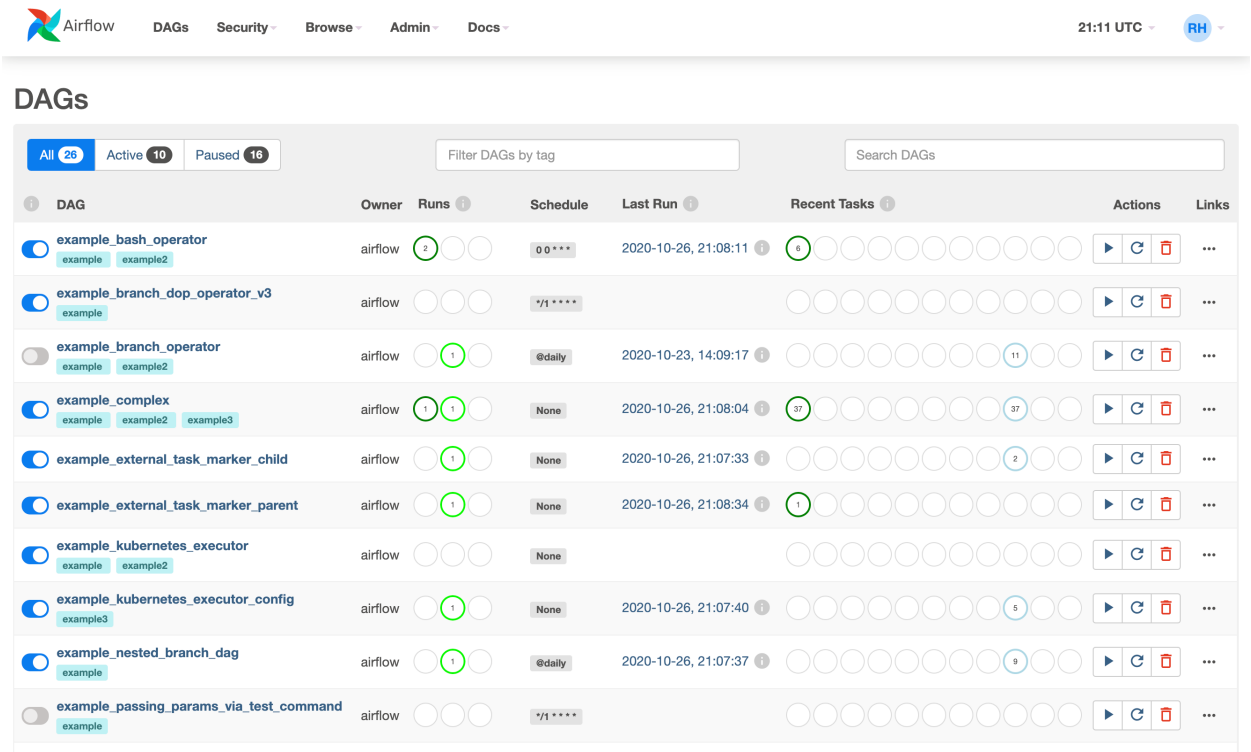
> ETL-ПАЙПЛАЙН

> Airflow: идея и интерфейс

Airflow — это библиотека, позволяющая очень легко и удобно работать с расписанием и мониторингом выполняемых задач. Как уже знакомый вам GitLab CI/CD — можете потом выбрать, какой планировщик задач вам нравится больше!

В целом Airflow — это удобный инструмент для решения **ETL**-задач (**E**xtract->**T**ransform->**L**oad). Чуть подробнее про концепцию тут, а также на нашем канале.

Интерфейс Airflow выглядит следующим образом:

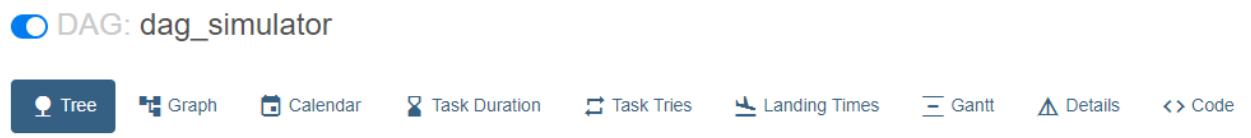


Здесь мы видим большую подпись **DAGs**. **DAG** — это основная единица работы с Airflow, мы обсудим его подробнее в другой части конспекта. Для начала можем считать, что это некоторая глобальная задача, решаемая путем последовательного выполнения более мелких, редуцированных задач.

