# Предыстория 🚀

**Запуски ракет**

Ракеты - настоящее творение инженерной мысли человечества. Каждый раз смотря в небо, я понимаю, что сейчас где-то там, в миллиардах километрах от нас летят Вояджеры, где-то там на МКС люди выходят на орбиту. А где-то новейший телескоп Джеймс Уэбб делает удивительные фотографии. Но ничего из этого не существовало бы без ракет.

# Задача 💻

- Представим, что вы захотели создать небольшой проект по автоматизации выгрузки данных из интернета обо всех запусках ракет, чтобы иметь своего рода личное представление о последних событиях, касающихся этих запусков. Допустим, сначала мы хотим автоматизировать сбор изображений этих ракет.

# Изучение данных 🔍

Для работы с данными мы будем использовать открытый API для всех на планете, в котором содержится информация о предыдущих и будущих запусках ракет из различных источников, собранное воедино от Launch Library 2. Ограничимся только предстоящими запусками ракет. Ссылка для API:

```
curl -L "https://ll.thespacedevs.com/2.0.0/launch/upcoming"
```

Или можно запустить небольшой код и посмотреть, какой же самый ближайший запуск ракеты в мире состоится:

```
import os
import json

info=os.popen('curl -L "https://ll.thespacedevs.com/2.0.0/launch/upcoming"').read()
info=json.loads(info)
launch_list=info['results']
print(len(launch_list)) #10 запусков зается изначально в запросе
print(json.dumps(launch_list[0], indent=4))
```

На что мы получаем вот такой json файл с информацией о ближайшем предстоящем запуске (по умолчанию дается 10 ближайших запусков, но можно запросить и все):


```
{
    "id": "1c099ba2-9b8f-4364-8ba0-0c26f7993dd5",
    "url": "https://ll.thespacedevs.com/2.0.0/launch/1c099ba2-9b8f-4364-8ba0-
0c26f7993dd5/",
    "launch_library_id": null,
    "slug": "falcon-9-block-5-starlink-group-6-18",
    "name": "Falcon 9 Block 5 | Starlink Group 6-18",
    "status": {
        "id": 2,
        "name": "TBD"
    },
    "net": "2023-09-24T00:06:00Z",
    "window_end": "2023-09-24T04:37:00Z",
    "window_start": "2023-09-24T00:06:00Z",
    "inhold": false,
    "tbdtime": false,
    "tbddate": false,
    "probability": null,
    "holdreason": "",
    "failreason": "",
    "hashtag": null,
    "launch_service_provider": {
        "id": 121,
        "url": "https://ll.thespacedevs.com/2.0.0/agencies/121/",
        "name": "SpaceX",
        "type": "Commercial"
```

```
    },
    "rocket": {
        "id": 7870,
        "configuration": {
            "id": 164,
            "launch_library_id": 188,
            "url": "https://ll.thespacedevs.com/2.0.0/config/launcher/164/",
            "name": "Falcon 9",
            "family": "Falcon",
            "full_name": "Falcon 9 Block 5",
            "variant": "Block 5"
        }
    },
    "mission": {
        "id": 6378,
        "launch_library_id": null,
        "name": "Starlink Group 6-18",
        "description": "A batch of satellites for the Starlink mega-constellation -
SpaceX's project for space-based Internet communication system.",
        "launch_designator": null,
        "type": "Communications",
        "orbit": {
            "id": 8,
            "name": "Low Earth Orbit",
            "abbrev": "LEO"
        }
    },
    "pad": {
        "id": 80,
        "url": "https://ll.thespacedevs.com/2.0.0/pad/80/",
        "agency_id": 121,
        "name": "Space Launch Complex 40",
        "info_url": null,
        "wiki_url":
"https://en.wikipedia.org/wiki/Cape_Canaveral_Air_Force_Station_Space_Launch_Complex_40",
        "map_url": "https://www.google.com/maps?q=28.56194122,-80.57735736",
        "latitude": "28.56194122",
        "longitude": "-80.57735736",
        "location": {
            "id": 12,
            "url": "https://ll.thespacedevs.com/2.0.0/location/12/",
            "name": "Cape Canaveral, FL, USA",
            "country_code": "USA",
```

```
            "map_image": "https://spacelaunchnow-prod-
east.nyc3.digitaloceanspaces.com/media/launch_images/location_12_20200803142519.jpg",
            "total_launch_count": 908,
            "total_landing_count": 37
        },
        "map_image": "https://spacelaunchnow-prod-
east.nyc3.digitaloceanspaces.com/media/launch_images/pad_80_20200803143323.jpg",
        "total_launch_count": 200
    },
    "webcast_live": false,
    "image": "https://spacelaunchnow-prod-
east.nyc3.digitaloceanspaces.com/media/launch_images/falcon2520925_image_20230522091403.p
ng",
    "infographic": null,
    "program": []
}
```


⚠ **Внимание!**

- Данных API очень и очень серьезен. Это официальные данные, которые декларируются в обязательном порядке.
- Вы можете заметить, что об одном запуске уже известно достаточно много информации, однако большая часть информации, к примеру, параметры и сводка о ракете, организаторе запуска, и так далее, можно найти дополнительную информацию. Так, можно получить дополнительные данные о ракете, если сделать еще один запрос.

