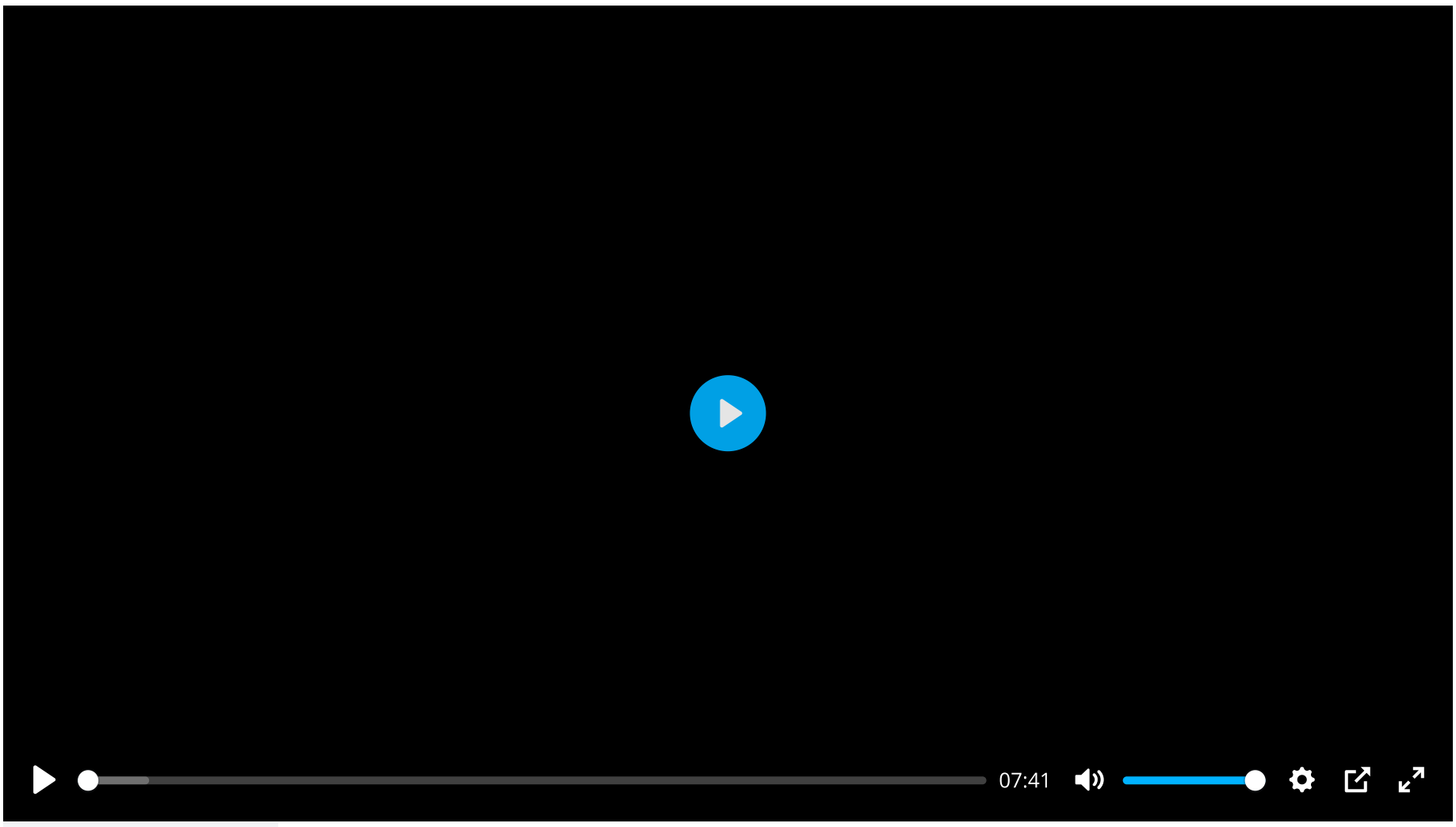
8.4 Работа с компонентами Airflow



Презентация к уроку 8.4.5

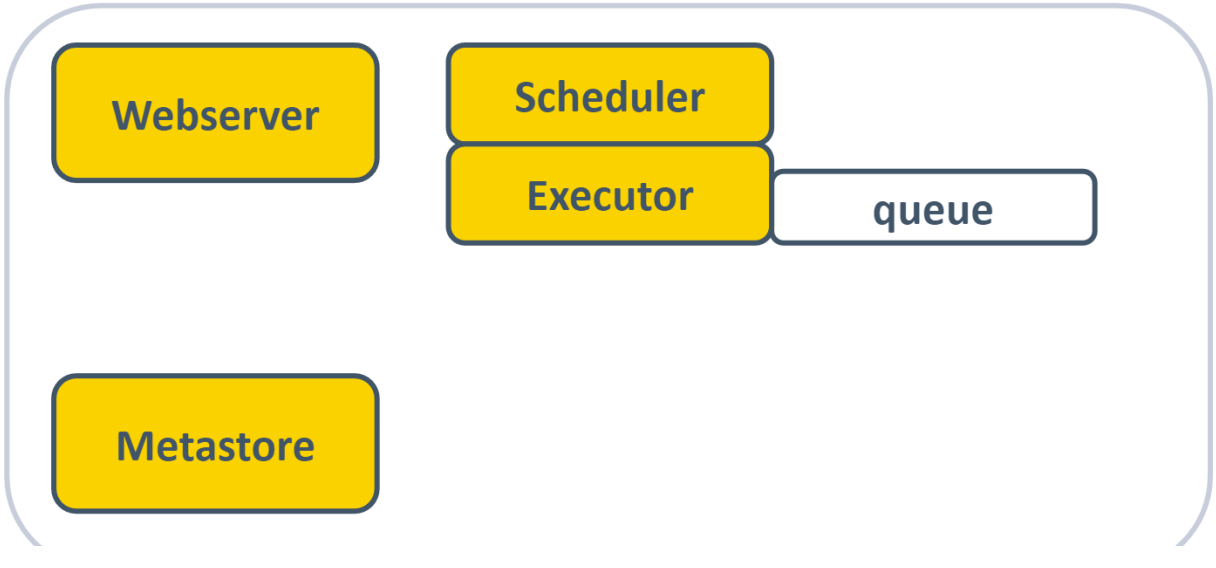
Текстовая расшифровка видео:

КОМПОНЕНТЫ AIRFLOW. ИСПОЛНИТЕЛЬ (EXECUTOR)

План:

- Базы данных и executor;
- Celery-кластер;
- Sequential Executor (последовательный);
- Local Executor;
- Celery Executor;
- Пример прохождения таском пайплайна в случае Celery Executor;
- Механизм работы Celery Executor.

Базы данных и executor



