Airflow Documentation

Release 1.10.3

Apache Airflow

Jun 20, 2019

CONTENTS

1 Principles **3**

2 Beyond the Horizon **5**

3 Content **7**

3.1 Project . . . 7
3.1.1 History . . . 7
3.1.2 Committers . . . 7
3.1.3 Resources & links . . . 8
3.1.4 Roadmap . . . 8
3.2 License . . . 8
3.3 Quick Start . . . 11
3.3.1 What's Next? . . . 12
3.4 Installation . . . 12
3.4.1 Getting Airflow . . . 12
3.4.2 Extra Packages . . . 13
3.4.3 Initiating Airflow Database . . . 14
3.5 Tutorial . . . 14
3.5.1 Example Pipeline definition . . . 14
3.5.2 It's a DAG definition file . . . 15
3.5.3 Importing Modules . . . 16
3.5.4 Default Arguments . . . 16
3.5.5 Instantiate a DAG . . . 16
3.5.6 Tasks . . . 16
3.5.7 Templating with Jinja . . . 17
3.5.8 Setting up Dependencies . . . 18
3.5.9 Recap . . . 18
3.5.10 Testing . . . 19
3.5.10.1 Running the Script . . . 19
3.5.10.2 Command Line Metadata Validation . . . 20
3.5.10.3 Testing . . . 20
3.5.10.4 Backfill . . . 20
3.5.11 What's Next? . . . 21
3.6 How-to Guides . . . 21
3.6.1 Add a new role in RBAC UI . . . 21
3.6.2 Setting Configuration Options . . . 22
3.6.3 Initializing a Database Backend . . . 23
3.6.4 Using Operators . . . 24
3.6.4.1 BashOperator . . . 24
3.6.4.2 Dingding Operators . . . 24

3.6.4.3 Google Cloud Operators . . . 26
3.6.4.4 Papermill . . . 104
3.6.4.5 PythonOperator . . . 105
3.6.5 Managing Connections . . . 106
3.6.5.1 Creating a Connection with the UI . . . 107
3.6.5.2 Editing a Connection with the UI . . . 108
3.6.5.3 Creating a Connection with Environment Variables . . . 108
3.6.5.4 Connection Types . . . 108
3.6.6 Securing Connections . . . 116
3.6.7 Rotating encryption keys . . . 116
3.6.8 Writing Logs . . . 117
3.6.8.1 Writing Logs Locally . . . 117
3.6.8.2 Writing Logs to Amazon S3 . . . 117
3.6.8.3 Writing Logs to Azure Blob Storage . . . 117
3.6.8.4 Writing Logs to Google Cloud Storage . . . 118
3.6.8.5 Writing Logs to Elasticsearch . . . 119
3.6.9 Scaling Out with Celery . . . 119
3.6.10 Scaling Out with Dask . . . 120
3.6.11 Running Airflow behind a reverse proxy . . . 121
3.6.12 Running Airflow with systemd . . . 122
3.6.13 Running Airflow with upstart . . . 122
3.6.14 Using the Test Mode Configuration . . . 122
3.6.15 Checking Airflow Health Status . . . 123
3.6.16 Define an operator extra link . . . 123
3.6.17 Setup Test Environment using MySQL . . . 125
3.7 UI / Screenshots . . . 125
3.7.1 DAGs View . . . 125
3.7.2 Tree View . . . 126
3.7.3 Graph View . . . 126
3.7.4 Variable View . . . 126
3.7.5 Gantt Chart . . . 127
3.7.6 Task Duration . . . 128
3.7.7 Code View . . . 128
3.7.8 Task Instance Context Menu . . . 129
3.8 Concepts . . . 129
3.8.1 Core Ideas . . . 130
3.8.1.1 DAGs . . . 130
3.8.1.2 Operators . . . 131
3.8.1.3 Tasks . . . 133
3.8.1.4 Task Instances . . . 133
3.8.1.5 Task Lifecycle . . . 133
3.8.1.6 Workflows . . . 133
3.8.2 Additional Functionality . . . 134
3.8.2.1 Hooks . . . 134
3.8.2.2 Pools . . . 134
3.8.2.3 Connections . . . 135
3.8.2.4 Queues . . . 135
3.8.2.5 XComs . . . 135
3.8.2.6 Variables . . . 136
3.8.2.7 Branching . . . 136
3.8.2.8 SubDAGs . . . 137
3.8.2.9 SLAs . . . 140
3.8.2.10 Trigger Rules . . . 140
3.8.2.11 Latest Run Only . . . 142

3.8.2.12 Zombies & Undeads . . . 143
3.8.2.13 Cluster Policy . . . 143
3.8.2.14 Documentation & Notes . . . 144
3.8.2.15 Jinja Templating . . . 144
3.8.3 Packaged DAGs . . . 145
3.8.4 .airflowignore . . . 145
3.9 Command Line Interface . . . 146
3.9.1 Positional Arguments . . . 147
3.9.2 Sub-commands: . . . 147
3.9.2.1 backfill . . . 147
3.9.2.2 list_dag_runs . . . 149
3.9.2.3 list_tasks . . . 149
3.9.2.4 clear . . . 149
3.9.2.5 pause . . . 150
3.9.2.6 unpause . . . 151
3.9.2.7 trigger_dag . . . 151
3.9.2.8 delete_dag . . . 152
3.9.2.9 pool . . . 152
3.9.2.10 variables . . . 152
3.9.2.11 kerberos . . . 153
3.9.2.12 render . . . 153
3.9.2.13 run . . . 154
3.9.2.14 initdb . . . 155
3.9.2.15 list_dags . . . 155
3.9.2.16 dag_state . . . 155
3.9.2.17 task_failed_deps . . . 156
3.9.2.18 task_state . . . 156
3.9.2.19 serve_logs . . . 156
3.9.2.20 test . . . 157
3.9.2.21 webserver . . . 157
3.9.2.22 resetdb . . . 158
3.9.2.23 upgradedb . . . 158
3.9.2.24 scheduler . . . 158
3.9.2.25 worker . . . 159
3.9.2.26 flower . . . 160
3.9.2.27 version . . . 160
3.9.2.28 connections . . . 160
3.9.2.29 create_user . . . 161
3.9.2.30 delete_user . . . 161
3.9.2.31 list_users . . . 162
3.9.2.32 sync_perm . . . 162
3.9.2.33 next_execution . . . 162
3.9.2.34 rotate_fernet_key . . . 162
3.10 Scheduling & Triggers . . . 162
3.10.1 DAG Runs . . . 163
3.10.2 Backfill and Catchup . . . 163
3.10.3 External Triggers . . . 164
3.10.4 To Keep in Mind . . . 164
3.11 Plugins . . . 165
3.11.1 What for? . . . 165
3.11.2 Why build on top of Airflow? . . . 165
3.11.3 Interface . . . 166
3.11.4 Example . . . 167
3.11.5 Note on role based views . . . 168

3.11.6 Plugins as Python packages . . . 169
3.12 Security . . . 169
3.12.1 Reporting Vulnerabilities . . . 169
3.12.2 Web Authentication . . . 170
3.12.2.1 Password . . . 170
3.12.2.2 LDAP . . . 170
3.12.2.3 Roll your own . . . 171
3.12.3 Kerberos . . . 171
3.12.3.1 Limitations . . . 171
3.12.3.2 Enabling kerberos . . . 172
3.12.3.3 Using kerberos authentication . . . 172
3.12.4 OAuth Authentication . . . 173
3.12.4.1 GitHub Enterprise (GHE) Authentication . . . 173
3.12.4.2 Google Authentication . . . 174
3.12.5 SSL . . . 175
3.12.6 Impersonation . . . 175
3.12.6.1 Default Impersonation . . . 175
3.12.7 Flower Authentication . . . 175
3.12.8 RBAC UI Security . . . 176
3.12.8.1 Default Roles . . . 176
3.12.8.2 Custom Roles . . . 178
3.13 Time zones . . . 178
3.13.1 Concepts . . . 179
3.13.1.1 Naïve and aware datetime objects . . . 179
3.13.1.2 Interpretation of naive datetime objects . . . 179
3.13.1.3 Default time zone . . . 179
3.13.2 Time zone aware DAGs . . . 180
3.13.2.1 Templates . . . 180
3.13.2.2 Cron schedules . . . 180
3.13.2.3 Time deltas . . . 180
3.14 Experimental Rest API . . . 180
3.14.1 Endpoints . . . 181
3.14.2 CLI . . . 181
3.14.3 Authentication . . . 181
3.15 Integration . . . 182
3.15.1 Azure: Microsoft Azure . . . 182
3.15.1.1 Azure Blob Storage . . . 182
3.15.1.2 Azure File Share . . . 183
3.15.1.3 Logging . . . 183
3.15.1.4 Azure CosmosDB . . . 183
3.15.1.5 Azure Data Lake . . . 183
3.15.1.6 Azure Container Instances . . . 183
3.15.2 AWS: Amazon Web Services . . . 184
3.15.2.1 AWS EMR . . . 184
3.15.2.2 AWS S3 . . . 184
3.15.2.3 AWS Batch Service . . . 184
3.15.2.4 AWS RedShift . . . 185
3.15.2.5 AWS DynamoDB . . . 185
3.15.2.6 AWS Lambda . . . 185
3.15.2.7 AWS Kinesis . . . 185
3.15.2.8 Amazon SageMaker . . . 185
3.15.3 Databricks . . . 186
3.15.4 GCP: Google Cloud Platform . . . 186
3.15.4.1 Logging . . . 186

3.15.4.2 GoogleCloudBaseHook . . . 186
3.15.4.3 BigQuery . . . 186
3.15.4.4 Cloud Spanner . . . 187
3.15.4.5 Cloud SQL . . . 187
3.15.4.6 Cloud Bigtable . . . 188
3.15.4.7 Cloud Build . . . 188
3.15.4.8 Compute Engine . . . 188
3.15.4.9 Cloud Functions . . . 188
3.15.4.10 Cloud DataFlow . . . 189
3.15.4.11 Cloud DataProc . . . 189
3.15.4.12 Cloud Datastore . . . 189
3.15.4.13 Cloud ML Engine . . . 190
3.15.4.14 Cloud Storage . . . 190
3.15.4.15 Transfer Service . . . 191
3.15.4.16 Cloud Vision . . . 191
3.15.4.17 Cloud Text to Speech . . . 192
3.15.4.18 Cloud Speech to Text . . . 192
3.15.5 Cloud Speech Translate Operators . . . 193
3.15.5.1 Cloud Translate . . . 193
3.15.5.2 Cloud Video Intelligence . . . 193
3.15.5.3 Google Kubernetes Engine . . . 193
3.15.5.4 Google Natural Language . . . 193
3.15.6 Qubole . . . 194
3.16 Metrics . . . 194
3.16.1 Setup . . . 194
3.16.2 Counters . . . 195
3.16.3 Gauges . . . 195
3.16.4 Timers . . . 195
3.17 Kubernetes . . . 195
3.17.1 Kubernetes Executor . . . 195
3.17.2 Kubernetes Operator . . . 196
3.17.3 Pod Mutation Hook . . . 198
3.18 Lineage . . . 198
3.18.1 Apache Atlas . . . 199
3.19 Changelog . . . 200
3.19.1 Airflow 1.10.3, 2019-04-09 . . . 200
3.19.1.1 New Feature . . . 200
3.19.1.2 Improvement . . . 200
3.19.1.3 Bug fixes . . . 203
3.19.1.4 Misc/Internal . . . 206
3.19.1.5 Doc-only changes . . . 207
3.19.2 Airflow 1.10.2, 2019-01-19 . . . 208
3.19.2.1 New features . . . 208
3.19.2.2 Improvements . . . 209
3.19.2.3 Bug fixes . . . 212
3.19.2.4 Doc-only changes . . . 214
3.19.3 Airflow 1.10.1, 2018-11-13 . . . 215
3.19.3.1 New features . . . 215
3.19.3.2 Improvements . . . 215
3.19.3.3 Doc-only changes . . . 216
3.19.3.4 Bug fixes . . . 217
3.19.4 Airflow 1.10.0, 2018-08-03 . . . 218
3.19.5 Airflow 1.9.0, 2018-01-02 . . . 239
3.19.6 Airflow 1.8.2, 2017-09-04 . . . 251

3.19.7 Airflow 1.8.1, 2017-05-09 252
3.19.8 Airflow 1.8.0, 2017-03-12 253
3.19.9 Airflow 1.7.2 258
3.19.10 Airflow 1.7.1, 2016-05-19 263
3.20 FAQ 268
3.20.1 Why isn't my task getting scheduled? 268
3.20.2 How do I trigger tasks based on another task's failure? 269
3.20.3 Why are connection passwords still not encrypted in the metadata db after I installed airflow[crypto]? 269
3.20.4 What's the deal with `start_date`? 269
3.20.5 How can I create DAGs dynamically? 270
3.20.6 What are all the `airflow run` commands in my process list? 270
3.20.7 How can my airflow dag run faster? 271
3.20.8 How can we reduce the airflow UI page load time? 271
3.20.9 How to fix Exception: Global variable explicit_defaults_for_timestamp needs to be on (1)? . . 271
3.20.10 How to reduce airflow dag scheduling latency in production? 271
3.21 Macros reference 272
3.21.1 Default Variables 272
3.21.2 Macros 273
3.22 API Reference 274
3.22.1 Operators 274
3.22.1.1 BaseOperator 274
3.22.1.2 BaseSensorOperator 275
3.22.1.3 Operators packages 275
3.22.2 Hooks 513
3.22.2.1 Hooks packages 513
3.22.3 Executors 644
3.22.3.1 Executors packages 644
3.22.4 Models 653
3.22.4.1 `airflow.models` 653
3.22.5 Core and community package 706

Python Module Index **707**

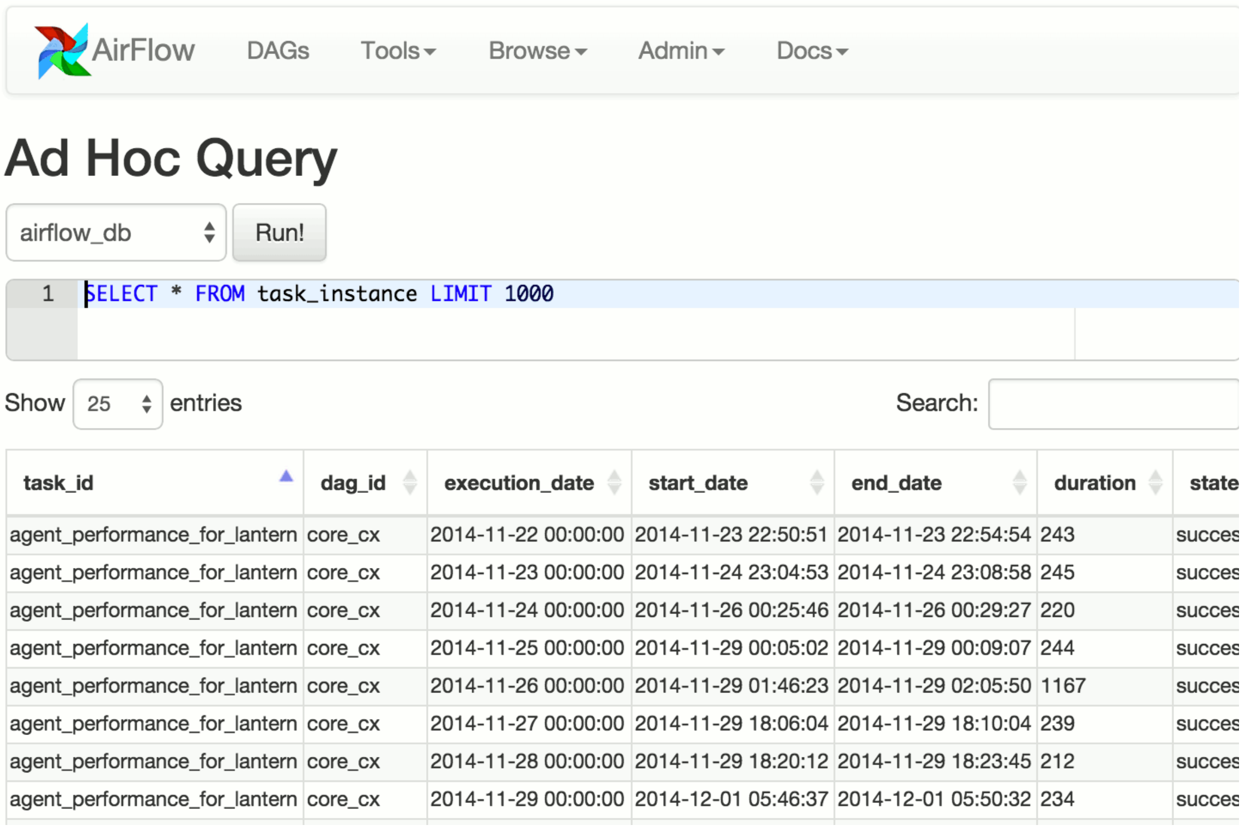
HTTP Routing Table **713**

Index **715**

Airflow is a platform to programmatically author, schedule and monitor workflows.

Use Airflow to author workflows as Directed Acyclic Graphs (DAGs) of tasks. The Airflow scheduler executes your tasks on an array of workers while following the specified dependencies. Rich command line utilities make performing complex surgeries on DAGs a snap. The rich user interface makes it easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed.

When workflows are defined as code, they become more maintainable, versionable, testable, and collaborative.



```
SELECT * FROM task_instance LIMIT 1000
```

task_id	dag_id	execution_date	start_date	end_date	duration	state
agent_performance_for_lantern	core_cx	2014-11-22 00:00:00	2014-11-23 22:50:51	2014-11-23 22:54:54	243	succes
agent_performance_for_lantern	core_cx	2014-11-23 00:00:00	2014-11-24 23:04:53	2014-11-24 23:08:58	245	succes
agent_performance_for_lantern	core_cx	2014-11-24 00:00:00	2014-11-26 00:25:46	2014-11-26 00:29:27	220	succes
agent_performance_for_lantern	core_cx	2014-11-25 00:00:00	2014-11-29 00:05:02	2014-11-29 00:09:07	244	succes
agent_performance_for_lantern	core_cx	2014-11-26 00:00:00	2014-11-29 01:46:23	2014-11-29 02:05:50	1167	succes
agent_performance_for_lantern	core_cx	2014-11-27 00:00:00	2014-11-29 18:06:04	2014-11-29 18:10:04	239	succes
agent_performance_for_lantern	core_cx	2014-11-28 00:00:00	2014-11-29 18:20:12	2014-11-29 18:23:45	212	succes
agent_performance_for_lantern	core_cx	2014-11-29 00:00:00	2014-12-01 05:46:37	2014-12-01 05:50:32	234	succes

CHAPTER

ONE

PRINCIPLES

- **Dynamic**: Airflow pipelines are configuration as code (Python), allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically.
- **Extensible**: Easily define your own operators, executors and extend the library so that it fits the level of abstraction that suits your environment.
- **Elegant**: Airflow pipelines are lean and explicit. Parameterizing your scripts is built into the core of Airflow using the powerful **Jinja** templating engine.
- **Scalable**: Airflow has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. Airflow is ready to scale to infinity.

CHAPTER

TWO

BEYOND THE HORIZON

Airflow **is not** a data streaming solution. Tasks do not move data from one to the other (though tasks can exchange metadata!). Airflow is not in the Spark Streaming or Storm space, it is more comparable to Oozie or Azkaban.

Workflows are expected to be mostly static or slowly changing. You can think of the structure of the tasks in your workflow as slightly more dynamic than a database structure would be. Airflow workflows are expected to look similar from a run to the next, this allows for clarity around unit of work and continuity.

CHAPTER THREE

CONTENT

3.1 Project

3.1.1 History

Airflow was started in October 2014 by Maxime Beauchemin at Airbnb. It was open source from the very first commit and officially brought under the Airbnb GitHub and announced in June 2015.

The project joined the Apache Software Foundation's Incubator program in March 2016 and the Foundation announced Apache Airflow as a Top-Level Project in January 2019.

3.1.2 Committers

- @mistercrunch (Maxime "Max" Beauchemin)
- @r39132 (Siddharth "Sid" Anand)
- @criccomini (Chris Riccomini)
- @bolkedebruin (Bolke de Bruin)
- @artwr (Arthur Wiedmer)
- @jlowin (Jeremiah Lowin)
- @aoen (Dan Davydov)
- @msumit (Sumit Maheshwari)
- @alexvanboxel (Alex Van Boxel)
- @saguziel (Alex Guziel)
- @joygao (Joy Gao)
- @fokko (Fokko Driesprong)
- @ash (Ash Berlin-Taylor)
- @kaxilnaik (Kaxil Naik)
- @feng-tao (Tao Feng)
- @hiteshs (Hitesh Shah)
- @jghoman (Jakob Homan)
- @XD-DENG (Xiaodong Deng)
- @dimberman (Daniel Imberman)
- @potiuk (Jarek Potiuk)

- @basph (Bas Harenslak)
- @jmcarp (Joshua Carp)
- @KevinYang21 (Kevin Yang)
- @mik-laj (Kamil Breguła)
- @aijamalnk (Aizhamal Nurmamat kyzy)

For the full list of contributors, take a look at Airflow's Github Contributor page:

3.1.3 Resources & links

- Airflow's official documentation
- Mailing lists:
  - Developer's mailing list: dev-subscribe@airflow.apache.org
  - All commits mailing list: commits-subscribe@airflow.apache.org
  - Airflow users mailing list: users-subscribe@airflow.apache.org
- Issues on Apache's Jira
- Slack (chat) Channel
- More resources and links to Airflow related content on the Wiki

3.1.4 Roadmap

Please refer to the Roadmap on the wiki

3.2 License

(continues on next page)

(continued from previous page)

```
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
```

(continues on next page)

(continued from previous page)

```
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(continues on next page)

(continued from previous page)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.
```

3.3 Quick Start

The installation is quick and straightforward.

```
# airflow needs a home, ~/airflow is the default,
# but you can lay foundation somewhere else if you prefer
# (optional)
export AIRFLOW_HOME=~/airflow

```

(continues on next page)

(continued from previous page)

```
# install from pypi using pip
pip install apache-airflow

# initialize the database
airflow initdb

# if you build with master
airflow users -c --username admin --firstname Peter --lastname Parker --role Admin --
↪email spiderman@superhero.org

# start the web server, default port is 8080
airflow webserver -p 8080

# start the scheduler
airflow scheduler

# visit localhost:8080 in the browser and use the admin account you just
# created to login. Enable the example dag in the home page
```

Upon running these commands, Airflow will create the `$AIRFLOW_HOME` folder and lay an "airflow.cfg" file with defaults that get you going fast. You can inspect the file either in `$AIRFLOW_HOME/airflow.cfg`, or through the UI in the `Admin->Configuration` menu. The PID file for the webserver will be stored in `$AIRFLOW_HOME/airflow-webserver.pid` or in `/run/airflow/webserver.pid` if started by systemd.

Out of the box, Airflow uses a sqlite database, which you should outgrow fairly quickly since no parallelization is possible using this database backend. It works in conjunction with the *`airflow.executors.sequential_executor.SequentialExecutor`* which will only run task instances sequentially. While this is very limiting, it allows you to get up and running quickly and take a tour of the UI and the command line utilities.

Here are a few commands that will trigger a few task instances. You should be able to see the status of the jobs change in the `example_bash_operator` DAG as you run the commands below.

```
# run your first task instance
airflow run example_bash_operator runme_0 2015-01-01
# run a backfill over 2 days
airflow backfill example_bash_operator -s 2015-01-01 -e 2015-01-02
```

3.3.1 What's Next?

From this point, you can head to the *Tutorial* section for further examples or the *How-to Guides* section if you're ready to get your hands dirty.

