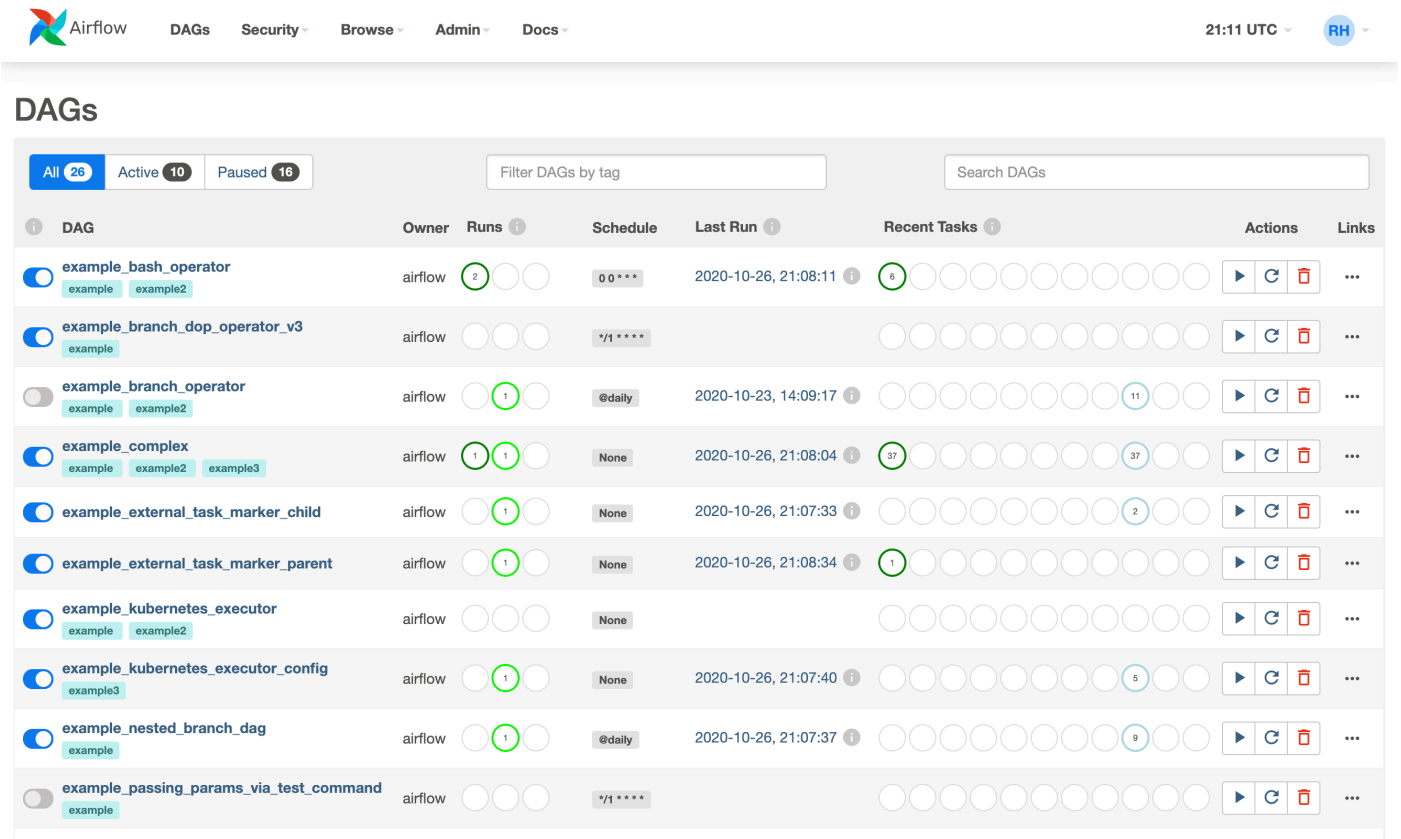
> ETL-ПАЙПЛАЙН

> 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:

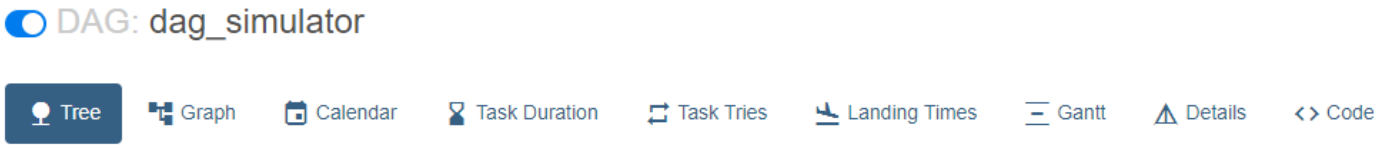


Здесь мы видим большую подпись **DAGs**. **DAG** — это основная единица работы с Airflow, мы обсудим его подробнее в другой части конспекта. Для начала можем считать, что это некоторая глобальная задача, решаемая путем последовательного выполнения более мелких, редуцированных задач.

