Министерство науки высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)

Факультет информационных технологий и программирования (ФИТиП)

Образовательная программа Программирование и интернет-технологии

Направление подготовки (специальность) 09.04.02 Информационные системы и технологии

ОТЧЕТ
О ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

по теме:

РЕАЛИЗАЦИЯ СЕРВИСА ДЛЯ ЗАПУСКА МАТЕМАТИЧЕСКИХ МЕТОДОВ

Обучающийся
Академическая группа M42051 — Степанов С. В.

Руководитель практики от профильной организации,
ООО «Цифровое проектирование»,
руководитель научной лаборатории — Ашихмин И. А.

Руководитель практики от университета,
Университет ИТМО,
факультет информационных технологий и программирования,
доцент — Зубок Д. А.

Практика пройдена с оценкой _____

Дата 16.04.2022

Санкт-Петербург
2022

Реферат

Данный отчет состоит из 16 страниц. Содержит в себе 3 иллюстрации, 4 листинга кода, 7 использованных источников. В данном отчете описывается проектирование и реализация сервиса, обеспечивающего преобразование данных предметной области в объекты математической библиотеки **nd_plan**, и имеющий возможность запускать методы данной библиотеки, учитывая особенности их выполнения. В рамках данной работы были проанализированы функциональные и нефункциональные требования, спроектирована архитектура сервиса и реализована запроектированная архитектура.

Содержание

Определения, обозначения и сокращения

В настоящем отчете применяют следующие термины с соответствующими определениями.

Генеральный план (генплан, ГП) в общем смысле — проектный документ, на основании которого осуществляется планировка, застройка, реконструкция и иные виды градостроительного освоения территорий. Основной частью генерального плана (также называемой собственно генеральным планом) является масштабное изображение, полученное методом графического наложения чертежа проектируемого объекта на топографический, инженерно-топографический или фотографический план территории.

Площадными объектами капитального строительства (ПО) в данной работе называются здания, строения, сооружения, а также объекты, строительство которых не завершено, за исключением некапитальных строений, сооружений и неотделимых улучшений земельного участка (замощение, покрытие и другие).

Введение

Автоматическое формирование генерального плана (ГП) площадного объекта капитального строительства является чрезвычайно сложной задачей, как в алгоритмическом, так и в технологическом плане. На генеральном плане помимо сооружений отражены внутриплощадочные проезды, различные трубопроводы, линии электропередач, технологические эстакады, пожарные гидранты и прочие объекты, необходимые для функционирования того или иного площадного объекта.

Для каждого объекта ГП заданы определенные требования к его размещению. Так как объектов много и они очень разнообразны по своей структуре, то использовать единый алгоритм для их расстановки невозможно. Поэтому процесс формирования ГП состоит из последовательного набора этапов, на каждом из которых рассчитывается местоположение определенного типа объектов.

Каждый этап представляет собой представляет собой применение одного или целого ряда алгоритмов. Алгоритмы находятся в отдельной математической библиотеке **nd_plan**. Библиотека **nd_plan** является внутренней разработкой, содержащей все алгоритмы, применяемые для решения поставленной задачи.

Из особенностей данной библиотеки стоит отметить, что в модели данных методов используются абстрактные структуры данных, отдаленные от предметной области понятной заказчику. Например, в рамках математических алгоритмов, мы будем рассматривать лишь граф, имеющий определенные свойства, а уже в терминах предметной области, данный граф станет совокупностью линий электропередач и трубопроводов.

К особенностям использования данной библиотеки стоит отнести типичные проблемы сложных алгоритмов: долгое время выполнения, высокую нагрузку на центральный процессор и высокий уровень потребления оперативной памяти.

Постановка задачи

Основной целью данной работы является спроектировать и реализовать сервис, который обеспечит преобразование данных предметной области в объекты математической библиотеки **nd_plan**, а также будет иметь возможность запускать методы данной библиотеки, учитывая особенности их выполнения.

Исходя из поставленной цели можно выделить следующие *задачи*:

1. Проанализировать функциональные и нефункциональные требования к сервису.
2. Опираясь на требования спроектировать архитектуру сервиса.
3. Реализовать спроектированную архитектуру.

