Upgrading to Airflow 2.0+

- Step 1: Upgrade to Python 3
- Step 2: Upgrade to Airflow 1.10.14 (a.k.a our “bridge” release)
- Step 3: Install and run the Upgrade check scripts
- Step 4: Import Operators from Backport Providers
- Step 5: Upgrade Airflow DAGs
- Step 6: Upgrade Configuration settings
- Step 7: Upgrade to Airflow 2.0
- Frequently Asked Questions on Upgrade
- Appendix
  - Changed Parameters for the KubernetesPodOperator
  - Migration Guide from Experimental API to Stable API v1
  - Changes to Exception handling for from DAG callbacks
  - Airflow CLI changes in 2.0
  - Changes to Airflow Plugins
  - Changes to extras names
  - Support for Airflow 1.10.x releases

Apache Airflow 2.0 is a major release and the purpose of this document is to assist users to migrate from Airflow 1.10.x to Airflow 2.0.

Step 1: Upgrade to Python 3

Airflow 1.10 will be the last release series to support Python 2. Airflow 2.0.0 requires Python 3.6+ and has been tested with Python versions 3.6, 3.7 and 3.8, but does not yet support Python 3.9.

If you have a specific task that still requires Python 2 then you can use the `PythonVirtualenvOperator` or the `KubernetesPodOperator` for this.

For a list of breaking changes between Python 2 and Python 3, please refer to this handy blog from the CouchBaseDB team.

Step 2: Upgrade to Airflow 1.10.14 (a.k.a our “bridge” release)

To minimize friction for users upgrading from Airflow 1.10 to Airflow 2.0 and beyond, Airflow 1.10.14 “a bridge release” has been created. This is intended to be the final 1.10 feature release. Airflow 1.10.14 includes support for various features that have been backported from Airflow 2.0 to make it easy for users to test their Airflow environment before upgrading to Airflow 2.0.

We strongly recommend that all users upgrading to Airflow 2.0, first upgrade to Airflow 1.10.14 and test their Airflow deployment and only then upgrade to Airflow 2.0. The Airflow 1.10.x release tree will be supported for six months from Airflow 2.0 release date.

Features in 1.10.14 include:

1. Most breaking DAG and architecture changes of Airflow 2.0 have been backported to Airflow 1.10.14. This backward-compatibility does not mean that 1.10.14 will process these DAGs the same way as Airflow 2.0. Instead, this means that most Airflow 2.0 compatible DAGs will work in Airflow 1.10.14. This backport will give users time to modify their DAGs over time without any service disruption.

2. We have also backported the updated Airflow 2.0 CLI commands to Airflow 1.10.14, so that users can modify their scripts to be compatible with Airflow 2.0 before the upgrade.

3. For users of the KubernetesExecutor, we have backported the `pod_template_file` capability for the KubernetesExecutor as well as a script that will generate a `pod_template_file` based on your `airflow.cfg` settings. To generate this file simply run the following command:

```
airflow generate_pod_template -o <output file path>
```

Once you have performed this step, simply write out the file path to this file in the `pod_template_file` config of the `kubernetes` section of your `airflow.cfg`

Step 3: Install and run the Upgrade check scripts

After upgrading to Airflow 1.10.14, we recommend that you install the "upgrade check" scripts. These scripts will read through your `airflow.cfg` and all of your DAGs and will give a detailed report of all changes required before upgrading. We are testing this script diligently, and our goal is that any Airflow setup that can pass these tests will be able to upgrade to 2.0 without any issues.

```
pip install apache-airflow-upgrade-check
```

Once this is installed, please run the upgrade check script.

```
airflow upgrade_check
```

More details about this process are here Upgrade Check Scripts.

Step 4: Import Operators from Backport Providers

Now that you are set up in Airflow 1.10.14 with Python a 3.6+ environment, you are ready to start porting your DAGs to Airflow 2.0 compliance!

The most important step in this transition is also the easiest step to do in pieces. All Airflow 2.0 operators are backwards compatible with Airflow 1.10 using the backport provider packages. In your own time, you can transition to using these backport-providers by pip installing the provider via PyPI and changing the import path.

For example: While historically you might have imported the DockerOperator in this fashion:

```
from airflow.operators.docker_operator import DockerOperator
```

You would now run this command to install the provider:

```
pip install apache-airflow-backport-providers-docker
```

and then import the operator with this path:

```
from airflow.providers.docker.operators.docker import DockerOperator
```

Please note that the backport provider packages are just backports of the provider packages compatible with Airflow 2.0. For example:

```
pip install 'apache-airflow[docker]'
```

automatically installs the `apache-airflow-providers-docker` package. But you can manage/upgrade/remove provider packages separately from the Airflow core.

