Geekbrains

**AIRFLOW ПРОВАЙДЕР «GUN» В СТИЛЕ TASKFLOW API ДЛЯ ВЗАИМОДЕЙСТВИЯ С HTTP, POSTGRESQL И CLICKHOUSE**

Выпускная квалификационная (дипломная) работа

Разработчик:

Программист Python

группа 4901

Соломатова Маргарита Сергеевна

Санкт-Петербург 2024

**ОГЛАВЛЕНИЕ**

[**Введение** 4](#_Toc156383473)

[**Глава 1. Python в контексте Airflow. Использование Python и Airflow для работы с HTTP-запросами, с PostgreSQL и Clickhouse.** 6](#_Toc156383474)

[**1.1. Основы и преимущества Python в контексте Airflow. Декораторы в Python.** 6](#_Toc156383475)

[**1.2. Apache Airflow как система оркестровки задач.** 10](#_Toc156383476)

[**1.3. Работа Airflow с базами данных PostgreSQL и Clickhouse.** 13](#_Toc156383477)

[**1.4. Особенности использования Python и Airflow для работы с HTTP-запросами.** 14](#_Toc156383478)

[**Глава 2. Кодовая база Airflow провайдера «Gun» для взаимодействия с HTTP, PostgeSQL и Clickhouse. Функционал и возможности.** 17](#_Toc156383479)

[**2.1. Особенности построения кода Airflow провайдера «Gun» в стиле TaskFlow API.** 17](#_Toc156383480)

[**2.2. Описание функционала и возможностей Airflow провайдера «Gun» для взаимодействия с HTTP.** 24](#_Toc156383481)

[**2.3. Дополнительные особенности Airflow провайдера «Gun» для взаимодействия с PostgeSQL и Clickhouse.** 34](#_Toc156383482)

[**Глава 3. Тестирование ряда функций и оптимизация кода Airflow провайдера «Gun» для взаимодействия с HTTP, PostgeSQL и Clickhouse.** 43](#_Toc156383483)

[**3.1. Тестирование Airflow провайдера «Gun» для взаимодействия с HTTP** 43](#_Toc156383484)

[**3.2. Тестирование Airflow провайдера «Gun» для взаимодействия Clickhouse** 45](#_Toc156383485)

[**Заключение** 48](#_Toc156383486)

[**Библиография** 50](#_Toc156383487)

[**Приложение 1** 52](#_Toc156383488)

[**Приложение 2** 74](#_Toc156383489)

[**Приложение 3** 79](#_Toc156383490)

**Введение**

В данной дипломной работе рассматриваются варианты применения Python с использованием Airflow провайдера для выполнения последовательных запросов по протоколу HTTP, при подключении к PostgreSQL, а также в Clickhouse. Данный провайдер выполнен в стиле TaskFlow API с использованием декораторов, что позволяет простым и понятным способом произвести последовательно операции для работы с данными.

Использование данного провайдера обусловлено необходимостью использовать Airflow для работы с данными и интеграцией с внешними базами данных и другими системами в стиле TaskFlow API, что в свою очередь вызвано потребностью перехода на отечественные программы и программы и компоненты с открытым исходным кодом.

**Объектом** данной работы является язык Python с использованием Airflow для выполнения последовательных запросов в виде декораторов.

**Предметом** становятся примеры и варианты использования данного провайдера для работы с данными.

**Целью** данной работы является изучение возможностей декорирования в Python в работа с Airflow в качестве провайдера для выполнения последовательных запросов HTTP, а также для работы с данными в PostgreSQL и ClickHouse. Кроме того, рассмотрение преимуществ применения данного провайдера для работы с данными, а также при переходе на ПО с открытым исходным кодом.

Исходя из указанной цели, в работе ставятся следующие **задачи**:

1. изучить существующие в настоящее время варианты использования Airflow для работы с DAG’ами;
2. проанализировать необходимость и удобство использования различных решений для работы с данными;
3. изучить возможности взаимодействия Airflow с различными системами управления базами данных (СУБД);
4. изучить преимущества декораторов в языке Python для работы с последовательными операциями;
5. разработать Airflow провайдер с учетом актуальных задач и направлений на языке Python для работы с данными по HTTP, с PostgreSQL, Clickhouse и в дальнейшем другими СУБД.

Занимаясь разработкой данной программы, мной были использованы не только устоявшиеся лучшие практики при написании кода на Python, а также современные подходы в структурировании и архитектуре программы.

Материалы для исследования были отобраны из открытых источников, таких, как Wikipedia, Github, Habr и другие общеизвестные и открытые ресурсы.

Данная дипломная работа состоит из нескольких частей, в которых последовательно представлены теоретическая основа, практическая составляющая и тестирование кода.

**В первой главе** рассматривается основы и преимущества Python, теоретическая база работы с Airflow, а также главные аспекты работы с данными посредством HTTP-запросов, с PostgreSQL и Clickhouse в контексте языка Python как средства для написания кроссплатформенного кода.

**Во второй главе** приводится кодовая база с её анализом и описанием функционала и возможностей.

**Третья глава** представляет собой тестирование ряда функций и посвящена оптимизации кода.

**В заключении** излагаются основные выводы и результаты, полученные в ходе работы.

**Глава 1. Python в контексте Airflow. Использование Python и Airflow для работы с HTTP-запросами, с PostgreSQL и Clickhouse.**

**1.1. Основы и преимущества Python в контексте Airflow. Декораторы в Python.**

Python – это язык программирования, который был разработан Гвидо ван Россумом в 1991 году и впервые выпущен в 1994 году. За 20 лет он стал одним из самых востребованных языков программирования благодаря своей простоте и понятному синтаксису, что сделало его легким для изучения и использования. Однако наряду с простотой этот язык также обладает и рядом других преимуществ:

* Большая стандартная библиотека: имеет богатую стандартную библиотеку, которая включает в себя различные модули для работы с файлами, базами данных, сетью, графикой и многим другим.
* Объектно-ориентированное программирование: поддерживает принципы объектно-ориентированного программирования (ООП), благодаря чему можно создавать модульные поддерживаемые программы, которые легко поддерживать.
* Динамическая типизация: типы переменных в Python определяются во время выполнения программы, а не во время компиляции. Это делает код более гибким и простым в написании.
* Интерпретируемость: является интерпретируемым языком, что означает, что программы на Python не нужно компилировать перед исполнением. Это ускоряет процесс разработки и отладки.
* Широкая область применения: Python используется в различных областях, таких как веб-разработка, научные исследования, data science, машинное обучение и другие.
* Открытый исходный код и большое сообщество: у Python есть обширное и активное сообщество, поскольку открытый исходный код позволяет постоянно улучшать и расширять возможности языка, что делает его более гибким и адаптируемым к изменяющимся требованиям.

Поскольку язык Python имеет довольно широкий спектр возможностей и преимуществ, легок в освоении, имеет открытый исходных код, он и был выбран разработчиками Apache Airflow в качестве оптимального варианта, который удовлетворяет современным требованиям в программировании. Данные факторы являются, пожалуй, самыми важными в мировой тенденции к цифровизации и информатизации, поскольку стоит острая необходимость привлечь как можно большее число людей в IT-отрасль. С набором популярности ИИ мы сталкиваемся с потребностью более глубокого понимания человеком современные технологий, приобретения узкоспециализированных знаний и умений, поэтому изучения прикладных навыков, таких как программирование, становится не привилегией, а базовой потребностью. Именно поэтому Python получил широкое распространение и область применения.

Разработчики Airflow наряду с вышеизложенными плюсами руководствовались также и особенностями, которые предлагает Python, а именно:

1. Кроссплатформенность.
2. Высокая производительность.
3. Легкость использования.
4. Хорошая поддержка для работы с базами данных.
5. Поддержка для работы с сетевыми протоколами.

Как уже было сказано ранее, Python имеет множество библиотек и выбрать из них наиболее подходящие, порой, представляет собой нетривиальную задачу. Однако есть некоторые стандартные библиотеки, которые зарекомендовали себя на протяжении долгого времени и повсеместно используются разработчиками. В контексте Airflow наиболее интересными и полезными библиотеками являются:

* Flask: микрофреймворк для создания веб-приложений на языке Python.
* SQLAlchemy: объектно-реляционное отображение (ORM) для работы с SQL.
* Celery: асинхронное выполнение задач с поддержкой оркестрации.
* Redis: хранилище данных типа “ключ-значение” для Celery.
* PostgreSQL: система управления базами данных.
* Gunicorn: сервер WSGI.
* Nginx: веб-сервер и обратный прокси.
* Docker: система контейнеризации.
* Jenkins: система непрерывной интеграции.

В Python есть несколько библиотек для работы с HTTP, PostgreSQL, и Cickhouse. Эти библиотеки позволяют подключаться к базам данных, выполнять SQL-запросы или HTTP-запросы, работать с данными и многое другое. Далее мы рассмотрим некоторые из них, которые представляют особый интерес в рамках данного проекта.

Для понимания сущности и уникальной особенности данного Airflow провайдера необходимо также раскрыть тему декораторов в Python. **Декораторы** в Python – это механизм, который позволяет добавлять дополнительное поведение или изменять поведение функций без изменения их исходного кода. Декораторы позволяют применять определенные функции или инструкции до или после выполнения определенной функции. Они могут быть использованы для добавления различных видов поведения к функциям. Вот некоторые из наиболее распространенных применений декораторов в Python:

1. Логирование: Декораторы можно использовать для добавления функциональности логирования к функциям. Это может быть полезно для отладки и мониторинга кода.

Пример:

def log\_decorator(func):

def wrapper(\*args, \*\*kwargs):

print("Entering function")

result = func(\*args, \*\*kwargs)

print("Leaving function")

return result

return wrapper

@log\_decorator

def my\_function():

print("Inside my\_function")

my\_function()

В этом примере была определена функция log\_decorator, которая в качестве аргумента принимает другую функцию. Затем мы создаем новую функцию wrapped\_func, которая вызывает исходную функцию и добавляет вывод в лог.

Декоратор используется с символом «эт» @ перед именем функции, чтобы показать, что нужно применить функцию-декоратор к данной функции.

1. Кэширование: Декораторы также могут использоваться для кэширования результатов функции, чтобы уменьшить время выполнения.

Пример:

from functools import lru\_cache

@lru\_cache(maxsize=128)

def factorial(n):

if n <= 1:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5))

1. Валидация входных данных: Декораторы могут использоваться для проверки входных данных перед вызовом функции.

Пример:

def check\_input(function):

def checked\_input(\*args, \*\*kwargs):

if args[0] < 0:

raise ValueError("Input cannot be negative")

else:

return function(\*args, \*\*kwargs)

return checked\_input

@check\_input

def square(number):

return number \*\* 2

try:

square(-5)

except ValueError as e:

print(e)

else:

pass

1. Параллельное выполнение: Декораторы могут также использоваться для параллельного выполнения функций.  
   Пример:

import threading

from time import sleep

def parallelize(function):

def decorated\_function(\*args, \*\*kwargs):

thread = threading.Thread(target=function, args=args, kwargs=kwargs)

thread.start()

thread.join()

return decorated\_function

@parallelize

def slow\_function(n):

Итак, декораторы в Python позволяют расширять и изменять поведение вызываемых объектов (функций, методов и классов) без постоянного изменения самого вызываемого объекта. В данном провайдере декораторы используются для упрощения кода внутри Python функций и прозрачности бизнес-логики внутри исполняемой функции.

**1.2. Apache Airflow как система оркестровки задач.**

«Apache Airflow – открытое программное обеспечение (ПО) для создания, выполнения, мониторинга и оркестровки потоков операций по обработке данных.» То есть, Airflow представляет собой инструмент с открытым-исходным кодом для управления рабочими процессами и автоматизации задач. Был разработан в октябре 2014 года и в настоящее время является уникальным продуктом для работы с данными.

Одной из главных особенностей Airflow является то, что он написан на языке Python и представляет собой библиотеку (ну или набор библиотек) для разработки, планирования и мониторинга рабочих процессов. То есть все рабочие процессы в Airflow представлены в виде кода на Python. Благодаря этому Airflow имеет следующие преимущества:

* **Динамичность**: конвейеры Airflow настраиваются в виде кода Python, что позволяет динамически генерировать конвейеры   
  (pipeline).
* **Модульность**: Airflow предоставляет модульную архитектуру, что позволяет разработчикам добавлять новые возможности через плагины и расширения.
* **Управление ресурсами**: Airflow позволяет управлять ресурсами и контролировать их использование, обеспечивая оптимизацию использования ресурсов.
* **Управление ресурсами**: Airflow позволяет управлять ресурсами и контролировать их использование, обеспечивая оптимизацию использования ресурсов.
* **Расширяемость**: фреймворк Airflow™ содержит операторы для подключения к множеству технологий. Все компоненты Airflow расширяемы для простой настройки под ваше окружение.
* **Гибкость**: параметризация рабочих процессов встроена с использованием шаблонного движка Jinja.

Основной сущностью в AirFlow является DAG – (направленный ациклический граф, DAG – directed acyclic graph) – он описывает процессы обработки данных и позволяет объединять задачи, определяя правила их совместной работы.

