МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
(НИЯУ МИФИ)

ИНСТИТУТ ЯДЕРНОЙ ФИЗИКИ И ТЕХНОЛОГИЙ
КАФЕДРА №40 «ФИЗИКА ЭЛЕМЕНТАРНЫХ ЧАСТИЦ»

УДК 004.42, 539.12

На правах рукописи

ПЛОТНИКОВ АРТЕМ ВАЛЕРЬЕВИЧ

СИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ ОБРАБОТКИ ДЛЯ ПРОГРАММНО-АППАРАТНОГО КОМПЛЕКСА SPD ONLINE FILTER

Направления подготовки:
09.04.04 «Программная инженерия»,
14.04.02 «Ядерные физика и технологии»
Диссертация на соискание степени магистра

Научный руководитель,
к.ф.-м.н. _____ Е. Ю. Солдатов

Научный консультант,
к.т.н. _____ Д. А. Олейник

Москва 2025

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

СИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ ОБРАБОТКИ ДЛЯ ПРОГРАММНО-АППАРАТНОГО КОМПЛЕКСА SPD ONLINE FILTER

Студент	_____ А. В. Плотников
Научный руководитель, д.ф.-м.н.	_____ Е. Ю. Солдатов
Научный консультант, к.т.н.	_____ Д. А. Олейник
Рецензент, к.ф.-м.н.	_____ А. С. Жемчугов
Рецензент, к.ф.-м.н.	_____ Л. Г. Афанасьев
Секретарь ГЭК, к.ф.-м.н.	_____ Е. Ю. Солдатов
Зав. каф. №40, д.ф.-м.н., проф.	_____ М. Д. Скорохватов
Рук. учеб. прог., к.ф.-м.н.	_____ Е. Ю. Солдатов

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Эксперименты в сфере физики высоких энергий всегда характеризовались большими объёмами собираемых и обрабатываемых данных. При столкновениях пучков частиц на коллайдерах регистрируется до нескольких миллионов событий в секунду. Хотя в настоящий момент сверхбыстрые детекторные системы позволяют регистрировать подобные события с достаточно высокой точностью, всё также остаётся необходимость эффективно управлять большим потоком данных с детекторов.

Учитывая ограниченные ресурсы на запись и последующий анализ данных, неизбежно встаёт задача быстрой фильтрации фоновых и малозначимых событий. В этом случае важной оказывается своевременная оценка физической ценности сигналов. Традиционно в таких случаях применяют аппаратные триггерные системы, но в некоторых ситуациях подобные решения не соответствуют особенностям эксперимента. В частности, необходимой может оказаться частичная реконструкция событий, для которой предполагается создание специализированных вычислительных систем, обрабатывающих данные в режиме реального времени.

Именно здесь на первый план выходит концепция программных триггеров, опирающихся на вычислительные мощности многопроцессорных серверов. Плюсом данного подхода является ещё и то, что он даёт возможность динамически менять критерии отбора в зависимости от целей текущего исследования, оперативно адаптируясь к изменяющимся условиям.

В данной работе будет рассмотрена подобная система для разрабатываемого эксперимента SPD на коллайдере NICA (Дубна, Россия) [1]. А также будет описан этап разработки прототипа одной из подсистем – системы управления процессами обработки.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы: разработка прототипа системы управления процессами обработки (workflow management system - WfMS), позволяющего параллельно управлять большим количеством процессов обработки данных и контролировать статусы выполнения цепочек обработки для физического эксперимента SPD.

Задачи:

1) ознакомиться с экспериментом SPD и особенностями его экспериментальной установки;
2) изучить основные принципы работы SPD Online Filter;
3) рассмотреть возможность использования готовых решений в рамках создания системы управления процессами обработки;
4) ознакомиться с проектом системы workflow management system (WfMS): определить основные сервисы WfMS, выявить их основной функционал и принципы взаимодействия с другими системами SPD Online Filter;
5) разработать API для работы с базой данных;
6) разработать сервисы WfMS с использованием предпочтительного стека:
   (а) создать сервис для взаимодействия с оператором обработки:
      i. авторизация пользователей;
      ii. обработка CWL-шаблонов;
      iii. вывод информации о заданиях и шаблонах;
      iv. работа со статусами шаблонов.
   (б) реализовать сервис генерации заданий:
      i. получение датасетов из data management system (DMS);
      ii. создание заданий по шаблонам для последовательностей.
   (в) разработать сервис для опроса DMS:
      i. опрос DMS о готовности входных датасетов для заданий;
      ii. создание выходных датасетов и датасетов логов в DMS;

iii. отправление заданий на обработку в workload management system (WMS).

(г) создать сервис для работы с WMS:

i. периодический опрос WMS для отслеживания статусов заданий;

ii. закрытие датасетов после завершения заданий;

iii. удаление входного и промежуточных датасетов после выполнения всех заданий в цепочке.

7) протестировать разработанные сервисы и совместную работу систем, исправить баги.

1 SPD — ЭКСПЕРИМЕНТ ПО ИЗУЧЕНИЮ СПИНОВОЙ СТРУКТУРЫ НУКЛОНОВ

Несмотря на значительные достижения квантовой хромодинамики при описании взаимодействия кварков и глюонов, остаётся неразрешённым вопрос о том, почему нуклоны такие, какими мы их наблюдаем. Понимание структуры и фундаментальных свойств нуклона на основе динамики его составляющих - кварков и глюонов, остаётся одной из основных нерешенных проблем квантовой хромодинамики.

Модель кварков успешно предсказывает большинство основных характеристик адронов, таких как заряд, чётность, изоспин и свойства симметрии, а также взаимосвязи между ними. Некоторая динамика взаимодействий частиц также может быть качественно понята в рамках этой модели. Однако она не даёт объяснения спиновым свойствам адронов с точки зрения их составляющих. Со времен «спинового кризиса», обозначенного в 1987 году, проблема спиновой структуры нуклонов остается загадкой современной физики высоких энергий. Одной из главных задач, привлекающих огромные теоретические и экспериментальные усилия, является вопрос о том, как спин нуклона формируется из спинов и орбитальных моментов его составных частей - валентных кварков, "морских" кварков и глюонов. Полное объяснение можно получить через так называемые функции распределения партонов, зависящие от их поперечного импульса.

Эксперименты по поляризованному глубоконеупругому рассеянию (CERN, DESY, JLab, SLAC) и высокоэнергетические поляризованные протон-протонные столкновения (RHIC в BNL) являются основным источником информации о спин-зависимых структурных функциях нуклонов. Тем не менее наши знания о внутренней структуре нуклона все еще ограничены. Особенно это касается глюонного вклада [2].

1.1 SPIN PHYSICS DETECTOR (SPD)

Экспериментальная установка SPD, которую планируется разместить в одной из двух точек пересечения пучков коллайдера NICA (рис. 1.1), предназначена для всестороннего изучения спиновой структуры протона и дейтрона в поляризованных p-p, d-d и p-d-столкновениях. Основное внимание будет уделено изучению их поляризованной глюонной компоненты в реакциях инклюзивного рождения чармониев, открытого чарма и прямых фотонов, а также прочих спинзависимых явлений в столкновениях поляризованных пучков протонов и дейтронов с энергией в системе центра масс до 27 ГэВ и светимостью до 10^{32} см$^{-2}$с$^{-1}$.

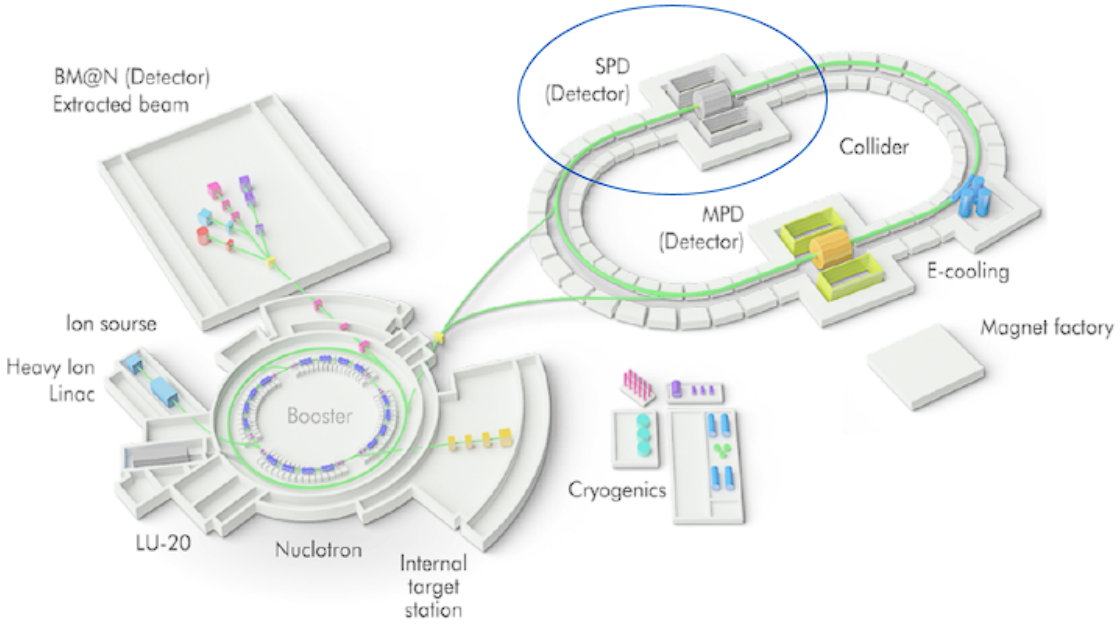


Рисунок 1.1 — Комплекс NICA [3]

Эксперимент SPD планирует получить данные о спиновых асимметриях для анализа корреляций между направлением спина протона (дейтрона), его импульсом и направлением спина, связанным с продольным и поперечным импульсами глюонов внутри протона (дейтрона). В начальной фазе работы установки, до достижения проектных светимости и энергии столкновения, основное внимание планируется уделить изучению спиновых эффектов в упругих p-p и d-d рассеяниях, поиску мультипартонных корреляций и новых связанных состояний, исследованию рождения чарма у порога, изучению поляризаций гиперонов и т.д [4].

Новые данные о спиновой структуре протона и дейтрона, которые будут получены на установке SPD, приблизят нас к пониманию фундаментальных основ квантовой хромодинамики.

Исследования на установке SPD с использованием поляризованных протонных пучков позволят заполнить пробелы между измерениями на низких энергиях на ускорителях ANKE-COSY и SATURNE и измерениями на высоких энергиях на коллайдере RHIC и планируемых на LHC экспериментах с неподвижной мишенью. Важно отметить, что возможность работать с поляризованными пучками дейтронов в этом диапазоне энергий является уникальной особенностью данного эксперимента.

1.1.1 ЭКСПЕРИМЕНТАЛЬНАЯ УСТАНОВКА SPD

Экспериментальная установка, изображенная на рисунке 1.2, будет обладать широким геометрическим охватом, приблизительно равным 4π, продвинутой системой для восстановления треков и вершин, а также разнообразными возможностями по идентификации частиц, основанными на передовых технологиях.

Кремниевый вершинный детектор (VD) обеспечит координатное разрешение при восстановлении вершины лучше 100 микрон, что критически важно для точной реконструкции вторичных вершин распада D-мезонов. Трековая система, использующая строу-трубки (ST) в соленоидальном магнитном поле до 1 Тл, позволит определить поперечные импульсы вторичных частиц с точностью до 2%.

Времяпролетная система (PID) с временным разрешением 60 пс обеспечит разделение пионов, каонов и протонов в широком кинематическом диапазоне. Использование черенковских детекторов на основе аэрогеля позволит расширить этот диапазон. Для регистрации фотонов будет использован электромагнитный калориметр типа «шашлык» (ECal). Для уменьшения эффектов многократного рассеяния и конверсии фотонов минимизировано количество вещества во внутренней части установки.

Мюонная (пробежная) система (RS) предназначена для идентификации мюонов. Она также может служить в качестве грубого адронного калориметра.

Пара пучковых счетчиков (BBC) и калориметров нулевого угла (ZDC) будут отвечать за локальную поляриметрию и контроль светимости.

Высокая частота столкновений (до 4 МГц) и большое количество каналов электроники определяют высокие требования к системе сбора данных, онлайн-мониторингу, компьютерной системе последующей обработки данных и программному обеспечению для реконструкции и анализа данных.

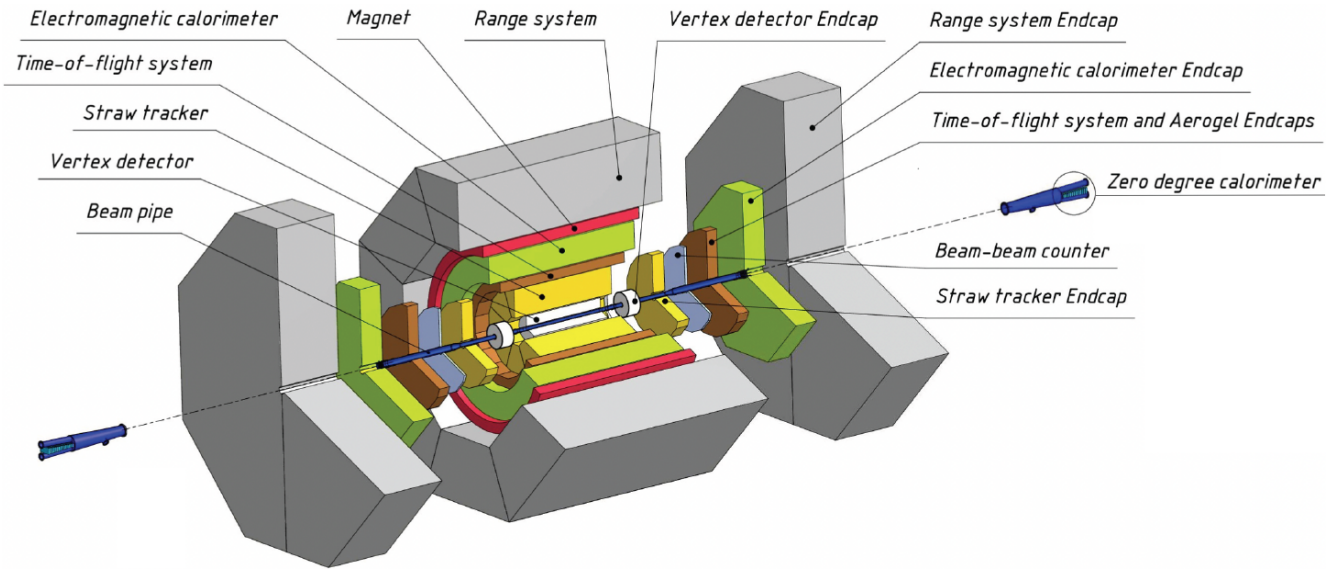


Рисунок 1.2 — Экспериментальная установка SPD [5]

1.1.2 ОБЪЁМ ДАННЫХ ЭКСПЕРИМЕНТА

В физике высоких энергий события, регистрируемые детектором, обрабатываются независимо, поэтому их можно считать наименьшей единицей данных. А значит, зная размер одного события, сложность обработки событий и ожидаемое количество событий для обработки, можно оценить требования к вычислительной системе для эксперимента.

Исходя из приблизительных оценок на SPD будет приходить 20 ГБ/с (или 200 ПБ/год) «необработанных» данных, что соответствует $3 \cdot 10^{13}$ событий/год.

Сбор, обработка и хранение такого объема данных представляет собой серьезную проблему для вычислительной инфраструктуры эксперимента и требует разработки новых методов и подходов для реконструкции событий, моделирования и физического анализа данных с использованием высокопроизводительных и распределенных вычислений.

1.1.3 ОСОБЕННОСТИ СБОРА ДАННЫХ

Системы сбора данных в большинстве подобных экспериментов используют триггеры (рис. 1.3) для отбора и записи только тех событий, которые являются интересными для дальнейшего анализа. Триггер – управляющий сигнал, по которому осуществляется запись с детекторных систем.

Триггеры могут быть настроены на различные критерии, например, на обнаружение определенных типов частиц, особенных шаблонов распределения энергии, наличие определенных вершин или корреляций между различными частицами. Такие системы триггеров позволяют экономить ресурсы хранения и обработки данных, фокусируясь только на событиях, которые соответствуют определенным критериям или интересам исследования.

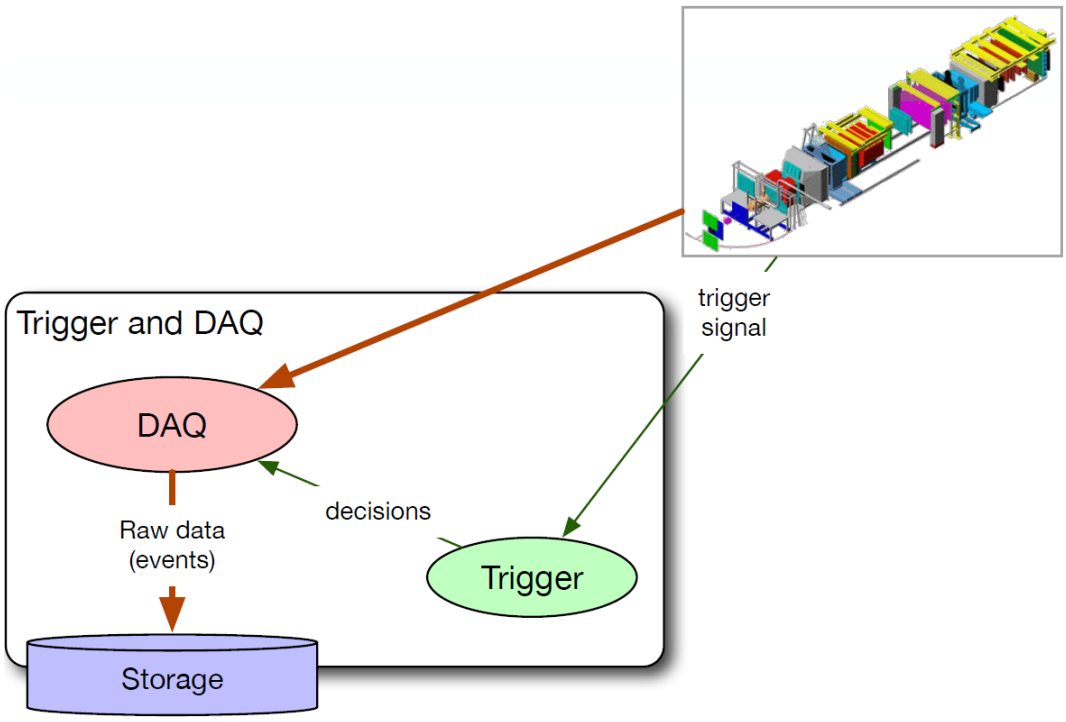


Рисунок 1.3 — Триггерная система сбора данных [6]

Однако ввиду сложности и широты изучаемых процессов (выбор физического сигнала требует реконструкции импульса и вершины) невозможно построить критерии отбора данных на аппаратном уровне, поэтому для минимизации возможных систематических эффектов SPD будет оснащен бестриггерной системой сбора данных (рис. 1.4).

Система сбора данных предоставляет данные, организованные по временным интервалам и разбитые на файлы разумного размера (несколько ГБ). При

таком подходе отдельные файлы могут обрабатываться параллельно, что повышает эффективность работы системы. Каждый файл может быть обработан независимо, что упрощает процесс.

