Apache AirFlow

NDBI046: Practical class 6

User Story

- We want to *automate* and *schedule* ETL workflow execution

Prerequisite: Setting up Apache Airflow in Docker (1/2)

- *Install Docker Desktop* (or Docker and Docker Compose)
  - Download: https://www.docker.com/products/docker-desktop/
- *Launch Docker Desktop* and verify that Docker is running

- *Create* a new *folder* for the *Airflow project* and navigate to the folder
  - e.g., mkdir ~/Projects/python-ndbi046/airflow
  - cd ~/Projects/python-ndbi046/airflow

- *Extend* the Docker *container* with additional Python dependencies
  - Download the Dockerfile, docker-compose.yaml, and requirements.txt from the practical class website
  - Execute docker build . --tag mff/airflow:latest

- Create *new folders for DAGs*, *logs*, customized *plugins*, and *configuration*
  - Execute mkdir -p ./dags ./logs ./plugins ./config

- Linux only: execute echo -e "AIRFLOW_UID=$(id -u)" > .env

```
% docker --version
% docker-compose --version

% mkdir ~/Projects/python-ndbi046/airflow

% cd ~/Projects/python-ndbi046/airflow

% curl -LfO 'https://gitlab.mff.cuni.cz/contosp/ndbi046/-/raw/master/class06/Dockerfile?ref_type=heads&inline=false'

% curl -LfO 'https://gitlab.mff.cuni.cz/contosp/ndbi046/-/raw/master/class06/docker-compose.yaml?ref_type=heads&inline=false'

% curl -LfO 'https://gitlab.mff.cuni.cz/contosp/ndbi046/-/raw/master/class06/requirements.txt?inline=false'

% docker build . --tag mff/airflow:latest

% mkdir -p ./dags ./logs ./plugins ./config
```

if you see version output, you are running Docker

downloading files

extending the official image

source: https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html#running-airflow-in-docker

Prerequisite: Setting up Apache Airflow in Docker (2/2)

- *Initialize the database* for Apache Airflow
  - Execute docker compose up airflow-init
  - At the same time, all docker dependencies are downloaded and the *initial user is created*

- *Start* Airflow *services*
  - Execute docker compose up

- In the second terminal you can *check the condition of the containers* and make sure that all of them are in a healthy condition
  - Execute docker ps

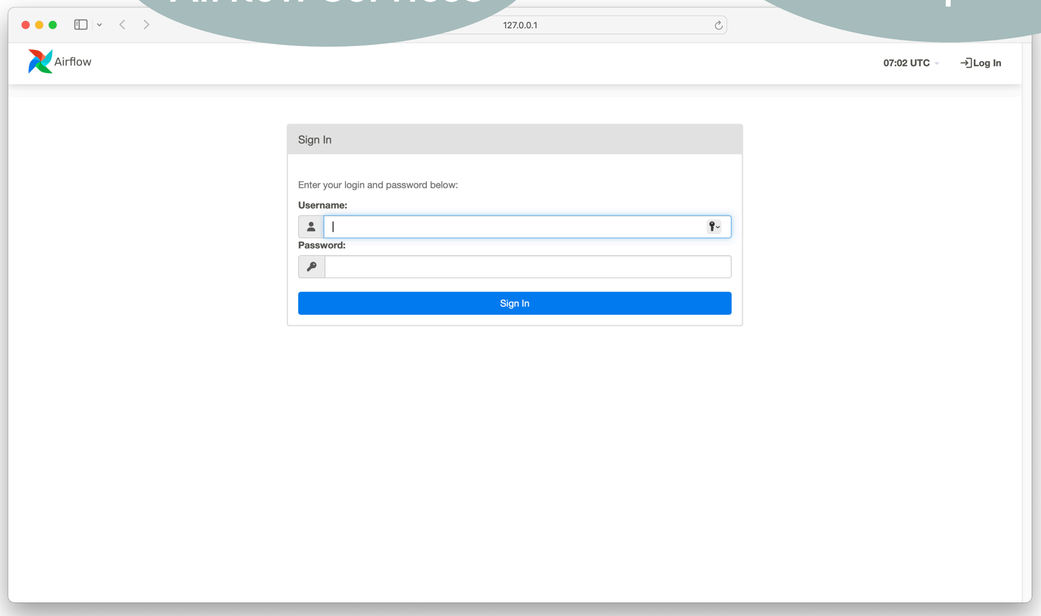
- Go to http://127.0.0.1:8080 and *check* if Airflow (*web server*) is running
  - Username: airflow
  - Password: airflow

```
% docker compose up airflow-init
airflow-init-1  | User "airflow" created with
role "Admin"
airflow-init-1  | 2.8.3
airflow-init-1 exited with code 0

% docker compose up