After you upgrade to Apache Airflow 2.0, those provider packages are installed automatically when you install Airflow with extras. Several of the providers (http, ftp, sqlite, imap) will also be installed automatically when you install Airflow even without extras. You can read more about providers at Provider packages.

Step 5: Upgrade Airflow DAGs

Change to undefined variable handling in templates

Prior to Airflow 2.0 Jinja Templates would permit the use of undefined variables. They would render as an empty string, with no indication to the user an undefined variable was used. With this release, any template rendering involving undefined variables will fail the task, as well as displaying an error in the UI when rendering.

The behavior can be reverted when instantiating a DAG.

```
import jinja2

dag = DAG('simple_dag', template_undefined=jinja2.Undefined)
```

Alternatively, it is also possible to override each Jinja Template variable on an individual basis by using the `| default` Jinja filter as shown below.

```
{{ a | default(1) }}
```

Changes to the KubernetesPodOperator

Much like the `KubernetesExecutor`, the `KubernetesPodOperator` will no longer take Airflow custom classes and will instead expect either a pod_template yaml file, or `kubernetes.client.models` objects.

The one notable exception is that we will continue to support the `airflow.kubernetes.secret.Secret` class.

Whereas previously a user would import each individual class to build the pod as so:

```
from airflow.kubernetes.pod import Port
from airflow.kubernetes.volume import Volume
from airflow.kubernetes.secret import Secret
from airflow.kubernetes.volume_mount import VolumeMount


volume_config = {
    'persistentVolumeClaim': {
        'claimName': 'test-volume'
    }
}
volume = Volume(name='test-volume', configs=volume_config)
volume_mount = VolumeMount('test-volume',
                           mount_path='/root/mount_file',
                           sub_path=None,
                           read_only=True)

port = Port('http', 80)
secret_file = Secret('volume', '/etc/sql_conn', 'airflow-secrets', 'sql_alchemy_conn')
secret_env = Secret('env', 'SQL_CONN', 'airflow-secrets', 'sql_alchemy_conn')

k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo", "10"],
    labels={"foo": "bar"},
    secrets=[secret_file, secret_env],
    ports=[port],
    volumes=[volume],
    volume_mounts=[volume_mount],
    name="airflow-test-pod",
    task_id="task",
    affinity=affinity,
    is_delete_operator_pod=True,
    hostnetwork=False,
    tolerations=tolerations,
    configmaps=configmaps,
    init_containers=[init_container],
    priority_class_name="medium",
)
```

Now the user can use the `kubernetes.client.models` class as a single point of entry for creating all k8s objects.

```
from kubernetes.client import models as k8s
from airflow.kubernetes.secret import Secret


configmaps = ['test-configmap-1', 'test-configmap-2']

volume = k8s.V1Volume(
    name='test-volume',
    persistent_volume_claim=k8s.V1PersistentVolumeClaimVolumeSource(claim_name='test-volume'),
)

port = k8s.V1ContainerPort(name='http', container_port=80)
secret_file = Secret('volume', '/etc/sql_conn', 'airflow-secrets', 'sql_alchemy_conn')
secret_env = Secret('env', 'SQL_CONN', 'airflow-secrets', 'sql_alchemy_conn')
secret_all_keys = Secret('env', None, 'airflow-secrets-2')
volume_mount = k8s.V1VolumeMount(
    name='test-volume', mount_path='/root/mount_file', sub_path=None, read_only=True
)

k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo", "10"],
    labels={"foo": "bar"},
    secrets=[secret_file, secret_env],
    ports=[port],
    volumes=[volume],
    volume_mounts=[volume_mount],
    name="airflow-test-pod",
    task_id="task",
    is_delete_operator_pod=True,
    hostnetwork=False)
```

We decided to keep the Secret class as users seem to really like that simplifies the complexity of mounting Kubernetes secrets into workers.

For a more detailed list of changes to the KubernetesPodOperator API, please read the section in the Appendix titled “Changed Parameters for the KubernetesPodOperator”

Change default value for dag_run_conf_overrides_params

DagRun configuration dictionary will now by default overwrite params dictionary. If you pass some key-value pairs through `airflow dags backfill -c` or `airflow dags trigger -c`, the key-value pairs will override the existing ones in params. You can revert this behaviour by setting `dag_run_conf_overrides_params` to `False` in your `airflow.cfg`.

DAG discovery safe mode is now case insensitive

When `DAG_DISCOVERY_SAFE_MODE` is active, Airflow will now filter all files that contain the string `airflow` and `dag` in a case insensitive mode. This is being changed to better support the new `@dag` decorator.