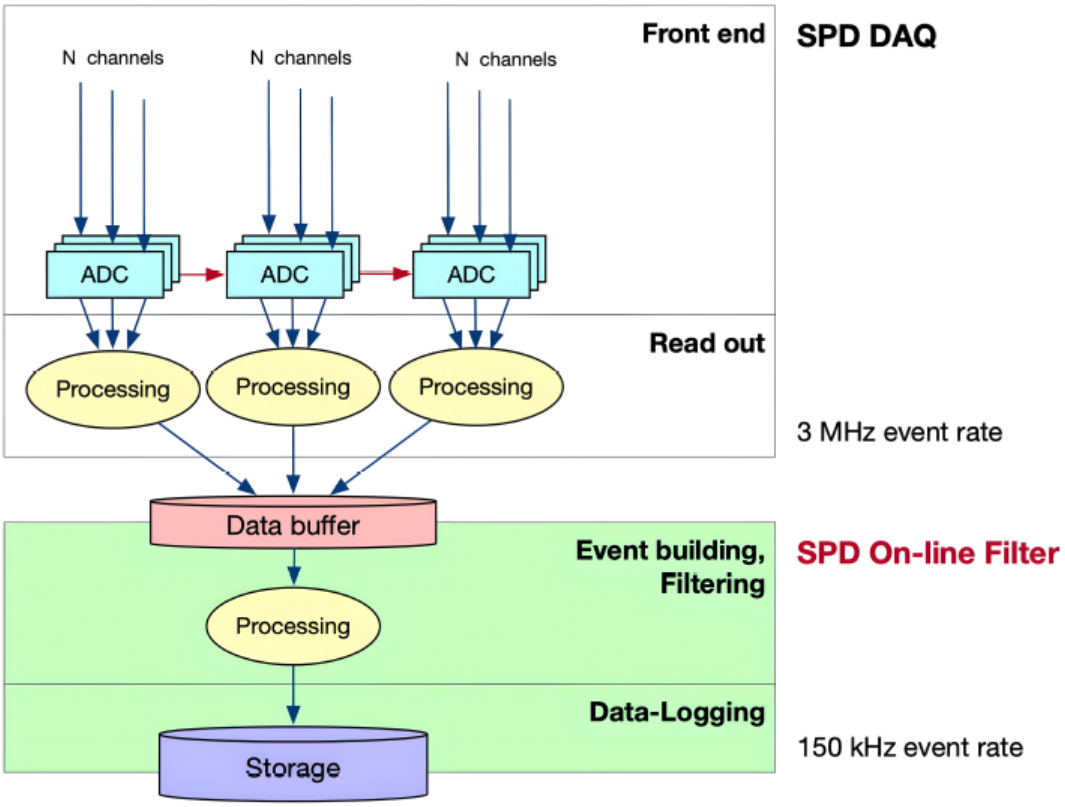


Рисунок 1.4 — Бестриггерная система SPD [6]

Такой подход особенно полезен, когда нет необходимости в обмене информацией между файлами на этапе обработки, и результаты обработки одного файла могут быть использованы как входные данные для следующего этапа. Это позволяет эффективно использовать вычислительные ресурсы и ускоряет процесс анализа данных в целом.

2 SPD ONLINE FILTER

2.1 ПРИНЦИП РАБОТЫ SPD ONLINE FILTER

SPD Online Filter — это высокопроизводительная вычислительная система для высокопропускной первичной обработки данных эксперимента SPD.

Эта вычислительная система должна осуществлять следующее преобразование данных: идентифицировать физические события во временных интервалах; реорганизовать данные в событийно-ориентированном формате; отфильтровать события, оставляя только интересные с точки зрения эксперимента; согласовывать выходные данные, объединять события в файлы и файлы в наборы данных для будущей обработки.

Концепция базовой обработки:

1) Реконструкция треков и связывание их с вершинами;
2) Связывание срабатываний электромагнитного калориметра и пробежной системы с каждой вершиной по времени;
3) Определение несвязанных срабатываний детектора;
4) Объединение с необработанными сигналами от других субдетекторов;
5) Формирование блоков данных и сохранение частично реконструированных событий.

Данные объединяются в наборы файлов в зависимости от стадии их обработки. Каждый файл в наборе может быть обработан независимо, однако наборы обрабатываются в некоторой заданной последовательности. Каждый файл можно обработать за некоторое разумное время на ограниченном вычислительном ресурсе (вычислительном узле).

Разбиение данных на более мелкие части позволяет управлять объемом обработки и минимизировать последствия при возникновении ошибок при обработке больших объемов данных. Работа с отдельными частями позволяет изо-

лировать ошибки, которые могут возникнуть в процессе обработки, и снизить их воздействие на всю систему.

Такой подход также обеспечивает масштабируемость системы: добавление дополнительных вычислительных ресурсов позволяет обрабатывать еще большие объемы данных без значительного изменения общей архитектуры системы. Это особенно полезно для систем, работающих с высокочастотными данными или при необходимости быстрой обработки информации в режиме реального времени.

SPD Online Filter представляет из себя совокупность прикладного и промежуточного программных обеспечений, а также высокопроизводительного вычислительного кластера.

По результатам анализа первичных требований к системе были выделены основные компоненты промежуточного программного обеспечения:

1) Workflow management system – система управления процессами обработки – создание формального описания цепочек обработки, формирование и контроль исполнения этапов обработки данных;
2) Data management system – система управления данными – регистрация, каталогизация, контроль целостности файлов и датасетов;
3) Workload management system & pilot – система управления нагрузкой с подсистемой pilot – реализация этапов обработки (формирование задач и отправка их pilot).

Pilot – приложение, работающее на вычислительном узле и исполняющее задачи.

Архитектура системы SPD Online Filter приведена на рис. 2.1.

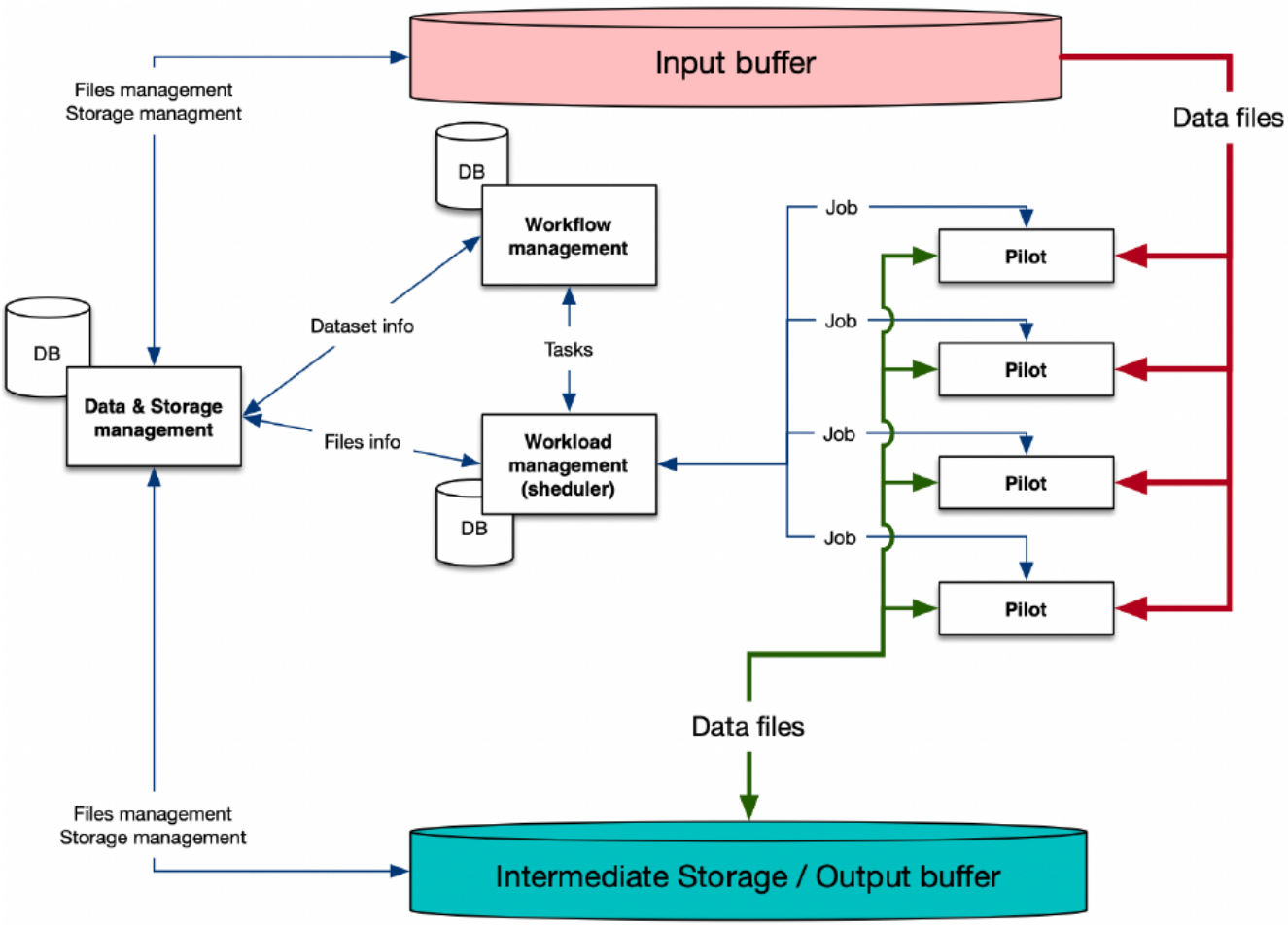


Рисунок 2.1 — Архитектура SPD Online Filter [7]

2.2 ВХОДНЫЕ ДАННЫЕ

Входные данные, приходят из системы сбора данных (DAQ) во входной буфер.

- Период набора – ассоциирован с физической задачей. Состоит из набора ранов. Имеет дату начала, дату окончания.
- Ран – интервал, когда условия эксперимента неизменны (калибровки, например). Измеряется часами. Состоит из фреймов.
- Фрейм – нумерованный блок данных с детектора. Может содержать информацию о множестве событий. Продолжительность секунды.
- Датасет – логическое объединение файлов в наборы для обработки. Датасет является единицей обработки для одного задания.
- Файл – количество данных для обработки одной задачей. [8]

3 СИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ ОБРАБОТКИ (WFMS)

3.1 ВОЗМОЖНОСТЬ ПРИМЕНЕНИЯ ГОТОВЫХ РЕШЕНИЙ

Рассматривая подобные эксперименты в сфере физики высоких энергий, становится ясно, что обойтись без создания собственной системы не получится.

В экспериментах Большого адронного коллайдера (LHC), таких как ATLAS и CMS, используются масштабные системы управления распределённой обработкой данных: PanDA (Production and Distributed Analysis) [9] и WMAgent/CRAB (CMS Remote Analysis Builder) [10].

Эти решения разрабатывались специально для распределённых сред, функционирующих в рамках WLCG (Worldwide LHC Computing Grid), и ориентированы на задачи офлайн обработки и анализа данных после прохождения событий через триггерные системы.

PanDA ориентирована на работу с высокоуровневыми задачами анализа, использующими большие наборы предварительно обработанных данных. Обработка организована в виде DAG-цепочек, но её адаптация под реальное время и нераспределённую среду, представляется чрезмерно громоздкой.

В CMS используется связка систем WMAgent, управляющая задачами обработки и реконструкции, и пользовательского интерфейса CRAB, позволяющего исследователям запускать анализ данных в Grid. Задания распределяются по пилотным агентам на основе HTCondor и GlideinWMS.

Несмотря на зрелость и масштабируемость указанных решений, они не соответствуют архитектуре и задачам SPD, по следующим причинам:

SPD предполагает работу в рамках локального или централизованного вычислительного кластера, не связанного с WLCG. Использование систем, ори-

ентированных на десятки центров обработки данных, в таком контексте приводит к избыточной сложности.

PanDA и WMAgent предназначены для офлайн обработки с высокой задержкой. В отличие от них, SPD Online Filter обрабатывает данные по мере поступления, в онлайн режиме. Это делает неподходящим использование брокеров, ожидающих долгого согласования данных и распределения задач.

Оба решения содержат модули, связанные с Grid-авторизацией, сложным контролем публикации, управлением ресурсами, которые в контексте SPD избыточны. Их интеграция привела бы к неоправданному усложнению поддержки и сопровождения.

В экспериментах ATLAS и CMS ключевую роль в отборе данных в режиме реального времени играет система High-Level Trigger (HLT). Это программный триггер, работающий после аппаратного триггера первого уровня (L1) и обеспечивающий более точную фильтрацию событий перед их записью в долговременное хранилище.

HLT выполняет: частичную или полную реконструкцию событий (треки, вершины, кластеры), сокращая поток данных со $\tilde{1}00$ кГц до $\tilde{1}$ кГц.

Он реализуется в виде высокопроизводительного кластера (Event Filter Farm), на котором события обрабатываются C++/Python-фреймворками (AthenaHLT в ATLAS, CMSSW в CMS), цепочки обработки задаются заранее в виде жёстко определённых последовательностей (trigger paths), а изменение логики отбора требует пересборки конфигурации и перекомпиляции фреймворка [11].

HLT ориентирован на работу с ROI (regions-of-interest) и использует специализированные алгоритмы триггерного уровня, максимально оптимизированные по скорости. Управление потоком данных, маршрутизация и буферизация глубоко интегрированы в архитектуру детектора и DAQ.

Прямое применение подхода HLT в эксперименте SPD невозможно и нецелесообразно по ряду причин:

1) В SPD концептуально отсутствует аппаратный триггер, что означает, что программная система фильтрации должна обрабатывать весь входной поток данных ($\tilde{2}0$ ГБ/с), а не выборку, отфильтрованную L1, как в CMS/ATLAS. HLT без L1 просто не справится с такой нагрузкой;

2) В HLT цепочки отбора и реконструкции жёстко заданы специалистами и не подлежат динамической модификации во время работы;
3) В HLT критически важна низкоуровневая интеграция с DAQ, чтение из буферов, прямой доступ к событиям и каналам. В SPD применяется модель, в которой данные поступают во внешний входной буфер в виде файлов, и обрабатываются уже в событийно-независимом виде.
4) HLT не сохраняет промежуточные результаты реконструкции для повторной обработки.

Интересным решением является продукт с открытым исходным кодом — Apache Airflow [12], который может использоваться в качестве оркестратора для разработки, планирования и мониторинга сложных рабочих процессов.

Работа Apache Airflow основана на создании направленного ациклического графа (DAG), описывающего процессы обработки данных и определяющего связи и правила выполнения задач.

Планировщик обрабатывает DAG, а затем запускает экземпляры задач после выполнения их зависимостей. В фоновом режиме планировщик запускает подпроцесс, который следит и синхронизирует все DAG в указанном каталоге.

WebServer отвечает за отображение веб-интерфейса и аутентификацию пользователей.

Apache Airflow обладает множеством преимуществ и широким функционалом, однако для обработки задач в реальном времени его использование представляется неоптимальным [13].

SPD Online Filter как раз работает с данными по мере их поступления. Помимо всего прочего, нужно учитывать, что WfMS должна взаимодействовать с другими системами, отправлять им запросы, получать ответы и на их основе принимать дальнейшие решения. С помощью Apache Airflow довольно тяжело организовать такую систему.

3.2 ПРИНЦИП РАБОТЫ РАЗРАБАТЫВАЕМОЙ WFMS

После попадания в SPD Online Filter, данные регистрируются в системе управления данными (Data Management System — DMS). Затем они подготавливаются к последующей обработке при помощи системы управления процессами обработки (Workflow Management System — WfMS).

WfMS принимает эти зарегистрированные данные и сопоставляет их с определенным шаблоном цепочки обработки, который предоставлен оператором обработки данных. Каждый этап такой цепочки обработки генерирует соответствующие задания для выполнения. Эти задания затем направляются в систему управления нагрузкой (Workload Management System - WMS), где они распределяются на вычислительные ресурсы для выполнения. После этого WfMS контролирует процесс выполнения задания путем периодического опроса WMS об их статусах и принимает решение об изменении приоритетов заданий или об их отмене.

Такой подход позволяет эффективно организовать и контролировать обработку данных по заданной цепочке этапов, что обеспечивает систему планирования и контроля рабочих процессов, повышающую оперативность обработки данных и предоставляющую возможность мониторинга за выполнением обработки.

Система управления процессами обработки (WfMS) имеет ряд функциональных требований, чтобы эффективно организовать и контролировать последовательности обработки данных:

1) Организация последовательностей обработки данных:
   - Последовательность: состоит из логически связанных шагов обработки данных, где результат одного шага является входными данными для следующего.
   - Шаг обработки: представляет собой действие, которое выполняется над набором данных. Эти шаги могут быть типа "map"(однотипная обработка для каждого элемента данных) или "merge"(объединение гомогенных данных).
   - Описание шаблона: выражается с помощью common workflow language (CWL) [14].
2) Формирование запроса на обработку данных:
   - Для каждого шага обработки определяются необходимые параметры для формирования задания.
3) Выполнение запроса на обработку данных:
   - Создание заданий для каждого шага обработки данных на основе сформированных параметров.

- Отправка созданных заданий в систему управления нагрузкой (WMS) для выполнения.
- Осуществление контроля выполнения обработки данных путем опроса системы управления нагрузкой.

Эти функциональные требования позволят системе WfMS эффективно управлять процессами обработки, обеспечивая оптимизацию обработки данных, контроль ошибок и достижение заданных целей обработки.

В основном процессе обработки данных в SPD Online Filter можно выделить следующие шаги:

1) Декодирование данных;
2) Выявление событий и реструктуризация данных;
3) Фильтрация выявленных событий;
4) Верификация данных;
5) Объединение файлов (уменьшение количества файлов, увеличение размера файлов);

Такой набор последовательных шагов обработки будет называться цепочкой обработки. Каждый из шагов в такой цепочке может быть определен шаблоном (рис. 3.1).

```
cwlVersion: v1.2
label: Decoding of data
class: Workflow

inputs:
  dataset_name: string
  processing_program: string
  processing_program_version: string
  cable_map: File
  input_params: File
steps:
  decoding:
    run:
      class: CommandLineTool
      baseCommand: echo

      inputs:
        dataset_name:
          type: string
        processing_program:
          type: string
        processing_program_version:
          type: string
        cable_map:
          type: File
        input_params:
          type: File
      outputs:
        output_dataset:
          type: File
        log_dataset:
          type: File

    in:
      dataset_name: dataset_name
      processing_program: processing_program
      processing_program_version: processing_program_version
      cable_map: cable_map
      input_params: input_params

    out: [output_dataset, log_dataset]

outputs:
  output_dataset:
    type: File
    outputSource: decoding/output_dataset
  log_dataset:
    type: File
    outputSource: decoding/log_dataset
```

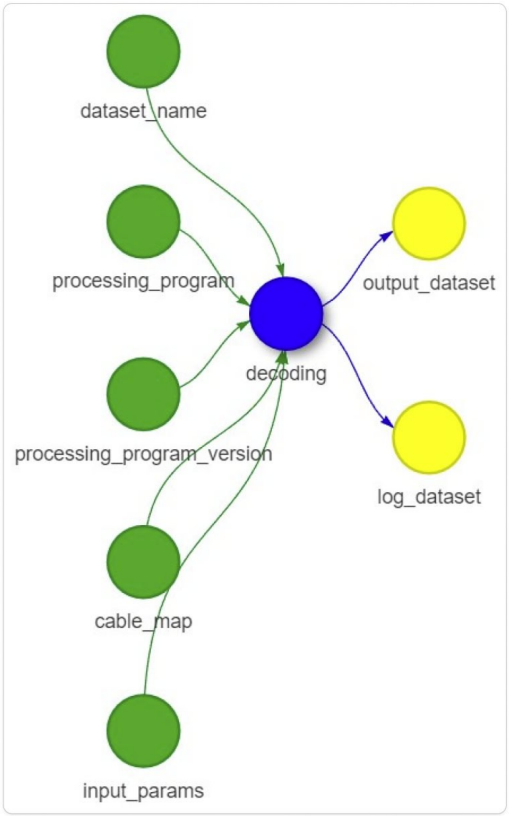


Рисунок 3.1 — Пример CWL-шаблона

Во время работы системы предполагается создание промежуточных данных, которые будут удаляться после получения выходного набора данных.

В результате проектирования были определены основные сервисы системы WfMS:

1) Сервис для взаимодействия с оператором обработки данных:
   (а) авторизация пользователей;
   (б) обработка CWL-шаблонов;
   (в) вывод информации о заданиях и шаблонах;
   (г) работа со статусами шаблонов;
2) Сервис для опроса DMS:
   (а) получение информации о готовности входных датасетов;
   (б) формирование выходных датасетов;
   (в) отправление заданий на обработку;
3) Сервис генерации заданий:
   (а) определение последовательностей обработки данных;
   (б) создание заданий по шаблонам для последовательностей.
4) Сервис для работы с WMS:
   (а) периодический опрос для отслеживания статуса;
   (б) завершение заданий с последующим закрытием датасетов;
   (в) удаление входных и промежуточных датасетов после обработки всех заданий цепочки;
   (г) изменение приоритетов заданий;
   (д) отмена заданий.

В данной системе применяется микросервисная архитектура. В рамках этой архитектуры приложение разбивается на отдельные сервисы, которые могут быть развернуты и масштабированы независимо друг от друга. Сервисы взаимодействуют между собой через API-интерфейсы, что позволяет каждому из них функционировать автономно. В отличие от монолитной архитектуры, микросервисы позволяют быстрее внедрять новые функции и вносить изменения, не требуя переработки большого объема существующего кода.

Архитектура WfMS приведена на рис. 3.2.