Интерфейс:

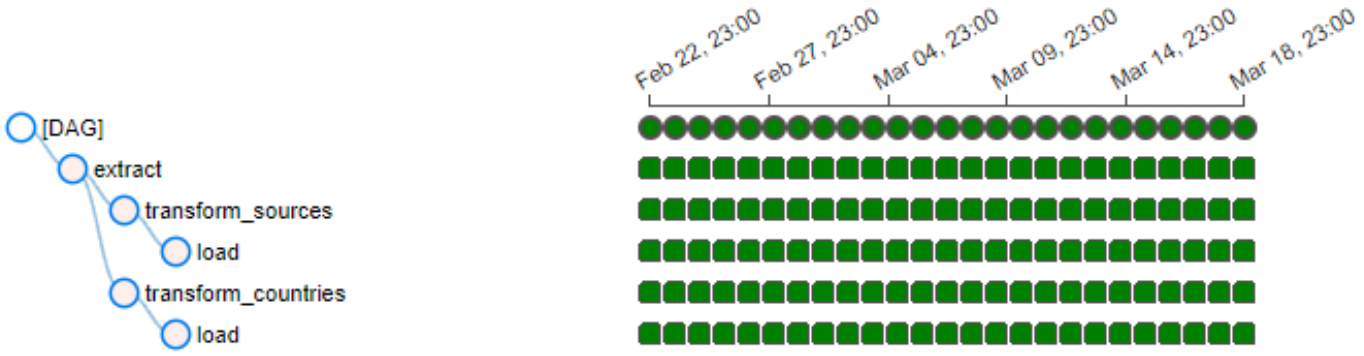
На главной страничке у нас перечислены все доступные DAGи, вкладки **All**, **Active** и **Paused** позволяют фильтровать DAGи в соответствии с состоянием их выполнения. У каждого DAGа стоит переключатель, отвечающий за то, активен ли DAG или нет, затем идет название, владелец, информация о запусках 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:

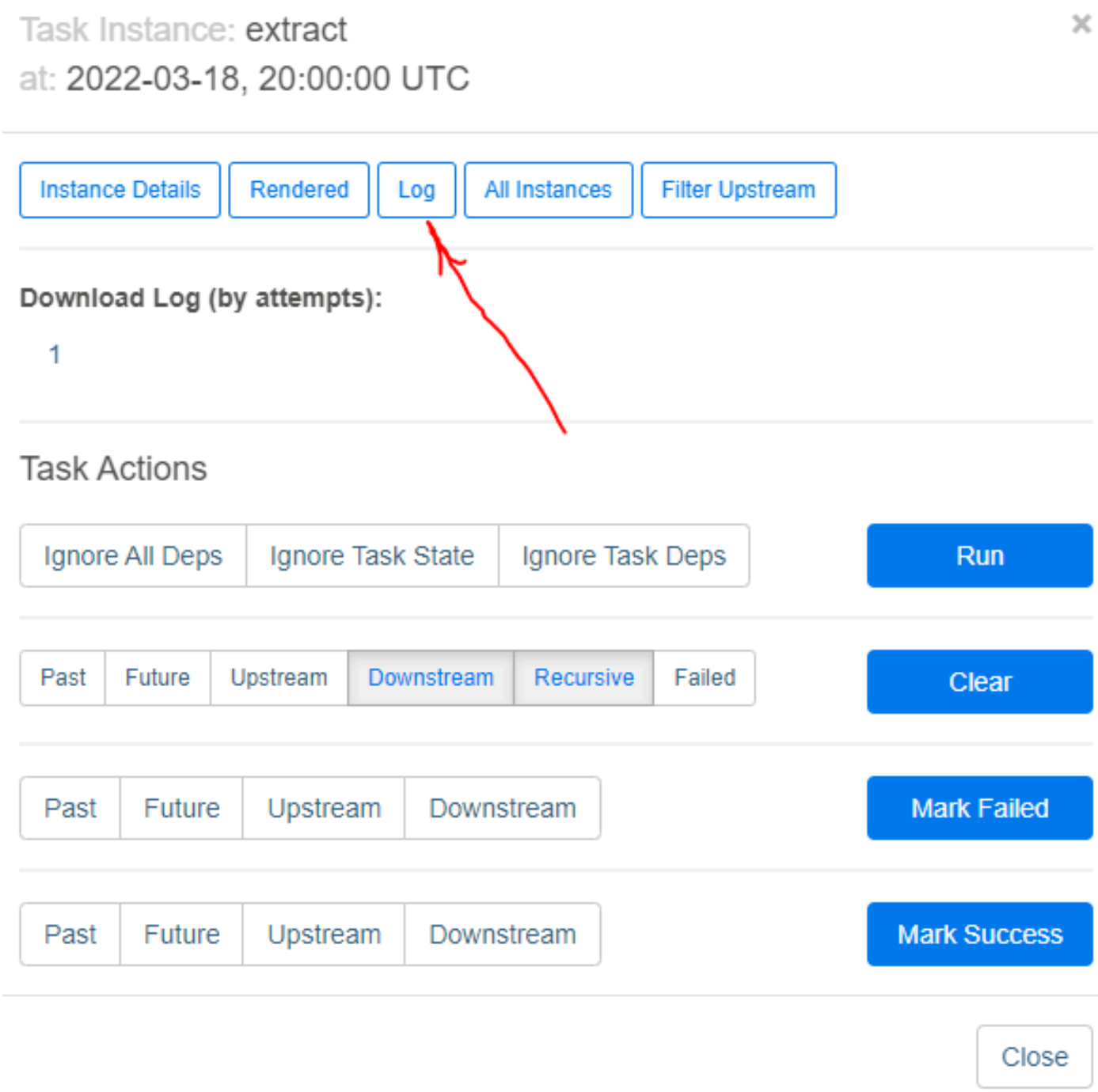


Здесь можно визуализировать DAG, смотреть на его расписание, продолжительность выполнения «маленьких» задач, количество запусков DAGа, а также другие интересные вещи.

