Hosted Project Packages Architecture

With the introduction of the new Queue-based architecture, a core goal is to decouple project dependencies and tasks into independent, self-sustaining modules. This allows for greater flexibility, resource management, and scalability.

New Architecture Overview

In contrast to the previous architecture, where each project's resolver handled all the logic for every task, the new design aims to isolate each task and project, giving them their own dependency criteria, tests, and logic. Each task is housed in a separate file, and a consistent, predictable pattern is enforced across projects to minimize the unique coding required for adding or changing tasks.

This modular approach offers several key advantages:

- **Independent Task Files:** Each task is associated with its parent project but is functionally independent. This makes it easier to scale specific tasks, especially those with high resource costs.
- **Resource Management:** Tasks can be grouped and processed based on their resource requirements. Tasks that require more computational power can be handled by different workers from tasks that are lightweight, even within the same project.
- **Dependency Management:** Each project can now define its own dependencies, decoupled from the main application. This keeps projects lightweight, improves maintainability, and avoids dependency conflicts.
- **Testability:** Each project and task comes with its own unit tests, allowing for isolated testing of functionality without affecting the rest of the system.

Class-Based Task Architecture IMPL

In the new system architecture, each task is structured as a class that behaves like a function. This is achieved through the `__call__` method of Python classes, enabling the class instance to be used as if it were a function. This approach provides several key advantages over traditional function-based task handling.

Benefits of Class-Based Task Design

1. **Metadata Management** By using classes for tasks, we can encapsulate all relevant metadata within the task object itself. This allows for:
   - Storing task-specific information such as `output_type` (whether it generates an artifact or a form for the next step), `data_type` (the type of data returned by the API), and flags such as whether the output contains files.
   - Avoiding clutter in function signatures by embedding this metadata within the task instance. This ensures a clean separation between task logic and operational details, without adding unnecessary function parameters or extra outputs.
2. **Separation of Lab Logic and Package Adaptation** Each task class inherits from a `BaseTask` class, which encapsulates general lab-related functionality, such as:
   - **Logging**: The `BaseTask` handles initiation, successful completion, and error logging.
   - **Metadata Saving**: Metadata related to the task can be automatically captured and stored, avoiding manual data management within the task's operational code.
   - **Exception Handling**: Any errors during the task's execution are automatically caught and logged, ensuring stability and maintainability.
   - **Output Formatting**: The task's output is formatted to conform to GenAI Lab standards without needing to embed this logic in the package-level code. This ensures that the task-specific logic remains focused on external package behavior, while Lab-specific requirements (such as output structures and validation) are handled by the `BaseTask` class.

This separation ensures a modular architecture where tasks can interact with both the lab and external packages seamlessly without mixing logic, making it easier to maintain, update, and extend tasks.

3. **Integration of Helper Functions** Class-based tasks can have inner methods that handle specific aspects of task logic, allowing for better organization and code reuse, providing clear and direct correlations between helper functions and the task they support.:
   - **Dynamic Form Generation**: The task class can include methods to handle the logic of dynamically generating the next step of an input form based on the task's current output. This makes it easier to manage multi-step workflows, where a task's output serves as the input for the next step.
   - **Helper Functions**: Helper methods can be added to handle sub-tasks or repetitive actions within the task. For example, a task might have helper methods for data validation, intermediate file processing, or API response parsing, all while being encapsulated within the same task class.

Structure of a Task Class

BaseTask

```python
class BaseTask:
    def __init__(self, task_name: str, project_name: str, output_type: str, data_type: str, contains_file_output: bool, file_suffix: str = None,
    ):
        self.task_name = task_name
        self.project_name = project_name
        self.data = {
            "output_type": output_type,
            "data_type": data_type,
            "contains_file_output": contains_file_output,
            "file_suffix": file_suffix,
        }
        logger.debug(f"Initialized {self.task_name} of {self.project_name}")

    async def __call__(self, *args, **kwargs) -> dict:
        try:
            result = await self.execute(*args, **kwargs)
            logger.debug(f"[{self.task_name}] Executed successfully")

            if isinstance(result, tuple) and len(result) == 2:
                raw, output = result
            else:
                raw = output = result

            response = {"code": http.HTTPStatus.OK, "value": output, "raw_output": raw}
            return response
        except Exception as e:
            logger.error(f"Error in {self.project_name}: {e}")
            raise Exception(f"Error in {self.project_name}: {e}")

    async def execute(self, *args, **kwargs):
        logger.error(f"Task {self.task_name} not implemented in {self.project_name}")
        raise NotImplementedError(f"Task {self.task_name} not implemented in {self.project_name}")