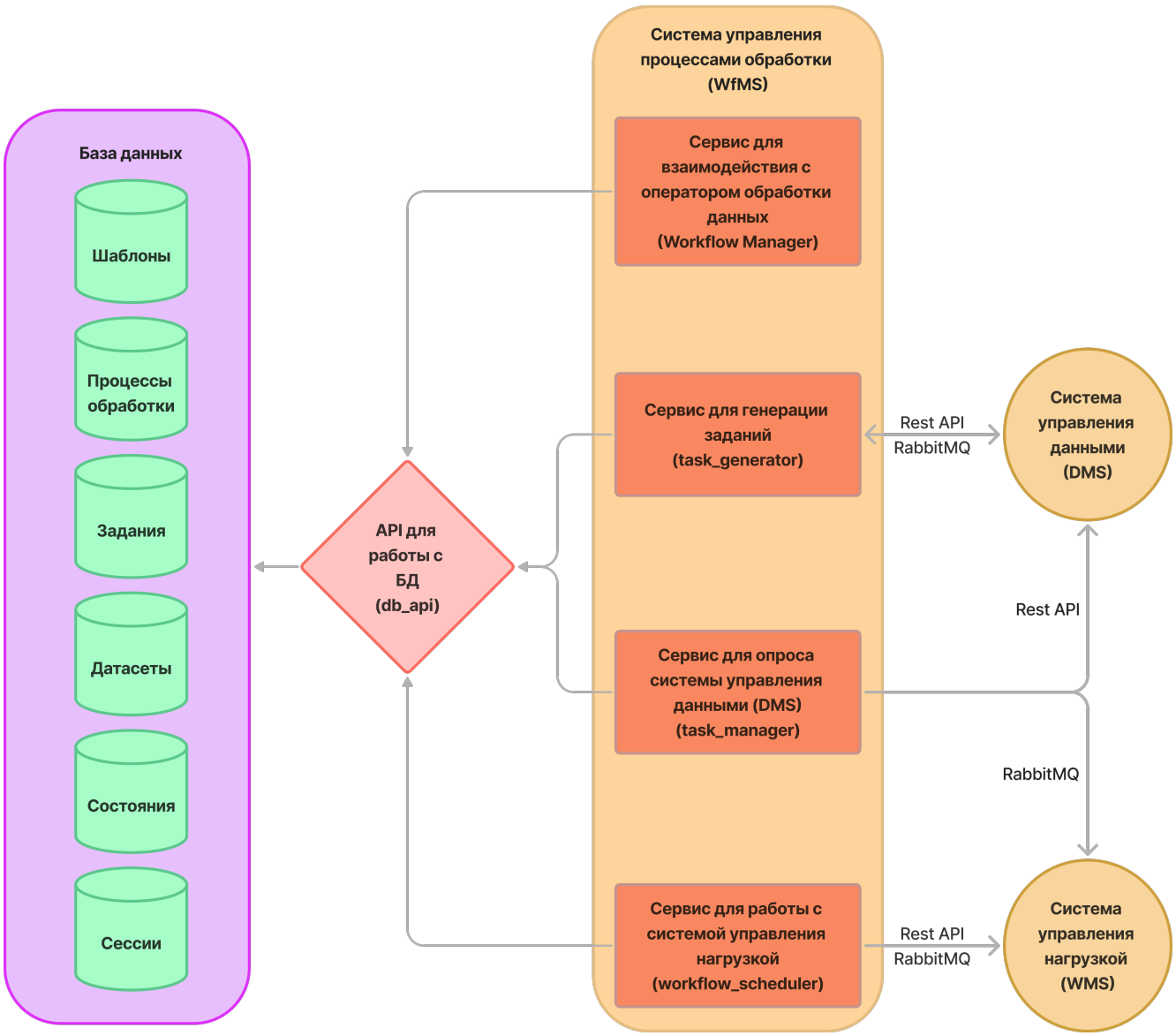


Рисунок 3.2 — Архитектура Workflow Management System

Основные преимущества микросервисной архитектуры включают:

- Повышенную доступность и отказоустойчивость. Сбой одного из сервисов не приводит к отказу всей системы.
- Масштабируемость. При достижении предельной нагрузки на микросервис можно развернуть дополнительные экземпляры этой службы и распределить нагрузку, поддерживая таким образом работоспособность большого количества экземпляров.

3.3 БАЗА ДАННЫХ И API К НЕЙ

Для хранения данных в системе WfMS используется объектно-реляционная база данных. В качестве СУБД была выбрана PostgreSQL [15], развернутая на отдельной виртуальной машине.

Структура базы данных представлена на рис. 3.3.

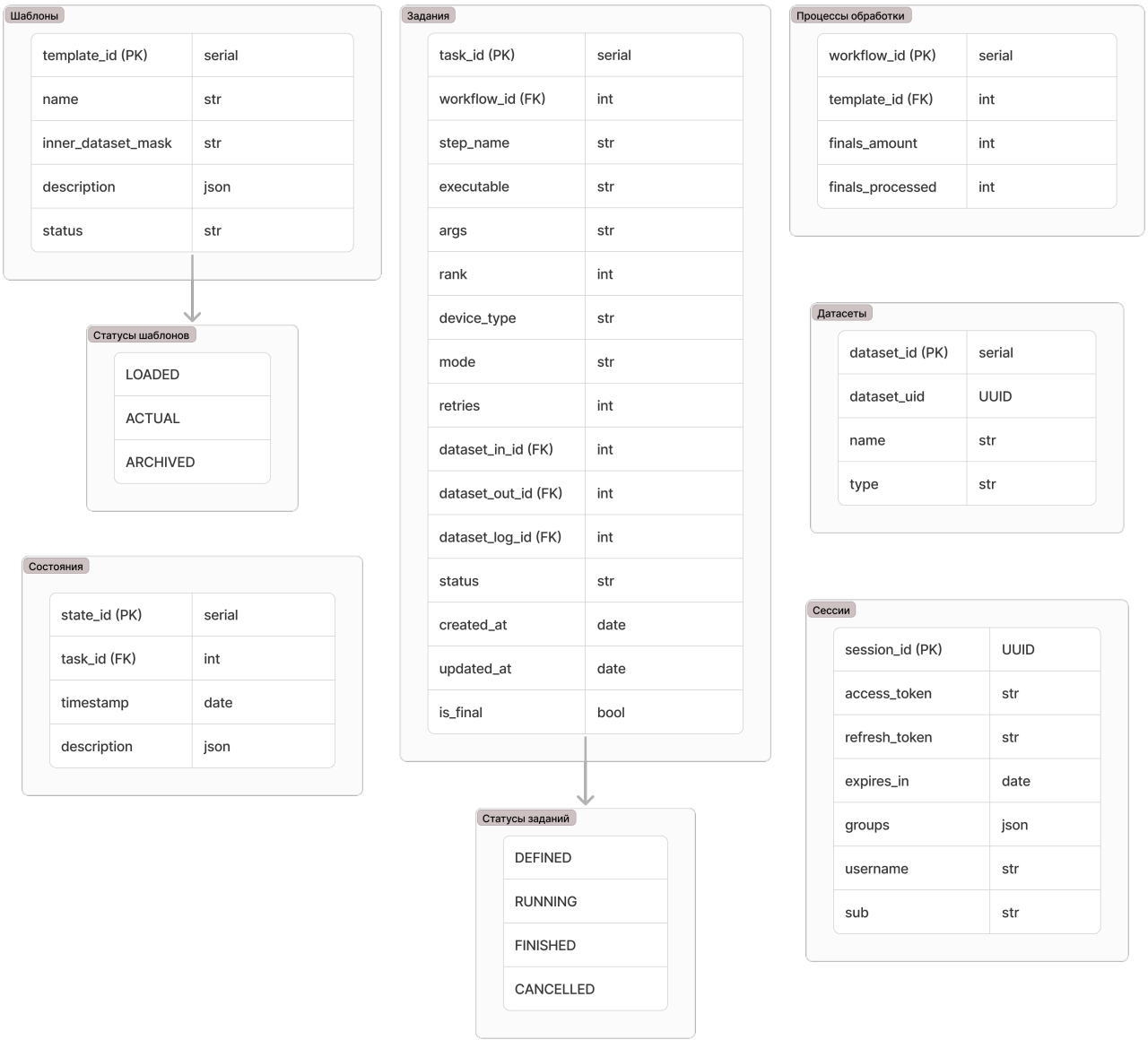


Рисунок 3.3 — Структура базы данных

Для создания и работы с базой данных используется библиотека SQLAlchemy (ORM) [16]. Миграции проводятся с помощью библиотеки Alembic [17]. Для достижения высокой скорости взаимодействия и асинхронного взаимодействия применяются асинхронные сессии и asyncpg [18].

Взаимодействие каждого из микросервисов с базой данных представлено на рис. 3.4.

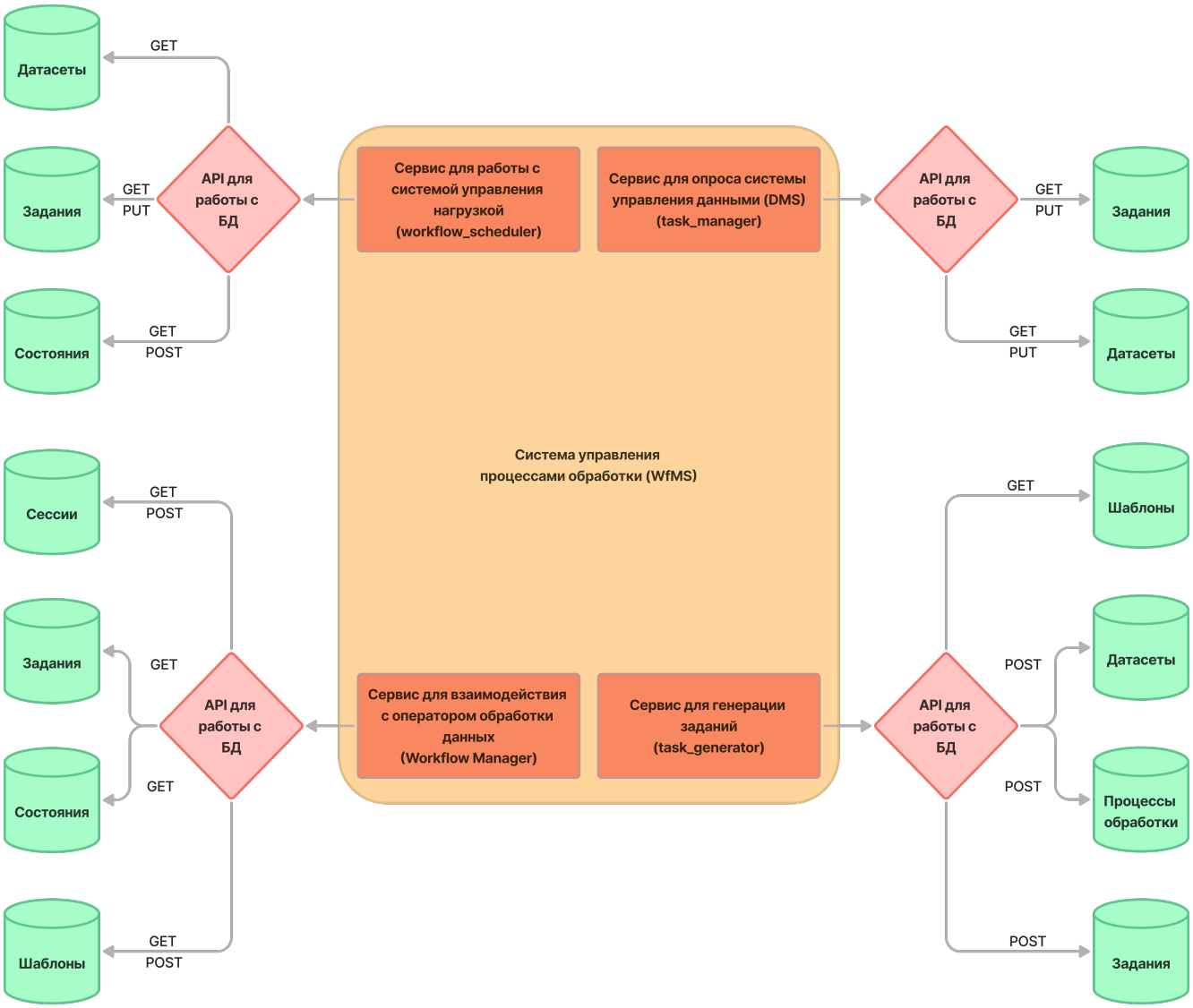


Рисунок 3.4 — Схема взаимодействия микросервисов с БД

Всё взаимодействие с базой данных осуществляется через REST API. Для этого используется фреймворк FastAPI [19]. Вся документация и проверка API производится во встроенном инструменте FastAPI — Swagger (рис. 3.5).

Рисунок 3.5 — DB API Swagger

3.4 СЕРВИС ДЛЯ ВЗАИМОДЕЙСТВИЯ С ОПЕРАТОРОМ ОБРАБОТКИ ДАННЫХ (WORKFLOW MANAGER)

Основной функционал сервиса:

1) Регистрация и авторизация пользователей с разными правами;
2) Вывод CWL-шаблонов;
3) Вывод заданий;
4) Создание CWL-шаблонов суперпользователями;
5) Предварительная валидация и сохранение CWL-шаблонов;
6) Изменение статусов шаблонов суперпользователями на «ACTUAL» и «ARCHIVED»;
7) Удаление суперпользователями шаблонов в статусе «LOADED».

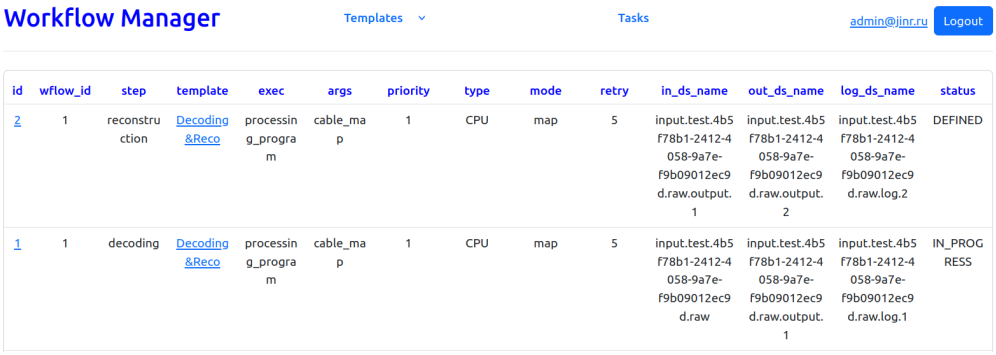
Сервис для взаимодействия с оператором обработки данных обеспечен пользовательским интерфейсом. Backend сервиса написан на FastAPI, для

frontend использовался bootstrap [20] (HTML + CSS + JS), а объединяет их воедино шаблонизатор Jinja2 [21].

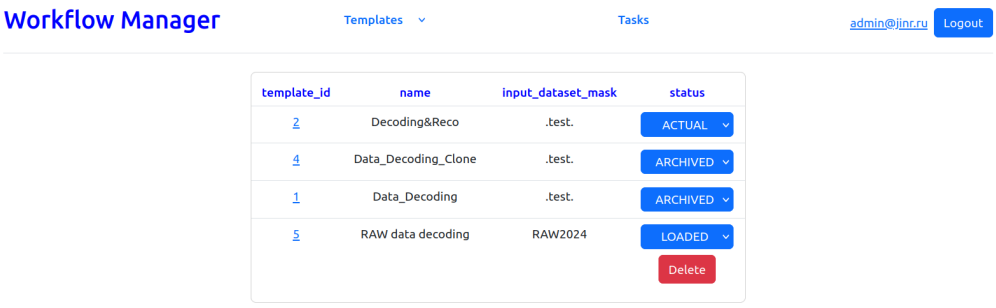
Изначально реализация была следующей: для использования сервиса необходимо было пройти процесс регистрации, а затем аутентификацию и авторизацию. Данная часть сервиса была реализована с помощью библиотеки FastAPI Users на сохранении JWT-токенов в cookie браузера. В дальнейшем было решено отказаться от внутренней регистрации и аутентификации пользователей, в угоду интеграции с новой системой SPD-IAM [22], объединяющей все приложения коллаборации SPD.

Категория пользователя определяется исходя из групп, в которых состоит пользователь в системе SPD-IAM. Суперпользователь (оператор) должен состоять в группе sof/operator.

Суперпользователям, помимо просмотра шаблонов и заданий (рис. 3.6), так же доступно создание шаблонов, в том числе и с помощью клонирования уже существующих шаблонов или из файла, путем его переноса в область создания, и смена их статусов: LOADED — загруженные шаблоны, которые можно удалить; ACTUAL — шаблоны, используемые при создании заданий; ARCHIVED — неактивные шаблоны. Вернуть шаблон в статус LOADED невозможно.



(а)



(б)

Рисунок 3.6 — а) Страница с заданиями; б) Страница с шаблонами

Отдельно можно посмотреть и сам шаблон, суперпользователю же доступна возможность его клонирования, для создания на его основе нового экземпляра. Показано на рис. 3.7.

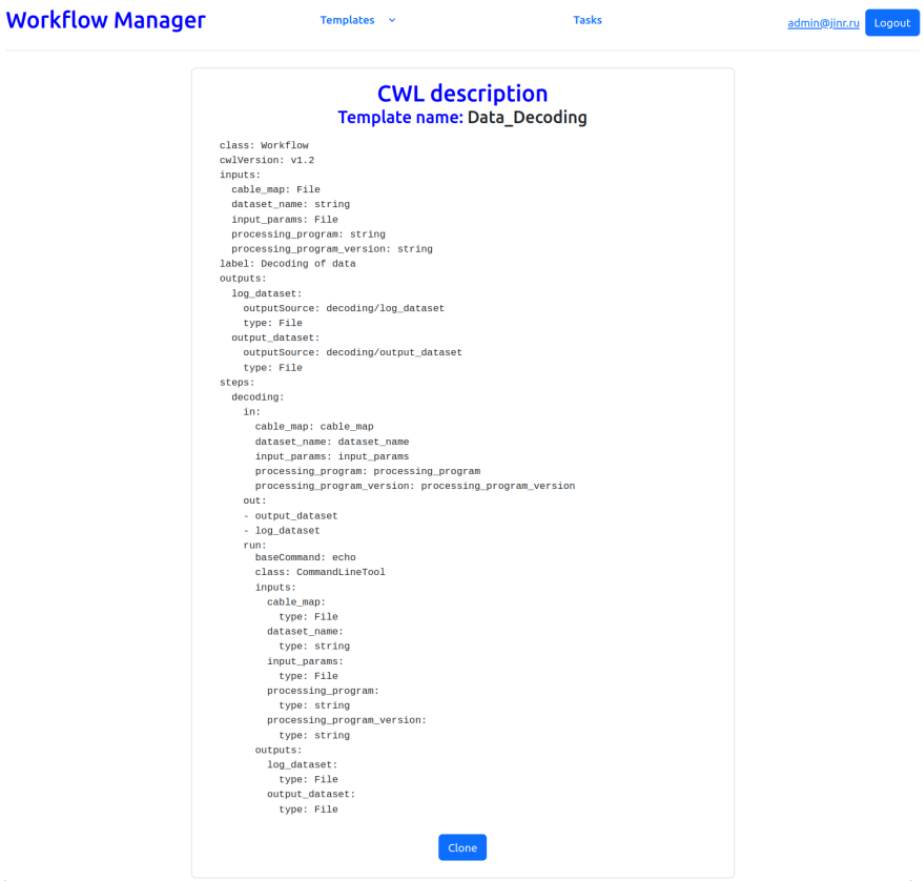


Рисунок 3.7 — Страница с отдельным шаблоном

Шаблон задается в формате Common Workflow Language (CWL). Перед сохранением в базу данных шаблон проходит процесс валидации с помощью инструментов cwltools. Страница создания шаблона представлена на рис. 3.8.

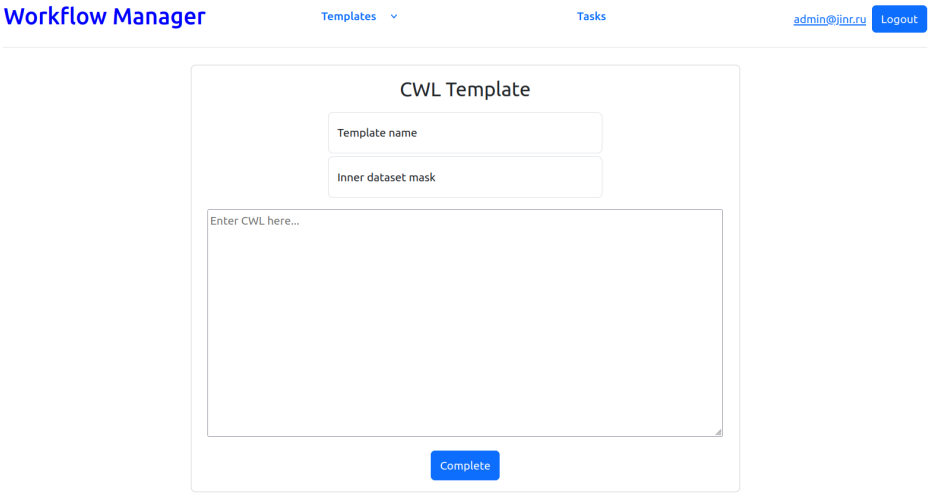


Рисунок 3.8 — Страница создания CWL-шаблона

Для заданий доступна возможность просмотра их состояний, полученных от WMS. В частности, присутствует триггер, сохраняющий состояние при смене статуса задания. Страница с состояниями задания представлена на рис. 3.9.