```

database initialization

launching Airflow services

if you can see the following, the database initialization is complete

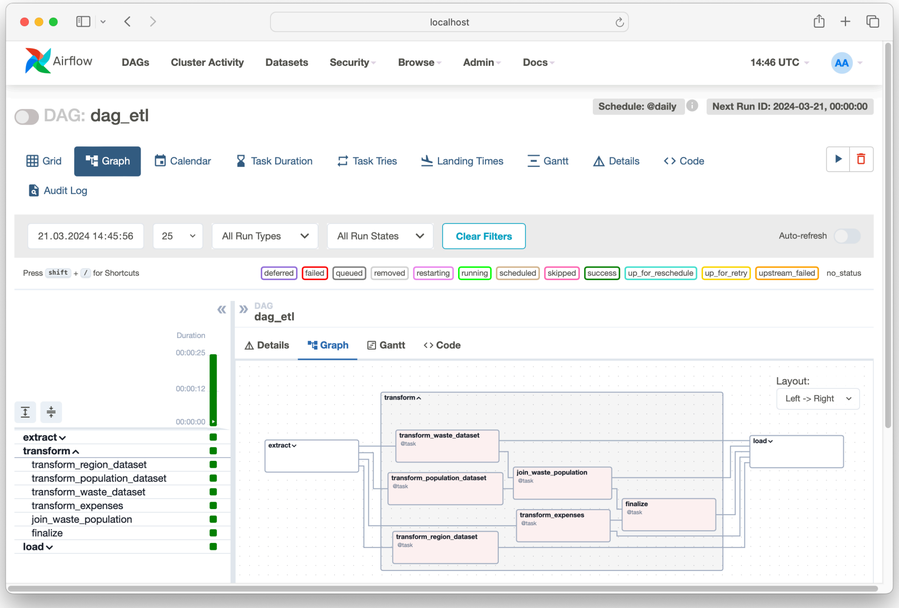


Apache Airflow

- Open-source platform for *developing*, *scheduling*, and *monitoring* batch-oriented *workflows*
- User-friendly interface allowing us to *visualize workflows* and *track the progress* of tasks
- Provides operators to connect with various technologies, e.g., database systems
- Deployable in various setups, from a single process on a single computer to distributed environments

- Architecture
  - The *scheduler organizes* the execution of tasks
  - The *executor* is responsible for the *execution of tasks*
  - *Workers* are distributed processes that *perform tasks*

- Website: https://airflow.apache.org/

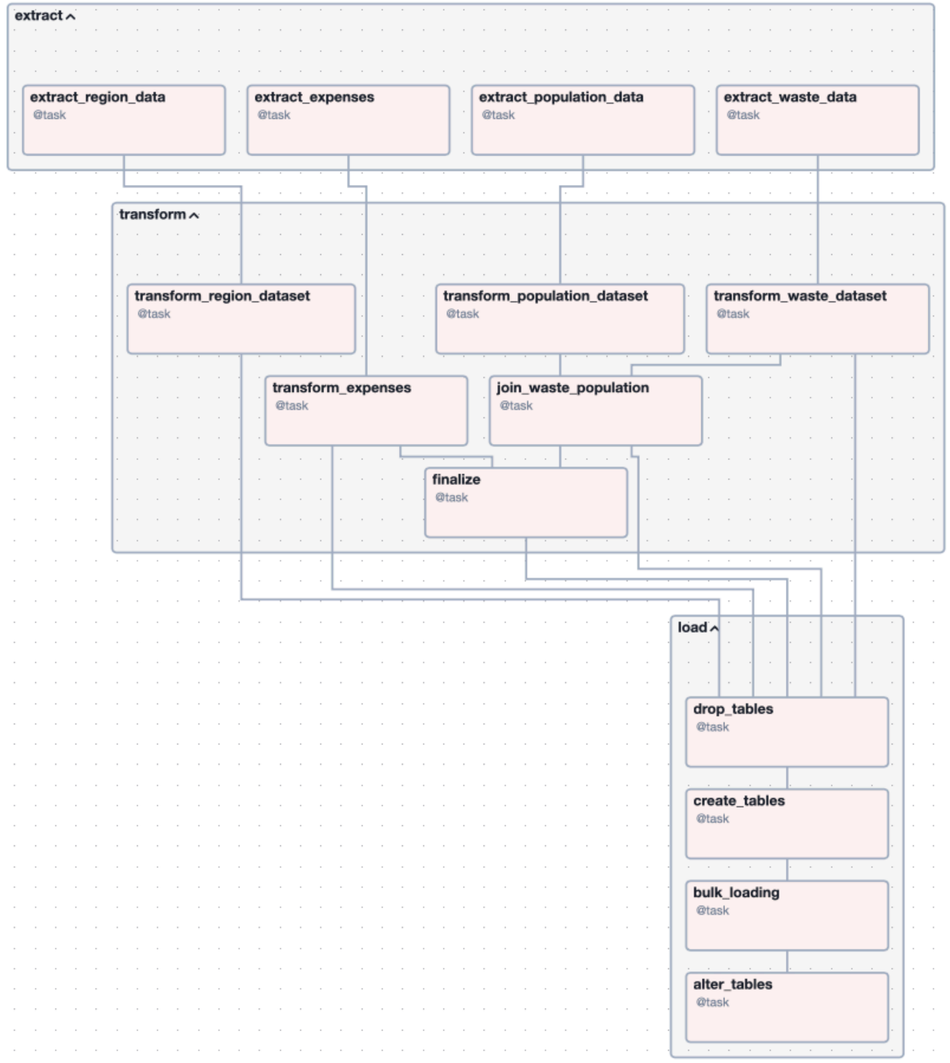


Apache Airflow: Workflow

- Represented as a *directed acyclic graph* (DAG)
  - Consists of *tasks* (i.e. individual parts of the work) and *dependencies* between them
  - The of dependencies determines the order of tasks execution

