Production Deployment

It is time to deploy your DAG in production. To do this, first, you need to make sure that the Airflow is itself production-ready. Let's see what precautions you need to take.

Database backend

Airflow comes with an `SQLite` backend by default. This allows the user to run Airflow without any external database. However, such a setup is meant to be used for testing purposes only; running the default setup in production can lead to data loss in multiple scenarios. If you want to run production-grade Airflow, make sure you configure the backend to be an external database such as PostgreSQL or MySQL.

You can change the backend using the following config

```
[core]
sql_alchemy_conn = my_conn_string
```

Once you have changed the backend, airflow needs to create all the tables required for operation. Create an empty DB and give airflow's user the permission to `CREATE/ALTER` it. Once that is done, you can run -

```
airflow db upgrade
```

`upgrade` keeps track of migrations already applied, so it's safe to run as often as you need.

Note

Do not use `airflow db init` as it can create a lot of default connections, charts, etc. which are not required in production DB.

Multi-Node Cluster

Airflow uses **`SequentialExecutor`** by default. However, by it nature, the user is limited to executing at most one task at a time. `Sequential Executor` also pauses the scheduler when it runs a task, hence not recommended in a production setup. You should use the **`LocalExecutor`** for a single machine. For a multi-node setup, you should use the Kubernetes executor or the Celery executor.

Once you have configured the executor, it is necessary to make sure that every node in the cluster contains the same configuration and dags. Airflow sends simple instructions such as "execute task X of dag Y", but does not send any dag files or configuration. You can use a simple cronjob or any other mechanism to sync DAGs and configs across your nodes, e.g., checkout DAGs from git repo every 5 minutes on all nodes.

Logging

If you are using disposable nodes in your cluster, configure the log storage to be a distributed file system (DFS) such as `S3` and `GCS`, or external services such as Stackdriver Logging, Elasticsearch or Amazon CloudWatch. This way, the logs are available even after the node goes down or gets replaced. See Logging for Tasks for configurations.

Note

The logs only appear in your DFS after the task has finished. You can view the logs while the task is running in UI itself.

Configuration

Airflow comes bundled with a default `airflow.cfg` configuration file. You should use environment variables for configurations that change across deployments e.g. metadata DB, password, etc. You can accomplish this using the format **`AIRFLOW__{SECTION}__{KEY}`**

```
AIRFLOW__CORE__SQL_ALCHEMY_CONN=my_conn_id
AIRFLOW__WEBSERVER__BASE_URL=http://host:port
```

Some configurations such as the Airflow Backend connection URI can be derived from bash commands as well:

```
sql_alchemy_conn_cmd = bash_command_to_run
```

Scheduler Uptime

Airflow users occasionally report instances of the scheduler hanging without a trace, for example in these issues:

- Scheduler gets stuck without a trace
- Scheduler stopping frequently

Strategies for mitigation:

- When running on kubernetes, use a `livenessProbe` on the scheduler deployment to fail if the scheduler has not heartbeat in a while. Example:.

Production Container Images

Production-ready reference Image

For the ease of deployment in production, the community releases a production-ready reference container image.

The docker image provided (as convenience binary package) in the Apache Airflow DockerHub is a bare image that has a few external dependencies and extras installed..

The Apache Airflow image provided as convenience package is optimized for size, so it provides just a bare minimal set of the extras and dependencies installed and in most cases you want to either extend or customize the image. You can see all possible extras in Reference for package extras. The set of extras used in Airflow Production image are available in the Dockerfile.

The production images are build in DockerHub from released version and release candidates. There are also images published from branches but they are used mainly for development and testing purpose. See Airflow Git Branching for details.

Customizing or extending the Production Image

Before you dive-deeply in the way how the Airflow Image is build, named and why we are doing it the way we do, you might want to know very quickly how you can extend or customize the existing image for Apache Airflow. This chapter gives you a short answer to those questions.

Airflow Summit 2020's Production Docker Image talk provides more details about the context, architecture and customization/extension methods for the Production Image.

Extending the image

Extending the image is easiest if you just need to add some dependencies that do not require compiling. The compilation framework of Linux (so called `build-essential`) is pretty big, and for the production images, size is really important factor to optimize for, so our Production Image does not contain `build-essential`. If you need compiler like gcc or g++ or make/cmake etc. - those are not found in the image and it is recommended that you follow the "customize" route instead.

How to extend the image - it is something you are most likely familiar with - simply build a new image using Dockerfile's `FROM` directive and add whatever you need. Then you can add your Debian dependencies with `apt` or PyPI dependencies with `pip install` or any other stuff you need.

You should be aware, about a few things:

- The production image of airflow uses "airflow" user, so if you want to add some of the tools as `root` user, you need to switch to it with `USER` directive of the Dockerfile. Also you should remember about following the best practises of Dockerfiles to make sure your image is lean and small.

```
```

```
FROM apache/airflow:2.0.0
USER root
RUN apt-get update \
  && apt-get install -y --no-install-recommends \
         my-awesome-apt-dependency-to-add \
  && apt-get autoremove -yqq --purge \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/*
USER airflow
```

- PyPI dependencies in Apache Airflow are installed in the user library, of the "airflow" user, so you need to install them with the `--user` flag and WITHOUT switching to airflow user. Note also that using –no-cache-dir is a good idea that can help to make your image smaller.

```
FROM apache/airflow:2.0.0
RUN pip install --no-cache-dir --user my-awesome-pip-dependency-to-add
```

- If your apt, or PyPI dependencies require some of the build-essentials, then your best choice is to follow the "Customize the image" route. However it requires to checkout sources of Apache Airflow, so you might still want to choose to add build essentials to your image, even if your image will be significantly bigger.

```
FROM apache/airflow:2.0.0
USER root
RUN apt-get update \
  && apt-get install -y --no-install-recommends \
         build-essential my-awesome-apt-dependency-to-add \
  && apt-get autoremove -yqq --purge \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/*
USER airflow
RUN pip install --no-cache-dir --user my-awesome-pip-dependency-to-add
```

- You can also embed your dags in the image by simply adding them with COPY directive of Airflow. The DAGs in production image are in /opt/airflow/dags folder.