Можно получить еще одну ссылку с данными о ракете, если добавить строку:
`rocket_info=launch_list[0]["rocket"]["configuration"]["url"]`

✕ **Защита от DDOS**

Обращаю внимание, что не стоит слишком часто обращаться к этому API, потому что тут, ОКАЗЫВАЕТСЯ, стоит защита от ddos атак (что можно увидеть нечасто). Мне заблочили доступ на 30 минут из-за частых отправок запросов на получение данных (скорее всего по ip, поэтому слишком упорным скажу - Обойти возможно 😆). Узнал также обычные значения - 15 запросов в час.

Крч можно получить дополнительную сводку о ракете, к примеру вот что я успел поймать пока не заблочили:

```
"variant": "Block 5",
            "alias": "",
            "min_stage": 1,
            "max_stage": 2,
            "length": 70.0,
            "diameter": 3.65,
            "maiden_flight": "2018-05-11",
            "launch_mass": 549,
            "leo_capacity": 22800,
            "gto_capacity": 8300,
            "to_thrust": 7607,
            "apogee": 200,
            "vehicle_range": null,
            "image_url": "[https://spacelaunchnow-prod-
east.nyc3.digitaloceanspaces.com/media/launcher_images/falcon_9_image_20230807133459.jpeg
](https://spacelaunchnow-prod-
east.nyc3.digitaloceanspaces.com/media/launcher_images/falcon_9_image_20230807133459.jpeg
)",
            "info_url": "[https://www.spacex.com/vehicles/falcon-9/]
(https://www.spacex.com/vehicles/falcon-9/)",
            "wiki_url": "[https://en.wikipedia.org/wiki/Falcon_9]
(https://en.wikipedia.org/wiki/Falcon_9)",
            "total_launch_count": 201,
            "consecutive_successful_launches": 201,
            "successful_launches": 201,
            "failed_launches": 0,
            "pending_launches": 123
```


## Итог

Как видите, данные представлены в формате JSON и предоставляют информацию о запуске ракеты, а для каждого запуска есть информация о конкретной ракете, такая как идентификатор, имя и URL-адрес изображения. Это именно то, что нам нужно. Давайте нарисуем карту, и сопоставим их с задачами для Airflow

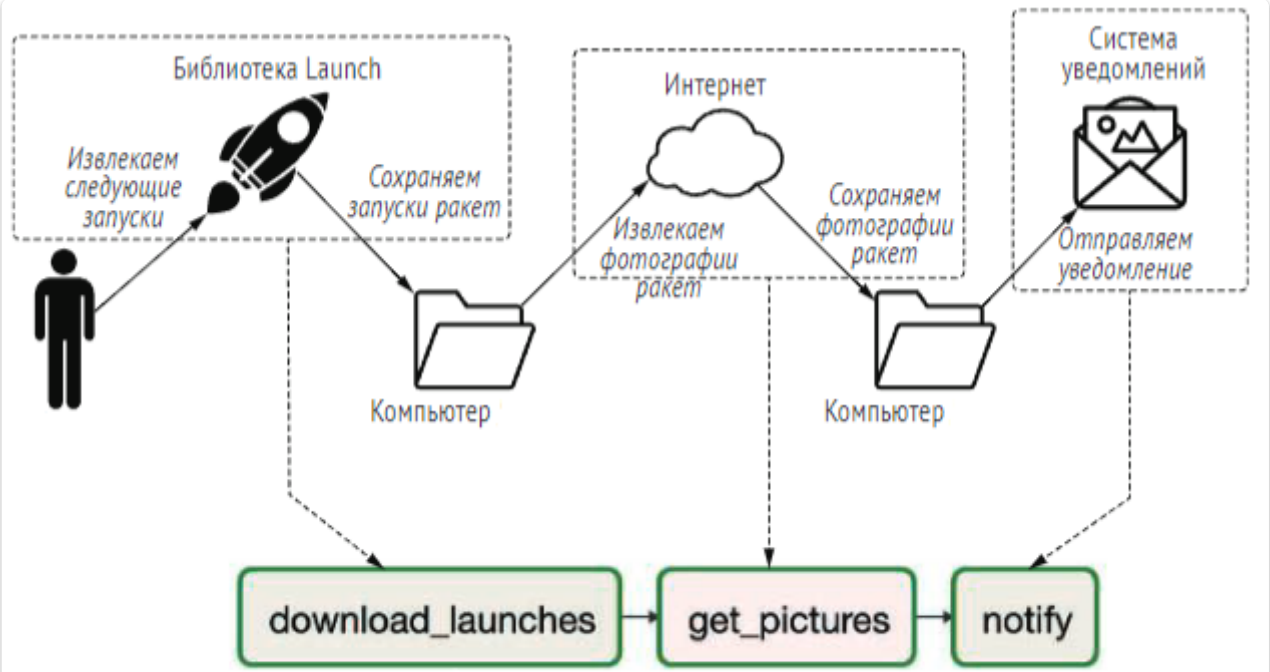



Рис. 2.3 Ментальная модель , сопоставленная с задачами в Airflow

## Пишем первый ОАГ 🧐

Начнем писать основную программу, заодно разберем каждую строку.