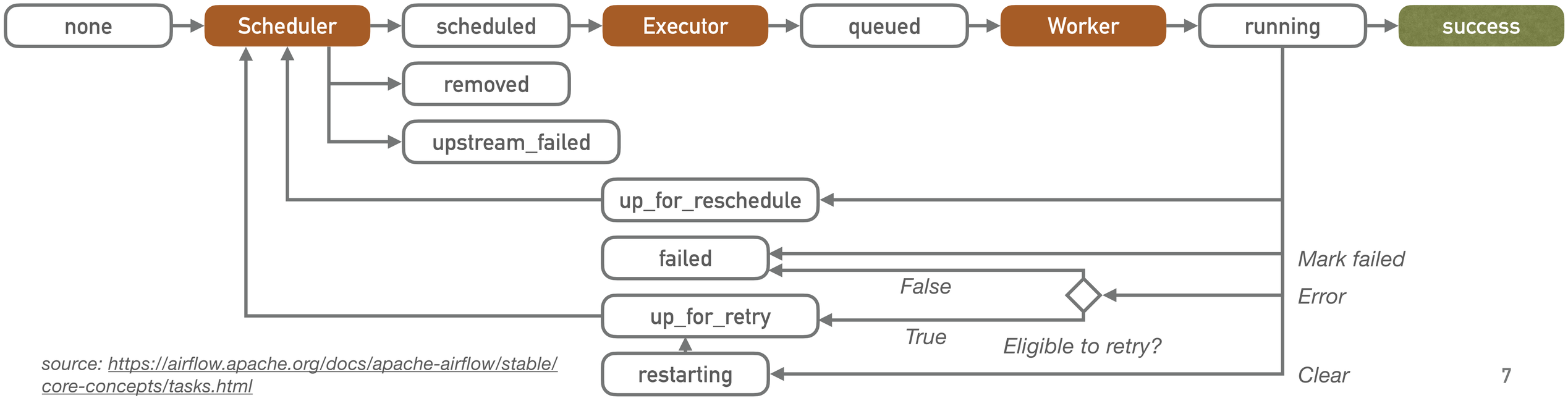
- Three basic kinds of tasks:
  - *Operators* represent *predefined task templates*, e.g.:
    - BashOperator: executes a bash command
    - PythonOperator: calls an arbitrary Python function
    - PostgresOperator: executes a particular SQL statement
  - *Sensors* are special cases of Operators *useful for waiting for en external event* to happen (e.g., upload of a required file)
  - *TaskFlow* allows an ordinary Python function to be decorated as a @task
    - Automatically *calculates the dependencies* between tasks

- Dependencies
  - *Upstream* task directly precedes the other task
  - *Downstream* task directly postpones the other task



Apache Airflow: Task lifecycle

- *none*: the task has not yet been queued for execution
- *scheduled*: Scheduler has determined the tasks should run
- *queued*: the task is assigned to an Executor and is a waiting a Worker
- *running*: the task is running on a worker
- *success*: the task finished running without errors
- *restarting*: while running, the task was externally requested to restart
- *failed*: the task had an error during execution and failed to run
- *skipped*: the task was skipped due to branching, LatestOnly, or similar
- *upstream_failed*: an upstream task failed and the Trigger Rule says we needed it
- *up_for_retry*: the task failed, but has retry attempts left and will be rescheduled
- *up_for_reschedule*: the task is a Sensor that is in reschedule mode
- *deferred*: the task has been deferred to a trigger
- *removed*: the task has vanished from the DAG since the run started



source: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/tasks.html

Example 6.1: BashOperator

- Create a *simple Apache Airflow workflow* consisting of the following tasks:
  - Print the content of webpage https://cs.wikipedia.org/wiki/Kraje_v_Česku
  - Record that the workflow was successfully completed

- Use BashOperator

- Copy the Python script into the dags folder within Airflow project
  - (see Setting up Apache Airflow in Docker)

- **Tip:** If the execution of any task fails, check the *task log* for the reason for the failure

Example 6.1: BashOperator (Solution)

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator

default_args = {"owner": "koupil", "retries": 3, "retry_delay": timedelta(minutes=5)}

with DAG(
  dag_id="dag_bash_operator",
  default_args=default_args,
  description="A simple Apache Airflow workflow to print Wikipedia page content",
  start_date=datetime(2024, 3, 22),
  schedule_interval="@daily",
) as dag:
  task_print_web_page = BashOperator(
    task_id="download_wiki_page",
    bash_command="curl https://cs.wikipedia.org/wiki/Kraje_v_Česku",
  )

  task_finish_work = BashOperator(
    task_id="finish_work",
    bash_command='echo "Work has finished"',
  )