Customizing the image

Customizing the image is an alternative way of adding your own dependencies to the image - better suited to prepare optimized production images.

The advantage of this method is that it produces optimized image even if you need some compile-time dependencies that are not needed in the final image. You need to use Airflow Sources to build such images from the official distribution folder of Apache Airflow for the released versions, or checked out from the GitHub project if you happen to do it from git sources.

The easiest way to build the image image is to use `breeze` script, but you can also build such customized image by running appropriately crafted docker build in which you specify all the `build-args` that you need to add to customize it. You can read about all the args and ways you can build the image in the #production-image-build-arguments chapter below.

Here just a few examples are presented which should give you general understanding of what you can customize.

This builds the production image in version 3.7 with additional airflow extras from 2.0.0 PyPI package and additional apt dev and runtime dependencies.

```
docker build . \
  --build-arg PYTHON_BASE_IMAGE="python:3.7-slim-buster" \
  --build-arg PYTHON_MAJOR_MINOR_VERSION=3.7 \
  --build-arg AIRFLOW_INSTALLATION_METHOD="apache-airflow" \
  --build-arg AIRFLOW_VERSION="2.0.0" \
  --build-arg AIRFLOW_INSTALL_VERSION="==2.0.0" \
  --build-arg AIRFLOW_CONSTRAINTS_REFERENCE="constraints-2-0" \
  --build-arg AIRFLOW_SOURCES_FROM="empty" \
  --build-arg AIRFLOW_SOURCES_TO="/empty" \
  --build-arg ADDITIONAL_AIRFLOW_EXTRAS="jdbc" \
  --build-arg ADDITIONAL_PYTHON_DEPS="pandas" \
  --build-arg ADDITIONAL_DEV_APT_DEPS="gcc g++" \
  --build-arg ADDITIONAL_RUNTIME_APT_DEPS="default-jre-headless" \
  --tag my-image
```

the same image can be built using `breeze` (it supports auto-completion of the options):

```
./breeze build-image \
    --production-image  --python 3.7 --install-airflow-version=2.0.0 \
    --additional-extras=jdbc --additional-python-deps="pandas" \
    --additional-dev-apt-deps="gcc g++" --additional-runtime-apt-deps="default-jre-headless"
```

You can customize more aspects of the image - such as additional commands executed before apt dependencies are installed, or adding extra sources to install your dependencies from. You can see all the arguments described below but here is an example of rather complex command to customize the image based on example in this comment:

```
docker build . -f Dockerfile \
  --build-arg PYTHON_BASE_IMAGE="python:3.7-slim-buster" \
  --build-arg PYTHON_MAJOR_MINOR_VERSION=3.7 \
  --build-arg AIRFLOW_INSTALLATION_METHOD="apache-airflow" \
  --build-arg AIRFLOW_VERSION="2.0.0" \
  --build-arg AIRFLOW_INSTALL_VERSION="==2.0.0" \
  --build-arg AIRFLOW_CONSTRAINTS_REFERENCE="constraints-2-0" \
  --build-arg AIRFLOW_SOURCES_FROM="empty" \
  --build-arg AIRFLOW_SOURCES_TO="/empty" \
  --build-arg ADDITIONAL_AIRFLOW_EXTRAS="slack" \
  --build-arg ADDITIONAL_PYTHON_DEPS="apache-airflow-backport-providers-odbc \
      apache-airflow-backport-providers-odbc \
      azure-storage-blob \
      sshtunnel \
      google-api-python-client \
      oauth2client \
      beautifulsoup4 \
      dateparser \
      rocketchat_API \
      typeform" \
  --build-arg ADDITIONAL_DEV_APT_DEPS="msodbcsql17 unixodbc-dev g++" \
  --build-arg ADDITIONAL_DEV_APT_COMMAND="curl https://packages.microsoft.com/keys/microsoft.asc | \
  apt-key add --no-tty - && \
  curl https://packages.microsoft.com/config/debian/10/prod.list > /etc/apt/sources.list.d/mssql-release.list" \
  --build-arg ADDITIONAL_DEV_ENV_VARS="ACCEPT_EULA=Y" \
  --build-arg ADDITIONAL_RUNTIME_APT_COMMAND="curl https://packages.microsoft.com/keys/microsoft.asc | \
  apt-key add --no-tty - && \
  curl https://packages.microsoft.com/config/debian/10/prod.list > /etc/apt/sources.list.d/mssql-release.list" \
  --build-arg ADDITIONAL_RUNTIME_APT_DEPS="msodbcsql17 unixodbc git procps vim" \
  --build-arg ADDITIONAL_RUNTIME_ENV_VARS="ACCEPT_EULA=Y" \
  --tag my-image
```

Customizing images in high security restricted environments

You can also make sure your image is only build using local constraint file and locally downloaded wheel files. This is often useful in Enterprise environments where the binary files are verified and vetted by the security teams.