Change to Permissions

The DAG-level permission actions, `can_dag_read` and `can_dag_edit` are deprecated as part of Airflow 2.0. They are being replaced with `can_read` and `can_edit`. When a role is given DAG-level access, the resource name (or “view menu”, in Flask App-Builder parlance) will now be prefixed with `DAG:`. So the action `can_dag_read` on `example_dag_id`, is now represented as `can_read` on `DAG:example_dag_id`. There is a special view called `DAGs` (it was called `all_dags` in versions 1.10.x) which allows the role to access all the DAGs. The default `Admin`, `Viewer`, `User`, `Op` roles can all access the `DAGs` view.

As part of running ``airflow db upgrade``, existing permissions will be migrated for you.

When DAGs are initialized with the `access_control` variable set, any usage of the old permission names will automatically be updated in the database, so this won’t be a breaking change. A DeprecationWarning will be raised.

Drop legacy UI in favor of FAB RBAC UI

Warning

Breaking change

Previously we were using two versions of the UI:

- non-RBAC UI
- Flask App Builder RBAC UI

This was difficult to maintain, because it meant we had to implement/update features in two places. With this release, we have removed the older UI in favor of the Flask App Builder RBAC UI, reducing a huge maintenance burden. There is no longer a need to set the RBAC UI explicitly in the configuration, as it is the only default UI.

If you previously used non-RBAC UI, you have to switch to the new RBAC-UI and create users to be able to access Airflow's webserver. For more details on CLI to create users see Command Line Interface and Environment Variables Reference

Please note that that custom auth backends will need re-writing to target new FAB based UI.

As part of this change, a few configuration items in `[webserver]` section are removed and no longer applicable, including `authenticate`, `filter_by_owner`, `owner_mode`, and `rbac`.

Before upgrading to this release, we recommend activating the new FAB RBAC UI. For that, you should set the `rbac` options in `[webserver]` in the `airflow.cfg` file to `True`

```
[webserver]
rbac = True
```

In order to login to the interface, you need to create an administrator account.

Assuming you have already installed Airflow 1.10.14, you can create a user with Airflow 2.0 CLI command syntax `airflow users create`. You don't need to make changes to the configuration file as the FAB RBAC UI is the only supported UI.

```
airflow users create \
    --role Admin \
    --username admin \
    --firstname FIRST_NAME \
    --lastname LAST_NAME \
    --email EMAIL@example.org
```

Breaking Change in OAuth

Note

When multiple replicas of the airflow webserver are running they need to share the same *secret_key* to access the same user session. Inject this via any configuration mechanism. The 1.10.14 bridge-release modifies this feature to use randomly generated secret keys instead of an insecure default and may break existing deployments that rely on the default.

The `flask-ouathlib` has been replaced with `authlib` because `flask-outhlib` has been deprecated in favor of `authlib`. The Old and New provider configuration keys that have changed are as follows

Old Keys	New keys
`consumer_key`	`client_id`
`consumer_secret`	`client_secret`
`base_url`	`api_base_url`
`request_token_params`	`client_kwargs`

For more information, visit https://flask-appbuilder.readthedocs.io/en/latest/security.html#authentication-oauth

Step 6: Upgrade Configuration settings

Airflow 2.0 is stricter with respect to expectations on configuration data and requires explicit specifications of configuration values in more cases rather than defaulting to a generic value.

Some of these are detailed in the Upgrade Check guide, but a significant area of change is with respect to the Kubernetes Executor. This is called out below for users of the Kubernetes Executor.

Upgrade KubernetesExecutor settings

The KubernetesExecutor Will No Longer Read from the airflow.cfg for Base Pod Configurations.

In Airflow 2.0, the KubernetesExecutor will require a base pod template written in yaml. This file can exist anywhere on the host machine and will be linked using the `pod_template_file` configuration in the `airflow.cfg` file. You can create a `pod_template_file` by running the following command: `airflow generate_pod_template`

The `airflow.cfg` will still accept values for the `worker_container_repository` , the `worker_container_tag` , and the default namespace.

The following `airflow.cfg` values will be deprecated:

```
worker_container_image_pull_policy
airflow_configmap
airflow_local_settings_configmap
dags_in_image
dags_volume_subpath
dags_volume_mount_point
dags_volume_claim
logs_volume_subpath
logs_volume_claim
dags_volume_host
logs_volume_host
env_from_configmap_ref
env_from_secret_ref
git_repo
git_branch
git_sync_depth
git_subpath
git_sync_rev
git_user
git_password
git_sync_root
git_sync_dest
git_dags_folder_mount_point
git_ssh_key_secret_name
git_ssh_known_hosts_configmap_name
git_sync_credentials_secret
git_sync_container_repository
git_sync_container_tag
git_sync_init_container_name
git_sync_run_as_user
worker_service_account_name
image_pull_secrets
gcp_service_account_keys
affinity
tolerations
run_as_user
fs_group
[kubernetes_node_selectors]
[kubernetes_annotations]
[kubernetes_environment_variables]
[kubernetes_secrets]
[kubernetes_labels]
```

The ``executor_config`` Will Now Expect a ``kubernetes.client.models.V1Pod`` Class When Launching Tasks

In Airflow 1.10.x, users could modify task pods at runtime by passing a dictionary to the `executor_config` variable. Users will now have full access the

Kubernetes API via the `kubernetes.client.models.V1Pod`.