Требования

Ниже перечислены те функциональные и нефункциональные требования для сервиса запуска математических методов библиотеки **nd_plan**.

Функциональные требования

В рамках функциональных требований к сервису можно выделить следующее:

1. Запуск математических методов библиотеки **nd_plan** должен осуществляться в отдельном процессе.
2. Запуск математических методов библиотеки **nd_plan** должен осуществляться через API.
3. API должен придерживаться концепции REST.
4. API должен предусматривать асинхронный запуск математического метода.
5. API должен использовать JSON в качестве обмена данных с клиентом.
6. API должен уметь обрабатывать внутреннюю модель расчетных данных системы **nd_plan_model**.
7. API должен предусматривать следующие функции для взаимодействия с расчетными задачами:
   - создание задачи.
   - запуск задачи.
   - получение статуса выполнения задачи.
   - отмена выполнения задачи.
   - получение результатов выполнения задачи.

Нефункциональные требования

Из нефункциональных требований выделим следующие:

1. Язык программирования **Python 3.8**
2. Развертывание сервиса осуществлять с помощью **Docker**

Архитектура сервиса

На диаграмме размещения системы(см. рис 1) расчетный сервис выделен голубым цветом.

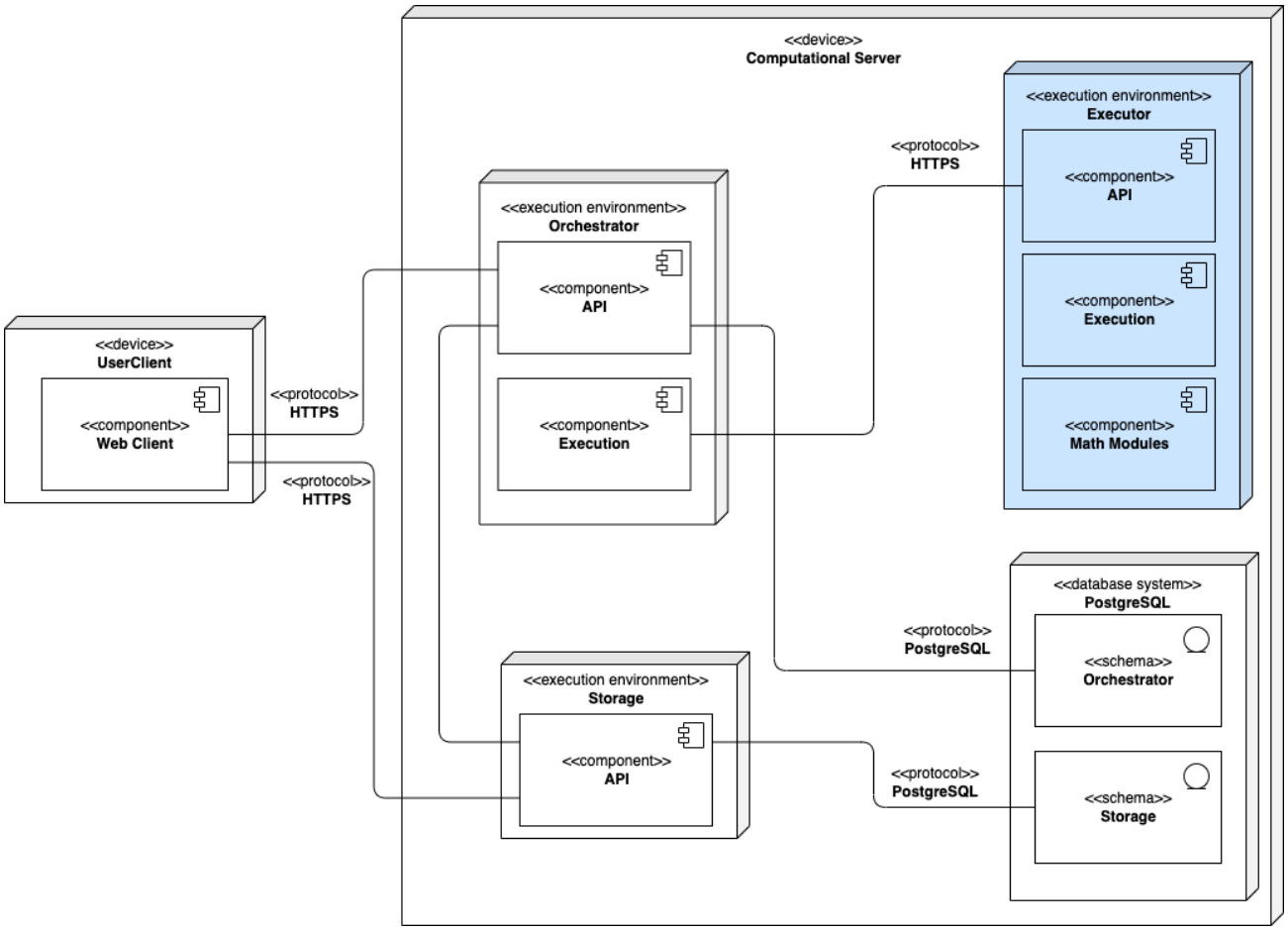

Рис. 1. Диаграмма размещения

Расчетный сервис представлен тремя компонентами: API, расчётным модулем и математической библиотекой **nd_plan**. В качестве цели данной работы является проектирование и разработка API и расчетного модуля, то поэтому только они и отражены на диаграммах ниже.

