Модуль 5. Конвейеры операций.

# Образовательный результат:

В результате прохождения этого модуля слушатели научатся

1. Декомпозировать процесс работы с моделью машинного обучения на отдельные операции
2. Создавать конвейеры операций с использованием Apache Airflow
3. Применять в эксплуатации моделей машинного обучения конвейеры операций, осуществлять запуск и мониторинг выполнения этих операций.

# В этом модуле:

Вы познакомитесь с понятием конвейера операций и инструментами для его создания. Конвейер операций позволяет описывать отдельные процессы и обеспечивать их взаимосвязанное последовательное выполнение. Конвейеры операций часто применяются в проектах машинного обучения для автоматизации отдельных этапов, от сбора данных до вывода модели машинного обучения в эксплуатацию.

Сначала вы узнаете о том какие бывают инструменты для создания и использования конвейеров, а также ключевые понятия и термины.

Затем мы сосредоточимся на одном из самых популярных инструментов Apache Airflow. Вы научитесь создавать и запускать конвейеры и отслеживать их состояние в Airflow.

В качестве практического примера использования Airflow мы рассмотрим одну из уже изученных нам задач машинного обучения “Titanic Disaster”.

Темы, изучаемые в модуле:

1. Теория конвейеров операций.
2. Создание конвейера операций в Apache Airflow.
3. Решение практической задачи с Apache Airflow.

# Модуль 5. Юнит 1. Конвейеры операций.

*Введение:* В этом юните описаны конвейеры операций и сделан обзор инструментов для их создания.

*Содержание юнита:*

Как вы уже знаете (например, из Модулей 1 и 2 данного курса) создание модели машинного обучения состоит из различных этапов, наиболее важные из которых:

* сбор и проверка данных,
* подбор и обучение модели,
* тестирование,
* подготовка к переводу модели в производственную эксплуатацию.

Каждый из этих этапов содержит множество рутинных операций, перечень и содержание которых зависит от конкретного проекта.

1. Сбор и проверка данных
   1. опрос источников данных по расписанию или триггеру,
   2. прием информации от источников в режиме прослушивания,
   3. формирование наборов данных от разнородных источников для использования в обучении модели или формировании аналитического отчета,
   4. проверка качества данных,
   5. предобработка данных, преобразование форматов, разархивирование,
   6. сохранение сформированных наборов данных, датасетов,
   7. передача данных в другие модули проекта.
2. Подбор и обучение модели
   1. настройка окружения для проведения эксперимента, установка необходимых библиотек,
   2. загрузка данных,
   3. проведение эксперимента для выбора оптимальной модели, подбор гиперпараметров,
   4. сохранение артефактов отдельных этапов: набора данных, кода, весов модели, метрик, параметров окружения,
   5. загрузка и запуск нужной версии модели.
3. Тестирование
   1. проверка правильности работы модели,
   2. тестирование устойчивости модели к шумам в данных,
   3. проверка окружения (библиотек, параметров),
   4. проверка быстродействия,
   5. интеграционное тестирование.
4. Подготовка к переводу модели в промышленную эксплуатацию
   1. настройка серверов для развертывания,
   2. сборка всего пакета для установки,
   3. установка и запуск.

Все эти операции взаимосвязаны между собой, некоторые могут выполняться параллельно, выполнение других же может начаться только после того, как выполнен ряд других операций. Удобнее всего такие взаимосвязанные операции представлять в виде ориентированного ациклического графа (DAG, Directed Acyclic Graph). С использованием DAG задачи объединяются в единый конвейер данных. Благодаря наглядности графового представления связи между отдельными узлами, представляющими операции, хорошо видны, что позволяет удобно прослеживать цепочку задач.

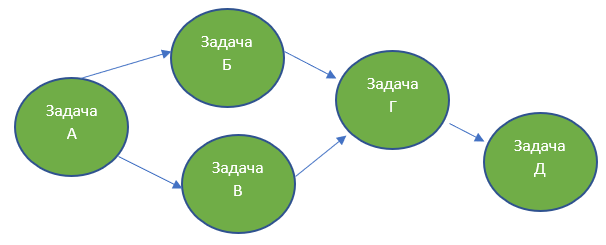


Рисунок «Пример ацикличного направленного графа».

На этом рисунке «Задача Д» является конечной задачей в цепочке, она зависит от выполнения всех остальных предшествующих ей задач. “Задача Б” и “Задача В” не зависят друг от друга, а только от “Задачи А”, поэтому могут выполняться одновременно, что сокращает продолжительность работы всего конвейера.

Анализ задач, выполняющихся последовательно, позволяет определить те из них, которые не зависят друг от друга. Также важно определить продолжительные по времени задачи, сбой выполнения которых может повлиять на другие задачи. Все это позволяет разместить задачи на графе наиболее эффективным образом. Для такого эффективного и надежного управления последовательностями операций необходимо выполнение ряда условий и наличие специальных функций:

* планировщик задач с удобный инструментом для запуска и контроля,
* централизованный мониторинг и обработка ошибок и внештатных ситуаций,
* управление зависимостями, например между функциями/классами,
* возможность восстановления в точке остановки, которая произошла вследствии ошибки или внештатной ситуации,
* удобный графический интерфейс,
* гибкий инструмент контроля, например, возможность использования командной строки.

Инструменты, реализующие набор вышеуказанных методов, активно разрабатываются. Краткий обзор open-source инструментов, которые используются для создания конвейеров операций:

|  |  |  |
| --- | --- | --- |
| Название | Разработчик, ссылка | Описание |
| Luigi | Spotify,  https://github.com/spotify/luigi | Инструмент для запуска и контроля операций. Не может запускать операции по расписанию, но для этих целей можно использовать crontab. Также неэффективно работает с распределением задач, для этих целей применяется Celery. До сих пор достаточно широко распространен в проектах, хотя чаще используется Airflow. |
| Apache Airflow | Разработан Airbnb, позже перешел в open-source Apache Foundition  https://airflow.apache.org | Эффективная обработка сложных потоков операций (workflow), запуск по расписанию, поддержка различных операций, например, bash, запросы в базы данных, работа с хранилищами данных.  Плюсы: удобный функциональный интерфейс, масштабируемость, большое сообщество и хорошая поддержка.  Минусы: негибкий инструмент.  Архитектура: flask, планировщик, воркер. |
| MLFlow | Изначально разработан Databricks, с 2020 года входит в Linux Foundation.  https://mlflow.org | Платформа для реализации полного цикла машинного обучения, упрощает создание и развертывание моделей машинного обучения, а также обмен ими. Инструмент предполагает набор [API](https://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:Application_Programming_Interface_(API)), которые работают с любой библиотекой, в том числе с [PyTorch](https://www.tadviser.ru/index.php/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82:PyTorch), [TensorFlow](https://www.tadviser.ru/index.php/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82:Google_TensorFlow), и XGBoost, а также в любой среде, включая облачные сервисы.  Есть встроенные интеграции с [Docker](https://www.tadviser.ru/index.php/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82:Docker), TensorFlow, PyTorch, [Kubernetes](https://www.tadviser.ru/index.php/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82:Kubernetes), [Java](https://www.tadviser.ru/index.php/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82:Java), [Spark](https://www.tadviser.ru/index.php/Apache_Spark) и другими открытыми проектами.  Позволяет сохранять модель, метрики, перезапускать, следить за экспериментом, сохранять код, артефакты.  Можно получить любую версию модели и развернуть ее. |
| Prefect | <https://www.prefect.io> | Фреймворк с открытым исходным кодом для описания и выполнения цепочек процессов обработки данных с использованием языка Python. Позволяет создавать, запускать и контролировать конвейеры данных различного масштаба. |
| Dagster | <https://dagster.io> | Оркестратор данных для машинного обучения, аналитики и ETL. |
| Oozie | Apache Foundition.  https://oozie.apache.org | Cистема планирования рабочих процессов для управления заданиями в экосистеме для работы с большими данными с Hadoop. |

Кроме того, существует множество других систем создания и контроля конвейеров операций, часто распространена ситуация, когда такая система разрабатывается самостоятельно под свои нужды. Также много коммерческих решений, как правило являющихся частью какого-то глобального решения по работе с данными. Создание системы управления операциями “с нуля” затратно по времени, а использование коммерческих решений не всегда оправдано экономически, поэтому частый сценарий развертывания конвейера операций это использование одного из open-source решений. Наиболее функциональный и популярный в настоящее время Apache Airflow, его рассмотрению и будут посвящены следующие юниты этого модуля.