3.4 Installation

3.4.1 Getting Airflow

The easiest way to install the latest stable version of Airflow is with `pip`:

```
pip install apache-airflow
```

You can also install Airflow with support for extra features like `gcp` or `postgres`:

```
pip install 'apache-airflow[postgres,gcp]'
```

3.4.2 Extra Packages

The `apache-airflow` PyPI basic package only installs what's needed to get started. Subpackages can be installed depending on what will be useful in your environment. For instance, if you don't need connectivity with Postgres, you won't have to go through the trouble of installing the `postgres-devel` yum package, or whatever equivalent applies on the distribution you are using.

Behind the scenes, Airflow does conditional imports of operators that require these extra dependencies.

Here's the list of the subpackages and what they enable:

subpackage	install command	enables
all	`pip install 'apache-airflow[all]'`	All Airflow features known to man
all_dbs	`pip install 'apache-airflow[all_dbs]'`	All databases integrations
atlas	`pip install 'apache-airflow[atlas]'`	Apache Atlas to use Data Lineage featur
async	`pip install 'apache-airflow[async]'`	Async worker classes for Gunicorn
aws	`pip install 'apache-airflow[aws]'`	Amazon Web Services
azure	`pip install 'apache-airflow[azure]'`	Microsoft Azure
cassandra	`pip install 'apache-airflow[cassandra]'`	Cassandra related operators & hooks
celery	`pip install 'apache-airflow[celery]'`	CeleryExecutor
cgroups	`pip install 'apache-airflow[cgroups]'`	Needed To use CgroupTaskRunner
cloudant	`pip install 'apache-airflow[cloudant]'`	Cloudant hook
crypto	`pip install 'apache-airflow[crypto]'`	Encrypt connection passwords in metada
dask	`pip install 'apache-airflow[dask]'`	DaskExecutor
databricks	`pip install 'apache-airflow[databricks]'`	Databricks hooks and operators
datadog	`pip install 'apache-airflow[datadog]'`	Datadog hooks and sensors
devel	`pip install 'apache-airflow[devel]'`	Minimum dev tools requirements
devel_hadoop	`pip install 'apache-airflow[devel_hadoop]'`	Airflow + dependencies on the Hadoop s
doc	`pip install 'apache-airflow[doc]'`	Packages needed to build docs
docker	`pip install 'apache-airflow[docker]'`	Docker hooks and operators
druid	`pip install 'apache-airflow[druid]'`	Druid related operators & hooks
elasticsearch	`pip install 'apache-airflow[elasticsearch]'`	Elastic Log Handler
gcp	`pip install 'apache-airflow[gcp]'`	Google Cloud Platform
github_enterprise	`pip install 'apache-airflow[github_enterprise]'`	GitHub Enterprise auth backend
google_auth	`pip install 'apache-airflow[google_auth]'`	Google auth backend
grpc	`pip install 'apache-airflow[grpc]'`	Grpc hooks and operators
hdfs	`pip install 'apache-airflow[hdfs]'`	HDFS hooks and operators
hive	`pip install 'apache-airflow[hive]'`	All Hive related operators
jdbc	`pip install 'apache-airflow[jdbc]'`	JDBC hooks and operators
jira	`pip install 'apache-airflow[jira]'`	Jira hooks and operators
kerberos	`pip install 'apache-airflow[kerberos]'`	Kerberos integration for Kerberized Had
kubernetes	`pip install 'apache-airflow[kubernetes]'`	Kubernetes Executor and operator
ldap	`pip install 'apache-airflow[ldap]'`	LDAP authentication for users
mongo	`pip install 'apache-airflow[mongo]'`	Mongo hooks and operators
mssql	`pip install 'apache-airflow[mssql]'`	Microsoft SQL Server operators and hoo
mysql	`pip install 'apache-airflow[mysql]'`	MySQL operators and hook, support as a
oracle	`pip install 'apache-airflow[oracle]'`	Oracle hooks and operators
papermill	`pip install 'apache-airflow[papermill]'`	Papermill hooks and operators
password	`pip install 'apache-airflow[password]'`	Password authentication for users

subpackage	install command	enables
pinot	`pip install 'apache-airflow[pinot]'`	Pinot DB hook
postgres	`pip install 'apache-airflow[postgres]'`	PostgreSQL operators and hook, support
qds	`pip install 'apache-airflow[qds]'`	Enable QDS (Qubole Data Service) supp
rabbitmq	`pip install 'apache-airflow[rabbitmq]'`	RabbitMQ support as a Celery backend
redis	`pip install 'apache-airflow[redis]'`	Redis hooks and sensors
salesforce	`pip install 'apache-airflow[salesforce]'`	Salesforce hook
samba	`pip install 'apache-airflow[samba]'`	*airflow.operators.hive_to_s*
sendgrid	`pip install 'apache-airflow[sendgrid]'`	Send email using sendgrid
segment	`pip install 'apache-airflow[segment]'`	Segment hooks and sensors
slack	`pip install 'apache-airflow[slack]'`	*airflow.operators.slack_ope*
snowflake	`pip install 'apache-airflow[snowflake]'`	Snowflake hooks and operators
ssh	`pip install 'apache-airflow[ssh]'`	SSH hooks and Operator
statsd	`pip install 'apache-airflow[statsd]'`	Needed by StatsD metrics
vertica	`pip install 'apache-airflow[vertica]'`	Vertica hook support as an Airflow backe
webhdfs	`pip install 'apache-airflow[webhdfs]'`	HDFS hooks and operators
winrm	`pip install 'apache-airflow[winrm]'`	WinRM hooks and operators

3.4.3 Initiating Airflow Database

Airflow requires a database to be initiated before you can run tasks. If you're just experimenting and learning Airflow, you can stick with the default SQLite option. If you don't want to use SQLite, then take a look at *Initializing a Database Backend* to setup a different database.

After configuration, you'll need to initialize the database before you can run tasks:

```
airflow initdb
```

3.5 Tutorial

This tutorial walks you through some of the fundamental Airflow concepts, objects, and their usage while writing your first pipeline.

3.5.1 Example Pipeline definition

Here is an example of a basic pipeline definition. Do not worry if this looks complicated, a line by line explanation follows below.

```
"""
Code that goes along with the Airflow tutorial located at:
https://github.com/apache/airflow/blob/master/airflow/example_dags/tutorial.py
"""
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
```

(continues on next page)

(continued from previous page)

```
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}

dag = DAG('tutorial', default_args=default_args, schedule_interval=timedelta(days=1))

# t1, t2 and t3 are examples of tasks created by instantiating operators
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)

templated_command = """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)

t2.set_upstream(t1)
t3.set_upstream(t1)
```

3.5.2 It's a DAG definition file

One thing to wrap your head around (it may not be very intuitive for everyone at first) is that this Airflow Python script is really just a configuration file specifying the DAG's structure as code. The actual tasks defined here will run in a different context from the context of this script. Different tasks run on different workers at different points in time, which means that this script cannot be used to cross communicate between tasks. Note that for this purpose we have a more advanced feature called `XCom`.

People sometimes think of the DAG definition file as a place where they can do some actual data processing - that is not the case at all! The script's purpose is to define a DAG object. It needs to evaluate quickly (seconds, not minutes) since the scheduler will execute it periodically to reflect the changes if any.

3.5.3 Importing Modules

An Airflow pipeline is just a Python script that happens to define an Airflow DAG object. Let's start by importing the libraries we will need.

```
# The DAG object; we'll need this to instantiate a DAG
from airflow import DAG

# Operators; we need this to operate!
from airflow.operators.bash_operator import BashOperator
```

3.5.4 Default Arguments

We're about to create a DAG and some tasks, and we have the choice to explicitly pass a set of arguments to each task's constructor (which would become redundant), or (better!) we can define a dictionary of default parameters that we can use when creating tasks.

```
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}
```

For more information about the BaseOperator's parameters and what they do, refer to the *airflow.models.BaseOperator* documentation.

Also, note that you could easily define different sets of arguments that would serve different purposes. An example of that would be to have different settings between a production and development environment.

3.5.5 Instantiate a DAG

We'll need a DAG object to nest our tasks into. Here we pass a string that defines the `dag_id`, which serves as a unique identifier for your DAG. We also pass the default argument dictionary that we just defined and define a `schedule_interval` of 1 day for the DAG.

```
dag = DAG(
    'tutorial', default_args=default_args, schedule_interval=timedelta(days=1))
```

3.5.6 Tasks

Tasks are generated when instantiating operator objects. An object instantiated from an operator is called a constructor. The first argument `task_id` acts as a unique identifier for the task.

```
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)
```

Notice how we pass a mix of operator specific arguments (`bash_command`) and an argument common to all operators (`retries`) inherited from BaseOperator to the operator's constructor. This is simpler than passing every argument for every constructor call. Also, notice that in the second task we override the `retries` parameter with `3`.

The precedence rules for a task are as follows:

1. Explicitly passed arguments
2. Values that exist in the `default_args` dictionary
3. The operator's default value, if one exists

A task must include or inherit the arguments `task_id` and `owner`, otherwise Airflow will raise an exception.

3.5.7 Templating with Jinja

Airflow leverages the power of Jinja Templating and provides the pipeline author with a set of built-in parameters and macros. Airflow also provides hooks for the pipeline author to define their own parameters, macros and templates.

This tutorial barely scratches the surface of what you can do with templating in Airflow, but the goal of this section is to let you know this feature exists, get you familiar with double curly brackets, and point to the most common template variable: `{{ ds }}` (today's "date stamp").

```
templated_command = """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7) }}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)
```

Notice that the `templated_command` contains code logic in `{% %}` blocks, references parameters like `{{ ds }}`, calls a function as in `{{ macros.ds_add(ds, 7)}}`, and references a user-defined parameter in `{{ params.my_param }}`.

The `params` hook in `BaseOperator` allows you to pass a dictionary of parameters and/or objects to your templates. Please take the time to understand how the parameter `my_param` makes it through to the template.

Files can also be passed to the `bash_command` argument, like `bash_command='templated_command.sh'`, where the file location is relative to the directory containing the pipeline file (`tutorial.py` in this case). This may be desirable for many reasons, like separating your script's logic and pipeline code, allowing for proper code highlighting in files composed in different languages, and general flexibility in structuring pipelines. It is also possible to define your `template_searchpath` as pointing to any folder locations in the DAG constructor call.

Using that same DAG constructor call, it is possible to define `user_defined_macros` which allow you to specify your own variables. For example, passing `dict(foo='bar')` to this argument allows you to use `{{ foo }}` in your templates. Moreover, specifying `user_defined_filters` allow you to register you own filters. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to use `{{ 'world' | hello }}` in your templates. For more information regarding custom filters have a look at the Jinja Documentation

For more information on the variables and macros that can be referenced in templates, make sure to read through the *Macros reference*

3.5.8 Setting up Dependencies

We have tasks *t1*, *t2* and *t3* that do not depend on each other. Here's a few ways you can define dependencies between them:

```
t1.set_downstream(t2)

# This means that t2 will depend on t1
# running successfully to run.
# It is equivalent to:
t2.set_upstream(t1)

# The bit shift operator can also be
# used to chain operations:
t1 >> t2

# And the upstream dependency with the
# bit shift operator:
t2 << t1

# Chaining multiple dependencies becomes
# concise with the bit shift operator:
t1 >> t2 >> t3

# A list of tasks can also be set as
# dependencies. These operations
# all have the same effect:
t1.set_downstream([t2, t3])
t1 >> [t2, t3]
[t2, t3] << t1
```

Note that when executing your script, Airflow will raise exceptions when it finds cycles in your DAG or when a dependency is referenced more than once.

3.5.9 Recap

Alright, so we have a pretty basic DAG. At this point your code should look something like this:

```
"""
Code that goes along with the Airflow tutorial located at:
https://github.com/apache/airflow/blob/master/airflow/example_dags/tutorial.py
"""
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

```

(continues on next page)

(continued from previous page)

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}

dag = DAG(
    'tutorial', default_args=default_args, schedule_interval=timedelta(days=1))

# t1, t2 and t3 are examples of tasks created by instantiating operators
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)

templated_command = """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)

t2.set_upstream(t1)
t3.set_upstream(t1)
```

3.5.10 Testing

3.5.10.1 Running the Script

Time to run some tests. First, let's make sure the pipeline is parsed successfully.

Let's assume we're saving the code from the previous step in `tutorial.py` in the DAGs folder referenced in your `airflow.cfg`. The default location for your DAGs is `~/airflow/dags`.

```
python ~/airflow/dags/tutorial.py
```

If the script does not raise an exception it means that you haven't done anything horribly wrong, and that your Airflow environment is somewhat sound.

3.5.10.2 Command Line Metadata Validation

Let's run a few commands to validate this script further.

```
# print the list of active DAGs
airflow list_dags

# prints the list of tasks in the "tutorial" DAG
airflow list_tasks tutorial

# prints the hierarchy of tasks in the "tutorial" DAG
airflow list_tasks tutorial --tree
```

3.5.10.3 Testing

Let's test by running the actual task instances on a specific date. The date specified in this context is an `execution_date`, which simulates the scheduler running your task or dag at a specific date + time:

```
# command layout: command subcommand dag_id task_id date

# testing print_date
airflow test tutorial print_date 2015-06-01

# testing sleep
airflow test tutorial sleep 2015-06-01
```

Now remember what we did with templating earlier? See how this template gets rendered and executed by running this command:

```
# testing templated
airflow test tutorial templated 2015-06-01
```

This should result in displaying a verbose log of events and ultimately running your bash command and printing the result.

Note that the `airflow test` command runs task instances locally, outputs their log to stdout (on screen), doesn't bother with dependencies, and doesn't communicate state (running, success, failed, …) to the database. It simply allows testing a single task instance.

3.5.10.4 Backfill

Everything looks like it's running fine so let's run a backfill. `backfill` will respect your dependencies, emit logs into files and talk to the database to record status. If you do have a webserver up, you'll be able to track the progress. `airflow webserver` will start a web server if you are interested in tracking the progress visually as your backfill progresses.

Note that if you use `depends_on_past=True`, individual task instances will depend on the success of the preceding task instance, except for the start_date specified itself, for which this dependency is disregarded.

The date range in this context is a `start_date` and optionally an `end_date`, which are used to populate the run schedule with task instances from this dag.

```
# optional, start a web server in debug mode in the background
# airflow webserver --debug &

# start your backfill on a date range
airflow backfill tutorial -s 2015-06-01 -e 2015-06-07
```

3.5.11 What's Next?

That's it, you've written, tested and backfilled your very first Airflow pipeline. Merging your code into a code repository that has a master scheduler running against it should get it to get triggered and run every day.

Here's a few things you might want to do next:

- Take an in-depth tour of the UI - click all the things!
- Keep reading the docs! Especially the sections on:
  - Command line interface
  - Operators
  - Macros
- Write your first pipeline!

3.6 How-to Guides

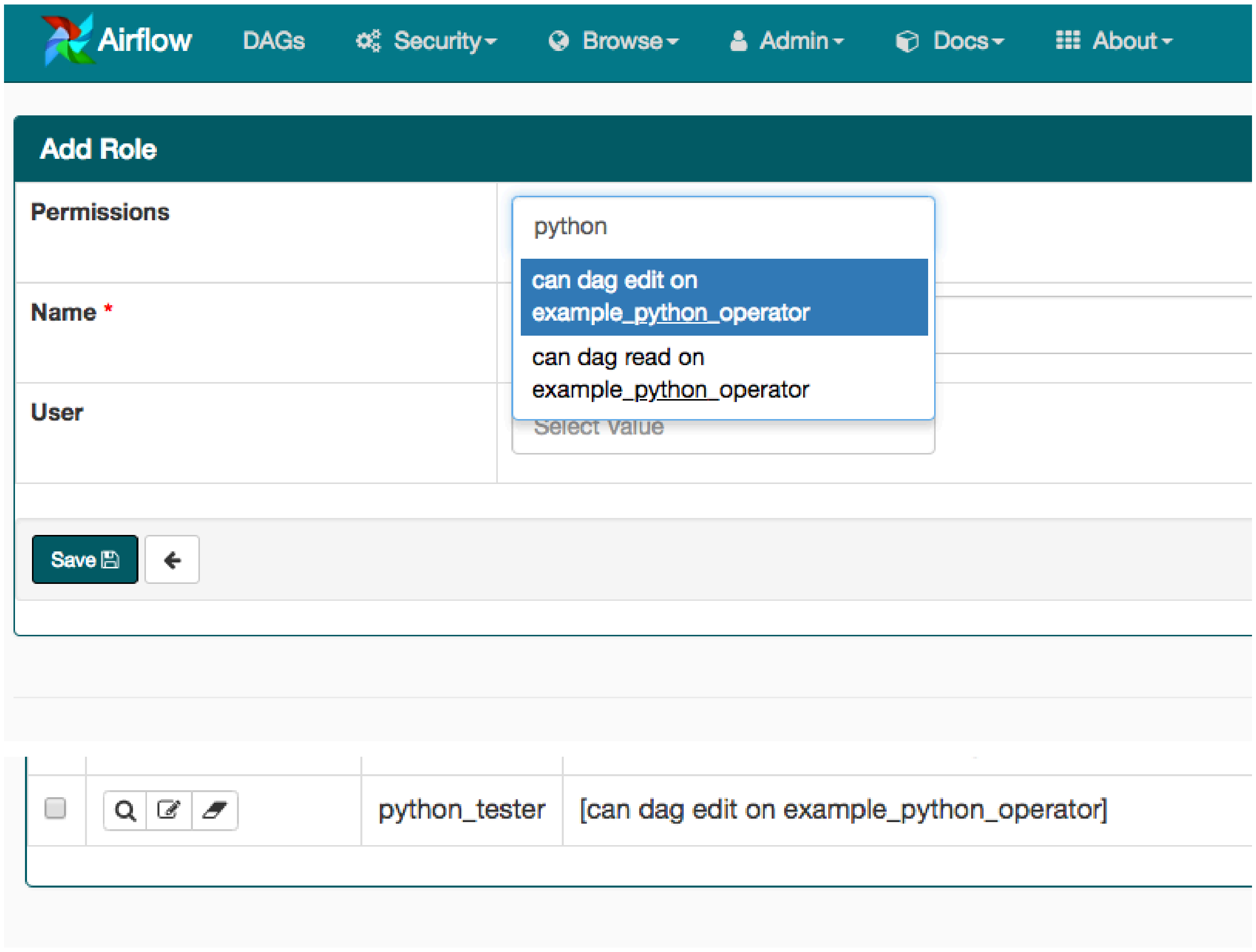
Setting up the sandbox in the *Quick Start* section was easy; building a production-grade environment requires a bit more work!

These how-to guides will step you through common tasks in using and configuring an Airflow environment.

3.6.1 Add a new role in RBAC UI

There are five roles created for Airflow by default: Admin, User, Op, Viewer, and Public. The master branch adds beta support for DAG level access for RBAC UI. Each DAG comes with two permissions: read and write.

The Admin could create a specific role which is only allowed to read / write certain DAGs. To configure a new role, go to `Security` tab and click `List Roles` in the new UI.



		python_tester	[can dag edit on example_python_operator]

The image shows the creation of a role which can only write to `example_python_operator`. You can also create roles via the CLI using the `airflow roles` command, e.g.:

airflow roles –create Role1 Role2

And we could assign the given role to a new user using the `airflow users --add-role` CLI command. Default roles(Admin, User, Viewer, Op) shipped with RBAC could view the details for every dag.

3.6.2 Setting Configuration Options

The first time you run Airflow, it will create a file called `airflow.cfg` in your `$AIRFLOW_HOME` directory (`~/airflow` by default). This file contains Airflow's configuration and you can edit it to change any of the settings. You can also set options with environment variables by using this format: `$AIRFLOW__{SECTION}__{KEY}` (note the double underscores).

For example, the metadata database connection string can either be set in `airflow.cfg` like this:

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

or by creating a corresponding environment variable:

```
AIRFLOW__CORE__SQL_ALCHEMY_CONN=my_conn_string
```

You can also derive the connection string at run time by appending `_cmd` to the key like this:

```
[core]
sql_alchemy_conn_cmd = bash_command_to_run
```

The following config options support this `_cmd` version:

- `sql_alchemy_conn` in `[core]` section
- `fernet_key` in `[core]` section
- `broker_url` in `[celery]` section
- `result_backend` in `[celery]` section
- `password` in `[atlas]` section
- `smtp_password` in `[smtp]` section
- `bind_password` in `[ldap]` section
- `git_password` in `[kubernetes]` section

The idea behind this is to not store passwords on boxes in plain text files.

The universal order of precedence for all configuration options is as follows:

1. set as an environment variable
2. set in `airflow.cfg`
3. command in `airflow.cfg`
4. Airflow's built in defaults

3.6.3 Initializing a Database Backend

If you want to take a real test drive of Airflow, you should consider setting up a real database backend and switching to the LocalExecutor.

As Airflow was built to interact with its metadata using the great SqlAlchemy library, you should be able to use any database backend supported as a SqlAlchemy backend. We recommend using **MySQL** or **Postgres**.

Note: We rely on more strict ANSI SQL settings for MySQL in order to have sane defaults. Make sure to have specified *explicit_defaults_for_timestamp=1* in your my.cnf under *[mysqld]*

Note: If you decide to use **Postgres**, we recommend using the `psycopg2` driver and specifying it in your SqlAlchemy connection string. (I.e., `postgresql+psycopg2://<user>:<password>@<host>/<db>`.) Also note that since SqlAlchemy does not expose a way to target a specific schema in the Postgres connection URI, you may want to set a default schema for your role with a command similar to `ALTER ROLE username SET search_path = airflow, foobar;`

Once you've setup your database to host Airflow, you'll need to alter the SqlAlchemy connection string located in your configuration file `$AIRFLOW_HOME/airflow.cfg`. You should then also change the "executor" setting to use "LocalExecutor", an executor that can parallelize task instances locally.

```
# initialize the database
airflow initdb
```

3.6.4 Using Operators

An operator represents a single, ideally idempotent, task. Operators determine what actually executes when your DAG runs.

See the *Operators Concepts* documentation and the *Operators API Reference* for more information.

3.6.4.1 BashOperator

Use the `BashOperator` to execute commands in a Bash shell.

airflow/example_dags/example_bash_operator.pyView Source

```
run_this = BashOperator(
    task_id='run_after_loop',
    bash_command='echo 1',
    dag=dag,
)
```

Templating

You can use *Jinja templates* to parameterize the `bash_command` argument.

airflow/example_dags/example_bash_operator.pyView Source

```
also_run_this = BashOperator(
    task_id='also_run_this',
    bash_command='echo "run_id={{ run_id }} | dag_run={{ dag_run }}"',
    dag=dag,
)
```

Troubleshooting

Jinja template not found

Add a space after the script name when directly calling a Bash script with the `bash_command` argument. This is because Airflow tries to apply a Jinja template to it, which will fail.