«Древесное» отображение DAGa удобно тем, что можно отслеживать историю выполнения задач в DAGe:

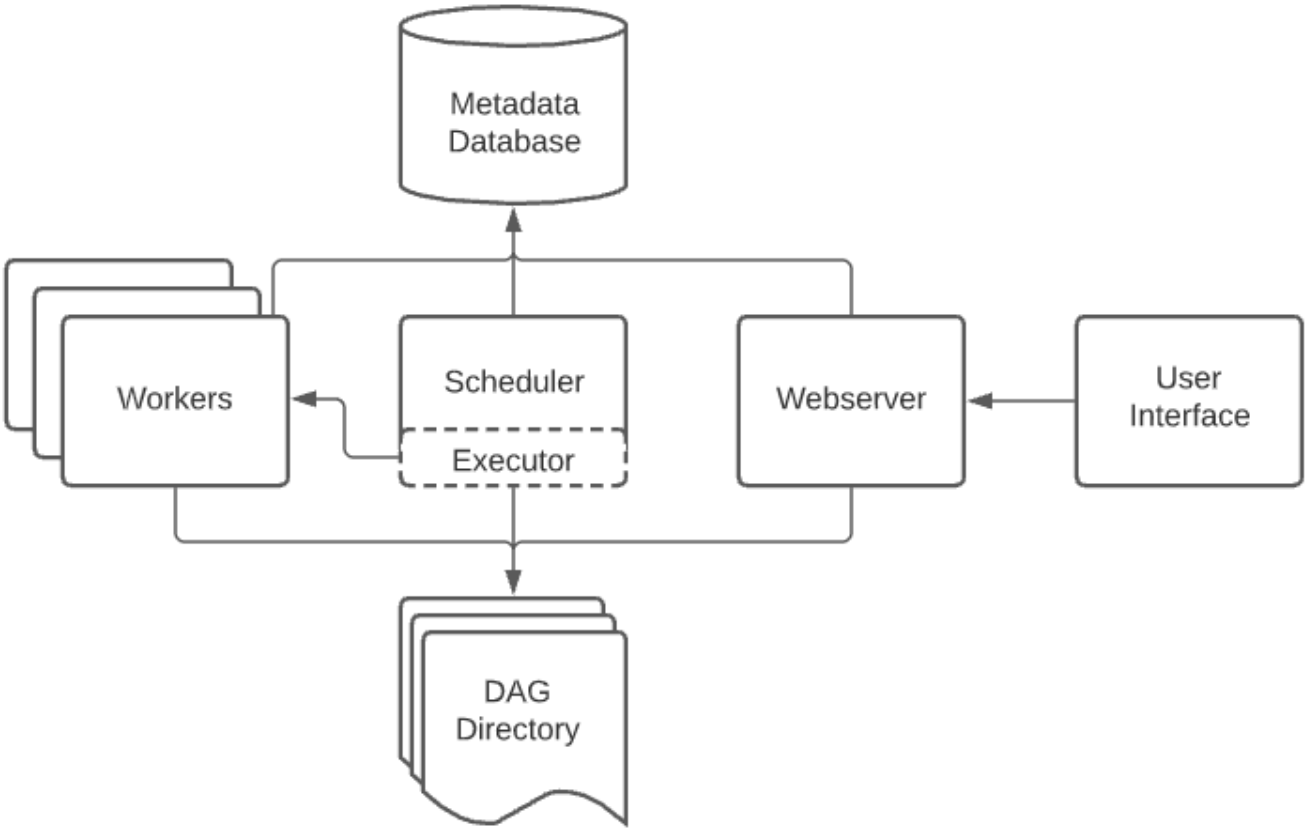


Если что-то сломалось (задача выделена не зелёным цветом, а каким-то другим), можно нажать на неё и в выбранном меню открыть логи. Это позволит вам отследить, из-за чего произошла поломка.