```

TaskExample

```python
class convert_vision_to_bpmn(BaseTask):
    def __init__(self, visio_to_bpmn):
        """
        Initializes the convert task.
        Dependencies:
```

```
            visio_to_bpmn (function): The function from the ntx_algo_to_bpmn_converter package that converts a Visio diagram to a BPMN diagram.
                Inputs:
                    VisioDiagram (bytearray): The Visio diagram to convert.
                    config (VisioConversionConfig): The configuration for the conversion.
                Outputs:
                    res_ba (bytearray): The converted BPMN diagram
        """
        super(
            task_name="convert_vision_to_bpmn",
            project_name="diagram_ingestion",
            output_type="artifact",
            data_type="bpmn",
            contains_file_output=True,
            file_suffix="bpmn",
        )

        self.visio_to_bpmn = visio_to_bpmn

    async def execute(self, VisioDiagram: str, **kwargs) -> str:
        """
        Executes the convert_vision_to_bpmn task.
        Args:
            VisioDiagram (str): The base64 encoded string representation of the Visio diagram.
            **kwargs: Additional optional parameters.
        Returns:
            str: The base64 encoded string representation of the converted BPMN diagram.
        """
        VisioDiagram: bytes = base64.b64decode(VisioDiagram)
        VisioDiagram: bytearray = bytearray(VisioDiagram)
        config = VisioConversionConfig()

        for value in kwargs.values():
            mappingType = BPMNMappingType(value)
            config.keyword_to_bpmn_mapping[value] = mappingType

        res_ba = self.visio_to_bpmn(VisioDiagram, config)

        base64_string = base64.b64encode(res_ba).decode("utf-8")
        return base64_string
```

Inversion of Control (IoC) Pattern

To make the system flexible and decoupled, the **Inversion of Control (IoC)** design pattern is being implemented. IoC moves the control of an object's behavior outside of the object itself, allowing the system to dictate the lifecycle, dependencies, and behaviors of objects. This is key for building a scalable and easily maintained architecture.

Benefits of IoC in This Context:

- **Loose Coupling:** Tasks can be swapped, changed, or extended without affecting the rest of the application. This makes introducing new tasks or modifying existing ones straightforward.
- **Flexibility:** The tasks interact with the Queue Consumer through IoC, meaning tasks no longer need to directly handle their dependencies. Instead, dependencies are injected or provided externally, improving modularity.

- **Maintainability:** By moving task-specific logic out of the queue system and into individual task files, it's easier to maintain and update code without impacting unrelated components.

Each project has an initiation file which injects all dependencies to project tasks:

```python
from ntx_algo_to_bpmn_converter import visio_to_bpmn, visio_to_bpmn_with_metrics

from .tasks import convert_vision_to_bpmn, tag_metrics


def init():
    return {
        "convert": convert_vision_to_bpmn(visio_to_bpmn),
        "tag_metrics": tag_metrics(visio_to_bpmn_with_metrics),
    }
```

Task Workflow and Worker Separation

Since each task is independent, workers consuming tasks from the queue can be configured to handle different tasks based on resource demand. For example:

- **Heavy Resource Tasks:** Tasks that consume significant resources (e.g., large data processing, complex machine learning inference) can be handled by dedicated workers with more processing power.
- **Lightweight Tasks:** Simpler tasks (e.g., file uploads, small data operations) can be processed by workers with fewer resources, allowing the system to optimize resource allocation.

This separation ensures that resource-heavy tasks from one project don't bottleneck the processing of lighter tasks from another project, or even from the same project.

Task Structure

Each task follows a standardized structure:

- **Task File:** Contains the logic specific to the task.
- **Task Dependencies:** Each task manages its own dependencies, using its own configuration file (e.g., a `pyproject.toml` for Python projects using Poetry).
- **Task Tests:** Each task includes its own unit tests, allowing for isolated testing.

Future Benefits of the New Architecture

- **Scalability:** As demand grows, more workers can be added, and projects can be hosted on separate machines or even containers, reducing the load on any single system.
- **Ease of Updates:** Since each project is independent, it can be updated or modified without impacting others.
- **Clear and Consistent Codebase:** The uniform structure across tasks and projects simplifies development and onboarding, making it easier for developers to work on multiple projects without needing to learn project-specific quirks.

This new architecture lays the foundation for a scalable, flexible, and maintainable environment in the GenAI Lab, allowing for efficient task processing, better resource management, and seamless project integration.