- Первым делом добавим библиотеку для работы с json файлами (словарями)

```
import json
```

- Дальше давайте вспомним, что в разных операционных системах существуют свои правила построения путей к файлам. Например, в Linux для путей используются прямые слэши ( `/` ), а в Windows — обратные слэши ( `\` ). Для нормальной работы в любой ОС, можно использовать модуль Pathlib, который обеспечит одинаковую работу на всех ОС

```
import pathlib
```

Остальные модули разберем в процессе.

Посмотрите теперь на код целиком, и после мы обсудим все необходимые моменты.

```
import json
import pathlib
import airflow
import requests
import requests.exceptions as requests_exceptions
from airflow import DAG
```

```
from airflow.operators.bash_operator import BashOperator #В новых версиях используется
метод .bash
from airflow.operators.python_operator import PythonOperator #В новых версиях
используется метод .python

dag = DAG(
    dag_id="download_rocket_launches",
    start_date=airflow.utils.dates.days_ago(14),
    schedule_interval=None,)

download_launches = BashOperator(
    task_id="download_launches",
    # bash_command='(if not exist tmp (mkdir tmp)) & curl -o tmp/launches.json -L
"https://ll.thespacedevs.com/2.0.0/launch/upcoming"', #Для отладки на Windows
    bash_command="curl -o tmp/launches.json -L
'https://ll.thespacedevs.com/2.0.0/launch/upcoming'", # Основная команда на Linux
    dag=dag, )

def _get_pictures(): # Убеждаемся, что каталог существует
    pathlib.Path("tmp/images").mkdir(parents=True, exist_ok=True)

    # Скачиваем все изображения в launches.json
    with open("tmp/launches.json") as f:
        launches = json.load(f)
        image_urls = [launch["image"] for launch in launches["results"]]
        for image_url in image_urls:
            try:
                response = requests.get(image_url)
                image_filename = image_url.split("/")[-1]
                target_file = f"tmp/images/{image_filename}"
                with open(target_file, "wb") as f:
                    f.write(response.content)
                print(f"Downloaded {image_url} to {target_file}")
            except requests_exceptions.MissingSchema:
                print(f"{image_url} appears to be an invalid URL.")

            except requests_exceptions.ConnectionError:
                print(f"Could not connect to {image_url}.")

get_pictures = PythonOperator(
    task_id="get_pictures",
    python_callable=_get_pictures,
    dag=dag, )

notify = BashOperator(
    task_id="notify",
    bash_command='echo "There are now $(ls tmp/images/ | wc -l) images."',
    dag=dag, )
```

```
download_launches >> get_pictures >> notify
```

## Разбор кода

Разберем этот рабочий процесс. ОАГ – это отправная точка любого рабочего процесса. Все задачи в рамках рабочего процесса ссылаются на этот объект ОАГ, чтобы Airflow знал, какие задачи какому ОАГ принадлежат.

```
dag = DAG( dag_id="download_rocket_launches",
start_date=airflow.utils.dates.days_ago(14), schedule_interval=None,)
```

**Что есть что**

- Класс DAG принимает два обязательных аргумента: dag_id, start_date.
- dag_id - это имя ОАГ, которое будет отображаться в интерфейсе Airflow.
- start_date - Дата и время, когда рабочий процесс должен быть запущен в первый раз
- schedule_interval=None - Означает, что ОАГ не будет запускаться автоматически. Позже мы займемся планированием, пока оставим запуск на ручное управление.

Затем сценарий рабочего процесса Airflow состоит обычно из одного или нескольких операторов, которые выполняют различную работу. Так, в коде следующим мы запускаем BashOperator для запуска команды Bash.

```
download_launches = BashOperator(
    task_id="download_launches",
    # bash_command='(if not exist tmp (mkdir tmp)) & curl -o tmp/launches.json -L
"https://ll.thespacedevs.com/2.0.0/launch/upcoming"', #Для отладки на Windows
    bash_command="curl -o tmp/launches.json -L
'https://ll.thespacedevs.com/2.0.0/launch/upcoming'", # Основная команда на Linux
    dag=dag, )
```

**А здесь что написано?**

task_id - Имя задачи
bash_command - Команда Bash для выполнения
dag=dag - Ссылка на переменную DAG
`curl -o /tmp/launches.json -L 'https://ll.thespacedevs.com/2.0.0/launch/upcoming'`:
curl -o - Это опция, которая говорит curl сохранить тело ответа в локальном файле. Опция -L позволяет переходить по любому перенаправлению, пока не достигнет конечного пункта назначения. В нашем случае не играет никакой роли, но оставим.

if not exist добавил для автоматического добавления папки tmp если ее изначально не было

## Определение порядка выполнения задачи

`download_launches >> get_pictures >> notify`

В Airflow можно использовать *бинарный оператор сдвига вправо* для определения зависимостей между задачами. Так можно гарантировать, что одна задача не запустится, пока не выполнится ей предыдущая.

> *В Python для сдвигов битов используется оператор* `>>`

## Задачи и операторы

Вы можете меня спросить: в чем разница между задачами и операторами?