While in the deprecated version a user would mount a volume using the following dictionary:

```
second_task = PythonOperator(
    task_id="four_task",
    python_callable=test_volume_mount,
    executor_config={
        "KubernetesExecutor": {
            "volumes": [
                {
                    "name": "example-kubernetes-test-volume",
                    "hostPath": {"path": "/tmp/"},
                },
            ],
            "volume_mounts": [
                {
                    "mountPath": "/foo/",
                    "name": "example-kubernetes-test-volume",
                },
            ]
        }
    }
)
```

In the new model a user can accomplish the same thing using the following code under the `pod_override` key:

```
from kubernetes.client import models as k8s

second_task = PythonOperator(
    task_id="four_task",
    python_callable=test_volume_mount,
    executor_config={"pod_override": k8s.V1Pod(
        spec=k8s.V1PodSpec(
            containers=[
                k8s.V1Container(
                    name="base",
                    volume_mounts=[
                        k8s.V1VolumeMount(
                            mount_path="/foo/",
                            name="example-kubernetes-test-volume"
                        )
                    ]
                )
            ],
            volumes=[
                k8s.V1Volume(
                    name="example-kubernetes-test-volume",
                    host_path=k8s.V1HostPathVolumeSource(
                        path="/tmp/"
                    )
                )
            ]
        )
    )
    }
)
```

For Airflow 2.0, the traditional `executor_config` will continue operation with a deprecation warning, but will be removed in a future version.

Step 7: Upgrade to Airflow 2.0

After running the upgrade checks as described above, installing the backported providers, modifying the DAGs to be compatible, and updating the configuration settings, you should be ready to upgrade to Airflow 2.0.

A final run of the upgrade checks is always a good idea to make sure you have missed anything. At this stage the problems detected should be either be zero or minimal which you plan to fix after upgrading the Airflow version.

At this point, just follow the standard Airflow version upgrade process:

- Make sure your Airflow meta database is backed up
- Pause all the DAGs and make sure there is nothing actively running
    - The reason to pause DAGs is to make sure that nothing is actively being written to the database during the database upgrade which will follow in a later step.
    - To be extra careful, it is best to have a database backup after the DAGs have been paused.
- Install / upgrade the Airflow version to the 2.0 version of choice
- Make sure to install the right providers
    - This can be done by using the "extras" option as part of the Airflow installation, or by individually installing the providers.
    - Please note that you may have to uninstall the backport providers before installing the new providers, if you are installing using pip. This would not apply if you are installing using an Airflow Docker image with a set of specified requirements, where the change automatically gets a fresh set of modules.
    - You can read more about providers at Provider packages.
- Upgrade the Airflow meta database using `airflow db upgrade`.
    - The above command may be unfamiliar, since it is shown using the Airflow 2.0 CLI syntax.
    - The database upgrade may modify the database schema as needed and also map the existing data to be compliant with the update database schema.

    Note

    The database upgrade may take a while depending on the number of DAGs in the database and the volume of history stored in the database for task history, xcom variables, etc. In our testing, we saw that performing the Airflow database upgrade from Airflow 1.10.14 to Airflow 2.0 took between two to three minutes on an Airflow database on PostgreSQL with around 35,000 task instances and 500 DAGs. For a faster database upgrade and for better overall performance, it is recommended that you periodically archive the old historical elements which are no longer of value.

- Restart Airflow Scheduler, Webserver, and Workers

Frequently Asked Questions on Upgrade

- Q. Why doesn't the list of connection types show up in the Airflow UI after I upgrade to 2.0? * A. It is because Airflow 2.0 does not ship with the provider packages. The connection type list in the Airflow UI is based on the providers you have installed with Airflow 2.0. Please note that these will only show up once you install the provider and restart Airflow. You can read more about providers at Provider packages.

