1: Project Introduction

Link:

Notes:

**Project: Data Pipelines with Airflow**

A music streaming company, Sparkify, has decided that it is time to introduce more automation and monitoring to their data warehouse ETL pipelines and come to the conclusion that the best tool to achieve this is Apache Airflow.

They have decided to bring you into the project and expect you to create high grade data pipelines that are dynamic and built from reusable tasks, can be monitored, and allow easy backfills. They have also noted that the data quality plays a big part when analyses are executed on top the data warehouse and want to run tests against their datasets after the ETL steps have been executed to catch any discrepancies in the datasets.

The source data resides in S3 and needs to be processed in Sparkify's data warehouse in Amazon Redshift. The source datasets consist of JSON logs that tell about user activity in the application and JSON metadata about the songs the users listen to.

2: Project Overview

Link:

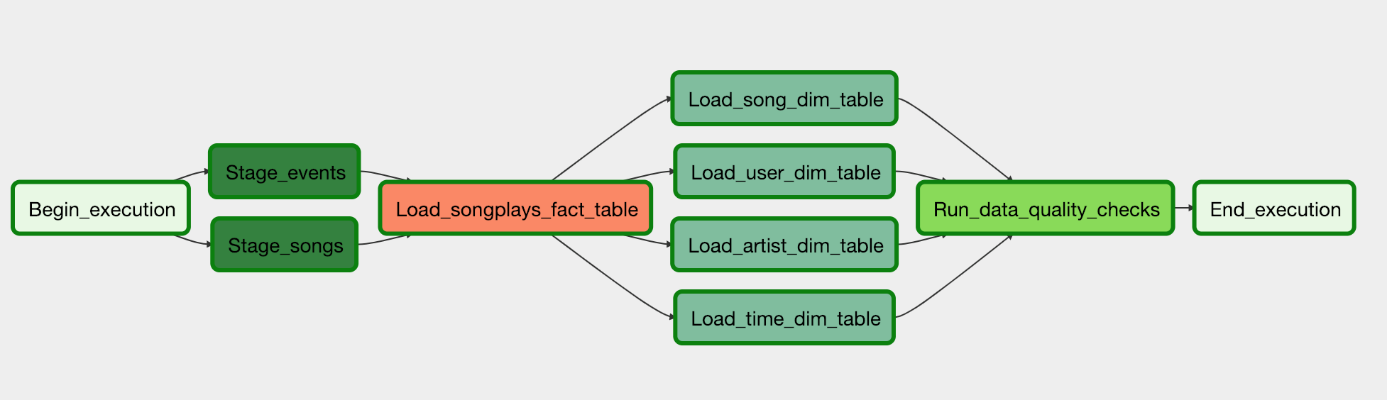
Notes:

**Project Overview**

This project will introduce you to the core concepts of Apache Airflow. To complete the project, you will need to create your own custom operators to perform tasks such as staging the data, filling the data warehouse, and running checks on the data as the final step.

We have provided you with a project template that takes care of all the imports and provides four empty operators that need to be implemented into functional pieces of a data pipeline. The template also contains a set of tasks that need to be linked to achieve a coherent and sensible data flow within the pipeline.

You'll be provided with a helpers class that contains all the SQL transformations. Thus, you won't need to write the ETL yourselves, but you'll need to execute it with your custom operators.



Example DAG

3: Add Airflow Connections to AWS

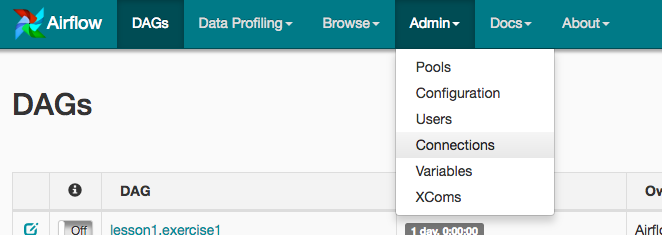
Link:

Notes:

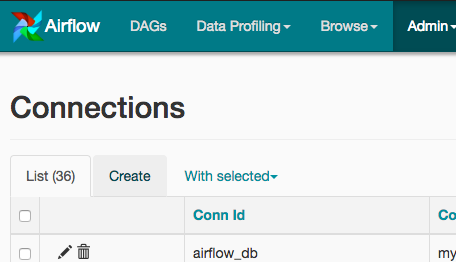
# Add Airflow Connections

Here, we'll use Airflow's UI to configure your AWS credentials and connection to Redshift.

1. To go to the Airflow UI:
   * You can use the Project Workspace here and click on the blue **Access Airflow** button in the bottom right.
   * If you'd prefer to run Airflow locally, open [**http://localhost:8080**](http://localhost:8080/) in Google Chrome (other browsers occasionally have issues rendering the Airflow UI).
2. Click on the **Admin** tab and select **Connections**.