⚠ В Airflow операторы существуют только для одной единицы работы. К примеру, существуют универсальные операторы, например:

- BashOperator используется для запуска сценария Bash
- EmailOperator используется для отправки на электронную почту
- PythonOperator использует для запуска функции Python
  Роль ОАГ состоит в том, чтобы организовать выполнение набора операторов. Задачи же - как оболочка для операторов. С этими задачами можно будет в будущем работать дополнительно, и потом уже делать зависимости между задачами для ОАГ

## Запуск функции Python с помощью PythonOperator

```
def _get_pictures(): # Убеждаемся, что каталог существует
    pathlib.Path("tmp/images").mkdir(parents=True, exist_ok=True)

    # Скачиваем все изображения в launches.json
    with open("tmp/launches.json") as f:
        launches = json.load(f)
        image_urls = [launch["image"] for launch in launches["results"]]
        for image_url in image_urls:
            try:
                response = requests.get(image_url)
                image_filename = image_url.split("/")[-1]
                target_file = f"tmp/images/{image_filename}"
                with open(target_file, "wb") as f:
                    f.write(response.content)
                print(f"Downloaded {image_url} to {target_file}")
            except requests_exceptions.MissingSchema:
                print(f"{image_url} appears to be an invalid URL.")
```

```
            except requests_exceptions.ConnectionError:
                print(f"Could not connect to {image_url}.")
```

- **Разбор функции**

1. pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True) - Создает по указанному пути папки. `parents=True` говорит о том, что метод создаст все родительские каталоги при их отсутствии. при `False` же, если их нет, выдаст ошибку. `exist_ok=True` убирает ошибку, если папки уже существуют.
2. Открывает json файл, оттуда читаем информацию и переводим ее в python словарь. После мы для каждого results находим ключ image - это и будет ссылка на скачивание фотографии.
3. Создаем отдельный файл с названием фотографии, разделяя путь запроса символом `/` и оставляя только последний элемент списка - это и будет название файла
4. Создается новый файл с названием фотографии, и в него записывается контент с ссылки на фотографию
5. Если ошибки с URL или с подключением, то пишем ошибки, но не останавливаем выполнение скрипта.

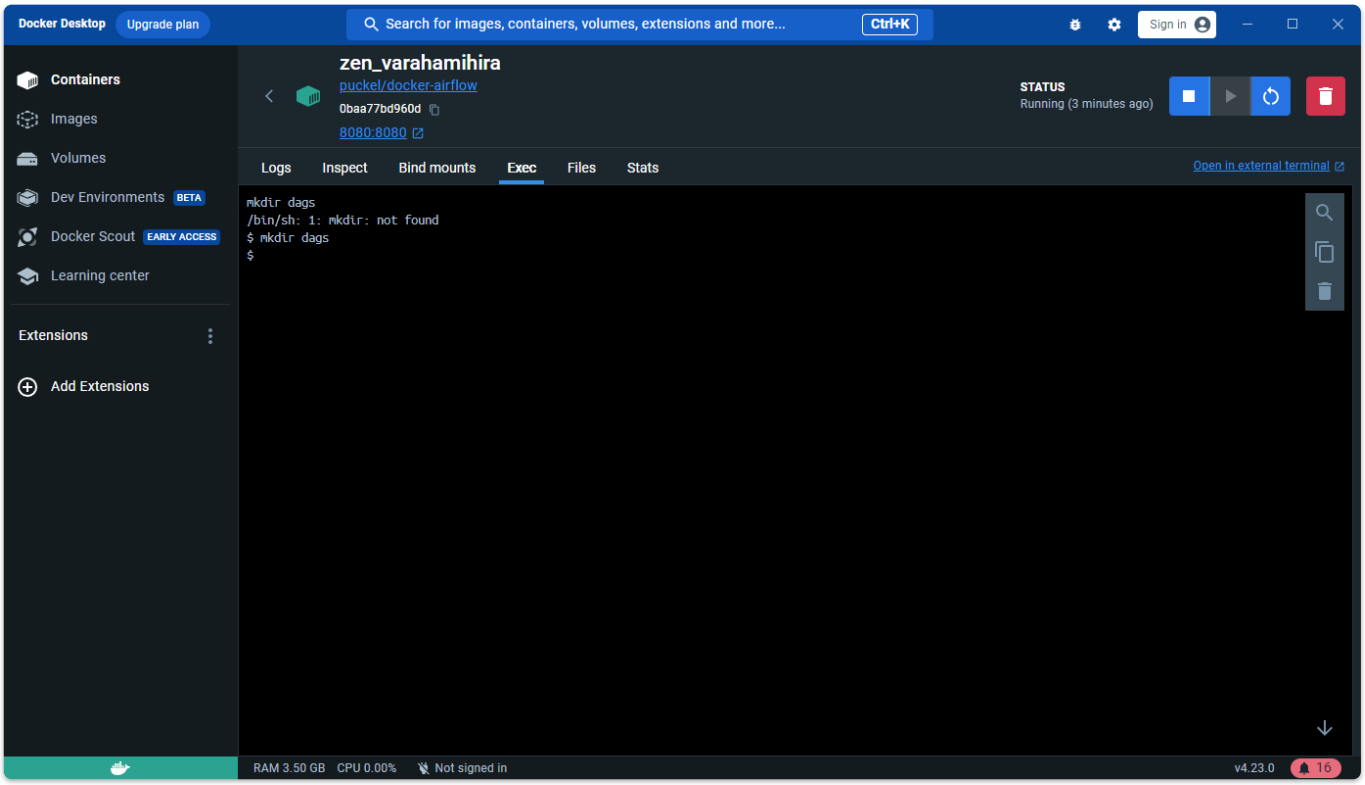
Дальше уже всё предельно понятно, третий оператор просто пишет кол-во фотографий в папке.