*Тест, практическое задание*

1. Что такое DAG? (0.25)
   1. Directory Access Gateway
   2. Dynamic Array Gain
   3. **Directed Acyclic Graph**
   4. Digital Alignment Growth
2. Какая компания разработала Luigi? (0.25)
   1. Microsoft
   2. Yandex
   3. **Spotify**
   4. Airbnb
3. Что можно отнести к полезным функциям системы управления цепочками операций? (0.25)
   1. **удобный графический интерфейс**
   2. **возможность просмотра логов**
   3. возможность использования GPU
   4. платформонезависимость
4. С использованием каких инструментом можно описать конвейер операций? (0.25)
   1. **Apache Airflow**
   2. **MLFlow**
   3. pandas
   4. **prefect**

# Итоги/выводы

В этом юните вы узнали о том, как работа с моделью машинного обучения может быть разбита на отдельные операции и об инструментах создания и контроля таких операций. В следующих юнитах мы подробно рассмотрим один из таких инструментов, Apache Airflow.

# Модуль 5. Юнит 2. Apache Airflow.

*Введение:* В этом юните вы подробнее узнаете об одном из самых популярных инструментов для организации и контроля выполнения конвейера операций, Apache Airflow. Вы научитесь устанавливать этот инструмент и настраивать его для работы. Эти знания понадобятся для выполнения практического задания в данном модуле.

*Содержание юнита:*

Apache Airflow это программное обеспечение для создания и управления потоками операций, многофункциональный workflow менеджер. Apache Airflow часто используется инженерами данных и специалистами DevOps/MLOps для автоматизации и контроля выполнения цепочек задач в проектах.

Airflow был создан в 2014 году компанией Airbnb. Через некоторое время Airflow был передан в организацию Apache Foundation (open-source проект), в которой в январе 2019 года получил статус Top-Level проекта. В настоящее время это один из самых популярных open-source инструментов для автоматизации и контроля выполнения операций.

Далее в этом модуле вы узнаете про установку, настройку и запуск конвейера операций в Airflow и создадите свой первый конвейер. Но сначала давайте познакомимся с важными понятиями и определениями Airflow, многие из которых используются и в других подобных инструментах. Понимание этой терминологии необходимо для правильной и эффективной работы с Airflow.

**DAG (Directed Acyclic Graph).** DAG это ориентированный ациклический граф, т.е. граф у которого отсутствуют циклы, но могут быть параллельные пути, выходящие из одного и того же узла. С использованием DAG задачи объединяются в единый конвейер (pipeline). Благодаря наглядности графового представления связи между отдельными узлами, представляющими операции, хорошо видны, что позволяет удобно прослеживать цепочку задач и, при необходимости, оптимизировать.

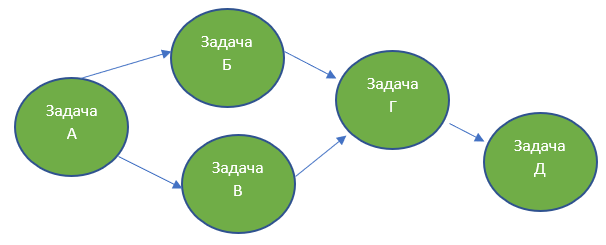


Рисунок «Пример цепочки задач, представленной в виде графа».

**Оператор (Operator).** Оператор в Airflow это звено в цепочке задач. С использованием операторов в Airflow описывается какую задачу необходимо выполнить. В Airflow есть набор готовых операторов для типовых задач, например:

* PythonOperator для исполнения python кода,
* BashOperator для запуска bash скриптов/команд,
* PostgresOperator для вызова SQL запросов в базе данных PostgreSQL,
* EmailOperator для отправки электронных писем.

Полный список стандартных операторов Airflow можно найти [в документации в Apache Airflow](https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/index.html) (https://airflow.apache.org/docs/apache-airflow/stable/\_api/airflow/operators/index.html).

Граф операций DAG графически показывает операторы и их связи между собой, на рисунке выше «Задача А», «Задача Б» и далее это отдельные операторы. Операторы полностью независимы друг от друга, разные операторы могут выполняться на разных аппаратных платформах, в разных программных средах, с разными параметрами. Связь двух операторов в DAG не означает передачу результата работы первого оператора для обработки второму оператору. В этом отличие DAG в Airflow от аналогичной конструкции DAG в Tensorflow, который связывает между собой отдельные функции и результат выполнения предыдущей функции передается в следующую. *В Airflow операции не передают в другие операции результат своей работы.*

**Хук (Hook)**.Хуки это интерфейсы для работы с различными внешними сервисами:

* базы данных, в т.ч. redis, memcached,
* распределенные хранилища, например Amazon S3, Google Disk,
* внешние ресурсы с доступом через API интерфейс.

Хуки являются частями операторов и реализуют логику взаимодействия с хранилищем конфигурационных файлов и доступов. Использование хуков позволяет решить проблему хранения секретной информации в коде (например, пароли к доступам), делая инфраструктуру более безопасной.

**Сенсор (Sensor)**. Сенсоры являются разновидностью операторов, их удобно использовать при реализации конвейеров, в которых необходимо учитывать возникновение определенных событий. Также как и для операторов, для сенсоров существуют типовые варианты, например:

* PythonSensor ожидает, когда функция вернёт True,
* S3Sensor проверяет наличие объекта по ключу в S3-бакете,
* RedisPubSubSensor проверяет наличие сообщения в pub-sub очереди redis,
* RedisKeySensor: проверяет существует ли переданный ключ в redis хранилище.

Полный перечень доступных типовых сенсоров можно найти [в официальной документации Airflow https://airflow.apache.org/docs/apache-airflow/stable/\_api/airflow/sensors/index.html?highlight=sensors#module-airflow.sensors](https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/sensors/index.html?highlight=sensors#module-airflow.sensors). Также есть возможность создать собственный сенсор, для этого можно использовать класс BaseSensorOperator, в котором требуется переопределить метод poke.

**Исполнители (Airflow Executors)**. Исполнители (Executors) в Airflow отвечают за выполнение отдельных задач. В Airflow есть несколько видов типовых исполнителей:

* SequentialExecutor
* LocalExecutor
* CeleryExecutor
* DaskExecutor
* KubernetesExecutor

На практике чаще всего встречается CeleryExecutor, предназначенный для работы с планировщиком задач [Celery](https://khashtamov.com/ru/celery-best-practices/). Рассмотрим подробнее отдельные типы исполнителей.

Исполнитель **SequentialExecutor** установлен в качестве значения по умолчанию в airflow.cfg (параметр executor), представляет из себя простой вид воркера, который не умеет запускать параллельные задачи. В этом случае в конкретный момент времени может выполняться только одна задача. Используется в основном в учебных целях или для простых экспериментов, для продуктивной среды он не подходит.

Исполнитель **LocalExecutor** наиболее похож на продуктивную среду в тестовом окружении или окружении разработки. С его помощью можно выполнять задачи параллельно, например, исполнять несколько DAGов одновременно, путём порождения дочерних процессов. Этот тип исполнителя также не предназначен для производственного окружения из-за ограничений:

* ограничение при масштабировании ресурсами аппаратного обеспечения, на котором он запущен,
* отсутствие отказоустойчивости, если система с этим типом воркера аварийно завершает работу, то задачи перестают исполняться до момента восстановления работоспособности.

Исполнитель LocalExecutor можно эффективно использовать при небольшом количестве задач, т.к. это проще, быстрее и не требует настройки дополнительных сервисов.

**CeleryExecutor** это наиболее часто встречающийся на практике исполнитель, использующий функции Celery для управления потоками выполнения операций. Так же, как и в Celery, в CeleryExecutor необходимо дополнительно настроить брокер сообщений, например Redis или RabbitMQ.

Исполнитель **DaskExecutor** аналогичен CeleryExecutor, но вместо Celery использует [Dask](https://dask.org/), в частности [dask-distributed](https://distributed.dask.org/en/latest/).