```
t2 = BashOperator(
    task_id='bash_example',

    # This fails with `Jinja template not found` error
    # bash_command="/home/batcher/test.sh",

    # This works (has a space after)
    bash_command="/home/batcher/test.sh ",
    dag=dag)
```

3.6.4.2 Dingding Operators

Prerequisite Tasks

To use this operators, you must do a few things:

- Add custom robot to Dingding group which you want to send Dingding message.
- Get the webhook token from Dingding custom robot.
- Put the Dingding custom robot token in the password field of the `dingding_default` Connection. Notice that you just need token rather than the whole webhook string.

Basic Usage

Use the `DingdingOperator` to send Dingding message:

airflow/contrib/example_dags/example_dingding_operator.pyView Source

```
text_msg_remind_none = DingdingOperator(
    task_id='text_msg_remind_none',
    dingding_conn_id='dingding_default',
    message_type='text',
    message='Airflow dingding text message remind none',
    at_mobiles=None,
    at_all=False,
    dag=dag,
)
```

Remind users in message

Use parameters `at_mobiles` and `at_all` to remind specific users when you send message, `at_mobiles` will be ignored When `at_all` is set to `True`:

airflow/contrib/example_dags/example_dingding_operator.pyView Source

```
text_msg_remind_all = DingdingOperator(
    task_id='text_msg_remind_all',
    dingding_conn_id='dingding_default',
    message_type='text',
    message='Airflow dingding text message remind all users in group',
    # list of user phone/email here in the group
    # when at_all is specific will cover at_mobiles
    at_mobiles=['156XXXXXXXX', '130XXXXXXXX'],
    at_all=True,
    dag=dag,
)
```

Send rich text message

The Dingding operator can send rich text messages including link, markdown, actionCard and feedCard. A rich text message can not remind specific users except by using markdown type message:

airflow/contrib/example_dags/example_dingding_operator.pyView Source

```
markdown_msg = DingdingOperator(
    task_id='markdown_msg',
    dingding_conn_id='dingding_default',
    message_type='markdown',
    message={
        'title': 'Airflow dingding markdown message',
```

(continues on next page)

(continued from previous page)

```
        'text': '# Markdown message title\n'
                'content content .. \n'
                '### sub-title\n'
                '![logo](http://airflow.apache.org/_images/pin_large.png)'
    },
    at_mobiles=['156XXXXXXXX'],
    at_all=False,
    dag=dag,
)
```

Sending messages from a Task callback

Dingding operator could handle task callback by writing a function wrapper dingding operators and then pass the function to `sla_miss_callback`, `on_success_callback`, `on_failure_callback`, or `on_retry_callback`. Here we use `on_failure_callback` as an example:

airflow/contrib/example_dags/example_dingding_operator.pyView Source

```
def failure_callback(context):
    message = 'AIRFLOW TASK FAILURE TIPS:\n' \
              'DAG:    {}\n' \
              'TASKS:  {}\n' \
              'Reason: {}\n' \
        .format(context['task_instance'].dag_id,
                context['task_instance'].task_id,
                context['exception'])
    return DingdingOperator(
        task_id='dingding_success_callback',
        dingding_conn_id='dingding_default',
        message_type='text',
        message=message,
        at_all=True,
    ).execute(context)


args['on_failure_callback'] = failure_callback
```

Changing connection host if you need

The Dingding operator post http requests using default host `https://oapi.dingtalk.com`, if you need to change the host used you can set the host field of the connection.

More information

See Dingding documentation on how to custom robot.

3.6.4.3 Google Cloud Operators

Google Cloud Bigtable Operators

- *BigtableInstanceCreateOperator*
- *BigtableInstanceDeleteOperator*
- *BigtableClusterUpdateOperator*
- *BigtableTableDeleteOperator*
- *BigtableTableWaitForReplicationSensor*

All examples below rely on the following variables, which can be passed via environment variables.

airflow/contrib/example_dags/example_gcp_bigtable_operators.pyView Source

```
GCP_PROJECT_ID = getenv('GCP_PROJECT_ID', 'example-project')
CBT_INSTANCE_ID = getenv('CBT_INSTANCE_ID', 'some-instance-id')
CBT_INSTANCE_DISPLAY_NAME = getenv('CBT_INSTANCE_DISPLAY_NAME', 'Human-readable name')
CBT_INSTANCE_TYPE = getenv('CBT_INSTANCE_TYPE', '2')
CBT_INSTANCE_LABELS = getenv('CBT_INSTANCE_LABELS', '{}')
CBT_CLUSTER_ID = getenv('CBT_CLUSTER_ID', 'some-cluster-id')
CBT_CLUSTER_ZONE = getenv('CBT_CLUSTER_ZONE', 'europe-west1-b')
CBT_CLUSTER_NODES = getenv('CBT_CLUSTER_NODES', '3')
CBT_CLUSTER_NODES_UPDATED = getenv('CBT_CLUSTER_NODES_UPDATED', '5')
CBT_CLUSTER_STORAGE_TYPE = getenv('CBT_CLUSTER_STORAGE_TYPE', '2')
CBT_TABLE_ID = getenv('CBT_TABLE_ID', 'some-table-id')
CBT_POKE_INTERVAL = getenv('CBT_POKE_INTERVAL', '60')
```

BigtableInstanceCreateOperator

Use the `BigtableInstanceCreateOperator` to create a Google Cloud Bigtable instance.

If the Cloud Bigtable instance with the given ID exists, the operator does not compare its configuration and immediately succeeds. No changes are made to the existing instance.

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_bigtable_operators.pyView Source

```
create_instance_task = BigtableInstanceCreateOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=CBT_INSTANCE_ID,
    main_cluster_id=CBT_CLUSTER_ID,
    main_cluster_zone=CBT_CLUSTER_ZONE,
    instance_display_name=CBT_INSTANCE_DISPLAY_NAME,
    instance_type=int(CBT_INSTANCE_TYPE),
    instance_labels=json.loads(CBT_INSTANCE_LABELS),
    cluster_nodes=int(CBT_CLUSTER_NODES),
    cluster_storage_type=int(CBT_CLUSTER_STORAGE_TYPE),
    task_id='create_instance_task',
)
create_instance_task2 = BigtableInstanceCreateOperator(
    instance_id=CBT_INSTANCE_ID,
    main_cluster_id=CBT_CLUSTER_ID,
    main_cluster_zone=CBT_CLUSTER_ZONE,
```

(continues on next page)

(continued from previous page)

```
    instance_display_name=CBT_INSTANCE_DISPLAY_NAME,
    instance_type=int(CBT_INSTANCE_TYPE),
    instance_labels=json.loads(CBT_INSTANCE_LABELS),
    cluster_nodes=int(CBT_CLUSTER_NODES),
    cluster_storage_type=int(CBT_CLUSTER_STORAGE_TYPE),
    task_id='create_instance_task2',
)
create_instance_task >> create_instance_task2
```

BigtableInstanceDeleteOperator

Use the *BigtableInstanceDeleteOperator* to delete a Google Cloud Bigtable instance.

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_bigtable_operators.pyView Source

```
delete_instance_task = BigtableInstanceDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=CBT_INSTANCE_ID,
    task_id='delete_instance_task',
)
delete_instance_task2 = BigtableInstanceDeleteOperator(
    instance_id=CBT_INSTANCE_ID,
    task_id='delete_instance_task2',
)
```

BigtableClusterUpdateOperator

Use the *BigtableClusterUpdateOperator* to modify number of nodes in a Cloud Bigtable cluster.

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_bigtable_operators.pyView Source

```
cluster_update_task = BigtableClusterUpdateOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=CBT_INSTANCE_ID,
    cluster_id=CBT_CLUSTER_ID,
    nodes=int(CBT_CLUSTER_NODES_UPDATED),
    task_id='update_cluster_task',
)
cluster_update_task2 = BigtableClusterUpdateOperator(
    instance_id=CBT_INSTANCE_ID,
    cluster_id=CBT_CLUSTER_ID,
```

(continues on next page)

(continued from previous page)

```
    nodes=int(CBT_CLUSTER_NODES_UPDATED),
    task_id='update_cluster_task2',
)
cluster_update_task >> cluster_update_task2
```

BigtableTableCreateOperator

Creates a table in a Cloud Bigtable instance.

If the table with given ID exists in the Cloud Bigtable instance, the operator compares the Column Families. If the Column Families are identical operator succeeds. Otherwise, the operator fails with the appropriate error message.

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_bigtable_operators.pyView Source

```
create_table_task = BigtableTableCreateOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=CBT_INSTANCE_ID,
    table_id=CBT_TABLE_ID,
    task_id='create_table',
)
create_table_task2 = BigtableTableCreateOperator(
    instance_id=CBT_INSTANCE_ID,
    table_id=CBT_TABLE_ID,
    task_id='create_table_task2',
)
create_table_task >> create_table_task2
```

Advanced

When creating a table, you can specify the optional `initial_split_keys` and `column_families`. Please refer to the Python Client for Google Cloud Bigtable documentation for Table and for Column Families.

BigtableTableDeleteOperator

Use the *`BigtableTableDeleteOperator`* to delete a table in Google Cloud Bigtable.

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_bigtable_operators.pyView Source

```
delete_table_task = BigtableTableDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=CBT_INSTANCE_ID,
    table_id=CBT_TABLE_ID,
    task_id='delete_table_task',
)
delete_table_task2 = BigtableTableDeleteOperator(
    instance_id=CBT_INSTANCE_ID,
    table_id=CBT_TABLE_ID,
    task_id='delete_table_task2',
)
```

BigtableTableWaitForReplicationSensor

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

Use the `BigtableTableWaitForReplicationSensor` to wait for the table to replicate fully.

The same arguments apply to this sensor as the *BigtableTableCreateOperator*.

Note: If the table or the Cloud Bigtable instance does not exist, this sensor waits for the table until timeout hits and does not raise any exception.

Using the operator

airflow/contrib/example_dags/example_gcp_bigtable_operators.pyView Source

```
wait_for_table_replication_task = BigtableTableWaitForReplicationSensor(
    project_id=GCP_PROJECT_ID,
    instance_id=CBT_INSTANCE_ID,
    table_id=CBT_TABLE_ID,
    poke_interval=int(CBT_POKE_INTERVAL),
    timeout=180,
    task_id='wait_for_table_replication_task',
)
wait_for_table_replication_task2 = BigtableTableWaitForReplicationSensor(
    instance_id=CBT_INSTANCE_ID,
    table_id=CBT_TABLE_ID,
    poke_interval=int(CBT_POKE_INTERVAL),
    timeout=180,
    task_id='wait_for_table_replication_task2',
)
```

Google Cloud Build Operators

The GoogleCloud Build is a service that executes your builds on Google Cloud Platform infrastructure. Cloud Build can import source code from Google Cloud Storage, Cloud Source Repositories, execute a build to your specifications, and produce artifacts such as Docker containers or Java archives.

- *Prerequisite Tasks*
- *Build configuration overview*

- *Trigger a build*
- *Reference*

Prerequisite Tasks

To use these operators, you must do a few things:

- Select or create a Cloud Platform project using Cloud Console.
- Enable billing for your project, as described in Google Cloud documentation.
- Enable API, as described in Cloud Console documentation.
- Install API libraries via **pip**.

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

  Detailed information is available *Installation*
- *Setup Connection*.

Build configuration overview

In order to trigger a build, it is necessary to pass the build configuration.

airflow/contrib/example_dags/example_gcp_cloud_build.pyView Source

```
create_build_from_storage_body = {
    "source": {"storageSource": GCP_SOURCE_ARCHIVE_URL},
    "steps": [
        {
            "name": "gcr.io/cloud-builders/docker",
            "args": ["build", "-t", "gcr.io/$PROJECT_ID/{}".format(GCP_SOURCE_BUCKET_
↪NAME), "."],
        }
    ],
    "images": ["gcr.io/$PROJECT_ID/{}".format(GCP_SOURCE_BUCKET_NAME)],
}
```

The source code for the build can come from Google Cloud Build Storage:

```
"source": {"storageSource": {"bucket": "bucket-name", "object": "object-name.tar.gz"}}
↪,
```

It is also possible to specify it using the URL:

```
"source": {"storageSource": "gs://bucket-name/object-name.tar.gz"},
```

In addition, a build can refer to source stored in Google Cloud Source Repositories.

```
"source": {
    "repoSource": {
        "projectId": "airflow-project",
        "repoName": "airflow-repo",
        "branchName": "master",
    }
},
```

It is also possible to specify it using the URL:

```
"source": {"repoSource": "https://source.developers.google.com/p/airflow-project/r/
↪airflow-repo"},
```

Read Build Configuration Overview to understand all the fields you can include in a build config file.

Trigger a build

Trigger a build is performed with the `CloudBuildCreateBuildOperator` operator.

airflow/contrib/example_dags/example_gcp_cloud_build.pyView Source

```
create_build_from_storage = CloudBuildCreateBuildOperator(
    task_id="create_build_from_storage", project_id=GCP_PROJECT_ID, body=create_build_
↪from_storage_body
)
```

You can use *Jinja templating* with **:template-fields:‘airflow.contrib.operators.gcp_cloud_build_operator.CloudBuildCreateBuildOper** parameters which allows you to dynamically determine values. The result is saved to *XCom*, which allows it to be used by other operators.

invalid class name airflow.contrib.operators.gcp_cloud_build_operator.CloudBuildCreateBuildOperator Error loading airflow.contrib.operators.gcp_cloud_build_operator module.

airflow/contrib/example_dags/example_gcp_cloud_build.pyView Source

```
create_build_from_storage_result = BashOperator(
    bash_command="echo '{{ task_instance.xcom_pull('create_build_from_storage')[
↪'images'][0] }}'",
    task_id="create_build_from_storage_result",
)
```

Reference

For further information, look at:

- Client Library Documentation
- Product Documentation

Google Compute Engine Operators

- *GceInstanceStartOperator*
- *GceInstanceStopOperator*
- *GceSetMachineTypeOperator*
- *GceInstanceTemplateCopyOperator*
- *GceInstanceGroupManagerUpdateTemplateOperator*

GceInstanceStartOperator

Use the *GceInstanceStartOperator* to start an existing Google Compute Engine instance.

Arguments

The following examples of OS environment variables used to pass arguments to the operator:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCE_ZONE = os.environ.get('GCE_ZONE', 'europe-west1-b')
GCE_INSTANCE = os.environ.get('GCE_INSTANCE', 'testinstance')
```

Using the operator

The code to create the operator:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
gce_instance_start = GceInstanceStartOperator(
    project_id=GCP_PROJECT_ID,
    zone=GCE_ZONE,
    resource_id=GCE_INSTANCE,
    task_id='gcp_compute_start_task'
)
```

You can also create the operator without project id - project id will be retrieved from the GCP connection id used:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
gce_instance_start2 = GceInstanceStartOperator(
    zone=GCE_ZONE,
    resource_id=GCE_INSTANCE,
    task_id='gcp_compute_start_task2'
)
```

Templating

```
template_fields = ('project_id', 'zone', 'resource_id', 'gcp_conn_id', 'api_version')
```

More information

See Google Compute Engine API documentation to start an instance.

GceInstanceStopOperator

Use the operator to stop Google Compute Engine instance.

For parameter definition, take a look at *GceInstanceStopOperator*

Arguments

The following examples of OS environment variables used to pass arguments to the operator:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCE_ZONE = os.environ.get('GCE_ZONE', 'europe-west1-b')
GCE_INSTANCE = os.environ.get('GCE_INSTANCE', 'testinstance')
```

Using the operator

The code to create the operator:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
gce_instance_stop = GceInstanceStopOperator(
    project_id=GCP_PROJECT_ID,
    zone=GCE_ZONE,
    resource_id=GCE_INSTANCE,
    task_id='gcp_compute_stop_task'
)
```

You can also create the operator without project id - project id will be retrieved from the GCP connection used:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
gce_instance_stop2 = GceInstanceStopOperator(
    zone=GCE_ZONE,
    resource_id=GCE_INSTANCE,
    task_id='gcp_compute_stop_task2'
)
```

Templating

```
template_fields = ('project_id', 'zone', 'resource_id', 'gcp_conn_id', 'api_version')
```

More information

See Google Compute Engine API documentation to stop an instance.

GceSetMachineTypeOperator

Use the operator to change machine type of a Google Compute Engine instance.

For parameter definition, take a look at `GceSetMachineTypeOperator`.

Arguments

The following examples of OS environment variables used to pass arguments to the operator:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCE_ZONE = os.environ.get('GCE_ZONE', 'europe-west1-b')
GCE_INSTANCE = os.environ.get('GCE_INSTANCE', 'testinstance')
```

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
GCE_SHORT_MACHINE_TYPE_NAME = os.environ.get('GCE_SHORT_MACHINE_TYPE_NAME', 'n1-
↪standard-1')
SET_MACHINE_TYPE_BODY = {
    'machineType': 'zones/{}/machineTypes/{}'.format(GCE_ZONE, GCE_SHORT_MACHINE_TYPE_
↪NAME)
}
```

Using the operator

The code to create the operator:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
gce_set_machine_type = GceSetMachineTypeOperator(
    project_id=GCP_PROJECT_ID,
    zone=GCE_ZONE,
    resource_id=GCE_INSTANCE,
    body=SET_MACHINE_TYPE_BODY,
    task_id='gcp_compute_set_machine_type'
)
```

You can also create the operator without project id - project id will be retrieved from the GCP connection used:

airflow/contrib/example_dags/example_gcp_compute.pyView Source

```
gce_set_machine_type2 = GceSetMachineTypeOperator(
    zone=GCE_ZONE,
    resource_id=GCE_INSTANCE,
    body=SET_MACHINE_TYPE_BODY,
    task_id='gcp_compute_set_machine_type2'
)
```

Templating

```
template_fields = ('project_id', 'zone', 'resource_id', 'gcp_conn_id', 'api_version')
```

More information

See Google Compute Engine API documentation to set the machine type.

GceInstanceTemplateCopyOperator

Use the operator to copy an existing Google Compute Engine instance template applying a patch to it.

For parameter definition, take a look at *GceInstanceTemplateCopyOperator*.

Arguments

The following examples of OS environment variables used to pass arguments to the operator:

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCE_ZONE = os.environ.get('GCE_ZONE', 'europe-west1-b')
```

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
GCE_TEMPLATE_NAME = os.environ.get('GCE_TEMPLATE_NAME', 'instance-template-test')
GCE_NEW_TEMPLATE_NAME = os.environ.get('GCE_NEW_TEMPLATE_NAME',
                                       'instance-template-test-new')
GCE_NEW_DESCRIPTION = os.environ.get('GCE_NEW_DESCRIPTION', 'Test new description')
GCE_INSTANCE_TEMPLATE_BODY_UPDATE = {
    "name": GCE_NEW_TEMPLATE_NAME,
    "description": GCE_NEW_DESCRIPTION,
    "properties": {
        "machineType": "n1-standard-2"
    }
}
```

Using the operator

The code to create the operator:

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
gce_instance_template_copy = GceInstanceTemplateCopyOperator(
    project_id=GCP_PROJECT_ID,
    resource_id=GCE_TEMPLATE_NAME,
    body_patch=GCE_INSTANCE_TEMPLATE_BODY_UPDATE,
    task_id='gcp_compute_igm_copy_template_task'
)
```

You can also create the operator without project id - project id will be retrieved from the GCP connection used:

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
gce_instance_template_copy2 = GceInstanceTemplateCopyOperator(
    resource_id=GCE_TEMPLATE_NAME,
    body_patch=GCE_INSTANCE_TEMPLATE_BODY_UPDATE,
    task_id='gcp_compute_igm_copy_template_task_2'
)
```

Templating

```
template_fields = ('project_id', 'resource_id', 'request_id',
                   'gcp_conn_id', 'api_version')
```

More information

See Google Compute Engine API documentation to create a new instance with an existing template.

GceInstanceGroupManagerUpdateTemplateOperator

Use the operator to update a template in Google Compute Engine Instance Group Manager.

For parameter definition, take a look at *GceInstanceGroupManagerUpdateTemplateOperator*.

Arguments

The following examples of OS environment variables used to pass arguments to the operator:

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCE_ZONE = os.environ.get('GCE_ZONE', 'europe-west1-b')
```

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
GCE_INSTANCE_GROUP_MANAGER_NAME = os.environ.get('GCE_INSTANCE_GROUP_MANAGER_NAME',
                                                 'instance-group-test')

SOURCE_TEMPLATE_URL = os.environ.get(
    'SOURCE_TEMPLATE_URL',
    "https://www.googleapis.com/compute/beta/projects/" + GCP_PROJECT_ID +
    "/global/instanceTemplates/instance-template-test")

DESTINATION_TEMPLATE_URL = os.environ.get(
    'DESTINATION_TEMPLATE_URL',
    "https://www.googleapis.com/compute/beta/projects/" + GCP_PROJECT_ID +
    "/global/instanceTemplates/" + GCE_NEW_TEMPLATE_NAME)

UPDATE_POLICY = {
    "type": "OPPORTUNISTIC",
    "minimalAction": "RESTART",
    "maxSurge": {
        "fixed": 1
    },
    "minReadySec": 1800
}
```

Using the operator

The code to create the operator:

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
gce_instance_group_manager_update_template = \
    GceInstanceGroupManagerUpdateTemplateOperator(
        project_id=GCP_PROJECT_ID,
        resource_id=GCE_INSTANCE_GROUP_MANAGER_NAME,
        zone=GCE_ZONE,
        source_template=SOURCE_TEMPLATE_URL,
        destination_template=DESTINATION_TEMPLATE_URL,
        update_policy=UPDATE_POLICY,
        task_id='gcp_compute_igm_group_manager_update_template'
    )
```

You can also create the operator without project id - project id will be retrieved from the GCP connection used:

airflow/contrib/example_dags/example_gcp_compute_igm.pyView Source

```
gce_instance_group_manager_update_template2 = \
    GceInstanceGroupManagerUpdateTemplateOperator(
        resource_id=GCE_INSTANCE_GROUP_MANAGER_NAME,
        zone=GCE_ZONE,
        source_template=SOURCE_TEMPLATE_URL,
        destination_template=DESTINATION_TEMPLATE_URL,
        task_id='gcp_compute_igm_group_manager_update_template_2'
    )
```

Templating

```
template_fields = ('project_id', 'resource_id', 'zone', 'request_id',
                   'source_template', 'destination_template',
                   'gcp_conn_id', 'api_version')
```

Troubleshooting

You might find that your GceInstanceGroupManagerUpdateTemplateOperator fails with missing permissions. To execute the operation, the service account requires the permissions that theService Account User role provides (assigned via Google Cloud IAM).

More information

See Google Compute Engine API documentation to manage a group instance.

Google Cloud Functions Operators

- *GcfFunctionDeleteOperator*
- *GcfFunctionDeployOperator*

GcfFunctionDeleteOperator

Use the operator to delete a function from Google Cloud Functions.

For parameter definition, take a look at *GcfFunctionDeleteOperator*.

Arguments

The following examples of OS environment variables show how you can build function name to use in the operator:

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_LOCATION = os.environ.get('GCP_LOCATION', 'europe-west1')
GCF_SHORT_FUNCTION_NAME = os.environ.get('GCF_SHORT_FUNCTION_NAME', 'hello').\
    replace("-", "_")  # make sure there are no dashes in function name (!)
FUNCTION_NAME = 'projects/{}/locations/{}/functions/{}'.format(GCP_PROJECT_ID,
                                                               GCP_LOCATION,
                                                               GCF_SHORT_FUNCTION_
↪NAME)
```

Using the operator

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
delete_task = GcfFunctionDeleteOperator(
    task_id="gcf_delete_task",
    name=FUNCTION_NAME
)
```

Templating

```
template_fields = ('name', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud Functions API documentation to delete a function.

GcfFunctionDeployOperator

Use the operator to deploy a function to Google Cloud Functions. If a function with this name already exists, it will be updated.

For parameter definition, take a look at *GcfFunctionDeployOperator*.