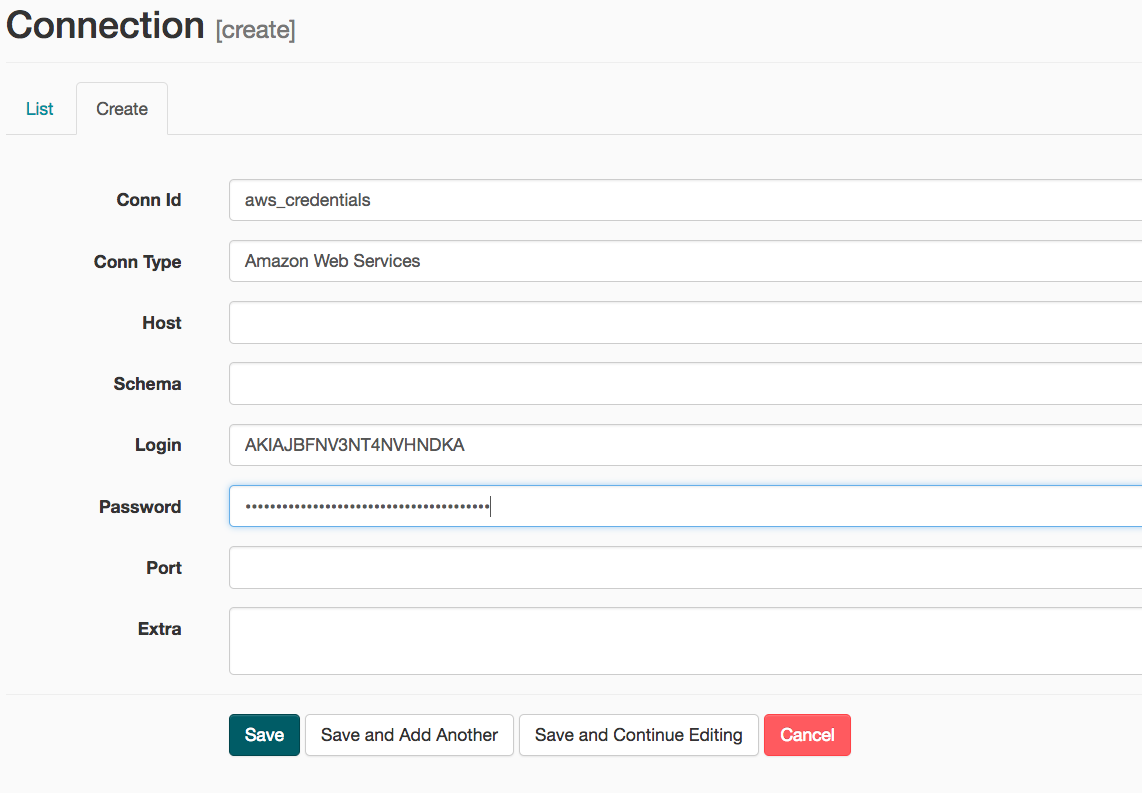


1. Under **Connections**, select **Create**.



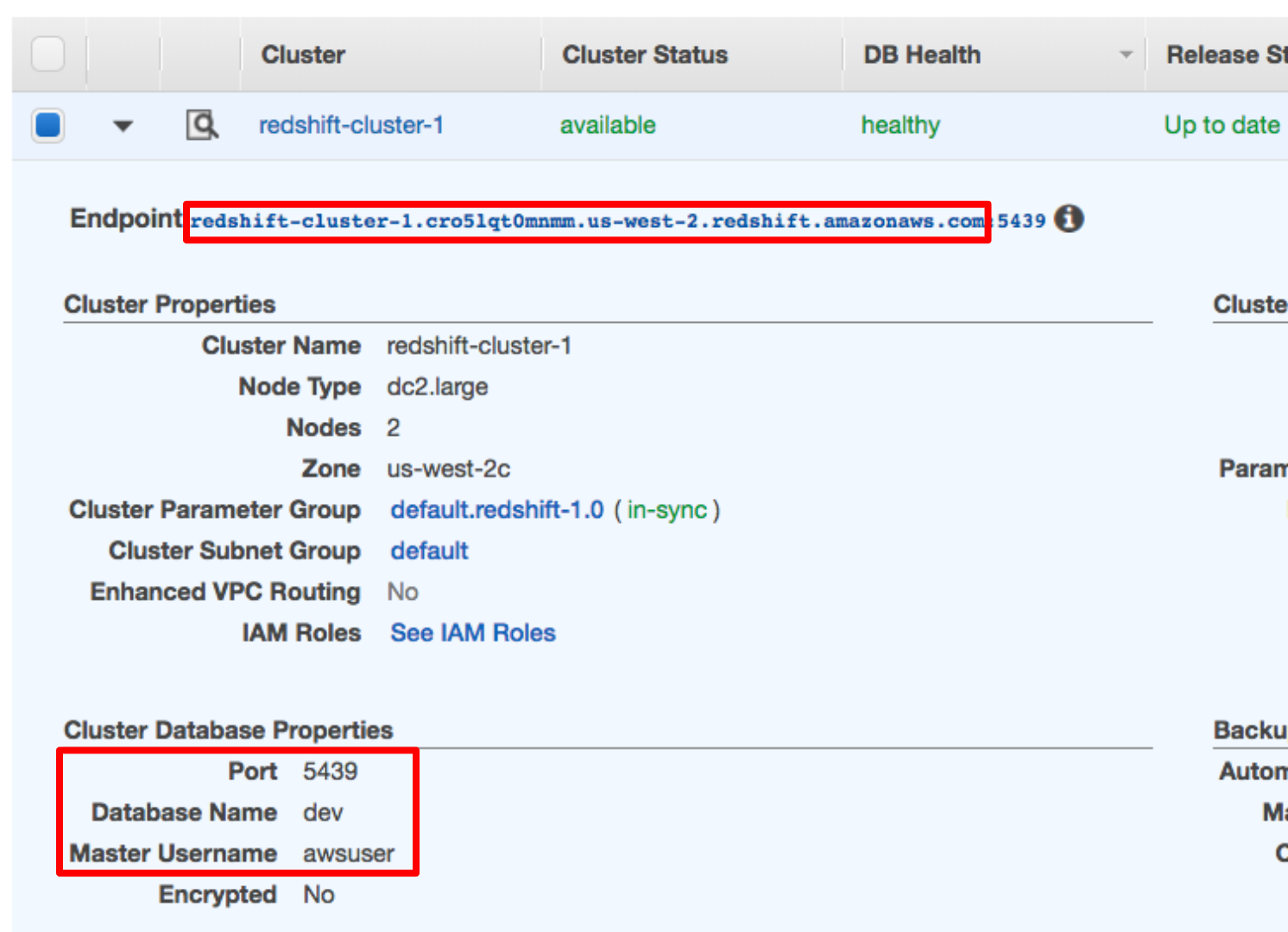
1. On the create connection page, enter the following values:
   * **Conn Id**: Enter aws\_credentials.
   * **Conn Type**: Enter Amazon Web Services.
   * **Login**: Enter your **Access key ID** from the IAM User credentials you downloaded earlier.
   * **Password**: Enter your **Secret access key** from the IAM User credentials you downloaded earlier.