Интерфейс:

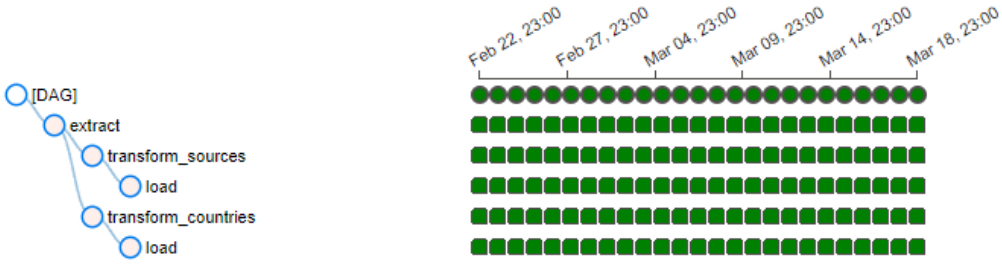
На главной страничке у нас перечислены все доступные DAGи, вкладки **All**, **Active** и **Paused** позволяют фильтровать DAGи в соответствии с состоянием их выполнения. У каждого DAGа стоит переключатель, отвечающий за то, активен ли DAG или нет, затем идет название, владелец, информация о запусках и их состояниях, расписание (в формате Cron), информация по последним выполненным задачам и некоторые хот-кеи для работы с DAGом: запуск мгновенно, перезагрузить и удалить.

Если открыть конкретный DAG, можно увидеть больше информации о нём:

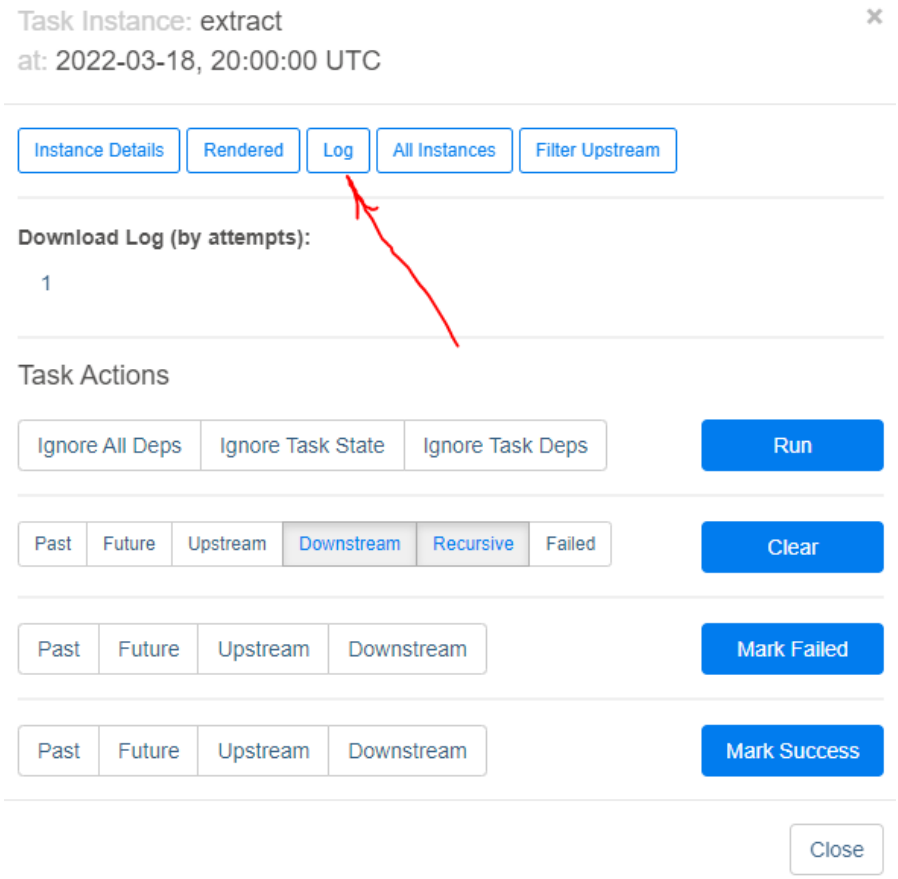


Здесь можно визуализировать DAG, смотреть на его расписание, продолжительность выполнения «маленьких» задач, количество запусков DAGа, а также другие интересные вещи.