Arguments

In the example DAG the following environment variables are used to parameterize the operator's definition:

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_LOCATION = os.environ.get('GCP_LOCATION', 'europe-west1')
GCF_SHORT_FUNCTION_NAME = os.environ.get('GCF_SHORT_FUNCTION_NAME', 'hello').\
    replace("-", "_")  # make sure there are no dashes in function name (!)
FUNCTION_NAME = 'projects/{}/locations/{}/functions/{}'.format(GCP_PROJECT_ID,
                                                               GCP_LOCATION,
                                                               GCF_SHORT_FUNCTION_
↪NAME)
```

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
GCF_SOURCE_ARCHIVE_URL = os.environ.get('GCF_SOURCE_ARCHIVE_URL', '')
GCF_SOURCE_UPLOAD_URL = os.environ.get('GCF_SOURCE_UPLOAD_URL', '')
GCF_SOURCE_REPOSITORY = os.environ.get(
    'GCF_SOURCE_REPOSITORY',
    'https://source.developers.google.com/'
    'projects/{}/repos/hello-world/moveable-aliases/master'.format(GCP_PROJECT_ID))
GCF_ZIP_PATH = os.environ.get('GCF_ZIP_PATH', '')
GCF_ENTRYPOINT = os.environ.get('GCF_ENTRYPOINT', 'helloWorld')
GCF_RUNTIME = 'nodejs6'
GCP_VALIDATE_BODY = os.environ.get('GCP_VALIDATE_BODY', True)
```

Some of those variables are used to create the request's body:

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
body = {
    "name": FUNCTION_NAME,
    "entryPoint": GCF_ENTRYPOINT,
    "runtime": GCF_RUNTIME,
    "httpsTrigger": {}
}
```

When a DAG is created, the default_args dictionary can be used to pass arguments common with other tasks:

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
default_args = {
    'start_date': dates.days_ago(1)
}
```

Note that the neither the body nor the default args are complete in the above examples. Depending on the variables set, there might be different variants on how to pass source code related fields. Currently, you can pass either `sourceArchiveUrl`, `sourceRepository` or `sourceUploadUrl` as described in the Cloud Functions API specification.

Additionally, `default_args` or direct operator args might contain `zip_path` parameter to run the extra step of uploading the source code before deploying it. In this case, you also need to provide an empty `sourceUploadUrl` parameter in the body.

Using the operator

Depending on the combination of parameters, the Function's source code can be obtained from different sources:

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
if GCF_SOURCE_ARCHIVE_URL:
    body['sourceArchiveUrl'] = GCF_SOURCE_ARCHIVE_URL
elif GCF_SOURCE_REPOSITORY:
    body['sourceRepository'] = {
        'url': GCF_SOURCE_REPOSITORY
    }
elif GCF_ZIP_PATH:
    body['sourceUploadUrl'] = ''
    default_args['zip_path'] = GCF_ZIP_PATH
elif GCF_SOURCE_UPLOAD_URL:
    body['sourceUploadUrl'] = GCF_SOURCE_UPLOAD_URL
else:
    raise Exception("Please provide one of the source_code parameters")
```

The code to create the operator:

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
deploy_task = GcfFunctionDeployOperator(
    task_id="gcf_deploy_task",
    project_id=GCP_PROJECT_ID,
    location=GCP_LOCATION,
    body=body,
    validate_body=GCP_VALIDATE_BODY
)
```

You can also create the operator without project id - project id will be retrieved from the GCP connection used:

airflow/contrib/example_dags/example_gcp_function.pyView Source

```
deploy2_task = GcfFunctionDeployOperator(
    task_id="gcf_deploy2_task",
    location=GCP_LOCATION,
    body=body,
    validate_body=GCP_VALIDATE_BODY
)
```

Templating

```
template_fields = ('project_id', 'location', 'gcp_conn_id', 'api_version')
```

Troubleshooting

If during the deploy you see an error similar to:

"HttpError 403: Missing necessary permission iam.serviceAccounts.actAs for on resource project-name@appspot.gserviceaccount.com. Please grant the roles/iam.serviceAccountUser role."

it means that your service account does not have the correct Cloud IAM permissions.

1. Assign your Service Account the Cloud Functions Developer role.
2. Grant the user the Cloud IAM Service Account User role on the Cloud Functions runtime service account.

The typical way of assigning Cloud IAM permissions with *gcloud* is shown below. Just replace PROJECT_ID with ID of your Google Cloud Platform project and SERVICE_ACCOUNT_EMAIL with the email ID of your service account.

```
gcloud iam service-accounts add-iam-policy-binding \
  PROJECT_ID@appspot.gserviceaccount.com \
  --member="serviceAccount:[SERVICE_ACCOUNT_EMAIL]" \
  --role="roles/iam.serviceAccountUser"
```

You can also do that via the GCP Web console.

See Adding the IAM service agent user role to the runtime service for details.

If the source code for your function is in Google Source Repository, make sure that your service account has the Source Repository Viewer role so that the source code can be downloaded if necessary.

More information

See Google Cloud API documentation to create a function.

Google Cloud Storage Operators

- *GoogleCloudStorageToBigQueryOperator*
- *GoogleCloudStorageBucketCreateAclEntryOperator*
- *GoogleCloudStorageObjectCreateAclEntryOperator*

GoogleCloudStorageToBigQueryOperator

Use the `GoogleCloudStorageToBigQueryOperator` to execute a BigQuery load job.

airflow/contrib/example_dags/example_gcs_to_bq_operator.pyView Source

```
load_csv = gcs_to_bq.GoogleCloudStorageToBigQueryOperator(
    task_id='gcs_to_bq_example',
    bucket='cloud-samples-data',
    source_objects=['bigquery/us-states/us-states.csv'],
    destination_project_dataset_table='airflow_test.gcs_to_bq_table',
    schema_fields=[
        {'name': 'name', 'type': 'STRING', 'mode': 'NULLABLE'},
        {'name': 'post_abbr', 'type': 'STRING', 'mode': 'NULLABLE'},
    ],
    write_disposition='WRITE_TRUNCATE',
    dag=dag)
```

GoogleCloudStorageBucketCreateAclEntryOperator

Creates a new ACL entry on the specified bucket.

For parameter definition, take a look at `GoogleCloudStorageBucketCreateAclEntryOperator`

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcs_acl.pyView Source

```
GCS_ACL_BUCKET = os.environ.get('GCS_ACL_BUCKET', 'example-bucket')
GCS_ACL_OBJECT = os.environ.get('GCS_ACL_OBJECT', 'example-object')
GCS_ACL_ENTITY = os.environ.get('GCS_ACL_ENTITY', 'example-entity')
GCS_ACL_BUCKET_ROLE = os.environ.get('GCS_ACL_BUCKET_ROLE', 'example-bucket-role')
GCS_ACL_OBJECT_ROLE = os.environ.get('GCS_ACL_OBJECT_ROLE', 'example-object-role')
```

Using the operator

airflow/contrib/example_dags/example_gcs_acl.pyView Source

```
gcs_bucket_create_acl_entry_task = GoogleCloudStorageBucketCreateAclEntryOperator(
    bucket=GCS_ACL_BUCKET,
    entity=GCS_ACL_ENTITY,
```

(continues on next page)

(continued from previous page)

```
    role=GCS_ACL_BUCKET_ROLE,
    task_id="gcs_bucket_create_acl_entry_task"
)
```

Templating

```
template_fields = ('bucket', 'entity', 'role', 'user_project')
```

More information

See Google Cloud Storage Documentation to create a new ACL entry for a bucket.

GoogleCloudStorageObjectCreateAclEntryOperator

Creates a new ACL entry on the specified object.

For parameter definition, take a look at *GoogleCloudStorageObjectCreateAclEntryOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcs_acl.pyView Source

```
GCS_ACL_BUCKET = os.environ.get('GCS_ACL_BUCKET', 'example-bucket')
GCS_ACL_OBJECT = os.environ.get('GCS_ACL_OBJECT', 'example-object')
GCS_ACL_ENTITY = os.environ.get('GCS_ACL_ENTITY', 'example-entity')
GCS_ACL_BUCKET_ROLE = os.environ.get('GCS_ACL_BUCKET_ROLE', 'example-bucket-role')
GCS_ACL_OBJECT_ROLE = os.environ.get('GCS_ACL_OBJECT_ROLE', 'example-object-role')
```

Using the operator

airflow/contrib/example_dags/example_gcs_acl.pyView Source

```
gcs_object_create_acl_entry_task = GoogleCloudStorageObjectCreateAclEntryOperator(
    bucket=GCS_ACL_BUCKET,
    object_name=GCS_ACL_OBJECT,
    entity=GCS_ACL_ENTITY,
    role=GCS_ACL_OBJECT_ROLE,
    task_id="gcs_object_create_acl_entry_task"
)
```

Templating

```
template_fields = ('bucket', 'object_name', 'entity', 'role', 'generation',
                   'user_project')
```

More information

See Google Cloud Storage insert documentation to create a ACL entry for ObjectAccess.

Google Cloud Natural Language Operators

The Google Cloud Natural Language can be used to reveal the structure and meaning of text via powerful machine learning models. You can use it to extract information about people, places, events and much more, mentioned in text documents, news articles or blog posts. You can use it to understand sentiment about your product on social media or parse intent from customer conversations happening in a call center or a messaging app.

- *Prerequisite Tasks*
- *Documents*
- *Analyzing Entities*
- *Analyzing Entity Sentiment*
- *Analyzing Sentiment*
- *Classifying Content*
- *Reference*

Prerequisite Tasks

To use these operators, you must do a few things:

- Select or create a Cloud Platform project using Cloud Console.
- Enable billing for your project, as described in Google Cloud documentation.
- Enable API, as described in Cloud Console documentation.
- Install API libraries via **pip**.

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

  Detailed information is available *Installation*
- *Setup Connection*.

Documents

Each operator uses a `Document` for representing text.

Here is an example of document with text provided as a string:

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
TEXT = """
Airflow is a platform to programmatically author, schedule and monitor workflows.

Use Airflow to author workflows as Directed Acyclic Graphs (DAGs) of tasks. The␣
↪Airflow scheduler executes
 your tasks on an array of workers while following the specified dependencies. Rich␣
↪command line utilities
```

(continues on next page)

(continued from previous page)

```
 make performing complex surgeries on DAGs a snap. The rich user interface makes it␣
↪easy to visualize
 pipelines running in production, monitor progress, and troubleshoot issues when␣
↪needed.
"""
document = Document(content=TEXT, type="PLAIN_TEXT")
```

In addition to supplying string, a document can refer to content stored in Google Cloud Storage.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
GCS_CONTENT_URI = "gs://my-text-bucket/sentiment-me.txt"
document_gcs = Document(gcs_content_uri=GCS_CONTENT_URI, type="PLAIN_TEXT")
```

Analyzing Entities

Entity Analysis inspects the given text for known entities (proper nouns such as public figures, landmarks, etc.), and returns information about those entities. Entity analysis is performed with the *CloudLanguageAnalyzeEntitiesOperator* operator.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_entities = CloudLanguageAnalyzeEntitiesOperator(document=document, task_id=
↪"analyze_entities")
```

You can use *Jinja templating* with `document`, `gcp_conn_id` parameters which allows you to dynamically determine values. The result is saved to *XCom*, which allows it to be used by other operators.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_entities_result = BashOperator(
    bash_command="echo \"{{ task_instance.xcom_pull('analyze_entities') }}\"",
    task_id="analyze_entities_result",
)
```

Analyzing Entity Sentiment

Sentiment Analysis inspects the given text and identifies the prevailing emotional opinion within the text, especially to determine a writer's attitude as positive, negative, or neutral. Sentiment analysis is performed through the *CloudLanguageAnalyzeEntitySentimentOperator* operator.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_entity_sentiment = CloudLanguageAnalyzeEntitySentimentOperator(
    document=document, task_id="analyze_entity_sentiment"
)
```

You can use *Jinja templating* with `document`, `gcp_conn_id` parameters which allows you to dynamically determine values. The result is saved to *XCom*, which allows it to be used by other operators.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_entity_sentiment_result = BashOperator(
    bash_command="echo \"{{ task_instance.xcom_pull('analyze_entity_sentiment') }}\"",
    task_id="analyze_entity_sentiment_result",
)
```

Analyzing Sentiment

Sentiment Analysis inspects the given text and identifies the prevailing emotional opinion within the text, especially to determine a writer's attitude as positive, negative, or neutral. Sentiment analysis is performed through the `CloudLanguageAnalyzeSentimentOperator` operator.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_sentiment = CloudLanguageAnalyzeSentimentOperator(document=document, task_id=
↪"analyze_sentiment")
```

You can use *Jinja templating* with `document`, `gcp_conn_id` parameters which allows you to dynamically determine values. The result is saved to *XCom*, which allows it to be used by other operators.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_sentiment_result = BashOperator(
    bash_command="echo \"{{ task_instance.xcom_pull('analyze_sentiment') }}\"",
    task_id="analyze_sentiment_result",
)
```

Classifying Content

Content Classification analyzes a document and returns a list of content categories that apply to the text found in the document. To classify the content in a document, use the `CloudLanguageClassifyTextOperator` operator.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_classify_text = CloudLanguageClassifyTextOperator(
    document=document, task_id="analyze_classify_text"
)
```

You can use *Jinja templating* with `document`, `gcp_conn_id` parameters which allows you to dynamically determine values. The result is saved to *XCom*, which allows it to be used by other operators.

airflow/contrib/example_dags/example_gcp_natural_language.pyView Source

```
analyze_classify_text_result = BashOperator(
    bash_command="echo \"{{ task_instance.xcom_pull('analyze_classify_text') }}\"",
    task_id="analyze_classify_text_result",
)
```

Reference

For further information, look at:

- Client Library Documentation
- Product Documentation

Google Cloud Spanner Operators

- *CloudSpannerInstanceDatabaseDeleteOperator*

- *CloudSpannerInstanceDatabaseDeployOperator*
- *CloudSpannerInstanceDatabaseUpdateOperator*
- *CloudSpannerInstanceDatabaseQueryOperator*
- *CloudSpannerInstanceDeleteOperator*

CloudSpannerInstanceDatabaseDeleteOperator

Deletes a database from the specified Cloud Spanner instance. If the database does not exist, no action is taken, and the operator succeeds.

For parameter definition, take a look at `CloudSpannerInstanceDatabaseDeleteOperator`.

Arguments

Some arguments in the example DAG are taken from environment variables.

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_SPANNER_INSTANCE_ID = os.environ.get('GCP_SPANNER_INSTANCE_ID', 'testinstance')
GCP_SPANNER_DATABASE_ID = os.environ.get('GCP_SPANNER_DATABASE_ID', 'testdatabase')
GCP_SPANNER_CONFIG_NAME = os.environ.get('GCP_SPANNER_CONFIG_NAME',
                                         'projects/example-project/instanceConfigs/
↪eur3')
GCP_SPANNER_NODE_COUNT = os.environ.get('GCP_SPANNER_NODE_COUNT', '1')
GCP_SPANNER_DISPLAY_NAME = os.environ.get('GCP_SPANNER_DISPLAY_NAME', 'Test Instance')
# OPERATION_ID should be unique per operation
OPERATION_ID = 'unique_operation_id'
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
spanner_database_delete_task = CloudSpannerInstanceDatabaseDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    task_id='spanner_database_delete_task'
)
spanner_database_delete_task2 = CloudSpannerInstanceDatabaseDeleteOperator(
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    task_id='spanner_database_delete_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance_id', 'gcp_conn_id')
```

More information

See Google Cloud Spanner API documentation to drop an instance of a database.

CloudSpannerInstanceDatabaseDeployOperator

Creates a new Cloud Spanner database in the specified instance, or if the desired database exists, assumes success with no changes applied to database configuration. No structure of the database is verified - it's enough if the database exists with the same name.

For parameter definition, take a look at *CloudSpannerInstanceDatabaseDeployOperator*.

Arguments

Some arguments in the example DAG are taken from environment variables.

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_SPANNER_INSTANCE_ID = os.environ.get('GCP_SPANNER_INSTANCE_ID', 'testinstance')
GCP_SPANNER_DATABASE_ID = os.environ.get('GCP_SPANNER_DATABASE_ID', 'testdatabase')
GCP_SPANNER_CONFIG_NAME = os.environ.get('GCP_SPANNER_CONFIG_NAME',
                                         'projects/example-project/instanceConfigs/
↪eur3')
GCP_SPANNER_NODE_COUNT = os.environ.get('GCP_SPANNER_NODE_COUNT', '1')
GCP_SPANNER_DISPLAY_NAME = os.environ.get('GCP_SPANNER_DISPLAY_NAME', 'Test Instance')
# OPERATION_ID should be unique per operation
OPERATION_ID = 'unique_operation_id'
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
spanner_database_deploy_task = CloudSpannerInstanceDatabaseDeployOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    ddl_statements=[
        "CREATE TABLE my_table1 (id INT64, name STRING(MAX)) PRIMARY KEY (id)",
        "CREATE TABLE my_table2 (id INT64, name STRING(MAX)) PRIMARY KEY (id)",
    ],
    task_id='spanner_database_deploy_task'
)
spanner_database_deploy_task2 = CloudSpannerInstanceDatabaseDeployOperator(
    instance_id=GCP_SPANNER_INSTANCE_ID,
```

(continues on next page)

(continued from previous page)

```
    database_id=GCP_SPANNER_DATABASE_ID,
    ddl_statements=[
        "CREATE TABLE my_table1 (id INT64, name STRING(MAX)) PRIMARY KEY (id)",
        "CREATE TABLE my_table2 (id INT64, name STRING(MAX)) PRIMARY KEY (id)",
    ],
    task_id='spanner_database_deploy_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance_id', 'database_id', 'ddl_statements',
                   'gcp_conn_id')
template_ext = ('.sql', )
```

More information

See Google Cloud Spanner API documentation to create a new database.

CloudSpannerInstanceDatabaseUpdateOperator

Runs a DDL query in a Cloud Spanner database and allows you to modify the structure of an existing database.

You can optionally specify an operation_id parameter which simplifies determining whether the statements were executed in case the update_database call is replayed (idempotency check). The operation_id should be unique within the database, and must be a valid identifier: *[a-z][a-z0-9_]**. More information can be found in the documentation of updateDdl API

For parameter definition take a look at `CloudSpannerInstanceDatabaseUpdateOperator`.

Arguments

Some arguments in the example DAG are taken from environment variables.

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_SPANNER_INSTANCE_ID = os.environ.get('GCP_SPANNER_INSTANCE_ID', 'testinstance')
GCP_SPANNER_DATABASE_ID = os.environ.get('GCP_SPANNER_DATABASE_ID', 'testdatabase')
GCP_SPANNER_CONFIG_NAME = os.environ.get('GCP_SPANNER_CONFIG_NAME',
                                         'projects/example-project/instanceConfigs/
↪eur3')
GCP_SPANNER_NODE_COUNT = os.environ.get('GCP_SPANNER_NODE_COUNT', '1')
GCP_SPANNER_DISPLAY_NAME = os.environ.get('GCP_SPANNER_DISPLAY_NAME', 'Test Instance')
# OPERATION_ID should be unique per operation
OPERATION_ID = 'unique_operation_id'
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
spanner_database_update_task = CloudSpannerInstanceDatabaseUpdateOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    ddl_statements=[
        "CREATE TABLE my_table3 (id INT64, name STRING(MAX)) PRIMARY KEY (id)",
    ],
    task_id='spanner_database_update_task'
)
```

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
spanner_database_update_idempotent1_task = CloudSpannerInstanceDatabaseUpdateOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    operation_id=OPERATION_ID,
    ddl_statements=[
        "CREATE TABLE my_table_unique (id INT64, name STRING(MAX)) PRIMARY KEY (id)",
    ],
    task_id='spanner_database_update_idempotent1_task'
)
spanner_database_update_idempotent2_task = CloudSpannerInstanceDatabaseUpdateOperator(
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    operation_id=OPERATION_ID,
    ddl_statements=[
        "CREATE TABLE my_table_unique (id INT64, name STRING(MAX)) PRIMARY KEY (id)",
    ],
    task_id='spanner_database_update_idempotent2_task'
)
```

Templating

```
template_fields = ('project_id', 'instance_id', 'database_id', 'ddl_statements',
                   'gcp_conn_id')
template_ext = ('.sql', )
```

More information

See Google Cloud Spanner API documentation for database update_ddl.

CloudSpannerInstanceDatabaseQueryOperator

Executes an arbitrary DML query (INSERT, UPDATE, DELETE).

For parameter definition take a look at `CloudSpannerInstanceDatabaseQueryOperator`.

Arguments

Some arguments in the example DAG are taken from environment variables.

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_SPANNER_INSTANCE_ID = os.environ.get('GCP_SPANNER_INSTANCE_ID', 'testinstance')
GCP_SPANNER_DATABASE_ID = os.environ.get('GCP_SPANNER_DATABASE_ID', 'testdatabase')
GCP_SPANNER_CONFIG_NAME = os.environ.get('GCP_SPANNER_CONFIG_NAME',
                                         'projects/example-project/instanceConfigs/
↪eur3')
GCP_SPANNER_NODE_COUNT = os.environ.get('GCP_SPANNER_NODE_COUNT', '1')
GCP_SPANNER_DISPLAY_NAME = os.environ.get('GCP_SPANNER_DISPLAY_NAME', 'Test Instance')
# OPERATION_ID should be unique per operation
OPERATION_ID = 'unique_operation_id'
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
spanner_instance_query_task = CloudSpannerInstanceDatabaseQueryOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    query=["DELETE FROM my_table2 WHERE true"],
    task_id='spanner_instance_query_task'
)
spanner_instance_query_task2 = CloudSpannerInstanceDatabaseQueryOperator(
    instance_id=GCP_SPANNER_INSTANCE_ID,
    database_id=GCP_SPANNER_DATABASE_ID,
    query=["DELETE FROM my_table2 WHERE true"],
    task_id='spanner_instance_query_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance_id', 'database_id', 'query', 'gcp_conn_id')
template_ext = ('.sql',)
```

More information

See Google Cloud Spanner API documentation for more information about DML syntax.

CloudSpannerInstanceDeleteOperator

Deletes a Cloud Spanner instance. If an instance does not exist, no action is taken, and the operator succeeds.

For parameter definition take a look at *CloudSpannerInstanceDeleteOperator*.

Arguments

Some arguments in the example DAG are taken from environment variables:

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_SPANNER_INSTANCE_ID = os.environ.get('GCP_SPANNER_INSTANCE_ID', 'testinstance')
GCP_SPANNER_DATABASE_ID = os.environ.get('GCP_SPANNER_DATABASE_ID', 'testdatabase')
GCP_SPANNER_CONFIG_NAME = os.environ.get('GCP_SPANNER_CONFIG_NAME',
                                         'projects/example-project/instanceConfigs/
↪eur3')
GCP_SPANNER_NODE_COUNT = os.environ.get('GCP_SPANNER_NODE_COUNT', '1')
GCP_SPANNER_DISPLAY_NAME = os.environ.get('GCP_SPANNER_DISPLAY_NAME', 'Test Instance')
# OPERATION_ID should be unique per operation
OPERATION_ID = 'unique_operation_id'
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_spanner.pyView Source

```
spanner_instance_delete_task = CloudSpannerInstanceDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance_id=GCP_SPANNER_INSTANCE_ID,
    task_id='spanner_instance_delete_task'
)
spanner_instance_delete_task2 = CloudSpannerInstanceDeleteOperator(
    instance_id=GCP_SPANNER_INSTANCE_ID,
    task_id='spanner_instance_delete_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance_id', 'gcp_conn_id')
```

More information

See Google Cloud Spanner API documentation to delete an instance.