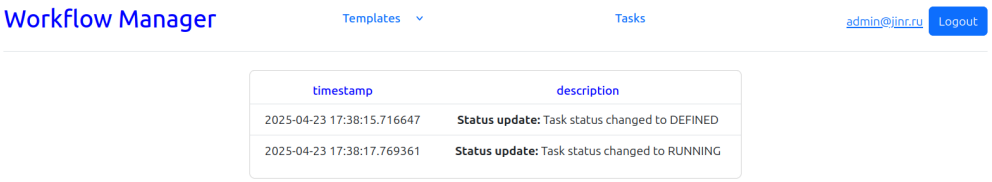


Рисунок 3.9 — Страница с состояниями задания

3.5 СЕРВИС ДЛЯ ГЕНЕРАЦИИ ЗАДАНИЙ (TASK_GENERATOR)

Сервис для генерации заданий обладает следующим функционалом:

1) Получение зарегистрированных датасетов от DMS из брокера сообщений RabbitMQ [23];
2) Сопоставление датасета по маске имени с нужным шаблоном;
3) Регистрация входного датасета в системе;
4) Создание процесса обработки по шаблону;
5) Создание выходного датасета и датасета логов в системе;
6) Создание заданий.

После того, как датасет был сформирован, DMS отправляет информацию о нем (name, dataset_uid) в брокер сообщений RabbitMQ. Информация о датасетах извлекается по порядку из очереди и сопоставляется по маске имени с шаблонами, написанными оператором обработки в соответствующем сервисе. Используются только те шаблоны, которые находятся в статусе ACTUAL.

Входные датасеты регистрируются в системе, после чего создается процесс обработки по найденному шаблону. Генерируются записи о выходных датасетах и датасетах логов в базе данных для каждого из этапов цепочки обработки с целью дальнейшего их создания в DMS перед отправкой конкретного задания на обработку. Данная реализация позволяет не создавать лишних датасетов в системе в случаях, когда задание отменено.

Далее формируется само задание. Ему присваивается статус DEFINED. Данное задание отличается от того, что будет отправлено на дальнейшую обработку из-за особенностей хранения в базе данных. Таким образом, в этом

сервисе формируется «шаблон» задания, который затем будет заполняться перед отправлением на обработку в WMS.

Все этапы генерации заданий осуществляются в асинхронном режиме.

Весь процесс наглядно можно видеть на рис. 3.10.

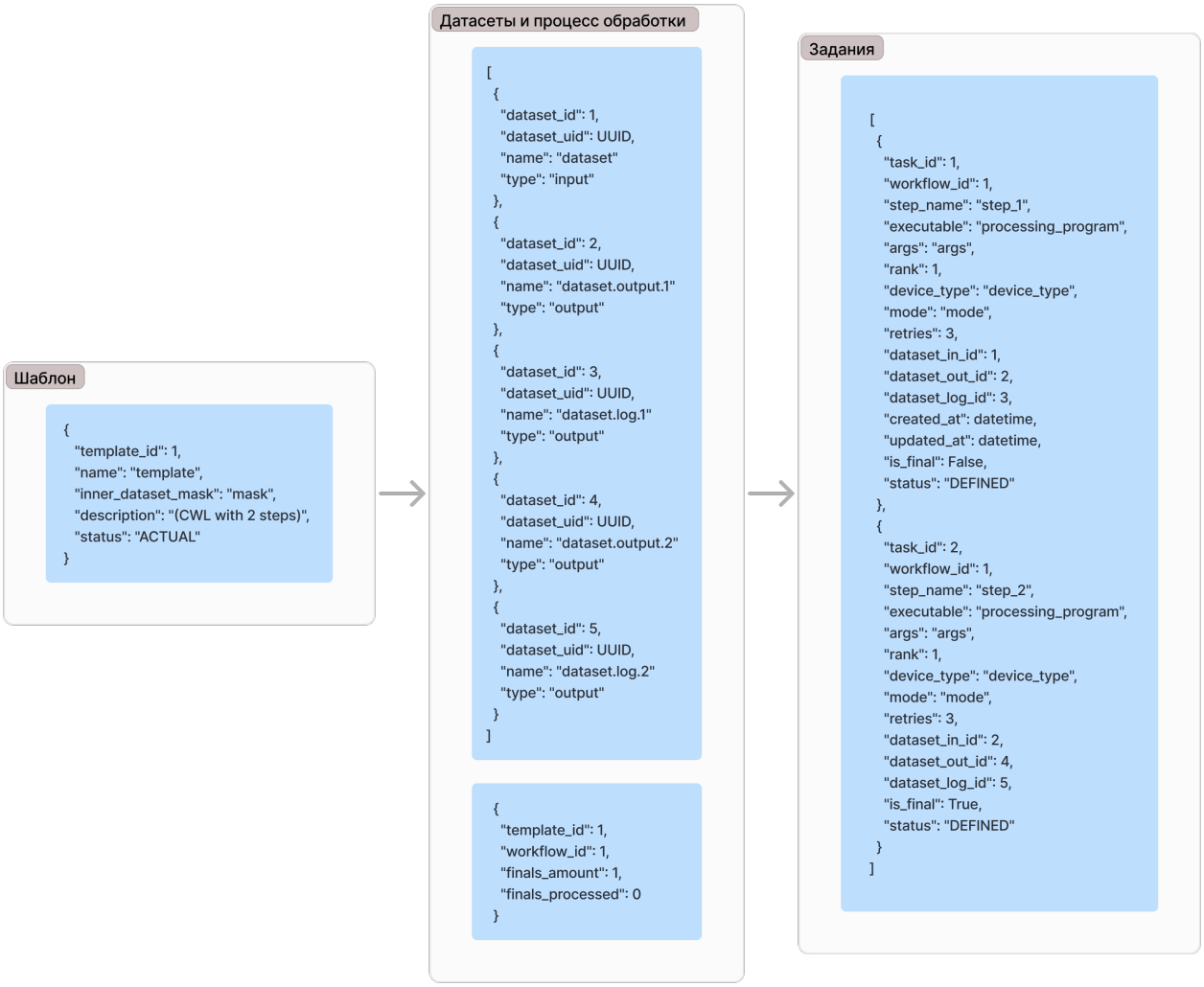


Рисунок 3.10 — Генерация задания по шаблону

Для данного этапа разработки прототипа системы достаточно создания линейной цепочки обработки. Однако в будущем может возникнуть необходимость перехода к обработке общего случая направленного графа. Для решения этой задачи предполагается возможность создания «датасетов-клонов» — разных датасетов, составленных из одних и тех же файлов. Такое усложнение позволяет в дальнейшем легко решать проблему «жадного» удаления датасетов (удаление входного датасета сразу после прохождения этапа обработки), а так

же избавляет от необходимости описывать в шаблоне сложную обработку сразу двух выходных датасетов. Подобное представление показано на рис. 3.11.

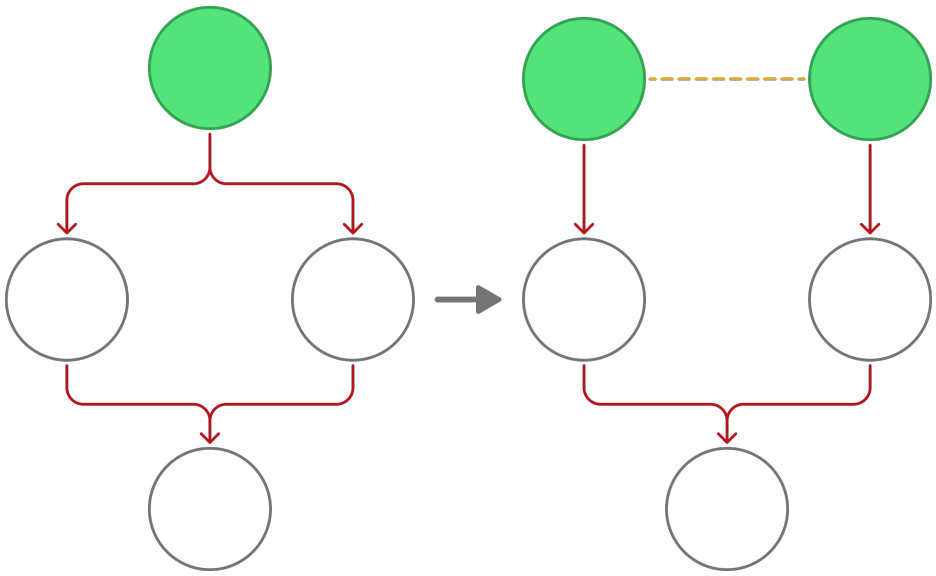


Рисунок 3.11 — Создание «датасетов-клонов»

Это поможет преобразовать ациклический граф общего вида следующим образом (рис. 3.12).

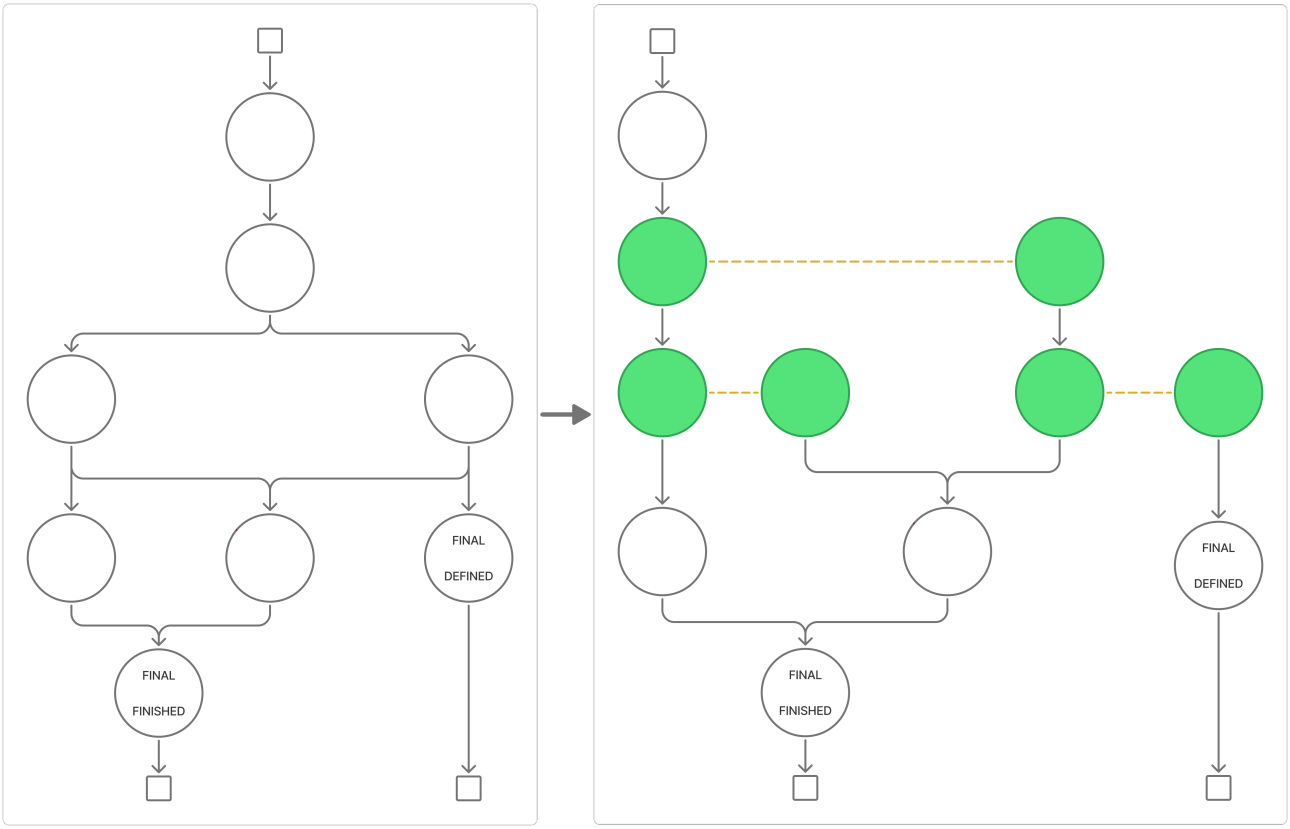


Рисунок 3.12 — Ациклический граф общего вида после создания «датасетов-клонов»

Также в таком случае понадобится еще одна смежная таблица для хранения нескольких датасетов в одном задании (рис. 3.13).

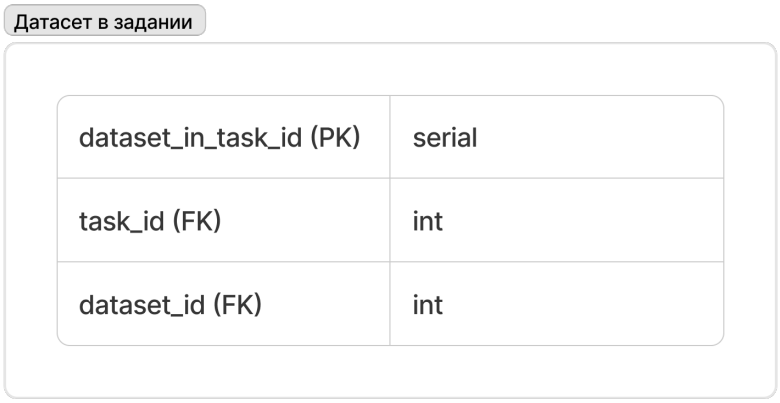


Рисунок 3.13 — Смежная таблица для хранения нескольких датасетов в задании

3.6 ВЗАИМОДЕЙСТВИЕ С DATA MANAGEMENT SYSTEM (DMS)

Основной задачей данного взаимодействия является получение новых данных, регистрация промежуточных данных и выгрузка выходных данных. Кроме основных функций, взаимодействие должно быть реализовано с учетом обеспечения высокой эффективности и максимальной скорости, исключения узких мест, а также минимизации возможных ошибок.

Взаимодействие с DMS включает следующие этапы:

1) Получение готового входного набора данных. После поступления данных в SPD Online Filter, DMS регистрирует их в своей системе. После этого WfMS может получить информацию об этом датасете из сообщения в брокере RabbitMQ.
2) Регистрация выходного набора данных. Перед любым этапом обработки данных WfMS должен зарегистрировать выходной набор данных, чтобы знать, где и как искать полученные данные. Для этого используется REST API. Промежуточные и выходные данные сопровождаются логами, которые регистрируются аналогичным образом.
3) Обработка промежуточных данных. В ходе обработки данных появляются промежуточные результаты. Если в цепочке обработки имеется n этапов, то при сохранении всех промежуточных данных после каждого

этапа объем используемой памяти значительно увеличивается. Поэтому видится возможной реализация двух различных стратегий: «жадной» и «безопасной».

При «жадной» стратегии хранится только один набор входных данных и один набор промежуточных данных. Это значит, что после завершения этапа обработки и появления нового датасета, входные данные для этого этапа удаляются. Входные данные для первого этапа остаются до завершения всей цепочки обработки, чтобы можно было восстановить данные в случае сбоя системы.

При «безопасной» стратегии все данные хранятся до полного прохождения всей цепочки обработки, после чего входные датасеты каждого этапа могут быть безопасно удалены. Плюсом такой стратегии является то, что в любой момент можно вернуться к этапу, где всё пошло не так, как предполагалось.

На сегодняшний момент решено не удалять датасеты логов для удобства отлаживания, однако при переходе от прототипа к финальной версии видится хорошим решением внедрение отдельной переменной окружения, указывающей на то, надо ли выгружать датасеты логов или же стоит их удалять.

В данном взаимодействии используется асинхронный метод, позволяющий не останавливать работу WfMS и не ждать удаления данных, так как оно производится через сообщения в RabbitMQ.

4) Опрос о заполненности выходного хранилища. Периодически DMS необходимо опрашивать о заполненности выходного хранилища, исходя из этого система в дальнейшем будет принимать решения об отмене заданий и изменении их приоритетов. Критический предел заполненности выходного хранилища должен будет задаваться через переменную окружения.

3.6.1 СЕРВИС ДЛЯ ОПРОСА DMS (TASK_MANAGER)

После того, как задания были сгенерированы в соответствующем сервисе они получают статус DEFINED. Данный сервис извлекает такие задания из базы данных и итерируется по ним. С помощью REST API в DMS (dsm-manager) отправляется запрос на получение статуса входного датасета для конкретно-

го задания. Данный датасет должен быть готов для обработки (находиться в статусе «CLOSED»).

Для входных датасетов в статусе «CLOSED» создаются выходные датасеты и датасеты логов в DMS через REST API (dsm-manager) при этом вся информация о них передаётся из внутренней базы данных WfMS, где они были созданы на этапе генерации заданий.

Задание формируется в необходимом для дальнейшей обработки виде и отправляется в очередь брокера сообщений RabbitMQ в формате JSON, после чего оно может быть извлечено WMS. Данная очередь создается в момент инициализации сервиса. Сохранность сообщений в очереди и их доставку до получателя обеспечивает RabbitMQ.

После того, как сообщение было отправлено, задание получает статус «RUNNING» (рис. 3.14), что обеспечит возможность их отслеживания и управления ими в дальнейшем.

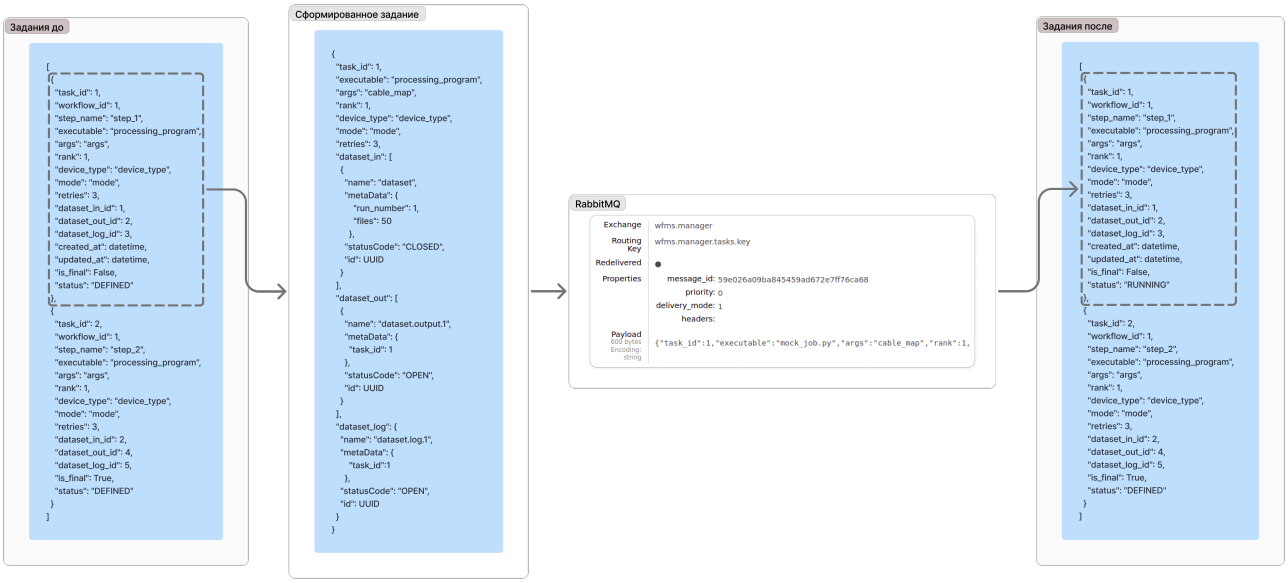


Рисунок 3.14 — Отправление заданий на обработку

Благодаря данной реализации, следующее задание в цепочке будет отправлено на обработку сразу после того, как завершится предыдущее.

3.7 ВЗАИМОДЕЙСТВИЕ С WORKLOAD MANAGEMENT SYSTEM (WMS)

Основной задачей данного взаимодействия является отправление заданий на обработку и контроль их выполнения. Взаимодействие следует выстроить так, чтобы оно обеспечивало оперативную обработку, устойчивость к сбоям и отсутствие узких мест, препятствующих эффективности системы.