Appendix

Changed Parameters for the KubernetesPodOperator

Port has migrated from a List[Port] to a List[V1ContainerPort]

Before:

```
from airflow.kubernetes.pod import Port
port = Port('http', 80)
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    ports=[port],
    task_id="task",
)
```

After:

```
from kubernetes.client import models as k8s
port = k8s.V1ContainerPort(name='http', container_port=80)
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    ports=[port],
    task_id="task",
)
```

Volume_mounts have migrated from a List[VolumeMount] to a List[V1VolumeMount]

Before:

```
from airflow.kubernetes.volume_mount import VolumeMount
volume_mount = VolumeMount('test-volume',
                           mount_path='/root/mount_file',
                           sub_path=None,
                           read_only=True)
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    volume_mounts=[volume_mount],
    task_id="task",
)
```

After:

```
from kubernetes.client import models as k8s
volume_mount = k8s.V1VolumeMount(
    name='test-volume', mount_path='/root/mount_file', sub_path=None, read_only=True
)
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    volume_mounts=[volume_mount],
    task_id="task",
)
```

Volume has migrated from a List[Volume] to a List[V1Volume]

Before:

```
from airflow.kubernetes.volume import Volume

volume_config = {
    'persistentVolumeClaim': {
        'claimName': 'test-volume'
}
}
volume = Volume(name='test-volume', configs=volume_config)
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    volumes=[volume],
    task_id="task",
)
```

After:

```
from kubernetes.client import models as k8s
volume = k8s.V1Volume(
    name='test-volume',
    persistent_volume_claim=k8s.V1PersistentVolumeClaimVolumeSource(claim_name='test-volume'),
)
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    volumes=[volume],
    task_id="task",
)
```

env_vars has migrated from a Dict to a List[V1EnvVar]

Before:

```
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    env_vars={"ENV1": "val1", "ENV2": "val2"},
    task_id="task",
)
```

After:

```
from kubernetes.client import models as k8s

env_vars = [
    k8s.V1EnvVar(
        name="ENV1",
        value="val1"
    ),
    k8s.V1EnvVar(
        name="ENV2",
        value="val2"
    )]

k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    env_vars=env_vars,
    task_id="task",
)
```

PodRuntimeInfoEnv has been removed

PodRuntimeInfoEnv can now be added to the `env_vars` variable as a `V1EnvVarSource`

Before:

```
from airflow.kubernetes.pod_runtime_info_env import PodRuntimeInfoEnv

k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    pod_runtime_info_envs=[PodRuntimeInfoEnv("ENV3", "status.podIP")],
    task_id="task",
)
```

After:

```
from kubernetes.client import models as k8s

env_vars = [
    k8s.V1EnvVar(
        name="ENV3",
        value_from=k8s.V1EnvVarSource(
            field_ref=k8s.V1ObjectFieldSelector(
                field_path="status.podIP"
            )
        )
    )
]

k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    env_vars=env_vars,
    task_id="task",
)
```

configmaps has been removed

Configmaps can now be added to the `env_from` variable as a `V1EnvVarSource`

Before:

```
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    configmaps=['test-configmap'],
    task_id="task"
)
```

After:

```
from kubernetes.client import models as k8s

configmap ="test-configmap"
env_from = [k8s.V1EnvFromSource(
               config_map_ref=k8s.V1ConfigMapEnvSource(
                   name=configmap
               )
           )]

k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    env_from=env_from,
    task_id="task"
)
```

Resources has migrated from a Dict to a V1ResourceRequirements

Before:

```
resources = {
    'limit_cpu': 0.25,
    'limit_memory': '64Mi',
    'limit_ephemeral_storage': '2Gi',
    'request_cpu': '250m',
    'request_memory': '64Mi',
    'request_ephemeral_storage': '1Gi',
}
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    labels={"foo": "bar"},
    name="test",
    task_id="task" + self.get_current_task_name(),
    in_cluster=False,
    do_xcom_push=False,
    resources=resources,
)
```

After:

```
from kubernetes.client import models as k8s

resources=k8s.V1ResourceRequirements(
    requests={
        'memory': '64Mi',
        'cpu': '250m',
        'ephemeral-storage': '1Gi'
    },
    limits={
        'memory': '64Mi',
        'cpu': 0.25,
        'nvidia.com/gpu': None,
        'ephemeral-storage': '2Gi'
    }
)
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    labels={"foo": "bar"},
    name="test-" + str(random.randint(0, 1000000)),
    task_id="task" + self.get_current_task_name(),
    in_cluster=False,
    do_xcom_push=False,
    resources=resources,
)
```

image_pull_secrets has migrated from a String to a List[k8s.V1LocalObjectReference]

Before:

```
k = KubernetesPodOperator(
    namespace='default',
    image="ubuntu:16.04",
    cmds=["bash", "-cx"],
    arguments=["echo 10"],
    name="test",
    task_id="task",
    image_pull_secrets="fake-secret",
    cluster_context='default'
)
```

After:

```
quay_k8s = KubernetesPodOperator(
    namespace='default',
    image='quay.io/apache/bash',
    image_pull_secrets=[k8s.V1LocalObjectReference('testquay')],
    cmds=["bash", "-cx"],
    name="airflow-private-image-pod",
    task_id="task-two",
)
```

Migration Guide from Experimental API to Stable API v1

In Airflow 2.0, we added the new REST API. Experimental API still works, but support may be dropped in the future.