Google Cloud Text to Speech Operators

GcpTextToSpeechSynthesizeOperator

Synthesizes text to audio file and stores it to Google Cloud Storage

For parameter definition, take a look at *airflow.contrib.operators.gcp_text_to_speech_operator.GcpTextToSpeechSynthesizeOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
GCP_PROJECT_ID = os.environ.get("GCP_PROJECT_ID", "example-project")
BUCKET_NAME = os.environ.get("GCP_SPEECH_TEST_BUCKET", "gcp-speech-test-bucket")
```

input, voice and audio_config arguments need to be dicts or objects of corresponding classes from google.cloud.texttospeech_v1.types module

for more information, see: https://googleapis.github.io/google-cloud-python/latest/texttospeech/gapic/v1/api.html#google.cloud.texttospeech_v1.TextToSpeechClient.synthesize_speech

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
INPUT = {"text": "Sample text for demo purposes"}
VOICE = {"language_code": "en-US", "ssml_gender": "FEMALE"}
AUDIO_CONFIG = {"audio_encoding": "LINEAR16"}
```

filename is a simple string argument:

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
FILENAME = "gcp-speech-test-file"
```

Using the operator

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
text_to_speech_synthesize_task = GcpTextToSpeechSynthesizeOperator(
    project_id=GCP_PROJECT_ID,
    input_data=INPUT,
    voice=VOICE,
    audio_config=AUDIO_CONFIG,
    target_bucket_name=BUCKET_NAME,
    target_filename=FILENAME,
    task_id="text_to_speech_synthesize_task",
)
```

Templating

```
template_fields = (
    "input_data",
    "voice",
    "audio_config",
    "project_id",
    "gcp_conn_id",
    "target_bucket_name",
    "target_filename",
)
```

Google Cloud Speech to Text Operators

GcpSpeechToTextRecognizeSpeechOperator

Recognizes speech in audio input and returns text.

For parameter definition, take a look at *airflow.contrib.operators.gcp_speech_to_text_operator.GcpSpeechToTextRecognizeSpeechOperator*

Arguments

config and audio arguments need to be dicts or objects of corresponding classes from google.cloud.speech_v1.types module

for more information, see: https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/api.html#google.cloud.speech_v1.SpeechClient.recognize

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
INPUT = {"text": "Sample text for demo purposes"}
VOICE = {"language_code": "en-US", "ssml_gender": "FEMALE"}
AUDIO_CONFIG = {"audio_encoding": "LINEAR16"}
```

filename is a simple string argument:

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
CONFIG = {"encoding": "LINEAR16", "language_code": "en_US"}
AUDIO = {"uri": "gs://{bucket}/{object}".format(bucket=BUCKET_NAME, object=FILENAME)}
```

Using the operator

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
speech_to_text_recognize_task = GcpSpeechToTextRecognizeSpeechOperator(
    project_id=GCP_PROJECT_ID, config=CONFIG, audio=AUDIO, task_id="speech_to_text_
↪recognize_task"
)
```

Templating

```
template_fields = ("audio", "config", "project_id", "gcp_conn_id", "timeout")
```

Google Cloud Sql Operators

- *CloudSqlInstanceDatabaseCreateOperator*
- *CloudSqlInstanceDatabaseDeleteOperator*
- *CloudSqlInstanceDatabasePatchOperator*
- *CloudSqlInstanceDeleteOperator*
- *CloudSqlInstanceExportOperator*
- *CloudSqlInstanceImportOperator*
- *CloudSqlInstanceCreateOperator*
- *CloudSqlInstancePatchOperator*

- *CloudSqlQueryOperator*

CloudSqlInstanceDatabaseCreateOperator

Creates a new database inside a Cloud SQL instance.

For parameter definition, take a look at `CloudSqlInstanceDatabaseCreateOperator`.

Arguments

Some arguments in the example DAG are taken from environment variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_db_create_task = CloudSqlInstanceDatabaseCreateOperator(
    project_id=GCP_PROJECT_ID,
    body=db_create_body,
    instance=INSTANCE_NAME,
    task_id='sql_db_create_task'
)
sql_db_create_task2 = CloudSqlInstanceDatabaseCreateOperator(
    body=db_create_body,
    instance=INSTANCE_NAME,
    task_id='sql_db_create_task2'
)
```

Example request body:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
db_create_body = {
    "instance": INSTANCE_NAME,
    "name": DB_NAME,
    "project": GCP_PROJECT_ID
}
```

Templating

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud SQL API documentation for to create a new database inside the instance.

CloudSqlInstanceDatabaseDeleteOperator

Deletes a database from a Cloud SQL instance.

For parameter definition, take a look at `CloudSqlInstanceDatabaseDeleteOperator`.

Arguments

Some arguments in the example DAG are taken from environment variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_db_delete_task = CloudSqlInstanceDatabaseDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance=INSTANCE_NAME,
    database=DB_NAME,
    task_id='sql_db_delete_task'
)
sql_db_delete_task2 = CloudSqlInstanceDatabaseDeleteOperator(
    instance=INSTANCE_NAME,
    database=DB_NAME,
    task_id='sql_db_delete_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance', 'database', 'gcp_conn_id',
                   'api_version')
```

More information

See Google Cloud SQL API documentation to delete a database.

CloudSqlInstanceDatabasePatchOperator

Updates a resource containing information about a database inside a Cloud SQL instance using patch semantics. See: https://cloud.google.com/sql/docs/mysql/admin-api/how-tos/performance#patch

For parameter definition, take a look at *CloudSqlInstanceDatabasePatchOperator*.

Arguments

Some arguments in the example DAG are taken from environment variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_db_patch_task = CloudSqlInstanceDatabasePatchOperator(
    project_id=GCP_PROJECT_ID,
    body=db_patch_body,
    instance=INSTANCE_NAME,
    database=DB_NAME,
    task_id='sql_db_patch_task'
)
sql_db_patch_task2 = CloudSqlInstanceDatabasePatchOperator(
    body=db_patch_body,
    instance=INSTANCE_NAME,
    database=DB_NAME,
    task_id='sql_db_patch_task2'
)
```

Example request body:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
db_patch_body = {
    "charset": "utf16",
    "collation": "utf16_general_ci"
}
```

Templating

```
template_fields = ('project_id', 'instance', 'database', 'gcp_conn_id',
                   'api_version')
```

More information

See Google Cloud SQL API documentation to update a database.

CloudSqlInstanceDeleteOperator

Deletes a Cloud SQL instance in Google Cloud Platform.

It is also used for deleting read and failover replicas.

For parameter definition, take a look at *CloudSqlInstanceDeleteOperator*.

Arguments

Some arguments in the example DAG are taken from OS environment variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_instance_delete_task = CloudSqlInstanceDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance=INSTANCE_NAME,
    task_id='sql_instance_delete_task'
)
sql_instance_delete_task2 = CloudSqlInstanceDeleteOperator(
    instance=INSTANCE_NAME2,
    task_id='sql_instance_delete_task2'
)
```

Note: If the instance has read or failover replicas you need to delete them before you delete the primary instance. Replicas are deleted the same way as primary instances:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_instance_failover_replica_delete_task = CloudSqlInstanceDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance=FAILOVER_REPLICA_NAME,
    task_id='sql_instance_failover_replica_delete_task'
)

sql_instance_read_replica_delete_task = CloudSqlInstanceDeleteOperator(
    project_id=GCP_PROJECT_ID,
    instance=READ_REPLICA_NAME,
    task_id='sql_instance_read_replica_delete_task'
)
```

Templating

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud SQL API documentation to delete a SQL instance.

CloudSqlInstanceExportOperator

Exports data from a Cloud SQL instance to a Cloud Storage bucket as a SQL dump or CSV file.

Note: This operator is idempotent. If executed multiple times with the same export file URI, the export file in GCS will simply be overridden.

For parameter definition take a look at `CloudSqlInstanceExportOperator`.

Arguments

Some arguments in the example DAG are taken from Airflow variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
EXPORT_URI = os.environ.get('GCSQL_MYSQL_EXPORT_URI', 'gs://bucketName/fileName')
IMPORT_URI = os.environ.get('GCSQL_MYSQL_IMPORT_URI', 'gs://bucketName/fileName')
```

Example body defining the export operation:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
export_body = {
    "exportContext": {
        "fileType": "sql",
        "uri": EXPORT_URI,
        "sqlExportOptions": {
            "schemaOnly": False
        }
    }
}
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_export_task = CloudSqlInstanceExportOperator(
    project_id=GCP_PROJECT_ID,
    body=export_body,
    instance=INSTANCE_NAME,
    task_id='sql_export_task'
)
sql_export_task2 = CloudSqlInstanceExportOperator(
    body=export_body,
    instance=INSTANCE_NAME,
    task_id='sql_export_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud SQL API documentation to export data.

Troubleshooting

If you receive an "Unauthorized" error in GCP, make sure that the service account of the Cloud SQL instance is authorized to write to the selected GCS bucket.

It is not the service account configured in Airflow that communicates with GCS, but rather the service account of the particular Cloud SQL instance.

To grant the service account with the appropriate WRITE permissions for the GCS bucket you can use the *`GoogleCloudStorageBucketCreateAclEntryOperator`*, as shown in the example:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_gcp_add_bucket_permission_task = GoogleCloudStorageBucketCreateAclEntryOperator(
    entity="user-{{ task_instance.xcom_pull("
           "'sql_instance_create_task', key='service_account_email') "
           "}}",
    role="WRITER",
    bucket=export_url_split[1],  # netloc (bucket)
    task_id='sql_gcp_add_bucket_permission_task'
)
```

CloudSqlInstanceImportOperator

Imports data into a Cloud SQL instance from a SQL dump or CSV file in Cloud Storage.

CSV import:

This operator is NOT idempotent for a CSV import. If the same file is imported multiple times, the imported data will be duplicated in the database. Moreover, if there are any unique constraints the duplicate import may result in an error.

SQL import:

This operator is idempotent for a SQL import if it was also exported by Cloud SQL. The exported SQL contains 'DROP TABLE IF EXISTS' statements for all tables to be imported.

If the import file was generated in a different way, idempotence is not guaranteed. It has to be ensured on the SQL file level.

For parameter definition take a look at `CloudSqlInstanceImportOperator`.

Arguments

Some arguments in the example DAG are taken from Airflow variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
EXPORT_URI = os.environ.get('GCSQL_MYSQL_EXPORT_URI', 'gs://bucketName/fileName')
IMPORT_URI = os.environ.get('GCSQL_MYSQL_IMPORT_URI', 'gs://bucketName/fileName')
```

Example body defining the import operation:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
import_body = {
    "importContext": {
        "fileType": "sql",
        "uri": IMPORT_URI
    }
}
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_import_task = CloudSqlInstanceImportOperator(
    project_id=GCP_PROJECT_ID,
    body=import_body,
    instance=INSTANCE_NAME2,
    task_id='sql_import_task'
)
sql_import_task2 = CloudSqlInstanceImportOperator(
    body=import_body,
    instance=INSTANCE_NAME2,
    task_id='sql_import_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud SQL API documentation to import data.

Troubleshooting

If you receive an "Unauthorized" error in GCP, make sure that the service account of the Cloud SQL instance is authorized to read from the selected GCS object.

It is not the service account configured in Airflow that communicates with GCS, but rather the service account of the particular Cloud SQL instance.

To grant the service account with the appropriate READ permissions for the GCS object you can use the *GoogleCloudStorageObjectCreateAclEntryOperator*, as shown in the example:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_gcp_add_object_permission_task = GoogleCloudStorageObjectCreateAclEntryOperator(
    entity="user-{{ task_instance.xcom_pull("
           "'sql_instance_create_task2', key='service_account_email')"
           " }}",
    role="READER",
    bucket=import_url_split[1],  # netloc (bucket)
    object_name=import_url_split[2][1:],  # path (strip first '/')
    task_id='sql_gcp_add_object_permission_task',
)
prev_task = next_dep(sql_gcp_add_object_permission_task, prev_task)

# For import to work we also need to add the Cloud SQL instance's Service Account
# write access to the whole bucket!.
sql_gcp_add_bucket_permission_2_task = GoogleCloudStorageBucketCreateAclEntryOperator(
    entity="user-{{ task_instance.xcom_pull("
           "'sql_instance_create_task2', key='service_account_email') "
           "}}",
    role="WRITER",
    bucket=import_url_split[1],  # netloc
    task_id='sql_gcp_add_bucket_permission_2_task',
)
```

CloudSqlInstanceCreateOperator

Creates a new Cloud SQL instance in Google Cloud Platform.

It is also used for creating read replicas.

For parameter definition, take a look at *CloudSqlInstanceCreateOperator*.

If an instance with the same name exists, no action will be taken and the operator will succeed.

Arguments

Some arguments in the example DAG are taken from OS environment variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

Some other arguments are created based on the arguments above:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
FAILOVER_REPLICA_NAME = INSTANCE_NAME + "-failover-replica"
READ_REPLICA_NAME = INSTANCE_NAME + "-read-replica"
```

Example body defining the instance with failover replica:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
body = {
    "name": INSTANCE_NAME,
    "settings": {
        "tier": "db-n1-standard-1",
        "backupConfiguration": {
            "binaryLogEnabled": True,
            "enabled": True,
            "startTime": "05:00"
        },
        "activationPolicy": "ALWAYS",
        "dataDiskSizeGb": 30,
        "dataDiskType": "PD_SSD",
        "databaseFlags": [],
        "ipConfiguration": {
            "ipv4Enabled": True,
            "requireSsl": True,
        },
        "locationPreference": {
            "zone": "europe-west4-a"
        },
        "maintenanceWindow": {
            "hour": 5,
            "day": 7,
            "updateTrack": "canary"
        },
        "pricingPlan": "PER_USE",
        "replicationType": "ASYNCHRONOUS",
        "storageAutoResize": True,
        "storageAutoResizeLimit": 0,
        "userLabels": {
            "my-key": "my-value"
        }
    },
    "failoverReplica": {
        "name": FAILOVER_REPLICA_NAME
    },
    "databaseVersion": "MYSQL_5_7",
```

(continues on next page)

(continued from previous page)

```
    "region": "europe-west4",
}
```

Example body defining read replica for the instance above:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
read_replica_body = {
    "name": READ_REPLICA_NAME,
    "settings": {
        "tier": "db-n1-standard-1",
    },
    "databaseVersion": "MYSQL_5_7",
    "region": "europe-west4",
    "masterInstanceName": INSTANCE_NAME,
}
```

Note: Failover replicas are created together with the instance in a single task. Read replicas need to be created in separate tasks.

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_instance_create_task = CloudSqlInstanceCreateOperator(
    project_id=GCP_PROJECT_ID,
    body=body,
    instance=INSTANCE_NAME,
    task_id='sql_instance_create_task'
)
```

Templating

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud SQL API documentation to create an instance.

CloudSqlInstancePatchOperator

Updates settings of a Cloud SQL instance in Google Cloud Platform (partial update).

For parameter definition, take a look at *CloudSqlInstancePatchOperator*.

This is a partial update, so only values for the settings specified in the body will be set / updated. The rest of the existing instance's configuration will remain unchanged.

Arguments

Some arguments in the example DAG are taken from OS environment variables:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME', 'test-mysql')
INSTANCE_NAME2 = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME2', 'test-mysql2')
DB_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'testdb')
```

Example body defining the instance:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
patch_body = {
    "name": INSTANCE_NAME,
    "settings": {
        "dataDiskSizeGb": 35,
        "maintenanceWindow": {
            "hour": 3,
            "day": 6,
            "updateTrack": "canary"
        },
        "userLabels": {
            "my-key-patch": "my-value-patch"
        }
    }
}
```

Using the operator

You can create the operator with or without project id. If project id is missing it will be retrieved from the GCP connection used. Both variants are shown:

airflow/contrib/example_dags/example_gcp_sql.pyView Source

```
sql_instance_patch_task = CloudSqlInstancePatchOperator(
    project_id=GCP_PROJECT_ID,
    body=patch_body,
    instance=INSTANCE_NAME,
    task_id='sql_instance_patch_task'
)

sql_instance_patch_task2 = CloudSqlInstancePatchOperator(
    body=patch_body,
    instance=INSTANCE_NAME,
    task_id='sql_instance_patch_task2'
)
```

Templating

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud SQL API documentation to patch an instance.

CloudSqlQueryOperator

Performs DDL or DML SQL queries in Google Cloud SQL instance. The DQL (retrieving data from Google Cloud SQL) is not supported. You might run the SELECT queries, but the results of those queries are discarded.

You can specify various connectivity methods to connect to running instance, starting from public IP plain connection through public IP with SSL or both TCP and socket connection via Cloud SQL Proxy. The proxy is downloaded and started/stopped dynamically as needed by the operator.

There is a *gcpcloudsql://* connection type that you should use to define what kind of connectivity you want the operator to use. The connection is a "meta" type of connection. It is not used to make an actual connectivity on its own, but it determines whether Cloud SQL Proxy should be started by *CloudSqlDatabaseHook* and what kind of database connection (Postgres or MySQL) should be created dynamically to connect to Cloud SQL via public IP address or via the proxy. The 'CloudSqlDatabaseHook' uses `CloudSqlProxyRunner` to manage Cloud SQL Proxy lifecycle (each task has its own Cloud SQL Proxy)

When you build connection, you should use connection parameters as described in `CloudSqlDatabaseHook`. You can see examples of connections below for all the possible types of connectivity. Such connection can be reused between different tasks (instances of *CloudSqlQueryOperator*). Each task will get their own proxy started if needed with their own TCP or UNIX socket.

For parameter definition, take a look at `CloudSqlQueryOperator`.

Since query operator can run arbitrary query, it cannot be guaranteed to be idempotent. SQL query designer should design the queries to be idempotent. For example, both Postgres and MySQL support CREATE TABLE IF NOT EXISTS statements that can be used to create tables in an idempotent way.

Arguments

If you define connection via *AIRFLOW_CONN_** URL defined in an environment variable, make sure the URL components in the URL are URL-encoded. See examples below for details.

Note that in case of SSL connections you need to have a mechanism to make the certificate/key files available in predefined locations for all the workers on which the operator can run. This can be provided for example by mounting NFS-like volumes in the same path for all the workers.

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_sql_query.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_REGION = os.environ.get('GCP_REGION', 'europe-west-1b')

GCSQL_POSTGRES_INSTANCE_NAME_QUERY = os.environ.get(
    'GCSQL_POSTGRES_INSTANCE_NAME_QUERY',
    'testpostgres')
GCSQL_POSTGRES_DATABASE_NAME = os.environ.get('GCSQL_POSTGRES_DATABASE_NAME',
                                              'postgresdb')
GCSQL_POSTGRES_USER = os.environ.get('GCSQL_POSTGRES_USER', 'postgres_user')
GCSQL_POSTGRES_PASSWORD = os.environ.get('GCSQL_POSTGRES_PASSWORD', 'password')
GCSQL_POSTGRES_PUBLIC_IP = os.environ.get('GCSQL_POSTGRES_PUBLIC_IP', '0.0.0.0')
GCSQL_POSTGRES_PUBLIC_PORT = os.environ.get('GCSQL_POSTGRES_PUBLIC_PORT', 5432)
```

(continues on next page)

(continued from previous page)

```
GCSQL_POSTGRES_CLIENT_CERT_FILE = os.environ.get('GCSQL_POSTGRES_CLIENT_CERT_FILE',
                                                 ".key/postgres-client-cert.pem")
GCSQL_POSTGRES_CLIENT_KEY_FILE = os.environ.get('GCSQL_POSTGRES_CLIENT_KEY_FILE',
                                                ".key/postgres-client-key.pem")
GCSQL_POSTGRES_SERVER_CA_FILE = os.environ.get('GCSQL_POSTGRES_SERVER_CA_FILE',
                                               ".key/postgres-server-ca.pem")

GCSQL_MYSQL_INSTANCE_NAME_QUERY = os.environ.get('GCSQL_MYSQL_INSTANCE_NAME_QUERY',
                                                 'testmysql')
GCSQL_MYSQL_DATABASE_NAME = os.environ.get('GCSQL_MYSQL_DATABASE_NAME', 'mysqldb')
GCSQL_MYSQL_USER = os.environ.get('GCSQL_MYSQL_USER', 'mysql_user')
GCSQL_MYSQL_PASSWORD = os.environ.get('GCSQL_MYSQL_PASSWORD', 'password')
GCSQL_MYSQL_PUBLIC_IP = os.environ.get('GCSQL_MYSQL_PUBLIC_IP', '0.0.0.0')
GCSQL_MYSQL_PUBLIC_PORT = os.environ.get('GCSQL_MYSQL_PUBLIC_PORT', 3306)
GCSQL_MYSQL_CLIENT_CERT_FILE = os.environ.get('GCSQL_MYSQL_CLIENT_CERT_FILE',
                                              ".key/mysql-client-cert.pem")
GCSQL_MYSQL_CLIENT_KEY_FILE = os.environ.get('GCSQL_MYSQL_CLIENT_KEY_FILE',
                                             ".key/mysql-client-key.pem")
GCSQL_MYSQL_SERVER_CA_FILE = os.environ.get('GCSQL_MYSQL_SERVER_CA_FILE',
                                            ".key/mysql-server-ca.pem")

SQL = [
    'CREATE TABLE IF NOT EXISTS TABLE_TEST (I INTEGER)',
    'CREATE TABLE IF NOT EXISTS TABLE_TEST (I INTEGER)',  # shows warnings logged
    'INSERT INTO TABLE_TEST VALUES (0)',
    'CREATE TABLE IF NOT EXISTS TABLE_TEST2 (I INTEGER)',
    'DROP TABLE TABLE_TEST',
    'DROP TABLE TABLE_TEST2',
]
```

Example connection definitions for all connectivity cases. Note that all the components of the connection URI should be URL-encoded:

airflow/contrib/example_dags/example_gcp_sql_query.pyView Source

```

HOME_DIR = expanduser("~")


def get_absolute_path(path):
    if path.startswith("/"):
        return path
    else:
        return os.path.join(HOME_DIR, path)


postgres_kwargs = dict(
    user=quote_plus(GCSQL_POSTGRES_USER),
    password=quote_plus(GCSQL_POSTGRES_PASSWORD),
    public_port=GCSQL_POSTGRES_PUBLIC_PORT,
    public_ip=quote_plus(GCSQL_POSTGRES_PUBLIC_IP),
    project_id=quote_plus(GCP_PROJECT_ID),
    location=quote_plus(GCP_REGION),
    instance=quote_plus(GCSQL_POSTGRES_INSTANCE_NAME_QUERY),
    database=quote_plus(GCSQL_POSTGRES_DATABASE_NAME),
    client_cert_file=quote_plus(get_absolute_path(GCSQL_POSTGRES_CLIENT_CERT_FILE)),
```

(continues on next page)

(continued from previous page)

```
    client_key_file=quote_plus(get_absolute_path(GCSQL_POSTGRES_CLIENT_KEY_FILE)),
    server_ca_file=quote_plus(get_absolute_path(GCSQL_POSTGRES_SERVER_CA_FILE))
)

# The connections below are created using one of the standard approaches - via␣
↪environment
# variables named AIRFLOW_CONN_* . The connections can also be created in the database
# of AIRFLOW (using command line or UI).