Несмотря на то, что executor не занимается непосредственным выполнением тасков (этим занимаются системы на воркерах), **executor** занимается роутингом тасков, то есть перенаправляет их по системам.

Executor'ы бывают:

- **Локальные** – те, которые выполняются на одной машине;
- **Удаленные** – те, которые выполняются на нескольких нодах и поддерживают распределенную архитектуру.

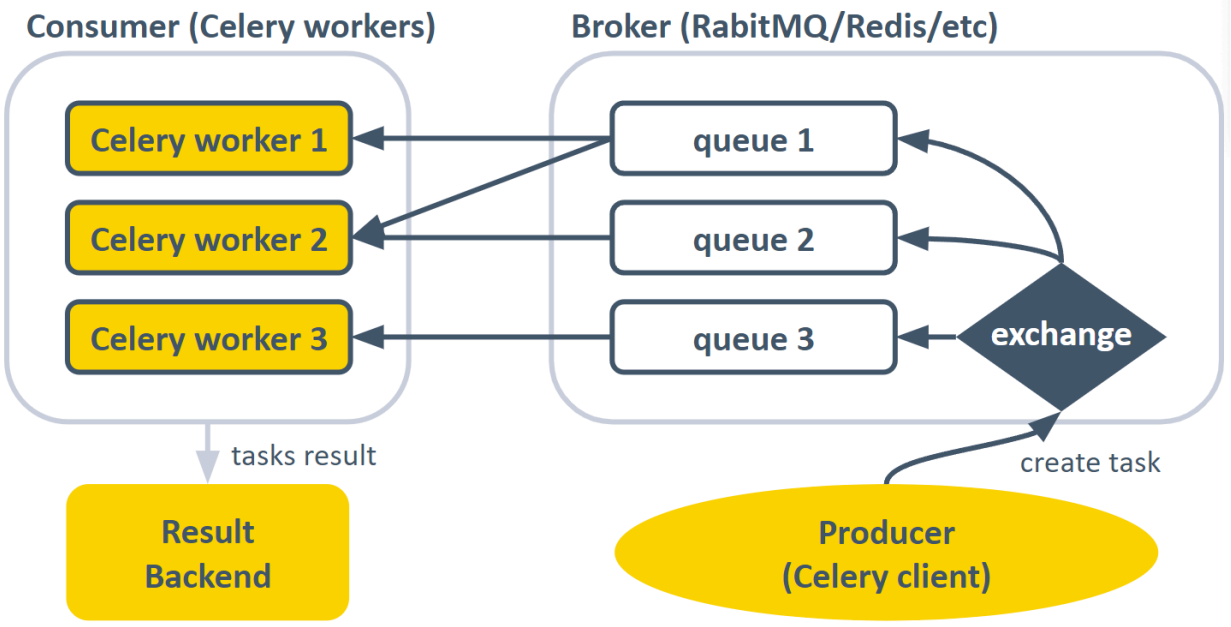
Бывают **последовательные executor'ы**, которые поддерживают выполнение одного таска в одно время. Можно задавать такой executor, который может выполнять сразу несколько тасков одновременно.