«Древесное» отображение DAGа удобно тем, что можно отслеживать историю выполнения задач в DAGе:

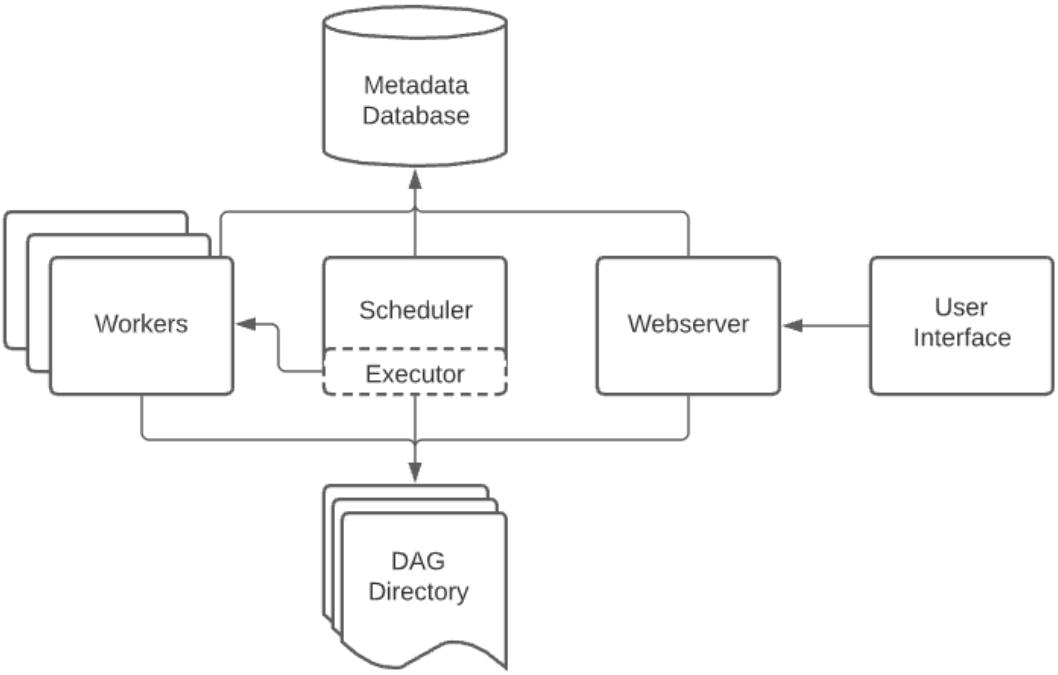


Если что-то сломалось (задача выделена не зелёным цветом, а каким-то другим), можно нажать на неё и в выбранном меню открыть логи. Это позволит вам отследить, из-за чего произошла поломка.



> Архитектура Airflow

В общем случае её можно выразить через эту картинку:



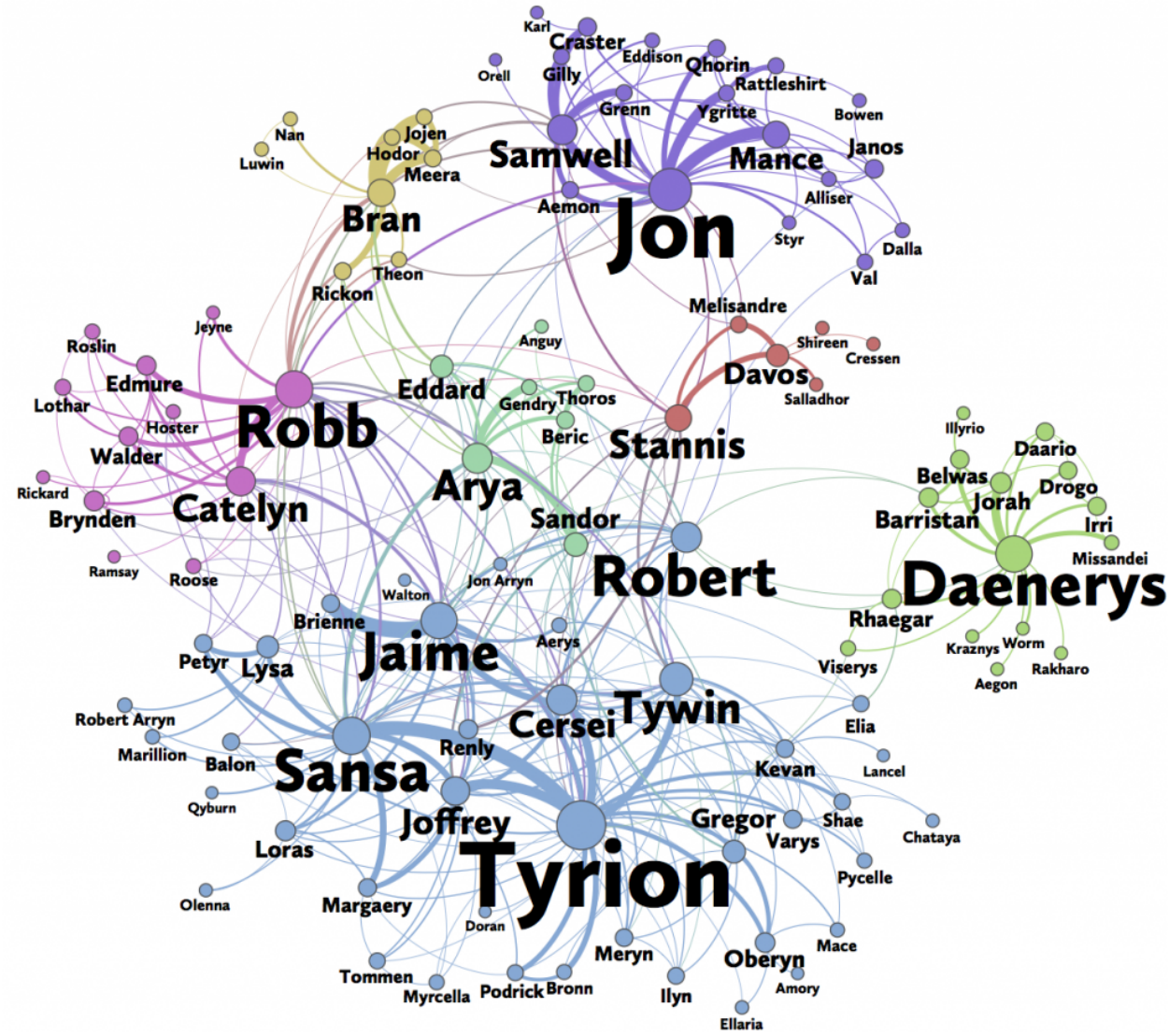
1. **Webserver** и **User Interface** — это то, что мы видели с вами на прошлом шаге конспекта. Здесь всё то, что видит пользователь и с чем может взаимодействовать напрямую.
2. **Scheduler** и **Executor** — запуск задач по расписанию и их исполнение. В реальной работе Executor часто «делегирует» исполнение задачи «рабочим» — **Workers**.
3. **DAG Directory** — место, где лежат сами задачи, объединённые в DAG-и.
4. **Metadata Database** — тут хранятся логи и другая полезная информация о состоянии Airflow

> Что такое DAG?

DAG расшифровывается как **Directed Acyclic Graph** — **направленный ациклический граф**. Стоит расшифровать каждое из этих слов, чтобы была понятна общая идея, после чего соотнести с процессами в Airflow.