Преимущества использования DAG’ов:

* **Зависимости в DAG**гарантируют, что ваши задачи обработки данных каждый раз выполняются в одном и том же порядке, это создает надежный процесс для вашей повседневной инфраструктуры обработки данных.
* **Визуализация в DAG**позволяет графически представить зависимости в пользовательском интерфейсе Airflow.
* Поскольку **каждый путь в DAG является линейным**, легко разрабатывать и тестировать конвейеры данных на соответствие ожидаемым результатами.

Логика процесса заключается в том, что DAG объединяет в себе набор связанных задач (task), которые могут быть выполнены параллельно или последовательно. Особенностью графа является отсутствие направленных циклов, но в нем могут быть «параллельные» пути, выходящие из одного узла и разными путями приходящие в конечный узел. DAG может запускаться тремя способами:

* 1. вручную или через API,
  2. по определенному расписанию, которое определено как часть DAG’а (например, каждый час или ежедневно),
  3. по событию (например, появлению файла в определенной директории).

Одной из удобных возможностей в Airflow является использование декораторов для определения зависимостей между задачами и для управления их выполнением. Например, можно использовать декоратор «depends\_on\_past» для определения зависимости между задачами, которые должны выполняться в определенном порядке.

Планировщик задач в Airflow построен на Python-библиотеке Celery, которая позволяет организовать очередь, а также исполнение задач асинхронным и распределенным методами. Со стороны Airflow все задачи делятся на пулы, которые создаются вручную и ими можно управлять через web-интерфейс. Пулы используются для управления ресурсами, доступными для выполнения задач, и позволяют, например, ограничить количество ресурсов, таких как CPU, память, диски и т. д. Это помогает обеспечить эффективное использование ресурсов и предотвратить перегрузку системы.

Airflow также предоставляет TaskFlow API, с помощью которого разработчики могут создавать задачи и управлять их выполнением. TaskFlow API позволяет создавать задачи с различными параметрами и зависимостями, а также управлять их выполнением и мониторингом.

Это может быть полезно для выполнения специализированных задач или для интеграции Airflow с нестандартными инструментами и системами. В данной работе представлен Airflow провайдер для работы с HTTP- запросами, для взаимодействия с такими системами управления базами данных (СУБД), как PostgreSQL и Clickhouse. Однако принципиально важной особенностью данного Airflow провайдера является возможность обработки нескольких запросов внутри одного таска посредством оборачивания запросов внутрь декораторов с последующей обработкой бизнес-логики внутри Python функции. Во второй главе будет подробно рассмотрена данная особенность.

**1.3. Работа Airflow с базами данных PostgreSQL и Clickhouse.**

Airflow взаимодействует с различными СУБД посредством библиотек Python. Основным преимуществом PostgreSQL и Clickhouse является открытый исходный код, что опять-таки представляет особый интерес для Российского бизнеса. Поэтому также хочу подчеркнуть актуальность данной работы. Основное отличие Clickhouse от других БД является ее колоночная структура.

Итак, например, для PostgreSQL используется библиотека **psycopg2**. Она предоставляет интерфейс для выполнения SQL-запросов, управления транзакциями, работы с курсорами, а также предоставляет возможности для взаимодействия с PostgreSQL через различные протоколы и другие возможности. psycopg2 имеет хорошую документацию, активное сообщество пользователей и является одной из наиболее популярных и широко используемых библиотек для работы с PostgreSQL.

Для работы с ClickHouse в Airflow используется библиотека**clickhouse-driver.** Отличительной особенностью ClickHouse от других БД является ее колоночная структура, это позволяет ClickHouse обрабатывать большие объемы данных с высокой скоростью и эффективностью. Библиотека clickhouse-driver поддерживает различные протоколы и форматы данных, а также позволяет работать с курсорами для обработки больших объёмов данных. Кроме того, она имеет хорошую производительность и оптимизирована для работы с большими объёмами данных. Более подробно данная тема будет рассмотрена на примере кода во 2 главе.

**1.4. Особенности использования Python и Airflow для работы с HTTP-запросами.**

HTTP (англ. HyperText Transfer Protocol – «протокол передачи гипертекста») – сетевой протокол прикладного уровня, который используют ещё с 1992 года. HTTP умеет передавать различные форматы файлов – например, видео, аудио, текст. Но при этом состоит только из текста, что крайне удобно при работе с базами данных.

HTTP запрос состоит из нескольких частей: метод, URI, заголовки, тело запроса и статус-код. Метод указывает, какое действие нужно выполнить (например, GET или POST). URI – это адрес ресурса, к которому обращается запрос. Заголовки содержат информацию о запросе, такую как тип контента, пользовательские агенты и т. д. Тело запроса содержит данные, которые отправляются на сервер (например, при использовании метода POST). Статус-код – это код, который сообщает об успешности запроса.

Для работы с HTTP в Python имеется ряд библиотек. В данной работе были использованы библиотеки **urllib** и **http**, которые являются стандартной частью языка, что означает, что они доступны для всех пользователей Python и не требуют установки дополнительных зависимостей. Далее будут рассмотрены детали использования данных библиотек.

Библиотека urllib.request, которая входит в библиотеку urllib является одной из наиболее часто используемых библиотек для работы с протоколами HTTP и HTTPS в Python. Она предоставляет широкий спектр функций для обработки запросов и ответов, а также позволяет работать с различными форматами данных, такими как JSON, XML и другие. Одним из главных преимуществ использования библиотеки urllib.request является ее простота и удобство использования, так как она имеет понятный и простой интерфейс.

Еще одним важным преимуществом использования библиотеки urllib.request является то, что она поддерживает различные форматы данных, что позволяет работать с данными в разных форматах без необходимости использования дополнительных библиотек. Это делает ее идеальной для провайдера, поскольку Airlfow также может работать с различными типами данных.

В Airflow нет встроенного HTTP-интерфейса, поэтому для формирования HTTP-запроса как правило используют это микрофреймворк Flask, который выступает одной из его зависимостей, для создания своего собственного HTTP-API. Flask позволяет создавать веб-приложения на языке Python и имеет простой в использовании синтаксис. Чтобы использовать Flask в Airflow, вам нужно создать новый оператор Python и использовать его для вызова функций из Flask.

Важно здесь то, что Airflow не может самостоятельно использовать HTTP-запросы и работа с данными в таком случае должна разбиваться на несколько тасков, что в свою очередь может создать ряд проблем и нарушить логику процесса работы с данными.

Поэтому в данной работе одной из задач Airflow провайдера является выполнение HTTP-запросов в стиле Fluent API, используя цепочку декораторов. Декорируемый метод, который выполняет HTTP-запрос, должен быть объявлен с определенными аргументами и возвращаемым типом, соответствующими протоколу HTTP. Данное решения обусловлено потребностью выполнять однотипные операции при работе с данными и служит для упрощения кода и его лучшей читабельности. А также благодаря конвейерной обработке мы получаем маневренный и масштабируемый вариант для работы с данными. Примеры и логика использования будут рассмотрены более детально во 2 главе.

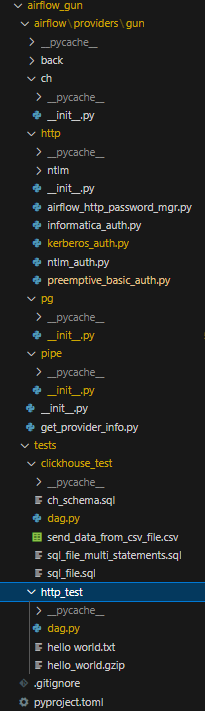
Итак, данный Airlfow провайдер позволяет конвейерным способом выполнять последовательно различные HTTP-запросы, пробрасывая результат выполнения декораторов в итоговую Python функцию, где будет производиться обработка бизнес-логики, без «мусора» в виде ряда HTTP-методов GET/POST/PUT/DELETE.

Таким образом, функционал языка Python позволяет нам упаковать ряд операций в один таск в Airflow и решить несколько задач при работе с базами данных, таких как, например, PostgreSQL или Clickhouse, а также при работе с HTTP-запросами. Благодаря декораторам мы можем вынести часть типичного функционала за рамки основной бизнес-логики функции. В качестве типичных процессов выступают формирование HTTP-запроса, установка соединения с СУБД, обработка ошибок в случае неудачного подключения к базе, повторное отправка запроса, упаковка информации или построчное копирование, например, в CSV-файл и т. д. То есть все те часто повторяющиеся задачи, которые встречаются при работе с данными, выносятся на уровень декораторов. Это позволяет нам выполнять несколько операций в рамках одного таска и использовать результаты их работы внутри одной функции. И тем самым код становится более понятным, масштабируемым и даже «красивым».

**Глава 2. Кодовая база Airflow провайдера «Gun» для взаимодействия с HTTP, PostgeSQL и Clickhouse. Функционал и возможности.**

**2.1. Особенности построения кода Airflow провайдера «Gun» в стиле TaskFlow API.**

Ознакомимся со структурой проекта, см. изображение 1:



Изображение 1

Как видно из структуры проекта, в провайдере используется модульная архитектура, где каждый вариант реализации находится в отдельной папке, а также выделена отдельная директория pipe, где описан алгоритм работы с последовательностью команд с конвейерной передачей данных.

Прежде чем приступить к описанию кода провайдера, рассмотрим стандартную последовательность действий пользователя в Airflow:

1. Пользователь создает DAG (Directed Acyclic Graph) — это граф, описывающий последовательность задач, которые необходимо выполнить. DAG состоит из тасков (tasks), которые представляют собой отдельные задачи или подзадачи.
2. Затем пользователь определяет параметры DAG, такие как время запуска, время выполнения, зависимости между операторами и т. д.
3. После этого DAG отправляется на выполнение. Airflow автоматически распределяет задачи между рабочими узлами и следит за их выполнением.
4. Когда задача выполнена, Airflow отправляет уведомление о результате выполнения. Если задача завершилась неудачно, Airflow может повторить попытку или уведомить пользователя об ошибке.

Как уже было сказано ранее, провайдер работает с тасками, для которых в провайдере был выделен отдельный класс PipeTask:

class PipeTask:

    def \_\_init\_\_(self, context\_key: str):

        self.template\_fields: Sequence[str] = ()

        self.context\_key = context\_key

        self.template\_render = lambda val, context: val

    def \_\_call\_\_(self, context):

        pass

    def set\_template\_fields(self, template\_fields: Sequence[str]):

        self.template\_fields = template\_fields

    def set\_template\_render(self, template\_render: Callable):

        self.template\_render = template\_render

    def render\_template\_fields(self, context):

        for name in self.template\_fields:

            val = getattr(self, name)

            val = self.template\_render(val, context)

            setattr(self, name, val)

Класс PipeTask представляет собой таск для использования в конвейере обработки данных. Основная цель класса состоит в том, чтобы иметь возможность применять шаблоны template\_render к полям таска, используя контекст выполнения context\_key. Это полезно, когда необходимо динамически изменять или настраивать задачи в конвейере обработки данных в зависимости от внешних факторов или переменных контекста.

class PipeTaskBuilder(PipeTask):

def \_\_init\_\_(

self,

context\_key: str,

last\_module: Callable,

):

super().\_\_init\_\_(context\_key)

random\_suffix = "".join(random.choices(string.ascii\_uppercase, k=5))

self.stack\_key = f"{last\_module.\_\_name\_\_}\_stack\_{random\_suffix}"

self.modules = []

self.last\_module = last\_module

def \_\_call\_\_(self, \*args, \*\*kwargs):

context = args[0]

for module in self.modules:

if not context.get(module.context\_key):

context[module.context\_key] = {}

module(context)

return self.last\_module(\*args, \*\*kwargs)

В качестве потомка выступает класс PipeTaskBuilder, который в свою очередь служит для построения цепочки тасков (pipe task). Данный класс добавляет модули в цепочку и затем, пробегаясь циклом for и принимая request в качестве аргумента, последовательно выполняет их в заданном порядке. То есть response очередного таска становится request’ом для последующего и так далее. В итоге работы класса мы получаем на выходе результат работы последнего таска.