The experimental API however does not require authentication, so it is disabled by default. You need to explicitly enable the experimental API if you want to use it. If your application is still using the experimental API, you should **seriously** consider migrating to the stable API.

The stable API exposes many endpoints available through the webserver. Here are the differences between the two endpoints that will help you migrate from the experimental REST API to the stable REST API.

Base Endpoint

The base endpoint for the stable API v1 is `/api/v1/`. You must change the experimental base endpoint from `/api/experimental/` to `/api/v1/`. The table below shows the differences:

Purpose	Experimental REST API Endpoint	Stable REST API Endpoint
Create a DAGRuns(POST)	`/api/experimental/dags/<DAG_ID>/dag_runs`	`/api/v1/dags/{dag_id}/dagRuns`
List DAGRuns(GET)	`/api/experimental/dags/<DAG_ID>/dag_runs`	`/api/v1/dags/{dag_id}/dagRuns`
Check Health status(GET)	`/api/experimental/test`	`/api/v1/health`
Task information(GET)	`/api/experimental/dags/<DAG_ID>/tasks/<TASK_ID>`	`/api/v1//dags/{dag_id}/tasks/{task_id}`
TaskInstance public variable(GET)	`/api/experimental/dags/<DAG_ID>/dag_runs/<string:execution_date>/tasks/<TASK_ID>`	`/api/v1/dags/{dag_id}/dagRuns/{dag_run_id}/taskInstances/{task_id}`
Pause DAG(PATCH)	`/api/experimental/dags/<DAG_ID>/paused/<string:paused>`	`/api/v1/dags/{dag_id}`
Information of paused DAG(GET)	`/api/experimental/dags/<DAG_ID>/paused`	`/api/v1/dags/{dag_id}`
Latest DAG Runs(GET)	`/api/experimental/latest_runs`	`/api/v1/dags/{dag_id}/dagRuns`
Get all pools(GET)	`/api/experimental/pools`	`/api/v1/pools`
Create a pool(POST)	`/api/experimental/pools`	`/api/v1/pools`
Delete a pool(DELETE)	`/api/experimental/pools/<string:name>`	`/api/v1/pools/{pool_name}`
DAG Lineage(GET)	`/api/experimental/lineage/<DAG_ID>/<string:execution_date>/`	`/api/v1/dags/{dag_id}/dagRuns/{dag_run_id}/taskInstances/{task_id}/xcomEntries`

This endpoint `/api/v1/dags/{dag_id}/dagRuns` also allows you to filter dag_runs with parameters such as `start_date`, `end_date`, `execution_date` etc in the query string. Therefore the operation previously performed by this endpoint:

```
/api/experimental/dags/<string:dag_id>/dag_runs/<string:execution_date>
```

can now be handled with filter parameters in the query string. Getting information about latest runs can be accomplished with the help of filters in the query string of this endpoint(`/api/v1/dags/{dag_id}/dagRuns`). Please check the Stable API reference documentation for more information

Changes to Exception handling for from DAG callbacks

Exception from DAG callbacks used to crash the Airflow Scheduler. As part of our efforts to make the Scheduler more performant and reliable, we have changed this behavior to log the exception instead. On top of that, a new dag.callback_exceptions counter metric has been added to help better monitor callback exceptions.

Airflow CLI changes in 2.0

The Airflow CLI has been organized so that related commands are grouped together as subcommands, which means that if you use these commands in your scripts, you have to make changes to them.

This section describes the changes that have been made, and what you need to do to update your scripts. The ability to manipulate users from the command line has been changed. `airflow create_user`, `airflow delete_user` and `airflow list_users` has been grouped to a single command `airflow users` with optional flags `create`, `list` and `delete`. The `airflow list_dags` command is now `airflow dags list`, `airflow pause` is `airflow dags pause`, etc.

In Airflow 1.10 and 2.0 there is an `airflow config` command but there is a difference in behavior. In Airflow 1.10, it prints all config options while in Airflow 2.0, it's a command group. `airflow config` is now `airflow config list`. You can check other options by running the command `airflow config --help`

For a complete list of updated CLI commands, see https://airflow.apache.org/cli.html.

You can learn about the commands by running `airflow --help`. For example to get help about the `celery` group command, you have to run the help command: `airflow celery --help`.