 task_print_web_page >> task_finish_work
```

import DAG and BashOperator

definition of common arguments

create an instance of DAG

schedule_interval specifies the frequency of execution using CRON

the required attributes are dag_id, default_args, start_date and schedule_interval

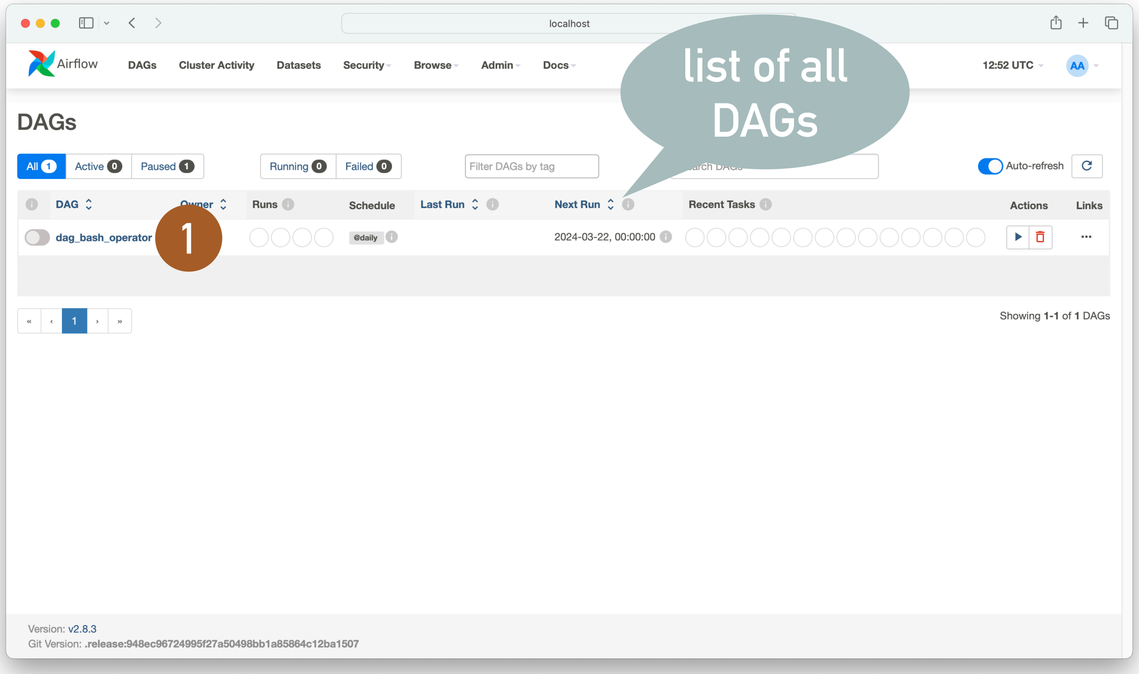
task printing webpage content

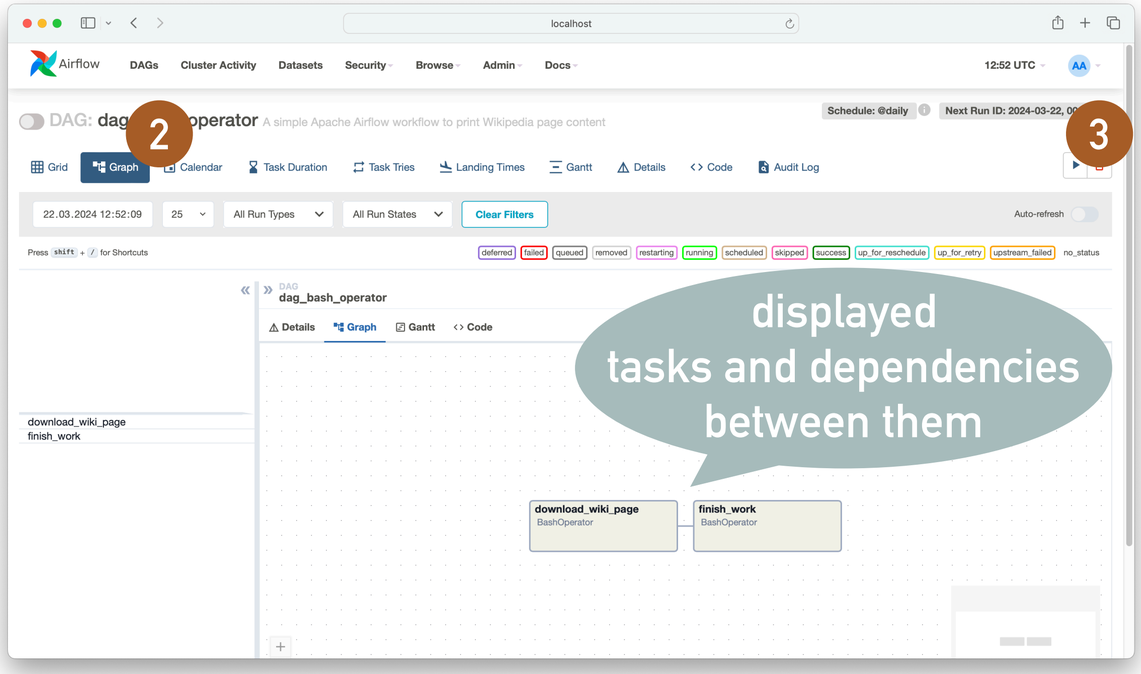
finishing task

task task dependency definition using the bitshift operator

Example 6.1: BashOperator (Solution)

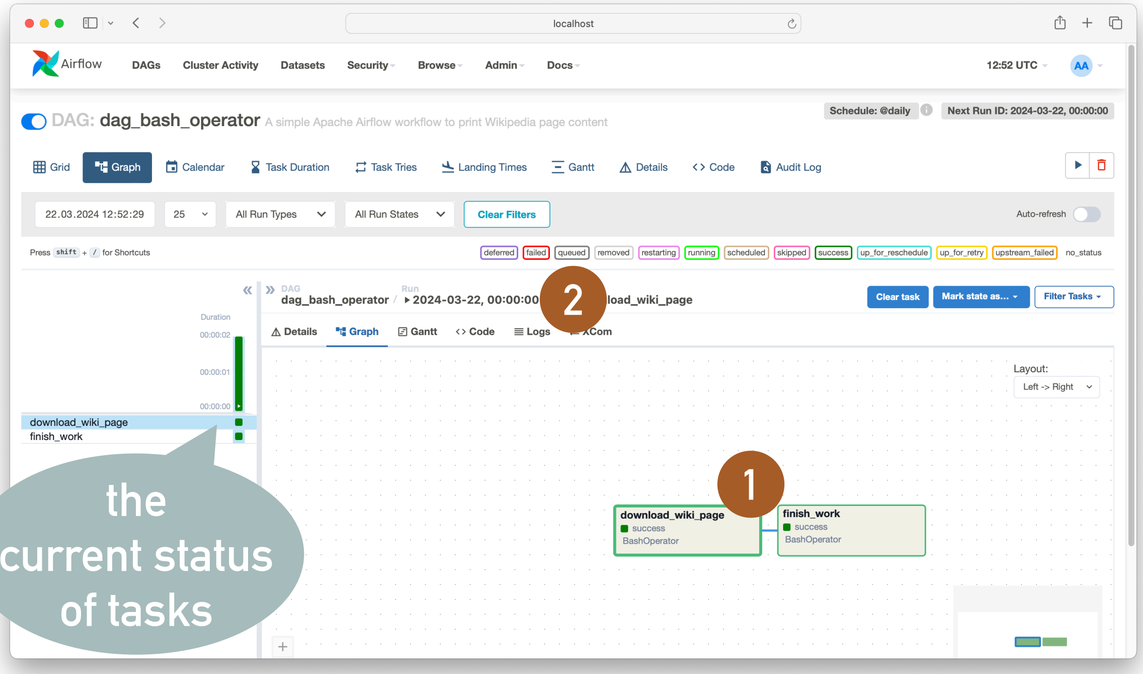
- Copy the Python script to the dags folder
  - On the *main* Apache Airflow *page*, the DAG is displayed (after a while)
- View the detail of a DAG by selecting its name 1
- Select *Graph View* 2
- *Trigger* DAG 3

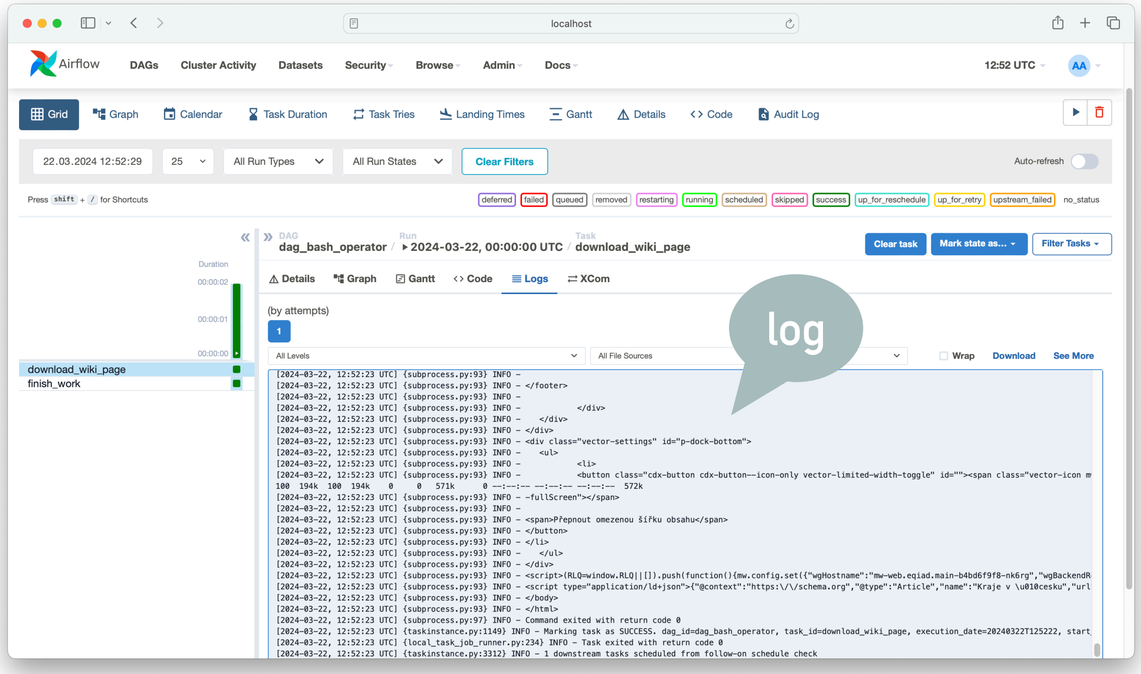




Example 6.1: BashOperator (Solution)

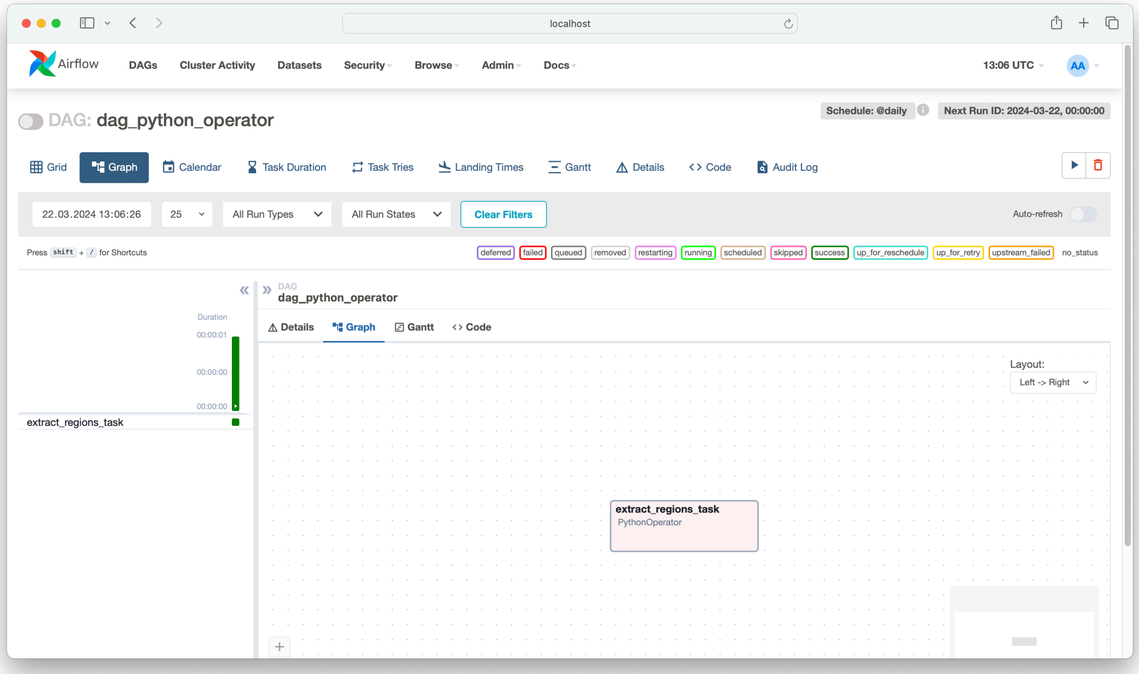
- The *indicator* (i.e., the left panel) reflects the *current task status*
- After (un)successful completion, select a *task* and view the *log* of its run ❶ ❷
  - The log is convenient to use *for debugging*, e.g., in case of an unsuccessful job run





Example 6.2: PythonOperator

- Create Airflow Workflow to *extract the table* 'Základní data o krajích' (Basic data about regions) *from* the *Wikipedia article* about Czech regions
  - Create a DAG consisting of a *single* PythonOperator *implementing* dataset *extraction*
  - Reuse existing solution from Example 2.3
  - The input parameters will be url and output_file_name



Example 6.2: PythonOperator (Solution)

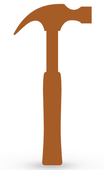
```python
import logging
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from library_extract import extract_table, fetch_html_content, save_as_csv