Для того, чтобы задать context\_key, используется функция-декоратор pipe\_switch(context\_key):

def pipe\_switch(context\_key: str):

def wrapper(pipe: PipeTaskBuilder):

pipe.context\_key = context\_key

return pipe

return wrapper

Далее используется класс-заглушка AirflowContextResolver и классы XComPull и JinjaRender, которые его реализуют. Это необходимо для извлечения значений XCom из контекста выполнения Airflow, по указанным таскам и ключу, а также для рендеринга шаблонов с использованием тасков и контекста, переданных в экземпляре класса JinjaRender.

class AirflowContextResolver:

def \_\_init\_\_(self):

pass

def \_\_call\_\_(self, context: AirflowContext):

raise NotImplementedError(f"{ \_\_name\_\_ } not implement \_\_call\_\_")

class XComPull(AirflowContextResolver):

def \_\_init\_\_(

self,

task\_ids: Optional[Union[str, Iterable[str]]] = None,

key: str = XCOM\_RETURN\_KEY,

):

self.task\_ids = task\_ids

self.key = key

self.task\_instance\_key = "task\_instance"

def \_\_call\_\_(self, context: AirflowContext) -> Any:

ti: TaskInstance = context[self.task\_instance\_key]

val = ti.xcom\_pull(key=self.key, task\_ids=self.task\_ids)

return val

class JinjaRender(AirflowContextResolver):

def \_\_init\_\_(

self,

val: Any,

):

self.val = val

self.task\_instance\_key = "task"

def \_\_call\_\_(self, context: AirflowContext) -> Any:

task: TaskInstance = context[self.task\_instance\_key]

jinja\_env = task.get\_template\_env()

val = task.render\_template(

self.val,

context,

jinja\_env,

set(),

)

return val

Далее в коде присутствуют классы PipeAirflowTaskBuilder(PipeTaskBuilder), PipeAirflowCallbackBuilder(PipeAirflowTaskBuilder), PipeAirflowCallbackArgsBuilder(PipeAirflowCallbackBuilder), которые расширяют функционал классов-родителей, переопределяя функцию вызова и некоторые аргументы внутри класса. Это необходимо для осуществления перестановки и пробрасывания аргументов и результата исполнения функций класса в следующий таск.

class PipeAirflowTaskBuilder(PipeTaskBuilder):

def \_\_init\_\_(

self,

context\_key: str,

last\_module: Callable,

):

super().\_\_init\_\_(context\_key, last\_module)

def render\_template(val, context):

if isinstance(val, XComPull):

val = val(context)

task = context["task"]

jinja\_env = task.get\_template\_env()

val = task.render\_template(

val,

context,

jinja\_env,

set(),

)

return val

self.set\_template\_render(render\_template)

self.last\_module = last\_module

self.\_\_name\_\_ = last\_module.\_\_name\_\_

self.\_\_doc\_\_ = last\_module.\_\_doc\_\_

self.\_\_annotations\_\_ = last\_module.\_\_annotations\_\_

def \_\_call\_\_(self, \*args, \*\*kwargs):

context = get\_current\_context()

self.render\_template\_fields(context)

with ExitStack() as stack:

context[self.stack\_key] = stack

for module in self.modules:

if not context.get(module.context\_key):

context[module.context\_key] = {}

module(context)

args = (context,) + args

return self.last\_module(\*args)

class PipeAirflowCallbackBuilder(PipeAirflowTaskBuilder):

def \_\_init\_\_(

self,

context\_key: str,

last\_module: Callable,

):

super().\_\_init\_\_(context\_key, last\_module)

def \_\_call\_\_(self, context):

random\_suffix = "".join(random.choices(string.ascii\_uppercase, k=5))

self.stack\_key = f"{self.last\_module.\_\_name\_\_}\_stack\_{random\_suffix}"

self.render\_template\_fields(context)

with ExitStack() as stack:

context[self.stack\_key] = stack

for module in self.modules:

if not context.get(module.context\_key):

context[module.context\_key] = {}

module(context)

return self.last\_module(context)

class PipeAirflowCallbackArgsBuilder(PipeAirflowCallbackBuilder):

def \_\_init\_\_(

self,

context\_key: str,

last\_module: Callable,

):

super().\_\_init\_\_(context\_key, last\_module)

def \_\_call\_\_(self, \*args, \*\*kwargs):

def wrapper\_last\_module(context):

args\_ = self.template\_render(args, context)

kwargs\_ = self.template\_render(kwargs, context)

return self.last\_module(context, \*args\_, \*\*kwargs\_)

wrapper\_last\_module.\_\_name\_\_ = self.last\_module.\_\_name\_\_

wrapper\_last\_module.\_\_doc\_\_ = self.last\_module.\_\_doc\_\_

wrapper\_last\_module.\_\_annotations\_\_ = self.last\_module.\_\_annotations\_\_

wrapper = PipeAirflowCallbackBuilder(

context\_key=self.context\_key,

last\_module=wrapper\_last\_module,

)

wrapper.modules = self.modules

return wrapper

В провайдере используется несколько однотипных декораторов:

Все три декоратора pipe\_airflow\_task, pipe\_airflow\_callback и pipe\_airflow\_callback\_args позволяют упростить процесс создания и настройки коллбэков (callback) в рамках Apache Airflow, предоставляя удобный и гибкий способ определения и использования функционала.

def pipe\_airflow\_task(context\_key: str = "pipe"):

def wrapper(module):

builder = PipeAirflowTaskBuilder(context\_key, module)

return builder

return wrapper

def pipe\_airflow\_callback(context\_key: str = "pipe"):

def wrapper(module):

builder = PipeAirflowCallbackBuilder(context\_key, module)

return builder

return wrapper

def pipe\_airflow\_callback\_args(context\_key: str = "pipe"):

def wrapper(module):

builder = PipeAirflowCallbackArgsBuilder(context\_key, module)

return builder

return wrapper

Далее мы собираем все методы в класс PipeDecoratorCollection для конвейерной обработки в Airflow. С помощью декоратора jinja\_render делаем рендеринг шаблона с использованием контекста.

class PipeDecoratorCollection:

switch = staticmethod(pipe\_switch)

airflow\_task = staticmethod(pipe\_airflow\_task)

airflow\_callback = staticmethod(pipe\_airflow\_callback\_args)

\_\_call\_\_: Any = airflow\_task

pipe = PipeDecoratorCollection()

def jinja\_render(decorated\_function):

def wrapper(\*args, \*\*context):

res = decorated\_function(\*args)

task = context["task"]

jinja\_env = task.get\_template\_env()

res = task.render\_template(

res,

context,

jinja\_env,

set(),

)

return res

wrapper.\_\_name\_\_ = decorated\_function.\_\_name\_\_

wrapper.\_\_doc\_\_ = decorated\_function.\_\_doc\_\_

wrapper.\_\_annotations\_\_ = decorated\_function.\_\_annotations\_\_

return wrapper

Таким образом, данная часть является настройкой этапов для обработки запросов конвейерным методом.

**2.2. Описание функционала и возможностей Airflow провайдера «Gun» для взаимодействия с HTTP.**

В модуле HTTP приведен код, позволяющий выполнить HTTP-запрос, написав его в стиле Fluent API. Важно подчеркнуть, что для этого используется ряд полей (переменных), которые являются типичными для выполнения HTTP-запроса. Собственно, именно поэтому данный провайдер и представляет особый интерес, поскольку позволяет последовательно пройтись по всем частям HTTP-запроса и сформировать единый response, который в дальнейшем будет проброшен к главную Python функцию, которая содержит бизнес-логику.

В данном модуле представлена как основная логика, так и дополнительный функционал для работы с авторизацией по NTLM, с Informatica, Kerberos, Preemptive, а также по Airflow-HTTP (см. Изображение 1).

Описание модуля начинается с класса HttpReq (подклассом PipeTask) который предназначен для выполнения HTTP-запросов.

class HttpReq(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

url,

method,

timeout,

ssl\_verify,

debuglevel,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(("url", "method", "timeout", "ssl\_verify"))

super().set\_template\_render(template\_render)

self.url = url

self.method = method

self.timeout = timeout

self.ssl\_verify = ssl\_verify

self.debuglevel = debuglevel

self.request\_key = "req"

self.timeout\_key = "timeout"

self.modules\_key = "modules"

self.ssl\_verify\_key = "ssl\_verify"

self.debuglevel\_key = "debuglevel"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

req = Request(

url=self.url,

method=self.method,

)

share = context[self.context\_key]

share[self.request\_key] = req

share[self.timeout\_key] = self.timeout

share[self.modules\_key] = []

share[self.ssl\_verify\_key] = self.ssl\_verify

share[self.debuglevel\_key] = self.debuglevel

Все декораторы объявлены с префиксом @http\_

Цепочка декораторов должна:

* начинаться с Airflow декоратора с python\_callable, например @task.python, @task.sensor
* заканчиваться декоратором @http\_run

Декорируемый метод выполняется последний в цепочке и должен быть объявлен с сигнатурой my\_func(\*args, \*\*kwargs).

Первые два аргумента всегда request и response:

* request: urllib.request.Request
* response: http.client.HTTPResponse

Остальные аргументы зависят от контекста выполнения и могут содержать в том числе xcom из предыдущего таска (task).

Декорируемый метод должен возвращать объект, который может быть сериализован в xcom, например простой словарь:

{

'code': res.getcode(),

'message': res.msg,

'headers': res.getheaders(),

'url': res.geturl(),

'body': res.read1(),

}

Можно передать xcom из предыдущего таска следующим образом:

wait\_job(run\_mapping())

* run\_mapping – первый таск, который запускает загрузку на внешней системе
* wait\_job – второй таск, который принимает
* request, response и результат выполнения первого таска (run\_mapping)

Пример:

# выполним HTTP POST-запрос

# сформируем URL запроса используя Jinja

# в конце обработаем результат в функции run\_mapping

# args[0] - это request

# args[1] - это response

@task.python

@http\_post(

"http://{{ params.host }}:{{ params.port }}/DataIntegrationService/modules/ms/v1/applications/{{ params.app }}/mappings/{{ params.mapping }}"

)

@http\_run()

def run\_mapping(\*args, \*\*kwargs):

return {

"JobId": "my\_job\_id",

}

# запустим сенсор, который будет выполнять HTTP-запросы в течение 3600 секунд, каждые 10 секунд

# сформируем URL запроса используя Jinja

# в конце запроса проверим показатель и выйдем из сенсора (is\_done=True)

@task.sensor(poke\_interval=10, timeout=3600, mode="reschedule")

@http\_post(

"http://{{ params.host }}:{{ params.port }}/DataIntegrationService/modules/ms/v1/applications/{{ params.app }}/mappings/{{ params.mapping }}"

)

@http\_run()

def wait\_job(\*args, \*\*kwargs) -> PokeReturnValue:

req = args[0]

res = args[1]

xcom = args[2]

jobid = xcom["JobId"]

return PokeReturnValue(is\_done=True, xcom\_value="xcom\_value")

wait\_job(run\_mapping())

Как видно из примера, данный Airflow провайдер использует декораторы в качестве этапов, формирующих аргументы в виде запросов (request) и ответов (response), передаваемых в основную Python функцию в виде параметров. А сам исполняемый код при этом не перегружен повторяющимися запросами и выглядит понятно и просто. С полным кодом модуля можно ознакомиться в Приложении 2.

Также в данном модуле дополнительно реализованы некоторые классы, необходимые для работы с различными источниками.

**Менеджер паролей для соединения Airflow-HTTP**

class AirflowHTTPConnectionPasswordMgr(HTTPPasswordMgrWithPriorAuth):

def \_\_init\_\_(self, http\_conn\_url: str, \*args, \*\*kwargs):

self.http\_conn\_url = http\_conn\_url

self.\_http\_connection\_processed = False

super().\_\_init\_\_(\*args, \*\*kwargs)

def add\_credentials\_from\_connection(self, authuri):

try:

conn = Connection.get\_connection\_from\_secrets(self.http\_conn\_url)

except AirflowNotFoundException as e:

print(e, file=sys.stderr)

raise e

except Exception as e:

print(e, file=sys.stderr)

raise e

realm = None

if conn.extra\_dejson and hasattr(conn.extra\_dejson, "realm"):

realm = conn.extra\_dejson.realm

uri = []

if authuri:

uri.append(authuri)

if hasattr(conn, "host"):

uri.append(conn.host)

username = None

if hasattr(conn, "login"):

username = conn.login

password = None

if hasattr(conn, "password"):

password = conn.password

self.add\_password(realm, uri, username, password, is\_authenticated=False)

def find\_user\_password(self, realm, authuri):

if not self.\_http\_connection\_processed:

self.add\_credentials\_from\_connection(authuri)

self.\_http\_connection\_processed = True

return super().find\_user\_password(realm, authuri)

Менеджер паролей AirflowHTTPConnectionPasswordMgr позволяет доставать данные из Airflow connections и может быть передан в любой модуль авторизации:

* HTTPBasicAuthHandler,
* HTTPDigestAuthHandler,
* PreemptiveBasicAuthHandler,
* InformaticaAuthHandler,
* HTTPNtlmAuthHandler
* или другой авторизационный модуль, который вызывает password manager для получения учётных записей.

В данном контексте функция add\_credentials\_from\_connection забирает регистрационные данные (credentials) из airflow connection и добавляет в password\_mgr.