Взаимодействие с WMS включает следующие этапы:

1) Отправление заданий на обработку. Сформированные задания отправляются в очередь RabbitMQ, откуда они забираются одним из сервисов WMS для последующего деления на задачи и их обработки.
2) Опрос о состоянии задания. Периодически WMS опрашивается по REST о состоянии конкретного задания. В ответ приходят метаданные о количестве обработанных файлов относительно всех файлов в датасете и аналогично для задач данного задания, а также текущий статус задания.
3) При достижении критического значения заполненности выходного хранилища будет приниматься решение об отмене заданий или изменении их приоритетов с последующим оповещением WMS. Система принятия решений представляется довольно трудоемкой и громоздкой, поэтому на данном этапе прототипирования было решено её отложить.
4) Приоритеты заданий могут быть изменены и по иным причинам, чем только по заполненности выходного хранилища, предполагается удобным использование следующей функции: $rank_{i+1} = \alpha x_i + \beta y_i + \gamma rank_i$, x_i - aging, y_i - retries.

3.7.1 СЕРВИС ДЛЯ РАБОТЫ С WMS (WORKFLOW_SCHEDULER)

Данный сервис извлекает из базы данных задания в статусе «RUNNING». После чего итерируется по ним, опрашивая систему управления нагрузкой WMS об их состоянии. Эти состояния записываются в базу данных для удобного мониторинга оператором обработки и дальнейшего принятия решений.

Если при опросе задание оказывается завершенным (находится в статусе finished в WMS), то выходной датасет данного задания закрывается путем отправления PATCH-запроса в DMS.

После этого задание получает статус «FINISHED» в WfMS. На изменение статуса срабатывает триггер в базе данных, который записывает изменение статуса в состояние.

Далее происходит удаление датасетов по одной из двух стратегий, описанных выше. На данный момент решено реализовать «безопасную» стратегию, так как она удовлетворяет всем необходимым потребностям. В этом случае каждый раз после обработки задания проверяется, является ли оно последним. Если является, то finals_processed (количество обработанных последних заданий) увеличивается на 1. Когда это число становится равным finals_amount (количество последних заданий), происходит удаление всех входных датасетов заданий данной цепочки.

В дальнейшем же переключение между стратегиями будет реализовано через задание переменной окружения.

С отменой задания дела обстоят куда сложнее. Она может быть инициирована как WfMS на основе данных о заполненности выходного хранилища с последующим оповещением системы обработки данных, так и самой WMS. В таких случаях после закрытия датасетов в DMS задания будут получать статус «CANCELLED». В дальнейшем такие задания предполагается обрабатывать по сложной статусной модели.

В будущем данный сервис будет ответственен за изменение приоритетов заданий на основе данных из состояний. Измененный приоритет будет отправляться в WMS с помощью REST.

3.8 ЛОГИРОВАНИЕ, КОНТЕЙНЕРИЗАЦИЯ И ОРКЕСТРАЦИЯ

Для получения необходимой информации о состоянии системы был разработан logger, записывающий необходимую для дальнейшего анализа информацию о работе каждого сервиса. В частности, записываются название сервиса, уровень сообщения (INFO, WARNING, ERROR, CRITICAL), дата и время получения сообщения и описание сообщения.

Одно из сообщений logger можно видеть на рис. 3.15.

```
task_generator 2024-09-24 20:59:44,380 INFO Logger started
task_generator 2024-09-24 20:59:44,665 INFO Templates successfully initialised.
```

Рисунок 3.15 — Сообщение logger

Для быстрого развертывания приложений на любой системе и налаживания их совместного взаимодействия используются методы контейнеризации и оркестрации с помощью Docker и Docker-Compose [24]. Представление реализованных сервисов в docker compose изображено на рис. 3.16.

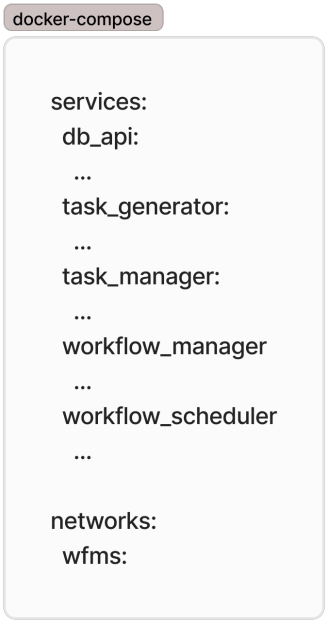


Рисунок 3.16 — Сервисы WfMS в docker-compose

3.9 ОБЩАЯ СТРУКТУРА СИСТЕМЫ

Структура кода системы организована согласно общепринятым практикам при разработке на языке Python.

```
wfms-spd-online
├── alembic.ini ..... конфигурационный файл для alembic
├── config.py ..... файл, подгружающий переменные окружения
├── database ..... модуль по взаимодействию с базой данных
    ├── api.py ..... инициализация FastAPI
```

```
│   ├── db
│   │   ├── engine.py ......... подключение к БД и выдача сессий
│   │   ├── models.py ..................... ORM модели SQLAlchemy
│   ├── db_api
│   │   ├── router_functions
│   │   │   ├── common_functions.py ........ взаимодействие с БД
│   │   └── routers ................. взаимодействие с таблицами
│   │       ├── dataset_router.py
│   │       ├── state_router.py
│   │       ├── task_router.py
│   │       ├── template_router.py
│   │       ├── user_session_router.py
│   │       └── workflow_router.py
│   ├── Dockerfile ..................... описание образа db_api
│   ├── migrations ............................ миграции alembic
│   │   ├── env.py
│   │   ├── README
│   │   ├── script.py.mako
│   │   └── versions ............................. все миграции
│   └── scripts
│       └── app.sh ......... bash-скрипт для запуска web-сервера
├── docker-compose.yml .... конфигурационный файл docker compose
├── .env ................................. переменные окружения
├── .env.example .......... пример файла с переменными окружения
├── .gitignore
├── logger
│   ├── custom_logger.py ............... преднастроенный logger
│   ├── logs ................................... все логи системы
├── paths ..................................... все внешние пути
│   ├── db_paths.py
│   ├── dms_paths.py
│   ├── iam_paths.py
│   └── wms_paths.py
├── README.md ................................ описание системы
```

```
├── response_models
│   ├── model_enums
│   │   ├── enums.py ..................... все перечисляемые типы
│   ├── models ................................ pydantic модели
│   │   ├── base_models.py
│   │   ├── dataset_models.py
│   │   ├── state_models.py
│   │   ├── task_models.py
│   │   ├── template_models.py
│   │   ├── user_session_model.py
│   │   └── workflow_models.py
│   ├── pk_types
│   │   ├── models_pk_types.py ........... типы первичных ключей
├── task_generator
│   ├── Dockerfile .............. описание образа task_generator
│   ├── rabbit_receiver
│   │   ├── consumer.py ......... получение сообщений из брокера
│   └── task_generator.py ....... основной скрипт task_generator
├── task_manager
│   ├── app
│   │   ├── producer.py ......... отправление сообщений в брокер
│   ├── Dockerfile ................ описание образа task_manager
│   └── task_manager.py ........... основной скрипт task_manager
├── workflow_manager
│   ├── app.py ........................... инициализация FastAPI
│   ├── auth
│   │   ├── auth_middleware.py ....... авторизация пользователей
│   │   ├── current_user.py ............. работа с пользователем
│   │   ├── session_service.py .............. работа с сессиями
│   │   └── token_manager.py .......... взаимодействие с SPD-IAM
│   ├── Dockerfile ............ описание образа workflow_manager
│   ├── html_templates
│   │   ├── app .......... страницы, доступные после авторизации
│   │   │   ├── all_tasks.html
```

```
        │   │   ├── all_templates.html
        │   │   ├── create_template.html
        │   │   ├── state.html
        │   │   └── template_description.html
        │   └── landing .. базовые элементы и общедоступные страницы
        │       ├── about.html
        │       ├── authed_header.html
        │       ├── base.html
        │       ├── footer.html
        │       ├── header.html
        │       └── index.html
        ├── logger.py ....................... инициализация logger
        ├── pages
        │   ├── pages.py .... общие функции приложения со страницами
        │   └── validation.py ..... функции приложения для валидации
        ├── paths
        │   ├── paths.py ..................... все пути приложения
        └── static ........................... статические элементы
            ├── favicon.ico
            └── style.css
    └── workflow_scheduler
        ├── producer
        │   ├── to_delete_msgs.py .. удаление датасетов через брокер
        ├── Dockerfile .......... описание образа workflow_scheduler
        └── workflow_scheduler.py ........ основной скрипт scheduler
```

3.10 ТЕСТИРОВАНИЕ СИСТЕМЫ

Логи task_generator после генерации датасета и отправления его в очередь:

task_generator 2025-04-25 16:24:36,562 INFO Task generator initialized

task_generator 2025-04-25 16:24:37,745 INFO Templates successfully initialized.

task_generator 2025-04-25 16:24:37,745 INFO Creating RabbitMQ connection...

task_generator 2025-04-25 16:24:37,767 INFO RabbitMQ connection is open and active.

task_generator 2025-04-25 16:24:37,767 INFO Creating RabbitMQ channel...

task_generator 2025-04-25 16:24:37,777 INFO RabbitMQ channel created.

task_generator 2025-04-25 16:24:37,780 INFO Queue 'dsm.register.dataset.input' successfully initialized.

task_generator 2025-04-25 16:40:42,335 INFO Input dataset received.

task_generator 2025-04-25 16:40:43,211 INFO Templates successfully initialized.

task_generator 2025-04-25 16:40:43,211 INFO Reusing existing RabbitMQ connection.

task_generator 2025-04-25 16:40:44,279 INFO Input dataset with name=input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw created in database.

task_generator 2025-04-25 16:40:45,118 INFO Workflow with workflow_id=36 created in database.

task_generator 2025-04-25 16:40:45,579 INFO Output dataset with name=input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.output.1 created in database.

task_generator 2025-04-25 16:40:45,864 INFO Log dataset with name=input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.log.1 created in database.

task_generator 2025-04-25 16:40:47,923 INFO Task with task_id=38 created in database.

task_generator 2025-04-25 16:40:48,094 INFO Output dataset with name=input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.output.2 created in database.

task_generator 2025-04-25 16:40:48,264 INFO Log dataset with name=input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.log.2 created in database.

task_generator 2025-04-25 16:40:48,532 INFO Task with task_id=39 created in database.

Логи task_manager:

task_manager 2025-04-25 16:24:36,505 INFO Task manager initialized

task_manager 2025-04-25 16:24:36,505 INFO Creating RabbitMQ connection...

task_manager 2025-04-25 16:24:36,570 INFO RabbitMQ connection created

task_manager 2025-04-25 16:40:48,949 INFO UID of dataset with ID=45 updated.

task_manager 2025-04-25 16:40:49,190 INFO UID of dataset with ID=46 updated.

task_manager 2025-04-25 16:40:49,202 INFO Task with id = 38 sent to RabbitMQ.

task_manager 2025-04-25 16:40:49,352 INFO Status of task with id = 38 updated to RUNNING.

Созданные задания в workflow_manager (рис. 3.17)

id	wflow_id	step	template	exec	args	priority	type	mode	retries	in_ds_name	out_ds_name	log_ds_name	status
38	36	decoding	d&r	processing_program	cable_map	1	CPU	Map	2	input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw	input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.output.1	input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.log.1	RUNNING
39	36	reco	d&r	processing_program	cable_map	1	CPU	Map	2	input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.output.1	input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.output.2	input.test.c238ef6e-adac-49ae-b8d8-67cb33da0d42.raw.log.2	DEFINED

Рисунок 3.17 — Вид созданных заданий по полученному датасету в workflow_manager

Состояния заданий в workflow_manager (рис. 3.18)

timestamp	description
2025-04-25 16:40:45.877739	**Status update:** Task status changed to DEFINED
2025-04-25 16:40:49.207776	**Status update:** Task status changed to RUNNING

timestamp	description
2025-04-25 16:40:48.270340	**Status update:** Task status changed to DEFINED

Рисунок 3.18 — Вид состояний заданий в workflow_manager

Сообщение с заданием в статусе «RUNNING» в очереди RabbitMQ (рис. 3.19).

Рисунок 3.19 — Сообщение в очереди RabbitMQ

РЕЗУЛЬТАТЫ РАБОТЫ И ДАЛЬНЕЙШИЕ ПЛАНЫ

Результаты:

1) разработан API для работы с базой данных, открывающий доступ для сервисов WfMS к определенному набору функций.
2) создан сервис для работы с оператором обработки данных, позволяющий просматривать задания и шаблоны, а также предоставляющий возможность суперпользователям генерировать шаблоны и управлять их статусами.
3) разработан сервис для генерации заданий, сопоставляющий датасеты с шаблонами по маске имени, создающий цепочки обработки и задания.
4) создан сервис для опроса DMS, опрашивающий его о готовности входного датасета для каждого готового задания, создающий выходные датасеты для таких заданий и отправляющий задания на обработку в WMS.
5) произведена контейнеризация всех приложений и настроена их оркестрация с помощью docker-compose.
6) внедрен logger, сохраняющий информацию об актуальном состоянии системы в каждый момент времени.
7) разработан первый этап сервиса для взаимодействия с WMS, позволяющий завершать задания с закрытием выходных датасетов и удалять датасеты после прохождения всех этапов цепочки обработки.

Дальнейшие планы:

1) Последующая доработка сервиса для взаимодействия с WMS;
2) Переход к полной асинхронности;
3) Тестирование системы.

ЗАКЛЮЧЕНИЕ

Современные эксперименты в области физики высоких энергий, сталкиваются с необходимостью эффективной обработки огромных объемов данных, генерируемых детекторными системами. Эксперимент SPD на коллайдере NICA предъявляет высокие требования к системе фильтрации данных в реальном времени, учитывая интенсивность событий и необходимость оперативной обработки информации. В этом контексте проектирование системы управления процессами обработки данных становится не просто задачей, а ключевым фактором для достижения успешных результатов в исследованиях физики высоких энергий. Эффективное управление процессами обработки данных в вычислительных комплексах имеет ключевое значение для обеспечения производительности и надежности системы.

В ходе работы были рассмотрены основные аспекты функционирования системы SPD Online Filter эксперимента SPD и разработана её подсистема - система управления процессами обработки.

СПИСОК ЛИТЕРАТУРЫ

1. International spin physics collaboration at the collider NICA [Электронный ресурс]. — URL: http://spd.jinr.ru/.
2. Conceptual design of the Spin Physics Detector / V. M. Abazov [и др.]. — 2022. — arXiv: 2102.00442 [hep-ex].
3. Комплекс NICA. [Электронный ресурс]. — URL: https://nica.jinr.ru/ru/complex.php.
4. On the physics potential to study the gluon content of proton and deuteron at NICA SPD / A. Arbuzov [и др.] // Prog. Part. Nucl. Phys. — 2021. — arXiv: 2011.15005 [hep-ex].
5. Экспериментальная установка SPD. [Электронный ресурс]. — URL: https://nica.jinr.ru/ru/projects/spd.php.
6. *Oleynik D.* Data processing in HEP experiments. [Электронный ресурс]. — URL: https://lit.jinr.ru/sites/lit.jinr.ru/files/pdf/HEPNICA_computing_2022_OleynikD.pdf.
7. Technical Design Report of the Spin Physics Detector at NICA / V. M. Abazov [и др.]. — 2024. — arXiv: 2404.08317v1 [hep-ex].
8. *Пономарев Е. В.* Проектирование сервиса управления процессом обработки данных на специализированной распределенной вычислительной системе SPD Online filter. — СПб, 2023.
9. PanDA. [Электронный ресурс]. — URL: https://github.com/PanDAWMS/pandawms.
10. A new era for central processing and production in CMS / E. Fajardo [и др.] // Journal of Physics: Conference Series. — 2012. — Т. 396, № 4. — С. 042018.

11. The ATLAS trigger system for LHC Run 3 and trigger performance in 2022 / G. Aad [и др.] // Journal of Instrumentation. — 2024. — Т. 19, № 06. — P06029.

12. Apache AirFlow. [Электронный ресурс]. — URL: https://airflow.apache.org/.

13. Основные сущности AirFlow. [Электронный ресурс]. — URL: https://cloud.vk.com/blog/airflow-what-it-is-how-it-works/.