This builds below builds the production image in version 3.7 with packages and constraints used from the local `docker-context-files` rather than installed from PyPI or GitHub. It also disables MySQL client installation as it is using external installation method.

Note that as a prerequisite - you need to have downloaded wheel files. In the example below we first download such constraint file locally and then use `pip download` to get the .whl files needed but in most likely scenario, those wheel files should be copied from an internal repository of such .whl files. Note that `AIRFLOW_INSTALL_VERSION` is only there for reference, the apache airflow .whl file in the right version is part of the .whl files downloaded.

Note that 'pip download' will only works on Linux host as some of the packages need to be compiled from sources and you cannot install them providing `--platform` switch. They also need to be downloaded using the same python version as the target image.

The `pip download` might happen in a separate environment. The files can be committed to a separate binary repository and vetted/verified by the security team and used subsequently to build images of Airflow when needed on an air-gaped system.

Preparing the constraint files and wheel files:

```
rm docker-context-files/*.whl docker-context-files/*.txt

curl -Lo "docker-context-files/constraints-2-0.txt" \
  https://raw.githubusercontent.com/apache/airflow/constraints-2-0/constraints-3.7.txt

pip download --dest docker-context-files \
  --constraint docker-context-files/constraints-2-0.txt  \
  apache-airflow[async,aws,azure,celery,dask,elasticsearch,gcp,kubernetes,mysql,postgres,redis,slack,ssh,statsd,virtualenv]==2.0.0
```

Since apache-airflow .whl packages are treated differently by the docker image, you need to rename the downloaded apache-airflow* files, for example:

```
pushd docker-context-files
for file in apache?airflow*
do
  mv ${file} _${file}
done
popd
```

Building the image:

```
./breeze build-image \
    --production-image --python 3.7 --install-airflow-version=2.0.0 \
    --disable-mysql-client-installation --disable-pip-cache --install-from-local-files-when-building \
    --constraints-location="/docker-context-files/constraints-2-0.txt"
```

or

```
docker build . \
  --build-arg PYTHON_BASE_IMAGE="python:3.7-slim-buster" \
  --build-arg PYTHON_MAJOR_MINOR_VERSION=3.7 \
  --build-arg AIRFLOW_INSTALLATION_METHOD="apache-airflow" \
  --build-arg AIRFLOW_VERSION="2.0.0" \
  --build-arg AIRFLOW_INSTALL_VERSION="==2.0.0" \
  --build-arg AIRFLOW_CONSTRAINTS_REFERENCE="constraints-2-0" \
  --build-arg AIRFLOW_SOURCES_FROM="empty" \
  --build-arg AIRFLOW_SOURCES_TO="/empty" \
  --build-arg INSTALL_MYSQL_CLIENT="false" \
  --build-arg AIRFLOW_PRE_CACHED_PIP_PACKAGES="false" \
  --build-arg INSTALL_FROM_DOCKER_CONTEXT_FILES="true" \
  --build-arg AIRFLOW_CONSTRAINTS_LOCATION="/docker-context-files/constraints-2-0.txt"
```

Customizing & extending the image together

You can combine both - customizing & extending the image. You can build the image first using `customize` method (either with docker command or with `breeze` and then you can `extend` the resulting image using `FROM` any dependencies you want.

Customizing PYPI installation

You can customize PYPI sources used during image build by adding a docker-context-files/.pypirc file This .pypirc will never be committed to the repository and will not be present in the final production image. It is added and used only in the build segment of the image so it is never copied to the final image.

External sources for dependencies

In corporate environments, there is often the need to build your Container images using other than default sources of dependencies. The docker file uses standard sources (such as Debian apt repositories or PyPI repository. However, in corporate environments, the dependencies are often only possible to be installed from internal, vetted repositories that are reviewed and approved by the internal security teams. In those cases, you might need to use those different sources.

This is rather easy if you extend the image - you simply write your extension commands using the right sources - either by adding/replacing the sources in apt configuration or specifying the source repository in pip install command.

It's a bit more involved in the case of customizing the image. We do not have yet (but we are working on it) a capability of changing the sources via build args. However, since the builds use Dockerfile that is a source file, you can rather easily simply modify the file manually and specify different sources to be used by either of the commands.

Comparing extending and customizing the image

Here is the comparison of the two types of building images.

	Extending the image	Customizing the image
Produces optimized image	No	Yes
Use Airflow Dockerfile sources to build the image	No	Yes
Requires Airflow sources	No	Yes
You can build it with Breeze	No	Yes
Allows to use non-default sources for dependencies	Yes	No [1]

[1] When you combine customizing and extending the image, you can use external sources in the "extend" part. There are plans to add functionality to add external sources option to image customization. You can also modify Dockerfile manually if you want to use non-default sources for dependencies.

Using the production image

The PROD image entrypoint works as follows:

- In case the user is not "airflow" (with undefined user id) and the group id of the user is set to 0 (root), then the user is dynamically added to /etc/passwd at entry using USER_NAME variable to define the user name. This is in order to accommodate the OpenShift Guidelines
- The `AIRFLOW_HOME` is set by default to `/opt/airflow/` - this means that DAGs are in default in the `/opt/airflow/dags` folder and logs are in the `/opt/airflow/logs`
- The working directory is `/opt/airflow` by default.
- If `AIRFLOW__CORE__SQL_ALCHEMY_CONN` variable is passed to the container and it is either mysql or postgres SQL alchemy connection, then the connection is checked and the script waits until the database is reachable. If `AIRFLOW__CORE__SQL_ALCHEMY_CONN_CMD` variable is passed to the container, it is evaluated as a command to execute and result of this evaluation is used as `AIRFLOW__CORE__SQL_ALCHEMY_CONN`. The `_CMD` variable takes precedence over the `AIRFLOW__CORE__SQL_ALCHEMY_CONN` variable.
- If no `AIRFLOW__CORE__SQL_ALCHEMY_CONN` variable is set then SQLite database is created in ${AIRFLOW_HOME}/airflow.db and db reset is executed.
- If first argument equals to "bash" - you are dropped to a bash shell or you can executes bash command if you specify extra arguments. For example:

```
docker run -it apache/airflow:master-python3.6 bash -c "ls -la"
total 16
drwxr-xr-x 4 airflow root 4096 Jun  5 18:12 .
drwxr-xr-x 1 root    root 4096 Jun  5 18:12 ..
drwxr-xr-x 2 airflow root 4096 Jun  5 18:12 dags
drwxr-xr-x 2 airflow root 4096 Jun  5 18:12 logs
```

- If first argument is equal to “python” - you are dropped in python shell or python commands are executed if you pass extra parameters. For example:

```
> docker run -it apache/airflow:master-python3.6 python -c "print('test')"
test
```

- If first argument equals to “airflow” - the rest of the arguments is treated as an airflow command to execute. Example:

```
docker run -it apache/airflow:master-python3.6 airflow webserver
```

- If there are any other arguments - they are simply passed to the “airflow” command

```
> docker run -it apache/airflow:master-python3.6 version
2.0.0.dev0
```

- If `AIRFLOW__CELERY__BROKER_URL` variable is passed and airflow command with scheduler, worker of flower command is used, then the script checks the broker connection and waits until the Celery broker database is reachable. If `AIRFLOW__CELERY__BROKER_URL_CMD` variable is passed to the container, it is evaluated as a command to execute and result of this evaluation is used as `AIRFLOW__CELERY__BROKER_URL` . The `_CMD` variable takes precedence over the `AIRFLOW__CELERY__BROKER_URL` variable.

Production image build arguments

The following build arguments (`--build-arg` in docker build command) can be used for production images:

Build argument	Default value	Description
`PYTHON_BASE_IMAGE`	`python:3.6-slim-buster`	Base python image.
`PYTHON_MAJOR_MINOR_VERSION`	`3.6`	major/minor version of Python (should match base image).
`AIRFLOW_VERSION`	`2.0.0.dev0`	version of Airflow.
`AIRFLOW_REPO`	`apache/airflow`	the repository from which PIP dependencies are pre-installed.
`AIRFLOW_BRANCH`	`master`	the branch from which PIP dependencies are pre-installed initially.
`AIRFLOW_CONSTRAINTS_LOCATION`		If not empty, it will override the source of the constraints with the specified URL or file. Note that the file has to be in docker context so it’s best to place such file in one of the folders included in .dockerignore.

Build argument	Default value	Description
AIRFLOW_CONSTRAINTS_REFERENCE	constraints-master	Reference (branch or tag) from GitHub where constraints file is taken from It can be constraints-master but also can be constraints-1-10 for 1.10.* installation. In case of building specific version you want to point it to specific tag, for example constraints-1.10.14 .
INSTALL_PROVIDERS_FROM_SOURCES	false	If set to true and image is built from sources, all provider packages are installed from sources rather than from packages. It has no effect when installing from PyPI or GitHub repo.
AIRFLOW_EXTRAS	(see Dockerfile)	Default extras with which airflow is installed.
INSTALL_FROM_PYPI	true	If set to true, Airflow is installed from PyPI. if you want to install Airflow from self-build package you can set it to false, put package in docker-context-files and set INSTALL_FROM_DOCKER_CONTEXT_FILES to true . For this you have to also keep AIRFLOW_PRE_CACHED_PIP_PACKAGES flag set to false .
AIRFLOW_PRE_CACHED_PIP_PACKAGES	false	Allows to pre-cache airflow PIP packages from the GitHub of Apache Airflow This allows to optimize iterations for Image builds and speeds up CI builds.
INSTALL_FROM_DOCKER_CONTEXT_FILES	false	If set to true, Airflow, providers and all dependencies are installed from from locally built/downloaded .whl and .tar.gz files placed in the docker-context-files . In certain corporate environments, this is required to install airflow from such pre-vetted packages rather than from PyPI. For this to work, also set INSTALL_FROM_PYPI . Note that packages starting with apache?airflow glob are treated differently than other packages. All apache?airflow packages are installed with dependencies limited by airflow constraints. All other packages are installed without dependencies 'as-is'. If you wish to install airflow via 'pip download' with all dependencies downloaded, you have to rename the apache airflow and provider packages to not start with apache?airflow glob.
UPGRADE_TO_NEWER_DEPENDENCIES	false	If set to true, the dependencies are upgraded to newer versions matching setup.py before installation.
CONTINUE_ON_PIP_CHECK_FAILURE	false	By default the image build fails if pip check fails for it. This is good for interactive building but on CI the image should be built regardless - we have a separate step to verify image.