## Запуск ОАГ в Airflow 😱

Теперь, когда у нас есть ОАГ, или же DAG файл, давайте запустим его в пользовательском интерфейсе Airflow.

Запустим Docker и наш готовый контейнер (Прочитать об этом можно в proj1_theory). Теперь напишем в терминале команду, которая переместит наш DAG файл в нужный контейнера:

1. Так как изначально папки dags у нас нет, то ее мы должны сначала создать, а потом уже записывать туда файл. Сделать это можно командой `mkdir dags` внутри консоли контейнера




2. Передать файл в контейнер командой:

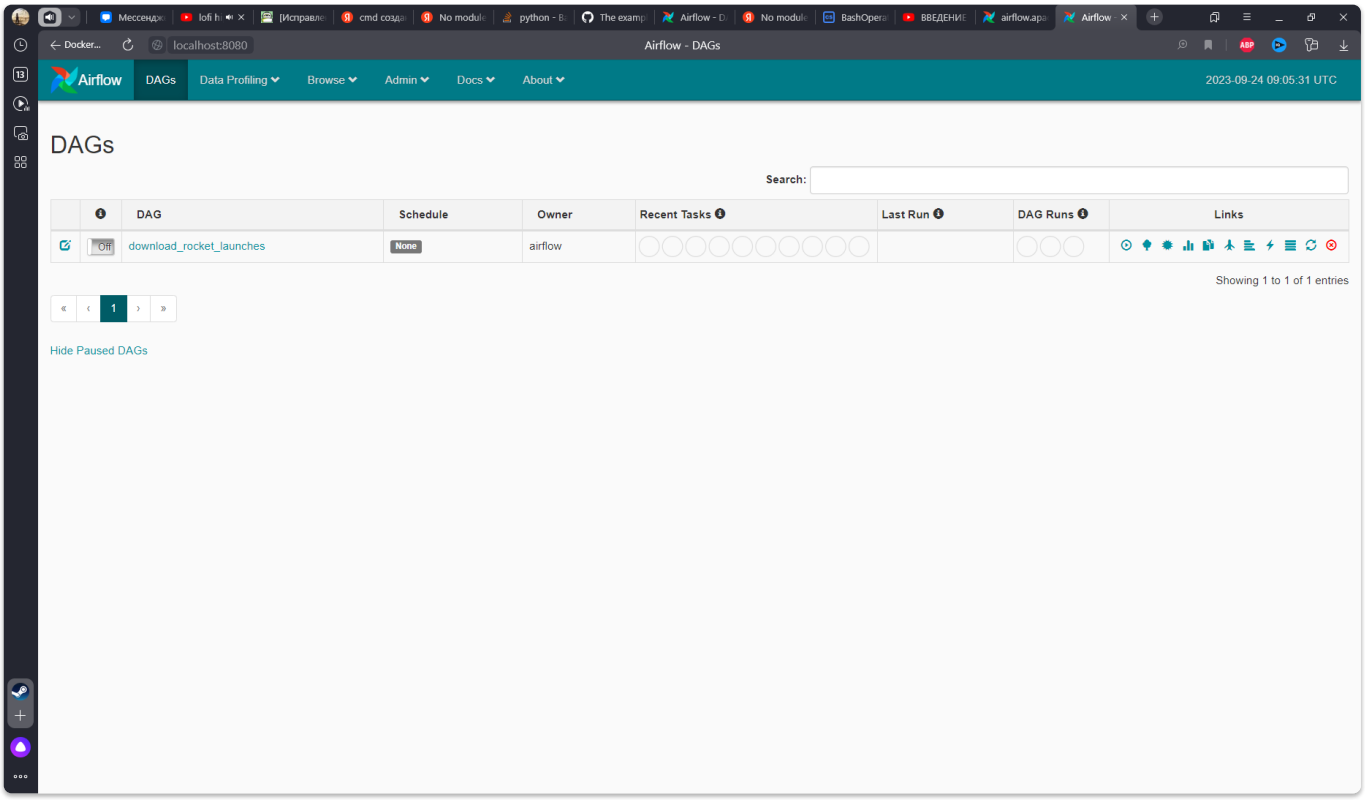
```
docker cp D:\Mindspace\Apache\proj1_practice\etl_try_1.py
0baa77bd960ddf01fd094d1e4de0dea52b76ab96e57a6cebb82a0e7180191581:/usr/local/airflow/dags/
etl_try_1.py
```

---

Если что-то пошло не так, можно заново поставить контейнер а тот удалить.

```
docker pull puckel/docker-airflow
docker run -d -p 8080:8080 puckel/docker-airflow webserver
```

Теперь зайдем на Airflow, и посмотрим, появился ли наш ОАГ в списке.