def extract_regions_dataset(url: str, output_file_path: str):
  try:
    html_content = fetch_html_content(url)
    table = extract_table(html_content)
    save_as_csv(table, output_file_path)
    logging.info("File was successfully saved as " + output_file_path)
  except Exception:
    logging.error("Failed to download file content.")
```

import DAG and PythonOperator

reuse already implemented code

create a simple Python function implementing our task, i.e., extracting the region dataset

Example 6.2: PythonOperator (Solution)



```
16 default_args = {
17   "owner": "koupil",
18   "retries": 3,
19   "retry_delay": timedelta(minutes=5),
20 }
21
22
23 with DAG(
24   dag_id="dag_python_operator",
25   default_args=default_args,
26   start_date=datetime(2024, 3, 22),
27   schedule_interval="@daily",
28 ) as dag:
29   extract_regions_task = PythonOperator(
30     task_id="extract_regions_task",
31     python_callable=extract_regions_dataset,
32     op_kwargs={
33       "url": "https://cs.wikipedia.org/wiki/Kraje_v_Česku",
34       "output_file_path": "dataset_regions.csv",
35     },
36   )
37
38 extract_regions_task
```

create an instance of DAG

our task is an instance of PythonOperator

each task has a task_id

python_callable determines the Python function to be executed

passing parameters using the op_kwargs dictionary

assembling the DAG

Exercise 6.3: Data Sharing via Airflow Xcoms

- Create the following *Apache Airflow Workflow*:
  - *Extract the table* 'Základní data o krajích' (Basic data about regions) *from* the *Wikipedia article* about Czech regions
  - *Transform* the input dataset into a dataset corresponding to the dim_regions dimension

- *Each of the tasks* should be implemented as a *separate* PythonOperator
  - Reuse the existing solution from Example 2.3 (extract) and 3.2 (transform)

- **Tip:** To share data between tasks, use *XComs*
  - Note that *maximum size* of Xcoms is *48 kB* (i.e., do not share large data directly using XComs)
  - Documentation: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/xcoms.html

Example 6.4: Connection to PostgreSQL and PostgreOperator

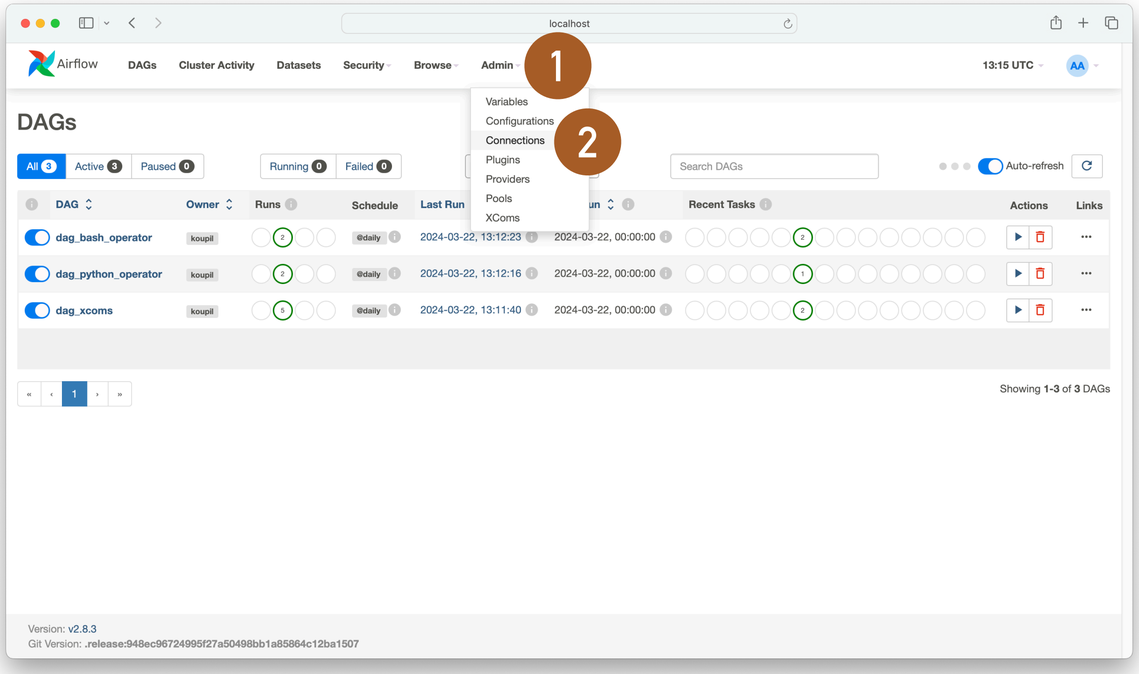
- Create the following *Apache Airflow Workflow*:
  - *Extract the table* 'Základní data o krajích' (Basic data about regions) *from* the *Wikipedia article* about Czech regions
  - *Transform* the input dataset into a dataset corresponding to the dim_regions dimension
  - *Load* dim_regions dataset into the dim_regions table in PostgreSQL

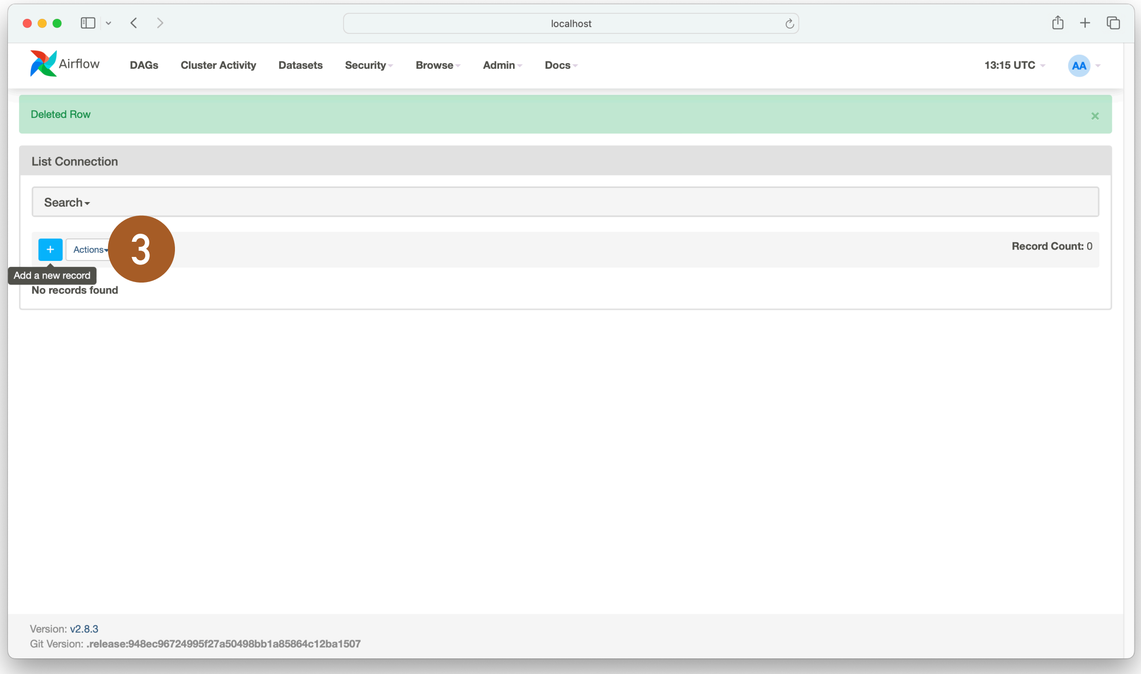
- Create a *new Connection* to access the PostgreSQL database system
- Use PythonOperator for extraction and transformation
- Use PostgreOperator to interact with PostgreSQL
  - You may implement a *custom operator* for bulk loading
- Reuse the existing solution from Examples 2.3 (extract), 3.2 (transform), and 4.3 (bulk loading)

Example 6.4: Connection to PostgreSQL and PostgreOperator (Solution)

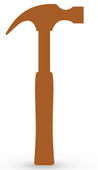
- Typically, when creating an ETL DAG, we need to connect to some external services
  - We can create and manage them using *Airflow Connections*
    - In the Airflow web server user interface, go to Admin → Connections 1 2

- Add a *new record* 3
  - A form will appear in which you fill in the Connection Id, Connection Type **select** 'Postgres', Host, Database, Login, Password, Port
    - Confirm the data by pressing the *Save* button





Example 6.4: Connection to PostgreSQL and PostgreOperator (Solution)



implement a custom operator for bulk loading

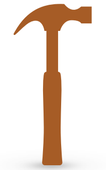
set required parameters, i.e., table_name and file_path, as template fields

template fields must be passed in the constructor

PostgresHook allows execution of statements in PostgreSQL