Build argument	Default value	Description
ADDITIONAL_AIRFLOW_EXTRAS		Optional additional extras with which airflow is installed.
ADDITIONAL_PYTHON_DEPS		Optional python packages to extend the image with some extra dependencies.
DEV_APT_COMMAND	(see Dockerfile)	Dev apt command executed before dev deps are installed in the Build image.
ADDITIONAL_DEV_APT_COMMAND		Additional Dev apt command executed before dev dep are installed in the Build image. Should start with `&&` .
DEV_APT_DEPS	(see Dockerfile)	Dev APT dependencies installed in the Build image.
ADDITIONAL_DEV_APT_DEPS		Additional apt dev dependencies installed in the Build image.
ADDITIONAL_DEV_APT_ENV		Additional env variables defined when installing dev deps.
RUNTIME_APT_COMMAND	(see Dockerfile)	Runtime apt command executed before deps are installed in the Main image.
ADDITIONAL_RUNTIME_APT_COMMAND		Additional Runtime apt command executed before runtime dep are installed in the Main image. Should start with `&&` .
RUNTIME_APT_DEPS	(see Dockerfile)	Runtime APT dependencies installed in the Main image.
ADDITIONAL_RUNTIME_APT_DEPS		Additional apt runtime dependencies installed in the Main image.
ADDITIONAL_RUNTIME_APT_ENV		Additional env variables defined when installing runtime deps.
AIRFLOW_HOME	/opt/airflow	Airflow's HOME (that's where logs and SQLite databases are stored).
AIRFLOW_UID	50000	Airflow user UID.
AIRFLOW_GID	50000	Airflow group GID. Note that most files created on behalf of airflow user belong to the `root` group (0) to keep OpenShift Guidelines compatibility.
AIRFLOW_USER_HOME_DIR	/home/airflow	Home directory of the Airflow user.
CASS_DRIVER_BUILD_CONCURRENCY	8	Number of processors to use for cassandra PIP install (speeds up installing in case cassandra extra is used).
INSTALL_MYSQL_CLIENT	true	Whether MySQL client should be installed The mysql extra is removed from extras if the client is not installed.

There are build arguments that determine the installation mechanism of Apache Airflow for the production image. There are three types of build:

- From local sources (by default for example when you use `docker build .`)
- You can build the image from released PyPI airflow package (used to build the official Docker image)
- You can build the image from any version in GitHub repository(this is used mostly for system testing).

Build argument	Default	What to specify

Build argument	Default	What to specify
AIRFLOW_INSTALLATION_METHOD	apache-airflow	Should point to the installation method of Apache Airflow. It can be apache-airflow for installation from packages and URL to installation from GitHub repository tag or branch or "." to install from sources. Note that installing from local sources requires appropriate values of the AIRFLOW_SOURCES_FROM and AIRFLOW_SOURCES_TO variables as described below. Only used when INSTALL_FROM_PYPI is set to true .
AIRFLOW_INSTALL_VERSION		Optional - might be used for package installation of different Airflow version for example"==2.0.0". For consistency, you should also set``AIRFLOW_VERSION`` to the same value AIRFLOW_VERSION is embedded as label in the image created.
AIRFLOW_CONSTRAINTS_REFERENCE	constraints-master	Reference (branch or tag) from GitHub where constraints file is taken from. It can be constraints-master but also can be``constraints-1-10`` for 1.10.* installations. In case of building specific version you want to point it to specific tag, for example constraints-2.0.0
AIRFLOW_WWW	www	In case of Airflow 2.0 it should be "www", in case of Airflow 1.10 series it should be "www_rbac".
AIRFLOW_SOURCES_FROM	empty	Sources of Airflow. Set it to "." when you install airflow from local sources.
AIRFLOW_SOURCES_TO	/empty	Target for Airflow sources. Set to "/opt/airflow" when you want to install airflow from local sources.

This builds production image in version 3.6 with default extras from the local sources (master version of 2.0 currently):

```
docker build .
```

This builds the production image in version 3.7 with default extras from 2.0.0 tag and constraints taken from constraints-2-0 branch in GitHub.

```
docker build . \
  --build-arg PYTHON_BASE_IMAGE="python:3.7-slim-buster" \
  --build-arg PYTHON_MAJOR_MINOR_VERSION=3.7 \
  --build-arg AIRFLOW_INSTALLATION_METHOD="https://github.com/apache/airflow/archive/2.0.0.tar.gz#egg=apache-airflow" \
  --build-arg AIRFLOW_CONSTRAINTS_REFERENCE="constraints-2-0" \
  --build-arg AIRFLOW_BRANCH="v1-10-test" \
  --build-arg AIRFLOW_SOURCES_FROM="empty" \
  --build-arg AIRFLOW_SOURCES_TO="/empty"
```

This builds the production image in version 3.7 with default extras from 2.0.0 PyPI package and constraints taken from 2.0.0 tag in GitHub and pre-installed pip dependencies from the top of v1-10-test branch.

```
docker build . \
  --build-arg PYTHON_BASE_IMAGE="python:3.7-slim-buster" \
  --build-arg PYTHON_MAJOR_MINOR_VERSION=3.7 \
  --build-arg AIRFLOW_INSTALLATION_METHOD="apache-airflow" \
  --build-arg AIRFLOW_VERSION="2.0.0" \
  --build-arg AIRFLOW_INSTALL_VERSION="==2.0.0" \
  --build-arg AIRFLOW_BRANCH="v1-10-test" \
  --build-arg AIRFLOW_CONSTRAINTS_REFERENCE="constraints-2.0.0" \
  --build-arg AIRFLOW_SOURCES_FROM="empty" \
  --build-arg AIRFLOW_SOURCES_TO="/empty"
```