> 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

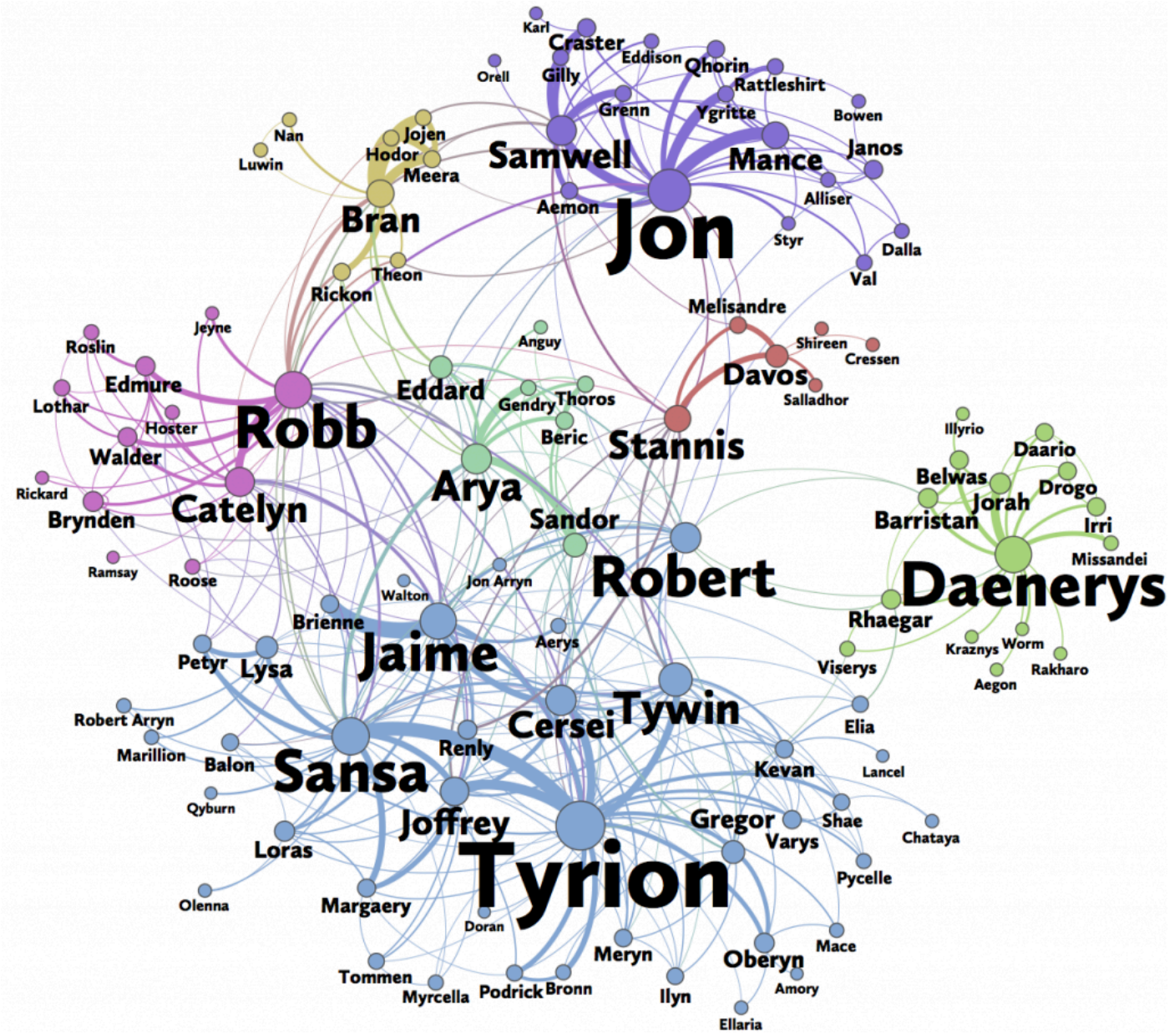
> Что такое DAG?

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

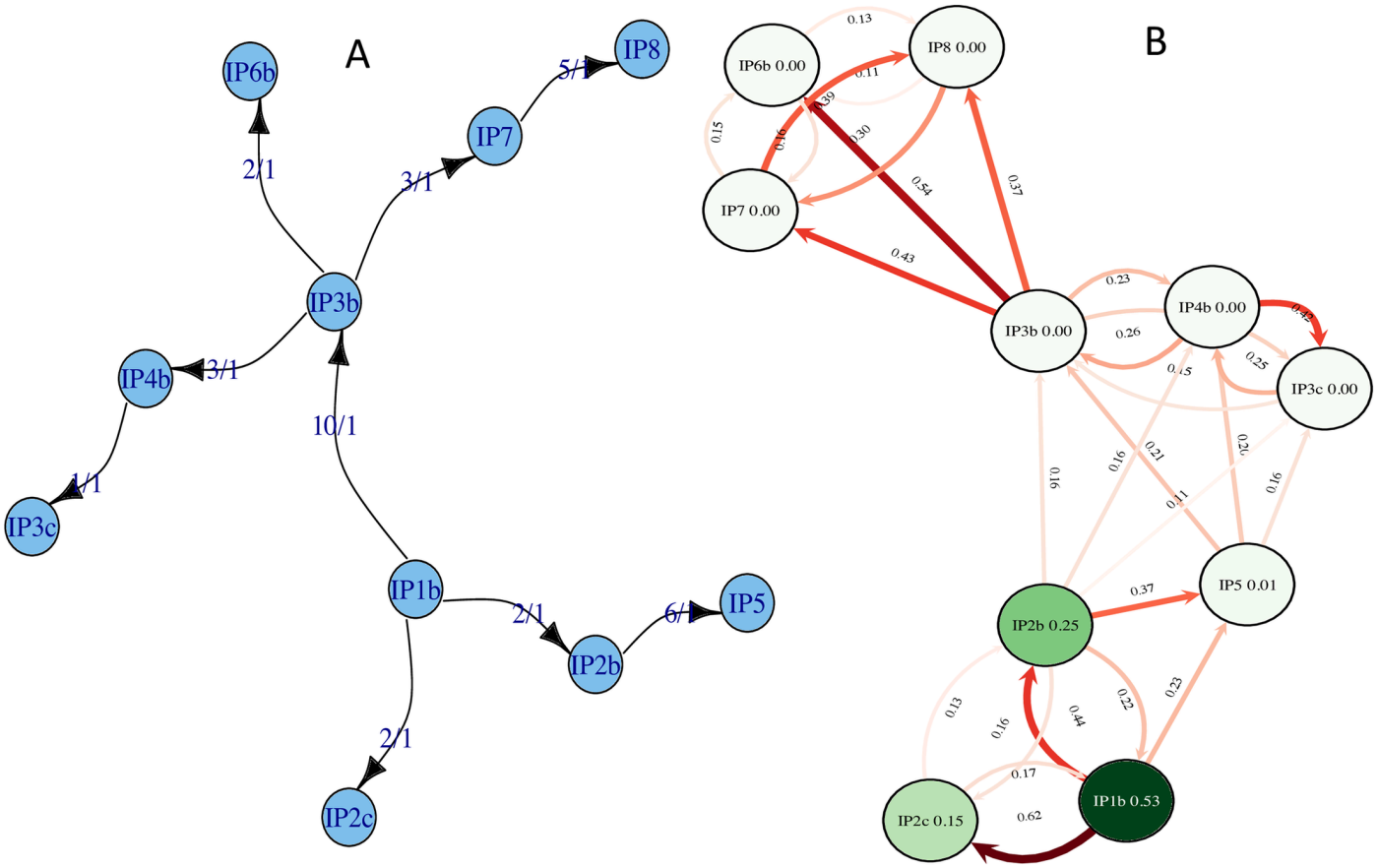
Грубо говоря, граф — это математическая абстракция, отражающая множество связанных между собой элементов. Например, ниже приведён граф связей