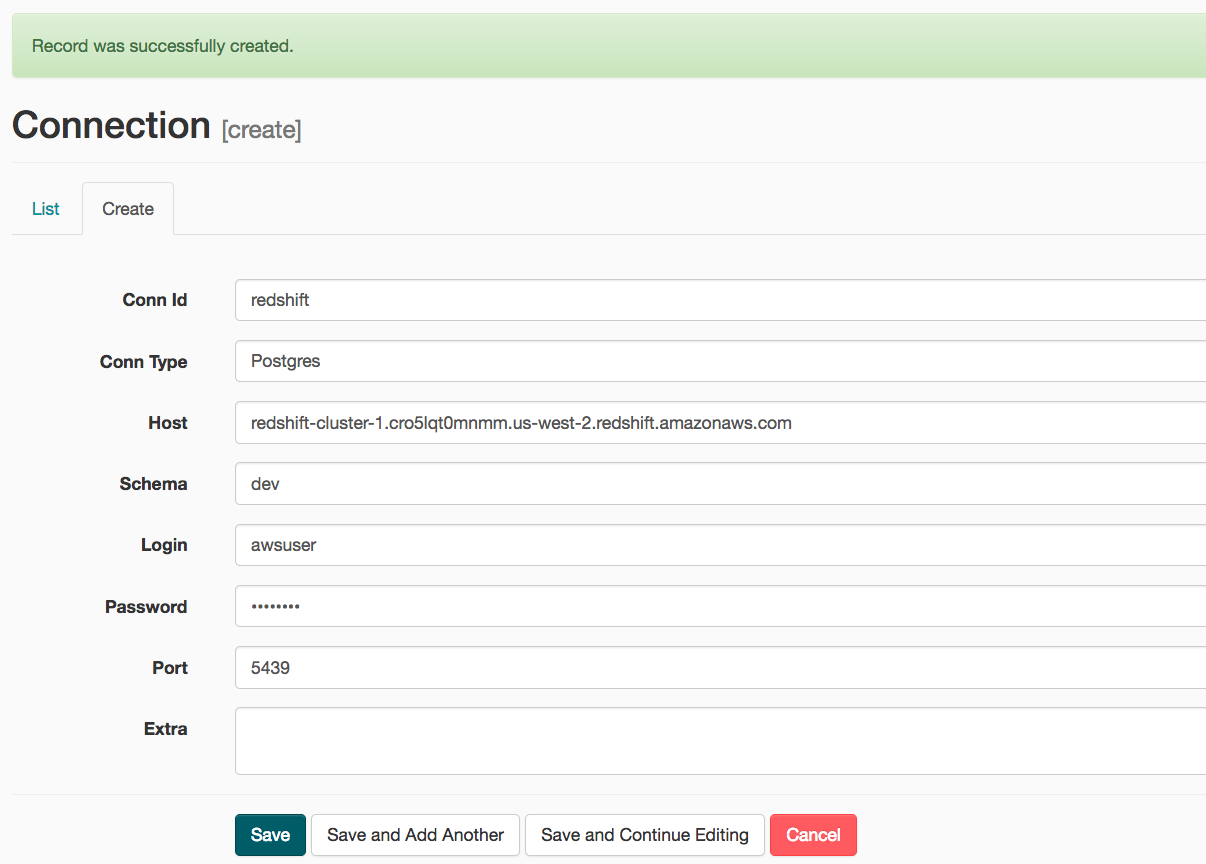
Once you've entered these values, select **Save and Add Another**.