```python
class PostgresBulkLoadOperator(BaseOperator):
  template_fields = ("table_name", "file_path")

  @apply_defaults
  def __init__(self, *, postgres_conn_id: str, table_name: str, file_path: str, **kwargs):
    super().__init__(**kwargs)
    self.postgres_conn_id = postgres_conn_id
    self.table_name = table_name
    self.file_path = file_path

  def execute(self, context):
    try:
      hook = PostgresHook(postgres_conn_id=self.postgres_conn_id)
      with open(self.file_path, "r") as f:
        columns = f.readline().strip().split(",")
        copy_sql = f"COPY {self.table_name} ({', '.join(columns)}) FROM STDIN WITH CSV HEADER"
        hook.copy_expert(copy_sql, f.name)
    except FileNotFoundError:
      logging.error(f"File '{self.file_path}' not found.")
    except Exception as ex:
      logging.error(f"An error occurred while reading data from file: {ex}")
```

Example 6.4: Connection to PostgreSQL and PostgreOperator (Solution)



```
drop_table_task = PostgresOperator(
  task_id="drop_table_task",
  sql=DimRegionsQueries.drop_table_query,
  postgres_conn_id="postgres_webik",
)

create_table_task = PostgresOperator(
  task_id="create_table_task",
  sql=DimRegionsQueries.create_table_query,
  postgres_conn_id="postgres_webik",
)

alter_table_task = PostgresOperator(
  task_id="alter_table_task",
  sql=DimRegionsQueries.alter_table_query,
  postgres_conn_id="postgres_webik",
)