Обычно executor'ам соответствуют **воркеры**, если они поддерживают асинхронное выполнение – **очереди**.

Celery-кластер

Среди удаленных executor'ов отметим Celery-кластер, где **Celery** – это софт, который позволяет работать с асинхронными очередями задач.

Чтобы определить executor, нам необходимо указать какие-то данные в файле конфигурации, как правило, это настройки подключения к базе данных:

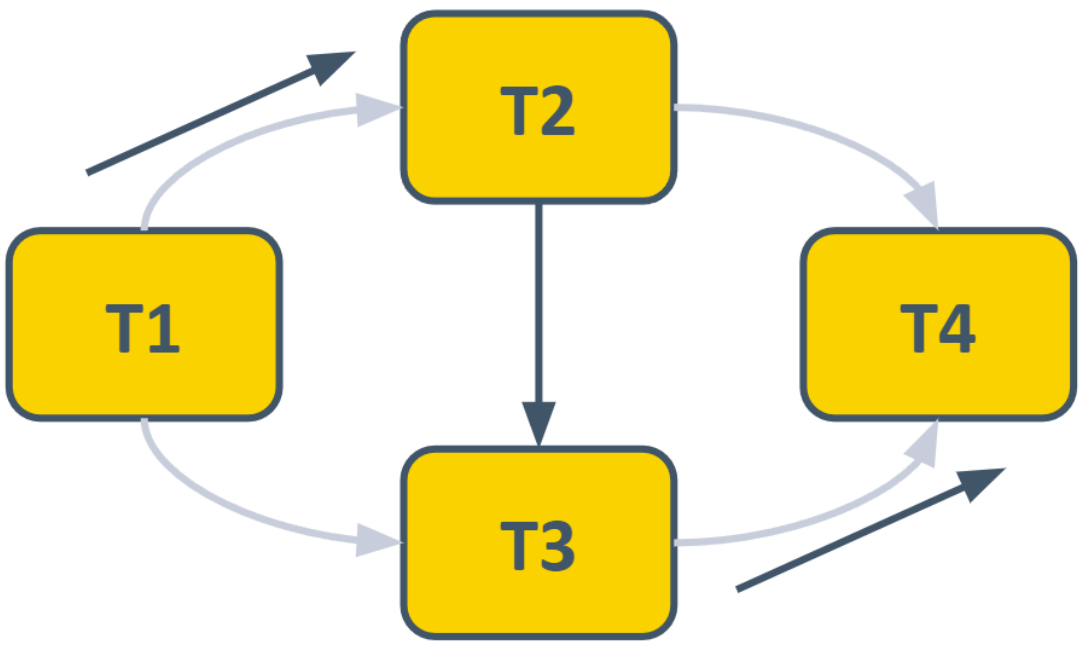


Sequential Executor (последовательный)

Это executor, который выполняет локально одну задачу все время на одной ноде.

В случае, если есть такая схема DAG, где таски «T2» и «T3» находятся на одном уровне, они могут выполняться одновременно. Однако из-за ограничений executor'а вначале выполняется один, потом другой таск.

Scheduler позволяет выполнять один таск в одно время, поэтому данному executor'у необязательно ставить очередь. Чтобы его задавать, мы пишем в конфиге «**executor=SequentialExecutor**»:



Дедлайн 04 августа, 23:59 Мск

Local Executor

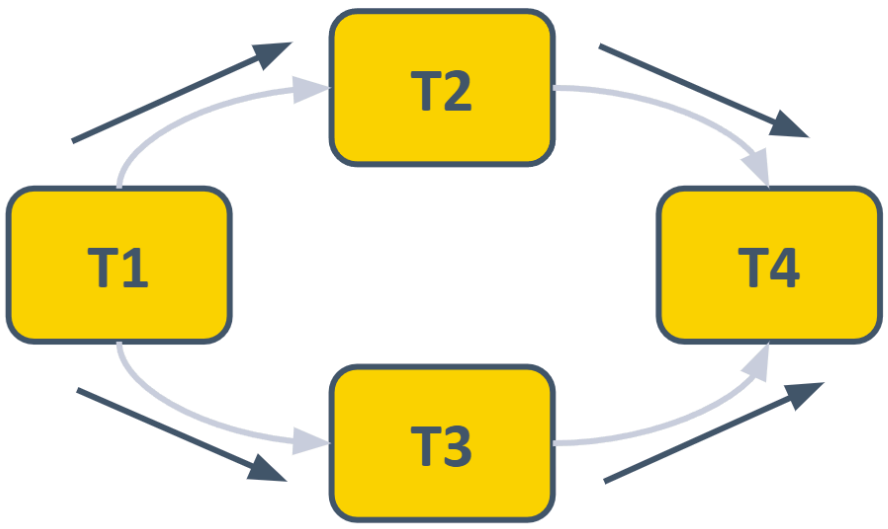
Local Executor – это локальный executor, который выполняется на одной машине, но уже поддерживает выполнение нескольких задач одновременно.

Здесь уже не подходит создаваемая по умолчанию SQLite, поскольку она не поддерживает несколько одновременных подключений, и нам необходимо самостоятельно прописывать подключение к базе данных (postgres, MySQL).

Чтобы указать Local Executor, мы добавляем файл в конфиг «**executor= LocalExecutor**», прописываем подключение к базе через параметр **sql_alchemy_conn**. Аналогичные параметры есть и в Docker-compose:

```
executor=LocalExecutor

sql_alchemy_conn=db+psycorg2://{user}:{password
}@{host}:{port}/airflow
```



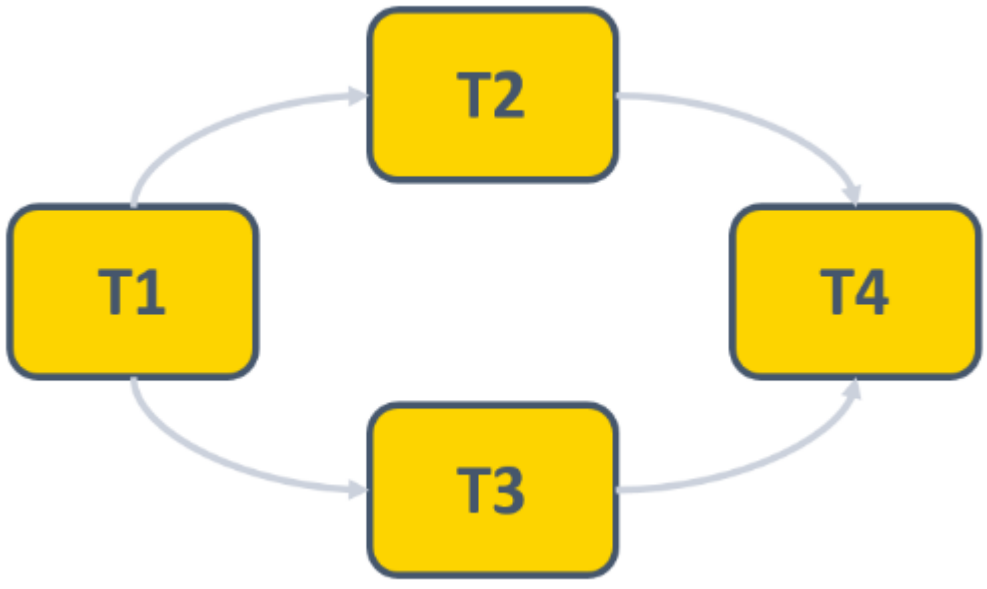
Здесь мы прописываем данные для подключения к базе.