# Postgres: connect via proxy over TCP
os.environ['AIRFLOW_CONN_PROXY_POSTGRES_TCP'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=postgres&" \
    "project_id={project_id}&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=True&" \
    "sql_proxy_use_tcp=True".format(**postgres_kwargs)

# Postgres: connect via proxy over UNIX socket (specific proxy version)
os.environ['AIRFLOW_CONN_PROXY_POSTGRES_SOCKET'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=postgres&" \
    "project_id={project_id}&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=True&" \
    "sql_proxy_version=v1.13&" \
    "sql_proxy_use_tcp=False".format(**postgres_kwargs)

# Postgres: connect directly via TCP (non-SSL)
os.environ['AIRFLOW_CONN_PUBLIC_POSTGRES_TCP'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=postgres&" \
    "project_id={project_id}&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=False&" \
    "use_ssl=False".format(**postgres_kwargs)

# Postgres: connect directly via TCP (SSL)
os.environ['AIRFLOW_CONN_PUBLIC_POSTGRES_TCP_SSL'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=postgres&" \
    "project_id={project_id}&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=False&" \
    "use_ssl=True&" \
    "sslcert={client_cert_file}&" \
    "sslkey={client_key_file}&" \
    "sslrootcert={server_ca_file}"\
    .format(**postgres_kwargs)

mysql_kwargs = dict(
    user=quote_plus(GCSQL_MYSQL_USER),
    password=quote_plus(GCSQL_MYSQL_PASSWORD),
```

(continues on next page)

(continued from previous page)

```
    public_port=GCSQL_MYSQL_PUBLIC_PORT,
    public_ip=quote_plus(GCSQL_MYSQL_PUBLIC_IP),
    project_id=quote_plus(GCP_PROJECT_ID),
    location=quote_plus(GCP_REGION),
    instance=quote_plus(GCSQL_MYSQL_INSTANCE_NAME_QUERY),
    database=quote_plus(GCSQL_MYSQL_DATABASE_NAME),
    client_cert_file=quote_plus(get_absolute_path(GCSQL_MYSQL_CLIENT_CERT_FILE)),
    client_key_file=quote_plus(get_absolute_path(GCSQL_MYSQL_CLIENT_KEY_FILE)),
    server_ca_file=quote_plus(get_absolute_path(GCSQL_MYSQL_SERVER_CA_FILE))
)

# MySQL: connect via proxy over TCP (specific proxy version)
os.environ['AIRFLOW_CONN_PROXY_MYSQL_TCP'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=mysql&" \
    "project_id={project_id}&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=True&" \
    "sql_proxy_version=v1.13&" \
    "sql_proxy_use_tcp=True".format(**mysql_kwargs)

# MySQL: connect via proxy over UNIX socket using pre-downloaded Cloud Sql Proxy␣
↪binary
try:
    sql_proxy_binary_path = subprocess.check_output(
        ['which', 'cloud_sql_proxy']).decode('utf-8').rstrip()
except subprocess.CalledProcessError:
    sql_proxy_binary_path = "/tmp/anyhow_download_cloud_sql_proxy"

os.environ['AIRFLOW_CONN_PROXY_MYSQL_SOCKET'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=mysql&" \
    "project_id={project_id}&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=True&" \
    "sql_proxy_binary_path={sql_proxy_binary_path}&" \
    "sql_proxy_use_tcp=False".format(
        sql_proxy_binary_path=quote_plus(sql_proxy_binary_path), **mysql_kwargs)

# MySQL: connect directly via TCP (non-SSL)
os.environ['AIRFLOW_CONN_PUBLIC_MYSQL_TCP'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=mysql&" \
    "project_id={project_id}&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=False&" \
    "use_ssl=False".format(**mysql_kwargs)

# MySQL: connect directly via TCP (SSL) and with fixed Cloud Sql Proxy binary path
os.environ['AIRFLOW_CONN_PUBLIC_MYSQL_TCP_SSL'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=mysql&" \
    "project_id={project_id}&" \
    "location={location}&" \
```

(continues on next page)

(continued from previous page)

```
    "instance={instance}&" \
    "use_proxy=False&" \
    "use_ssl=True&" \
    "sslcert={client_cert_file}&" \
    "sslkey={client_key_file}&" \
    "sslrootcert={server_ca_file}".format(**mysql_kwargs)

# Special case: MySQL: connect directly via TCP (SSL) and with fixed Cloud Sql
# Proxy binary path AND with missing project_id

os.environ['AIRFLOW_CONN_PUBLIC_MYSQL_TCP_SSL_NO_PROJECT_ID'] = \
    "gcpcloudsql://{user}:{password}@{public_ip}:{public_port}/{database}?" \
    "database_type=mysql&" \
    "location={location}&" \
    "instance={instance}&" \
    "use_proxy=False&" \
    "use_ssl=True&" \
    "sslcert={client_cert_file}&" \
    "sslkey={client_key_file}&" \
    "sslrootcert={server_ca_file}".format(**mysql_kwargs)
```

Using the operator

Example operators below are using all connectivity options. Note connection id from the operator matches the *AIRFLOW_CONN_** postfix uppercase. This is standard AIRFLOW notation for defining connection via environment variables):

airflow/contrib/example_dags/example_gcp_sql_query.pyView Source

```
connection_names = [
    "proxy_postgres_tcp",
    "proxy_postgres_socket",
    "public_postgres_tcp",
    "public_postgres_tcp_ssl",
    "proxy_mysql_tcp",
    "proxy_mysql_socket",
    "public_mysql_tcp",
    "public_mysql_tcp_ssl",
    "public_mysql_tcp_ssl_no_project_id"
]

tasks = []


with models.DAG(
    dag_id='example_gcp_sql_query',
    default_args=default_args,
    schedule_interval=None
) as dag:
    prev_task = None

    for connection_name in connection_names:
```

(continues on next page)

(continued from previous page)

```
        task = CloudSqlQueryOperator(
            gcp_cloudsql_conn_id=connection_name,
            task_id="example_gcp_sql_task_" + connection_name,
            sql=SQL
        )
        tasks.append(task)
        if prev_task:
            prev_task >> task
        prev_task = task
```

Templating

```
template_fields = ('sql', 'gcp_cloudsql_conn_id', 'gcp_conn_id')
template_ext = ('.sql',)
```

More information

See Google Cloud SQL documentation for MySQL and PostgreSQL related proxies.

Google Cloud Transfer Service Operators

- *GcpTransferServiceJobCreateOperator*
- *GcpTransferServiceJobDeleteOperator*
- *GcpTransferServiceJobUpdateOperator*
- *GcpTransferServiceOperationCancelOperator*
- *GcpTransferServiceOperationGetOperator*
- *GcpTransferServiceOperationsListOperator*
- *GcpTransferServiceOperationPauseOperator*
- *GcpTransferServiceOperationResumeOperator*
- *GCPTransferServiceWaitForJobStatusSensor*

GcpTransferServiceJobCreateOperator

Create a transfer job.

The function accepts dates in two formats:

- consistent with Google API

  ```
  { "year": 2019, "month": 2, "day": 11 }
  ```

- as an `datetime` object

The function accepts time in two formats:

- consistent with Google API

```
{ "hours": 12, "minutes": 30, "seconds": 0 }
```

- as an `time` object

If you want to create a job transfer that copies data from AWS S3 then you must have a connection configured. Information about configuration for AWS is available: *Amazon Web Services Connection* The selected connection for AWS can be indicated by the parameter `aws_conn_id`.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
gcs_to_gcs_transfer_body = {
    DESCRIPTION: GCP_DESCRIPTION,
    STATUS: GcpTransferJobsStatus.ENABLED,
    PROJECT_ID: GCP_PROJECT_ID,
    SCHEDULE: {
        SCHEDULE_START_DATE: datetime(2015, 1, 1).date(),
        SCHEDULE_END_DATE: datetime(2030, 1, 1).date(),
        START_TIME_OF_DAY: (datetime.utcnow() + timedelta(minutes=2)).time(),
    },
    TRANSFER_SPEC: {
        GCS_DATA_SOURCE: {BUCKET_NAME: GCP_TRANSFER_FIRST_TARGET_BUCKET},
        GCS_DATA_SINK: {BUCKET_NAME: GCP_TRANSFER_SECOND_TARGET_BUCKET},
        TRANSFER_OPTIONS: {ALREADY_EXISTING_IN_SINK: True},
    },
}  # type: Dict[str, Any]
```

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
aws_to_gcs_transfer_body = {
    DESCRIPTION: GCP_DESCRIPTION,
    STATUS: GcpTransferJobsStatus.ENABLED,
    PROJECT_ID: GCP_PROJECT_ID,
```

(continues on next page)

(continued from previous page)

```
    SCHEDULE: {
        SCHEDULE_START_DATE: datetime(2015, 1, 1).date(),
        SCHEDULE_END_DATE: datetime(2030, 1, 1).date(),
        START_TIME_OF_DAY: (datetime.utcnow() + timedelta(minutes=2)).time(),
    },
    TRANSFER_SPEC: {
        AWS_S3_DATA_SOURCE: {BUCKET_NAME: GCP_TRANSFER_SOURCE_AWS_BUCKET},
        GCS_DATA_SINK: {BUCKET_NAME: GCP_TRANSFER_FIRST_TARGET_BUCKET},
        TRANSFER_OPTIONS: {ALREADY_EXISTING_IN_SINK: True},
    },
}
```

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
create_transfer_job_from_aws = GcpTransferServiceJobCreateOperator(
    task_id="create_transfer_job_from_aws", body=aws_to_gcs_transfer_body
)
```

Templating

```
template_fields = ('body', 'gcp_conn_id', 'aws_conn_id')
```

More information

See Google Cloud Transfer Service - Method: transferJobs.create.

GcpTransferServiceJobDeleteOperator

Deletes a transfer job.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
delete_transfer_from_aws_job = GcpTransferServiceJobDeleteOperator(
    task_id="delete_transfer_from_aws_job",
    job_name="{{task_instance.xcom_pull('create_transfer_job_from_aws')['name']}}",
    project_id=GCP_PROJECT_ID,
)
```

Templating

```
template_fields = ('job_name', 'project_id', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud Transfer Service - REST Resource: transferJobs - Status

GcpTransferServiceJobUpdateOperator

Updates a transfer job.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
update_body = {
    PROJECT_ID: GCP_PROJECT_ID,
    TRANSFER_JOB: {DESCRIPTION: "{}_updated".format(GCP_DESCRIPTION)},
```

(continues on next page)

(continued from previous page)

```
    TRANSFER_JOB_FIELD_MASK: "description",
}
```

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
update_job = GcpTransferServiceJobUpdateOperator(
    task_id="update_job",
    job_name="{{task_instance.xcom_pull('create_transfer_job_from_aws')['name']}}",
    body=update_body,
)
```

Templating

```
template_fields = ('job_name', 'body', 'gcp_conn_id', 'aws_conn_id')
```

More information

See Google Cloud Transfer Service - Method: transferJobs.patch

GcpTransferServiceOperationCancelOperator

Gets a transfer operation. The result is returned to XCOM.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
cancel_operation = GcpTransferServiceOperationCancelOperator(
    task_id="cancel_operation",
    operation_name="{{task_instance.xcom_pull("
    "'wait_for_second_operation_to_start', key='sensed_operations')[0]['name']}}",
)
```

Templating

```
template_fields = ('operation_name', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud Transfer Service - Method: transferOperations.cancel

GcpTransferServiceOperationGetOperator

Gets a transfer operation. The result is returned to XCOM.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
get_operation = GcpTransferServiceOperationGetOperator(
    task_id="get_operation", operation_name="{{task_instance.xcom_pull('list_
↪operations')[0]['name']}}"
)
```

Templating

```
template_fields = ('operation_name', 'gcp_conn_id')
```

More information

See Google Cloud Transfer Service - Method: transferOperations.get

GcpTransferServiceOperationsListOperator

List a transfer operations. The result is returned to XCOM.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
list_operations = GcpTransferServiceOperationsListOperator(
    task_id="list_operations",
    request_filter={
        FILTER_PROJECT_ID: GCP_PROJECT_ID,
        FILTER_JOB_NAMES: ["{{task_instance.xcom_pull('create_transfer_job_from_aws')[
↪'name']}}"],
    },
)
```

Templating

```
template_fields = ('filter', 'gcp_conn_id')
```

More information

See Google Cloud Transfer Service - Method: transferOperations.list

GcpTransferServiceOperationPauseOperator

Pauses a transfer operations.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
pause_operation = GcpTransferServiceOperationPauseOperator(
    task_id="pause_operation",
    operation_name="{{task_instance.xcom_pull('wait_for_operation_to_start', "
    "key='sensed_operations')[0]['name']}}",
)
```

Templating

```
template_fields = ('operation_name', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud Transfer Service - Method: transferOperations.pause

GcpTransferServiceOperationResumeOperator

Resumes a transfer operations.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
resume_operation = GcpTransferServiceOperationResumeOperator(
    task_id="resume_operation", operation_name="{{task_instance.xcom_pull('get_
↪operation')['name']}}"
)
```

Templating

```
template_fields = ('operation_name', 'gcp_conn_id', 'api_version')
```

More information

See Google Cloud Transfer Service - Method: transferOperations.resume

GCPTransferServiceWaitForJobStatusSensor

Waits for at least one operation belonging to the job to have the expected status.

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
GCP_PROJECT_ID = os.environ.get('GCP_PROJECT_ID', 'example-project')
GCP_DESCRIPTION = os.environ.get('GCP_DESCRIPTION', 'description')
GCP_TRANSFER_TARGET_BUCKET = os.environ.get('GCP_TRANSFER_TARGET_BUCKET')
```

(continues on next page)

(continued from previous page)

```
WAIT_FOR_OPERATION_POKE_INTERVAL = int(os.environ.get('WAIT_FOR_OPERATION_POKE_
↪INTERVAL', 5))

GCP_TRANSFER_SOURCE_AWS_BUCKET = os.environ.get('GCP_TRANSFER_SOURCE_AWS_BUCKET')
GCP_TRANSFER_FIRST_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_FIRST_TARGET_BUCKET', 'gcp-transfer-first-target'
)
GCP_TRANSFER_SECOND_TARGET_BUCKET = os.environ.get(
    'GCP_TRANSFER_SECOND_TARGET_BUCKET', 'gcp-transfer-second-target'
)
```

Using the operator

airflow/contrib/example_dags/example_gcp_transfer.pyView Source

```
wait_for_operation_to_end = GCPTransferServiceWaitForJobStatusSensor(
    task_id="wait_for_operation_to_end",
    job_name="{{task_instance.xcom_pull('create_transfer_job_from_aws')['name']}}",
    project_id=GCP_PROJECT_ID,
    expected_statuses={GcpTransferOperationStatus.SUCCESS},
    poke_interval=WAIT_FOR_OPERATION_POKE_INTERVAL,
)
```

Templating

```
template_fields = ('job_name',)
```

Google Cloud Translate Operators

• *CloudTranslateTextOperator*

CloudTranslateTextOperator

Translate a string or list of strings.

For parameter definition, take a look at `CloudTranslateTextOperator`

Using the operator

Basic usage of the operator:

airflow/contrib/example_dags/example_gcp_translate.pyView Source

```
product_set_create = CloudTranslateTextOperator(
    task_id='translate',
    values=['zażółć gęślą jaźń'],
    target_language='en',
```

(continues on next page)

(continued from previous page)

```
    format_='text',
    source_language=None,
    model='base',
)
```

The result of translation is available as dictionary or array of dictionaries accessible via the usual XCom mechanisms of Airflow:

airflow/contrib/example_dags/example_gcp_translate.pyView Source

```
translation_access = BashOperator(
    task_id='access',
    bash_command="echo '{{ task_instance.xcom_pull(\"translate\")[0] }}'"
)
product_set_create >> translation_access
```

Templating

```
template_fields = ('values', 'target_language', 'format_', 'source_language', 'model',
↪ 'gcp_conn_id')
```

More information

See Google Cloud Translate documentation.

Google Cloud Speech Translate Operators

- *GcpTranslateSpeechOperator*

GcpTranslateSpeechOperator

Recognizes speech in audio input and translates it.

For parameter definition, take a look at *airflow.contrib.operators.gcp_translate_speech_operator.GcpTranslateSpeechOperator*

Arguments

Config and audio arguments need to be dicts or objects of corresponding classes from `google.cloud.speech_v1.types` module.

for more information, see: https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/api.html#google.cloud.speech_v1.SpeechClient.recognize

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
CONFIG = {"encoding": "LINEAR16", "language_code": "en_US"}
AUDIO = {"uri": "gs://{bucket}/{object}".format(bucket=BUCKET_NAME, object=FILENAME)}
```

Arguments for translation need to be specified.

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
TARGET_LANGUAGE = 'pl'
FORMAT = 'text'
MODEL = 'base'
SOURCE_LANGUAGE = None
```

Using the operator

airflow/contrib/example_dags/example_gcp_speech.pyView Source

```
translate_speech_task = GcpTranslateSpeechOperator(
    project_id=GCP_PROJECT_ID,
    audio=AUDIO,
    config=CONFIG,
    target_language=TARGET_LANGUAGE,
    format_=FORMAT,
    source_language=SOURCE_LANGUAGE,
    model=MODEL,
    task_id='translate_speech_task'
)
```

Templating

```
template_fields = ('target_language', 'format_', 'source_language', 'model', 'project_
↪id', 'gcp_conn_id')
```

Google Cloud Video Intelligence Operators

- *CloudVideoIntelligenceDetectVideoLabelsOperator*
- *CloudVideoIntelligenceDetectVideoExplicitContentOperator*
- *CloudVideoIntelligenceDetectVideoShotsOperator*

CloudVideoIntelligenceDetectVideoLabelsOperator

Performs video annotation, annotating video labels.

For parameter definition, take a look at *airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVideoLabelsOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
GCP_BUCKET_NAME = os.environ.get(
    "GCP_VIDEO_INTELLIGENCE_BUCKET_NAME", "test-bucket-name"
)
```

Input uri is an uri to a file in Google Cloud Storage

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
INPUT_URI = "gs://{}/video.mp4".format(GCP_BUCKET_NAME)
```

Using the operator

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
detect_video_label = CloudVideoIntelligenceDetectVideoLabelsOperator(
    input_uri=INPUT_URI,
    output_uri=None,
    video_context=None,
    timeout=5,
    task_id="detect_video_label",
)
```

You can use the annotation output via Xcom:

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
detect_video_label_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('detect_video_label')"
                 "['annotationResults'][0]['shotLabelAnnotations'][0]['entity']}}",
    task_id="detect_video_label_result",
)
```

Templating

```
template_fields = ("input_uri", "output_uri", "gcp_conn_id")
```

More information

Note: The duration of video annotation operation is equal or longer than the annotation video itself.

CloudVideoIntelligenceDetectVideoExplicitContentOperator

Performs video annotation, annotating explicit content.

For parameter definition, take a look at *airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVideoExplicitContentOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
GCP_BUCKET_NAME = os.environ.get(
    "GCP_VIDEO_INTELLIGENCE_BUCKET_NAME", "test-bucket-name"
)
```

Input uri is an uri to a file in Google Cloud Storage

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
INPUT_URI = "gs://{}/video.mp4".format(GCP_BUCKET_NAME)
```

Using the operator

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
detect_video_explicit_content =␣
↪CloudVideoIntelligenceDetectVideoExplicitContentOperator(
    input_uri=INPUT_URI,
    output_uri=None,
    video_context=None,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id="detect_video_explicit_content",
)
```

You can use the annotation output via Xcom:

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
detect_video_explicit_content_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('detect_video_explicit_content')"
                 "['annotationResults'][0]['explicitAnnotation']['frames'][0]}}",
    task_id="detect_video_explicit_content_result",
)
```

Templating

```
template_fields = ("input_uri", "output_uri", "gcp_conn_id")
```

More information

Note: The duration of video annotation operation is equal or longer than the annotation video itself.

CloudVideoIntelligenceDetectVideoShotsOperator

Performs video annotation, annotating explicit content.

For parameter definition, take a look at *airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVideoShotsOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
GCP_BUCKET_NAME = os.environ.get(
    "GCP_VIDEO_INTELLIGENCE_BUCKET_NAME", "test-bucket-name"
)
```

Input uri is an uri to a file in Google Cloud Storage

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
INPUT_URI = "gs://{}/video.mp4".format(GCP_BUCKET_NAME)
```

Using the operator

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
detect_video_shots = CloudVideoIntelligenceDetectVideoShotsOperator(
    input_uri=INPUT_URI,
    output_uri=None,
    video_context=None,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id="detect_video_shots",
)
```

You can use the annotation output via Xcom:

airflow/contrib/example_dags/example_gcp_video_intelligence.pyView Source

```
detect_video_shots_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('detect_video_shots')"
                 "['annotationResults'][0]['shotAnnotations'][0]}}",
    task_id="detect_video_shots_result",
)
```

Templating

```
template_fields = ("input_uri", "output_uri", "gcp_conn_id")
```

More information

Note: The duration of video annotation operation is equal or longer than the annotation video itself.

Google Cloud Vision Operators

- *CloudVisionAddProductToProductSetOperator*

- *CloudVisionAnnotateImageOperator*
- *CloudVisionProductCreateOperator*
- *CloudVisionProductDeleteOperator*
- *CloudVisionProductGetOperator*
- *CloudVisionProductSetCreateOperator*
- *CloudVisionProductSetDeleteOperator*
- *CloudVisionProductSetGetOperator*
- *CloudVisionProductSetUpdateOperator*
- *CloudVisionProductUpdateOperator*
- *CloudVisionReferenceImageCreateOperator*
- *CloudVisionRemoveProductFromProductSetOperator*

CloudVisionAddProductToProductSetOperator

Creates a new `ReferenceImage` resource.

For parameter definition, take a look at *CloudVisionAddProductToProductSetOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_ID = os.environ.get('GCP_VISION_PRODUCT_ID', 'product_explicit_id')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_SET_ID = os.environ.get('GCP_VISION_PRODUCT_SET_ID', 'product_set_
↪explicit_id')
```

Using the operator

We are using the `Product`, `ProductSet` and `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import ProductSet
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import Product
```

If `product_set_id` and `product_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
add_product_to_product_set = CloudVisionAddProductToProductSetOperator(
    location=GCP_VISION_LOCATION,
    product_set_id="{{ task_instance.xcom_pull('product_set_create') }}",
    product_id="{{ task_instance.xcom_pull('product_create') }}",
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='add_product_to_product_set',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
add_product_to_product_set_2 = CloudVisionAddProductToProductSetOperator(
    location=GCP_VISION_LOCATION,
    product_set_id=GCP_VISION_PRODUCT_SET_ID,
    product_id=GCP_VISION_PRODUCT_ID,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='add_product_to_product_set_2',
)
```

Templating

```
template_fields = ("location", "product_set_id", "product_id", "project_id", "gcp_
↪conn_id")
```

More information

See Google Cloud Vision Add Product To Product Set documentation.

CloudVisionAnnotateImageOperator

Run image detection and annotation for an image.