Исполнитель **KubernetesExecutor** предназначен для выполнения задач на кластере kubernetes. Многие системы сейчас имеют микросервисную архитектуру, поэтому этот новый вид исполнителя является актуальным для использования, но и одновременно сложным, так как для использования контейнеров и этого исполнителя требуется настроить kubernetes кластер, что является непростой задачей.

В стандартной конфигурации Airflow по умолчанию используется SequentialExecutor, для использования исполнителей других типов необходимо в файле конфигураций airflow.cfg изменить значение параметра executor, например установить значение в LocalExecutor.

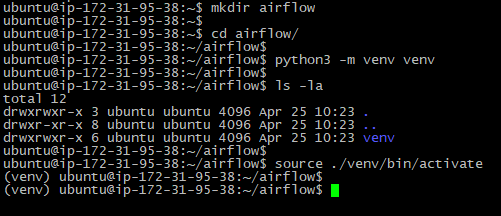
Итак, вы теперь знаете основные сущности Airflow:

* направленный граф операций (DAG),
* операторы (Operators),
* исполнители (Executors),
* хуки (Hooks),
* сенсоры (Sensors).

В этом юните мы рассмотрим использование графов операций, операторов и исполнителей. Примеры использования хуков и сенсоров вы можете найти самостоятельно в документации к Airflow.

Давайте рассмотрим процесс установки и запуска Apache Airflow, который описан [в документации на официальном сайте](https://airflow.apache.org/docs/apache-airflow/stable/installation/index.html) https://airflow.apache.org/docs/apache-airflow/stable/installation/index.html. Существуют различные способы развернуть Airflow, например с использованием docker и helm. Далее рассмотрен самый простой вариант это установка с использованием python установщика pip.

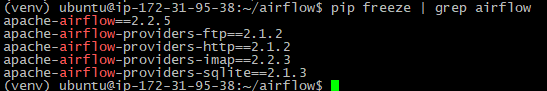
Сначала создадим рабочую папку и виртуальное окружение, в которое будем устанавливать необходимое программное обеспечение



Процесс установки выполним с использованием установщика pip



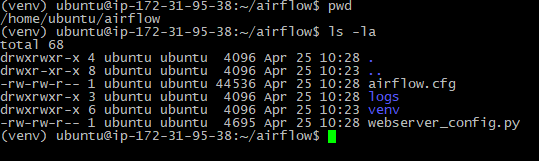
Мы видим установленные пакеты:



Кроме того, сейчас доступна команда **airflow**:

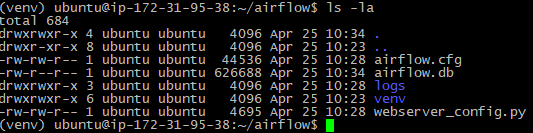


Apache Airflow свои настройки хранит в файле airflow.cfg, который по умолчанию будет создан в домашней директории пользователя с использованием следующего пути  ~/airflow/airflow.cfg.



Для изменения этого пути необходимо переменной окружения AIRFLOW\_PATH присвоить другое значение.

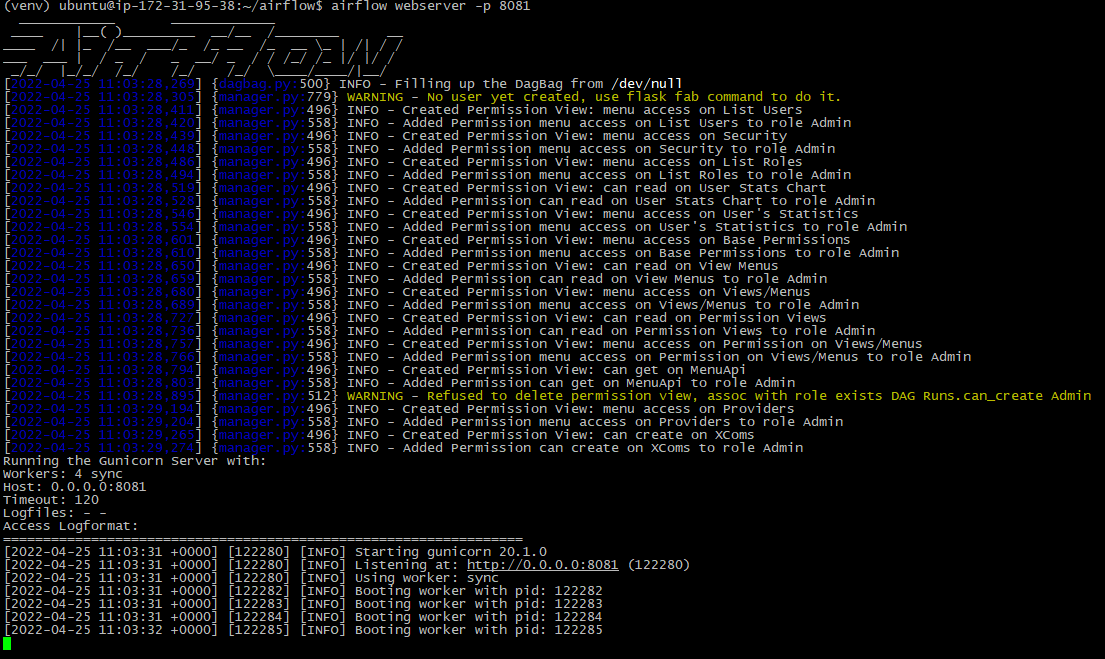
Для начала работы с Airflow требуется выполнить процедуру инициализации базы данных. При этом создаются служебные таблицы в базе данных, необходимые для работы в Airflow. По умолчанию это база данных sqlite. Если вы имеете опыт работы с фреймворком Django, то увидите сходство в этой процедуре с инициализацией служебной базы данных Django. В более ранних версиях эта операция делалась с использованием команды **airflow initdb**, сейчас она заменена на команду **airflow db init**. Эта команда создаст необходимые таблицы в базе данных, в рабочей папке появится файл airflow.db.

****

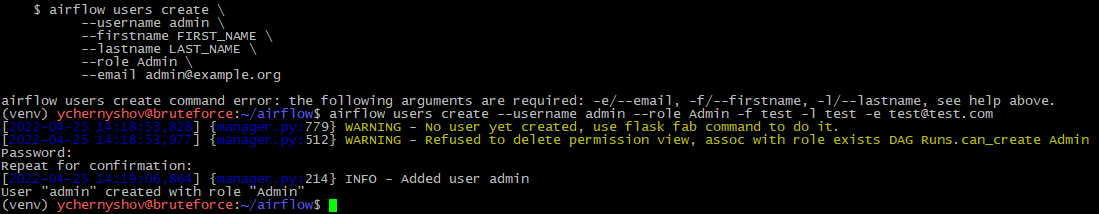
Файл базы данных \*.db является файлом sqlite, которая используется в Airflow по умолчанию. *Sqlite не подходит для производственной среды, поскольку не обладает достаточной производительностью, поэтому при развертывании в производственной среде рекомендуется использовать Postgresql или mysql*. Здесь опять же уместна аналогия с Django, где также sqlite используется в качестве базы данных по умолчанию для учебных целей или простых проектов, который заменяется на postgresql или другие более производительные базы данных при выводе проекта в производственное окружение.