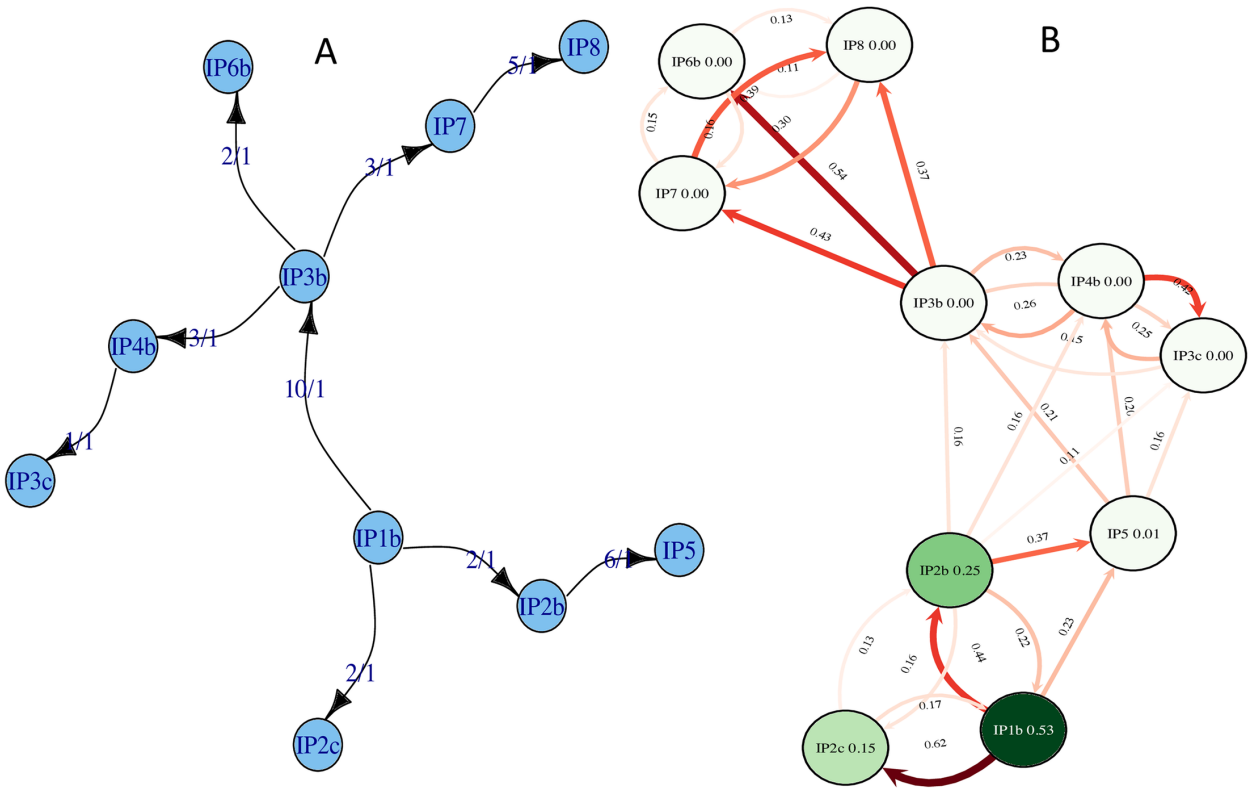
Граф

Грубо говоря, граф — это математическая абстракция, отражающая множество связанных между собой элементов. Например, ниже приведён граф связей между героями Игры Престолов:



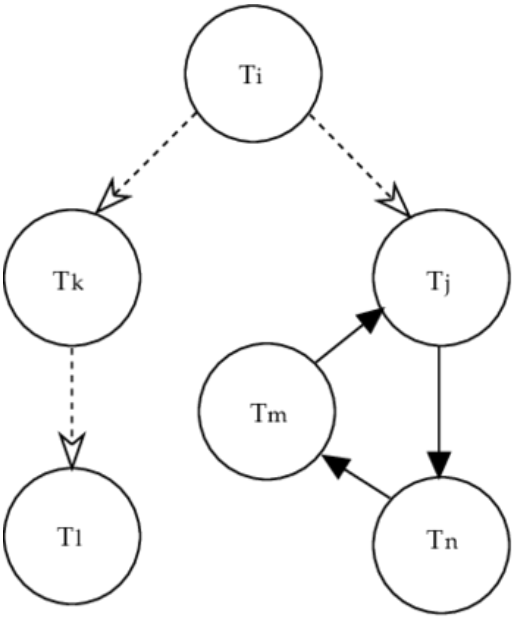
Направленность

Порой в графе мы можем лишь констатировать наличие либо отсутствие связи. Но в некоторых графах мы можем говорить о конкретном её **направлении**. Например, ниже приведены два графа, отражающие возможное распространение эпидемии между больными:

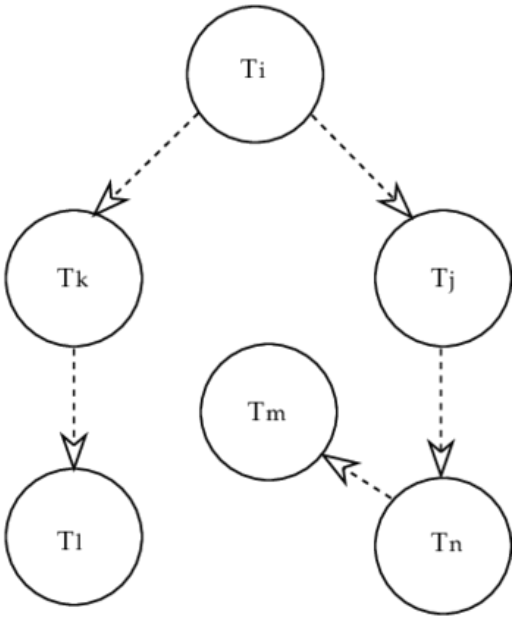


Ацикличность

В некоторых направленных графах развитие процесса может развиваться в строго определённом направлении — вернуться обратно к тому же элементу невозможно, если ты уже из него вышел. Такие графы называют **ациклическими**, потому что в них **нет циклов**. Пример можно видеть ниже:



(a) Wait-for graph with a cycle

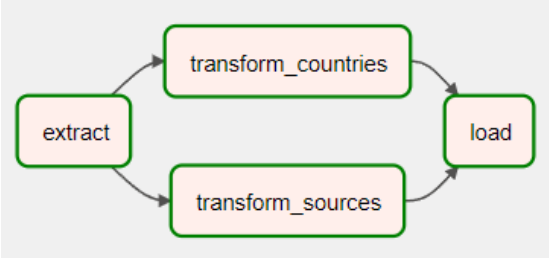


(b) Wait-for graph with no cycles

Граф с циклом (слева) и без циклов (справа)

Какое это отношение имеет к Airflow?

Взглянем на типичный DAG в Airflow:



1. Это вполне похоже на **граф** — его элементами являются отдельные задачи (их ещё называют **тасками**), которые связаны друг с другом.
2. Характер их взаимодействия имеет строго определённое **направление**: сначала срабатывает таск **extract**, на основе её результатов работают таски **transform_countries** и **transform_sources**, а уже их «выхлоп» идёт в таск **load**.
3. Здесь **нет циклов** — в рамках одного запуска DAG-а каждый таск отрабатывает ровно один раз, без возвращения к прошлым. Почему это так важно? Если бы у нас был хотя бы один цикл, то наш процесс застрял бы в нём навечно — не было бы какого-то правила, позволившего выйти из этого порочного круга. Правило ацикличности по определению разрешает подобную проблему.

Заметим, что ацикличность — это свойство исключительно DAG-а. Внутри тасок такого ограничения нет — пользуйтесь циклами на здоровье :)

> Структура DAG

Блок импортов

Здесь мы импортируем всё то, что нам нужно для создания DAG-а и его «содержания»:

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator # Так как мы пишет такси в питоне
from datetime import datetime
```

Блок кода

Здесь вы прописываете ваши функции:

```
def foo1():
def foo2():
def foo3():
```

Блок инициализации

Задаем параметры в DAG:

```
default_args = {
    'owner': 'your_name', # Владелец операции
    'depends_on_past': False, # Зависимость от прошлых запусков

    'retries': 1, # Кол-во попыток выполнить DAG
    'retry_delay': timedelta(minutes=5), # Промежуток между перезапусками

    'email': '', # Почта для уведомлений
    'email_on_failure': '', # Почта для уведомлений при ошибке
    'email_on_retry': '', # Почта для уведомлений при перезапуске

    'retry_exponential_backoff': '', # Для установления экспоненциального времени между перезапусками
    'max_retry_delay': '', # Максимальный промежуток времени для перезапуска

    'start_date': '', # Дата начала выполнения DAG
    'end_date': '', # Дата завершения выполнения DAG

    'on_failure_callback': '', # Запустить функцию, если DAG упал
    'on_success_callback': '', # Запустить функцию, если DAG выполнился
    'on_retry_callback': '', # Запустить функцию, если DAG ушел на повторный запуск
    'on_execute_callback': '', # Запустить функцию, если DAG начал выполняться
     # Задать документацию
    'doc': '',
    'doc_md': '',
    'doc_rst': '',
    'doc_json': '',
    'doc_yaml': ''
}

schedule_interval = '0 12 * * *' # cron-выражение, также можно использовать '@daily', '@weekly', а также timedelta
dag = DAG('DAG_name', default_args=default_args, schedule_interval=schedule_interval)
```

Инициализируем таски:

```
t1 = PythonOperator(task_id='foo1', # Название таска
                    python_callable=foo1, # Название функции
                    dag=dag) # Параметры DAG

t2 = PythonOperator(task_id='foo2', # Название таска
                    python_callable=foo2, # Название функции
                    dag=dag) # Параметры DAG

t3 = PythonOperator(task_id='foo2', # Название таска
                    python_callable=foo2, # Название функции
                    dag=dag) # Параметры DAG
```

Блок логики

Задаём логику выполнения:

```
# Python-операторы
t1 >> t2 >> t3
```

```
# Методы таска
t1.set_downstream(t2)
t2.set_downstream(t3)
```

Для параллельного выполнения тасков используется структура Python операторов в вот таком виде:

```
A >> [B, C] >> D
```

или прописываются зависимости через методы таски:

```
A.set_downstream(B)
A.set_downstream(C)
B.set_downstream(D)
C.set_downstream(D)
```

Таким образом таски B и C будут выполняться параллельно, а D выполнится только после успешного выполнения B и C.

> Декораторы

Декораторы — это специальные функции в Python, которые позволяют нам «обернуть» любую функцию и добавить к ней новый функционал. Почитать про это можно, например, тут.

Чтобы было понятнее, разберём пример.

Представим, что у нас есть функция, которая печатает «Привет!»:

```
def say_hello():
    print('Hello!')
```

Мы можем написать декоратор, который будет также печатать текущее время при каждом использовании функции `say_hello()`

Нам необходимо написать функцию, которая в качестве входного параметра будет принимать функцию, а впоследствии сможет её вызывать. Назовём её `current_time`.

```
import datetime
def current_time(function):
  def wrapper():
    print(datetime.datetime.now().time())
    function()
```

Чтобы обернуть функцию с помощью `current_time`, необходимо добавить строчку `@current_time` перед объявлением функции:

```
@current_time
def say_hello():
    print('Hello!')
```

Теперь, помимо печати 'Hello', наша функция также будет печатать текущее время.

> Task Flow API

Task flow API — это дополнение, которое впервые появилось в AirFlow версии 2.0, сильно упрощающее процесс написания DAG-ов.

Основные элементы, с которыми мы теперь можем работать — уже знакомые нам **декораторы.** Теперь, когда мы задаем нашу функцию в Python, мы можем пометить её декораторами `@dag()` **и** `@task()` — таким образом мы даём интерпретатору понять, что он работает с DAG-ом или таском.

Для того, чтобы воспользоваться Task Flow API, необходимо также импортировать соответствующие функции.

```
from airflow.decorators import dag, task
```

Чтобы создать DAG, теперь достаточно создать функцию, **внутри которой находятся другие функции** — **таски,** и написать перед ней соответствующий декоратор `@dag`.

Пример может выглядеть так — обращайте внимание главным образом на структуру кода:

```
default_args = {
    'owner': 'a.batalov',
    'depends_on_past': False,
    'retries': 2,
    'retry_delay': timedelta(minutes=5),
    'start_date': datetime(2022, 3, 10),
}

# Интервал запуска DAG
schedule_interval = '0 23 * * *'

@dag(default_args=default_args, schedule_interval=schedule_interval, catchup=False)
def top_10_airflow_2():
  pass
```

В декоратор `@dag` мы также можем передавать аргументы **default_args**, которые нам уже известны по лекции. Эти аргументы задают особенности поведения наших DAG-ов. Можно задавать и другие — в том числе schedule_interval, задающий частоту и время выполнения процесса.

Чтобы создать таск, добавляем в функцию-DAG новую функцию, которую помечаем декоратором `@task()`

В декоратор `@task()` также можно передавать параметры. Например, **retries** указывает количество повторов DAG-а, если он почему-то не сработал**,** а **retry_delay** — временной промежуток между этими повторами:

```
@dag(default_args=default_args, catchup=False)
def top_10_airflow_2():
    @task(retries=3)
    def get_data():
        top_doms = requests.get(TOP_1M_DOMAINS, stream=True)
        zipfile = ZipFile(BytesIO(top_doms.content))
        top_data = zipfile.read(TOP_1M_DOMAINS_FILE).decode('utf-8')
        return top_data

    @task(retries=4, retry_delay=timedelta(10))
    def get_table_ru(top_data):
        top_data_df = pd.read_csv(StringIO(top_data), names=['rank', 'domain'])
        top_data_ru = top_data_df[top_data_df['domain'].str.endswith('.ru')]
        return top_data_ru.to_csv(index=False)
```

ВАЖНО! Убедитесь, что не запускаете созданный вами DAG внутри самого DAG-а. Вещь, кажущаяся очевидной, но крайне коварная — как и все проблемы отступов. Студенты на ней периодически спотыкаются.

Вот этот DAG не заработает:

```
@dag(default_args=default_args, catchup=False)
def my_dag():
    (...)
    my_dag = my_dag()
```

А вот этот заработает:

```
@dag(default_args=default_args, catchup=False)
def my_dag():
    (...)

my_dag = my_dag()
```

> Как добавить свой DAG

1. Клонируете репозиторий.
2. В локальной копии внутри папки **dags** создаёте свою папку — она должна совпадать по названию с вашим именем пользователя, которое через @ в профиле GitLab.
3. Создаёте там DAG — он должен быть в файле с форматом **.py.**
4. Запушиваете результат (не забудьте подтянуть все свежие изменения, чтобы не удалить папки своих коллег!).
5. Включаете DAG, когда он появится в Airflow.

Если DAG долго не появляется, то с высокой вероятностью случилось одно из следующего набора событий:

1. Ваш DAG не в той папке/не в той ветке (должен быть в DAG в ветке master).
2. Неправильно назвали папку (в результате система не может найти вашего имени).
3. У вашего DAG-а не уникальное название (проверьте, что оно не дублирует названия ваших коллег).
4. Код по той или иной причине нерабочий.

> Документация

Рекомендуем ознакомиться с информацией по данным ссылкам — она поможет вам при решении задачи этого урока!

Основная документация Airflow

TaskFlow

Полный список переменных контекста

> Код из урока

Его также можно найти в репозитории, связанном с нашим Airflow, но для удобства прикрепляем его и тут:

```
# coding=utf-8

from datetime import datetime, timedelta
```

```
import pandas as pd
from io import StringIO
import requests

from airflow.decorators import dag, task
from airflow.operators.python import get_current_context

# Функция для CH

def ch_get_df(query='Select 1', host='https://clickhouse.lab.karpov.courses', user='student', password='dpo_python_2020'):
    r = requests.post(host, data=query.encode("utf-8"), auth=(user, password), verify=False)
    result = pd.read_csv(StringIO(r.text), sep='\t')
    return result

query = """SELECT
toDate(time) as event_date,
country,
source,
count() as likes
FROM
simulator.feed_actions
where
toDate(time) = '2022-01-26'
and action = 'like'
group by
event_date,
country,
source
format TSVWithNames"""

# Дефолтные параметры, которые прокидываются в таски

default_args = {
'owner': 'a.batalov',
'depends_on_past': False,
'retries': 2,
'retry_delay': timedelta(minutes=5),
'start_date': datetime(2022, 3, 10),
}

# Интервал запуска DAG

schedule_interval = '0 23 * * *'

@dag(default_args=default_args, schedule_interval=schedule_interval, catchup=False)
def dag_sim_example():

```
```
@task()
def extract():
    query = """SELECT
                    toDate(time) as event_date,
                    country,
                    source,
                    count() as likes
                FROM
                    simulator.feed_actions
                where
                    toDate(time) = '2022-01-26'
                    and action = 'like'
                group by
                    event_date,
                    country,
                    source
                format TSVWithNames"""
    df_cube = ch_get_df(query=query)
    return df_cube

@task
def transfrom_source(df_cube):
    df_cube_source = df_cube[['event_date', 'source', 'likes']]\\
```

```
        .groupby(['event_date', 'source'])\\
        .sum()\\
        .reset_index()
    return df_cube_source

@task
def transfrom_countries(df_cube):
    df_cube_country = df_cube[['event_date', 'country', 'likes']]\\
        .groupby(['event_date', 'country'])\\
        .sum()\\
        .reset_index()
    return df_cube_country

@task
def load(df_cube_source, df_cube_country):
    context = get_current_context()
    ds = context['ds']
    print(f'Likes per source for {ds}')
    print(df_cube_source.to_csv(index=False, sep='\\t'))
    print(f'Likes per country for {ds}')
    print(df_cube_country.to_csv(index=False, sep='\\t'))

df_cube = extract()
df_cube_source = transfrom_source(df_cube)
df_cube_country = transfrom_countries(df_cube)
load(df_cube_source, df_cube_country)
```

```
dag_sim_example = dag_sim_example()
```