For parameter definition, take a look at *`CloudVisionAnnotateImageOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_ANNOTATE_IMAGE_URL = os.environ.get('GCP_VISION_ANNOTATE_IMAGE_URL', 'gs://
↪bucket/image2.jpg')
```

Using the operator

We are using the `enums` and `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision import enums
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
annotate_image = CloudVisionAnnotateImageOperator(
    request=annotate_image_request, retry=Retry(maximum=10.0), timeout=5, task_id=
↪'annotate_image'
)
```

The result can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
annotate_image_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('annotate_image')"
    "['logoAnnotations'][0]['description'] }}",
    task_id='annotate_image_result',
)
```

Templating

```
template_fields = ('request', 'gcp_conn_id')
```

More information

See Google Cloud Vision Annotate Image documentation.

CloudVisionProductCreateOperator

Creates and returns a new product resource.

Possible errors regarding the `Product` object provided:

- Returns INVALID_ARGUMENT if *display_name* is missing or longer than 4096 characters.
- Returns INVALID_ARGUMENT if *description* is longer than 4096 characters.
- Returns INVALID_ARGUMENT if *product_category* is missing or invalid.

For parameter definition, take a look at *CloudVisionProductCreateOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_ID = os.environ.get('GCP_VISION_PRODUCT_ID', 'product_explicit_id')
```

Using the operator

We are using the `Product` and `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import Product
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product = Product(display_name='My Product 1', product_category='toys')
```

The `product_id` argument can be omitted (it will be generated by the API):

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_create = CloudVisionProductCreateOperator(
    location=GCP_VISION_LOCATION,
    product=product,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='product_create',
)
```

Or it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_create_2 = CloudVisionProductCreateOperator(
    product_id=GCP_VISION_PRODUCT_ID,
    location=GCP_VISION_LOCATION,
    product=product,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='product_create_2',
)
```

Templating

```
template_fields = ('location', 'project_id', 'product_id', 'gcp_conn_id')
```

More information

See Google Cloud Vision Product create documentation.

CloudVisionProductDeleteOperator

Permanently deletes a product and its reference images.

Metadata of the product and all its images will be deleted right away, but search queries against `ProductSets` containing the product may still work until all related caches are refreshed.

Possible errors:

- Returns NOT_FOUND if the product does not exist.

For parameter definition, take a look at *CloudVisionProductDeleteOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_ID = os.environ.get('GCP_VISION_PRODUCT_ID', 'product_explicit_id')
```

Using the operator

If `product_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_delete = CloudVisionProductDeleteOperator(
    location=GCP_VISION_LOCATION,
    product_id="{{ task_instance.xcom_pull('product_create') }}",
    task_id='product_delete',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_delete_2 = CloudVisionProductDeleteOperator(
    location=GCP_VISION_LOCATION, product_id=GCP_VISION_PRODUCT_ID, task_id='product_
↪delete_2'
)
```

Templating

```
template_fields = ('location', 'project_id', 'product_id', 'gcp_conn_id')
```

More information

See Google Cloud Vision Product delete documentation.

CloudVisionProductGetOperator

Gets information associated with a `Product`.

Possible errors:

- Returns NOT_FOUND if the *Product* does not exist.

For parameter definition, take a look at *`CloudVisionProductGetOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_ID = os.environ.get('GCP_VISION_PRODUCT_ID', 'product_explicit_id')
```

Using the operator

If `product_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_get = CloudVisionProductGetOperator(
    location=GCP_VISION_LOCATION,
    product_id="{{ task_instance.xcom_pull('product_create') }}",
    task_id='product_get',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_get_2 = CloudVisionProductGetOperator(
    location=GCP_VISION_LOCATION, product_id=GCP_VISION_PRODUCT_ID, task_id='product_
↪get_2'
)
```

Templating

```
template_fields = ('location', 'project_id', 'product_id', 'gcp_conn_id')
```

More information

See Google Cloud Vision Product get documentation.

CloudVisionProductSetCreateOperator

Creates a new `ProductSet` resource.

For parameter definition, take a look at *`CloudVisionProductSetCreateOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_SET_ID = os.environ.get('GCP_VISION_PRODUCT_SET_ID', 'product_set_
↪explicit_id')
```

Using the operator

We are using the `ProductSet` and `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import ProductSet
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set = ProductSet(display_name='My Product Set')
```

The `product_set_id` argument can be omitted (it will be generated by the API):

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_create = CloudVisionProductSetCreateOperator(
    location=GCP_VISION_LOCATION,
    product_set=product_set,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='product_set_create',
)
```

Or it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_create_2 = CloudVisionProductSetCreateOperator(
    product_set_id=GCP_VISION_PRODUCT_SET_ID,
    location=GCP_VISION_LOCATION,
    product_set=product_set,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='product_set_create_2',
)
```

Templating

```
template_fields = ("location", "project_id", "product_set_id", "gcp_conn_id")
```

More information

See Google Cloud Vision ProductSet create documentation.

CloudVisionProductSetDeleteOperator

Permanently deletes a `ProductSet`. `Products` and `ReferenceImages` in the `ProductSet` are not deleted. The actual image files are not deleted from Google Cloud Storage.

For parameter definition, take a look at *`CloudVisionProductSetDeleteOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_SET_ID = os.environ.get('GCP_VISION_PRODUCT_SET_ID', 'product_set_
↪explicit_id')
```

Using the operator

If `product_set_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_delete = CloudVisionProductSetDeleteOperator(
    location=GCP_VISION_LOCATION,
    product_set_id="{{ task_instance.xcom_pull('product_set_create') }}",
    task_id='product_set_delete',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_delete_2 = CloudVisionProductSetDeleteOperator(
    location=GCP_VISION_LOCATION, product_set_id=GCP_VISION_PRODUCT_SET_ID, task_id=
↪'product_set_delete_2'
)
```

Templating

```
template_fields = ('location', 'project_id', 'product_set_id', 'gcp_conn_id')
```

More information

See Google Cloud Vision ProductSet delete documentation.

CloudVisionProductSetGetOperator

Gets information associated with a `ProductSet`.

For parameter definition, take a look at *`CloudVisionProductSetGetOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_SET_ID = os.environ.get('GCP_VISION_PRODUCT_SET_ID', 'product_set_
↪explicit_id')
```

Using the operator

If `product_set_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_get = CloudVisionProductSetGetOperator(
    location=GCP_VISION_LOCATION,
    product_set_id="{{ task_instance.xcom_pull('product_set_create') }}",
    task_id='product_set_get',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_get_2 = CloudVisionProductSetGetOperator(
    location=GCP_VISION_LOCATION, product_set_id=GCP_VISION_PRODUCT_SET_ID, task_id=
↪'product_set_get_2'
)
```

Templating

```
template_fields = ('location', 'project_id', 'product_set_id', 'gcp_conn_id')
```

More information

See Google Cloud Vision ProductSet get documentation.

CloudVisionProductSetUpdateOperator

Makes changes to a `ProductSet` resource. Only `display_name` can be updated currently.

Note: To locate the *ProductSet* resource, its *name* in the form `projects/PROJECT_ID/locations/LOC_ID/productSets/PRODUCT_SET_ID` is necessary.

You can provide the *name* directly as an attribute of the *product_set* object. However, you can leave it blank and provide *location* and *product_set_id* instead (and optionally *project_id* - if not present, the connection default will be used) and the *name* will be created by the operator itself.

This mechanism exists for your convenience, to allow leaving the *project_id* empty and having Airflow use the connection default *project_id*.

For parameter definition, take a look at *`CloudVisionProductSetUpdateOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_SET_ID = os.environ.get('GCP_VISION_PRODUCT_SET_ID', 'product_set_
↪explicit_id')
```

Using the operator

We are using the `ProductSet` object from the Google Cloud Vision library:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import ProductSet
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set = ProductSet(display_name='My Product Set')
```

Initialization of the task:

If `product_set_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_update = CloudVisionProductSetUpdateOperator(
    location=GCP_VISION_LOCATION,
    product_set_id="{{ task_instance.xcom_pull('product_set_create') }}",
    product_set=ProductSet(display_name='My Product Set 2'),
    task_id='product_set_update',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_set_update_2 = CloudVisionProductSetUpdateOperator(
    location=GCP_VISION_LOCATION,
    product_set_id=GCP_VISION_PRODUCT_SET_ID,
    product_set=ProductSet(display_name='My Product Set 2'),
    task_id='product_set_update_2',
)
```

Templating

```
template_fields = ('location', 'project_id', 'product_set_id', 'gcp_conn_id')
```

More information

See Google Cloud Vision ProductSet update documentation.

CloudVisionProductUpdateOperator

Makes changes to a `Product` resource. Only the `display_name`, `description`, and `labels` fields can be updated right now. If labels are updated, the change will not be reflected in queries until the next index time.

Note: To locate the *Product* resource, its *name* in the form `projects/PROJECT_ID/locations/LOC_ID/products/PRODUCT_ID` is necessary.

You can provide the *name* directly as an attribute of the *product* object. However, you can leave it blank and provide *location* and *product_id* instead (and optionally *project_id* - if not present, the connection default will be used) and the *name* will be created by the operator itself.

This mechanism exists for your convenience, to allow leaving the *project_id* empty and having Airflow use the connection default *project_id*.

Possible errors:

- Returns NOT_FOUND if the *Product* does not exist.

- Returns INVALID_ARGUMENT if *display_name* is present in *update_mask* but is missing from the request or longer than 4096 characters.
- Returns INVALID_ARGUMENT if *description* is present in *update_mask* but is longer than 4096 characters.
- Returns INVALID_ARGUMENT if *product_category* is present in *update_mask*.

For parameter definition, take a look at `CloudVisionProductUpdateOperator`

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_ID = os.environ.get('GCP_VISION_PRODUCT_ID', 'product_explicit_id')
```

Using the operator

We are using the `Product` object from the Google Cloud Vision library:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import Product
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product = Product(display_name='My Product 1', product_category='toys')
```

If `product_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_update = CloudVisionProductUpdateOperator(
    location=GCP_VISION_LOCATION,
    product_id="{{ task_instance.xcom_pull('product_create') }}",
    product=Product(display_name='My Product 2', description='My updated description
↪'),
    task_id='product_update',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
product_update_2 = CloudVisionProductUpdateOperator(
    location=GCP_VISION_LOCATION,
    product_id=GCP_VISION_PRODUCT_ID,
    product=Product(display_name='My Product 2', description='My updated description
↪'),
    task_id='product_update_2',
)
```

Templating

```
template_fields = ('location', 'project_id', 'product_id', 'gcp_conn_id')
```

More information

See Google Cloud Vision Product update documentation.

CloudVisionReferenceImageCreateOperator

Creates a new `ReferenceImage` resource.

For parameter definition, take a look at *`CloudVisionReferenceImageCreateOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_REFERENCE_IMAGE_ID = os.environ.get('GCP_VISION_REFERENCE_IMAGE_ID',
↪'reference_image_explicit_id')
GCP_VISION_REFERENCE_IMAGE_URL = os.environ.get('GCP_VISION_REFERENCE_IMAGE_URL',
↪'gs://bucket/image1.jpg')
```

Using the operator

We are using the `ReferenceImage` and `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import ReferenceImage
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
reference_image = ReferenceImage(uri=GCP_VISION_REFERENCE_IMAGE_URL)
```

The `product_set_id` argument can be omitted (it will be generated by the API):

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
reference_image_create = CloudVisionReferenceImageCreateOperator(
    location=GCP_VISION_LOCATION,
    reference_image=reference_image,
    product_id="{{ task_instance.xcom_pull('product_create') }}",
```

(continues on next page)

(continued from previous page)

```
    reference_image_id=GCP_VISION_REFERENCE_IMAGE_ID,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='reference_image_create',
)
```

Or it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
reference_image_create_2 = CloudVisionReferenceImageCreateOperator(
    location=GCP_VISION_LOCATION,
    reference_image=reference_image,
    product_id=GCP_VISION_PRODUCT_ID,
    reference_image_id=GCP_VISION_REFERENCE_IMAGE_ID,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='reference_image_create_2',
)
```

Templating

```
template_fields = (
    "location",
    "reference_image",
    "product_id",
    "reference_image_id",
    "project_id",
    "gcp_conn_id",
)
```

More information

See Google Cloud Vision ReferenceImage create documentation.

CloudVisionRemoveProductFromProductSetOperator

Creates a new `ReferenceImage` resource.

For parameter definition, take a look at *`CloudVisionRemoveProductFromProductSetOperator`*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_LOCATION = os.environ.get('GCP_VISION_LOCATION', 'europe-west1')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_ID = os.environ.get('GCP_VISION_PRODUCT_ID', 'product_explicit_id')
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_PRODUCT_SET_ID = os.environ.get('GCP_VISION_PRODUCT_SET_ID', 'product_set_
↪explicit_id')
```

Using the operator

We are using the `Product`, `ProductSet` and `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import ProductSet
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.cloud.vision_v1.types import Product
```

If `product_set_id` and `product_id` was generated by the API it can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
remove_product_from_product_set = CloudVisionRemoveProductFromProductSetOperator(
    location=GCP_VISION_LOCATION,
    product_set_id="{{ task_instance.xcom_pull('product_set_create') }}",
    product_id="{{ task_instance.xcom_pull('product_create') }}",
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='remove_product_from_product_set',
)
```

Otherwise it can be specified explicitly:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
remove_product_from_product_set_2 = CloudVisionRemoveProductFromProductSetOperator(
    location=GCP_VISION_LOCATION,
    product_set_id=GCP_VISION_PRODUCT_SET_ID,
    product_id=GCP_VISION_PRODUCT_ID,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id='remove_product_from_product_set_2',
)
```

Templating

```
template_fields = ("location", "product_set_id", "product_id", "project_id", "gcp_
↪conn_id")
```

More information

See Google Cloud Vision Remove Product From Product Set documentation.

CloudVisionDetectTextOperator

Run text detection for an image.

For parameter definition, take a look at `CloudVisionDetectTextOperator`

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_ANNOTATE_IMAGE_URL = os.environ.get('GCP_VISION_ANNOTATE_IMAGE_URL', 'gs://
↪bucket/image2.jpg')
```

Using the operator

We are using the `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
detect_text = CloudVisionDetectTextOperator(
    image=DETECT_IMAGE,
    retry=Retry(maximum=10.0),
    timeout=5,
    task_id="detect_text",
    language_hints="en",
    web_detection_params={'include_geo_results': True},
)
```

The result can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
detect_text_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('detect_text')['textAnnotations
↪'][0] }}",
    task_id="detect_text_result",
)
```

Templating

```
template_fields = ("image", "max_results", "timeout", "gcp_conn_id")
```

More information

See Google Cloud Vision Text Detection documentation.

CloudVisionDetectDocumentTextOperator

Run document text detection for an image.

For parameter definition, take a look at *CloudVisionDetectDocumentTextOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_ANNOTATE_IMAGE_URL = os.environ.get('GCP_VISION_ANNOTATE_IMAGE_URL', 'gs://
↪bucket/image2.jpg')
```

Using the operator

We are using the `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
document_detect_text = CloudVisionDetectDocumentTextOperator(
    image=DETECT_IMAGE, retry=Retry(maximum=10.0), timeout=5, task_id="document_
↪detect_text"
)
```

The result can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
document_detect_text_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('document_detect_text')[
↪'textAnnotations'][0] }}",
    task_id="document_detect_text_result",
)
```

Templating

```
template_fields = ("image", "max_results", "timeout", "gcp_conn_id")
```

More information

See Google Cloud Vision Document Text Detection documentation.

CloudVisionDetectImageLabelsOperator

Run image label detection for an image.

For parameter definition, take a look at *CloudVisionDetectImageLabelsOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_ANNOTATE_IMAGE_URL = os.environ.get('GCP_VISION_ANNOTATE_IMAGE_URL', 'gs://
↪bucket/image2.jpg')
```

Using the operator

We are using the `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
detect_labels = CloudVisionDetectImageLabelsOperator(
    image=DETECT_IMAGE, retry=Retry(maximum=10.0), timeout=5, task_id="detect_labels"
)
```

The result can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
detect_labels_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('detect_labels')['labelAnnotations
↪'][0] }}",
    task_id="detect_labels_result",
)
```

Templating

```
template_fields = ("image", "max_results", "timeout", "gcp_conn_id")
```

More information

See Google Cloud Vision Label Detection documentation.

CloudVisionDetectImageSafeSearchOperator

Run image label detection for an image.

For parameter definition, take a look at *CloudVisionDetectImageSafeSearchOperator*

Arguments

Some arguments in the example DAG are taken from the OS environment variables:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
GCP_VISION_ANNOTATE_IMAGE_URL = os.environ.get('GCP_VISION_ANNOTATE_IMAGE_URL', 'gs://
↪bucket/image2.jpg')
```

Using the operator

We are using the `Retry` objects from Google libraries:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
from google.api_core.retry import Retry
```

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
detect_safe_search = CloudVisionDetectImageSafeSearchOperator(
    image=DETECT_IMAGE, retry=Retry(maximum=10.0), timeout=5, task_id="detect_safe_
↪search"
)
```

The result can be extracted from XCOM:

airflow/contrib/example_dags/example_gcp_vision.pyView Source

```
detect_safe_search_result = BashOperator(
    bash_command="echo {{ task_instance.xcom_pull('detect_safe_search') }}",
    task_id="detect_safe_search_result",
)
```

Templating

```
template_fields = ("image", "max_results", "timeout", "gcp_conn_id")
```

More information

See Google Cloud Vision Safe Search Detection documentation.

3.6.4.4 Papermill

Apache Airflow supports integration with Papermill. Papermill is a tool for parameterizing and executing Jupyter Notebooks. Perhaps you have a financial report that you wish to run with different values on the first or last day of a month or at the beginning or end of the year. Using *parameters* in your notebook and using the *PapermillOperator* makes this a breeze.

Usage

Creating a notebook

To parameterize your notebook designate a cell with the tag parameters. Papermill looks for the parameters cell and treats this cell as defaults for the parameters passed in at execution time. Papermill will add a new cell tagged with injected-parameters with input parameters in order to overwrite the values in parameters. If no cell is tagged with parameters the injected cell will be inserted at the top of the notebook.

Note that Jupyter notebook has out of the box support for tags but you need to install the celltags extension for Jupyter Lab: *jupyter labextension install @jupyterlab/celltags*

Make sure that you save your notebook somewhere so that Airflow can access it. Papermill supports S3, GCS, Azure and Local. HDFS is *not* supported.

Example DAG

```python
import airflow

from airflow.models import DAG
from airflow.operators.papermill_operator import PapermillOperator

from datetime import timedelta

args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2)
}

dag = DAG(
    dag_id='example_papermill_operator', default_args=args,
    schedule_interval='0 0 * * *',
    dagrun_timeout=timedelta(minutes=60))

run_this = PapermillOperator(
    task_id="run_example_notebook",
    dag=dag,
    input_nb="/tmp/hello_world.ipynb",
    output_nb="/tmp/out-{{ execution_date }}.ipynb",
    parameters={"msgs": "Ran from Airflow at {{ execution_date }}!"}
)
```

This DAG will use Papermill to run the notebook “hello_world”, based on the execution date it will create an output notebook “out-<date>”. All fields, including the keys in the parameters, are templated.

3.6.4.5 PythonOperator

Use the `PythonOperator` to execute Python callables.

airflow/example_dags/example_python_operator.pyView Source

```python
def print_context(ds, **kwargs):
    """Print the Airflow context and ds variable from the context."""
    pprint(kwargs)
    print(ds)
```

(continues on next page)

(continued from previous page)

```
    return 'Whatever you return gets printed in the logs'


run_this = PythonOperator(
    task_id='print_the_context',
    provide_context=True,
    python_callable=print_context,
    dag=dag,
)
```

Passing in arguments

Use the `op_args` and `op_kwargs` arguments to pass additional arguments to the Python callable.

airflow/example_dags/example_python_operator.pyView Source

```
def my_sleeping_function(random_base):
    """This is a function that will run within the DAG execution"""
    time.sleep(random_base)


# Generate 5 sleeping tasks, sleeping from 0.0 to 0.4 seconds respectively
for i in range(5):
    task = PythonOperator(
        task_id='sleep_for_' + str(i),
        python_callable=my_sleeping_function,
        op_kwargs={'random_base': float(i) / 10},
        dag=dag,
    )