Для запуска графического web интерфейса для работы с Airflow необходимо запустить веб сервер с помощью команды

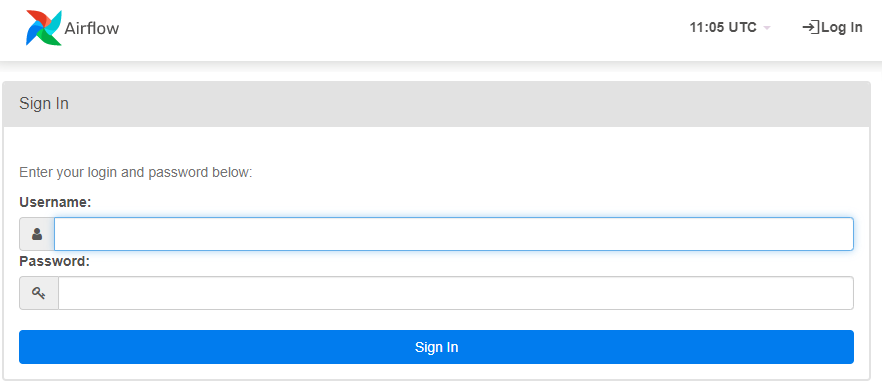
**airflow webserver -p “номер порта”**



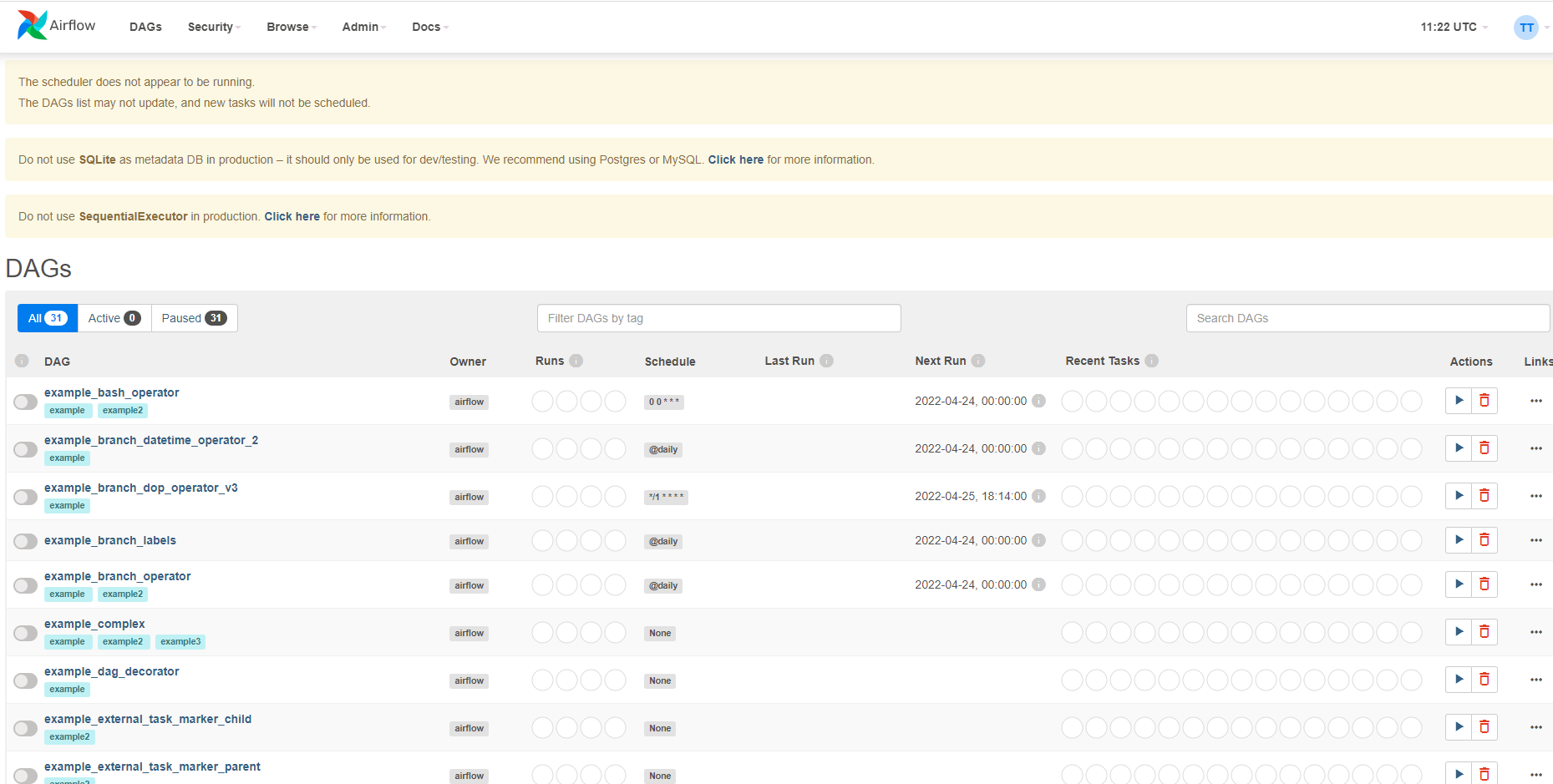
Для работы в системе понадобится пользователь и пароль, который создается с использованием команды **airflow users create**



Теперь можно по адресу **http://”имя или IP адрес сервера”:”номер порта”** открыть графический web интерфейс для работы (в котором предсказуемо вас попросят сначала авторизоваться)



Теперь необходимо ввести регистрационные данные, после чего вы увидите главное меню для работы с Airflow.



Сразу же можно увидеть несколько уведомлений, например, что не запущен планировщик задач (“The scheduler does not appear to be running”). Также можно увидеть две полезные рекомендации: не использовать базу данных SQLite и исполнитель SequentialExecutor в продуктивной среде, однако на этапе обучения для наших простых проектов можно с этим смириться. Тем не менее нужно помнить, что установка и настройка Airflow сильно зависят от характеристик проекта. При использовании Airflow в производственном окружении необходимо использовать более производительные и безопасные компоненты.

На главной странице веб сервера Airflow можно увидеть различные готовые примеры, например, example\_bash\_operator, в котором демонстрируются возможности работы с bash в Airflow.

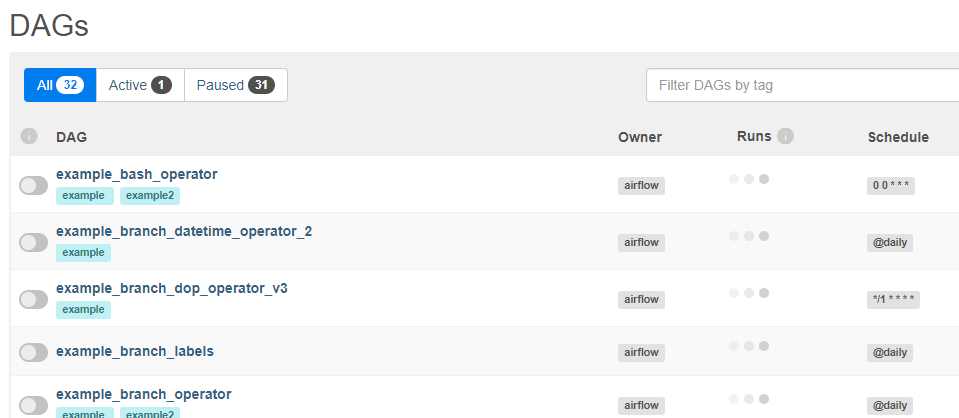


Рисунок “Различные примеры Airflow”

Использование этих примеров позволяет быстро создавать свои конвейеры операций. Например, кликнув на example\_bash\_operator, можно увидеть его структуру

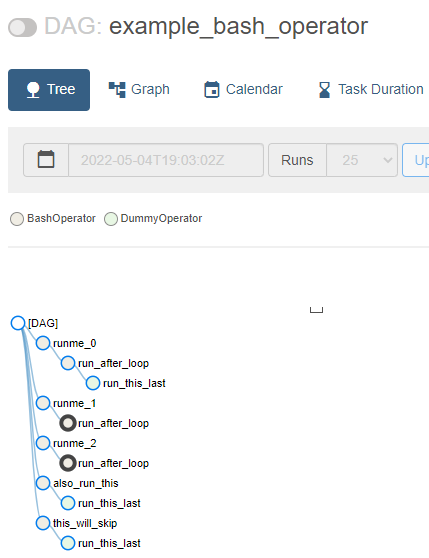


Рисунок “Древовидный вид конвейера операций example\_bash\_operator”

Также для удобства можно открыть этот конвейер в виде графа, выбрав вкладку “Graph”

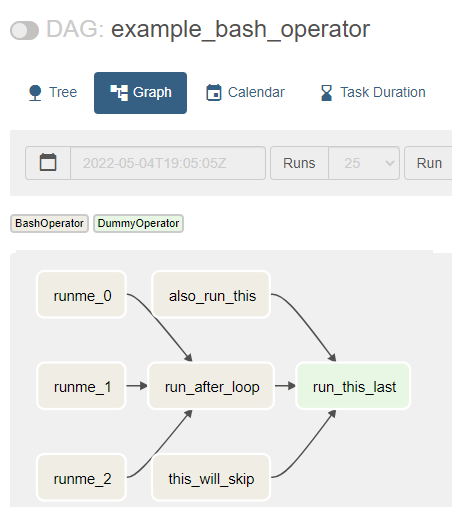


Рисунок “Конвейер example\_bash\_operator в виде графа”.

