

Plugins

Airflow has a simple plugin manager built-in that can integrate external features to its core by simply dropping files in your `$AIRFLOW_HOME/plugins` folder.

The python modules in the `plugins` folder get imported, and **macros** and web **views** get integrated to Airflow's main collections and become available for use.

To troubleshoot issue with plugins, you can use `airflow plugins` command. This command dumps information about loaded plugins.

Changed in version 2.0: Importing operators, sensors, hooks added in plugins via `airflow.{operators,sensors,hooks}.<plugin_name>` is no longer supported, and these extensions should just be imported as regular python modules. For more information, see: [Modules Management](#) and [Creating a custom Operator](#)

What for?

Airflow offers a generic toolbox for working with data. Different organizations have different stacks and different needs. Using Airflow plugins can be a way for companies to customize their Airflow installation to reflect their ecosystem.

Plugins can be used as an easy way to write, share and activate new sets of features.

There's also a need for a set of more complex applications to interact with different flavors of data and metadata.

Examples:

- A set of tools to parse Hive logs and expose Hive metadata (CPU /IO / phases/ skew /...)
- An anomaly detection framework, allowing people to collect metrics, set thresholds and alerts
- An auditing tool, helping understand who accesses what
- A config-driven SLA monitoring tool, allowing you to set monitored tables and at what time they should land, alert people, and expose visualizations of outages
- ...

Why build on top of Airflow?

Airflow has many components that can be reused when building an application:

- A web server you can use to render your views
- A metadata database to store your models
- Access to your databases, and knowledge of how to connect to them
- An array of workers that your application can push workload to
- Airflow is deployed, you can just piggy back on its deployment logistics
- Basic charting capabilities, underlying libraries and abstractions

When are plugins (re)loaded?

Plugins are by default lazily loaded and once loaded, they are never reloaded (except the UI plugins are automatically loaded in Webserver). To load them at the start of each Airflow process, set `[core] lazy_load_plugins = False` in `airflow.cfg`.

This means that if you make any changes to plugins and you want the webserver or scheduler to use that new code you will need to restart those processes.

By default, task execution will use forking to avoid the slow down of having to create a whole new python interpreter and re-parse all of the Airflow code and start up routines – this is a big benefit for shorter running tasks. This does mean that if you use plugins in your tasks, and want them to update you will either need to restart the worker (if using CeleryExecutor) or scheduler (Local or Sequential executors). The other option is you can accept the speed hit at start up set the `core.execute_tasks_new_python_interpreter` config setting to True, resulting in launching a whole new python interpreter for tasks.

(Modules only imported by DAG files on the other hand do not suffer this problem, as DAG files are not loaded/parsed in any long-running Airflow process.)

Interface

To create a plugin you will need to derive the `airflow.plugins_manager.AirflowPlugin` class and reference the objects you want to plug into Airflow. Here's what the class you need to derive looks like:

```
class AirflowPlugin:
    # The name of your plugin (str)
    name = None
    # A list of class(es) derived from BaseHook
    hooks = []
    # A list of references to inject into the macros namespace
    macros = []
    # A list of Blueprint object created from flask.Blueprint. For use with the flask_appbuilder based GUI
    flask_blueprints = []
    # A list of dictionaries containing FlaskAppBuilder BaseView object and some metadata. See example below
    appbuilder_views = []
    # A list of dictionaries containing FlaskAppBuilder BaseView object and some metadata. See example below
    appbuilder_menu_items = []
    # A callback to perform actions when airflow starts and the plugin is loaded.
    # NOTE: Ensure your plugin has *args, and **kwargs in the method definition
    # to protect against extra parameters injected into the on_load(...)
    # function in future changes
    def on_load(*args, **kwargs):
        # ... perform Plugin boot actions
        pass

    # A list of global operator extra links that can redirect users to
    # external systems. These extra links will be available on the
    # task page in the form of buttons.
    #
    # Note: the global operator extra link can be overridden at each
    # operator level.
    global_operator_extra_links = []

    # A list of operator extra links to override or add operator links
    # to existing Airflow Operators.
    # These extra links will be available on the task page in form of
    # buttons.
    operator_extra_links = []
```

You can derive it by inheritance (please refer to the example below). In the example, all options have been defined as class attributes, but you can also define them as properties if you need to perform additional initialization. Please note `name` inside this class must be specified.

Make sure you restart the webserver and scheduler after making changes to plugins so that they take effect.