**Авторизационный модуль для Informatica**

class InformaticaAuthHandler(BaseHandler):

auth\_header\_username = "username"

auth\_header\_encryptedpassword = "encryptedpassword"

auth\_header\_securitydomain = "securitydomain"

def \_\_init\_\_(self, password\_mgr=None):

if password\_mgr is None:

password\_mgr = HTTPPasswordMgr()

self.passwd = password\_mgr

self.add\_password = self.passwd.add\_password

def http\_request(self, req):

url = req.get\_full\_url()

realm = None

user, passwd = self.passwd.find\_user\_password(realm, url)

if passwd:

user\_parts = user.split("\\", 1)

if len(user\_parts) == 1:

domain\_name = "Native"

user\_name = user\_parts[0]

else:

domain\_name = user\_parts[0]

user\_name = user\_parts[1]

req.add\_unredirected\_header(self.auth\_header\_username, user\_name)

req.add\_unredirected\_header(self.auth\_header\_encryptedpassword, passwd)

req.add\_unredirected\_header(self.auth\_header\_securitydomain, domain\_name)

return req

https\_request = http\_request

Данный авторизационный модуль для Informatica формирует три заголовка:

* username – имя пользователя. Имя пользователя может быть в двух вариантах написания:  
  – username – непосредственно только username.  
  В таком случае securitydomain станет равен Native  
  – securitydomain\\username – передан securitydomain  
  и username в одном моле через разделитель \\
* encryptedpassword – зашифрованный пароль пользователя.  
  Зашифровать можно с помощью pmpasswd утилиты, которая расположена на сервере с установленной informatiuca  
  – https://knowledge.informatica.com/s/article/528158?language=en\_US  
  – INFA\_HOME/server/bin/pmpasswd <password to encrypt> -e CRYPT\_DATA  
  securitydomain – доменное имя (как в Informatica Web Administator), чувствительно к регистру. По умолчанию используется Native

Авторизация в Informatica работает аналогично preemptive варианту  
– заголовки подставляются в запрос, не дожидаясь повторного запроса из-за ошибки 401.

**Другие модули авторизации**

Подобным образом работают авторизационный модуль для Kerberos, модуль ntlm авторизации для HTTP-клиента, и базовая авторизация с Preemptive. Модуль Kerberos был взят из первоисточника без изменений. Модуль ntlm был переписан и адаптирован под Python 3.

class AbstractNtlmAuthHandler:

"""AbstractNtlmAuthHandler"""

def \_\_init\_\_(self, password\_mgr=None):

if password\_mgr is None:

password\_mgr = HTTPPasswordMgr()

self.passwd = password\_mgr

self.add\_password = self.passwd.add\_password

def http\_error\_authentication\_required(self, auth\_header\_field, req, fp, headers):

"""http\_error\_authentication\_required"""

auth\_header\_value\_list = headers.get\_all(auth\_header\_field)

if auth\_header\_value\_list:

if any([hv.lower() == "ntlm" for hv in auth\_header\_value\_list]):

fp.close()

return self.retry\_using\_http\_NTLM\_auth(

req, auth\_header\_field, None, headers

)

def retry\_using\_http\_NTLM\_auth(self, req, auth\_header\_field, realm, headers):

"""retry\_using\_http\_NTLM\_auth"""

user, pw = self.passwd.find\_user\_password(realm, req.get\_full\_url())

if pw is not None:

user\_parts = user.split("\\", 1)

if len(user\_parts) == 1:

UserName = user\_parts[0]

DomainName = ""

type1\_flags = (

ntlm.NTLM\_TYPE1\_FLAGS & ~ntlm.NTLM\_NegotiateOemDomainSupplied

)

else:

DomainName = user\_parts[0].upper()

UserName = user\_parts[1]

type1\_flags = ntlm.NTLM\_TYPE1\_FLAGS

# ntlm secures a socket, so we must use the same socket for the complete handshake

headers = dict(req.headers)

headers.update(req.unredirected\_hdrs)

auth = f"NTLM {ntlm.create\_NTLM\_NEGOTIATE\_MESSAGE(user, type1\_flags)}"

if req.headers.get(auth\_header\_field, None) == auth:

return None

headers[auth\_header\_field] = auth

host = req.host

if not host:

raise URLError("no host given")

h = None

if req.get\_full\_url().startswith("https://"):

h = HTTPSConnection(host) # will parse host:port

else:

h = HTTPConnection(host) # will parse host:port

# we must keep the connection because NTLM authenticates the connection, not single requests

headers["Connection"] = "Keep-Alive"

headers = dict((name.title(), val) for name, val in list(headers.items()))

h.request(req.get\_method(), req.selector, req.data, headers)

r = h.getresponse()

r.begin()

r.\_safe\_read(int(r.getheader("content-length")))

try:

if r.getheader("set-cookie"):

# this is important for some web applications that store authentication-related info in cookies (it took a long time to figure out)

headers["Cookie"] = r.getheader("set-cookie")

except TypeError:

pass

r.fp = None # remove the reference to the socket, so that it can not be closed by the response object (we want to keep the socket open)

auth\_header\_value = r.getheader(auth\_header\_field, None)

# some Exchange servers send two WWW-Authenticate headers, one with the NTLM challenge

# and another with the 'Negotiate' keyword - make sure we operate on the right one

m = re.match("(NTLM [A-Za-z0-9+\-/=]+)", auth\_header\_value)

if m:

(auth\_header\_value,) = m.groups()

(ServerChallenge, NegotiateFlags) = ntlm.parse\_NTLM\_CHALLENGE\_MESSAGE(

auth\_header\_value[5:]

)

auth = f"NTLM {ntlm.create\_NTLM\_AUTHENTICATE\_MESSAGE(ServerChallenge, UserName, DomainName, pw, NegotiateFlags)}"

headers[auth\_header\_field] = auth

headers["Connection"] = "Close"

headers = dict((name.title(), val) for name, val in list(headers.items()))

try:

h.request(req.get\_method(), req.selector, req.data, headers)

# none of the configured handlers are triggered, for example redirect-responses are not handled!

response = h.getresponse()

def notimplemented():

raise NotImplementedError

response.readline = notimplemented

return addinfourl(

response, response.msg, req.get\_full\_url(), response.code

)

except socket.error as err:

raise URLError("socket error") from err

else:

return None

class HTTPNtlmAuthHandler(AbstractNtlmAuthHandler, BaseHandler):

"""Модуль ntlm авторизации"""

auth\_header = "Authorization"

def http\_error\_401(self, req, fp, \_code, \_msg, headers):

""" "Обработка ошибки 401"""

return self.http\_error\_authentication\_required(

"www-authenticate", req, fp, headers

)

class ProxyNtlmAuthHandler(AbstractNtlmAuthHandler, BaseHandler):

"""

CAUTION: this class has NOT been tested at all!!!

use at your own risk

"""

auth\_header = "Proxy-authorization"

def http\_error\_407(self, req, fp, \_code, \_msg, headers):

""" "Обработка ошибки 407"""

return self.http\_error\_authentication\_required(

"proxy-authenticate", req, fp, headers

)

В модуле с Preemptive происходит Basic авторизация с превентивным выставлением Authorization заголовка, не дожидаясь ошибки 401. При обычной basic авторизации сначала выполняется тестовый запрос. Если возвращается ошибка 401, то формируется второй запрос с Authorization заголовком. PreemptiveBasic при этом формирует authorization заголовок заранее и посылает его в первом же запросе.

class PreemptiveBasicAuthHandler(HTTPBasicAuthHandler):

def http\_request(self, req):

url = req.get\_full\_url()

realm = None

user, passwd = self.passwd.find\_user\_password(realm, url)

if passwd:

raw = f"{user}:{passwd}"

raw = raw.encode("ascii")

auth = b64encode(raw)

auth = auth.decode("ascii")

auth = f"Basic {auth}"

req.add\_unredirected\_header(self.auth\_header, auth)

return req

https\_request = http\_request

**2.3. Дополнительные особенности Airflow провайдера «Gun» для взаимодействия с PostgeSQL и Clickhouse.**

Базовый функционал по взаимодействию с PostgreSQL обеспечивается через работу с Python библиотекой psycopg. Подобно модулю по работе с HTTP-запросам данный модуль представляет собой набор классов, функций и декораторов, которые позволяют выполнить последовательные запросы, используя одно созданное соединение (connection) к базе данных PostgeSQL.

Все декораторы объявлены с префиксом @pg\_ и цепочка декораторов должна начинаться с Airflow декоратора с python\_callable, например @task.python, @task.sensor.

Примеры использования:

# выполним запрос из первого файла

# потом выполним запрос из второго файла

# выполним коммит в конце

# используем Airflow Connection для подключения

@task.python

@pg\_execute\_file("{{ params.second\_sql\_file }}")

@pg\_execute\_file("{{ params.first\_sql\_file }}")

@pg\_auth\_conn\_id("{{ params.gp\_conn\_id }}")

def postgres(\*args, \*\*kwargs):

conn = args[0]

conn.commit()

# выполним запрос 'copy to', скопируем результат в файл params.gzip\_file (используя запрос из файла params.first\_sql\_file)

# выполним запрос 'copy from' скопируем данные из файла params.gzip\_file (используя запрос из файла params.second\_sql\_file)

# выполним коммит в конце

# используем Airflow Connection для подключения

@task.python

@pg\_copy\_from\_gzip\_use\_sql\_file(

"{{ params.second\_sql\_file }}",

"{{ params.gzip\_file }}"

)

@pg\_copy\_to\_gzip\_use\_sql\_file(

"{{ params.first\_sql\_file }}",

"{{ params.gzip\_file }}"

)

@pg\_auth\_conn\_id("{{ params.gp\_conn\_id }}")

@pipe("pg")

def postgres(\*args, \*\*kwargs):

conn = args[0]

conn.commit()

Для подключения к PostgreSQL в начале используется обязательная функция-декоратор pg\_auth\_airflow\_conn, при этом postgres\_conn\_id может быть указано в виде Jinja шаблона.

Airflow connection парсится на словарь атрибутов:

* host – host в psycopg2
* schema – dbname в psycopg2
* login – user в psycopg2
* password – password в psycopg2
* port – port в psycopg2
* extra – сюда можно передать остальные аргументы подключения в виде json словаря, данный словарь объединится с первичными аргументами.

Из указанных атрибутов будет сформирована dsn строка подключения, например:

{'password': 'secret', 'user': 'postgres', 'dbname': 'test'}

'dbname=test user=postgres password=secret'

{'host': 'example.com', 'user': 'someone', 'dbname': 'somedb', 'connect\_timeout': '10'}

"postgresql://someone@example.com/somedb?connect\_timeout=10"

Все DNS-атрибуты можно посмотреть тут: https://www.psycopg.org/docs/extensions.html#psycopg2.extensions.ConnectionInfo

pg\_auth\_conn\_id и pg\_auth\_dsn можно использовать совместно если указаны оба варианта, атрибуты сливаются в единый словарь. Поверх pg\_auth\_conn\_id накладываются атрибуты pg\_auth\_dsn и переопределяют их.

def pg\_auth\_airflow\_conn(

conn\_id: str,

dsn: Optional[str] = None,

conn\_factory: Optional[Any] = None,

cursor\_factory: Optional[Any] = None,

cursor\_name: Optional[str] = None,

cursor\_withhold: Optional[bool] = None,

cursor\_scrollable: Optional[bool] = None,

):

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

PostgresAuthAirflowConnection(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

conn\_id=conn\_id,

dsn=dsn,

connection\_factory=conn\_factory,

cursor\_factory=cursor\_factory,

cursor\_name=cursor\_name,

cursor\_scrollable=cursor\_scrollable,

cursor\_withhold=cursor\_withhold,

)

)

def pg\_auth\_dsn(

dsn: str,

conn\_factory: Optional[Any] = None,

cursor\_factory: Optional[Any] = None,

cursor\_name: Optional[str] = None,

cursor\_withhold: Optional[bool] = None,

cursor\_scrollable: Optional[bool] = None,

):

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

PostgresAuthAirflowConnection(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

dsn=dsn,

connection\_factory=conn\_factory,

cursor\_factory=cursor\_factory,

cursor\_name=cursor\_name,

cursor\_scrollable=cursor\_scrollable,

cursor\_withhold=cursor\_withhold,

)

)

return builder

return wrapper

Также обязательной является функция-декоратор pg\_execute или подобные, которые отвечают за непосредственное выполнение SQL-запроса.

Декоратор pg\_execute выполняет SQL-запрос без коммита (commit).

Для выполнения с коммита можно использовать декораторы:

@pg\_commit

@pg\_execute\_and\_commit

Также SQL-запрос может содержать Jinja шаблоны.

def pg\_execute(sql: str, params=None):

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

PostgresExecuteModule(

builder.context\_key,

builder.template\_render,

sql,

params,

)

)

return builder

return wrapper

Остальные декораторы являются опциональными и используются по необходимости.

Декоратор pg\_execute\_file выполняет запрос из указанного SLQ-файла, при этом sql\_file может быть передан в как Jinja шаблон, либо контент в файле также может содержать Jinja шаблоны.

Декоратор pg\_execute\_file\_and\_commit выполняет запрос из указанного SLQ-файла и делает коммит.

Декоратор pg\_fetch\_to\_stdout считывает данные построчно и выводит их в стандартный вывод stdout. Данные выводятся с помощью print в том виде, в котором их принял драйвер psycopg.

Декоратор pg\_commit выполняет коммит.

Декоратор pg\_module добавляет кастомный[[1]](#footnote-1) модуль обработки в конвейер PostgresBuilder.