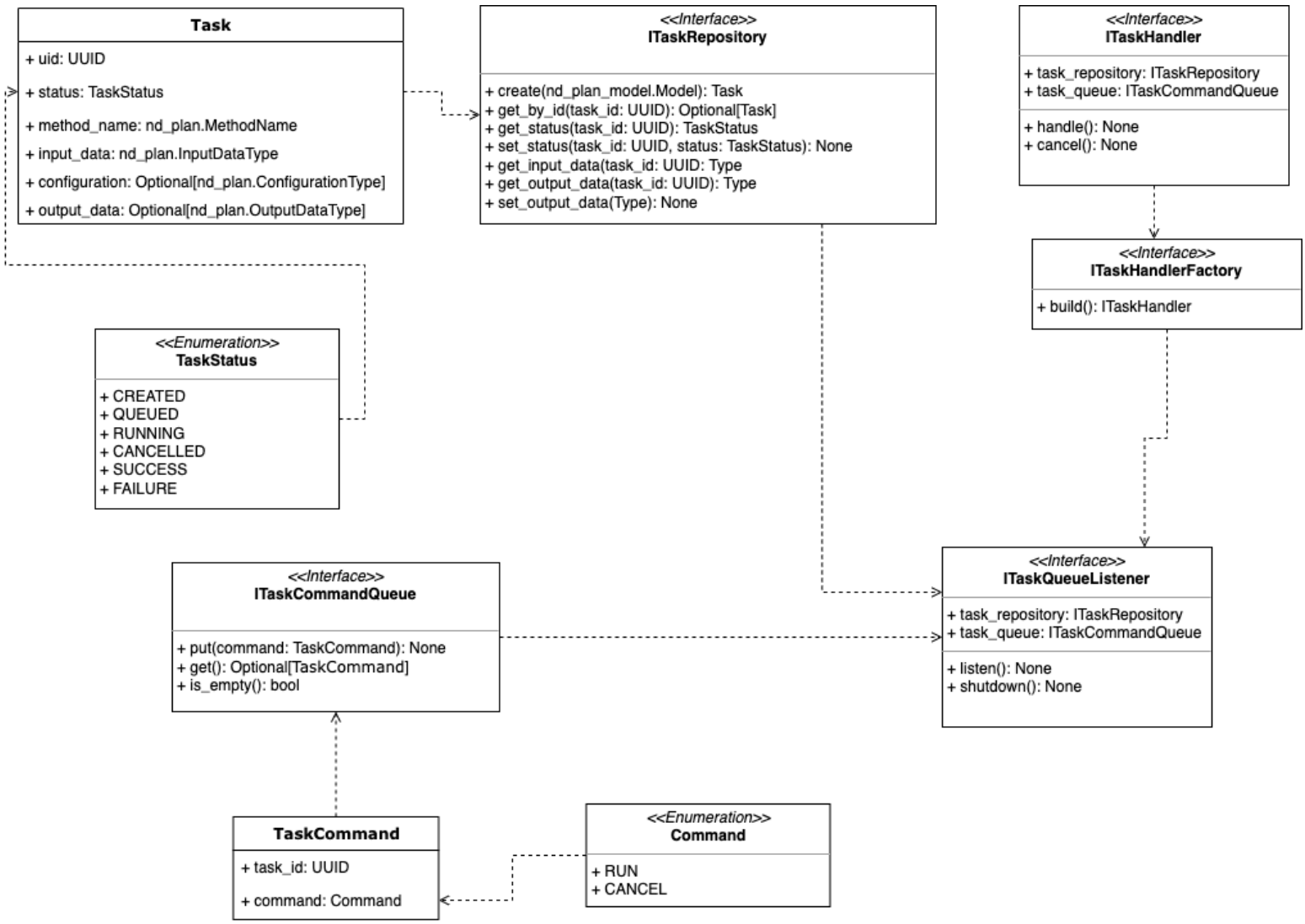

Рис. 2. Диаграмма классов расчетного модуля

Архитектура расчетного модуля представлена на диаграмме классов(см. рис 2). Основные классы и интерфейсы:

1. *Task* – расчетная задача.
2. *TaskStatus* – статус выполнения расчетной задачи.
3. *ITaskRepository* – получения данных по расчётной задаче.
4. *ITaskHandler* – запуск расчетных задач в отдельном процессе.
5. *ITaskCommandQueue* – очередь задач.
6. *ITaskQueueListener* – обработчик очереди задач, инициализация процесса расчета задач.