14. Common Workflow Language. [Электронный ресурс]. — URL: https://www.commonwl.org/.

15. PostgreSQL. [Электронный ресурс]. — URL: https://www.postgresql.org/.

16. SQL Alchemy 2.0. [Электронный ресурс]. — URL: https://docs.sqlalchemy.org/en/20/.

17. Alembic. [Электронный ресурс]. — URL: https://alembic.sqlalchemy.org/en/latest/.

18. Asyncpg. [Электронный ресурс]. — URL: https://pypi.org/project/asyncpg/.

19. FastAPI. [Электронный ресурс]. — URL: https://fastapi.tiangolo.com/.

20. Bootstrap. [Электронный ресурс]. — URL: https://getbootstrap.com/.

21. Jinja. [Электронный ресурс]. — URL: https://jinja.palletsprojects.com/en/stable/.

22. SPD-IAM. [Электронный ресурс]. — URL: https://spd-iam.jinr.ru.

23. RabbitMQ. [Электронный ресурс]. — URL: https://www.rabbitmq.com/.

24. Docker-compose. [Электронный ресурс]. — URL: https://docs.docker.com/compose/.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД WFMS

```
class Template(Base):
    """
    Template table
        Operator-created templates to define tasks based on the input datasets
    """

    __tablename__ = "template"

    template_id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(32), nullable=False)
    inner_dataset_mask: Mapped[str] = mapped_column(String(32), nullable=
        False)
    description: Mapped[JSON] = mapped_column(JSON, nullable=False)
    status: Mapped[TemplateStatus] = mapped_column(
        Enum(TemplateStatus, validate_strings=True),
        nullable=False,
        default=TemplateStatus.LOADED.value
    )

    workflow: Mapped[List["Workflow"]] = relationship(back_populates="
        template")


class Dataset(Base):
    """
    Dataset table
        Main datasets info for creating tasks
```

```
        dataset_uid – common UID of dataset in SOF, creating by DMS
    """

    __tablename__ = "dataset"

    dataset_id: Mapped[int] = mapped_column(primary_key=True)
    dataset_uid: Mapped[UUID] = mapped_column(UUID, nullable=True)
    name: Mapped[str] = mapped_column(String(256), nullable=False)
    type: Mapped[DatasetType] = mapped_column(Enum(DatasetType,
        validate_strings=True), nullable=False)


class Workflow(Base):
    """Workflow table"""

    __tablename__ = "workflow"
    __table_args__ = (CheckConstraint("finals_amount > 0"),
        CheckConstraint("finals_processed <= finals_amount"))

    workflow_id: Mapped[int] = mapped_column(primary_key=True)
    template_id: Mapped[int] = mapped_column(ForeignKey('template.
        template_id'), nullable=False)
    finals_amount: Mapped[int] = mapped_column(Integer, nullable=False)
    finals_processed: Mapped[int] = mapped_column(Integer, nullable=False,
        default=0)

    template: Mapped["Template"] = relationship(back_populates="workflow")
    task: Mapped[List["Task"]] = relationship(back_populates="workflow2")


class Task(Base):
    """Task table"""

    __tablename__ = "task"
```

```
__table_args__ = (CheckConstraint("retries > 0"), CheckConstraint("
    rank > 0"))

task_id: Mapped[int] = mapped_column(primary_key=True)
workflow_id: Mapped[int] = mapped_column(ForeignKey('workflow.
    workflow_id'), nullable=False)
step_name: Mapped[str] = mapped_column(String(64), nullable=False)
executable: Mapped[str] = mapped_column(String(64), nullable=False)
args: Mapped[str] = mapped_column(String(256), nullable=False)
rank: Mapped[int] = mapped_column(nullable=False)
device_type: Mapped[str] = mapped_column(Enum(TaskDeviceType,
    validate_strings=True), nullable=False)
mode: Mapped[str] = mapped_column(Enum(TaskMode, validate_strings=
    True), nullable=False)
retries: Mapped[int] = mapped_column(nullable=False)
dataset_in_id: Mapped[int] = mapped_column(ForeignKey('dataset.
    dataset_id'), nullable=False)
dataset_out_id: Mapped[int] = mapped_column(ForeignKey('dataset.
    dataset_id'), nullable=False)
dataset_log_id: Mapped[int] = mapped_column(ForeignKey('dataset.
    dataset_id'), nullable=False)
status: Mapped[str] = mapped_column(
    Enum(TaskStatus, validate_strings=True),
    nullable=False,
    default = TaskStatus.DEFINED.value
)
created_at: Mapped[datetime] = mapped_column(DateTime, default=func.
    now(), nullable=False)
updated_at: Mapped[datetime] = mapped_column(DateTime, default=func.
    now(), onupdate=func.now(), nullable=False)
is_final: Mapped[bool] = mapped_column(Boolean, nullable=False)

dataset_in: Mapped["Dataset"] = relationship(argument="Dataset",
    foreign_keys=[dataset_in_id])
```

```
    dataset_out: Mapped["Dataset"] = relationship(argument="Dataset",
        foreign_keys=[dataset_out_id])
    dataset_log: Mapped["Dataset"] = relationship(argument="Dataset",
        foreign_keys=[dataset_log_id])
    workflow2: Mapped["Workflow"] = relationship(back_populates="task")
    state: Mapped[List["State"]] = relationship(back_populates="task")


class State(Base):
    """State table"""

    __tablename__ = "state"

    state_id: Mapped[int] = mapped_column(primary_key=True)
    task_id: Mapped[int] = mapped_column(ForeignKey('task.task_id'),
        nullable=False)
    timestamp: Mapped[datetime] = mapped_column(DateTime, server_default
        =func.now(), nullable=False)
    description: Mapped[JSON] = mapped_column(JSON, nullable=False)

    task: Mapped["Task"] = relationship(back_populates="state")


class UserSession(Base):
    """Users' session table"""

    __tablename__ = "user_session"

    session_id: Mapped[UUID] = mapped_column(UUID, primary_key=True,
        server_default=text("gen_random_uuid()"))
    access_token: Mapped[str] = mapped_column(TEXT, nullable=False)
    refresh_token: Mapped[str] = mapped_column(TEXT, nullable=False)
    expires_in: Mapped[datetime] = mapped_column(DateTime, nullable=False)
    groups: Mapped[JSON] = mapped_column(JSON, nullable=True)
```

```
    username: Mapped[str] = mapped_column(String(256), nullable=True)
    sub: Mapped[str] = mapped_column(String(256), nullable=False)


class Database:
    """Allow to interact with the database"""

    def __init__(self, db_url: str) -> None:
        self._engine: AsyncEngine = create_async_engine(db_url, echo=True)
        self._session_maker: async_sessionmaker[AsyncSession] =
            async_sessionmaker(self._engine, expire_on_commit=False)
        self.logger: CustomLogger = CustomLogger('db_api')
        self.logger.info('Logger started')

    async def check_database(self) -> None:
        """Checking database availability when starting an application"""

        try:
            async with self._session_maker() as session:
                await session.execute(text("SELECT 1"))
            self.logger.info("Connection to database established successfully.")
        except SQLAlchemyError as err:
            self.logger.critical(f"Unable to connect to the database due to a
                SQLAlchemy error: {err}")
            sys.exit(1)

    async def get_async_session(self) -> AsyncGenerator[AsyncSession, None]:
        """Async session generator"""

        try:
            async with self._session_maker() as session:
                yield session
        except SQLAlchemyError as err:
            self.logger.error(f"Database session error: {err}", exc_info=True)
```

```
            raise

    async def dispose(self) -> None:
        """Closing database connection"""

        await self._engine.dispose()


def validate_model_type(model: Type[Base]) -> None:
    """Checks if the model type is correct"""

    if model not in Base.__subclasses__():
        raise HTTPException(status_code=status.
            HTTP_400_BAD_REQUEST, detail="Invalid model type")


def get_constraints(model: Type[Base]) -> List[str]:
    """Adds model's constraints into exception message"""

    constraints = []
    for constraint in model.metadata.tables.get(model.__tablename__).
        constraints:
        if isinstance(constraint, CheckConstraint):
            constraints.append(str(constraint.sqltext))
    return constraints


async def get_all_records(model: Type[Base], session: AsyncSession) ->
    Sequence[Base]:
    """Transfers a list of all records from the database"""

    validate_model_type(model)
    stmt = select(model)
    if model is Template:
```

```
        stmt = stmt.order_by(Template.status)
    records = (await session.scalars(stmt)).all()

    if not records:
        raise HTTPException(status_code=status.
            HTTP_204_NO_CONTENT, detail=f"{model.__name__} not
            found")

    return records


async def get_record_by_id(model: Type[Base], specific_id: int | UUID, session:
    AsyncSession) -> Type[Base]:
    """Transfers record from database by its id"""

    validate_model_type(model)
    record = await session.get(model, specific_id)
    if not record:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
            detail=f"{model.__name__} not found")
    return record


async def create_record(model: Type[Base], data: dict, returning_field: str,
    session: AsyncSession) -> int | UUID:
    """Creates a record of workflow in database"""

    stmt = insert(model).values(**data).returning(getattr(model, returning_field
        ))
    try:
        async with session.begin():
            record_id = await session.scalar(stmt)
        return record_id
    except IntegrityError as err:
```

```
            constraints = get_constraints(model)
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"Integrity error occurred: {str(err.orig)}, Constraints: {
                    constraints}"
            )


async def update_record(model: Type[Base],
                        specific_id: int | UUID,
                        updates: dict,
                        id_field: str,
                        session: AsyncSession) -> Message:
    """Changes records in database"""

    validate_model_type(model)

    id_column = getattr(model, id_field, None)
    if not isinstance(id_column, Mapped):
        raise HTTPException(status_code=status.
            HTTP_400_BAD_REQUEST,
                            detail=f"{id_field} is not a valid column for {
                                model.__name__}")

    try:
        stmt = update(model).where(id_column == specific_id).values(**
            updates)
        result = await session.execute(stmt)

        if result.rowcount == 0:
            raise HTTPException(status_code=status.
                HTTP_404_NOT_FOUND, detail=f"{model.__name__} not
                found")
    except IntegrityError as err:
```

```
            constraints = get_constraints(model)
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"Integrity error occurred: {str(err.orig)}, Constraints: {
                    constraints}"
            )

        await session.commit()
        update_field, update_value = next(iter(updates.items()))

        return Message(message=f'{update_field.capitalize()} of {model.
            __name__.lower()} '
                               f'{specific_id} changed to {update_value}')


async def delete_record(model: Type[Base], specific_id: int | UUID, session:
    AsyncSession) -> Message:
    """Deletes the record from database"""

    async with session.begin():
        record = await session.get(model, specific_id)
        if not record:
            raise HTTPException(status_code=status.
                HTTP_404_NOT_FOUND, detail=f"{model.__name__} not
                found")

        if model is Template:
            if cast(Template, record).status != "LOADED":
                raise HTTPException(status_code=status.
                    HTTP_403_FORBIDDEN, detail="Unavailable status")

        await session.delete(record)
```

```
    return Message(message=f'Record of {model.__name__} with id = {
        specific_id} deleted')


@router.get(path=ALL_PATH, status_code=status.HTTP_200_OK,
    response_model=List[DatasetResponse])
async def get_all_datasets(session: AsyncSession = Depends(db.
    get_async_session)) -> List[DatasetResponse]:
    """Transfers a list of all datasets from the database"""

    return [DatasetResponse.model_validate(record) for record in await
        get_all_records(Dataset, session)]


@router.get(path=SPECIFIC_PATH, status_code=status.HTTP_200_OK,
    response_model=DatasetResponse)
async def dataset_response(specific_id: Ids.DatasetIdType,
                           session: AsyncSession = Depends(db.
                               get_async_session)) -> DatasetResponse:
    """Transfers a dataset from database by its id"""

    return DatasetResponse.model_validate(await get_record_by_id(Dataset,
        specific_id, session))


@router.get(path=WORKFLOW_PATH+DMS_PATH+DELETE_PATH,
    status_code=status.HTTP_200_OK, response_model=List[DmsDatasetId])
async def dataset_uids_to_delete(workflow_id: Ids.WorkflowIdType,
                                 session: AsyncSession = Depends(db.
                                     get_async_session)) -> List[DmsDatasetId
                                     ]:
    """Transfers a list of dataset_uids from database for all input datasets of
        tasks with same workflow_id"""
```

```
    stmt = (
        select(Dataset.dataset_uid)
        .join(Task, Dataset.dataset_id == Task.dataset_in_id)
        .where(Task.workflow_id == workflow_id)
        .distinct()
    )

    result = (await session.execute(stmt)).scalars().all()
    return [DmsDatasetId(id=uid) for uid in result]


@router.post(path=CREATE_PATH, status_code=status.
    HTTP_201_CREATED, response_model=DatasetId)
async def create_dataset(data: DatasetCreate, session: AsyncSession = Depends(
    db.get_async_session)) -> DatasetId:
    """Creates a record of dataset in database"""

    dataset_id = await create_record(Dataset, data.model_dump(), 'dataset_id'
        , session)
    return DatasetId(dataset_id=dataset_id)


@router.patch(path=SPECIFIC_PATH+UID_PATH, status_code=status.
    HTTP_200_OK, response_model=Message)
async def change_rank(specific_id: Ids.DatasetIdType,
                      uid: Ids.DmsDatasetIdType,
                      session: AsyncSession = Depends(db.get_async_session))
                          -> Message:
    """Changes UID of dataset in database"""

    return await update_record(Dataset, specific_id, {'dataset_uid': uid}, '
        dataset_id', session)
```

```
@router.get(path=STATE_BY_TASK_ID_PATH, status_code=status.
    HTTP_200_OK, response_model=List[StateResponse])
async def get_state_by_task_id(task_id: Ids.TaskIdType,
                               session: AsyncSession = Depends(db.
                                   get_async_session)) -> List[StateResponse]:
    """Transfers a list of tasks from the database by its status"""

    stmt = select(State).where(State.task_id == task_id)
    states = (await session.scalars(stmt)).all()
    if not states:
        raise HTTPException(status_code=status.
            HTTP_204_NO_CONTENT, detail=f"There are no states for task:
            {task_id}")
    return [StateResponse.model_validate(record) for record in states]


@router.post(path=CREATE_PATH, status_code=status.
    HTTP_201_CREATED, response_model=StateId)
async def create_state(data: StateCreate, session: AsyncSession = Depends(db.
    get_async_session)) -> StateId:
    """Creates a record of state in database"""

    state_id = await create_record(State, data.model_dump(), 'state_id',
        session)
    return StateId(state_id=state_id)


@router.get(path=ALL_PATH, status_code=status.HTTP_200_OK,
    response_model=List[TaskResponse])
async def get_all_tasks(session: AsyncSession = Depends(db.get_async_session)
    ) -> List[TaskResponse]:
    """Transfers a list of all tasks from the database"""
```

```
    return [TaskResponse.model_validate(record) for record in await
        get_all_records(Task, session)]


@router.get(path=JOINED_TASKS_PATH, status_code=status.
    HTTP_200_OK, response_model=List[JoinedTaskResponse])
async def get_joined_tasks(session: AsyncSession = Depends(db.
    get_async_session)) -> List[JoinedTaskResponse]:
    """Transfers a list of all tasks from the database joined with necessary
        information"""

    input_datasets = aliased(Dataset)
    output_datasets = aliased(Dataset)
    logs_datasets = aliased(Dataset)

    stmt = (
        select(
            Task.task_id,
            Task.workflow_id,
            Template.template_id,
            Task.step_name,
            Template.name.label('template_name'),
            Task.executable,
            Task.args,
            Task.rank,
            Task.device_type,
            Task.mode,
            Task.retries,
            input_datasets.name.label('input_dataset_name'),
            output_datasets.name.label('output_dataset_name'),
            logs_datasets.name.label('logs_datasets_name'),
            Task.status
        )
        .join(Workflow, Task.workflow_id == Workflow.workflow_id)
```

```
        .join(Template, Workflow.template_id == Template.template_id)
        .join(input_datasets, Task.dataset_in_id == input_datasets.dataset_id
            )
        .join(output_datasets, Task.dataset_out_id == output_datasets.
            dataset_id)
        .join(logs_datasets, Task.dataset_log_id == logs_datasets.dataset_id)
    )

    tasks = (await session.execute(stmt)).mappings().all()
    if not tasks:
        raise HTTPException(status_code=status.
            HTTP_204_NO_CONTENT, detail=f"No tasks found")
    return [JoinedTaskResponse.model_validate(record) for record in tasks]


@router.get(path=SPECIFIC_PATH, status_code=status.HTTP_200_OK,
    response_model=TaskResponse)
async def task_response(specific_id: Ids.TaskIdType, session: AsyncSession =
    Depends(db.get_async_session)) -> TaskResponse:
    """Transfers a task from database by its id"""

    return TaskResponse.model_validate(await get_record_by_id(Task,
        specific_id, session))


@router.get(path=STATUS_PATH+TASK_BY_STATUS_PATH, status_code=
    status.HTTP_200_OK, response_model=List[TaskResponse])
async def get_tasks_by_status(status_name: TaskStatus,
                              session: AsyncSession = Depends(db.
                                  get_async_session)) -> List[TaskResponse]:
    """Transfers a list of tasks from the database by its status"""

    stmt = select(Task).where(Task.status == status_name)
    tasks = (await session.scalars(stmt)).all()
```

```
    if not tasks:
        raise HTTPException(status_code=status.
            HTTP_204_NO_CONTENT, detail=f"No tasks found with status
            {status_name}")
    return [TaskResponse.model_validate(record) for record in tasks]


@router.post(path=CREATE_PATH, status_code=status.
    HTTP_201_CREATED, response_model=TaskId)
async def create_task(tsk: TaskCreate, session: AsyncSession = Depends(db.
    get_async_session)) -> TaskId:
    """Creates a record of task in database"""

    task_id = await create_record(Task, tsk.model_dump(), 'task_id', session)
    return TaskId(task_id=task_id)


@router.patch(path=SPECIFIC_PATH+RANK_PATH, status_code=status.
    HTTP_200_OK, response_model=Message)
async def change_rank(specific_id: Ids.TaskIdType, new_rank: int, session:
    AsyncSession = Depends(db.get_async_session)) -> Message:
    """Changes rank of task in database"""

    return await update_record(Task, specific_id, {'rank': new_rank}, 'task_id'
        , session)


@router.patch(path=SPECIFIC_PATH+STATUS_PATH, status_code=status.
    HTTP_200_OK, response_model=Message)
async def change_status(specific_id: Ids.TaskIdType,
                        new_status: TaskStatus,
                        session: AsyncSession = Depends(db.get_async_session)
                            ) -> Message:
    """Changes status of task in database"""
```

```
    return await update_record(Task, specific_id, {'status': new_status}, '
        task_id', session)


@router.get(path=ALL_PATH, status_code=status.HTTP_200_OK,
    response_model=List[TemplateResponse])
async def get_all_templates(session: AsyncSession = Depends(db.
    get_async_session)) -> List[TemplateResponse]:
    """Transfers a list of all templates from the database"""

    return [TemplateResponse.model_validate(record) for record in await
        get_all_records(Template, session)]


@router.get(path=ACTUAL_TEMPLATES_PATH, status_code=status.
    HTTP_200_OK, response_model=List[TemplateActual])
async def get_actual_templates(session: AsyncSession = Depends(db.
    get_async_session)) -> List[TemplateActual]:
    """Transfers a list of actual templates from the database"""

    stmt = select(Template).where(Template.status == TemplateStatus.
        ACTUAL.value)
    templates = (await session.scalars(stmt)).all()
    if not templates:
        raise HTTPException(status_code=status.
            HTTP_204_NO_CONTENT, detail="There are no templates in
            ACTUAL status")
    return [TemplateActual.model_validate(record) for record in templates]


@router.get(path=SPECIFIC_PATH, status_code=status.HTTP_200_OK,
    response_model=TemplateResponse)
async def template_response(specific_id: Ids.TemplateIdType,
```

```
                                session: AsyncSession = Depends(db.
                                    get_async_session)) -> TemplateResponse:
    """Transfers a template from database by its id"""

    return TemplateResponse.model_validate(await get_record_by_id(
        Template, specific_id, session))


@router.post(path=CREATE_PATH, status_code=status.
    HTTP_201_CREATED, response_model=TemplateId)
async def create_template(tmplt: TemplateCreate, session: AsyncSession =
    Depends(db.get_async_session)) -> TemplateId:
    """Creates a record of template in database"""

    template_id = await create_record(Template, tmplt.model_dump(), '
        template_id', session)
    return TemplateId(template_id=template_id)


@router.patch(path=SPECIFIC_PATH+STATUS_PATH, status_code=status.
    HTTP_200_OK, response_model=Message)
async def change_template(specific_id: Ids.TemplateIdType,
                          request: Request,
                          new_status: TemplateStatus | None = None,
                          session: AsyncSession = Depends(db.
                              get_async_session)) -> Message:
    """Changes status of template in database"""

    if not new_status:
        try:
            template_data = await request.json()
            template_status = template_data.get("status")
            if template_status:
                new_status = template_status
```

```
            except (JSONDecodeError, KeyError):
                raise HTTPException(status_code=status.
                    HTTP_400_BAD_REQUEST, detail='Decoding error')

        if new_status not in [TemplateStatus.ACTUAL.value, TemplateStatus.
            ARCHIVED.value]:
            raise HTTPException(status_code=status.
                HTTP_400_BAD_REQUEST, detail='Unavailable status')

        return await update_record(Template, specific_id, {'status': new_status}, '
            template_id', session)


@router.delete(path=SPECIFIC_PATH+DELETE_PATH, status_code=status.
    HTTP_200_OK, response_model=Message)
async def delete_template(specific_id: Ids.TemplateIdType,
                          session: AsyncSession = Depends(db.
                              get_async_session)) -> Message:
    """Deletes the record of template from database"""

    return await delete_record(Template, specific_id, session)


@router.get(path=SPECIFIC_PATH, status_code=status.HTTP_200_OK,
    response_model=UserSessionResponse)
async def session_response(specific_id: Ids.SessionIdType,
                           session: AsyncSession = Depends(db.
                               get_async_session)) -> UserSessionResponse:
    """Transfers a user's session from database by its id"""

    return UserSessionResponse.model_validate(await get_record_by_id(
        UserSession, specific_id, session))
```

```
@router.get(path=USER_BY_SUB_PATH, status_code=status.
    HTTP_200_OK, response_model=List[UserSessionResponse])
async def get_user_session_by_sub(sub: str,
                                  session: AsyncSession = Depends(db.
                                      get_async_session)) -> List[
                                      UserSessionResponse]:
    """Transfers a list of user's sessions from the database by its sub"""

    stmt = select(UserSession).where(UserSession.sub == sub)
    user_sessions = (await session.scalars(stmt)).all()
    if not user_sessions:
        raise HTTPException(status_code=status.
            HTTP_204_NO_CONTENT, detail=f"No user's sessions found
            with sub {sub}")
    return [UserSessionResponse.model_validate(record) for record in
        user_sessions]


@router.post(path=CREATE_PATH, status_code=status.
    HTTP_201_CREATED, response_model=UserSessionId)
async def create_session(user_session: UserSessionCreate,
                         session: AsyncSession = Depends(db.
                             get_async_session)) -> UserSessionId:
    """Creates a record of template in database"""

    user_session = user_session.model_dump()
    user_session["expires_in"] = user_session["expires_in"].replace(tzinfo=