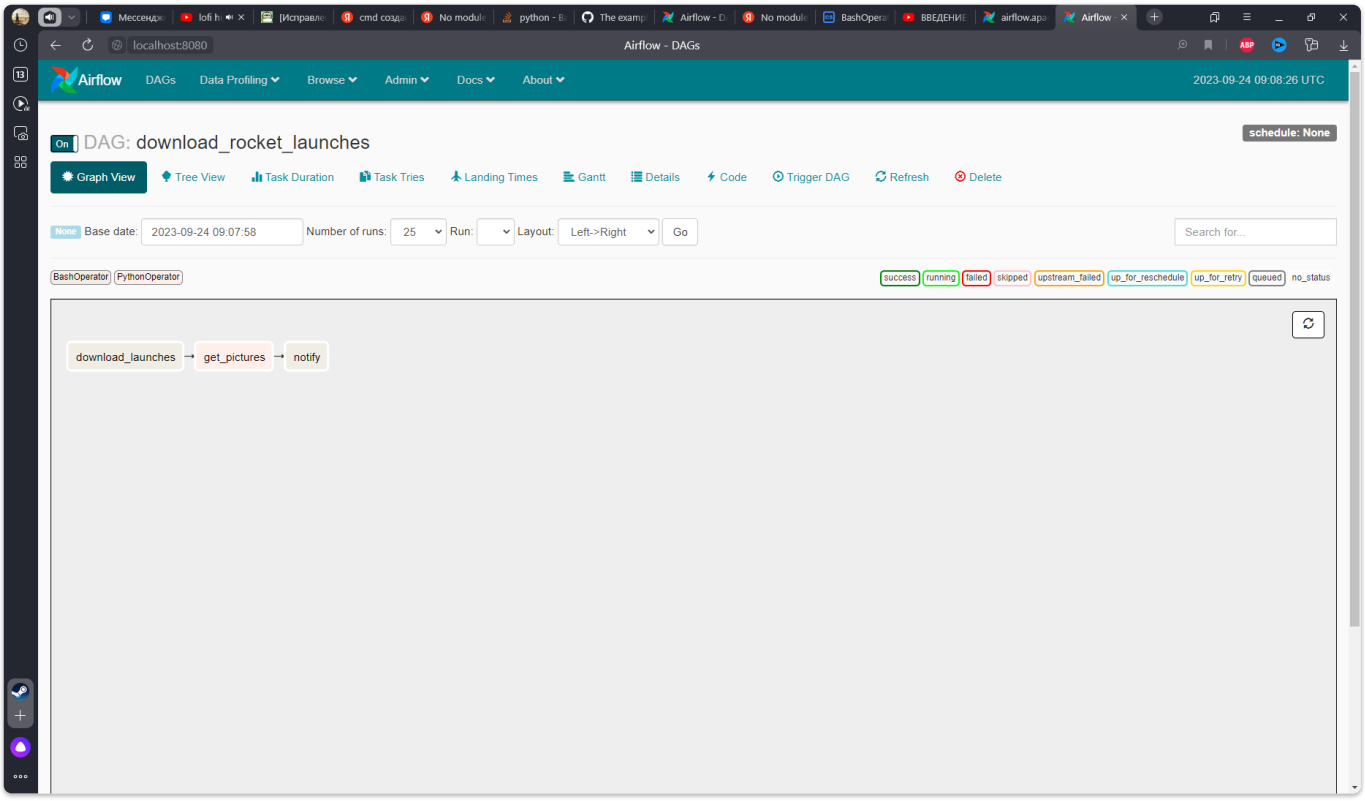



## Поздравляю, вы успешно добавили свой первый DAG файл!

# Разбор интерфейса Airflow/DAGs 🗒️

Откроем наш DAG файл, и посмотрим, какие средства за контролем у нас имеются.