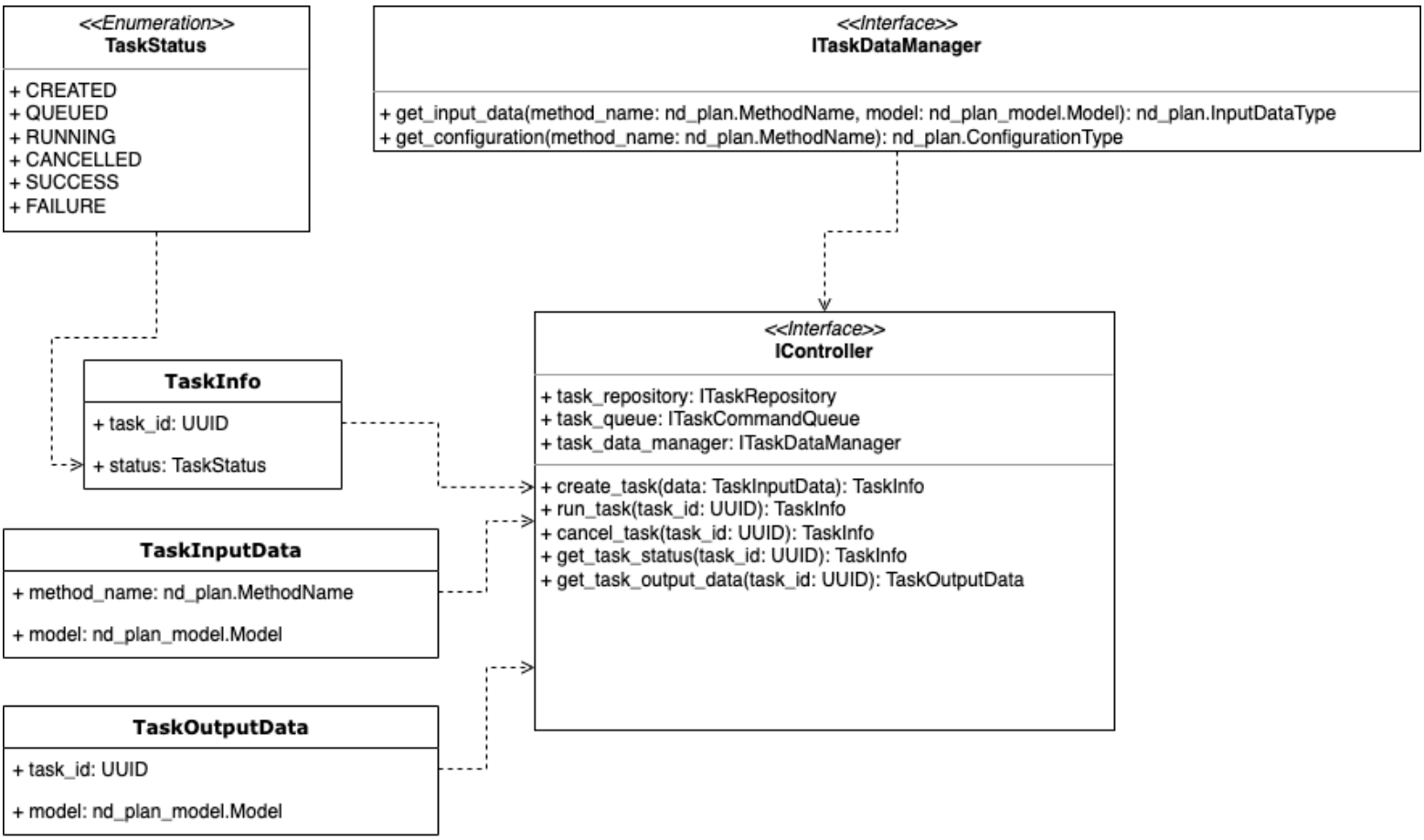

Рис. 3. Диаграмма классов API

Архитектура API представлена на диаграмме классов(см. рис 3). Основные классы и интерфейсы:

1. *TaskInfo* – информация о расчетной задаче.
2. *TaskStatus* – статус выполнения расчетной задачи.
3. *TaskInputData* – входные данные для создания расчетной задачи.
4. *TaskOutputData* – результат расчета.
5. *ITaskDataManager* – преобразование модели данных **nd_plan_model** в модель **nd_plan**.
6. *IController* – логика обработки запросов REST API.

Реализация

При реализации описанной архитектуры была получена следующая структура проекта.

```
/
└─ app
   └─ api
      ├─ __init__.py
      └─ controller.py
```

```
│   │
│   │
│   │   ├── exception_handlers.py
│   │   ├── middlewares.py
│   │   ├── model.py
│   │   ├── responses.py
│   │   ├── serialization.py
│   │   └── views.py
│   ├── data
│   │   ├── mappers
│   │   │   ├── services
│   │   │   │   └── services mappings implementations
│   │   │   ├── __init__.py
│   │   │   ├── registry.py
│   │   │   └── utils.py
│   │   ├── __init__.py
│   │   ├── data_manager.py
│   │   ├── features_registry.py
│   │   ├── repository.py
│   │   └── storage.py
│   ├── domain
│   │   ├── __init__.py
│   │   ├── listener.py
│   │   ├── model.py
│   │   ├── queue.py
│   │   └── repository.py
│   ├── execution
│   │   ├── __init__.py
│   │   ├── executor.py
│   │   ├── handler.py
│   │   ├── listener.py
│   │   ├── queue.py
│   │   └── storage.py
│   ├── __init__.py
│   ├── app.py
│   ├── exceptions.py
│   └── logger.py
├── Dockerfile
├── README.md
├── main.py
└── requirements.txt
```

Само приложение находится в директории **app**. Всего получилось 4 python-пакета, в которых и скрыта основная логика работы.

1. **api** – реализует API сервисы через HTTP. То есть endpoint-ы, модели запросов/ответов к серверу, а также взаимодействие с *ITaskRepository* и *ITaskDataManager*

2. **data** – в этом пакете находится реализация интерфейсов из **domain**.

3.

4. **domain** – пакет, в котором определены только интерфейсы взаимодействия между модулями программы. Именно здесь и реализована диаграмма классов (см. рис 2). Непосредственная реализация обозначенных интерфейсов находится в других пакетах.

5. **execution** – пакет, в котором вызывается код, отвечающий за запуск математических методов в отдельном процессе.