Example

The code below defines a plugin that injects a set of dummy object definitions in Airflow.

```
# This is the class you derive to create a plugin
from airflow.plugins_manager import AirflowPlugin

from flask import Blueprint
from flask_appbuilder import expose, BaseView as AppBuilderBaseView

# Importing base classes that we need to derive
from airflow.hooks.base import BaseHook
from airflow.models.baseoperator import BaseOperatorLink
from airflow.providers.amazon.aws.transfers.gcs_to_s3 import GCSToS3Operator

# Will show up in Connections screen in a future version
class PluginHook(BaseHook):
    pass

# Will show up under airflow.macros.test_plugin.plugin_macro
# and in templates through {{ macros.test_plugin.plugin_macro }}
```

```

# and in templates through {{ macros.test_plugin.plugin_macro }}
def plugin_macro():
    pass

# Creating a flask blueprint to integrate the templates and static folder
bp = Blueprint(
    "test_plugin", __name__,
    template_folder='templates', # registers airflow/plugins/templates as a Jinja template folder
    static_folder='static',
    static_url_path='/static/test_plugin')

# Creating a flask appbuilder BaseView
class TestAppBuilderBaseView(AppBuilderBaseView):
    default_view = "test"

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

# Creating a flask appbuilder BaseView
class TestAppBuilderBaseNoMenuView(AppBuilderBaseView):
    default_view = "test"

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

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

v_appbuilder_nomenu_view = TestAppBuilderBaseNoMenuView()
v_appbuilder_nomenu_package = {
    "view": v_appbuilder_nomenu_view
}

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

# A global operator extra link that redirect you to
# task logs stored in S3
class GoogleLink(BaseOperatorLink):
    name = "Google"

    def get_link(self, operator, dtm):
        return "https://www.google.com"

# A list of operator extra links to override or add operator links
# to existing Airflow Operators.
# These extra links will be available on the task page in form of
# buttons.
class S3LogLink(BaseOperatorLink):
    name = 'S3'
    operators = [GCSToS3Operator]

    def get_link(self, operator, dtm):
        return 'https://s3.amazonaws.com/airflow-logs/{dag_id}/{task_id}/{execution_date}'.format(
            dag_id=operator.dag_id,
            task_id=operator.task_id,
            execution_date=dtm,
        )

# Defining the plugin class
class AirflowTestPlugin(AirflowPlugin):
    name = "test_plugin"
    hooks = [PluginHook]
    macros = [plugin_macro]
    flask_blueprints = [bp]
    appbuilder_views = [v_appbuilder_package, v_appbuilder_nomenu_package]
    appbuilder_menu_items = [appbuilder_mitem]

```

```
global_operator_extra_links = [GoogleLink(),]
operator_extra_links = [S3LogLink(), ]
```

Note on role based views

Airflow 1.10 introduced role based views using FlaskAppBuilder. You can configure which UI is used by setting `rbac = True`. To support plugin views and links for both versions of the UI and maintain backwards compatibility, the fields `appbuilder_views` and `appbuilder_menu_items` were added to the `AirflowTestPlugin` class.

`appbuilder_views` supports both views-with-menu and views-without-menu - to add a view with menu link, add a "name" key in view's package dictionary, otherwise the view is added to flask appbuilder without menu item.

Exclude views from CSRF protection

We strongly suggest that you should protect all your views with CSRF. But if needed, you can exclude some views using a decorator.

```
from airflow.www.app import csrf

@csrf.exempt
def my_handler():
    # ...
    return 'ok'
```

Plugins as Python packages

It is possible to load plugins via [setuptools entrypoint](#) mechanism. To do this link your plugin using an entrypoint in your package. If the package is installed, airflow will automatically load the registered plugins from the entrypoint list.

Note

Neither the entrypoint name (eg, `my_plugin`) nor the name of the plugin class will contribute towards the module and class name of the plugin itself.

```
# my_package/my_plugin.py
from airflow.plugins_manager import AirflowPlugin
from flask import Blueprint

# Creating a flask blueprint to integrate the templates and static folder
bp = Blueprint(
    "test_plugin", __name__,
    template_folder='templates', # registers airflow/plugins/templates as a Jinja template folder
    static_folder='static',
    static_url_path='/static/test_plugin')


class MyAirflowPlugin(AirflowPlugin):
    name = 'my_namespace'
    flask_blueprints = [bp]
```

```
from setuptools import setup


setup(
    name="my-package",
    ...
    entry_points = {
        'airflow.plugins': [
            'my_plugin = my_package.my_plugin:MyAirflowPlugin'
        ]
    }
)
```

Automatic reloading webserver

To enable automatic reloading of the webserver, when changes in a directory with plugins has been detected, you should set `reload_on_plugin_change` option in `[webserver]` section to `True`.

 Note

For more information on setting the configuration, see [Setting Configuration Options](#)

 Note

See [Modules Management](#) for details on how Python and Airflow manage modules.

Troubleshooting

You can use [the Flask CLI](#) to troubleshoot problems. To run this, you need to set the variable `FLASK_APP` to `airflow.www.app:create_app`.

For example, to print all routes, run:

```
FLASK_APP=airflow.www.app:create_app flask routes
```

[Previous](#)

[Next](#)

Was this entry helpful?



Want to be a part of Apache Airflow?

Join community

License Donate Thanks

Security

© 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.