1. On the next create connection page, enter the following values:
   * **Conn Id**: Enter redshift.
   * **Conn Type**: Enter Postgres.
   * **Host**: Enter the endpoint of your Redshift cluster, excluding the port at the end. You can find this by selecting your cluster in the **Clusters** page of the Amazon Redshift console. See where this is located in the screenshot below. IMPORTANT: Make sure to **NOT** include the port at the end of the Redshift endpoint string.
   * **Schema**: Enter dev. This is the Redshift database you want to connect to.
   * **Login**: Enter awsuser.
   * **Password**: Enter the password you created when launching your Redshift cluster.
   * **Port**: Enter 5439.

Once you've entered these values, select **Save**.





Awesome! You're now all configured to run Airflow with Redshift.

### WARNING: Remember to DELETE your cluster each time you are finished working to avoid large, unexpected costs.

4: Project Instructions

Link:

Notes:

# Datasets

For this project, you'll be working with two datasets. Here are the s3 links for each:

* Log data: s3://udacity-dend/log\_data
* Song data: s3://udacity-dend/song\_data

# Project Template

To get started with the project:

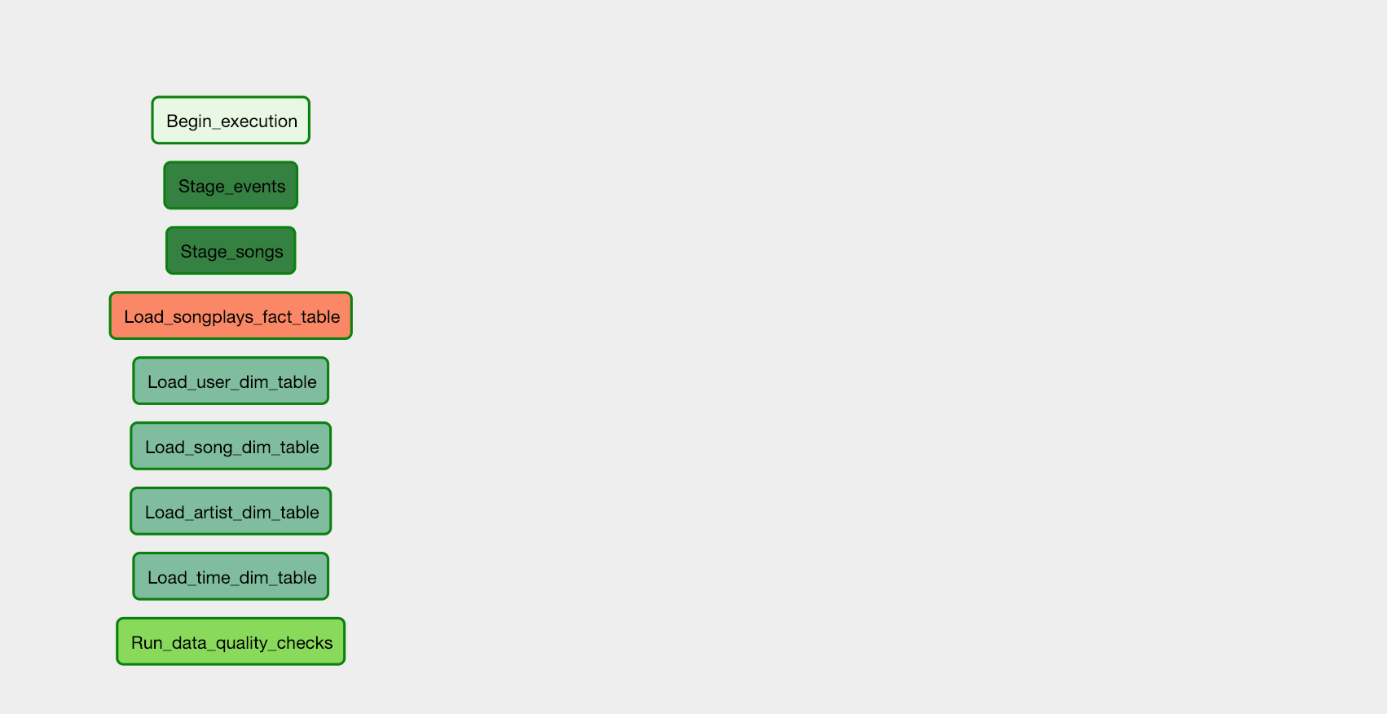
1. Go to the workspace on the next page, where you'll find the project template. You can work on your project and submit your work through this workspace.

Alternatively, you can download the [**project template package**](https://s3.amazonaws.com/video.udacity-data.com/topher/2019/February/5c6058dc_project-template/project-template.zip) and put the contents of the package in their respective folders in your local Airflow installation.