Примеры кода

```python
from app import startup_app
import uvicorn

if __name__ == '__main__':
    app = startup_app()
    uvicorn.run(app, host="0.0.0.0", port=8080)
```

Листинг 1. main.py

```
FROM python:3.8@sha256:4c4e6735f46e7727965d1523015874ab08f71377b3536b8789ee5742fc737059

WORKDIR /app

ENV LC_ALL C.UTF-8
ENV LANG C.UTF-8
ENV N_WORKERS 8

COPY requirements.txt .
RUN pip3 install --no-cache-dir -r requirements.txt

RUN pip3 check

COPY main.py .

COPY /app ./app

ENTRYPOINT /bin/bash -c "gunicorn run_app:app --workers=${N_WORKERS} --bind 0.0.0.0:8080 --worker-class
```

Листинг 2. Dockerfile

```python
from dataclasses import dataclass
from enum import Enum
from typing import Optional
from uuid import UUID

from dataclasses_json import dataclass_json

from nd_plan.interfaces import MethodName


class TaskStatus(Enum):
    CREATED = "CREATED"
    QUEUED = "QUEUED"
    RUNNING = "RUNNING"
    CANCELLED = "CANCELLED"
```

```
    SUCCESS = "SUCCESS"
    FAILURE = "FAILURE"


@dataclass_json
@dataclass
class Task:
    task_id: UUID
    status: TaskStatus
    method_name: nd_plan.MethodName
    input_data: nd_plan.InputDataType
    configuration: Optional[nd_plan.ConfigurationType] = None
    output_data: Optional[nd_plan.SolutionType] = None
```