между героями Игры Престолов:



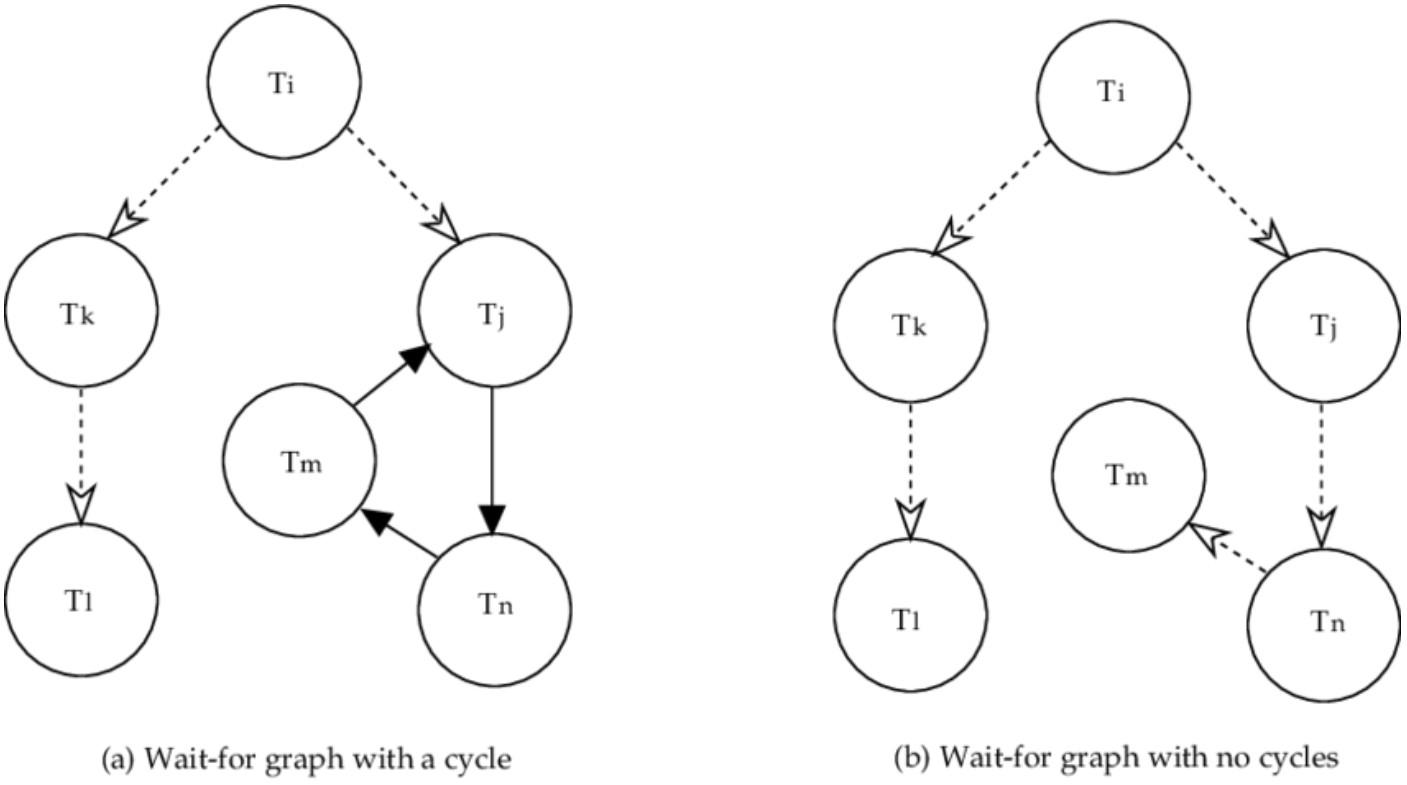
Направленность

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

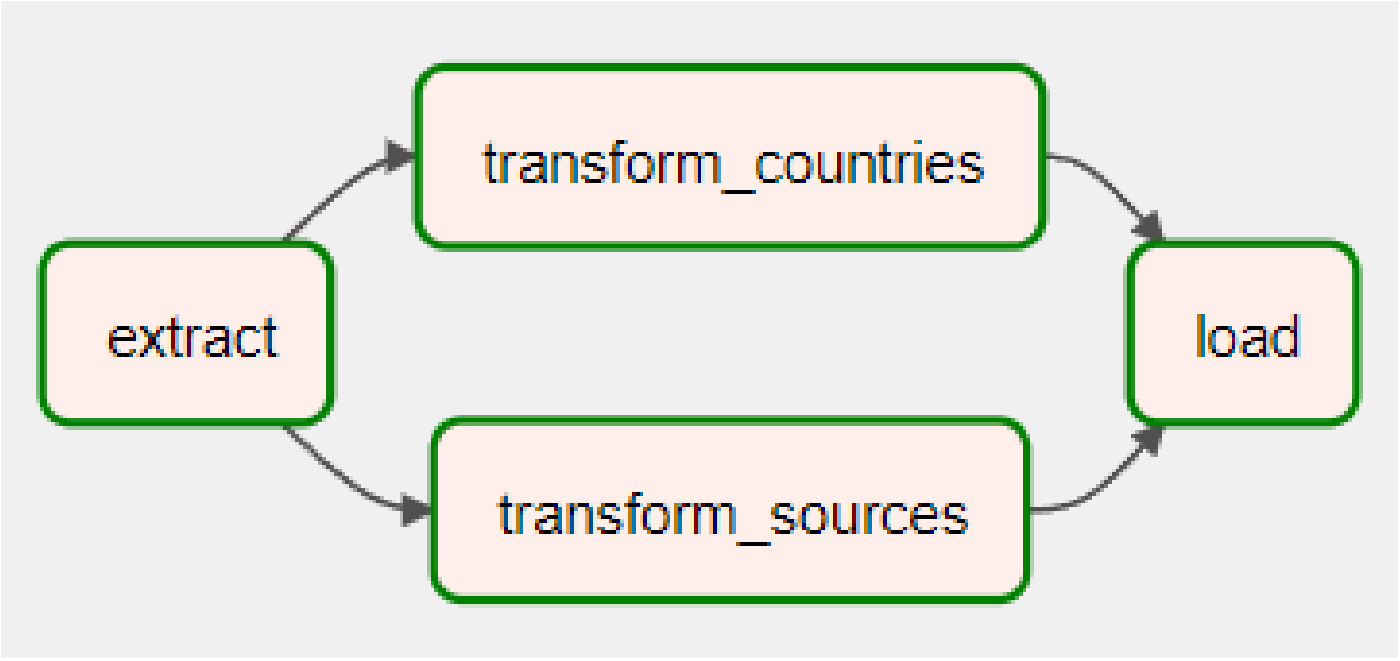
В некоторых направленных графах развитие процесса может развиваться в строго определённом направлении — вернуться обратно к тому же элементу невозможно, если ты уже из него вышел. Такие графы называют **ациклическими**, потому что в них **нет циклов**. Пример можно видеть ниже:



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»:

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

Блок кода

Здесь вы прописываете ваши функции:

```
1 def foo1():
2 def foo2():
3 def foo3():
```

Блок инициализации

Задаем параметры в DAG:

```
1 default_args = {
2     'owner': 'your_name', # Владелец операции
3     'depends_on_past': False, # Зависимость от прошлых запусков
4
```

```
    '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
```

```
11                      dag=dag) # Параметры DAG
```

Блок логики

Задаём логику выполнения:

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

```
1 # Методы таска
2 t1.set_downstream(t2)
3 t2.set_downstream(t3)
```

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

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

или прописываются зависимости через методы таски:

```
1 A.set_downstream(B)
2 A.set_downstream(C)
3 B.set_downstream(D)
4 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:

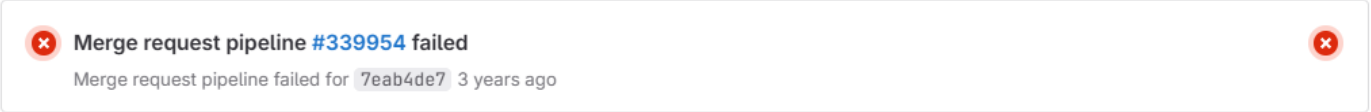
```
@dag(default_args=default_args, catchup=False)
def my_dag():
    (...)