The project template package contains three major components for the project:

* + The **dag template** has all the imports and task templates in place, but the task dependencies have not been set
  + The **operators** folder with operator templates
  + A **helper class** for the SQL transformations

1. With these template files, you should be able see the new DAG in the Airflow UI. The graph view should look like this:



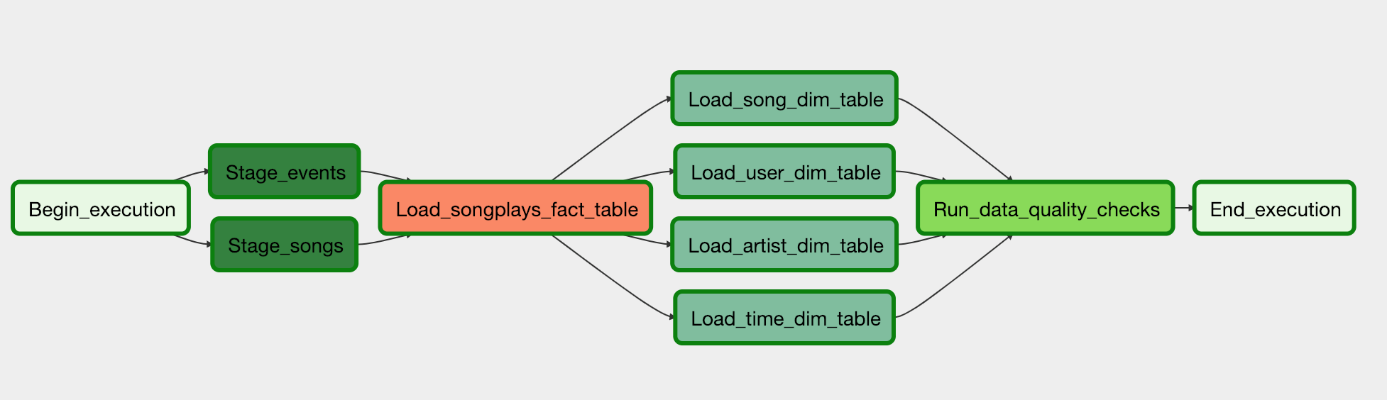
You should be able to execute the DAG successfully, but if you check the logs, you will see only operator not implemented messages.

## Configuring the DAG

In the DAG, add default parameters according to these guidelines

* The DAG does not have dependencies on past runs
* On failure, the task are retried 3 times
* Retries happen every 5 minutes
* Catchup is turned off
* Do not email on retry

In addition, configure the task dependencies so that after the dependencies are set, the graph view follows the flow shown in the image below.



Working DAG with correct task dependencies

# Building the operators

To complete the project, you need to build four different operators that will stage the data, transform the data, and run checks on data quality.

You can reuse the code from Project 2, but remember to utilize Airflow's built-in functionalities as connections and hooks as much as possible and let Airflow do all the heavy-lifting when it is possible.

All of the operators and task instances will run SQL statements against the Redshift database. However, using parameters wisely will allow you to build flexible, reusable, and configurable operators you can later apply to many kinds of data pipelines with Redshift and with other databases.

### Stage Operator

The stage operator is expected to be able to load any JSON formatted files from S3 to Amazon Redshift. The operator creates and runs a SQL COPY statement based on the parameters provided. The operator's parameters should specify where in S3 the file is loaded and what is the target table.

The parameters should be used to distinguish between JSON file. Another important requirement of the stage operator is containing a templated field that allows it to load timestamped files from S3 based on the execution time and run backfills.

### Fact and Dimension Operators

With dimension and fact operators, you can utilize the provided SQL helper class to run data transformations. Most of the logic is within the SQL transformations and the operator is expected to take as input a SQL statement and target database on which to run the query against. You can also define a target table that will contain the results of the transformation.

Dimension loads are often done with the truncate-insert pattern where the target table is emptied before the load. Thus, you could also have a parameter that allows switching between insert modes when loading dimensions. Fact tables are usually so massive that they should only allow append type functionality.

### Data Quality Operator

The final operator to create is the data quality operator, which is used to run checks on the data itself. The operator's main functionality is to receive one or more SQL based test cases along with the expected results and execute the tests. For each the test, the test result and expected result needs to be checked and if there is no match, the operator should raise an exception and the task should retry and fail eventually.

For example one test could be a SQL statement that checks if certain column contains NULL values by counting all the rows that have NULL in the column. We do not want to have any NULLs so expected result would be 0 and the test would compare the SQL statement's outcome to the expected result.

### Note about Workspace

After you have updated the DAG, you will need to run /opt/airflow/start.sh command to start the Airflow web server. Once the Airflow web server is ready, you can access the Airflow UI by clicking on the blue Access Airflow button.