Celery Executor

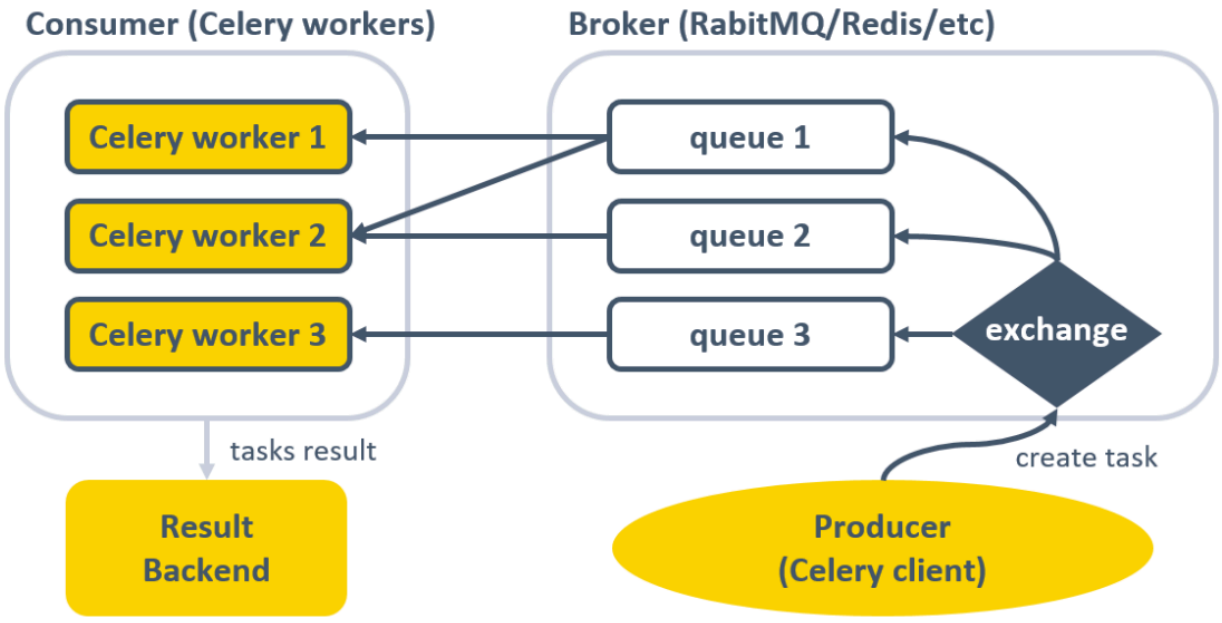
Celery Executor поддерживает исполнение тасков на нескольких узлах одновременно.

Здесь уже обязательно присутствуют брокеры очередей. Задачи могут выполняться по несколько одновременно, их ограничения прописываются в файле конфигурации.

Celery – это система для исполнения нескольких тасков одновременно на разных машинах или в разных потоках. Здесь возможно использование сразу нескольких воркеров, так как архитектура предполагает распределение тасков.



Дедлайн 04 августа, 23:59 Мск



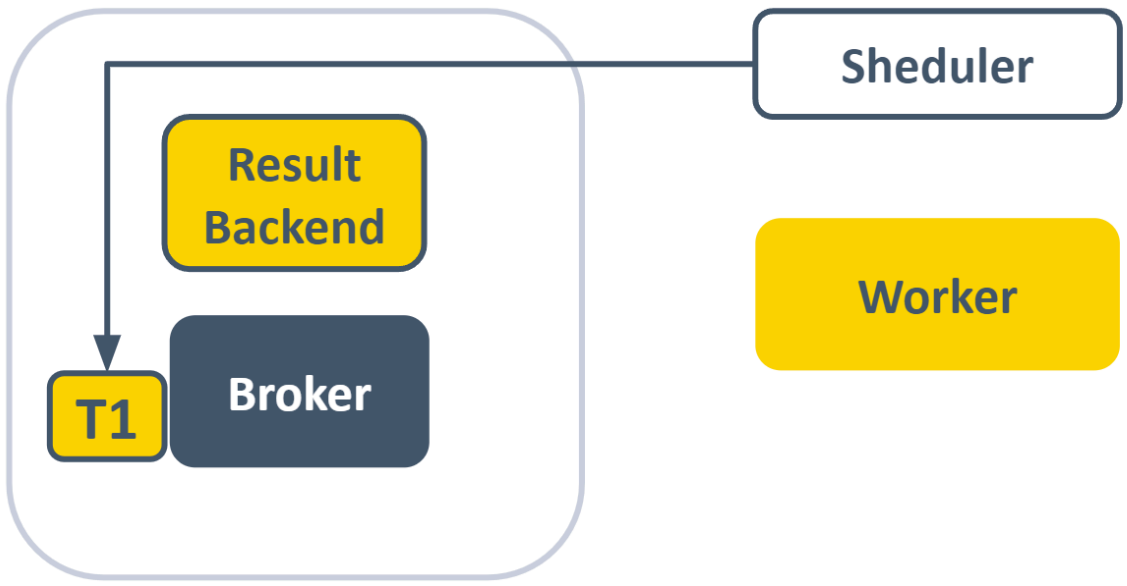
Celery состоит из двух компонентов:

- **Result Backend**, где таски сохраняют свое значение после исполнения (успешного/неуспешного);
- **Очереди**.