This builds the production image in version 3.7 with additional airflow extras from 2.0.0 PyPI package and additional python dependencies and pre-installed pip dependencies from 2.0.0 tagged constraints.

```
docker build . \
  --build-arg PYTHON_BASE_IMAGE="python:3.7-slim-buster" \
  --build-arg PYTHON_MAJOR_MINOR_VERSION=3.7 \
  --build-arg AIRFLOW_INSTALLATION_METHOD="apache-airflow" \
  --build-arg AIRFLOW_VERSION="2.0.0" \
  --build-arg AIRFLOW_INSTALL_VERSION="==2.0.0" \
  --build-arg AIRFLOW_BRANCH="v1-10-test" \
  --build-arg AIRFLOW_CONSTRAINTS_REFERENCE="constraints-2.0.0" \
  --build-arg AIRFLOW_SOURCES_FROM="empty" \
  --build-arg AIRFLOW_SOURCES_TO="/empty" \
  --build-arg ADDITIONAL_AIRFLOW_EXTRAS="mssql,hdfs" \
  --build-arg ADDITIONAL_PYTHON_DEPS="sshtunnel oauth2client"
```

This builds the production image in version 3.7 with additional airflow extras from 2.0.0 PyPI package and additional apt dev and runtime dependencies.

```
docker build . \
  --build-arg PYTHON_BASE_IMAGE="python:3.7-slim-buster" \
  --build-arg PYTHON_MAJOR_MINOR_VERSION=3.7 \
  --build-arg AIRFLOW_INSTALLATION_METHOD="apache-airflow" \
  --build-arg AIRFLOW_VERSION="2.0.0" \
  --build-arg AIRFLOW_INSTALL_VERSION="==2.0.0" \
  --build-arg AIRFLOW_CONSTRAINTS_REFERENCE="constraints-2-0" \
  --build-arg AIRFLOW_SOURCES_FROM="empty" \
  --build-arg AIRFLOW_SOURCES_TO="/empty" \
  --build-arg ADDITIONAL_AIRFLOW_EXTRAS="jdbc" \
  --build-arg ADDITIONAL_DEV_APT_DEPS="gcc g++" \
  --build-arg ADDITIONAL_RUNTIME_APT_DEPS="default-jre-headless"
```

Actions executed at image start

If you are using the default entrypoint of the production image, there are a few actions that are automatically performed when the container starts. In some cases, you can pass environment variables to the image to trigger some of that behaviour.

The variables that control the “execution” behaviour start with `_AIRFLOW` to distinguish them from the variables used to build the image starting with `AIRFLOW`.

Creating system user

Airflow image is Open-Shift compatible, which means that you can start it with random user ID and group id 0. Airflow will automatically create such a user and make it’s home directory point to `/home/airflow`. You can read more about it in the “Support arbitrary user ids” chapter in the Openshift best practices.

Waits for Airflow DB connection

In case Postgres or MySQL DB is used, the entrypoint will wait until the airflow DB connection becomes available. This happens always when you use the default entrypoint.

The script detects backend type depending on the URL schema and assigns default port numbers if not specified in the URL. Then it loops until the connection to the host/port specified can be established It tries `CONNECTION_CHECK_MAX_COUNT` times and sleeps `CONNECTION_CHECK_SLEEP_TIME` between checks

Supported schemes:

- `postgres://` - default port 5432
- `mysql://` - default port 3306
- `sqlite://`

In case of SQLite backend, there is no connection to establish and waiting is skipped.

Upgrading Airflow DB

If you set `_AIRFLOW_DB_UPGRADE` variable to a non-empty value, the entrypoint will run the `airflow db upgrade` command right after verifying the connection. You can also use this when you are running airflow with internal SQLite database (default) to upgrade the db and create admin users at

entrypoint, so that you can start the webserver immediately. Note - using SQLite is intended only for testing purpose, never use SQLite in production as it has severe limitations when it comes to concurrency.

Creating admin user

The entrypoint can also create webserver user automatically when you enter it. you need to set `_AIRFLOW_WWW_USER_CREATE` to a non-empty value in order to do that. This is not intended for production, it is only useful if you would like to run a quick test with the production image. You need to pass at least password to create such user via `_AIRFLOW_WWW_USER_PASSWORD_CMD` or `_AIRFLOW_WWW_USER_PASSWORD_CMD` similarly like for other `*_CMD` variables, the content of the `*_CMD` will be evaluated as shell command and it's output will be set ass password.

User creation will fail if none of the `PASSWORD` variables are set - there is no default for password for security reasons.

Parameter	Default	Environment variable
username	admin	`_AIRFLOW_WWW_USER_USERNAME`
password		`_AIRFLOW_WWW_USER_PASSWORD_CMD` or `_AIRFLOW_WWW_USER_PASSWORD`
firstname	Airflow	`_AIRFLOW_WWW_USER_FIRSTNAME`
lastname	Admin	`_AIRFLOW_WWW_USER_LASTNAME`
email	airflowadmin@example.com	`_AIRFLOW_WWW_USER_EMAIL`
role	Admin	`_AIRFLOW_WWW_USER_ROLE`

In case the password is specified, the user will be attempted to be created, but the entrypoint will not fail if the attempt fails (this accounts for the case that the user is already created).

You can, for example start the webserver in the production image with initializing the internal SQLite database and creating an `admin/admin` Admin user with the following command:

```
docker run -it -p 8080:8080 \
  --env "_AIRFLOW_DB_UPGRADE=true" \
  --env "_AIRFLOW_WWW_USER_CREATE=true" \
  --env "_AIRFLOW_WWW_USER_PASSWORD=admin" \
    apache/airflow:master-python3.8 webserver
```

```
docker run -it -p 8080:8080 \
  --env "_AIRFLOW_DB_UPGRADE=true" \
  --env "_AIRFLOW_WWW_USER_CREATE=true" \
  --env "_AIRFLOW_WWW_USER_PASSWORD_CMD=echo admin" \
    apache/airflow:master-python3.8 webserver
```

The commands above perform initialization of the SQLite database, create admin user with admin password and Admin role. They also forward local port `8080` to the webserver port and finally start the webserver.