5: Workspace instructions

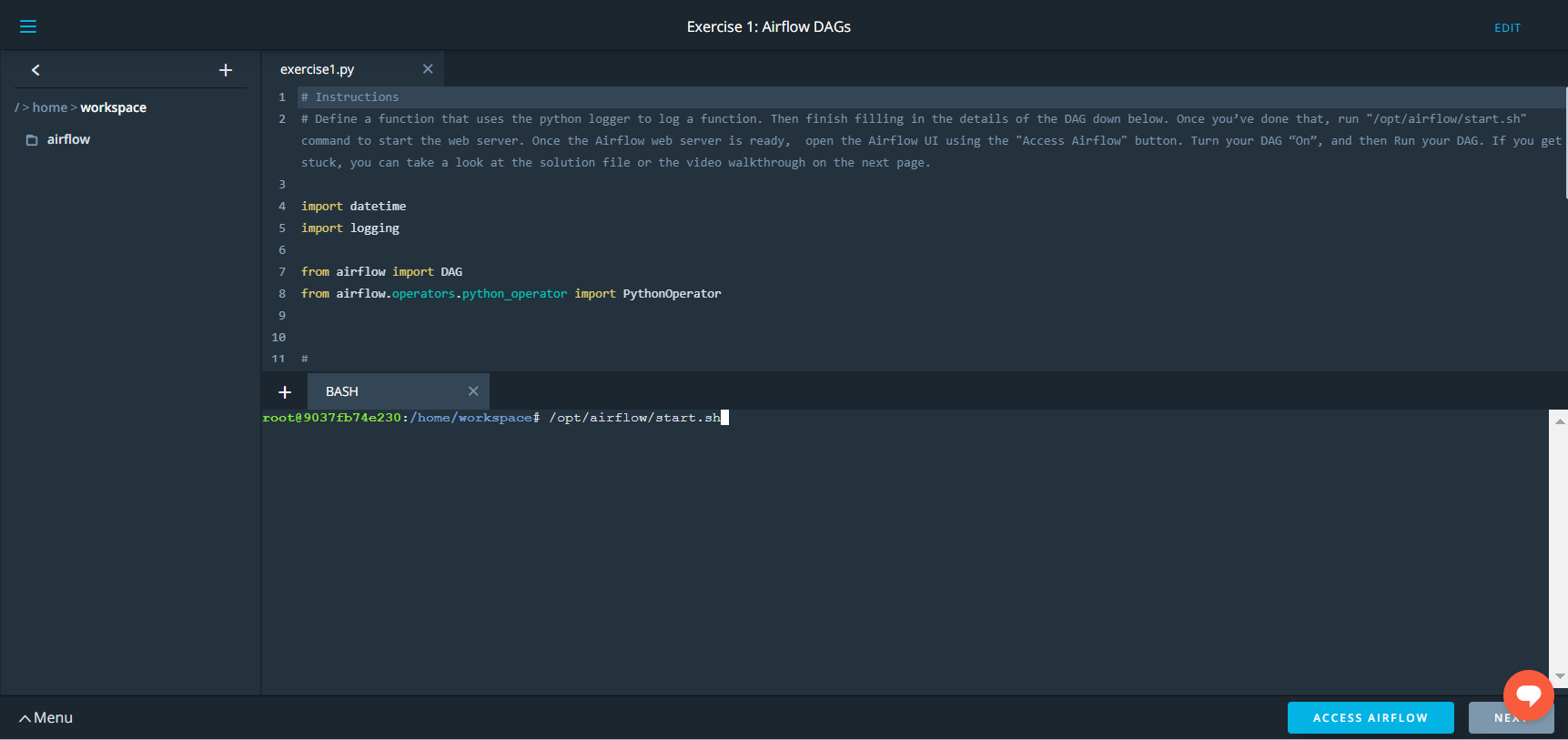
Link:

Notes:

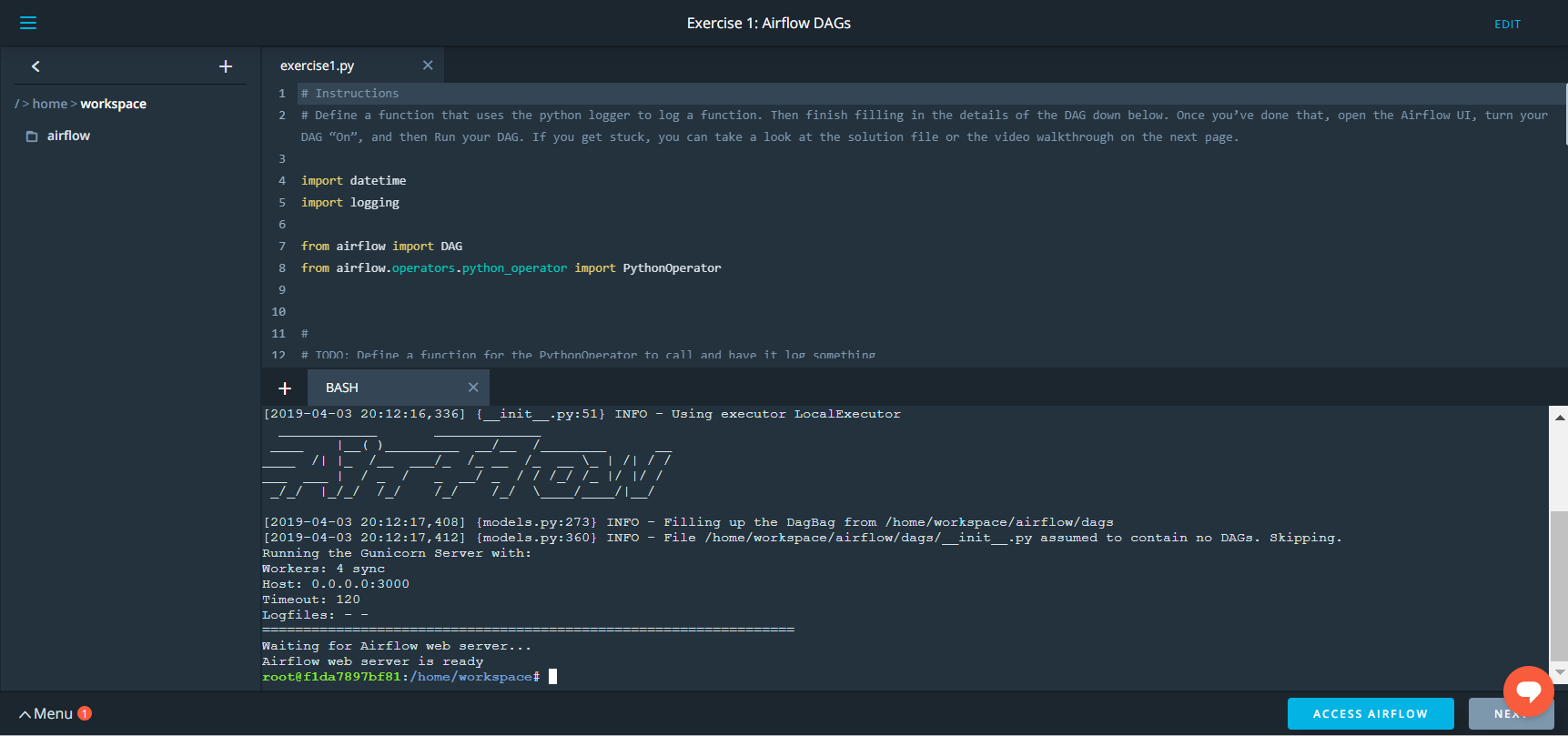
## Workspace Instructions

Before you start on your first exercise, please note the following instruction.

1. After you have updated the DAG, you will need to run /opt/airflow/start.sh command to start the Airflow webserver. See the screenshot below for the Exercise 1 Workspace.



1. Wait for the Airflow web server to be ready (see screenshot below).



1. Access the Airflow UI by clicking on the blue "Access Airflow" button.

This should be able to access the Airflow UI without any delay.

**Please note:** Because the files located in the s3 bucket 'udacity-dend' are very large, Airflow can take up to 10 minutes to make the connection.

6: Project Workspace

Link:

Notes:

Helps:

"""

COPY {vault}.{table}

FROM {bucket}

CREDENTIALS {role\_arn}

REGION {region}

COMPUPDATE ON

FORMAT AS JSON {jsonpaths} -- or 'auto'

EMPTYASNULL

BLANKSASNULL;

"""

staging\_events\_copy = (""" copy staging\_events from {} iam\_role {} compupdate off region 'us-west-2' json {} """).format(config.get('S3','LOG\_DATA'), config.get('IAM\_ROLE','ARN'), config.get('S3','LOG\_JSONPATH'))

7: Project: Data Pipelines

Link:

Notes:

## **Project Submission**

**Have project questions?** Ask a technical mentor or search for existing answers!

ASK A MENTOR

###### DUE DATE

**Dec 10**

###### STATUS

**Unsubmitted**

Project past due

Your **deliverables** will be a zip file or **Github repo** containing the files for your project. Alternatively, you can submit your project through the workspace.

Ensure you meet specifications for all items in the [**Project Rubric**](https://review.udacity.com/#!/rubrics/2478/view). Your project must **Meet Specifications** in each category in order for your submission to pass.

It can take us up to a week to grade the project, but in most cases it is much faster. You will get an email once your submission has been reviewed. In the meantime, you should feel free to proceed with your learning journey by continuing on to the next module in the program.

## PROJECT SPECIFICATION

**Data Pipelines with Airflow**

General

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| The dag and plugins do not give an error when imported to Airflow | DAG can be browsed without issues in the Airflow UI |
| All tasks have correct dependencies | The dag follows the data flow provided in the instructions,  all the tasks have a dependency and DAG begins with  a start\_execution task and ends with a end\_execution task. |

Dag configuration

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Default\_args object is used in the DAG | DAG contains default\_args dict, with the  following keys:   * Owner * Depends\_on\_past * Start\_date * Retries * Retry\_delay * Catchup |
| Defaults\_args are bind to the DAG | The DAG object has default args set |
| The DAG has a correct schedule | The DAG should be scheduled to run once an hour |

Staging the data

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Task to stage JSON data is included in the DAG and uses the RedshiftStage operator | There is a task that to stages data from S3 to  Redshift. (Runs a Redshift copy statement) |
| Task uses params | Instead of running a static SQL statement to  stage the data, the task uses params to  generate the copy statement dynamically |
| Logging used | The operator contains logging in different steps  of the execution |
| The database connection is created by using a hook and a connection | The SQL statements are executed by using  a Airflow hook |

Loading dimensions and facts

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Set of tasks using the dimension load operator is in the DAG | Dimensions are loaded with on the LoadDimension  operator |
| A task using the fact load operator is in the DAG | Facts are loaded with on the LoadFact operator |
| Both operators use params | Instead of running a static SQL statement to stage  the data, the task uses params to generate the copy  statement dynamically |
| The dimension task contains a param to allow switch between append and insert-delete functionality | The DAG allows to switch between append-only  and delete-load functionality |

Data Quality Checks

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| A task using the data quality operator is in the DAG and at least one data quality check is done | Data quality check is done with correct operator |
| The operator raises an error if the check fails pass | The DAG either fails or retries n times |
| The operator is parametrized | Operator uses params to get the tests and the results,  tests are not hard coded to the operator |

## Suggestions to Make Your Project Stand Out!

* Simple and dynamic operators, as little hard coding as possible
* Effective use of parameters in tasks
* Clean formatting of values in SQL strings
* Load dimensions with a subdag

# Data Pipelines with Airflow

* [**REVIEW**](https://review.udacity.com/)
* [**CODE REVIEW**](https://review.udacity.com/)
* [**HISTORY**](https://review.udacity.com/)

## Requires Changes

### **1 specification requires changes**

Good job on your first try. Your work is almost perfect! I like how you have managed to pass in a table name in the staging operation. This makes your DAG a bit more versatile in a situation where you need to add more tables.