Декораторы pg\_copy\_to\_handle и pg\_copy\_to\_stdout выполняют копирование в указанный обработчик handle (open() от файла или stdout) или в стандартный вывод stdout, используя переданный SQL-запрос (выводит в консоль результат выполнения запроса). SQL-запрос может содержать jinja шаблоны. С документацией по copy можно ознакомиться тут: https://postgrespro.ru/docs/postgresql/14/sql-copy

Пример copy запроса:

copy (

select \* from public.pgbench\_history where tid > 4

)

to stdout with (

format csv,

header,

delimiter ';',

quote '"',

escape '"'

)

Подобным образом ведет себя декоратор pg\_copy\_to\_file, но производит копирование в файл, при этом to\_file - это путь к файлу и может быть передан как Jinja шаблон. Декоратор pg\_copy\_from\_file прозводит копирование из файла. То есть файл будет прочитан как поток байт и отправлен в PostgreSQL.

Декоратор pg\_copy\_to\_gzip выполняет копирование в файл, архивируя поток в формате gzip, используя переданный SQL-запрос. pg\_copy\_from\_gzip выполняет копирование из заархивированного файла (в формате gzip), используя переданный SQL-запрос. Внутри архива gzip может быть любой файл, он будет прочитан как поток байт и отправлен в PostgreSQL.

Декоратор pg\_copy\_to\_file\_use\_sql\_file выполняет копирование в файл используя переданный SQL-файл. Аналогичным образом pg\_copy\_from\_file\_use\_sql\_file выполняет копирование из файла, используя переданный SQL-файл, при этом файл будет прочитан как поток байт и отправлен в PostgreSQL.

Декоратор pg\_copy\_to\_gzip\_use\_sql\_file выполняет копирование в файл (архивируя поток в формате gzip), используя переданный SQL-файл.

Декоратор pg\_copy\_from\_gzip\_use\_sql\_file выполняет копирование из заархивированного файла (в формате gzip), используя переданный SQL-файл. При этом внутри архива gzip может быть любой файл, он будет прочитан как поток байт и отправлен в PostgreSQL.

Модуль pg\_save\_to\_xcom позволяет сохранить любую информацию в Airflow XCom для последующего использования.

Для сохранения информации в xcom нужно передать:

* xcom\_name - это имя xcom записи. По умолчанию имя в xcom используется "return\_value". Необходимо использовать любое другое имя, отличное от этого для передачи касомного xcom между задачами.
* xcom\_gen - это функция, которая будет использована для генерации значения.
* xcom\_gen принимает 1 параметр: psycopg2.extensions.cursor - текущий postgres cursor
* xcom\_gen должна возвращать то значение, которое можно сериализовать в xcom

Например, можно сохранить кол-во строк, которые вернул PostgreSQL:

@pg\_save\_to\_xcom("myxcom", lambda cur, context: {

'target\_row': pg\_cur.rownumber,

'source\_row': 0,

'error\_row': 0,

"query": cur.query,

})

Модуль pg\_save\_to\_context позволяет сохранить любую информацию в Airflow Context для последующего использования

Для сохранения информации в context нужно передать:

* context\_name - это имя context ключа.
* context\_gen - это функция, которая будет использована для генерации значения, которое будет добавлено в context

context\_gen принимает 1 параметр context и функция должна возвращать то значение, которое нужно сохранить в context.

Например, можно сохранить кол-во строк, которые вернул PostgreSQL:

@pg\_save\_to\_context("my\_context\_key", lambda cur, context: {

'target\_row': "{{ params.target\_row }}",

'source\_row': 0,

'error\_row': 0,

})

Также модуль содержит ряд функций, которые позволяют добавить поддержку uuid, json, inet типов данных, а также добавляют поддержку hstore хранения, composite, ipaddress.

В заключении приведен декоратор get\_postgres\_logger, который возвращает логер (logger), который пишет словарь (dict) в таблицу PostgreSQL. Также он умеет писать Json-поля в виде Json.

Пример использования:

logger = get\_postgres\_logger('mylogger', 'airflow\_connection\_id', 'my\_postgres\_schema', 'my\_postgres\_table')

logger.info({

"log\_id": uuid.uuid4(),

"integration\_id": 0,

"process\_name": "dag\_name",

"task\_name": "my task name",

"status": 1,

"payload": {

"target\_row": pg\_cur.rownumber,

"source\_row": 0,

"error\_row": 0,

},

})

* logger\_name - имя логера
* conn\_id - Airflow Connection Id, в котором содержится учетная запись для подключения к PostgreSQL
* schema\_name - схема по умолчанию, в которую буде писать logger
* table\_name - таблица по умолчанию, в которую буде писать logger
* conn\_dsn - дополнительные DNS-параметры для PostgreSQL, например, можно передать ApplicationName: "application\_name=my\_application\_name"
* extra\_register - обработчик позволяющий сконфигурировать pg cursor перед выполнением вставки (insert). Может потребоваться, если объект логирования (Dict) содержит сложные типы данных, например hstore.

По аналогии с модулем для PostgreSQL построен модуль для работы с Clickhouse. Существуют два обязательных декоратора:

* @ch\_auth\_airflow\_conn - Представляет собой соединение между клиентом и сервером ClickHouse.
* @ch\_send\_query - Отправляет SQL-запрос в Clickhouse. Подробнее с настройками можно ознакомиться тут: <https://clickhouse.com/docs/en/operations/settings/settings>

Благодаря архитектуре провайдера также добавлен декоратор ch\_send\_queries, который может отправлять несколько запросов. Также реализована возможность отправлять запрос(ы) из файла в соответствующих декораторах ch\_send\_file\_query и ch\_send\_file\_queries. Далее представлен декоратор ch\_save\_to\_csv, который позволяет сохранять данные в CSV-файл, а также ch\_save\_to\_gzip\_csv, если файлы должны быть заархивированы.

Модуль ch\_save\_to\_xcom позволяет сохранить любую информацию в Airflow XCom для последующего использования.

Для сохранения информации в xcom нужно передать:

* xcom\_name - это имя xcom записи. Напомню, что по умолчанию имя в xcom используется "return\_value". Необходимо использовать любое другое имя, отличное от этого для передачи касомного xcom между задачами.
* xcom\_gen - это функция, которая будет использована для генерации значения. xcom\_gen принимает 1 параметр: ClickhouseCursor - текущий clickhouse cursor. xcom\_gen должна возвращать то значение, которое можно сериализовать в xcom.

Пример:

@ch\_save\_to\_xcom("myxcom", lambda cur, context: {

'source\_row': 0,

'error\_row': 0,

})

Модуль ch\_save\_to\_context позволяет сохранить любую информацию в Airflow Context для последующего использования. Для сохранения информации в context нужно передать:

* context\_name - это имя context ключа.
* context\_gen - это функция, которая будет использована для генерации значения, которое будет добавлено в context.

При этом context\_gen принимает 1 параметра: context и функция должна возвращать то значение, которое нужно сохранить в context.

Например:

@ch\_save\_to\_context("my\_context\_key", lambda cur, context: {

'target\_row': "{{ params.target\_row }}",

'source\_row': 0,

'error\_row': 0,

})

Декоратор ch\_send\_data\_from\_csv отправляет данные из CSV-файла в Clickhouse.

Итак, в данной главе были рассмотрены ключевые особенности кода для Airflow провайдера для работы по HTTP, с PostgrSQL и Clickhouse. Общий принцип архитектуры заключается в схожей структуре и однотипных функциях, выполненных с учетом особенностей, присущих определенным СУБД и протоколам. Поэтому данный провайдер является интуитивно понятным легко и масштабируемым.

**Глава 3. Тестирование ряда функций и оптимизация кода Airflow провайдера «Gun» для взаимодействия с HTTP, PostgeSQL и Clickhouse.**

**3.1. Тестирование Airflow провайдера «Gun» для взаимодействия с HTTP**

В данной главе приведены ряд тестов для Airflow провайдера «Gun». Каждый тест выполнен в виде DAG’а, поскольку именно это и является единицей Airflow. Полный код тестов можно найти в Приложении 2.

Итак, рассмотрим некоторые тесты для HTTP-запросов.

@task.python()

@http\_run()

@http\_req("{{ params.get\_url }}")

@pipe.airflow\_task("pipe")

def http\_1(context):

res = context["pipe"]["res"]

for r in res:

print(r)

http\_1()

В первом тесте основной вызываемой функцией является http\_1(). С помощью декораторов выполняются поэтапно следующие шаги (читаем снизу вверх):

* @pipe.airflow\_task("pipe") используется для построения объекта таска при создании таска для системы потоков данных в Apache Airflow.
* @http\_req("{{ params.get\_url }}") выполняет HTTP-запрос по указанному URL.
* @http\_run() запускает выполнение HTTP-запроса.
* @task.python() используется для создания задачи (или оператора) на основе функции на языке Python.

Когда функция декорируется декоратором @task.python(), она превращается в объект задачи, который может быть запущен в рамках DAG’а. Этот декоратор предоставляет основные функциональности, необходимые для запуска и управления таском, включая параллельное выполнение, управление зависимостями и обработку ошибок.

Следующий тест проверяет обработку ошибок 400 и 405 и запускает повторный таск 10 раз в случае ошибки:

@task.python()

@http\_run()

@http\_retry\_if\_code([400, 405], 10)

@http\_post("https://google.com", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_5(context):

pass

http\_5()

В данном тесте само тело функции оставлено пустым, поскольку для теста не представляется важным тестировать работу самой функции. Также написаны тесты, обрабатывающие другие ошибки.

Для того, чтобы проверить итерируемость была написала дополнительная функция iterable\_generator:

def iterable\_generator():

yield b"1"

yield b"2"

yield b"3"

yield b"4"

yield b"5"

yield b"6"

@task.python()

@http\_run()

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_iterable(iterable\_generator())

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_send\_iterable\_transfer\_encoding(context):

res = context["pipe"]["res"]

res = res.read1()

res = json.loads(res)

assert res["data"] == "123456"

http\_send\_iterable\_transfer\_encoding()

Ещё в качестве примера приведем тест на вывод тела ответа в консоль:

@task.python()

@http\_run()

@http\_response\_body\_to\_stdout()

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_bytes(b"hello world")

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_save\_response\_to\_stdout(context):

pass

http\_save\_response\_to\_stdout()

Как видно из тестов, все они обрамляются декораторами @pipe.airflow\_task и @http\_run, в промежутке добавляются другие декораторы, которые необходимо протестировать согласно заложенной логике.

Таким образом, с помощью тестов был протестирован весь функционал HTTP-запросов для Airflow провайдера «Gun».

**3.2. Тестирование Airflow провайдера «Gun» для взаимодействия Clickhouse**

Аналогично предыдущим тестам рассмотрим несколько примеров для Clickhouse. Полный текст кода можно посмотреть в Приложении 3.

Первый тест позволяет выбрать одну строку и вывести ее в консоль без дополнительной обработки:

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select 1")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def select\_one\_row(\_):

pass

select\_one\_row()

Далее мы добавляем ещё один декоратор, который содержит client\_name и также выводим результат в консоль:

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select 1")

@ch\_auth\_airflow\_conn(

"{{ params.conn\_id }}",

settings={

"client\_name": "my\_super\_name",

},

)

@pipe()

def select\_one\_row\_with\_client\_name(\_):

pass

select\_one\_row\_with\_client\_name()

По нарастающей продолжаем тестировать декораторы, используя различные запросы, например, выбирая 2 строки или формат CSV. При этом важно соблюдать правильную последовательность декораторов, иначе запрос не будет обработан должным образом и тест не будет пройден.

Также протестировали возможность делать несколько запросов:

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("show databases")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select version()")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select 1 as t")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def multiple\_query(\_):

pass

multiple\_query()

Это возможно благодаря тому, что мы имеем одно подключение к базе в начале и далее делаем несколько запросов, выводя каждый из них в консоль.

И, пожалуй, одним из сложных тестов можно назвать тест на создание временной таблицы со вставкой данных:

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"select \* from temp1",

settings={

"xcx": 3,

},

)

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("insert into temp1(id) select number from numbers(0, 100)")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"create temporary table temp1 (id UInt8) COMMENT 'The temporary table'",

settings={

"max\_block\_size": 3,

},

)

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

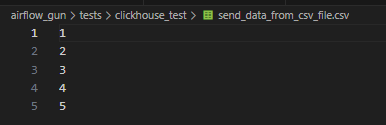
@pipe()

def create\_temp\_table\_and\_insert\_data(\_):

pass

create\_temp\_table\_and\_insert\_data()

Во время написания данного теста была использована директория dags/tests/clickhouse\_test/send\_data\_from\_csv\_file.csv и как можно увидеть в последствии был создан файл с данными:



Изображение 2

Не стоит забывать, что в параметрах DAG’а изначально были прописаны некоторые пути к файлам, которые были использовали в данном контексте:

params={

"conn\_id": Param("clickhouse\_\_spb99dkl-adqm01d"),

"sql\_file": Param("dags/tests/clickhouse\_test/sql\_file.sql"),

"sql\_file\_multi\_statements": Param(

"dags/tests/clickhouse\_test/sql\_file\_multi\_statements.sql"

),

"csv\_file": Param("data/test\_2.csv"),

"gzip\_csv\_file": Param("data/test.gzip"),

},

Аналогичным образом будут выглядеть в последующем тесты для PostgreSQL, однако в данной работе на этом завершается рассмотрение Airflow провайдера «Gun» для взаимодействия с HTTP, PostgeSQL и Clickhouse. С полным кодом можно ознакомиться по ссылке: https://github.com/karmusha/airflow\_gun

**Заключение**

Исходя из поставленной цели, в данной работе были рассмотрены следующие задачи:

1. Изучение основных возможностей языка Python и его применение для работы с Airflow.
2. Рассмотрение особенностей использования Airflow в качестве провайдера для выполнения HTTP-запросов.
3. Изучение методов работы с PostgreSQL и Clickhouse с использованием Airflow.
4. Анализ преимуществ использования TaskFlow API и декораторов в Python для работы с данными.
5. Исследование необходимости перехода на отечественное ПО и программы с открытым исходным кодом, а также обоснование использования Airflow для интеграции с внешними системами.