my_dag = my_dag()
```

> Как добавить свой DAG

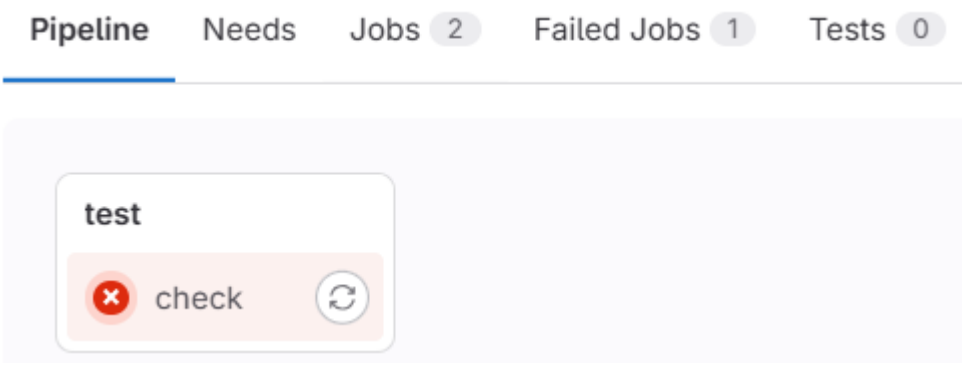
1. Клонируете репозиторий (через терминал).
2. В локальной копии внутри папки **dags** создаёте свою папку — она должна совпадать по названию с вашим именем пользователя, которое через @ в профиле GitLab.
3. Создаёте там DAG — он должен быть в файле с форматом **.py.**
4. Создаёте ветку и переходите в неё

5. Находясь в папке **airflow** (именно там!), добавляете вашу папку как `git add dags/название_папки`. Если вы будете выполнять команды `git` в других папках внутри репозитория airflow, то результаты могут быть непредсказуемы.
6. Коммитите изменения и пушите
7. Создаёте merge request, сами нажимаете на кнопку Merge
8. Ждёте 30-60 минут, когда в интерфейсе Airflow появляется ваш DAG — включаете его

Чтобы студенты не создавали папки в неподходящих местах и не меняли скрипты других студентов, у нас в GitLab стоит специальная автоматическая проверка merge request-а. Из-за неё может случиться так, что пункт 7 не будет получаться — в таких случаях в merge request появляется подобное сообщение:



Нажмите на номер пайплайна (на скриншоте это #339954), внизу открывшейся странички нажмите на кнопку с надписью check — вы получите полный лог ошибки, из-за которой не проходит merge request.



Самые частые проблемы, из-за которых такое случается:

1. Ваша папка не находится внутри папки dags, а лежит снаружи неё либо внутри чужой папки.
2. Название папки не совпадает с именем юзера — например, в GitLab ваш юзернейм **v-pupkin-44**, а вы назвали свою папку **v-pupkin.**
3. В ваш коммит по той или иной причине попали чужие файлы. В конспекте урока “Знакомство с рабочим окружением” описывается, как можно отменить коммит, а также отменить `git add` на отдельный файл. Однако лучше избегать этой проблемы в зародыше: **внимательно смотрите, какие файлы вы добавляете через git add и не добавляйте ничего, что вам**

не принадлежит. Если не уверены — проверяйте себя периодически через `git status`.

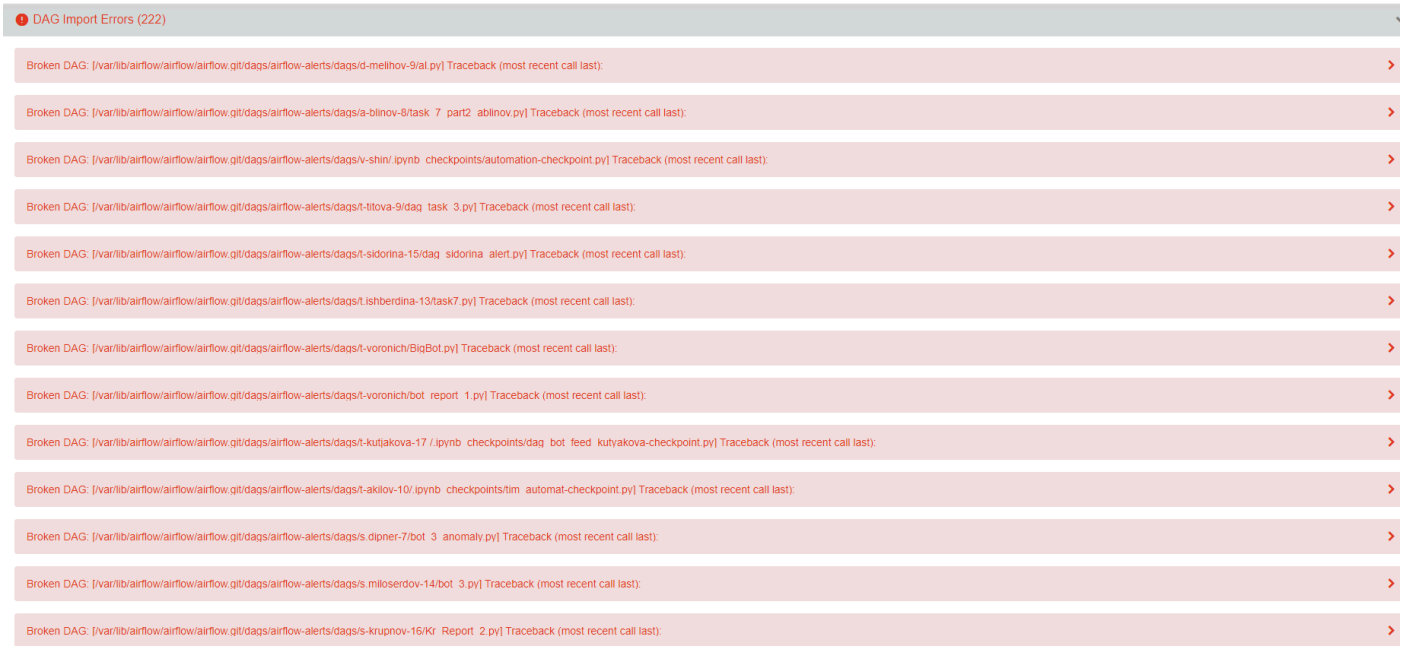
4. В коммит попала папка `.ipynb_checkpoints`. Избавляться от этого так же, как в 3 пункте. В целом внутри вашей папки никаких других папок быть не должно.