Waits for celery broker connection

In case Postgres or MySQL DB is used, and one of the `scheduler`, `celery`, `worker`, or `flower` commands are used the entrypoint will wait until the celery broker DB connection is available.

The script detects backend type depending on the URL schema and assigns default port numbers if not specified in the URL. Then it loops until connection to the host/port specified can be established It tries `CONNECTION_CHECK_MAX_COUNT` times and sleeps `CONNECTION_CHECK_SLEEP_TIME` between checks

Supported schemes:

- `amqp(s)://` (rabbitmq) - default port 5672
- `redis://` - default port 6379
- `postgres://` - default port 5432

- `mysql://` - default port 3306
- `sqlite://`

In case of SQLite backend, there is no connection to establish and waiting is skipped.

Recipes

Users sometimes share interesting ways of using the Docker images. We encourage users to contribute these recipes to the documentation in case they prove useful to other members of the community by submitting a pull request. The sections below capture this knowledge.

Google Cloud SDK installation

Some operators, such as **`airflow.providers.google.cloud.operators.kubernetes_engine.GKEStartPodOperator`**, **`airflow.providers.google.cloud.operators.dataflow.DataflowStartSqlJobOperator`**, require the installation of Google Cloud SDK (includes `gcloud`). You can also run these commands with BashOperator.

Create a new Dockerfile like the one shown below.

docs/apache-airflow/docker-images-recipes/gcloud.Dockerfile

```
# Licensed to the Apache Software Fo
# contributor license agreements.  S
# this work for additional informati
# The ASF licenses this file to You
# (the "License"); you may not use t
# the License.  You may obtain a cop
#
#    http://www.apache.org/licenses/
#
# Unless required by applicable law
# distributed under the License is d
# WITHOUT WARRANTIES OR CONDITIONS O
# See the License for the specific l
# limitations under the License.
ARG BASE_AIRFLOW_IMAGE
FROM ${BASE_AIRFLOW_IMAGE}

SHELL ["/bin/bash", "-o", "pipefail"

USER 0

ARG CLOUD_SDK_VERSION=322.0.0
ENV GCLOUD_HOME=/opt/google-cloud-sd

ENV PATH="${GCLOUD_HOME}/bin/:${PATH

RUN DOWNLOAD_URL="https://dl.google.
    && TMP_DIR="$(mktemp -d)" \
    && curl -fL "${DOWNLOAD_URL}" --
    && mkdir -p "${GCLOUD_HOME}" \
    && tar xzf "${TMP_DIR}/google-cl
    && "${GCLOUD_HOME}/install.sh" \
       --bash-completion=false \
       --path-update=false \
       --usage-reporting=false \
       --additional-components alpha
       --quiet \
    && rm -rf "${TMP_DIR}" \
    && gcloud --version

USER ${AIRFLOW_UID}
```

Then build a new image.

```
docker build . \
  --build-arg BASE_AIRFLOW_IMAGE="apache/airflow:2.0.0" \
  -t my-airflow-image
```

Apache Hadoop Stack installation

Airflow is often used to run tasks on Hadoop cluster. It required Java Runtime Environment (JRE) to run. Below are the steps to take tools that are frequently used in Hadoop-world:

- Java Runtime Environment (JRE)
- Apache Hadoop
- Apache Hive
- Cloud Storage connector for Apache Hadoop

Create a new Dockerfile like the one shown below.

docs/apache-airflow/docker-images-recipes/hadoop.Dockerfile

```
# Licensed to the Apache Software Fo
# contributor license agreements.  S
# this work for additional informati
# The ASF licenses this file to You
# (the "License"); you may not use t
# the License.  You may obtain a cop
#
#    http://www.apache.org/licenses/
#
# Unless required by applicable law
# distributed under the License is d
# WITHOUT WARRANTIES OR CONDITIONS O
# See the License for the specific l
# limitations under the License.
ARG BASE_AIRFLOW_IMAGE

FROM ${BASE_AIRFLOW_IMAGE}

SHELL ["/bin/bash", "-o", "pipefail"

USER 0

# Install Java
RUN mkdir -pv /usr/share/man/man1 \
    && mkdir -pv /usr/share/man/man7
    && curl -fsSL https://adoptopenj
    && echo 'deb https://adoptopenjd
        /etc/apt/sources.list.d/adop
    && apt-get update \
    && apt-get install --no-install-
      adoptopenjdk-8-hotspot-jre \
    && apt-get autoremove -yqq --pur
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*
ENV JAVA_HOME=/usr/lib/jvm/adoptoper

RUN mkdir -p /opt/spark/jars

# Install Apache Hadoop
ARG HADOOP_VERSION=2.10.1
ENV HADOOP_HOME=/opt/hadoop
ENV HADOOP_CONF_DIR=/etc/hadoop
ENV MULTIHOMED_NETWORK=1
ENV USER=root

RUN HADOOP_URL="https://archive.apac
    && curl 'https://dist.apache.org
    && curl -fSL "$HADOOP_URL" -o /t
    && curl -fSL "$HADOOP_URL.asc" -
    && gpg --verify /tmp/hadoop.tar.
    && tar -xvf /tmp/hadoop.tar.gz
```

```
    && tar -xvf /tmp/hadoop.tar.gz -
    && rm /tmp/hadoop.tar.gz /tmp/ha
    && ln -s "${HADOOP_HOME}/etc/hac
    && mkdir "${HADOOP_HOME}/logs" \
    && mkdir /hadoop-data

ENV PATH="$HADOOP_HOME/bin/:$PATH"

# Install Apache Hive
ARG HIVE_VERSION=2.3.7
ENV HIVE_HOME=/opt/hive
ENV HIVE_CONF_DIR=/etc/hive

RUN HIVE_URL="https://archive.apache
    && curl -fSL 'https://downloads.
    && curl -fSL "$HIVE_URL" -o /tmp
    && curl -fSL "$HIVE_URL.asc" -o
    && gpg --verify /tmp/hive.tar.gz
    && mkdir -p "${HIVE_HOME}" \
    && tar -xf /tmp/hive.tar.gz -C '
    && rm /tmp/hive.tar.gz /tmp/hive
    && ln -s "${HIVE_HOME}/etc/hive'
    && mkdir "${HIVE_HOME}/logs"

ENV PATH="$HIVE_HOME/bin/:$PATH"

# Install GCS connector for Apache H
# See: https://cloud.google.com/data
ARG GCS_VARIANT="hadoop2"
ARG GCS_VERSION="2.1.5"

RUN GCS_JAR_PATH="/opt/spark/jars/gc
    && GCS_JAR_URL="https://storage.
    && curl "${GCS_JAR_URL}" -o "${G

ENV HADOOP_CLASSPATH="/opt/spark/jar

USER ${AIRFLOW_UID}
```