И, наконец, можно взглянуть на программный код операций, выбрав вкладку “Code”

Содержание программного кода, позволяющего автоматизировать цепочку выполнения bash операций, кода следующее:

*# Licensed to the Apache Software Foundation (ASF) under one*

*# or more contributor license agreements. See the NOTICE file*

*# distributed with this work for additional information*

*# regarding copyright ownership. The ASF licenses this file*

*# to you under the Apache License, Version 2.0 (the*

*# "License"); you may not use this file except in compliance*

*# with the License. You may obtain a copy of the License at*

*#*

*# http://www.apache.org/licenses/LICENSE-2.0*

*#*

*# Unless required by applicable law or agreed to in writing,*

*# software distributed under the License is distributed on an*

*# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY*

*# KIND, either express or implied. See the License for the*

*# specific language governing permissions and limitations*

*# under the License.*

*"""Example DAG demonstrating the usage of the BashOperator."""*

**import** **datetime**

**import** **pendulum**

**from** **airflow** **import** DAG

**from** **airflow.operators.bash** **import** BashOperator

**from** **airflow.operators.dummy** **import** DummyOperator

**with** DAG(

dag\_id='example\_bash\_operator',

schedule\_interval='0 0 \* \* \*',

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

catchup=**False**,

dagrun\_timeout=datetime.timedelta(minutes=60),

tags=['example', 'example2'],

params={"example\_key": "example\_value"},

) **as** dag:

run\_this\_last = DummyOperator(

task\_id='run\_this\_last',

)

*# [START howto\_operator\_bash]*

run\_this = BashOperator(

task\_id='run\_after\_loop',

bash\_command='echo 1',

)

*# [END howto\_operator\_bash]*

run\_this >> run\_this\_last

**for** i **in** range(3):

task = BashOperator(

task\_id='runme\_' + str(i),

bash\_command='echo "{{ task\_instance\_key\_str }}" && sleep 1',

)

task >> run\_this

*# [START howto\_operator\_bash\_template]*

also\_run\_this = BashOperator(

task\_id='also\_run\_this',

bash\_command='echo "run\_id={{ run\_id }} | dag\_run={{ dag\_run }}"',

)

*# [END howto\_operator\_bash\_template]*

also\_run\_this >> run\_this\_last

*# [START howto\_operator\_bash\_skip]*

this\_will\_skip = BashOperator(

task\_id='this\_will\_skip',

bash\_command='echo "hello world"; exit 99;',

dag=dag,

)

*# [END howto\_operator\_bash\_skip]*

this\_will\_skip >> run\_this\_last

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

dag.cli()

Разберем этот код подробнее.

Сначала устанавливаются настройки DAG

**with** DAG(

dag\_id='example\_bash\_operator',

schedule\_interval='0 0 \* \* \*',

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

catchup=**False**,

dagrun\_timeout=datetime.timedelta(minutes=60),

tags=['example', 'example2'],

params={"example\_key": "example\_value"},

) **as** dag

DAG имеет следующие параметры:

* dag\_id - идентификатор DAG,
* shedule\_interval - расписание запуска в формате crontab,
* start\_date - время старта,
* catchup - позволяет рассматривать формальную дату начала работы (start\_date) как текущую дату,
* dagrun\_timeout - таймаут между запусками DAG,
* tags - теги, по которым можно найти DAG, например, показываются в графическом интерфейсе,
* params - параметры DAG.

Далее в DAG добавляются операторы. Например, добавляется “пустой” оператор DummyOperator, который завершает всю цепочку операторов. Этот оператор получает идентификатор run\_this\_last.

run\_this\_last = DummyOperator(

task\_id='run\_this\_last',

)

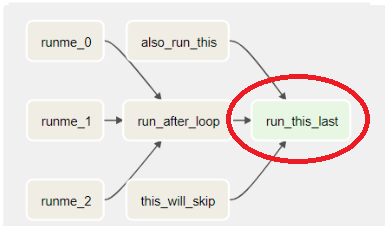


Рисунок “Пустой оператор run\_this\_last”

Затем создается bash оператор, который выполняет команду “echo 1”, этому оператору присваивается идентификатор run\_after\_loop, оператор сохраняется под именем run\_this, которая используется ниже для задания последовательности выполнения операторов

*# [START howto\_operator\_bash]*

run\_this = BashOperator(

task\_id='run\_after\_loop',

bash\_command='echo 1',

)

*# [END howto\_operator\_bash]*

После этого мы можем установить последовательность выполнения операторов с использованием оператора побитового сдвига >>

run\_this >> run\_this\_last

Поясним алгоритм работы этого оператора. Например, при выполнении команды

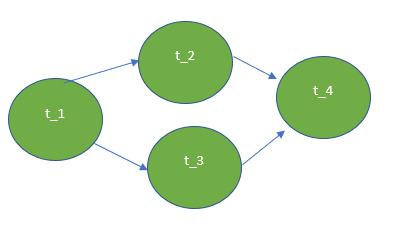
**t\_3 >> t\_2 >> t\_1**

сначала выполнится задача t\_3, затем t\_2 и после нее t\_3. Параллельное выполнение выполнение отдельных задач задается следующим образом:

**t\_1 >> t\_2 >> t\_4**

**t\_1 >> t\_3 >> t\_4**

В результате получается следующий граф операций

****

Таким образом, задачи t\_2 и t\_3 будут выполняться параллельно. Задача в DAG начнет выполнение только в том случае, если предшествующие ей задачи были успешно выполнены. То есть t\_4 начнет выполнение, после успешного завершения t\_2 и t\_3.

Далее везде используется аналогичная нотация. Например можно добавить сразу три оператора, выполняющихся одновременно:

**for** i **in** range(3):

task = BashOperator(

task\_id='runme\_' + str(i),

bash\_command='echo "{{ task\_instance\_key\_str }}" && sleep 1',

)

task >> run\_this

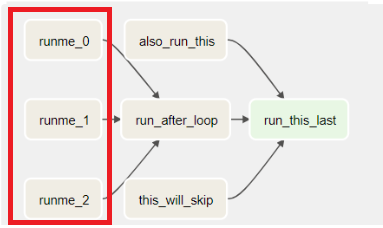


Рисунок “”

Из этого и остальных примерах работы с DAG Airflow легко понять основные правила и синтаксис скрипта python, описывающего DAG. В следующем юните мы рассмотрим создание собственного DAG.

*Тест.*

1. Какие существуют способы установки Airflow? (0.25)
   1. **pip**
   2. **docker**
   3. установочный exe-файл для windows
   4. apt get
2. Что является сущностями Airflow? (0.25)
   1. **Исполнители**
   2. Постановщики
   3. **Операторы**
   4. Клиенты
3. С помощью какого оператора задается последовательность выполнения операций в Airflow? (0.25)
   1. >
   2. ->
   3. **>>**
   4. from…to…
4. Что не рекомендуется делать при запуске Airflow в производственной среде? (0.25)
   1. **использовать sqlite в качестве служебной базы данных**
   2. использовать Postgresql в качестве служебной базы данных
   3. **использовать SequentialExecutor**
   4. разворачивать Airflow в docker контейнере

# Итоги/выводы

В этом юните вы подробно изучили Apache Airflow, позволяющий описывать и контролировать выполнение операций. Теперь вы знаете основные сущности Airflow: граф операций DAG, операторы, хуки, сенсоры, исполнители. Также вы знаете как установить и настроить Airflow для работы. В следующем юните вы создадите свой первый конвейер операций в Airflow.

# Модуль 5. Юнит 3. Создание конвейера операций в Airflow.

*Введение:* В этом юните вы создадите простой конвейер операций в Airflow. В качестве примера используем уже хорошо знакомый нам набор данных из задачи “Titanic Disaster”.

*Содержание юнита:*

Теперь давайте построим наш первый конвейер данных в Apache Airflow. Для работы с Airflow необходимо знать python, так как код, описывающий DAG, пишется именно на python.