To pass this project, you only need to improve the Quality check operator.

If you need more help with this project, do not hesitate to ask questions in the [Knowledge Hub](https://knowledge.udacity.com/).

Good luck with your next submission!

## General

**DAG can be browsed without issues in the Airflow UI**

**The dag follows the data flow provided in the instructions, all the tasks have a dependency and DAG begins with a start\_execution task and ends with a end\_execution task.**

The DAG's graph is correct and all tasks have dependencies. Good work. We expect the create\_tables task to run on subsequent DAG runs. To ensure that the DAG does not return an error from creating existing tables, you may do either of these methods in the table-creation process:

1. Use CREATE table IF NOT EXISTS
2. Add retries=0 in the operator's parameters (but this is against one of the specifications of this project)
3. Or, if you don't mind having an error in the table creation process, set depends\_on\_past to False so the next operator will still run.

## Dag configuration

**DAG contains default\_args dict, with the following keys:**

* **Owner**
* **Depends\_on\_past**
* **Start\_date**
* **Retries**
* **Retry\_delay**
* **Catchup**

Good job setting proper arguments for your DAG. I'd like to bring your attention to this very useful configuration called max\_active\_runs. Sometimes, you may want to have exactly one task per DAG that is actively running. Perhaps when there is only one cluster of resources to be shared between the tasks. Setting max\_active\_runs to 1 would fit this requirement.

This setting is very useful to ensure we avoid any racing condition as the [documentation](https://airflow.apache.org/docs/stable/_api/airflow/models/dag/index.html) states:

max\_active\_runs (int) – maximum number of active DAG runs, beyond this number of DAG runs in a running state, the scheduler won’t create new active DAG runs

Remember that this setting would still allow for multiple DAGs, the only constraint here is one run per DAG.

**The DAG object has default args set**

**The DAG should be scheduled to run once an hour**

Good work with the DAG scheduling.

I see that you are using a cron expression 0 \*/1 \* \* \* in your DAG code. With cron expressions, you may write even more complicated scheduling rules.

You may learn the full extent of cron expressions in this [CronTrigger Tutorial](http://www.quartz-scheduler.org/documentation/quartz-2.3.0/tutorials/crontrigger.html).

## Staging the data

**There is a task that to stages data from S3 to Redshift. (Runs a Redshift copy statement)**

The staging step has been done properly, I like how concise your code is structured for this feature.

When running the COPY command, it is recommended to include the region setting as such:

COPY {}

...

REGION 'us-west-2'

...

As explained in the [documentation for COPY from Amazon S3](https://docs.aws.amazon.com/redshift/latest/dg/copy-parameters-data-source-s3.html), REGION setting is needed if the Amazon S3 buckets that hold the data files don't reside in the same AWS Region as your cluster, as is often the case for many ETL projects.

**Instead of running a static SQL statement to stage the data, the task uses params to generate the copy statement dynamically**

Good job setting the COPY statement to accept parameters.

**The operator contains logging in different steps of the execution**

Logging has been implemented, well done.

**The SQL statements are executed by using a Airflow hook**

Good job using Airflow hooks in your project to load the SQL statements.

## Loading dimensions and facts

**Dimensions are loaded with on the LoadDimension operator**

**Facts are loaded with on the LoadFact operator**

**Instead of running a static SQL statement to stage the data, the task uses params to generate the copy statement dynamically**

**The DAG allows to switch between append-only and delete-load functionality**

The option to switch between append-only and delete-load functionality has been included through the parameter clear\_content in your code.

## Data Quality Checks

**Data quality check is done with correct operator**

Good job performing data quality checks. Quality checking may seem like a trivial task, but doing this would save you from headaches later on. In my case, I often thought I have transformed the data properly, only to find that some observations were missing due to errors in the ETL process. To work around this, I get into the habit of thinking about the number of observations before and after the ETL process and use the conclusion to create a quality check.

**The DAG either fails or retries n times**

**Operator uses params to get the tests and the results, tests are not hard coded to the operator**

This specification requires that **"Operator uses params to get the tests and the results"**. You have done well passing the **tests** part into a parameter, but the **results** is not currently a part of the parameter.

You may pass, for example, a list of dict objects with both the SQL queries to check and their expected results, like so:

checks = [

{'test\_sql': ..., 'expected\_result': ...},

{'test\_sql': ..., 'expected\_result': ...}

]

(Optional) You can be creative in the formatting of your parameters. For example, you may include a **comparison operator** to the dict object to allow more flexibility e.g.

{'test\_sql': "SELECT COUNT(\*) FROM songs", 'expected\_result': 0, comparison: '>'}

Which could be translated as "The songs table should not be empty".