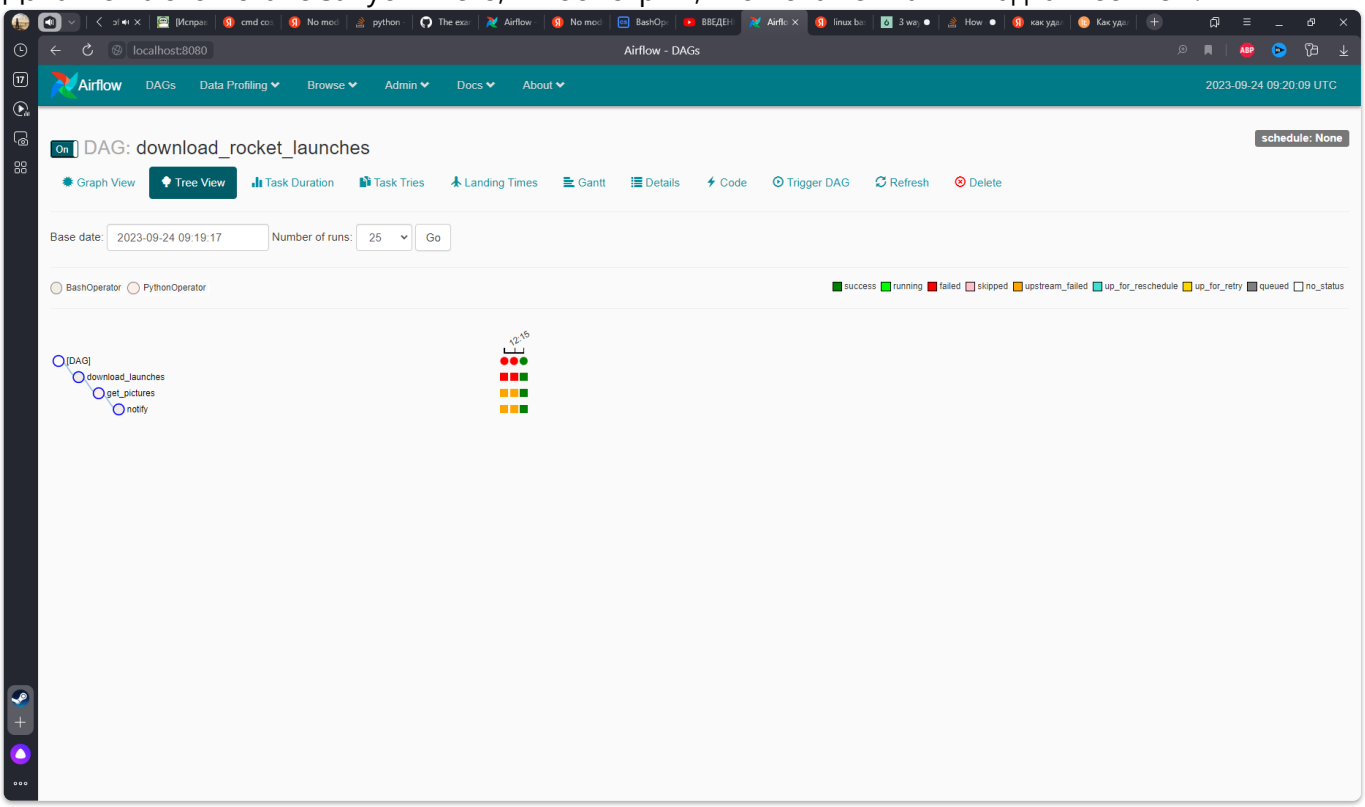
## Graph View 📊



Данная вкладка очень для нас полезна, потому что тут можно смотреть, правильно ли у нас выстроились зависимости. И уж поверьте, в будущем у нас будут настолько масштабные проекты, что эта возможность очень нас выручит, так как у нас появится возможность следить за всеми процессами.

## Tree View

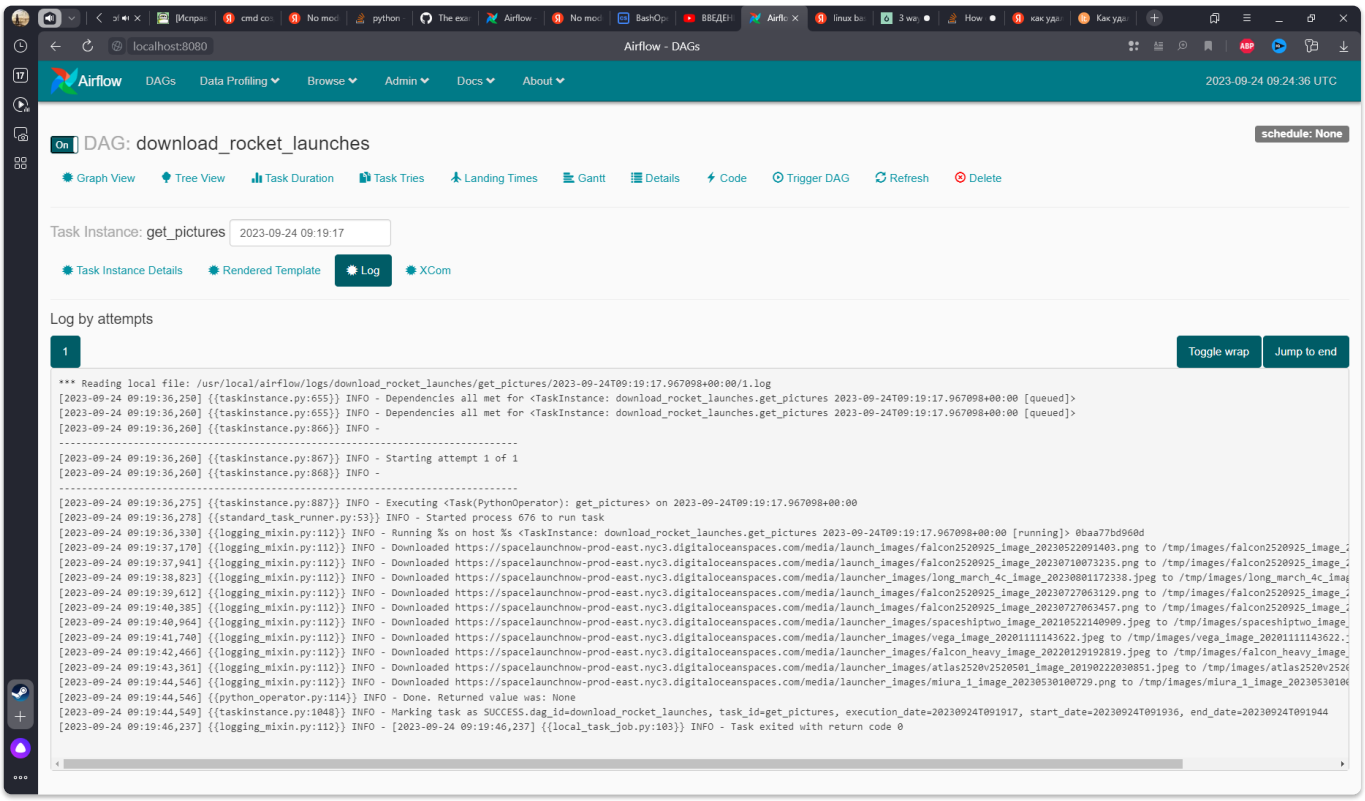
Давайте на этом этапе запустим его, и посмотрим, что покажет нам вкладка Tree View.