5. В названии ваших файлов есть пробелы (скажем, ваш файл называется `my work.py`). Исправляется путём избавления от пробелов в названии (например, `my_work.py`).

> Когда DAG долго не появляется

Как мы уже упоминали, обычно DAG появляется в Airflow в течение 30-60 минут. Однако бывает так, что DAG не появляется там и после этого; в таком случае стоит проверить ваш DAG на работоспособность.

Первое место, куда вы должны заглянуть — это раздел DAG Import Errors:

Если его развернуть, то вы увидите список всех DAG-ов, которые не прошли в Airflow из-за ошибки в коде:



Попробуйте найти там свой DAG — используйте для этого поиск по странице (`Ctrl + F` на Windows, `Command + F` на MacOS). Искать лучше всего по названию вашей папки. Если нашли себя — разверните раздел с собой и прочитайте текст

ошибки. После этого остаётся только успешно исправить её, повторить весь процесс вплоть до мерджа и снова подождать 🙂

Однако иногда ошибки бывают такие, что Airflow просто не может распознать ваш файл как валидный DAG. В таком случае в DAG Import Errors ваш DAG не появится, и нужно искать ошибку самостоятельно.

Наиболее распространённые проблемы такого типа:

1. Ваш файл имеет формат **.ipynb**, а не **.py**
2. Вы подошли к вопросу создания файла типа **.py** путём переименования файла (то есть у вас файл `my_work.ipynb`, и вы просто вручную поменяли его название на `my_work.py`). **Так делать нельзя.** Файл типа **.py** можно создать в интерфейсе JupyterHub (упоминалось в конспекте урока “Знакомство с рабочим окружением”).
3. Вы забыли нажать на кнопку Merge в вашем merge request. Airflow не увидит ваш файл, если вы забудете его вмерджить 🙂
4. Название вашего DAG повторяет уже существующее. Например, вы назвали ваш DAG `dag_sim_example` — так назывался DAG преподавателя урока, и второго такого названия быть не может. Используйте уникальные названия.
5. Вы забыли указать необходимые импорты в скрипте — в первую очередь `from airflow.decorators import dag, task`.
6. У вас нарушена структура DAG-а — она должна быть примерно такой:

```
@dag(...)
def my_great_dag():

    @task
      def my_great_task():
        ...
        ...

      result = my_great_task()
    ...

my_great_dag = my_great_dag()
```

1. Названия тасок не согласованы внутри скрипта — например:

```
@task
  def my_great_task():
    ...
    ...

    result = my_best_task()
```

Таск называется `my_great_task`, однако потом вместо него фигурирует название `my_best_task`. Это проблема не уникальна для Airflow, но встречается в этом уроке довольно часто.

1. Один таск используется внутри другого таска — **так делать нельзя**.

```
@task
  def my_great_task():
    ...
    ...

@task
  def my_greater_task():
    my_res = my_great_task()
    ...
```

Если вы хотите создать какую-то вспомогательную функцию, чтобы использовать её внутри таска — сделайте её обычной Python-функцией и создайте её в скрипте перед DAG-ом.

> Как использовать pandahouse для решения задания

В решении этого задания вам очень поможет пакет pandahouse. Кратко пробежимся по тому, как вам пригодятся три основные функции этого пакета.

pandahouse.read_clickhouse()

Эта функция позволяет вам выполнять любой запрос и выгружать его результат в виде pandas DataFrame. Вы уже могли использовать её при решении задач прошлых уроков, но напомним, как её использовать:

```python
import pandahouse as ph

query = '''
здесь мы пишем запрос как в Clickhouse
'''

#здесь мы задаём параметры подключения
#в качестве database указывайте ту схему, с которой вы работаете
connection = {'host': 'https://clickhouse.lab.karpov.courses',
                      'database':'simulator_XXXXXXXX',
                      'user':'student',
                      'password':'dpo_python_2020'
                     }

#результат запроса сохранится как датафрейм
df = ph.read_clickhouse(query, connection=connection)
```

pandahouse.execute()

Результат `pandahouse.read_clickhouse()` обязательно должен иметь на выходе табличку — иначе запрос выполнится, но функция выдаст ошибку. Это не супер критично, но это раздражает — и запрос для создания таблички в базе данных как раз является таковым.

Здесь нам и пригодится `pandahouse.execute()`, который может выполнить запрос создания новой таблички без лишних проблем:

```python
#для создания студенческих табличек у нас есть отдельная база test
#параметры подключения у неё другие
connection_test = {'host': 'https://clickhouse.lab.karpov.courses',
                      'database':'test',
                      'user':'student-rw',
                      'password':'656e2b0c9c'
                     }

#запрос создания таблички имеет примерно следующую структуру
query_test = '''CREATE TABLE IF NOT EXISTS test.название_таблицы
                    (название_колонки ТипКолонки,
                    ...
                    )
                    ENGINE = MergeTree()
                    ORDER BY название_колонки_по_которой_сортируются_данные
'''

```

```
18 ph.execute(query_test, connection=connection_test)
```

pandahouse.to_clickhouse()

Загружать обработанные данные в созданную вами табличку можно так:

```
1 #функция работает путём конкатенации
2 #т.е. она просто присоединит новые данные к низу таблички
3
4 ph.to_clickhouse(df=df, table="название_таблицы",
5             index=False, connection=connection_test)
```

Будьте внимательны: названия колонок, их порядок и тип должны совпадать — какую вы создали таблицу в Clickhouse, такой же должен быть pandas DataFrame.

> 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:

```
1 # coding=utf-8
2
3 from datetime import datetime, timedelta
4 import pandas as pd
5 from io import StringIO
6 import requests
7
8 from airflow.decorators import dag, task
```

```python
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()
```