В файле настроек airflow.cfg есть параметр dags\_folder, он указывает на путь, где лежат файлы с описаниями DAG. По умолчанию значение переменной dags\_folder равно $AIRFLOW\_HOME/dags. В эту папку необходимо скопировать код с описанием конвейера операций, который вы создадите далее.

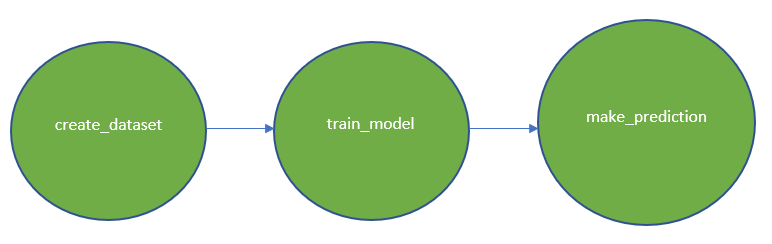
В качестве примера будем решать ту же задачу, что и в Юните 5 Модуля 2, в котором вы автоматизировали решение задачи «Titanic Disaster» с использованием Jenkins. Для создания конвейера операций нам понадобятся три функции:

* create\_dataset() – создает набор данных
* train\_model() – обучает модель
* make\_prediction() – выполняет предсказание

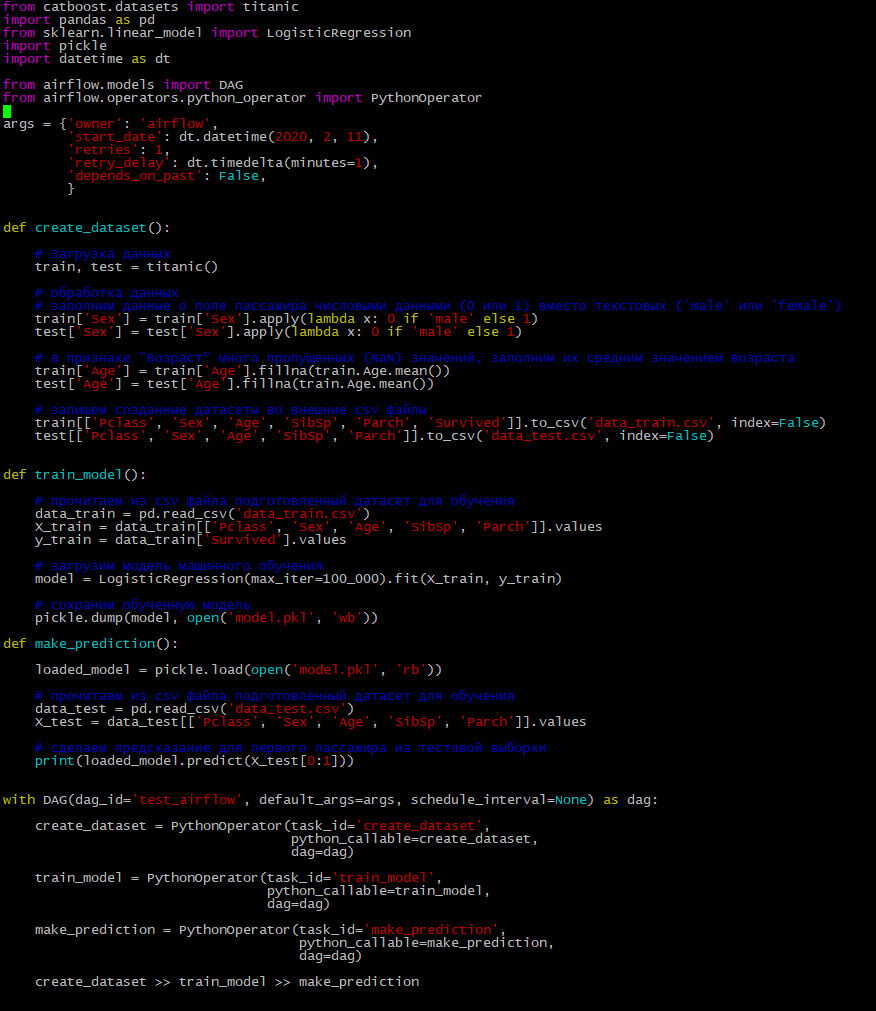
Кроме того, необходимо установить настройки DAG в Airflow, к которым относятся

* Время начала выполнения конвейера(start\_date)
* Периодичность запуска (schedule\_interval)
* Информация о владельце DAG (owner)
* Количество повторений в случае неудач (retries)
* Пауза между повторами (retry\_delay)

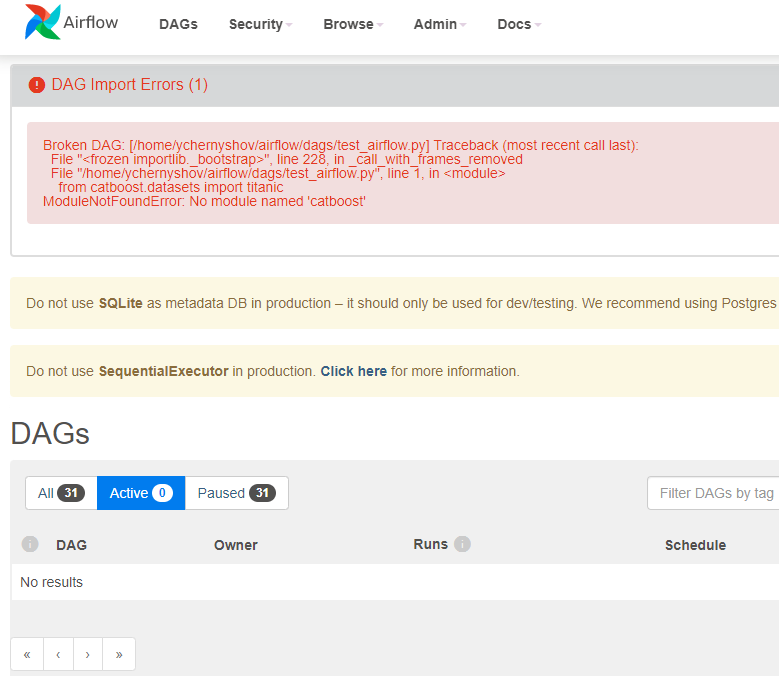
Схематично конвейер операций DAG, который мы планируем построить, выглядит так:



Код python файла, который необходимо загрузить в папку dags, приведен в листинге ниже (код имеет простую и понятную структуру, рекомендуем вам его изучить и попробовать написать самостоятельно, обращаясь к листингу в случае затруднений):



После того как мы выполнили перечисленные выше действия в web интерфейсе панели управления Airflow появится информация об описанном нами DAG, либо сообщение об ошибке, если мы что-то сделали неправильно. Например, мы не установили библиотеку catboost в нашем виртуальном окружении, поэтому нам Airflow напоминает об этом:

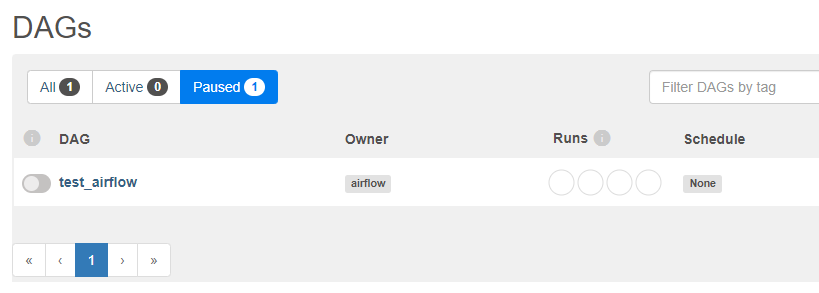


То же самое относится и к библиотеке sklearn, хотя и Airflow не сразу пишет об этом. Поэтому надо установить эти две библиотеки с помощью команд