В ходе выполнения данной дипломной работы были изучены различные варианты применения Python с Airflow, а также были рассмотрены примеры использования данного провайдера для выполнения запросов и работы с базами данных.

Были выявлены преимущества использования TaskFlow API и декорирования функций в Python, которые позволяют упростить и сделать более понятным выполнение сложных операций с данными. Данное исследование может быть полезно специалистам в области программирования, работающим с Airflow и базами данных, а также тем, кто интересуется вопросами перехода на отечественное ПО.

Самой главной особенностью этого Airflow-провайдера является способность обрабатывать несколько запросов в рамках одного таска посредством обертывания запросов внутри декораторов с последующей обработкой бизнес-логики внутри Python-функции. Такое решение обусловлено необходимостью выполнять однотипные действия при работе с данными и предназначено для упрощения и улучшения читаемости кода. Таким образом, мы можем выполнять несколько операций внутри одного таска и использовать результаты их выполнения внутри одной функции. Кроме того, благодаря конвейерной обработке, мы получаем гибкий и масштабируемый подход для работы с данными.

**Библиография**

1. <https://www.python.org/> – Официальная документация Python
2. <https://ru.wikipedia.org/wiki/Python> – Определение Python
3. <https://ru.wikipedia.org/wiki/Apache_Airflow> – Определение Airflow
4. <https://airflow.apache.org/docs/apache-airflow/> – Официальная документация к Airflow
5. <https://habr.com/ru/companies/vk/articles/339392/> – Airflow – инструмент, чтобы удобно и быстро разрабатывать и поддерживать batch-процессы обработки данных
6. <https://developers.sber.ru/docs/ru/sdp/sdpanalytics/airflow/guidelines-airflow> – Руководство по использованию AirFlow (оркестратор для задач ETL)
7. <https://habr.com/ru/companies/alfa/articles/676926/> – Знакомство с Apache Airflow: установка и запуск первого DAGа
8. <https://habr.com/ru/articles/682384/> – Все, что вам нужно знать об Airflow DAGs, ч.1 – Основы и расписания
9. <https://habr.com/ru/articles/682460/> – Все, что вам нужно знать об Airflow DAGs, ч.2 – Операторы и Датчики
10. <https://bigdataschool.ru/blog/taskflow-api-in-apache-airflow.html> - Создавайте графы в Apache Airflow с помощью TaskFlow API
11. <https://ru.wikipedia.org/wiki/HTTP> – Определение HTTP
12. <https://habr.com/ru/articles/215117/> – Простым языком об HTTP
13. <https://postgrespro.ru/docs/postgresql> – Официальная документация PostgreSQL
14. <https://clickhouse.com/docs/ru> – Официальная документация Clickhouse

**Справочная информация:**

1. <https://github.com/karmusha/airflow_gun> – полный код Airflow провайдера «Gun» для взаимодействия с HTTP, PostgeSQL и Clickhouse
2. <https://www.vedu.ru/expdic/122012/> – Толковый словарь русского языка, слово «кастомный»
3. <https://postgrespro.ru/docs/postgresql/14/sql-copy> – Документация по copy для PostgreSQL
4. https://clickhouse.com/docs/en/operations/settings/settings – Документация по настройкам Clickhouse.

**Приложение 1**

Модуль HTTP:

\_\_init\_\_.py

import sys

import json

import ssl

import gzip

import shutil

from typing import Dict, List, Optional

from collections.abc import Iterable

from pathlib import Path

from urllib.request import (

Request,

build\_opener,

HTTPHandler,

BaseHandler,

HTTPSHandler,

HTTPError,

HTTPDefaultErrorHandler,

HTTPBasicAuthHandler,

HTTPDigestAuthHandler,

)

from airflow.providers.gun.http.informatica\_auth import InformaticaAuthHandler

from airflow.providers.gun.http.kerberos\_auth import HTTPKerberosAuthHandler

from airflow.providers.gun.http.ntlm\_auth import HTTPNtlmAuthHandler

from airflow.providers.gun.http.preemptive\_basic\_auth import PreemptiveBasicAuthHandler

from airflow.providers.gun.http.airflow\_http\_password\_mgr import (

AirflowHTTPConnectionPasswordMgr,

)

from airflow.providers.gun.pipe import PipeTask, PipeTaskBuilder

\_\_all\_\_ = [

"http\_req",

"http\_get",

"http\_post",

"http\_put",

"http\_run",

"http\_headers",

"http\_body\_dict",

"http\_body\_file",

"http\_body\_string",

"http\_body\_bytes",

"http\_body\_iterable",

"http\_auth\_basic",

"http\_auth\_preemptive\_basic",

"http\_auth\_digest",

"http\_auth\_informatica",

"http\_auth\_ntlm",

"http\_auth\_kerberos",

"http\_auth\_preemptive\_basic\_with\_conn\_id",

"http\_auth\_basic\_from\_conn\_id",

"http\_auth\_digest\_from\_conn\_id",

"http\_auth\_informatica\_from\_conn\_id",

"http\_auth\_ntlm\_from\_conn\_id",

"http\_auth\_kerberos\_from\_conn\_id",

"http\_auth\_conn\_id",

"http\_error\_if\_code",

"http\_error\_supress",

"http\_retry\_if\_code",

"http\_response\_body\_to\_gzip",

"http\_response\_body\_to\_stdout",

]

def \_flatten(code):

"""

превращает вложенные списки в один

[[1, 2, 4], 5, [34]] => [1, 2, 4, 5, 34]

"""

match code:

case int(code):

return code

case str(code):

code = int(code)

return code

case code if isinstance(code, Iterable):

for co in code:

if isinstance(co, Iterable) and not isinstance(co, (str, bytes)):

for sub\_x in \_flatten(co):

yield sub\_x

else:

yield co

return code

case \_:

raise ValueError(

f'{\_\_name\_\_} requires "code" be passed, example: 401 or [401,404,500], or range(400, 600)'

)

class HttpReq(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

url,

method,

timeout,

ssl\_verify,

debuglevel,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(("url", "method", "timeout", "ssl\_verify"))

super().set\_template\_render(template\_render)

self.url = url

self.method = method

self.timeout = timeout

self.ssl\_verify = ssl\_verify

self.debuglevel = debuglevel

self.request\_key = "req"

self.timeout\_key = "timeout"

self.modules\_key = "modules"

self.ssl\_verify\_key = "ssl\_verify"

self.debuglevel\_key = "debuglevel"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

req = Request(

url=self.url,

method=self.method,

)

share = context[self.context\_key]

share[self.request\_key] = req

share[self.timeout\_key] = self.timeout

share[self.modules\_key] = []

share[self.ssl\_verify\_key] = self.ssl\_verify

share[self.debuglevel\_key] = self.debuglevel

def http\_req(

url: str,

method: str = "GET",

timeout: Optional[int] = None,

ssl\_verify=False,

debuglevel: int = 0,

):

"""

Выполнить http запрос по указанному url

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpReq(

builder.context\_key,

builder.template\_render,

url,

method,

timeout,

ssl\_verify,

debuglevel,

)

)

return builder

return wrapper

def http\_get(url: str, timeout: Optional[int] = None, ssl\_verify=False, debuglevel=0):

"""

Выполнить GET запрос по указанному URL

"""

return http\_req(url, "GET", timeout, ssl\_verify, debuglevel)

def http\_post(url: str, timeout: Optional[int] = None, ssl\_verify=False, debuglevel=0):

"""

Выполнить POST запрос по указанному URL

"""

return http\_req(url, "POST", timeout, ssl\_verify, debuglevel)

def http\_put(url: str, timeout: Optional[int] = None, ssl\_verify=False, debuglevel=0):

"""

Выполнить PUT запрос по указанному URL

"""

return http\_req(url, "PUT", timeout, ssl\_verify, debuglevel)

def http\_delete(

url: str, timeout: Optional[int] = None, ssl\_verify=False, debuglevel=0

):

"""

Выполнить DELETE запрос по указанному URL

"""

return http\_req(url, "DELETE", timeout, ssl\_verify, debuglevel)

class HttpRes(PipeTask):

def \_\_init\_\_(

self,

context\_key,

stack\_key,

template\_render,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(())

super().set\_template\_render(template\_render)

self.stack\_key = stack\_key

self.request\_key = "req"

self.timeout\_key = "timeout"

self.modules\_key = "modules"

self.ssl\_verify\_key = "ssl\_verify"

self.debuglevel\_key = "debuglevel"

self.response\_key = "res"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

stack = context[self.stack\_key]

req = share[self.request\_key]

timeout = share[self.timeout\_key]

modules: List = share[self.modules\_key]

debuglevel = share[self.debuglevel\_key]

ssl\_verify = share[self.ssl\_verify\_key]

if not ssl\_verify:

ssl.\_create\_default\_https\_context = ssl.\_create\_unverified\_context

modules.insert(0, HTTPSHandler(debuglevel))

modules.insert(0, HTTPHandler(debuglevel))

opener = build\_opener(\*modules)

res = opener.open(fullurl=req, timeout=timeout)

res = stack.enter\_context(res)

share[self.response\_key] = res

def http\_run():

"""

Запустить выполнение http запроса

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpRes(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

)

)

return builder

return wrapper

class HttpHeadersFromDictModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

headers,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"headers",

]

)

super().set\_template\_render(template\_render)

self.headers: Dict[str, str] = headers

self.request\_key = "req"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

req: Request = share[self.request\_key]

for key, val in self.headers.items():

req.add\_header(key, val)

def http\_headers(headers: dict):

"""

Добавить дополнительные заголовки в http запрос, например Content-Type

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpHeadersFromDictModule(

builder.context\_key,

builder.template\_render,

headers,

)

)

return builder

return wrapper

class HttpReqBodyFromDictModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

body: dict,

encoding="utf-8",

ensure\_ascii=False,

default=str,

errors="backslashreplace",

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"body",

"encoding",

"ensure\_ascii",

"default",

"errors",

]

)

super().set\_template\_render(template\_render)

self.body = body

self.encoding = encoding

self.ensure\_ascii = ensure\_ascii

self.default = default

self.errors = errors

self.request\_key = "req"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

req: Request = share[self.request\_key]

body = json.dumps(

self.body,

ensure\_ascii=self.ensure\_ascii,

default=self.default,

)

body = body.encode(

encoding=self.encoding,

errors=self.errors,

)

req.data = body

def http\_body\_dict(

body: dict,

encoding="utf-8",

ensure\_ascii=False,

default=str,

errors="backslashreplace",

):

"""

Добавить request body в запрос из переданного словаря

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpReqBodyFromDictModule(

builder.context\_key,

builder.template\_render,

body,

encoding,

ensure\_ascii,

default,

errors,

)

)

return builder

return wrapper

class HTTPRetryRequestIfCode(HTTPDefaultErrorHandler):

"""

Повторяет запрос при колучении указанного кода/кодов

можно передать:

- 401 - одиночный код

- [401,404,500] - набор кодов

- range(400, 600) - диапазон кодов

"""

def \_\_init\_\_(self, code, retry, body\_print):

self.retry = retry

self.body\_print = body\_print

if isinstance(code, Iterable):

self.code = \_flatten(code)

else:

self.code = [code]

for code in self.code:

setattr(self, f"http\_error\_{code}", self.\_default\_handler)

setattr(self, f"https\_error\_{code}", self.\_default\_handler)

def \_default\_handler(self, req, res, errcode, errmsg, headers):

if self.body\_print:

body = res.read()

print("----- body -----")

print(body)

if self.retry <= 0:

raise HTTPError(

url=res.geturl(),

code=errcode,

msg=errmsg,

hdrs=headers,

fp=res,

)

else:

print(

f"\n> retry module: There are {self.retry} attempts left, I am executing the request again\n"

)

self.retry -= 1

return self.parent.open(req, timeout=req.timeout)

class HttpRetryModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

code,

retry,

body\_print,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"code",

"retry",

"body\_print",

]