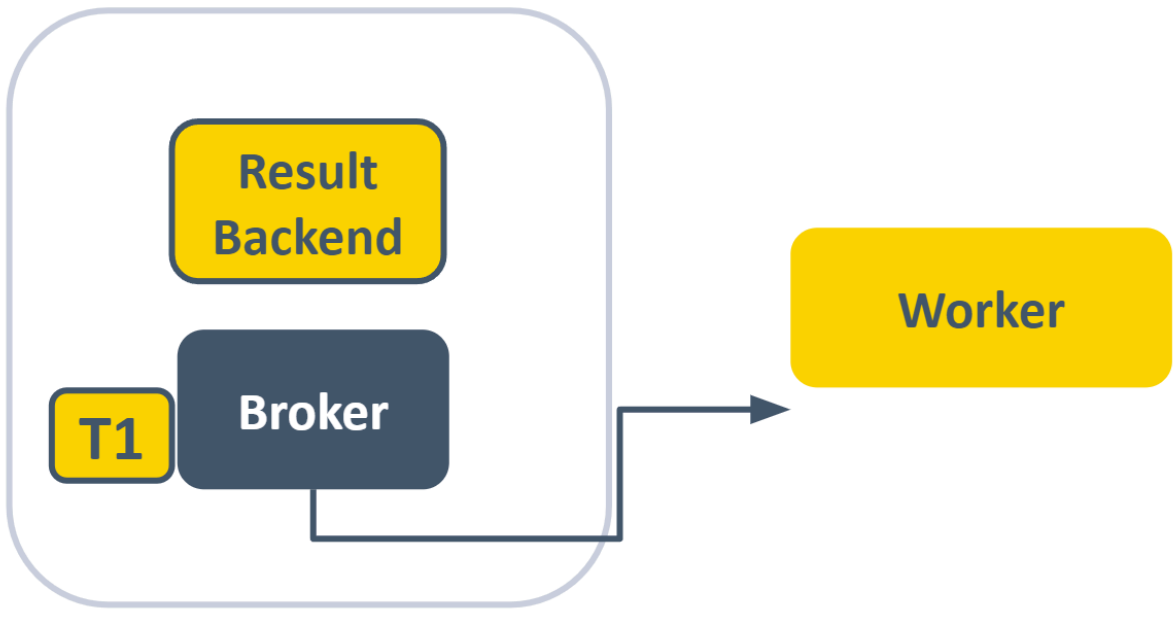
Scheduler отправляет таски в брокер, откуда отправляет их через exchange в очереди; из очередей таски отправляются по воркерам.

Пример прохождения таском пайплайна в случае Celery Executor

Рассмотрим, как работает данная схема:

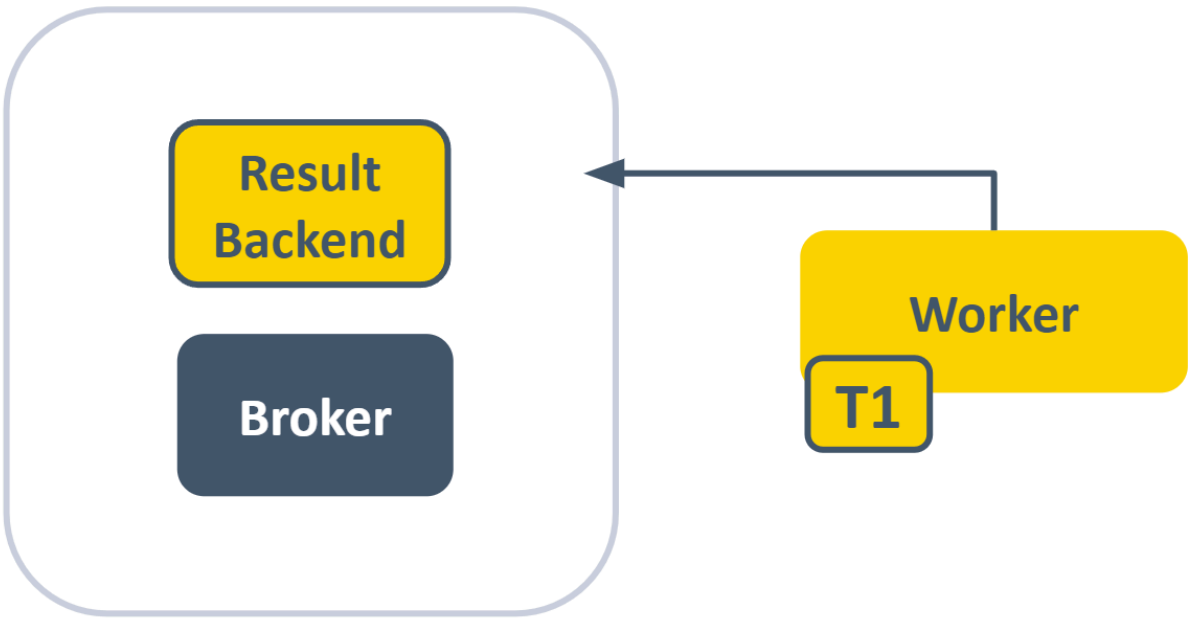


Scheduler отправляет некий таск (T1) в брокер.



Воркер забирает таск из брокера (из очереди). После выполнения отправляет таск в Result Backend. Result Backend – наш Metastore/База Данных.

Дедлайн 04 августа, 23:59 Мск



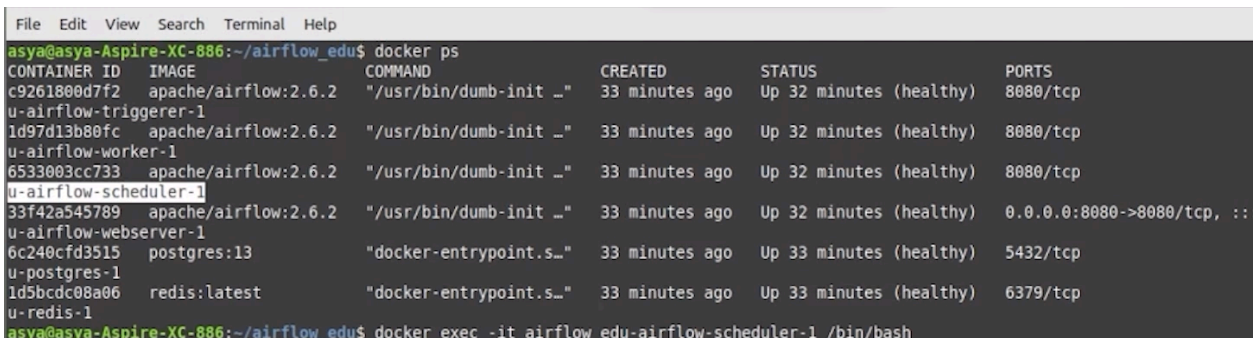
Механизм работы Celery Executor

Чтобы настроить Celery Executor нам потребуется задать брокер очередей, подключение к брокеру, подключение к базе данных, а также подключение к Result Backend (это может быть та же самая база, что и Metastore); в executor'е пишем «executor=CeleryExecutor»:

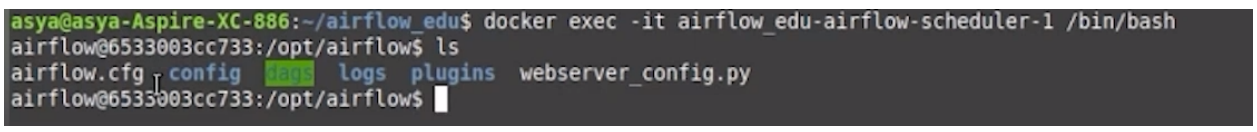
```
executor=CeleryExecutor
sql_alchemy_conn=db+psycorg2://{user}:{password}@{host}:{port}/airflow
celery_result_backend=db+postgresql://airflow:****@postgres/airflow
broker_url= pyamqp://guest:guest@{RabbitMQ-HOSTNAME}:5672/
```

Рассмотрим на практике:

Предварительно запустив docker-compose, сделав docker compose app, зайдем в контейнер Scheduler, чтобы посмотреть файл конфигурации:



Здесь лежит файл «AirFlow.cfg» – файл конфигурации:



Посмотрим его начало. Здесь есть параметр Executor – SequentialExecutor:

Дедлайн 04 августа, 23:59 Мск

```
File  Edit  View  Search  Terminal  Help
asya@asya-Aspire-XC-886:~/airflow_edu$ docker exec -it airflow_edu-airflow-scheduler-1 /bin/bash
airflow@6533003cc733:/opt/airflow$ ls
airflow.cfg  config  dags  logs  plugins  webserver_config.py
airflow@6533003cc733:/opt/airflow$ head -n 40 airflow.cfg
[core]
# The folder where your airflow pipelines live, most likely a
# subfolder in a code repository. This path must be absolute.
dags_folder = /opt/airflow/dags

# Hostname by providing a path to a callable, which will resolve the hostname.
# The format is "package.function".
#
# For example, default value "airflow.utils.net.getfqdn" means that result from patched
# version of socket.getfqdn() - see https://github.com/python/cpython/issues/49254.
#
# No argument should be required in the function specified.
# If using IP address as hostname is preferred, use value ``airflow.utils.net.get_host_ip_address``
hostname_callable = airflow.utils.net.getfqdn

# A callable to check if a python file has airflow dags defined or not
# with argument as: `(file_path: str, zip_file: zipfile.ZipFile | None = None)`
# return True if it has dags otherwise False
# If this is not provided, Airflow uses its own heuristic rules.
might_contain_dag_callable = airflow.utils.file.might_contain_dag_via_default_heuristic

# Default timezone in case supplied date times are naive
# can be utc (default), system, or any IANA timezone string (e.g. Europe/Amsterdam)
default_timezone = utc

# The executor class that airflow should use. Choices include
# ``SequentialExecutor``, ``LocalExecutor``, ``CeleryExecutor``, ``DaskExecutor``,
# ``KubernetesExecutor``, ``CeleryKubernetesExecutor`` or the
# full import path to the class when using a custom executor.
executor = SequentialExecutor

# This defines the maximum number of task instances that can run concurrently per scheduler in
# Airflow, regardless of the worker count. Generally this value, multiplied by the number of
# schedulers in your cluster, is the maximum number of task instances with the running
# state in the metadata database.
parallelism = 32

# The maximum number of task instances allowed to run concurrently in each DAG. To calculate
# the number of tasks that is running concurrently for a DAG, add up the number of running
# tasks for all DAG runs of the DAG. This is configurable at the DAG level with ``max_active_tasks``,
airflow@6533003cc733:/opt/airflow$
```


Как мы знаем, это последовательный, который поддерживает выполнение одного таска последовательно на одной ноде.

В docker-compose.yaml прописан другой параметр. Когда мы запускаем через docker-compose, это считывается оттуда. Так, у нас стоит Celery Executor и подключена очередь через docker-compose-файл.

Посмотрим подключение к базе данных, прописанное в файле конфигурации:

```
File  Edit  View  Search  Terminal  Help
# This defines the port that Celery Flower runs on
airflow@6533003cc733:/opt/airflow$
airflow@6533003cc733:/opt/airflow$
airflow@6533003cc733:/opt/airflow$ grep sql_alchemy airflow.cfg -B 10 -A 10

# Kwargs to supply to dataset manager.
# Example: dataset_manager_kwargs = {"some_param": "some_value"}
# dataset_manager_kwargs =

[database]
# The SqlAlchemy connection string to the metadata database.
# SqlAlchemy supports many different database engines.
# More information here:
# http://airflow.apache.org/docs/apache-airflow/stable/howto/set-up-database.html#database-uri
sql_alchemy_conn = sqlite:////opt/airflow/airflow.db

# Extra engine specific keyword args passed to SQLAlchemy's create_engine, as a JSON-encoded value
# Example: sql_alchemy_engine_args = {"arg1": True}
# sql_alchemy_engine_args =

# The encoding for the databases
sql_engine_encoding = utf-8

# Collation for ``dag_id``, ``task_id``, ``key``, ``external_executor_id`` columns
# in case they have different encoding.
# By default this collation is the same as the database collation, however for ``mysql`` and ``mariadb``
# the default is ``utf8mb3_bin`` so that the index sizes of our index keys will not exceed
# the maximum size of allowed index when collation is set to ``utf8mb4`` variant
# (see https://github.com/apache/airflow/pull/17603#issuecomment-901121618).
# sql_engine_collation_for_ids =

# If SqlAlchemy should pool database connections.
sql_alchemy_pool_enabled = True

# The SqlAlchemy pool size is the maximum number of database connections
# in the pool. 0 indicates no limit.
sql_alchemy_pool_size = 5

# The maximum overflow size of the pool.
# When the number of checked-out connections reaches the size set in pool_size,
# additional connections will be returned up to this limit.
# When those additional connections are returned to the pool, they are disconnected and discarded.
# It follows then that the total number of simultaneous connections the pool will allow
# is pool_size + max_overflow,
```


Здесь прописано подключение к SQLite.

Мы познакомились со всеми компонентами Apache AirFlow, узнали для чего они нужны, а также познакомились с различными видами архитектуры.

Дедлайн 04 августа, 23:59 Мск