### Что за кружочки

Как можно заметить, я запускал этот DAG не 1, а три раза. В первые 2 раза у меня вылезла ошибка, потому что я неправильно поставил указание tmp папки (она изначально есть в Linux, а я решил записать файлы в другую tmp, которую не создал еще к тому же). Но, как вы можете увидеть, в последний раз задача была выполнена правильно! И теперь, мы можем пойти и посмотреть, что за фотографии ближайших к запуску ракет наш скрипт выцепил.

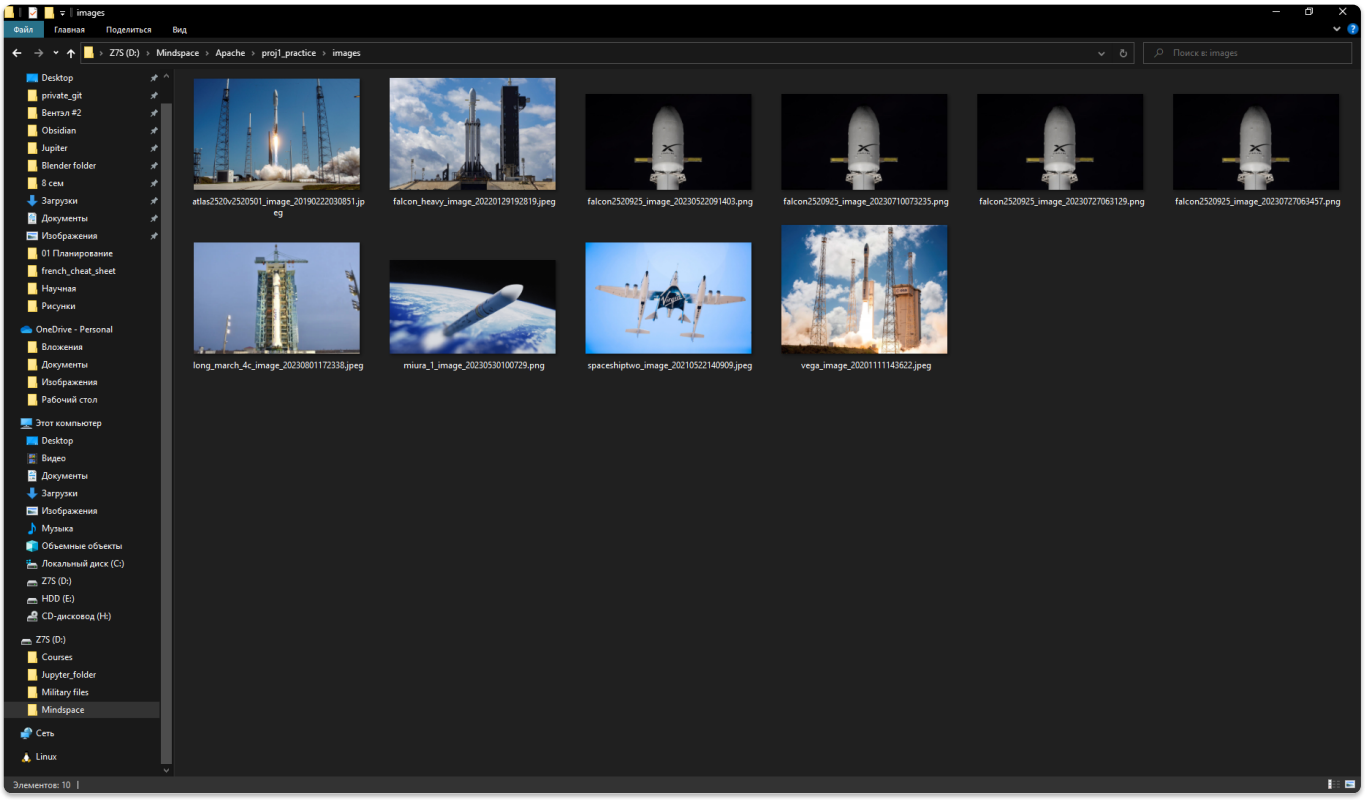
Также можно посмотреть Логи задачи get_pictures, внутри можно увидеть, что все фотографии были успешно загружены:




Ну, и момент истины, сами фотографии, которые находятся в папке /tmp/images/.
Чтобы их вытащить из контейнера, воспользуемся обратной как в начале командой для экспорта всей папки:

```
docker cp 0baa77bd960ddf01fd094d1e4de0dea52b76ab96e57a6cebb82a0e7180191581:/tmp/images
D:\Mindspace\Apache\proj1_practice\
```

Получилось!
Вот наши фотографии у нас, на нашем компе:




Конечно, можно было добавить еще пару задач на автоматическую выгрузку к нам на ПК, но да ладно уж, и так достаточно))

---

## Заключение 🥳

Сегодня мы разобрали очень важную и интересную тему, научились создавать свои DAG файлы. В следующем занятии мы изучим **запуск DAG через равные промежутки времени**, чтобы получать такие вот картинки, ну, допустим каждый день. Также можно будет рассмотреть все возможные **Операторы**.
Спасибо за прочтение, если вы это сделали, то вы **Огромнейший молодец**)