Листинг 3. domain/model.py

```
import asyncio
import time
from dataclasses import dataclass
from typing import List
from uuid import UUID

from app.domain.listener import ITaskQueueListener
from app.domain.model import TaskStatus
from app.domain.queue import ITaskQueue
from app.domain.repository import ITaskRepository
from app.execution.executor import ExecutionConfig
from app.execution.executor import ExecutorExitCode
from app.execution.handler import CancellationReason
from app.execution.handler import TaskExecutionProcess
from app.execution.handler import handle_task
from app.execution.storage import build_executor_task_data_storage
from app.logger import get_logger

logger = get_logger(__name__)

__all__ = [
    "build_task_queue_listener",
    "ListenerConfig"
]


@dataclass
class ListenerConfig:
    execution_config: ExecutionConfig
    max_running_tasks: int = 1


class TaskQueueListener(ITaskQueueListener):
    def __init__(self, task_repository: ITaskRepository, task_queue: ITaskQueue, config: ListenerConfig)
        self._task_repository = task_repository
        self._task_queue = task_queue
        self._config = config
        self._executor_task_data_storage = build_executor_task_data_storage(
```

```
39              self._config.execution_config.executor_task_data_storage_config
40          )
41          self._running_tasks: List[TaskExecutionProcess] = []
42
43      async def listen(self):
44          logger.info("Task queue listener has started.")
45
46          while True:
47              if len(self._running_tasks) == 0 \
48                      or not self._task_queue.is_empty() and len(self._running_tasks) < self._config.max_
49                  await self._acquire_task()
50
51              still_running_tasks = []
52              for task in self._running_tasks:
53                  if self._task_queue.is_task_cancelled(task.task_id) and task.cancellation_reason is None
54                      task.cancellation_reason = CancellationReason.cancelled_by_user
55                      task.process.kill()
56                      logger.info(f'Task id={task.task_id} in process pid={task.process.pid} cancelled by
57
58                  current_unix_time = int(time.time())
59                  if current_unix_time - task.start_unix_time >= task.estimated_execution_time_sec:
60                      task.cancellation_reason = CancellationReason.cancelled_by_timeout_limit
61                      task.process.kill()
62                      logger.info(
63                          f'Task id={task.task_id} in process pid={task.process.pid} cancelled by time lim
64                          f'Estimated time is {task.estimated_execution_time_sec / 60} min. '
65                          f'Task run {(current_unix_time - task.start_unix_time) / 60} min.'
66                      )
67
68                  if task.process.is_alive():
69                      still_running_tasks.append(task)
70
71                  elif task.cancellation_reason == CancellationReason.cancelled_by_timeout_limit:
72                      await self._task_repository.set_task_status(task.task_id, TaskStatus.CANCELLED)
73                      logger.info(f'Compute of task id={task.task_id} cancelled by time limit.')
74                  elif task.cancellation_reason == CancellationReason.cancelled_by_user:
75                      await self._task_repository.set_task_status(task.task_id, TaskStatus.CANCELLED)
76                      logger.info(f'Compute of task id={task.task_id} cancelled by user.')
77
78                  elif task.process.exitcode == ExecutorExitCode.Ok:
79                      try:
80                          output_data = self._executor_task_data_storage.get_output_data(task.task_id, tas