Then build a new image.

```
docker build . \
  --build-arg BASE_AIRFLOW_IMAGE="apache/airflow:2.0.0" \
  -t my-airflow-image
```

More details about the images

You can read more details about the images - the context, their parameters and internal structure in the IMAGES.rst document.

Kerberos-authenticated workers

Apache Airflow has a built-in mechanism for authenticating the operation with a KDC (Key Distribution Center). Airflow has a separate command `airflow kerberos` that acts as token refresher. It uses the pre-configured Kerberos Keytab to authenticate in the KDC to obtain a valid token, and then refreshing valid token at regular intervals within the current token expiry window.

Each request for refresh uses a configured principal, and only keytab valid for the principal specified is capable of retrieving the authentication token.

The best practice to implement proper security mechanism in this case is to make sure that worker workloads have no access to the Keytab but only have access to the periodically refreshed, temporary authentication tokens. This can be achieved in docker environment by running the `airflow kerberos` command and the worker command in separate containers - where only the `airflow kerberos` token has access to the Keytab file (preferably configured as secret resource). Those two containers should share a volume where the temporary token should be written by the `airflow kerberos` and read by the workers.

In the Kubernetes environment, this can be realized by the concept of side-car, where both Kerberos token refresher and worker are part of the same Pod. Only the Kerberos side-car has access to Keytab secret and both containers in the same Pod share the volume, where temporary token is written by the side-care container and read by the worker container.

This concept is implemented in the development version of the Helm Chart that is part of Airflow source code.

Secured Server and Service Access on Google Cloud

This section describes techniques and solutions for securely accessing servers and services when your Airflow environment is deployed on Google Cloud, or you connect to Google services, or you are connecting to the Google API.

IAM and Service Accounts

You should not rely on internal network segmentation or firewalling as our primary security mechanisms. To protect your organization's data, every request you make should contain sender identity. In the case of Google Cloud, the identity is provided by the IAM and Service account. Each Compute Engine instance has an associated service account identity. It provides cryptographic credentials that your workload can use to prove its identity when making calls to Google APIs or third-party services. Each instance has access only to short-lived credentials. If you use Google-managed service account keys, then the private key is always held in escrow and is never directly accessible.

If you are using Kubernetes Engine, you can use Workload Identity to assign an identity to individual pods.

For more information about service accounts in the Airflow, see Google Cloud Connection

Impersonate Service Accounts

If you need access to other service accounts, you can impersonate other service accounts to exchange the token with the default identity to another service account. Thus, the account keys are still managed by Google and cannot be read by your workload.

It is not recommended to generate service account keys and store them in the metadata database or the secrets backend. Even with the use of the backend secret, the service account key is available for your workload.

Access to Compute Engine Instance

If you want to establish an SSH connection to the Compute Engine instance, you must have the network address of this instance and credentials to access it. To simplify this task, you can use `ComputeEngineHook` instead of `SSHHook`

The `ComputeEngineHook` support authorization with Google OS Login service. It is an extremely robust way to manage Linux access properly as it stores short-lived ssh keys in the metadata service, offers PAM modules for access and sudo privilege checking and offers the `nsswitch` user lookup into the metadata service as well.

It also solves the discovery problem that arises as your infrastructure grows. You can use the instance name instead of the network address.

Access to Amazon Web Service

Thanks to the Web Identity Federation, you can exchange the Google Cloud Platform identity to the Amazon Web Service identity, which effectively means access to Amazon Web Service platform. For more information, see: Google Cloud to AWS authentication using Web Identity Federation

Was this entry helpful?

☆ ☆ ☆ ☆ ☆

Want to be a part of Apache Airflow? Join community

License Donate Thanks
Security
© The Apache Software Foundation 2019

© The Apache Software Foundation 2019

Apache Airflow, Apache, Airflow, the Airflow logo, and the Apache feather logo are either registered trademarks or trademarks of The Apache Software Foundation. All other products or name brands are trademarks of their respective holders, including The Apache Software Foundation.