insert_data_task = PostgresBulkLoadOperator(
  task_id="insert_data_task",
  postgres_conn_id="postgres_webik",
  table_name="dim_regions",
  file_path="{{ ti.xcom_pull(task_ids='transform_regions_task', key='dim_regions') }}",
)
```

PostgreOperator **task implementing drop table**

PostgreOperator **implementing create table**

PostgreOperator **task implementing alter table**

custom operator task implementing bulk loading

passing the argument from xcom

Apache Airflow: TaskFlow API Decorators

- @dag() creates a DAG
- @task() creates a Python Task
- @task_group() creates a TaskGroup
- @task.sensor() changes Python function into a Sensor
- @task.docker() creates a DockerOperator task
- @task.branch() creates a branch in DAG based on evaluated condition
- @task.short_circuit() evaluates a condition and skips downstream tasks if the condition is False
- @task.virtualenv() runs Python task in a virtual environment

- It is also possible to add custom decorator to the TaskFlow API
  - see https://airflow.apache.org/docs/apache-airflow/stable/howto/create-custom-decorator.html

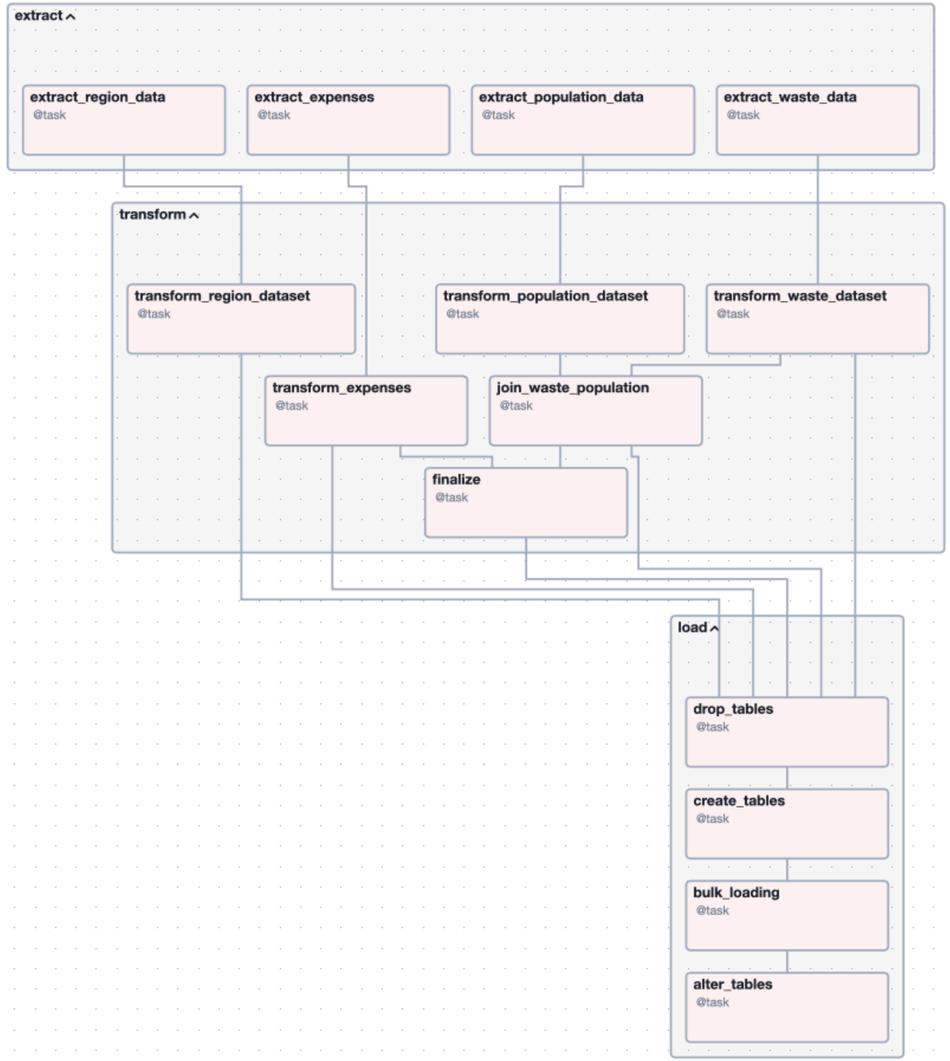
Exercise 6.5: TaskFlow API

- Create the following *Apache Airflow Workflow*:
  - *Extract the table* 'Základní data o krajích' (Basic data about regions) *from* the *Wikipedia article* about Czech regions
  - *Transform* the input dataset into a dataset corresponding to the dim_regions dimension
  - *Load* dim_regions dataset into the dim_regions table in PostgreSQL

- This time *use* the *TaskFlow API*

Exercise 6.6: Complete ETL Workflow in Apache Airflow

- Create an *Apache Airflow workflow* that implements the following ETL:
  - *Extract datasets*:
    - Production of industrial and municipal waste (see Example 2.1)
    - Costs of environmental protection (see Exercise 2.2)
    - Regions dataset (see Exercise 2.3)
    - Population dataset (see Example 2.4)
  - *Transform the input datasets* according to the data transformation workflow (see Example 3.1)
  - Finally, *perform bulk loading* of the transformed datasets into the corresponding tables in PostgreSQL (see Exercise 4.4)



- Use existing solutions from previous practical classes
- Use *Operators*, *TaskFlow API* or a suitable combination of both approaches
- You can implement *Sensor* that detect if a particular dataset is available
- *Schedule* an execution workflow *once a day*

CRON expression

- The CRON expression is a string of five fields separated by a white space that represents a set of times
  - Airflow already provides some presets for the schedule_interval (see Table)

preset	meaning	cron
None	Do not schedule, use exclusively "externally triggered" DAGs	
@once	Schedule once and only once	
@continuous	Run as soon as the previous run finishes	
@hourly	Run once an hour at the beginning of the hour	0 * * * *
@daily	Run once a day at midnight	0 0 * * *
@weekly	Run once a day at midnight on Sunday morning	0 0 * * 0
@monthly	Run once a month at midnight of the first day of the month	0 0 1 * *
@quarterly	Run once a quarter at midnight on the first day	0 0 1 */3 *
@yearly	Run once a year at midnight of January 1	0 0 1 1 *

source: https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/cron.html

Exercise 6.7: Scheduling using the cron expression

- Adjust the *Apache Airflow workflow* that implements ETL (see Exercise 6.6) so that it executes once a month
  - Use *CRON expression* for scheduling
  - Instead of overwriting the database, just *update the data* valid for the current year in which the task instance is executed

References

Apache Airflow

- Running Airflow in Docker: https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html#running-airflow-in-docker
- Core Concepts: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/index.html
- DAGs: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html#
- Tasks: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/tasks.html#
- TaskFlow: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/taskflow.html#
- XComs: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/xcoms.html

Docker

- Docker: https://www.docker.com/
- Docker Docs: https://docs.docker.com/

CRON

- Python-crontab: https://pypi.org/project/python-crontab/
- Cron & Time Intervals (Airflow): https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/cron.html
- Editor from cron schedule expressions: https://crontab.guru/