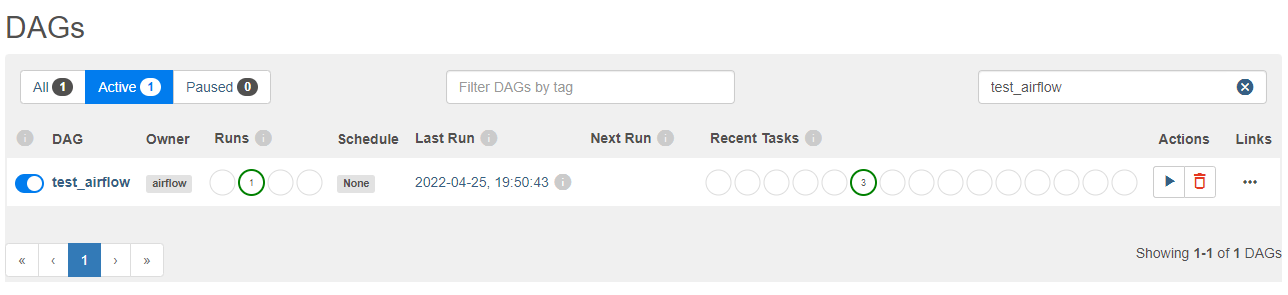
**pip install catboost**

**pip install scikit-learn**

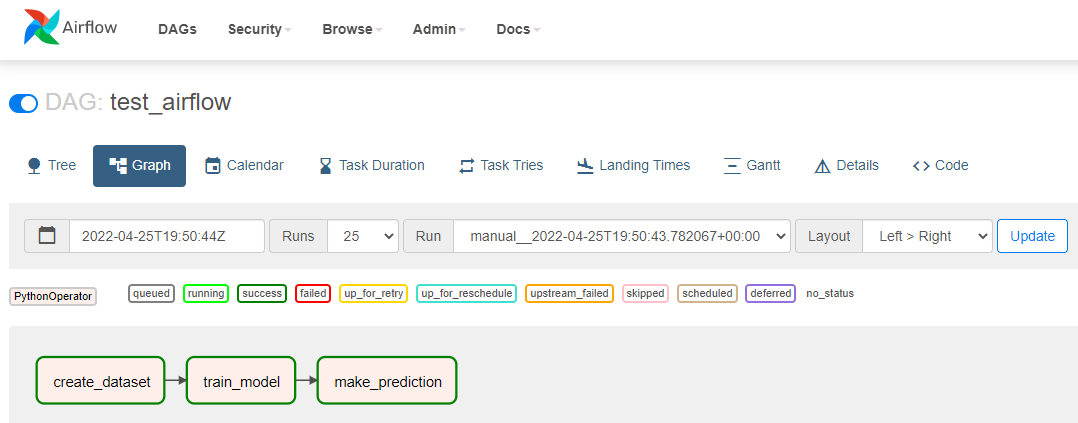
После этого вы увидите созданный DAG в списке Airflow



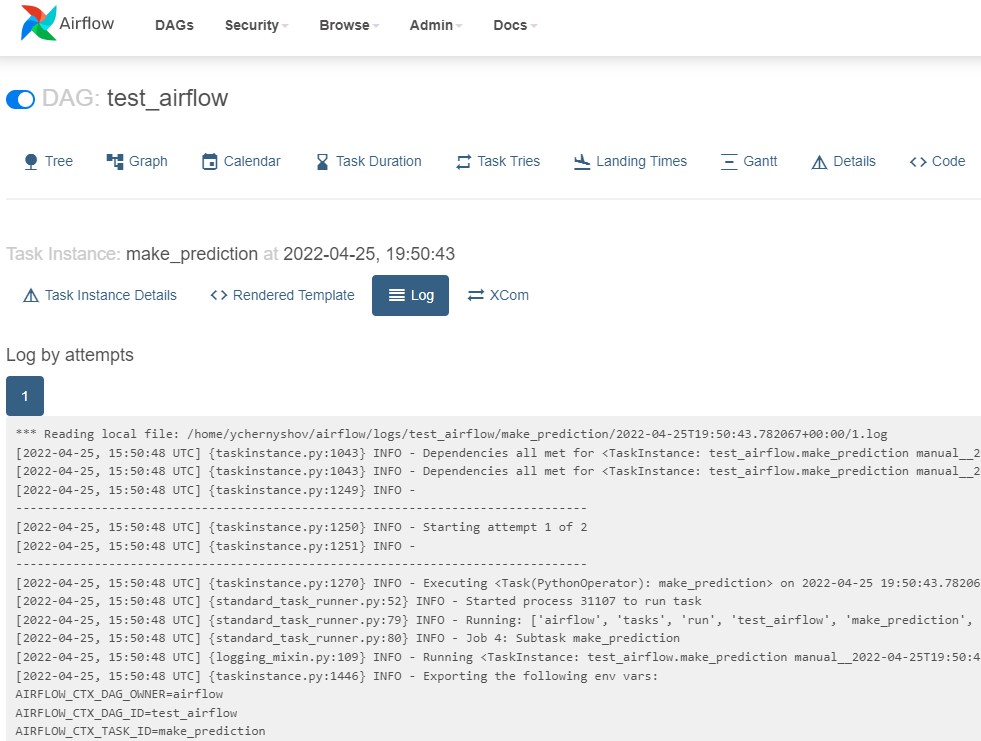
Вы можете запустить выполнение этого конвейера, нажав на треугольник в части «Actions» справа. После этого зелеными кругами будут отмечены успешные запуски:

**

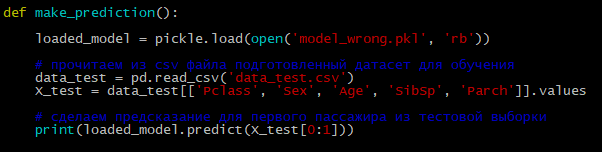
Теперь можно посмотреть детализацию нашего конвейера, например, увидеть цепочку успешно выполненных операций.



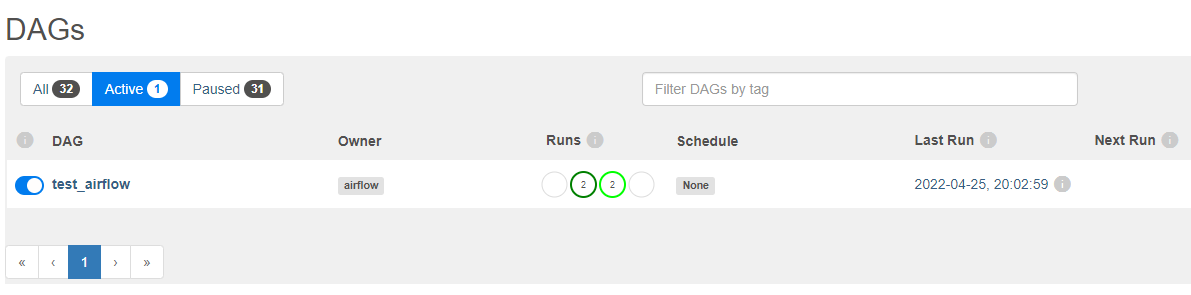
Например, выбрав отдельную операцию и нажав на кнопку «Log» можно увидеть log-файл выполнения этой операции.

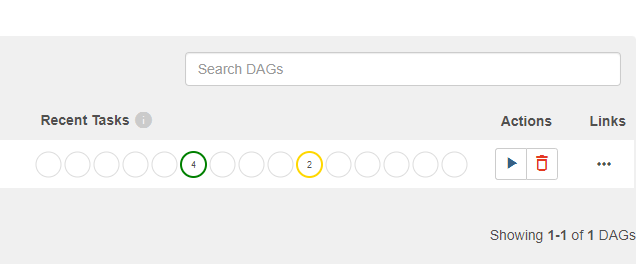


Давайте теперь намеренно “испортим” операцию model\_predict(), чтобы увидеть в Airflow проблемы с ее выполнением. Например, укажем имя несуществующего pickle-файла, из которого функция make\_prediction() пытается загрузить веса модели:

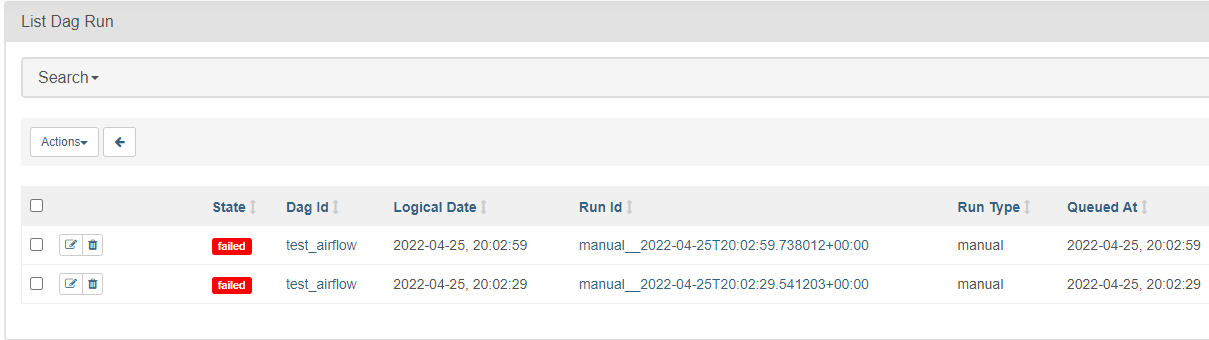


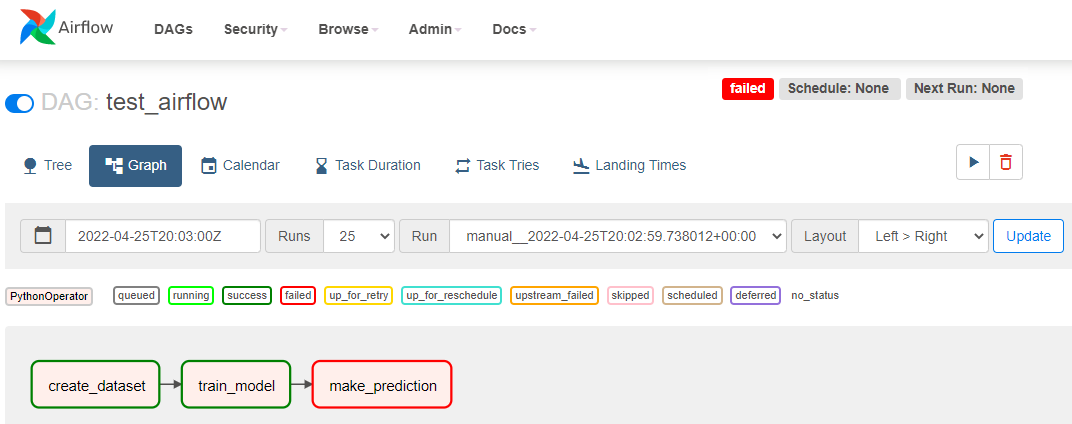
Осуществив запуск этого конвейера мы увидим, что система пытается запускать операцию, однако при этом последняя функция возвращает ошибку

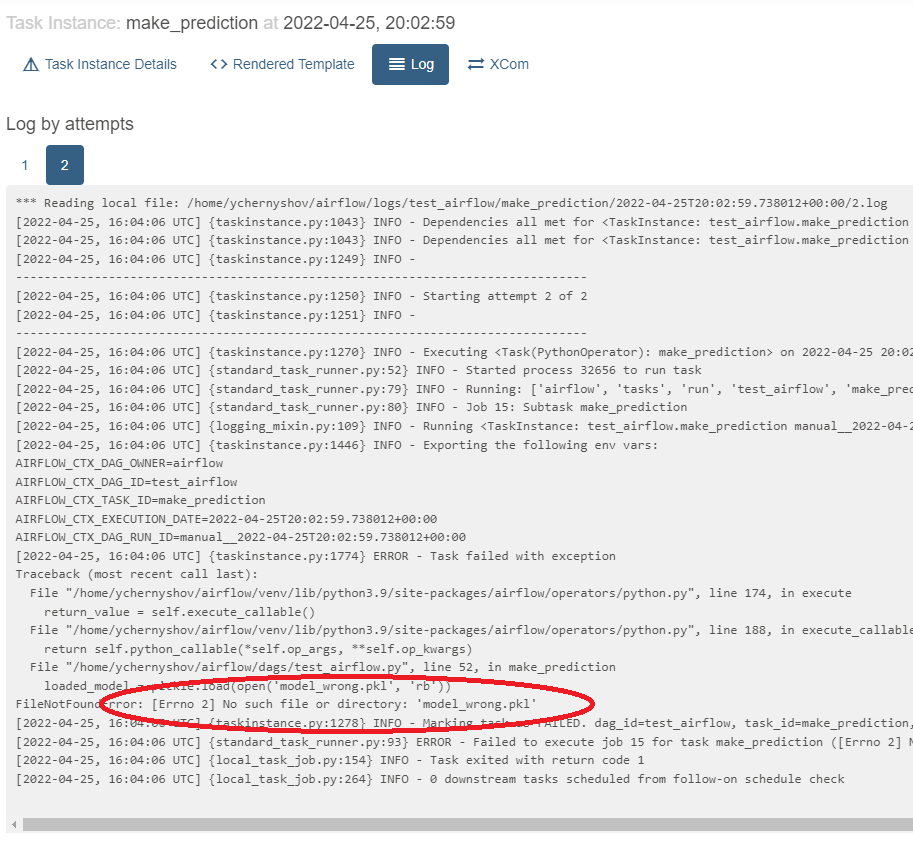




Посмотрев детализацию в log-файле мы увидим проблему:







Итак, вы создали конвейер в Airflow, автоматизирующий операции подготовки датасета, обучению модели и выполнению инференса.

*Тест*

1. В каком параметре airflow.cfg хранится информация о папке с файлами, описывающими DAG? (0.25)
   1. dag\_path
   2. **dag\_folder**
   3. dag\_files
   4. dag\_info
2. Какой параметр DAG устанавливает интервал периодического запуска задач? (0.25)
   1. pause
   2. time\_delay
   3. **schedule\_interval**
   4. task\_delay
3. Какая переменная окружения определяет путь до домашней директории Airflow? (0.25)
   1. AIRFLOW\_PATH
   2. AIRFLOW\_PLACE
   3. AIRFLOW\_DIR
   4. **AIRFLOW\_HOME**
4. Какая команда запускает графическое веб-приложение для работы с airflow? (0.25)
   1. airflow start
   2. **airflow webserver**
   3. airflow run
   4. airflow go

# Итоги/выводы

В юните вы научились создавать и запускать свой собственный конвейер операций для работы с типовым набором данных “Titanic Disaster”. Конечно, применение Airflow для решения задачи “Titanic Disaster” явно избыточно. Основная цель данного юнита заключалась в том, чтобы на конкретной задаче продемонстрировать работу с Airflow. Практический учебный опыт применения Airflow на простых задачах позволяет лучше разобраться с инструментом и быстрее перейти к использованию Airflow в больших проектах со сложной структурой и большим количеством операций, для которых он и предназначен.

# Итоги/выводы по модулю

В этом модуле были рассмотрены конвейеры операций и различные инструменты для их создания. Более подробно был рассмотрен самый популярный инструмент Apache Airflow. Конвейеры операций позволяют решать самые разные задачи в проектах разработки программного обеспечения. Основное их назначение: контролировать выполнение большого числа взаимосвязанных операций.

# Практическое задание

Цель задания: закрепить практические навыки работы с Apache Airflow.

Этапы выполнения задания:

1. создать виртуальную среду выполнения, например, с использованием venv
2. установить Airflow
3. создайте конвейер операций из следующих шагов:
   1. создайте искусственно 100 текстовых файлов, состоящих из символов латинского алфавита с помощью скрипта предоставленного преподавателем
   2. создание 100 параллельных процессов для подсчета количества вхождений символа ‘a’, по одному процессу на файл, результат необходимо записать в текстовый файл n.res, где n - номер исходного файла
   3. создать итоговый процесс, который выполняется после п.3.б, суммирующий данные во всех файлах \*.res.

Что необходимо получить в итоге: сумму вхождений символа ‘a’ во все исходные файлы. В качестве результата опубликовать ссылку на git репозиторий с соответствующим кодом.

# Список источников

<https://airflow.apache.org/docs/>

<https://www.youtube.com/watch?v=cVDIbEsCTow>

<https://www.bigdataschool.ru/blog/airflow-use-cases.html>

https://www.bigdataschool.ru/wiki/airflow