Old command	New command	Group
`airflow worker`	`airflow celery worker`	`celery`
`airflow flower`	`airflow celery flower`	`celery`
`airflow trigger_dag`	`airflow dags trigger`	`dags`
`airflow delete_dag`	`airflow dags delete`	`dags`
`airflow show_dag`	`airflow dags show`	`dags`
`airflow list_dag`	`airflow dags list`	`dags`
`airflow dag_status`	`airflow dags status`	`dags`
`airflow backfill`	`airflow dags backfill`	`dags`
`airflow list_dag_runs`	`airflow dags list-runs`	`dags`
`airflow pause`	`airflow dags pause`	`dags`
`airflow unpause`	`airflow dags unpause`	`dags`
`airflow next_execution`	`airflow dags next-execution`	`dags`
`airflow test`	`airflow tasks test`	`tasks`
`airflow clear`	`airflow tasks clear`	`tasks`
`airflow list_tasks`	`airflow tasks list`	`tasks`
`airflow task_failed_deps`	`airflow tasks failed-deps`	`tasks`
`airflow task_state`	`airflow tasks state`	`tasks`
`airflow run`	`airflow tasks run`	`tasks`
`airflow render`	`airflow tasks render`	`tasks`
`airflow initdb`	`airflow db init`	`db`
`airflow resetdb`	`airflow db reset`	`db`
`airflow upgradedb`	`airflow db upgrade`	`db`
`airflow checkdb`	`airflow db check`	`db`
`airflow shell`	`airflow db shell`	`db`
`airflow pool`	`airflow pools`	`pools`
`airflow create_user`	`airflow users create`	`users`
`airflow delete_user`	`airflow users delete`	`users`

Old command	New command	Group
airflow list_users	airflow users list	users
airflow rotate_fernet_key	airflow rotate-fernet-key	
airflow sync_perm	airflow sync-perm	

Example Usage for the ``users`` group

To create a new user:

```
airflow users create --username jondoe --lastname doe --firstname jon --email jdoe@apache.org --role Viewer --password test
```

To list users:

```
airflow users list
```

To delete a user:

```
airflow users delete --username jondoe
```

To add a user to a role:

```
airflow users add-role --username jondoe --role Public
```

To remove a user from a role:

```
airflow users remove-role --username jondoe --role Public
```

Use exactly single character for short option style change in CLI

For Airflow short option, use exactly one single character. New commands are available according to the following table:

Old command	New command
airflow (dags|tasks|scheduler) [-sd, --subdir]	airflow (dags|tasks|scheduler) [-S, --subdir]
airflow test [-dr, --dry_run]	airflow tasks test [-n, --dry-run]
airflow test [-tp, --task_params]	airflow tasks test [-t, --task-params]
airflow test [-pm, --post_mortem]	airflow tasks test [-m, --post-mortem]
airflow run [-int, --interactive]	airflow tasks run [-N, --interactive]
airflow backfill [-dr, --dry_run]	airflow dags backfill [-n, --dry-run]
airflow clear [-dx, --dag_regex]	airflow tasks clear [-R, --dag-regex]
airflow kerberos [-kt, --keytab]	airflow kerberos [-k, --keytab]

Old command	New command
`airflow webserver [-hn, --hostname]`	`airflow webserver [-H, --hostname]`
`airflow worker [-cn, --celery_hostname]`	`airflow celery worker [-H, --celery-hostname]`
`airflow flower [-hn, --hostname]`	`airflow celery flower [-H, --hostname]`
`airflow flower [-fc, --flower_conf]`	`airflow celery flower [-c, --flower-conf]`
`airflow flower [-ba, --basic_auth]`	`airflow celery flower [-A, --basic-auth]`

For Airflow long option, use [kebab-case](https://en.wikipedia.org/wiki/Letter_case) instead of [snake_case](https://en.wikipedia.org/wiki/Snake_case)

Old option	New option
`--task_regex`	`--task-regex`
`--start_date`	`--start-date`
`--end_date`	`--end-date`
`--dry_run`	`--dry-run`
`--no_backfill`	`--no-backfill`
`--mark_success`	`--mark-success`
`--donot_pickle`	`--donot-pickle`
`--ignore_dependencies`	`--ignore-dependencies`
`--ignore_first_depends_on_past`	`--ignore-first-depends-on-past`
`--delay_on_limit`	`--delay-on-limit`
`--reset_dagruns`	`--reset-dagruns`
`--rerun_failed_tasks`	`--rerun-failed-tasks`
`--run_backwards`	`--run-backwards`
`--only_failed`	`--only-failed`
`--only_running`	`--only-running`
`--exclude_subdags`	`--exclude-subdags`
`--exclude_parentdag`	`--exclude-parentdag`
`--dag_regex`	`--dag-regex`
`--run_id`	`--run-id`
`--exec_date`	`--exec-date`
`--ignore_all_dependencies`	`--ignore-all-dependencies`
`--ignore_depends_on_past`	`--ignore-depends-on-past`
`--ship_dag`	`--ship-dag`
`--job_id`	`--job-id`
`--cfg_path`	`--cfg-path`
`--ssl_cert`	`--ssl-cert`