)

super().set\_template\_render(template\_render)

self.code = code

self.retry = retry

self.body\_print = body\_print

self.modules\_key = "modules"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

modules: List = share[self.modules\_key]

modules.append(

HTTPRetryRequestIfCode(

self.code,

self.retry,

self.body\_print,

)

)

def http\_retry\_if\_code(

code: int | list | range,

retry: int,

body\_print: bool = True,

):

"""

Добавить дополнительные попытки в случае получения указанных http кодов

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpRetryModule(

builder.context\_key,

builder.template\_render,

code,

retry,

body\_print,

)

)

return builder

return wrapper

class HTTPErrorSuppressProcessor(HTTPDefaultErrorHandler):

"""

для того, что бы не генерировалось исключение в случае http ошибок

выносим решение об исключении на уровень бизнес использования

"""

def http\_error\_default(self, req, fp, code, msg, hdrs):

return fp

class HTTPErrorIfCode(HTTPErrorSuppressProcessor):

"""

Выдаёт исключение только при получении указанного/указанных http кодов

можно передать:

- 401 - одиночный код

- [401,404,500] - набор кодов

- range(400, 600) - диапазон кодов

"""

def \_\_init\_\_(self, code):

if isinstance(code, Iterable):

self.code = \_flatten(code)

else:

self.code = [code]

def http\_error\_default(self, req, fp, code, msg, hdrs):

if code in self.code:

return self.\_default\_handler(req, fp, code, msg, hdrs)

return fp

def \_default\_handler(self, req, fp, code, msg, hdrs):

raise HTTPError(

url=fp.geturl(),

code=code,

msg=msg,

hdrs=hdrs,

fp=fp,

)

class HttpErrorSupressModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields([])

super().set\_template\_render(template\_render)

self.modules\_key = "modules"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

modules: List = share[self.modules\_key]

modules.append(HTTPErrorSuppressProcessor())

def http\_error\_supress():

"""

Подавить HTTPError если код ответа от сервера не является 200

\* позволяет вынести обработку, в том числе, ошибочных ответом в to\_xcom модуль

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpErrorSupressModule(

builder.context\_key,

builder.template\_render,

)

)

return builder

return wrapper

class HttpErrorIfCodeModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

code,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"code",

]

)

super().set\_template\_render(template\_render)

self.code = code

self.modules\_key = "modules"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

modules: List = share[self.modules\_key]

modules.append(

HTTPErrorIfCode(

self.code,

)

)

def http\_error\_if\_code(

code: int | list | range,

):

"""

Добавить дополнительные попытки в случае получения указанных http кодов

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpErrorIfCodeModule(

builder.context\_key,

builder.template\_render,

code,

)

)

return builder

return wrapper

class HttpBodyFromFileModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

stack\_key,

template\_render,

file,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"file",

]

)

super().set\_template\_render(template\_render)

self.stack\_key = stack\_key

self.file = file

self.request\_key = "req"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

stack = context[self.stack\_key]

req: Request = share[self.request\_key]

body = Path(self.file)

body = body.open(mode="rb")

body = stack.enter\_context(body)

req.data = body

def http\_body\_file(

file: str,

):

"""

Сформировать request body из переданного файла

Файл должен существовать на момент выполнения Task

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpBodyFromFileModule(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

file,

)

)

return builder

return wrapper

class HttpBodyFromStringModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

content: str,

encoding: str = "utf-8",

errors="backslashreplace",

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"content",

"encoding",

"errors",

]

)

super().set\_template\_render(template\_render)

self.content = content

self.encoding = encoding

self.errors = errors

self.request\_key = "req"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

req: Request = share[self.request\_key]

body = self.content.encode(

encoding=self.encoding,

errors=self.errors,

)

req.data = body

def http\_body\_string(

content: str,

encoding: str = "utf-8",

errors="backslashreplace",

):

"""

Сформировать request body из обычной строки текста.

Строка декодится в набор байт согласно указанной кодировке

В случае невозможности декодировать символ, сработает указанное правило в errors

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpBodyFromStringModule(

builder.context\_key,

builder.template\_render,

content,

encoding,

errors,

)

)

return builder

return wrapper

class HttpBodyFromBytesModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

content,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields([])

super().set\_template\_render(template\_render)

self.content = content

self.request\_key = "req"

def \_\_call\_\_(self, context):

share = context[self.context\_key]

req: Request = share[self.request\_key]

req.data = self.content

def http\_body\_bytes(

content: bytes,

):

"""

Сформировать request body из указанного набора байт

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpBodyFromBytesModule(

builder.context\_key,

builder.template\_render,

content,

)

)

return builder

return wrapper

class HttpBodyFromIterableModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

content,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields([])

super().set\_template\_render(template\_render)

self.content = content

self.request\_key = "req"

def \_\_call\_\_(self, context):

share = context[self.context\_key]

req: Request = share[self.request\_key]

req.data = self.content

def http\_body\_iterable(

content: Iterable,

):

"""

Сформировать request body из указанного итерируемого объекта

Итератор должен возвращать набор bytes при каждой итерации

Запрос с таким объектом выполняется в потоке отсылая каждый next по открытому socket

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpBodyFromIterableModule(

builder.context\_key,

builder.template\_render,

content,

)

)

return builder

return wrapper

class HttpAddhandler(PipeTask):

def \_\_init\_\_(

self,

context\_key,

template\_render,

\*handler,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields([])

super().set\_template\_render(template\_render)

self.handler = handler

self.modules\_key = "modules"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

modules: List = share[self.modules\_key]

for m in self.handler:

modules.append(m)

def http\_handler(

\*handler: BaseHandler,

):

"""

Добавить дополнительные кастомные модули обработки запросов

- Модуль должен быть подклассом базового типа: urllib.request.BaseHandler

- https://docs.python.org/3/library/urllib.request.html

- модули добавляются в указанной последовательности

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.template\_render,

\*handler,

)

)

return builder

return wrapper

class HTTPSaveResponseBodyToFileGzip(BaseHandler):

"""

Сохраняет response body в файл

"""

def \_\_init\_\_(self, file\_path: str):

self.file\_path = file\_path

def http\_response(self, request, response):

with gzip.open(Path(self.file\_path), "wb") as f\_out:

shutil.copyfileobj(response, f\_out)

return response

https\_response = http\_response

class HttpResBodyToGzipModule(PipeTask):

def \_\_init\_\_(

self,

context\_key,

stack\_key,

template\_render,

file\_path,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"file\_path",

]

)

super().set\_template\_render(template\_render)

self.stack\_key = stack\_key

self.file\_path = file\_path

self.modules\_key = "modules"

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

modules: List = share[self.modules\_key]

modules.append(HTTPSaveResponseBodyToFileGzip(self.file\_path))

def http\_response\_body\_to\_gzip(

file\_path: str,

):

"""

Сохранить response body в файл "file\_path"

file\_path мжно указать в виде jinja шаблона:

{{ params.file\_path }}

{{ task\_instance.xcom\_pull(task\_ids='prev\_task\_id', key='file\_path') }}

Помни что response body можно прочитать из сокета только один раз

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpResBodyToGzipModule(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

file\_path,

)

)

return builder

return wrapper

class HTTPPrintResponseBodyToStdout(BaseHandler):

"""

Принтует response body в stdout

"""

def http\_response(self, request, response):

shutil.copyfileobj(response, sys.stdout.buffer)

return response

https\_response = http\_response

def http\_response\_body\_to\_stdout():

"""

Притануть response body в stdout

Помни что response body можно прочитать из сокета только один раз

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.template\_render,

HTTPPrintResponseBodyToStdout(),

)

)

return builder

return wrapper

def http\_auth\_basic(

password\_mgr,

):

"""

Добавляет модуль basic авторизации

password\_mgr - менеджер паролей из которого будет взять username и password:

https://docs.python.org/3/library/urllib.request.html#httppasswordmgr-objects

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.template\_render,

HTTPBasicAuthHandler(password\_mgr),

)

)

return builder

return wrapper

def http\_auth\_preemptive\_basic(password\_mgr):

"""

Добавляет модуль preemptive basic авторизации

basic авторизация выполняется в два этапа:

- формируется запрос без авторизации

- в случае получения кода 401, формируется второй запрос с Authorization заголовком

preemptive\_basic - выполняется в один этап, заголовок Authorization

отправляются в первом же запросе

password\_mgr - менеджер паролей из которого будет взять username и password:

https://docs.python.org/3/library/urllib.request.html#httppasswordmgr-objects

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.template\_render,

PreemptiveBasicAuthHandler(password\_mgr),

)

)

return builder

return wrapper

def http\_auth\_digest(password\_mgr):

"""

Добавляет модуль digest авторизации

password\_mgr - менеджер паролей из которого будет взять

username и password:

https://docs.python.org/3/library/urllib.request.html#httppasswordmgr-objects

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

HTTPDigestAuthHandler(password\_mgr),

)

)

return builder

return wrapper

def http\_auth\_informatica(password\_mgr):

"""

Добавляет модуль informatica авторизации, которая необходима для rest api

password\_mgr - менеджер паролей из которого будет взять

username и password:

https://docs.python.org/3/library/urllib.request.html#httppasswordmgr-objects

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.template\_render,

InformaticaAuthHandler(password\_mgr),

)

)

return builder

return wrapper

def http\_auth\_ntlm(password\_mgr):

"""

Добавляет модуль ntlm авторизации

password\_mgr - менеджер паролей из которого будет взять

username и password:

https://docs.python.org/3/library/urllib.request.html#httppasswordmgr-objects

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.template\_render,

HTTPNtlmAuthHandler(password\_mgr),

)

)

return builder

return wrapper

def http\_auth\_kerberos(password\_mgr):

"""

Добавляет модуль kerberos авторизации

password\_mgr - менеджер паролей из которого будет взять

username и password:

https://docs.python.org/3/library/urllib.request.html#httppasswordmgr-objects

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddhandler(

builder.context\_key,

builder.template\_render,

HTTPKerberosAuthHandler(password\_mgr),

)

)

return builder

return wrapper

class HttpAddAuthConnIdHandler(PipeTask):

def \_\_init\_\_(

self,

context\_key,

stack\_key,

template\_render,

http\_conn\_id,

handler\_gen,

):

super().\_\_init\_\_(context\_key)

super().set\_template\_fields(

[

"http\_conn\_id",

]

)

super().set\_template\_render(template\_render)

self.stack\_key = stack\_key

self.modules\_key = "modules"

self.http\_conn\_id = http\_conn\_id

self.handler\_gen = handler\_gen

def \_\_call\_\_(self, context):

self.render\_template\_fields(context)

share = context[self.context\_key]

modules: List = share[self.modules\_key]

modules.append(

self.handler\_gen(self),

)

def http\_auth\_preemptive\_basic\_with\_conn\_id(http\_conn\_id):

"""

Добавляет модуль preemptive basic авторизации

Учётка берется из укзаанного Airflow http\_conn\_id

basic авторизация выполняется в два этапа:

- формируется запрос без авторизации

- в случае получения кода 401, формируется второй запрос с Authorization заголовком

preemptive\_basic - выполняется в один этап,

заголовок Authorization отправляются в первом же запросе

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddAuthConnIdHandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

http\_conn\_id,

lambda b: PreemptiveBasicAuthHandler(

AirflowHTTPConnectionPasswordMgr(b.http\_conn\_id)

),

)

)

return builder

return wrapper

def http\_auth\_basic\_from\_conn\_id(http\_conn\_id: str):

"""

Добавляет модуль basic авторизации

Учётка берется из укзаанного Airflow http\_conn\_id

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddAuthConnIdHandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

http\_conn\_id,

lambda b: HTTPBasicAuthHandler(

AirflowHTTPConnectionPasswordMgr(b.http\_conn\_id)

),

)

)

return builder

return wrapper

def http\_auth\_digest\_from\_conn\_id(http\_conn\_id: str):

"""

Добавляет модуль digest авторизации

Учётка берется из укзаанного Airflow http\_conn\_id

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddAuthConnIdHandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

http\_conn\_id,

lambda b: HTTPDigestAuthHandler(

AirflowHTTPConnectionPasswordMgr(b.http\_conn\_id)

),

)

)

return builder

return wrapper

def http\_auth\_informatica\_from\_conn\_id(http\_conn\_id: str):

"""

Добавляет модуль informatica авторизации, которая необходима для rest api

Учётка берется из укзаанного Airflow http\_conn\_id

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddAuthConnIdHandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

http\_conn\_id,

lambda b: InformaticaAuthHandler(

AirflowHTTPConnectionPasswordMgr(b.http\_conn\_id),

),

)

)

return builder

return wrapper

def http\_auth\_ntlm\_from\_conn\_id(http\_conn\_id: str):

"""

Добавляет модуль ntlm авторизации

Учётка берется из укзаанного Airflow http\_conn\_id

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddAuthConnIdHandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

http\_conn\_id,

lambda b: HTTPNtlmAuthHandler(

AirflowHTTPConnectionPasswordMgr(b.http\_conn\_id)

),

)