        None)
    session_id = await create_record(UserSession, user_session, 'session_id',
        session)
    return UserSessionId(session_id=session_id)
```

```
@router.put(path=SPECIFIC_PATH+TOKEN_PATH, status_code=status.
    HTTP_200_OK, response_model=Message)
async def update_session(specific_id: Ids.SessionIdType,
                         update_data: UserSessionUpdate,
                         session: AsyncSession = Depends(db.
                             get_async_session)) -> Message:
    """Changes rank of task in database"""

    return await update_record(UserSession,
                               specific_id,
                               updates={
                                   'access_token': update_data.access_token,
                                   'refresh_token': update_data.refresh_token,
                                   'expires_in': update_data.expires_in.replace
                                       (tzinfo=None)
                               },
                               id_field='session_id',
                               session=session)


@router.delete(path=SPECIFIC_PATH+DELETE_PATH, status_code=status.
    HTTP_200_OK, response_model=Message)
async def delete_session(specific_id: Ids.SessionIdType,
                         session: AsyncSession = Depends(db.
                             get_async_session)) -> Message:
    """Deletes the record of user's session from database"""

    return await delete_record(UserSession, specific_id, session)


@router.get(path=ALL_PATH, status_code=status.HTTP_200_OK,
    response_model=List[WorkflowResponse])
async def get_all_workflows(session: AsyncSession = Depends(db.
    get_async_session)) -> List[WorkflowResponse]:
```

```
    """Transfers a list of all workflows from the database"""

    return [WorkflowResponse.model_validate(record) for record in await
        get_all_records(Workflow, session)]


@router.get(path=SPECIFIC_PATH, status_code=status.HTTP_200_OK,
    response_model=WorkflowResponse)
async def workflow_response(specific_id: Ids.WorkflowIdType,
                            session: AsyncSession = Depends(db.
                                get_async_session)) -> WorkflowResponse:
    """Transfers a workflow from database by its id"""

    return WorkflowResponse.model_validate(await get_record_by_id(
        Workflow, specific_id, session))


@router.post(path=CREATE_PATH, status_code=status.HTTP_200_OK,
    response_model=WorkflowId)
async def create_workflow(wflow: WorkflowCreate, session: AsyncSession =
    Depends(db.get_async_session)) -> WorkflowId:
    """Creates a record of workflow in database"""

    workflow_id = await create_record(Workflow, wflow.model_dump(), '
        workflow_id', session)
    return WorkflowId(workflow_id=workflow_id)


@router.patch(path=SPECIFIC_PATH+FINALS_PATH, status_code=status.
    HTTP_200_OK, response_model=WorkflowFinals)
async def inc_finals_processed(specific_id: Ids.WorkflowIdType,
                               session: AsyncSession = Depends(db.
                                   get_async_session)) -> WorkflowFinals:
    """Increments finals_processed field of workflow in database"""
```

```
    try:
        stmt = (
            update(Workflow)
            .where(Workflow.workflow_id == specific_id)
            .values(finals_processed=Workflow.finals_processed + 1)
            .returning(Workflow.finals_processed, Workflow.finals_amount)
        )
        result = await session.execute(stmt)
        await session.commit()

        updated_row = result.fetchone()
        if not updated_row:
            raise HTTPException(status_code=404, detail="Workflow not
                found")

    except IntegrityError as err:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"Integrity error occurred: {str(err.orig)}"
        )

    return WorkflowFinals(finals_processed=updated_row.finals_processed,
        finals_amount=updated_row.finals_amount)


@asynccontextmanager
async def lifespan(_app: FastAPI) -> AsyncGenerator[None, Any]:
    """Lifespan of the app"""

    await db.check_database()
    yield
    await db.dispose()
```

```
app = FastAPI(title="Database", lifespan=lifespan)
app.include_router(template_router.router, tags=["Templates"])
app.include_router(dataset_router.router, tags=["Datasets"])
app.include_router(workflow_router.router, tags=["Workflows"])
app.include_router(task_router.router, tags=["Tasks"])
app.include_router(user_session_router.router, tags=["UserSessions"])
app.include_router(state_router.router, tags=["States"])

app.add_middleware(
    CORSMiddleware,
    allow_origins=[f"http://0.0.0.0:{WORKFLOW_MANAGER_PORT}",
                   f"http://{WORKFLOW_MANAGER_HOST}:{
                       WORKFLOW_MANAGER_PORT}"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


class AsyncRabbitReceiver:
    """Consumer getting message one at a time"""

    def __init__(self, queue_name: str, logger: CustomLogger, max_retries:
        int = 1, retry_delay: int = 0) -> None:
        """Base settings  initialization """

        self._queue_name: str = queue_name
        self._connection: AbstractRobustConnection | None = None
        self._channel: AbstractChannel | None = None
        self._queue: AbstractQueue | None = None
        self._logs: CustomLogger = logger
        self._max_retries: int = max_retries
```

```
        self._retry_delay: int = retry_delay

    async def connect(self) -> None:
        """Connect to RabbitMQ and get the queue."""

        if self._connection and not self._connection.is_closed:
            self._logs.info("Reusing existing RabbitMQ connection.")
            return

        for attempt in range(1, self._max_retries + 1):
            try:
                self._logs.info("Creating RabbitMQ connection...")
                self._connection = await connect_robust(
                    host=RABBITMQ_HOST,
                    port=RABBITMQ_PORT,
                    login=RABBITMQ_USER,
                    password=RABBITMQ_PASSWD,
                    virtualhost=RABBITMQ_VHOST,
                    timeout=5
                )
                if not self._connection.is_closed:
                    self._logs.info("RabbitMQ connection is open and active."
                        )
                else:
                    self._logs.critical("RabbitMQ connection is closed.")
                    raise AMQPConnectionError("Failed to open RabbitMQ
                        connection.")

                self._logs.info("Creating RabbitMQ channel...")
                self._channel = await self._connection.channel()
                self._logs.info("RabbitMQ channel created.")

                self._queue = await self._channel.declare_queue(self.
                    _queue_name, passive=True)
```

```
                self._logs.info(f"Queue '{self._queue_name}' successfully
                    initialized.")
                break
            except (AMQPConnectionError, TypeError) as e:
                self._logs.error(f"Connection attempt failed on attempt {
                    attempt}. Error: {e}", exc_info=True)
                if attempt == self._max_retries:
                    self._logs.critical("Final attempt to connect to
                        RabbitMQ failed.", exc_info=True)
                    raise RuntimeError("Unable to connect to RabbitMQ after
                        multiple attempts.") from e
                await asyncio.sleep(self._retry_delay)

async def consume_message(self) -> Any:
    """Get message body"""

    if not self._queue:
        await self.connect()
        if not self._queue:
            self._logs.error("Failed to connect to RabbitMQ")
            raise RuntimeError("Failed to initialize RabbitMQ queue.")

    async with self._queue.iterator() as queue_iter:
        async for message in queue_iter:
            return message

    self._logs.error("Failed to drop message to RabbitMQ")
    raise RuntimeError("Failed to initialize RabbitMQ queue.")

async def close(self) -> None:
    """Close connection"""

    if self._connection:
        await self._connection.close()
```

```
class TaskGenerator:
    """Generation of tasks"""

    def __init__(self, logger: CustomLogger) -> None:
        self._templates: List[TemplateResponse] = []
        self._logs: CustomLogger = logger
        self._logs.info("Task generator initialized")

    async def __aenter__(self):
        self._client: AsyncClient = AsyncClient()
        self._rabbit_receiver: AsyncRabbitReceiver = AsyncRabbitReceiver(
            DMS_CONSUMER_QUEUE, self._logs)
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        await self._client.aclose()
        await self._rabbit_receiver.close()

    async def post_json(self, path: str, data: dict) -> dict | None:
        try:
            response = await self._client.post(url=f'http://{DB_API_HOST
                }:{DB_API_PORT}'+path, json=data)
            response.raise_for_status()
            return response.json()
        except RequestError as err:
            self._logs.error(f"Request error during POST to {path}: {err}",
                exc_info=True)
        except ValidationError as err:
            self._logs.error(f"Validation error: {err}", exc_info=True)
        return None

    async def create_dataset(self,
```

```
                                name: str,
                                ds_type: DatasetType,
                                uid: Ids.DmsDatasetIdType = None) -> int |
                                    None:
        payload = DatasetCreate(dataset_uid=uid, name=name, type=ds_type)
            .model_dump(mode="json")
        result = await self.post_json(
            path=f"{DATASETS_PREFIX}{CREATE_PATH}",
            data=payload
        )
        return DatasetId.model_validate(result).dataset_id if result else None

    async def initialize_templates(self) -> None:
        """Getting templates"""

        templates_response = await self._client.get(
            f'http://{DB_API_HOST}:{DB_API_PORT}{
                TEMPLATES_PREFIX}{ACTUAL_TEMPLATES_PATH}'
        )
        templates_response.raise_for_status()
        if templates_response.content:
            self._templates = [TemplateResponse.model_validate(tmplt) for
                tmplt in templates_response.json()]

        if not self._templates:
            await asyncio.sleep(10)
            return None

        self._logs.info("Templates successfully initialized.")
        return None

    async def process_tasks(self) -> None:
        """Getting input dataset"""
```

```
        while True:
            await self.initialize_templates()
            if not self._templates:
                continue
            await self._rabbit_receiver.connect()
            try:
                message = await self._rabbit_receiver.consume_message()
                asyncio.create_task(self.process_single_message(message))
            except AMQPConnectionError as err:
                self._logs.error(f"RabbitMQ connection error: {err}", exc_info
                    =True)
                self._logs.info("Reconnecting to RabbitMQ in 5 seconds...")
                await asyncio.sleep(5)

    async def process_single_message(self, dms_message: Any) -> None:
        try:
            inner_dataset = InputDatasetResponse.model_validate(json.loads(
                dms_message.body.decode("utf-8")))
            self._logs.info("Input dataset received.")
            if await self.handle_inner_dataset(inner_dataset) == 0:
                await dms_message.reject(requeue=True)
            else:
                await dms_message.ack()
        except AttributeError as err:
            await dms_message.reject(requeue=True)
            self._logs.error(f"Unexpected error in accessing queue or message:
                {err}", exc_info=True)
        except ValueError as err:
            await dms_message.reject(requeue=True)
            self._logs.error(f"Error in validating model from DMS: {err}",
                exc_info=True)

    async def handle_inner_dataset(self, inner_dataset: InputDatasetResponse)
        -> int:
```

```
"""Processing of input dataset"""

input_dataset_id = await self.create_dataset(inner_dataset.name,
    DatasetType.input, inner_dataset.id)
if not input_dataset_id:
    return 0
self._logs.info(f"Input dataset with name={inner_dataset.name}
    created in database.")

count = 0
for template in self._templates:
    if not re.search(pattern=rf'{template.inner_dataset_mask}', string
        =inner_dataset.name):
        continue

    try:
        if not isinstance(template.description, dict):
            raise TypeError("Template description must be a
                dictionary")
        steps = template.description.get('steps')
    except (ValueError, TypeError) as err:
        self._logs.error(f"Invalid 'description' in template: {err}")
        continue

    count += 1
    create_workflow_db = await self.post_json(
        path=f"{WORKFLOWS_PREFIX}{CREATE_PATH}",
        data=WorkflowCreate(template_id=template.template_id,
            finals_amount=1).model_dump()
    )
    workflow_id = WorkflowId.model_validate(create_workflow_db).
        workflow_id if create_workflow_db else None
    if not workflow_id:
        continue
```

```
            self._logs.info(f"Workflow with workflow_id={workflow_id}
                created in database.")

            for task_number, (step_name, step) in enumerate(steps.items(),
                start=1):
                new_input_id = await self.handle_step(
                    inner_dataset.name,
                    task_number,
                    workflow_id,
                    step_name,
                    step,
                    input_dataset_id,
                    len(steps)
                )
                if new_input_id:
                    input_dataset_id = new_input_id

        return count

    async def handle_step(self,
                          dataset_name: str,
                          task_number: int,
                          workflow_id: Ids.WorkflowIdType,
                          step_name: str,
                          step: dict,
                          input_dataset_id: Ids.DatasetIdType,
                          total_steps: int) -> int | None:
        output_dataset_name = f"{dataset_name}.output.{task_number}"
        log_dataset_name = f"{dataset_name}.log.{task_number}"

        output_dataset_id = await self.create_dataset(output_dataset_name,
            DatasetType.output)
        if not output_dataset_id:
            return None
```

```
        self._logs.info(f"Output dataset with name={output_dataset_name}
            created in database.")

        log_dataset_id = await self.create_dataset(log_dataset_name,
            DatasetType.output)
        if not log_dataset_id:
            return None
        self._logs.info(f"Log dataset with name={log_dataset_name}created
            in database.")

        create_task_db = await self.post_json(
            path=f"{TASKS_PREFIX}{CREATE_PATH}",
            data=TaskCreate(
                workflow_id=workflow_id,
                step_name=step_name,
                executable=step.get('in', {}).get('processing_program'),
                args=step.get('in', {}).get('cable_map'),
                device_type=TaskDeviceType.CPU,
                mode=TaskMode.Map,
                retries=2,
                dataset_in_id=input_dataset_id,
                dataset_out_id=output_dataset_id,
                dataset_log_id=log_dataset_id,
                rank=1,
                is_final=(task_number == total_steps)
            ).model_dump()
        )
        task_id = TaskId.model_validate(create_task_db).task_id if
            create_task_db else None
        if not task_id:
            return None
        self._logs.info(f"Task with task_id={task_id} created in database.")
        return output_dataset_id
```

```
    async def run(self) -> None:
        """Task Generator launch"""

        try:
            await self.process_tasks()
        except Exception as err:
            self._logs.critical(f"[{type(err).__name__}] Critical error during run: {err}", exc_info=True)
            sys.exit(1)


if __name__ == "__main__":
    """Program start"""

    async def main():
        custom_logger = CustomLogger('task_generator')
        async with TaskGenerator(custom_logger) as tg:
            await tg.run()

    asyncio.run(main())


class AsyncRabbitmqServer:
    """Async connection RabbitMQ for publishing messages."""

    def __init__(self, logger: CustomLogger) -> None:
        """Params initialization."""

        self._password: str = parse.quote_plus(RABBITMQ_PASSWD)
        self._url: str = f"amqp://{RABBITMQ_USER}:{self._password}@{
            RABBITMQ_HOST}:{RABBITMQ_PORT}/{
            RABBITMQ_VHOST}"
        self._connection: AbstractRobustConnection | None = None
        self._channel: AbstractChannel | None = None
```

```
        self._logs = logger

    async def connect(self, retries: int = 1, delay: int = 0) -> None:
        """Creation of connection, exchange and queue."""

        if self._connection and not self._connection.is_closed:
            self._logs.info("Reusing existing RabbitMQ connection.")
            return None

        for attempt in range(1, retries + 1):
            try:
                self._logs.info("Creating RabbitMQ connection...")
                self._connection = await connect_robust(self._url, timeout=5)
                self._channel = await self._connection.channel()
                exchange = await self._channel.declare_exchange(
                    RABBITMQ_EXCHANGE, ExchangeType.DIRECT, durable
                    =True)
                queue = await self._channel.declare_queue(f'{
                    RABBITMQ_EXCHANGE}.{
                    WMS_TASKS_ROUTING_KEY}', durable=True)
                await queue.bind(exchange, WMS_TASKS_ROUTING_KEY)
                self._logs.info("RabbitMQ connection created")
                break
            except Exception as err:
                if attempt < retries:
                    await asyncio.sleep(delay)
                else:
                    self._logs.critical(f"Error while attempting to connect: {
                        err}", exc_info=True)
                    raise ConnectionError(f"Failed to connect: {err}")
        return None

    async def publish(self, body: str) -> None:
        """Message publishing"""
```

```
            exchange = await self._channel.get_exchange(
                RABBITMQ_EXCHANGE)
            message = Message(body.encode())
            await exchange.publish(message, routing_key=
                WMS_TASKS_ROUTING_KEY)

        async def close(self) -> None:
            """Connection closing."""

            if self._connection:
                await self._connection.close()


class TaskManager:
    """Managing of tasks"""

    def __init__(self, logger: CustomLogger) -> None:
        self._logs: CustomLogger = logger
        self._logs.info("Task manager initialized")

    async def __aenter__(self):
        self._rabbit_server: AsyncRabbitmqServer = AsyncRabbitmqServer(self.
            _logs)
        await self._rabbit_server.connect()
        self._client: AsyncClient = AsyncClient()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb) -> None:
        await self._client.aclose()
        await self._rabbit_server.close()

    async def request(self, method: str, url: str, data: Any = None) -> dict |
        None:
```

```
        """Make an HTTP request with the specified method."""
        try:
            response = await self._client.request(method, url, json=data)
            response.raise_for_status()
            return response.json() if response.content else None
        except HTTPStatusError as err:
            self._logs.error(f"HTTP error during {method.upper()} request to
                {url}: {err}")
        except RequestError as err:
            self._logs.error(f"Connection error during {method.upper()}
                request to {url}: {err}")
        return None

    async def get_defined_tasks(self) -> List[TaskResponse] | list:
        """Getting tasks with 'DEFINED' status."""

        response = await self.request(
            "get",
            f"http://{DB_API_HOST}:{DB_API_PORT}{TASKS_PREFIX
                }{STATUS_PATH}/{TaskStatus.DEFINED.value}"
        )
        return [TaskResponse.model_validate(result) for result in response] if
            response else []

    async def get_dataset(self, dataset_id: Ids.DatasetIdType) ->
        DatasetResponse | None:
        """Getting specific dataset."""

        response = await self.request(
            "get",
            f"http://{DB_API_HOST}:{DB_API_PORT}{
                DATASETS_PREFIX}/{dataset_id}"
        )
        return DatasetResponse.model_validate(response) if response else None
```

```
async def get_dms_dataset_status(self, dataset_uid: Ids.DmsDatasetIdType
    ) -> DmsDatasetResponse | None:
    """Getting dataset status from Data Management System (DMS)."""

    response = await self.request(
        "get",
        f"http://{DMS_HOST}:{DMS_PORT}{DMS_DATASET_PATH
            }/{dataset_uid}"
    )
    return DmsDatasetResponse.model_validate(response) if response else
        None

async def create_dms_dataset(self, name: str, task_id: Ids.TaskIdType) ->
    DmsDatasetResponse | None:
    """Creating dataset in Data Management System (DMS)."""

    data = DmsDatasetCreate(name=name, metaData={"task_id": task_id
        }).model_dump()
    response = await self.request("post", f"http://{DMS_HOST}:{
        DMS_PORT}{DMS_DATASET_PATH}/", data)
    return DmsDatasetResponse.model_validate(response) if response else
        None

async def update_dataset_uid_in_db(self, dataset_id: Ids.DatasetIdType,
    dataset_uid: Ids.DmsDatasetIdType) -> None:
    """Saving created dataset's UID."""

    await self.request(
        "patch",
        f"http://{DB_API_HOST}:{DB_API_PORT}{
            DATASETS_PREFIX}/{dataset_id}{UID_PATH}?uid={
            dataset_uid}"
    )
```

```
        self._logs.info(f"UID of dataset with ID={dataset_id} updated.")

    async def send_task_to_queue(self, task_data: WmsTaskCreate) -> None:
        """Sending task to RabbitMQ for processing by Workload Management
            System (WMS)."""

        try:
            task_json = task_data.model_dump_json()
            await self._rabbit_server.publish(task_json)
            self._logs.info(f"Task with id = {task_data.task_id} sent to
                RabbitMQ.")
        except RuntimeError as err:
            self._logs.error(f"Failed to send task with id = {task_data.
                task_id} to RabbitMQ: {err}")

    async def update_task_status(self, task_id: Ids.TaskIdType, status:
        TaskStatus) -> None:
        """Updating task's status."""

        await self.request(
            "patch",
            f"http://{DB_API_HOST}:{DB_API_PORT}{TASKS_PREFIX
                }/{task_id}{STATUS_PATH}?new_status={status.value}"
        )
        self._logs.info(f"Status of task with id = {task_id} updated to {
            status.value}.")

    async def process_tasks(self) -> None:
        """Main async cycle to process tasks."""

        while True:
            try:
                defined_tasks = await self.get_defined_tasks()
                if not defined_tasks:
```

```
            await asyncio.sleep(10)
            continue

        for task in defined_tasks:
            input_dataset = await self.get_dataset(task.dataset_in_id)
            if not input_dataset:
                continue

            input_dataset_dms = await self.get_dms_dataset_status(
                input_dataset.dataset_uid)
            if not input_dataset_dms:
                continue

            if input_dataset_dms.statusCode !=
                DmsDatasetStatusCodes.CLOSED:
                continue

            output_dataset_id = task.dataset_out_id
            out_dataset = await self.get_dataset(output_dataset_id)
            if not out_dataset:
                continue

            if not out_dataset.dataset_uid:
                output_dataset_dms = await self.create_dms_dataset(
                    out_dataset.name, task.task_id)
                if not output_dataset_dms:
                    continue
                await self.update_dataset_uid_in_db(
                    output_dataset_id, output_dataset_dms.id)
            else:
                output_dataset_dms = await self.
                    get_dms_dataset_status(out_dataset.dataset_uid)

            log_dataset_id = task.dataset_log_id
```

```
                log_dataset = await self.get_dataset(log_dataset_id)
                if not log_dataset:
                    continue

                if not log_dataset.dataset_uid:
                    log_dataset_dms = await self.create_dms_dataset(
                        log_dataset.name, task.task_id)
                    if not log_dataset_dms:
