.name} to have run and __call__ methods."
            )

            # Update the model in the model pool
            self.model_pool[self.model_pool.index(model)] = model

        return "Model classes successfully refactored."

    def refactor_model_class_if_invoke_class(
        self, model: callable, *args, **kwargs
    ) -> callable:
        """
        Refactors the model class if it has an 'invoke' method.

        Checks to see if the model pool has a model with an 'invoke' method and refactors it to have a
'run' method and '__call__' method.

        Returns:
            str: A success message indicating that the model classes have been refactored.
        """
        if hasattr(model, "invoke"):
            model.run = model.invoke
            model.__call__ = model.invoke
            logger.info(
                f"Refactored model {model.name} to have run and __call__ methods."
            )
        return model


    def find_model_by_name_and_run(
        self, model_name: str, task: str, *args, **kwargs
    ) -> str:
        """
        Finds a model by its name and runs a task on it.


        Args:
            model_name (str): The name of the model.
            task (str): The task to be run on the model.


        Returns:
            str: The result of running the task on the model.


        Raises:
            ValueError: If the model with the given name is not found in the model pool.
        """
        model = self.get_model_by_name(model_name)


        if model is None:
            raise ValueError(
                f"Model '{model_name}' not found in the model pool."
            )
            return model.run(task=task, *args, **kwargs)



    def run_model_router(
        model_name: str, task: str, *args, **kwargs
    ) -> str:
        logger.info(
            f"Running model router with {model_name} on task: {task}"
        )
        return ModelRouter().find_model_by_name_and_run(
            model_name, task, *args, **kwargs
        )



    # model = ModelRouter()
    # print(model.to_dict())
    # print(model.get_model_pool())
    # print(model.get_model_by_index(0))
    # print(model.get_model_by_id("stability-ai/stable-diffusion:"))
    # print(model.get_multiple_models_by_name(["gpt-4o", "gpt-4"]))
    print(run_model_router("gpt-4o-mini", "what's your name?"))
```