)

return builder

return wrapper

def http\_auth\_kerberos\_from\_conn\_id(http\_conn\_id: str):

"""

Добавляет модуль kerberos авторизации

Учётка берется из укзаанного Airflow http\_conn\_id

"""

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddAuthConnIdHandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

http\_conn\_id,

lambda b: HTTPKerberosAuthHandler(

AirflowHTTPConnectionPasswordMgr(b.http\_conn\_id)

),

)

)

return builder

return wrapper

def http\_auth\_conn\_id(http\_conn\_id: str, schema: str):

"""

Добавляет универсальный модуль авторизации, согласно переданной схеме

Учётка берется из укзаанного Airflow http\_conn\_id

Поддерживаемые схемы:

- basic

- preemptive\_basic

- digest

- ntlm

- informatica

"""

def schema\_verify(schema, password\_mgr):

match schema:

case "basic":

return HTTPBasicAuthHandler(password\_mgr)

case "preemptive\_basic":

return PreemptiveBasicAuthHandler(password\_mgr)

case "digest":

return HTTPDigestAuthHandler(password\_mgr)

case "ntlm":

return HTTPNtlmAuthHandler(password\_mgr)

case "kerberos":

return HTTPKerberosAuthHandler(password\_mgr)

case "informatica":

return InformaticaAuthHandler(password\_mgr)

case \_:

raise ValueError(f'unsupported authorization schema: "{schema}"')

def wrapper(builder: PipeTaskBuilder):

builder.modules.append(

HttpAddAuthConnIdHandler(

builder.context\_key,

builder.stack\_key,

builder.template\_render,

http\_conn\_id,

lambda b: schema\_verify(

schema, AirflowHTTPConnectionPasswordMgr(b.http\_conn\_id)

),

)

)

return builder

return wrapper

**Приложение 2**

Тесты:

http\_test/dag.py

import json

import gzip

import pendulum

from pathlib import Path

from airflow.models.dag import DAG

from airflow.models.param import Param

from airflow.decorators import task

from airflow.providers.gun.pipe import pipe

from airflow.providers.gun.http import (

http\_body\_bytes,

http\_body\_dict,

http\_body\_file,

http\_body\_iterable,

http\_body\_string,

http\_headers,

http\_req,

http\_get,

http\_post,

http\_run,

http\_response\_body\_to\_stdout,

http\_retry\_if\_code,

http\_error\_supress,

http\_error\_if\_code,

http\_handler,

http\_response\_body\_to\_gzip,

)

from urllib.request import (

Request,

BaseHandler,

)

with DAG(

dag\_id="tests\_\_http\_test",

default\_args={

"owner": "solomatova.ms",

},

start\_date=pendulum.datetime(2021, 1, 1, tz="UTC"),

schedule=None,

catchup=False,

render\_template\_as\_native\_obj=True,

max\_active\_runs=1,

max\_active\_tasks=30,

tags=[],

params={

"get\_url": Param("https://httpbin.org/get"),

"post\_url": Param("https://httpbin.org/post"),

"header\_value\_3": "header value",

"body\_param\_4": "Body Params",

"hello\_world\_file": Param("dags/tests/http\_test/hello world.txt"),

"hello\_world\_string": Param("hello world"),

"http\_save\_response\_to\_gzip": Param("dags/tests/http\_test/hello\_world.gzip"),

},

) as dag:

dag.doc\_md = \_\_doc\_\_

@task.python()

@http\_run()

@http\_req("{{ params.get\_url }}")

@pipe.airflow\_task("pipe")

def http\_1(context):

res = context["pipe"]["res"]

for r in res:

print(r)

http\_1()

@task.python()

@http\_run()

@http\_post("{{ params.post\_url }}")

@pipe.airflow\_task("pipe")

def http\_2(context):

res = context["pipe"]["res"]

for r in res:

print(r)

http\_2()

@task.python()

@http\_run()

@http\_headers(

{

"My-Header": "{{ params.header\_value\_3 }}",

}

)

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_3(context):

res = context["pipe"]["res"]

res = res.read1()

res = json.loads(res)

res = res["headers"]

res = res["My-Header"]

print(res)

assert res == "header value"

http\_3()

@task.python()

@http\_run()

@http\_body\_dict(

{

"my\_params": "{{ params.body\_param\_4 }}",

}

)

@http\_headers(

{

"Content-Type": "application/json",

}

)

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_4(context):

res = context["pipe"]["res"]

res = res.read1()

res = json.loads(res)

res = res["json"]

res = res["my\_params"]

print(res)

assert res == "Body Params"

http\_4()

@task.python()

@http\_run()

@http\_retry\_if\_code([400, 405], 10)

@http\_post("https://google.com", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_5(context):

pass

http\_5()

@task.python()

@http\_run()

@http\_error\_supress()

@http\_post("https://google.com", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_6(context):

res = context["pipe"]["res"]

assert res.code == 405

http\_6()

@task.python()

@http\_run()

@http\_error\_if\_code(

[range(400, 405), 500]

) # 405 не включена в диапазон, ошибки не будет

@http\_post("https://google.com", debuglevel=1) # post http://google.com выдает 405

@pipe.airflow\_task("pipe")

def http\_7(context):

res = context["pipe"]["res"]

assert res.code == 405

http\_7()

@task.python()

@http\_run()

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_file("{{ params.hello\_world\_file }}")

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_send\_transfer\_encoding(context):

res = context["pipe"]["res"]

res = res.read1()

res = json.loads(res)

assert res["data"] == "hello world"

http\_send\_transfer\_encoding()

@task.python()

@http\_run()

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_string("{{ params.hello\_world\_string }}")

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_send\_string\_encoding(context):

res = context["pipe"]["res"]

res = res.read1()

res = json.loads(res)

assert res["data"] == "hello world"

http\_send\_string\_encoding()

@task.python()

@http\_run()

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_bytes(b"hello world")

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_send\_bytes(context):

res = context["pipe"]["res"]

res = res.read1()

res = json.loads(res)

assert res["data"] == "hello world"

http\_send\_bytes()

def iterable\_generator():

yield b"1"

yield b"2"

yield b"3"

yield b"4"

yield b"5"

yield b"6"

@task.python()

@http\_run()

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_iterable(iterable\_generator())

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_send\_iterable\_transfer\_encoding(context):

res = context["pipe"]["res"]

res = res.read1()

res = json.loads(res)

assert res["data"] == "123456"

http\_send\_iterable\_transfer\_encoding()

class HTTP405ErrorSupressorHandler(BaseHandler):

def http\_error\_405(self, req, fp, code, msg, hdrs):

return fp

@task.python()

@http\_run()

@http\_handler(HTTP405ErrorSupressorHandler())

@http\_post("https://google.com", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_custom\_module(context):

res = context["pipe"]["res"]

assert res.code == 405

http\_custom\_module()

@task.python()

@http\_run()

@http\_response\_body\_to\_gzip("{{ params.http\_save\_response\_to\_gzip }}")

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_bytes(b"hello world")

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_save\_response\_to\_gzip(context, saved\_file):

with gzip.open(saved\_file, mode="r") as f:

res = f.read()

res = json.loads(res)

assert res["data"] == "hello world"

http\_save\_response\_to\_gzip("{{ params.http\_save\_response\_to\_gzip }}")

@task.python()

@http\_run()

@http\_response\_body\_to\_stdout()

@http\_headers(

{

"Content-Type": "text/plain",

}

)

@http\_body\_bytes(b"hello world")

@http\_post("{{ params.post\_url }}", debuglevel=1)

@pipe.airflow\_task("pipe")

def http\_save\_response\_to\_stdout(context):

pass

http\_save\_response\_to\_stdout()

if \_\_name\_\_ == "\_\_main\_\_":

dag.test()

**Приложение 3**

Тесты:

ch\_test/dag.py

import pendulum

from airflow.models.dag import DAG

from airflow.models.param import Param

from airflow.decorators import task

from airflow.providers.gun.pipe import pipe

from airflow.providers.gun.ch import (

ch\_auth\_airflow\_conn,

ch\_send\_data\_from\_csv,

ch\_send\_query,

ch\_send\_queries,

ch\_send\_file\_query,

ch\_send\_file\_queries,

ch\_send\_external\_tables,

ch\_recv\_blocks\_to\_stdout,

ch\_save\_to\_csv,

ch\_save\_to\_gzip\_csv,

)

with DAG(

dag\_id="tests\_\_clickhouse\_test",

default\_args={

"owner": "solomatova.ms",

},

start\_date=pendulum.datetime(2021, 1, 1, tz="UTC"),

schedule=None,

catchup=False,

render\_template\_as\_native\_obj=True,

max\_active\_runs=1,

max\_active\_tasks=30,

tags=[],

params={

"conn\_id": Param("clickhouse\_\_spb99dkl-adqm01d"),

"sql\_file": Param("dags/tests/clickhouse\_test/sql\_file.sql"),

"sql\_file\_multi\_statements": Param(

"dags/tests/clickhouse\_test/sql\_file\_multi\_statements.sql"

),

"csv\_file": Param("data/test\_2.csv"),

"gzip\_csv\_file": Param("data/test.gzip"),

},

) as dag:

dag.doc\_md = \_\_doc\_\_

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select 1")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def select\_one\_row(\_):

pass

select\_one\_row()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select 1")

@ch\_auth\_airflow\_conn(

"{{ params.conn\_id }}",

settings={

"client\_name": "my\_super\_name",

},

)

@pipe()

def select\_one\_row\_with\_client\_name(\_):

pass

select\_one\_row\_with\_client\_name()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select 1 union all select 2")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def select\_two\_row(\_):

pass

select\_two\_row()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"select number as n, number+1 as m from numbers(0, 100) format csv",

settings={

"format\_csv\_delimiter": "|",

},

)

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def select\_format\_csv(\_):

pass

select\_format\_csv()

@task.python()

@ch\_send\_queries(

"select 1; select 1 one, 2 two; select version(); show databases; select number+1 as k from numbers(0, 10);"

)

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def queries\_comma\_separated(\_):

pass

queries\_comma\_separated()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"select number from numbers(0, 100000)",

settings={

"max\_block\_size": 65635,

},

)

@ch\_auth\_airflow\_conn(

"{{ params.conn\_id }}",

settings={

"compression": True,

},

)

@pipe()

def select\_dohera\_rows\_with\_compression(\_):

pass

select\_dohera\_rows\_with\_compression()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_external\_tables(

[

{

"name": "ext",

"structure": [("x", "Int32"), ("y", "Array(Int32)")],

"data": [

{"x": 100, "y": [2, 4, 6, 8]},

{"x": 500, "y": [1, 3, 5, 7]},

],

}

]

)

@ch\_send\_query("SELECT sum(x) FROM ext", end\_query=False)

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def select\_with\_external\_tables(\_):

pass

select\_with\_external\_tables()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("show databases")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select version()")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("select 1 as t")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def multiple\_query(\_):

pass

multiple\_query()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_file\_query("{{ params.sql\_file }}")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def file\_query(\_):

pass

file\_query()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_file\_queries("{{ params.sql\_file\_multi\_statements }}")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def file\_queries\_comma\_separated(\_):

pass

file\_query()

@task.python()

@ch\_save\_to\_csv("{{ params.csv\_file }}")

@ch\_send\_query("select number as number, number+1 mumbers from numbers(0, 10000)")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def save\_to\_csv(\_):

pass

save\_to\_csv()

@task.python()

@ch\_save\_to\_gzip\_csv("{{ params.gzip\_csv\_file }}")

@ch\_send\_query("show databases")

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def save\_to\_gzip\_csv(\_):

pass

save\_to\_gzip\_csv()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("SELECT name, comment FROM system.tables where name = 'temp1'")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"create temporary table temp1 (id UInt8) COMMENT 'The temporary table'"

)

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def create\_temp\_table\_and\_select\_description(\_):

pass

create\_temp\_table\_and\_select\_description()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"select \* from temp1",

settings={

"xcx": 3,

},

)

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query("insert into temp1(id) select number from numbers(0, 100)")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"create temporary table temp1 (id UInt8) COMMENT 'The temporary table'",

settings={

"max\_block\_size": 3,

},

)

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def create\_temp\_table\_and\_insert\_data(\_):

pass

create\_temp\_table\_and\_insert\_data()

@task.python()

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_data\_from\_csv("dags/tests/clickhouse\_test/send\_data\_from\_csv\_file.csv")

@ch\_send\_query("insert into temp1 FORMAT CSV")

@ch\_recv\_blocks\_to\_stdout()

@ch\_send\_query(

"create temporary table temp1 (id UInt8) COMMENT 'The temporary table'",

settings={

"max\_block\_size": 3,

},

)

@ch\_auth\_airflow\_conn("{{ params.conn\_id }}")

@pipe()

def insert\_data\_from\_csv\_file(\_):

pass

insert\_data\_from\_csv\_file()

if \_\_name\_\_ == "\_\_main\_\_":

dag.test()

1. кастомный – (от англ. custom) – произвольный, отличающийся от стандартного набора чего-либо, сделанный на заказ и под определенные не типовые требования. [↑](#footnote-ref-1)