Old option	New option
`--ssl_key`	`--ssl-key`
`--worker_timeout`	`--worker-timeout`
`--access_logfile`	`--access-logfile`
`--error_logfile`	`--error-logfile`
`--dag_id`	`--dag-id`
`--num_runs`	`--num-runs`
`--do_pickle`	`--do-pickle`
`--celery_hostname`	`--celery-hostname`
`--broker_api`	`--broker-api`
`--flower_conf`	`--flower-conf`
`--url_prefix`	`--url-prefix`
`--basic_auth`	`--basic-auth`
`--task_params`	`--task-params`
`--post_mortem`	`--post-mortem`
`--conn_uri`	`--conn-uri`
`--conn_type`	`--conn-type`
`--conn_host`	`--conn-host`
`--conn_login`	`--conn-login`
`--conn_password`	`--conn-password`
`--conn_schema`	`--conn-schema`
`--conn_port`	`--conn-port`
`--conn_extra`	`--conn-extra`
`--use_random_password`	`--use-random-password`
`--skip_serve_logs`	`--skip-serve-logs`

Remove serve_logs command from CLI

The `serve_logs` command has been deleted. This command should be run only by internal application mechanisms and there is no need for it to be accessible from the CLI interface.

dag_state CLI command

If the DAGRun was triggered with conf key/values passed in, they will also be printed in the dag_state CLI response ie. running, {"name": "bob"} whereas in in prior releases it just printed the state: ie. running

Deprecating ignore_first_depends_on_past on backfill command and default it to True

When doing backfill with `depends_on_past` dags, users will need to pass `--ignore-first-depends-on-past`. We should default it as `true` to avoid confusion

Changes to Airflow Plugins

If you are using Airflow Plugins and were passing `admin_views` & `menu_links` which were used in the non-RBAC UI (`flask-admin` based UI), update it to use `flask_appbuilder_views` and `flask_appbuilder_menu_links` .

Old:

```
from airflow.plugins_manager import AirflowPlugin

from flask_admin import BaseView, expose
from flask_admin.base import MenuLink


class TestView(BaseView):
    @expose('/')
    def test(self):
        # in this example, put your test_plugin/test.html template at airflow/plugins/templates/test_plugin/test.html
        return self.render("test_plugin/test.html", content="Hello galaxy!")

v = TestView(category="Test Plugin", name="Test View")

ml = MenuLink(
    category='Test Plugin',
    name='Test Menu Link',
    url='https://airflow.apache.org/'
)

class AirflowTestPlugin(AirflowPlugin):
    admin_views = [v]
    menu_links = [ml]
```

Change it to:

```
from airflow.plugins_manager import AirflowPlugin
from flask_appbuilder import expose, BaseView as AppBuilderBaseView

class TestAppBuilderBaseView(AppBuilderBaseView):
    default_view = "test"

    @expose("/")
    def test(self):
        return self.render("test_plugin/test.html", content="Hello galaxy!")

v_appbuilder_view = TestAppBuilderBaseView()
v_appbuilder_package = {"name": "Test View",
                        "category": "Test Plugin",
                        "view": v_appbuilder_view}

# Creating a flask appbuilder Menu Item
appbuilder_mitem = {"name": "Google",
                    "category": "Search",
                    "category_icon": "fa-th",
                    "href": "https://www.google.com"}


# Defining the plugin class
class AirflowTestPlugin(AirflowPlugin):
    name = "test_plugin"
    appbuilder_views = [v_appbuilder_package]
    appbuilder_menu_items = [appbuilder_mitem]
```

Changes to extras names

The `all` extra were reduced to include only user-facing dependencies. This means that this extra does not contain development dependencies. If you were using it and depending on the development packages then you should use `devel_all` .

Support for Airflow 1.10.x releases

The Airflow 1.10.x release tree will be supported for **six months** from Airflow 2.0 release date. Specifically, only “critical fixes” defined as fixes to bugs that take down Production systems, will be backported to 1.10.x core for **six months** after Airflow 2.0.0 is released.

In addition, Backport providers within 1.10.x, will be supported for critical fixes for **three months** from Airflow 2.0.0 release date.

We plan to take a strict Semantic Versioning approach to our versioning and release process. This means that we do not plan to make any backwards-incompatible changes in the 2.* releases. Any breaking changes, including the removal of features deprecated in Airflow 2.0 will happen as part of the Airflow 3.0 release.

Previous Next

Was this entry helpful?