                        continue
                    await self.update_dataset_uid_in_db(log_dataset_id,
                        log_dataset_dms.id)
                else:
                    log_dataset_dms = await self.get_dms_dataset_status(
                        log_dataset.dataset_uid)

                task_data = WmsTaskCreate(
                    task_id=task.task_id,
                    executable=task.executable,
                    args=task.args,
                    rank=task.rank,
                    device_type=task.device_type,
                    mode=task.mode,
                    retries=task.retries,
                    dataset_in=[input_dataset_dms],
                    dataset_out=[output_dataset_dms],
                    dataset_log=log_dataset_dms
                )

                await self.send_task_to_queue(task_data)
                await self.update_task_status(task.task_id, TaskStatus.
                    RUNNING)
            except ValidationError as err:
                self._logs.error(f"Error in model validation: {err}")
```

```
async def main() -> None:
    """Task Manager launching"""

    logger = CustomLogger("task_manager")
    try:
        async with TaskManager(logger) as task_manager:
            await task_manager.process_tasks()
    except Exception as err:
        logger.critical(f"Found critical error: {err}")

if __name__ == "__main__":
    """Program start"""

    asyncio.run(main())


def is_public_path(path: str) -> bool:
    """Checking path availability without SPD-IAM auth"""
    if path == '/':
        return True

    return any(path.startswith(public_path) for public_path in
        PUBLIC_PATHS)


class AuthMiddleware(BaseHTTPMiddleware):
    """Auth middleware"""

    async def dispatch(self, request: Request, call_next) -> Response:
        if "code" in request.query_params:
            async with SessionService(logger) as service:
                return await service.handle_authorization_code_flow(request)
```

```
        if is_public_path(request.url.path):
            return await call_next(request)

        session_id = request.cookies.get("session_id")
        async with SessionService(logger) as service:
            session = await service.try_get_valid_session(session_id)

        if session:
            request.state.session = session
            return await call_next(request)

        return await TokenManager.redirect_to_iam(session_id)


async def is_superuser(session) -> bool:
    """Detect if user is superuser"""

    return 'sof/operator' in session.groups

async def current_active_user(request: Request) -> UserResponse | None:
    """Get current active user"""

    if hasattr(request.state, "session"):
        session = request.state.session
    else:
        session_id = request.cookies.get("session_id")
        if session_id:
            async with SessionService(logger) as service:
                session = await service.get_valid_session(session_id)
        else:
            return None

    if not session:
        return None
```

```
    user = UserResponse(username=session.username, is_superuser=await
        is_superuser(session))
    return user


class SessionService:
    """Session service for managing users' connections"""

    def __init__(self, logger: CustomLogger):
        """Initialize session service"""

        self._logger: CustomLogger = logger
        self._client: AsyncClient | None = None

    async def __aenter__(self):
        self._client = AsyncClient()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        await self._client.aclose()

    async def try_get_valid_session(self, session_id: str | None) ->
        UserSessionResponse | None:
        """Attempt to get valid session"""

        if not session_id:
            return None
        try:
            return await self.get_valid_session(session_id)
        except Exception as err:
            self._logger.error(f"Session validation failed: {err}")
            return None
```

```
async def get_valid_session(self, session_id: str) -> UserSessionResponse
    | None:
    """Get valid session for user"""

    url = f"http://{DB_API_HOST}:{DB_API_PORT}{
        USER_SESSION_PREFIX}/{session_id}"
    response = await self.request("get", url)

    session = UserSessionResponse.model_validate(response)
    if datetime.now() < session.expires_in:
        return session

    if session.refresh_token:
        try:
            tokens = await TokenManager.refresh_tokens(session.
                refresh_token)
            update_data = UserSessionUpdate(
                access_token=tokens["access_token"],
                refresh_token=tokens.get("refresh_token"),
                expires_in=datetime.now() + timedelta(seconds=tokens.get(
                    "expires_in", 3600)),
            )
            await self.request(
                "put",
                f"{url}{TOKEN_PATH}",
                json=update_data.model_dump(mode="json")
            )
            refreshed = await self.request("get", url)
            return UserSessionResponse.model_validate(refreshed)
        except Exception as err:
            self._logger.error(f"Session deleting failed: {err}")
            await self.request("delete", f"{url}{DELETE_PATH}")
            return None
    else:
```

```
        await self.request("delete", f"{url}{DELETE_PATH}")
        return None

async def request(self, method: str, url: str, expected_statuses=None, **
    kwargs) -> dict | None:
    """HTTP request via client"""

    if expected_statuses is None:
        expected_statuses = [200]
    response = await self._client.request(method, url, **kwargs)
    if response.status_code not in expected_statuses:
        raise HTTPException(status_code=response.status_code, detail=
            response.text)
    if response.content:
        return response.json()
    return None

async def handle_authorization_code_flow(self, request: Request) ->
    JSONResponse | RedirectResponse:
    """Auth via SPD-IAM"""

    code = request.query_params["code"]
    try:
        tokens = await TokenManager.fetch_tokens(code)
        session_id = await self.create_session(tokens)
        response = RedirectResponse(url="/")
        response.set_cookie(
            key="session_id",
            value=str(session_id),
            httponly=True,
            secure=False,
            samesite="lax"
        )
        return response
```

```
        except Exception as err:
            return JSONResponse(status_code=400, content={"detail": str(err)
                })

    async def create_session(self, tokens: dict) -> UUID:
        """Create session in database"""

        await self.delete_existing_sessions(tokens["sub"])

        data = UserSessionCreate(
            access_token=tokens["access_token"],
            refresh_token=tokens.get("refresh_token"),
            expires_in=datetime.now() + timedelta(seconds=tokens.get("
                expires_in", 3600)),
            groups=tokens.get("groups") or [],
            username=tokens.get("username"),
            sub=tokens.get("sub"),
        )

        response = await self.request(
            "post",
            f"http://{DB_API_HOST}:{DB_API_PORT}{
                USER_SESSION_PREFIX}{CREATE_PATH}",
            [201],
            json=data.model_dump(mode="json")
        )

        return UserSessionId.model_validate(response).session_id

    async def delete_existing_sessions(self, sub: str) -> None:
        """Delete existing session from database"""

        url = f"http://{DB_API_HOST}:{DB_API_PORT}{
            USER_SESSION_PREFIX}/sub/{sub}"
```

```
        response = await self.request("get", url, expected_statuses=[200, 204])
        if response:
            sessions = [UserSessionResponse.model_validate(ssn) for ssn in
                response]
            for session in sessions:
                await self.request(
                    "delete",
                    f"http://{DB_API_HOST}:{DB_API_PORT}{
                        USER_SESSION_PREFIX}/{session.session_id}{
                        DELETE_PATH}"
                )


class TokenManager:
    """Manage tokens via SPD-IAM"""

    _client: AsyncClient | None = None

    @classmethod
    async def _get_client(cls) -> AsyncClient:
        """Get async client"""

        if cls._client is None:
            cls._client = AsyncClient(verify=False)
        return cls._client

    @classmethod
    async def fetch_tokens(cls, authorization_code: str) -> dict:
        """Fetch tokens from SPD-IAM"""

        client = await cls._get_client()
        response = await client.post(
            f"{IAM_URL}/token",
            data={
```

```
            "grant_type": "authorization_code",
            "code": authorization_code,
            "redirect_uri": IAM_REDIRECT_URI,
            "client_id": IAM_CLIENT_ID,
            "client_secret": IAM_CLIENT_SECRET,
            "scope": "offline_access␣openid"
        },
    )

    if response.status_code != 200:
        raise HTTPException(
            status_code=response.status_code,
            detail=f"Failed␣to␣obtain␣token.␣Response:␣{response.text}",
        )

    tokens = response.json()
    id_token = tokens.get("id_token")

    if not id_token:
        raise HTTPException(status_code=400, detail="id_token␣is␣
            missing␣in␣response")

    try:
        decoded = jwt.decode(id_token, options={"verify_signature": False
            })
        tokens["sub"] = decoded.get("sub")
        tokens["username"] = decoded.get("preferred_username")
        tokens["groups"] = decoded.get("groups")
    except jwt.DecodeError:
        raise HTTPException(status_code=400, detail="Failed␣to␣decode␣
            id_token")

    return tokens
```

```
@classmethod
async def refresh_tokens(cls, refresh_token: str) -> dict:
    """Refresh outdated tokens"""

    client = await cls._get_client()
    response = await client.post(
        f"{IAM_URL}/token",
        data={
            "grant_type": "refresh_token",
            "refresh_token": refresh_token,
            "client_id": IAM_CLIENT_ID,
            "client_secret": IAM_CLIENT_SECRET,
        },
    )

    if response.status_code == 200:
        return response.json()

    raise HTTPException(
        status_code=response.status_code,
        detail=f"Failed to refresh token. Response: {response.text}",
    )

@classmethod
async def redirect_to_iam(cls, session_id: str | None) ->
    RedirectResponse:
    """Redirect to SPD-IAM for auth"""

    response = RedirectResponse(
        f"{IAM_URL}{IAM_AUTHORIZE_URI}?response_type=code&
            client_id={IAM_CLIENT_ID}&redirect_uri={
            IAM_REDIRECT_URI}"
    )
    if session_id:
```

```
            response.delete_cookie("session_id")
        return response

    @classmethod
    async def aclose(cls) -> None:
        """Close client connection"""

        if cls._client:
            await cls._client.aclose()
            cls._client = None


@router.get(INDEX_PATH, status_code=status.HTTP_200_OK,
    response_class=HTMLResponse)
async def get_index_page(request: Request, user: UserResponse = Depends(
    current_active_user)) -> HTMLResponse:
    """Render index page"""

    return templates.TemplateResponse('landing/index.html', {
        "request": request,
        "user": user
    })


@router.get(ABOUT_PATH, status_code=status.HTTP_200_OK,
    response_class=HTMLResponse)
async def get_about_page(request: Request, user: UserResponse = Depends(
    current_active_user)) -> HTMLResponse:
    """Render page about project"""

    return templates.TemplateResponse('landing/about.html', {
        "request": request,
        "user": user
    })
```

```
@router.get(TEMPLATE_PATH, status_code=status.HTTP_200_OK,
    response_class=Response)
async def get_templates_page(request: Request, user: UserResponse = Depends(
    current_active_user)) -> Response:
    """
    Render page with templates
        Superusers users can:
            - change template statuses;
            - delete loaded templates;
    """

    if not user:
        return JSONResponse(status_code=401, content={'message': '
            Unauthorized'})

    async with httpx.AsyncClient() as client:
        response = await client.request(
            "get",
            f'http://{DB_API_HOST}:{DB_API_PORT}{
                TEMPLATES_PREFIX}{ALL_PATH}'
        )
    response.raise_for_status()
    all_templates = [TmpltResponse.model_validate(tmplt) for tmplt in
        response.json()] if response.content else None

    return templates.TemplateResponse('app/all_templates.html', {
        "request": request,
        "user": user,
        "templates": all_templates
    })
```

```
@router.get(TEMPLATE_PATH+CREATE_PATH, status_code=status.
    HTTP_200_OK, response_class=Response)
async def create_template_page(request: Request, user: UserResponse =
    Depends(current_active_user)) -> Response:
    """
    Render page with creating template
        Automatic template CWL-validation
    """

    if not user:
        return JSONResponse(status_code=401, content={'message': '
            Unauthorized'})

    if user.is_superuser:
        return templates.TemplateResponse('app/create_template.html', {
            "request": request,
            "user": user
        })
    else:
        return JSONResponse(status_code=403, content={'message': 'Superuser
            rights required'})


@router.get(TEMPLATE_PATH+SPECIFIC_PATH, status_code=status.
    HTTP_200_OK, response_class=Response)
async def get_template_page(request: Request,
                            specific_id: int,
                            user: UserResponse = Depends(current_active_user)) ->
                                Response:
    """
    Render page with specific template
        Allow superusers to clone template
    """
```

```
    if not user:
        return JSONResponse(status_code=401, content={'message': '
            Unauthorized'})

    async with httpx.AsyncClient() as client:
        response = await client.request(
            "get",
            f'http://{DB_API_HOST}:{DB_API_PORT}{
                TEMPLATES_PREFIX}/{specific_id}'
        )
    if response.status_code == 404:
        template = None
    else:
        template = TmpltResponse.model_validate(response.json())
        template.description = yaml.dump(template.description, allow_unicode=
            True, default_flow_style=False)

    return templates.TemplateResponse('app/template_description.html', {
        "request": request,
        "user": user,
        "template": template
    })


@router.get(TASK_PATH, status_code=status.HTTP_200_OK, response_class
    =Response)
async def get_tasks_page(request: Request, user: UserResponse = Depends(
    current_active_user)) -> Response:
    """
    Render page with tasks
        Allow to load specific task page for checking its statuses
        Ability to load specific template page
    """
```

```
    if not user:
        return JSONResponse(status_code=401, content={'message': '
            Unauthorized'})

    async with httpx.AsyncClient() as client:
        response = await client.request(
            "get",
            f'http://{DB_API_HOST}:{DB_API_PORT}{TASKS_PREFIX
                }{JOINED_TASKS_PATH}'
        )
    response.raise_for_status()
    tasks = [JoinedTaskResponse.model_validate(task) for task in response.json
        ()] if response.content else None

    return templates.TemplateResponse('app/all_tasks.html', {
        "request": request,
        "user": user,
        "tasks": tasks
    })


@router.get(TASK_PATH+SPECIFIC_PATH, status_code=status.
    HTTP_200_OK, response_class=Response)
async def get_task_page(request: Request, specific_id: int, user: UserResponse
    = Depends(current_active_user)) -> Response:
    """Render page with task's state"""

    if not user:
        return JSONResponse(status_code=401, content={'message': '
            Unauthorized'})

    async with httpx.AsyncClient() as client:
        response = await client.request(
            "get",
```

```
            f'http://{DB_API_HOST}:{DB_API_PORT}{STATE_PREFIX
                }/{specific_id}'
        )
    response.raise_for_status()
    states = [StateResponse.model_validate(state) for state in response.json()]
        if response.content else None

    return templates.TemplateResponse('app/state.html', {
        "request": request,
        "user": user,
        "states": states
    })


@router.post(CWL_PATH)
async def validate_cwl(request: Request, user: UserResponse = Depends(
    current_active_user)) -> dict:
    """Validating CWL-template before adding to database"""

    result = {"valid": False}
    if not user or not user.is_superuser:
        return result

    data = await request.json()
    cwl_text = data.get("cwl_text")

    with tempfile.NamedTemporaryFile(delete=False, suffix=".cwl", mode="w")
        as temp_file:
        temp_file.write(cwl_text)
        temp_file_path = temp_file.name

    command = ["cwltool", "--validate", temp_file_path]
    process = subprocess.run(command, stdout=subprocess.PIPE, stderr=
        subprocess.PIPE, text=True)
```

```
    if process.returncode == 0:
        result["valid"] = True

    return result


@router.get(LOGOUT_PATH)
async def logout(request: Request) -> Response:
    """Logout user from IAM and clear session"""

    session_id = request.cookies.get("session_id")

    if session_id:
        async with httpx.AsyncClient() as client:
            await client.delete(f"http://{DB_API_HOST}:{DB_API_PORT
                }{USER_SESSION_PREFIX}/{session_id}{DELETE_PATH}"
                )

    response = JSONResponse(content={"status": "ok"})
    response.delete_cookie("session_id", path="/")
    return response


@asynccontextmanager
async def lifespan(_app: FastAPI):
    """Lifespan of the app"""
    logger.info("Workflow Manager initialized")
    yield
    await TokenManager.aclose()


app = FastAPI(lifespan=lifespan)
```

```
app.mount(STATIC_PATH, StaticFiles(directory="workflow_manager/static"),
    name="static")

app.include_router(pages.router, prefix=PAGES_PREFIX)
app.include_router(validation.router, prefix=VALIDATION_PREFIX)

app.add_middleware(AuthMiddleware)


class WorkflowScheduler:
    """Schedule workflows"""

    def __init__(self, logger):
        self._logs = logger
        self._logs.info("Workflow scheduler initialized")

    async def __aenter__(self):
        self._rabbit_server: AsyncRabbit = AsyncRabbit(self._logs)
        await self._rabbit_server.connect()
        self._client: AsyncClient = AsyncClient()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb) -> None:
        await self._client.aclose()
        await self._rabbit_server.close()

    async def request(self, method, url, data=None) -> dict | None:
        """Make an HTTP request with the specified method"""
        try:
            response = await self._client.request(method, url, json=data)
            response.raise_for_status()
            return response.json() if response.content else None
        except HTTPStatusError as err:
```

```
            self._logs.error(f"HTTP error during {method.upper()} request to
                {url}: {err}")
        except RequestError as err:
            self._logs.error(f"Connection error during {method.upper()}
                request to {url}: {err}")
        return None

    async def get_running_tasks(self) -> List[TaskResponse] | list:
        """Getting tasks with 'RUNNING' status"""

        response = await self.request(
            "get",
            f"http://{DB_API_HOST}:{DB_API_PORT}{TASKS_PREFIX
                }{STATUS_PATH}/{TaskStatus.RUNNING.value}"
        )
        return [TaskResponse.model_validate(task) for task in response] if
            response else []

    async def get_wms_state(self, task_id: Ids.TaskIdType) ->
        WmsStateResponse:
        """Getting task's status from WMS"""

        response = await self.request(
            "get",
            f"http://{WMS_HOST}:{WMS_PORT}{WMS_STATE_PATH}",
        )

        return WmsStateResponse.model_validate(response) if response else
            None

    async def create_state(self, task_id: Ids.TaskIdType, description: dict) ->
        StateId:
        """Save a state of the task"""
```

```
        data = StateCreate(task_id=task_id, description=description).
            model_dump()

        response = await self.request(
            "post",
            f"http://{DB_API_HOST}:{DB_API_PORT}{STATE_PREFIX
                }{CREATE_PATH}",
            data=data
        )

        return StateId.model_validate(response) if response else None

    async def get_dataset(self, dataset_id: Ids.DatasetIdType) ->
        DatasetResponse:
        """Getting tasks with 'RUNNING' status"""

        response = await self.request(
            "get",
            f"http://{DB_API_HOST}:{DB_API_PORT}{
                DATASETS_PREFIX}/{dataset_id}"
        )

        return DatasetResponse.model_validate(response) if response else None

    async def get_datasets_to_delete(self, workflow_id: Ids.WorkflowIdType)
        -> List[DmsDatasetId]:
        """Getting datasets' uids to delete in DMS"""

        response = await self.request(
            "get",
            f"http://{DB_API_HOST}:{DB_API_PORT}{
                DATASETS_PREFIX}/{workflow_id}{DMS_PATH}{
                DELETE_PATH}"
        )
```

```
        return [DmsDatasetId.model_validate(uid) for uid in response]

    async def delete_dataset(self, uid: DmsDatasetId) -> None:
        """Delete dataset in DMS"""

        try:
            uid_json = uid.model_dump_json()
            await self._rabbit_server.publish(uid_json)
            self._logs.info(f"Message for deletion of dataset with uid = {uid.
                id} published to RabbitMQ.")
        except RuntimeError as err:
            self._logs.error(f"Failed to send deletion message for dataset with
                uid = {uid.id}: {err}")

    async def close_dataset(self, dataset_uid: Ids.DmsDatasetIdType) ->
        None:
        """Close dataset in DMS"""

        await self.request(
            "patch",
            f"http://{DMS_HOST}:{DMS_PORT}{DMS_DATASET_PATH
                }/{dataset_uid}"
            f"?dataset_status={DmsDatasetStatusCodes.CLOSED.value}"
        )

        self._logs.info(f"Dataset with uid = {dataset_uid} closed")

    async def update_task_status(self, task_id: Ids.TaskIdType, status:
        TaskStatus) -> None:
        """Change status of task"""

        if status not in [TaskStatus.FINISHED, TaskStatus.CANCELLED]:
            raise ValueError("Got unable status")
```

```
        else:
            await self.request(
                "patch",
                f"http://{DB_API_HOST}:{DB_API_PORT}{
                    TASKS_PREFIX}/{task_id}{STATUS_PATH}?new_status
                    ={status.value}"
            )
            self._logs.info(f"Status of task with id = {task_id} updated to
                {status.value}.")

    async def update_finals(self, workflow_id: Ids.WorkflowIdType) ->
        WorkflowFinals:
        """Update processed finals amount of workflow"""

        response = await self.request(
            "patch",
            f"http://{DB_API_HOST}:{DB_API_PORT}{
                WORKFLOWS_PREFIX}/{workflow_id}{FINALS_PATH}"
        )

        return WorkflowFinals.model_validate(response) if response else None

    async def manage_state(self):
        while True:
            tasks = await self.get_running_tasks()
            for task in tasks:
                state = await self.get_wms_state(task.task_id)
                await self.create_state(state.task_id, state.metadata)
                if state.status == WmsTaskStatuses.FINISHED:
                    dataset_out = await self.get_dataset(task.dataset_out_id)
                    dataset_log = await self.get_dataset(task.dataset_log_id)
                    await self.close_dataset(dataset_out.dataset_uid)
                    await self.close_dataset(dataset_log.dataset_uid)
```

```
                        await self.update_task_status(task.task_id, TaskStatus.
                            FINISHED)
                        if task.is_final:
                            finals = await self.update_finals(task.workflow_id)
                            if finals.finals_processed >= finals.finals_amount:
                                datasets_to_delete = await self.
                                    get_datasets_to_delete(task.workflow_id)
                                for ds in datasets_to_delete:
                                    await self.delete_dataset(ds)
                    await asyncio.sleep(15)


async def main():
    """Task Manager launching"""

    logger = CustomLogger("workflow_scheduler")
    try:
        async with WorkflowScheduler(logger) as ws:
            await ws.manage_state()
    except Exception as err:
        logger.critical(f"Found critical error: {err}")
        sys.exit(1)

if __name__ == "__main__":
    """Program start."""

    asyncio.run(main())
```