81                          await self._task_repository.add_task_output_data(task.task_id, output_data)
82                          await self._task_repository.set_task_status(task.task_id, TaskStatus.SUCCESS)
83                          logger.info(f"Execution of task id=`{task.task_id}` is completed successfully.")
84                      except Exception as e:
85                          logger.info(f"Execution of task id=`{task.task_id}` is failed by I/O error: {e}.
86                          await self._task_repository.set_task_status(task.task_id, TaskStatus.FAILURE)
87                  elif task.process.exitcode == ExecutorExitCode.InternalErr:
88                      logger.info(f"Execution of task id=`{task.task_id}` is failed by internal error.")
89                      await self._task_repository.set_task_status(task.task_id, TaskStatus.FAILURE)
90                  else:
91                      logger.info(f"Execution of task id=`{task.task_id}` is failed.")
92                      await self._task_repository.set_task_status(task.task_id, TaskStatus.FAILURE)
93
```

```
94              self._running_tasks = still_running_tasks
95
96              await asyncio.sleep(2.0)
97
98      async def shutdown(self):
99          for task in self._running_tasks:
100             if task.process.is_alive():
101                 task.cancellation_reason = CancellationReason.cancelled_by_system_shutdown
102                 task.process.kill()
103                 await self._task_repository.set_task_status(task.task_id, TaskStatus.CANCELLED)
104                 logger.info(f'Task id={task.task_id} in process pid={task.process.pid} cancelled by syst
105
106     async def _acquire_task(self):
107         task_id: UUID = await self._task_queue.get()
108         logger.info(f"Received task id=`{task_id}`.")
109
110         is_task_cancelled = self._task_queue.is_task_cancelled(task_id)
111         if is_task_cancelled:
112             return
113
114         running_task = await handle_task(
115             self._task_repository,
116             self._executor_task_data_storage,
117             self._config.execution_config,
118             task_id
119         )
120         self._running_tasks.append(running_task)
121
122
123 def build_task_queue_listener(
124         task_repository: ITaskRepository,
125         task_queue: ITaskQueue,
126         config: ListenerConfig
127 ) -> ITaskQueueListener:
128     return TaskQueueListener(
129         task_repository,
130         task_queue,
131         config
132     )
```

Листинг 4. execution/listener.py

Заключение

В данной работе требовалось спроектировать и реализовать сервис, обеспечивающий преобразование данных предметной области в объекты математической библиотеки **nd_plan**, и имеющий возможность запускать методы данной библиотеки, учитывая особенности их выполнения. С учетом функциональных и нефункциональных требований была спроектирована архитектура системы, а так же эта архитектура была реализована.

Список литературы

1. Python 3.8 Documentation [Интернет ресурс]:
   URL:https://docs.python.org/3.8/(дата обращения: 28.01.22)

2. Glenford J. Myers, Corey Sandler, Tom Badgett "The Art of Software Testing 3rd Edition, 16 Dec 2011

3. Docker Reference[Интернет ресурс]:
   URL:https://docs.docker.com (дата обращения: 28.01.22)

4. Sphinx Python Documentation [Интернет ресурс]:
   URL:https://www.sphinx-doc.org/en/master/ (дата обращения: 28.01.22)

5. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition

6. Martin Robert C. "Clean Code: A Handbook of Agile Software Craftsmanship"Aug 1, 2008

7. Martin Fowler "UML Distilled: A Brief Guide to the Standard Object Modeling Language 3rd Edition"