    run_this >> task
```

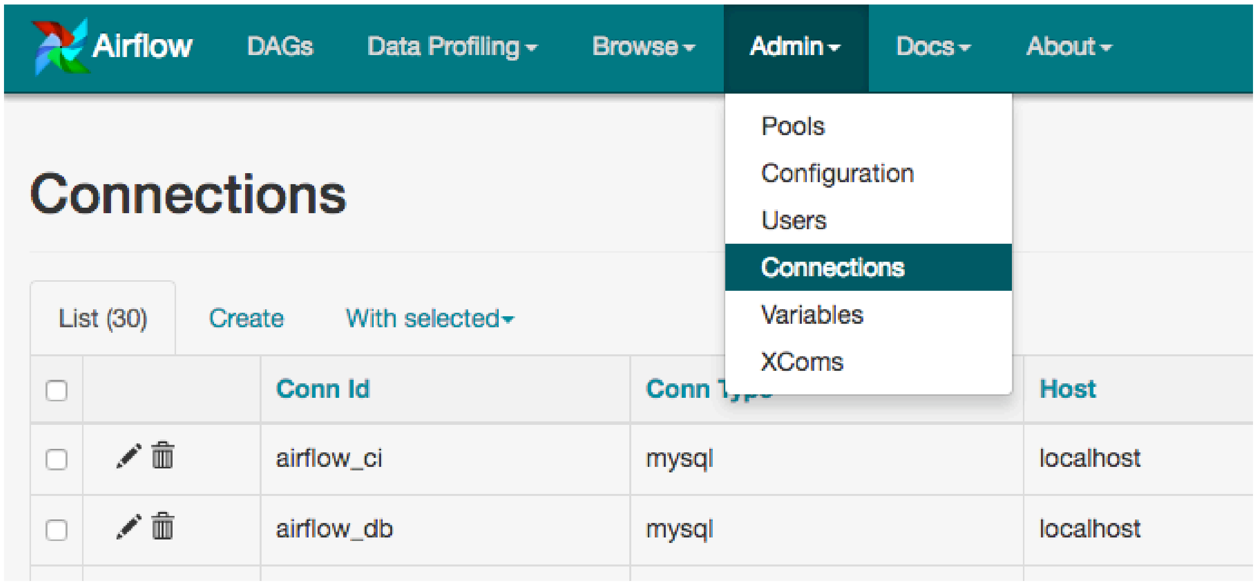
Templating

When you set the `provide_context` argument to `True`, Airflow passes in an additional set of keyword arguments: one for each of the *Jinja template variables* and a `templates_dict` argument.

The `templates_dict` argument is templated, so each value in the dictionary is evaluated as a *Jinja template*.

3.6.5 Managing Connections

Airflow needs to know how to connect to your environment. Information such as hostname, port, login and passwords to other systems and services is handled in the `Admin->Connections` section of the UI. The pipeline code you will author will reference the 'conn_id' of the Connection objects.



Connections

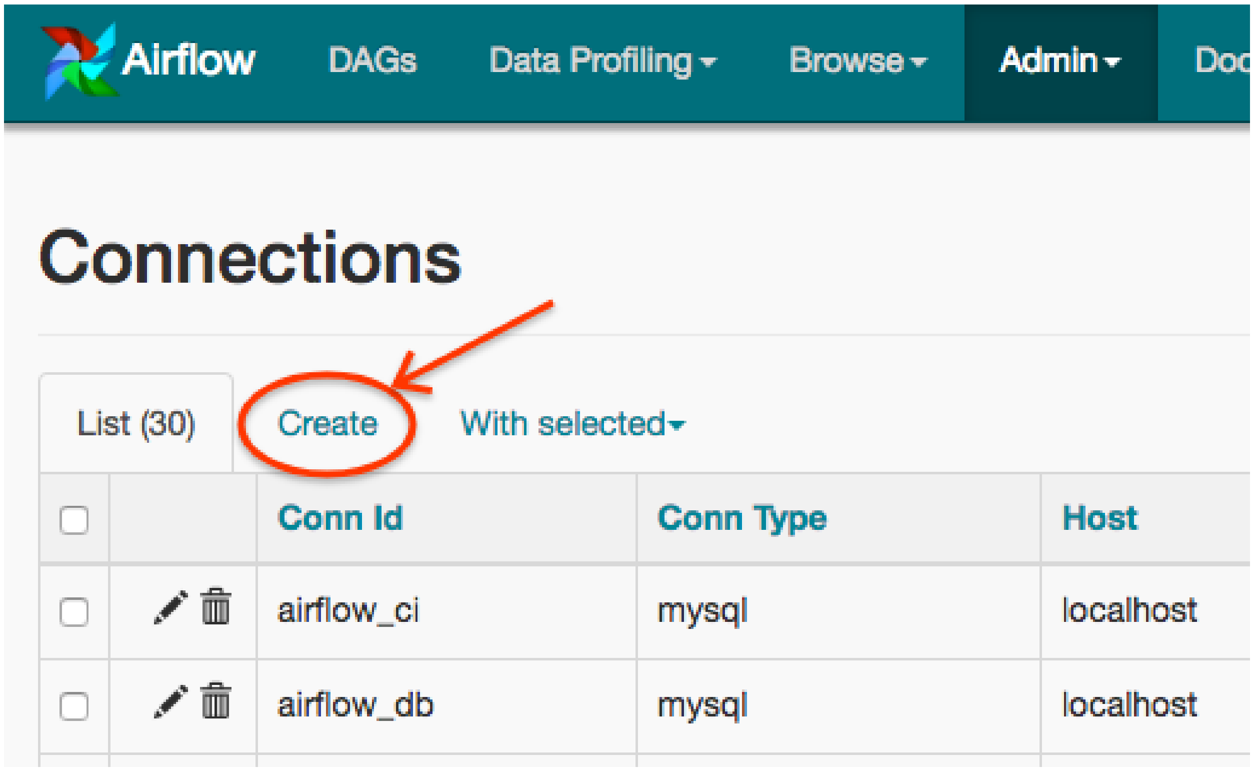
Conn Id	Conn Type	Host
airflow_ci	mysql	localhost
airflow_db	mysql	localhost

Connections can be created and managed using either the UI or environment variables.

See the *Connections Concepts* documentation for more information.

3.6.5.1 Creating a Connection with the UI

Open the `Admin->Connections` section of the UI. Click the `Create` link to create a new connection.



Connections

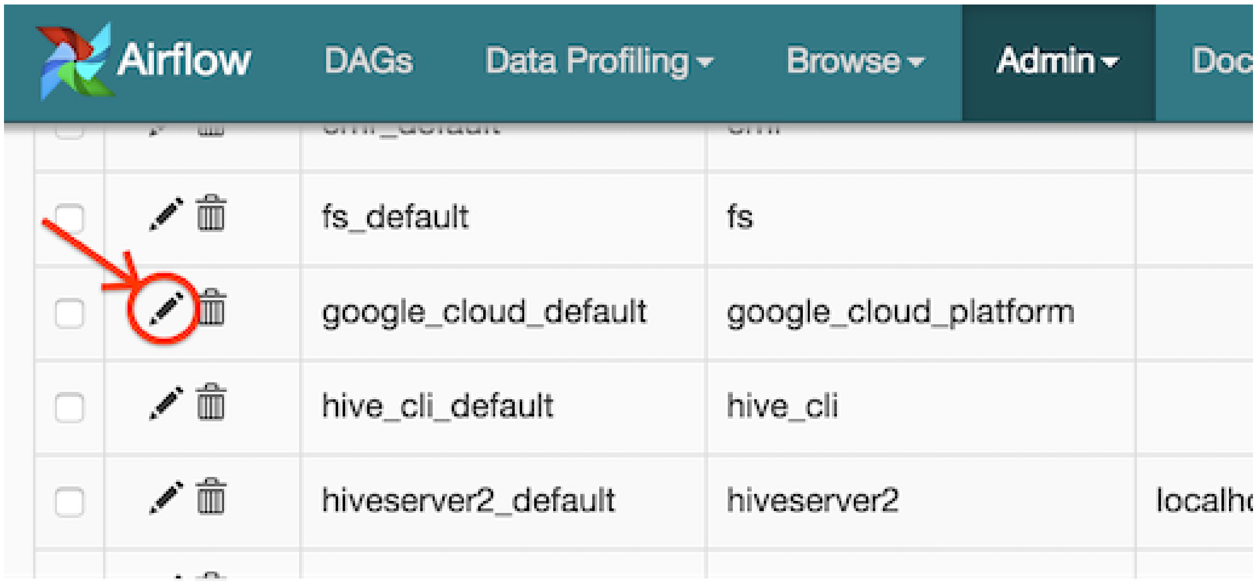
Conn Id	Conn Type	Host
airflow_ci	mysql	localhost
airflow_db	mysql	localhost

1. Fill in the `Conn Id` field with the desired connection ID. It is recommended that you use lower-case characters and separate words with underscores.
2. Choose the connection type with the `Conn Type` field.

3. Fill in the remaining fields. See *Connection Types* for a description of the fields belonging to the different connection types.
4. Click the `Save` button to create the connection.

3.6.5.2 Editing a Connection with the UI

Open the `Admin->Connections` section of the UI. Click the pencil icon next to the connection you wish to edit in the connection list.



fs_default	fs	
google_cloud_default	google_cloud_platform	
hive_cli_default	hive_cli	
hiveserver2_default	hiveserver2	localh

Modify the connection properties and click the `Save` button to save your changes.

3.6.5.3 Creating a Connection with Environment Variables

Connections in Airflow pipelines can be created using environment variables. The environment variable needs to have a prefix of `AIRFLOW_CONN_` for Airflow with the value in a URI format to use the connection properly.

When referencing the connection in the Airflow pipeline, the `conn_id` should be the name of the variable without the prefix. For example, if the `conn_id` is named `postgres_master` the environment variable should be named `AIRFLOW_CONN_POSTGRES_MASTER` (note that the environment variable must be all uppercase).

Airflow assumes the value returned from the environment variable to be in a URI format (e.g. `postgres://user:password@localhost:5432/master` or `s3://accesskey:secretkey@S3`). The underscore character is not allowed in the scheme part of URI, so it must be changed to a hyphen character (e.g. *google-compute-platform* if *conn_type* is *google_compute_platform*). Query parameters are parsed to one-dimensional dict and then used to fill extra.

3.6.5.4 Connection Types

Amazon Web Services Connection

The Amazon Web Services connection type enables the *AWS Integrations*.

Authenticating to AWS

Authentication may be performed using any of the boto3 options. Alternatively, one can pass credentials in as a Connection initialisation parameter.

To use IAM instance profile, create an "empty" connection (i.e. one with no Login or Password specified).

Default Connection IDs

The default connection ID is `aws_default`.

Configuring the Connection

Login (optional) Specify the AWS access key ID.

Password (optional) Specify the AWS secret access key.

Extra (optional) Specify the extra parameters (as json dictionary) that can be used in AWS connection. The following parameters are supported:

- **aws_account_id**: AWS account ID for the connection
- **aws_iam_role**: AWS IAM role for the connection
- **external_id**: AWS external ID for the connection
- **host**: Endpoint URL for the connection
- **region_name**: AWS region for the connection
- **role_arn**: AWS role ARN for the connection

Example "extras" field:

```
{
   "aws_iam_role": "aws_iam_role_name",
   "region_name": "ap-southeast-2"
}
```

Google Cloud Platform Connection

The Google Cloud Platform connection type enables the *GCP Integrations*.

Authenticating to GCP

There are two ways to connect to GCP using Airflow.

1. Use Application Default Credentials, such as via the metadata server when running on Google Compute Engine.
2. Use a service account key file (JSON format) on disk.

Default Connection IDs

All hooks and operators related to Google Cloud Platform use `google_cloud_default` by default.

Configuring the Connection

Project Id (optional) The Google Cloud project ID to connect to. It is used as default project id by operators using it and can usually be overridden at the operator level.

Keyfile Path Path to a service account key file (JSON format) on disk.

Not required if using application default credentials.

Keyfile JSON Contents of a service account key file (JSON format) on disk. It is recommended to *Secure your connections* if using this method to authenticate.

Not required if using application default credentials.

Scopes (comma separated) A list of comma-separated Google Cloud scopes to authenticate with.

Number of Retries Integer, number of times to retry with randomized exponential backoff. If all retries fail, the `googleapiclient.errors.HttpError` represents the last request. If zero (default), we attempt the request only once.

When specifying the connection in environment variable you should specify it using URI syntax, with the following requirements:

- scheme part should be equals `google-cloud-platform` (Note: look for a hyphen character)
- authority (username, password, host, port), path is ignored
- query parameters contains information specific to this type of connection. The following keys are accepted:
  - `extra__google_cloud_platform__project` - Project Id
  - `extra__google_cloud_platform__key_path` - Keyfile Path
  - `extra__google_cloud_platform__key_dict` - Keyfile JSON
  - `extra__google_cloud_platform__scope` - Scopes
  - `extra__google_cloud_platform__num_retries` - Number of Retries

Note that all components of the URI should be URL-encoded.

For example:

```
google-cloud-platform://?extra__google_cloud_platform__key_path=%2Fkeys%2Fkey.
↪json&extra__google_cloud_platform__scope=https%3A%2F%2Fwww.googleapis.com%2Fauth
↪%2Fcloud-platform&extra__google_cloud_platform__project=airflow&extra__google_
↪cloud_platform__num_retries=5
```

Google Cloud SQL Connection

The gcpcloudsql:// connection is used by *`airflow.contrib.operators.gcp_sql_operator.CloudSqlQueryOperator`* to perform query on a Google Cloud SQL database. Google Cloud SQL database can be either Postgres or MySQL, so this is a "meta" connection type. It introduces common schema for both MySQL and Postgres, including what kind of connectivity should be used. Google Cloud SQL supports connecting via public IP or via Cloud SQL Proxy. In the latter case the *`CloudSqlDatabaseHook`* uses *`CloudSqlProxyRunner`* to automatically prepare and use temporary Postgres or MySQL connection that will use the proxy to connect (either via TCP or UNIX socket.

Configuring the Connection

Host (required) The host to connect to.

Schema (optional) Specify the schema name to be used in the database.

Login (required) Specify the user name to connect.

Password (required) Specify the password to connect.

Extra (optional) Specify the extra parameters (as JSON dictionary) that can be used in Google Cloud SQL connection.

Details of all the parameters supported in extra field can be found in *CloudSqlDatabaseHook*

Example "extras" field:

```
{
   "database_type": "mysql",
   "project_id": "example-project",
   "location": "europe-west1",
   "instance": "testinstance",
   "use_proxy": true,
   "sql_proxy_use_tcp": false
}
```

When specifying the connection as URI (in AIRFLOW_CONN_* variable), you should specify it following the standard syntax of DB connection, where extras are passed as parameters of the URI. Note that all components of the URI should be URL-encoded.

For example:

```
gcpcloudsql://user:XXXXXXXXX@1.1.1.1:3306/mydb?database_type=mysql&project_
↪id=example-project&location=europe-west1&instance=testinstance&use_proxy=True&
↪sql_proxy_use_tcp=False
```

gRPC

The gRPC connection type enables integrated connection to a gRPC service

Authenticating to gRPC

There are several ways to connect to gRPC service using Airflow.

1. Using *NO_AUTH* mode, simply setup an insecure channel of connection.
2. Using *SSL* or *TLS* mode, supply a credential pem file for the connection id, this will setup SSL or TLS secured connection with gRPC service.
3. Using *JWT_GOOGLE* mode. It is using google auth default credentials by default, further use case of getting credentials from service account can be add later on.
4. Using *OATH_GOOGLE* mode. Scopes are required in the extra field, can be setup in the UI. It is using google auth default credentials by default, further use case of getting credentials from service account can be add later on.
5. Using *CUSTOM* mode. For this type of connection, you can pass in a connection function takes in the connection object and return a gRPC channel and supply whatever authentication type you want.

Default Connection IDs

The following connection IDs are used by default.

grpc_default Used by the *GrpcHook* hook.

Configuring the Connection

Host The host url of the gRPC server

Port (Optional) The port to connect to on gRPC server

Auth Type Authentication type of the gRPC connection. *NO_AUTH* by default, possible values are *NO_AUTH*, *SSL*, *TLS*, *JWT_GOOGLE*, *OATH_GOOGLE*, *CUSTOM*

Credential Pem File (Optional) Pem file that contains credentials for *SSL* and *TLS* type auth Not required for other types.

Scopes (comma separated) (Optional) A list of comma-separated Google Cloud scopes to authenticate with. Only for *OATH_GOOGLE* type connection

MySQL Connection

The MySQL connection type provides connection to a MySQL database.

Configuring the Connection

Host (required) The host to connect to.

Schema (optional) Specify the schema name to be used in the database.

Login (required) Specify the user name to connect.

Password (required) Specify the password to connect.

Extra (optional) Specify the extra parameters (as json dictionary) that can be used in MySQL connection. The following parameters are supported:

- **charset**: specify charset of the connection
- **cursor**: one of "sscursor", "dictcursor, "ssdictcursor" . Specifies cursor class to be used
- **local_infile**: controls MySQL's LOCAL capability (permitting local data loading by clients). See MySQLdb docs for details.
- **unix_socket**: UNIX socket used instead of the default socket.
- **ssl**: Dictionary of SSL parameters that control connecting using SSL. Those parameters are server specific and should contain "ca", "cert", "key", "capath", "cipher" parameters. See MySQLdb docs for details. Note that to be useful in URL notation, this parameter might also be a string where the SSL dictionary is a string-encoded JSON dictionary.

Example "extras" field:

```
{
   "charset": "utf8",
   "cursor": "sscursor",
   "local_infile": true,
   "unix_socket": "/var/socket",
   "ssl": {
     "cert": "/tmp/client-cert.pem",
     "ca": "/tmp/server-ca.pem'",
     "key": "/tmp/client-key.pem"
   }
}
```

or

```
{
   "charset": "utf8",
   "cursor": "sscursor",
   "local_infile": true,
   "unix_socket": "/var/socket",
   "ssl": "{\"cert\": \"/tmp/client-cert.pem\", \"ca\": \"/tmp/server-ca.pem\", \
↪"key\": \"/tmp/client-key.pem\"}"
}
```

When specifying the connection as URI (in AIRFLOW_CONN_* variable) you should specify it following the standard syntax of DB connections - where extras are passed as parameters of the URI. Note that all components of the URI should be URL-encoded.

For example:

```
mysql://mysql_user:XXXXXXXXXXXX@1.1.1.1:3306/mysqldb?ssl=%7B%22cert%22%3A+%22
↪%2Ftmp%2Fclient-cert.pem%22%2C+%22ca%22%3A+%22%2Ftmp%2Fserver-ca.pem%22%2C+
↪%22key%22%3A+%22%2Ftmp%2Fclient-key.pem%22%7D
```

Note: If encounter UnicodeDecodeError while working with MySQL connection, check the charset defined is matched to the database charset.

Oracle Connection

The Oracle connection type provides connection to a Oracle database.

Configuring the Connection

Dsn (required) The Data Source Name. The host address for the Oracle server.

Sid (optional) The Oracle System ID. The uniquely identify a particular database on a system.

Service_name (optional) The db_unique_name of the database.

Port (optional) The port for the Oracle server, Default 1521.

Login (required) Specify the user name to connect.

Password (required) Specify the password to connect.

Extra (optional) Specify the extra parameters (as json dictionary) that can be used in Oracle connection. The following parameters are supported:

- **encoding** - The encoding to use for regular database strings. If not specified, the environment variable *NLS_LANG* is used. If the environment variable *NLS_LANG* is not set, *ASCII* is used.
- **nencoding** - The encoding to use for national character set database strings. If not specified, the environment variable *NLS_NCHAR* is used. If the environment variable *NLS_NCHAR* is not used, the environment variable *NLS_LANG* is used instead, and if the environment variable *NLS_LANG* is not set, *ASCII* is used.
- **threaded** - Whether or not Oracle should wrap accesses to connections with a mutex. Default value is False.
- **events** - Whether or not to initialize Oracle in events mode.
- **mode** - one of *sysdba*, *sysasm*, *sysoper*, *sysbkp*, *sysdgd*, *syskmt* or *sysrac* which are defined at the module level, Default mode is connecting.
- **purity** - one of *new*, *self*, *default*. Specify the session acquired from the pool. configuration parameter.

More details on all Oracle connect parameters supported can be found in cx_Oracle documentation.

Example "extras" field:

```
{
   "encoding": "UTF-8",
   "nencoding": "UTF-8",
   "threaded": false,
   "events": false,
   "mode": "sysdba",
   "purity": "new"
}
```

When specifying the connection as URI (in AIRFLOW_CONN_* variable) you should specify it following the standard syntax of DB connections, where extras are passed as parameters of the URI (note that all components of the URI should be URL-encoded).

For example:

```
oracle://oracle_user:XXXXXXXXXXXX@1.1.1.1:1521?encoding=UTF-8&nencoding=UTF-8&
↪threaded=False&events=False&mode=sysdba&purity=new
```

PostgresSQL Connection

The Postgres connection type provides connection to a Postgres database.

Configuring the Connection

Host (required) The host to connect to.

Schema (optional) Specify the schema name to be used in the database.

Login (required) Specify the user name to connect.

Password (required) Specify the password to connect.

Extra (optional) Specify the extra parameters (as json dictionary) that can be used in postgres connection. The following parameters out of the standard python parameters are supported:

- **sslmode** - This option determines whether or with what priority a secure SSL TCP/IP connection will be negotiated with the server. There are six modes: 'disable', 'allow', 'prefer', 'require', 'verify-ca', 'verify-full'.
- **sslcert** - This parameter specifies the file name of the client SSL certificate, replacing the default.
- **sslkey** - This parameter specifies the file name of the client SSL key, replacing the default.
- **sslrootcert** - This parameter specifies the name of a file containing SSL certificate authority (CA) certificate(s).
- **sslcrl** - This parameter specifies the file name of the SSL certificate revocation list (CRL).
- **application_name** - Specifies a value for the application_name configuration parameter.
- **keepalives_idle** - Controls the number of seconds of inactivity after which TCP should send a keepalive message to the server.

More details on all Postgres parameters supported can be found in Postgres documentation.

Example "extras" field:

```
{
   "sslmode": "verify-ca",
   "sslcert": "/tmp/client-cert.pem",
   "sslca": "/tmp/server-ca.pem",
   "sslkey": "/tmp/client-key.pem"
}
```

When specifying the connection as URI (in AIRFLOW_CONN_* variable) you should specify it following the standard syntax of DB connections, where extras are passed as parameters of the URI (note that all components of the URI should be URL-encoded).

For example:

```
postgresql://postgres_user:XXXXXXXXXXXX@1.1.1.1:5432/postgresdb?sslmode=verify-ca&
↪sslcert=%2Ftmp%2Fclient-cert.pem&sslkey=%2Ftmp%2Fclient-key.pem&sslrootcert=
↪%2Ftmp%2Fserver-ca.pem
```

SSH Connection

The SSH connection type provides connection to use *SSHHook* to run commands on a remote server using *SSHOperator* or transfer file from/to the remote server using `SFTPOperator`.

Configuring the Connection

Host (required) The Remote host to connect.

Username (optional) The Username to connect to the remote_host.

Password (optional) Specify the password of the username to connect to the remote_host.

Port (optional) Port of remote host to connect. Default is 22.

Extra (optional) Specify the extra parameters (as json dictionary) that can be used in ssh connection. The following parameters out of the standard python parameters are supported:

- **timeout** - An optional timeout (in seconds) for the TCP connect. Default is `10`.
- **compress** - `true` to ask the remote client/server to compress traffic; *false* to refuse compression. Default is `true`.
- **no_host_key_check** - Set to `false` to restrict connecting to hosts with no entries in `~/.ssh/known_hosts` (Hosts file). This provides maximum protection against trojan horse attacks, but can be troublesome when the `/etc/ssh/ssh_known_hosts` file is poorly maintained or connections to new hosts are frequently made. This option forces the user to manually add all new hosts. Default is `true`, ssh will automatically add new host keys to the user known hosts files.
- **allow_host_key_change** - Set to `true` if you want to allow connecting to hosts that has host key changed or when you get 'REMOTE HOST IDENTIFICATION HAS CHANGED' error. This wont protect against Man-In-The-Middle attacks. Other possible solution is to remove the host entry from `~/.ssh/known_hosts` file. Default is `false`.

Example "extras" field:

```
{
   "timeout": "10",
   "compress": "false",
   "no_host_key_check": "false",
```

(continues on next page)

(continued from previous page)

```
    "allow_host_key_change": "false"
}
```

When specifying the connection as URI (in AIRFLOW_CONN_* variable) you should specify it following the standard syntax of connections, where extras are passed as parameters of the URI (note that all components of the URI should be URL-encoded).

For example:

```
ssh://user:pass@localhost:22?timeout=10&compress=false&no_host_key_check=false&
↪allow_host_key_change=true
```

3.6.6 Securing Connections

By default, Airflow will save the passwords for the connection in plain text within the metadata database. The `crypto` package is highly recommended during installation. The `crypto` package does require that your operating system has `libffi-dev` installed.

If `crypto` package was not installed initially, it means that your Fernet key in `airflow.cfg` is empty.

You can still enable encryption for passwords within connections by following below steps:

1. Install crypto package `pip install 'apache-airflow[crypto]'`
2. Generate fernet_key, using this code snippet below. `fernet_key` must be a base64-encoded 32-byte key:

   ```
   from cryptography.fernet import Fernet
   fernet_key= Fernet.generate_key()
   print(fernet_key.decode()) # your fernet_key, keep it in secured place!
   ```

3. Replace `airflow.cfg` fernet_key value with the one from *Step 2*. *Alternatively,* you can store your fernet_key in OS environment variable - You do not need to change `airflow.cfg` in this case as Airflow will use environment variable over the value in `airflow.cfg`:

   ```
   # Note the double underscores
   export AIRFLOW__CORE__FERNET_KEY=your_fernet_key
   ```

4. Restart the webserver
5. For existing connections (the ones that you had defined before installing `airflow[crypto]` and creating a Fernet key), you need to open each connection in the connection admin UI, re-type the password, and save the change

3.6.7 Rotating encryption keys

Once connection credentials and variables have been encrypted using a fernet key, changing the key will cause decryption of existing credentials to fail. To rotate the fernet key without invalidating existing encrypted values, prepend the new key to the `fernet_key` setting, run `airflow rotate_fernet_key`, and then drop the original key from `fernet_keys`:

1. Set `fernet_key` to `new_fernet_key,old_fernet_key`
2. Run `airflow rotate_fernet_key` to re-encrypt existing credentials with the new fernet key
3. Set `fernet_key` to `new_fernet_key`

3.6.8 Writing Logs

3.6.8.1 Writing Logs Locally

Users can specify the directory to place log files in `airflow.cfg` using `base_log_folder`. By default, logs are placed in the `AIRFLOW_HOME` directory.

The following convention is followed while naming logs: `{dag_id}/{task_id}/{execution_date}/{try_number}.log`

In addition, users can supply a remote location to store current logs and backups.

In the Airflow Web UI, local logs take precedence over remote logs. If local logs can not be found or accessed, the remote logs will be displayed. Note that logs are only sent to remote storage once a task is complete (including failure); In other words, remote logs for running tasks are unavailable.

Before you begin

Remote logging uses an existing Airflow connection to read or write logs. If you don't have a connection properly setup, this process will fail.

3.6.8.2 Writing Logs to Amazon S3

Enabling remote logging

To enable this feature, `airflow.cfg` must be configured as follows:

```
[core]
# Airflow can store logs remotely in AWS S3. Users must supply a remote
# location URL (starting with either 's3://...') and an Airflow connection
# id that provides access to the storage location.
remote_logging = True
remote_base_log_folder = s3://my-bucket/path/to/logs
remote_log_conn_id = MyS3Conn
# Use server-side encryption for logs stored in S3
encrypt_s3_logs = False
```

In the above example, Airflow will try to use `S3Hook('MyS3Conn')`.

3.6.8.3 Writing Logs to Azure Blob Storage

Airflow can be configured to read and write task logs in Azure Blob Storage.

Follow the steps below to enable Azure Blob Storage logging:

1. Airflow's logging system requires a custom *.py* file to be located in the `PYTHONPATH`, so that it's importable from Airflow. Start by creating a directory to store the config file, `$AIRFLOW_HOME/config` is recommended.
2. Create empty files called `$AIRFLOW_HOME/config/log_config.py` and `$AIRFLOW_HOME/config/__init__.py`.
3. Copy the contents of `airflow/config_templates/airflow_local_settings.py` into the `log_config.py` file created in *Step 2*.
4. Customize the following portions of the template:

```
# wasb buckets should start with "wasb" just to help Airflow select␣
↪correct handler
REMOTE_BASE_LOG_FOLDER = 'wasb-<whatever you want here>'

# Rename DEFAULT_LOGGING_CONFIG to LOGGING CONFIG
LOGGING_CONFIG = ...
```

5. Make sure a Azure Blob Storage (Wasb) connection hook has been defined in Airflow. The hook should have read and write access to the Azure Blob Storage bucket defined above in `REMOTE_BASE_LOG_FOLDER`.
6. Update `$AIRFLOW_HOME/airflow.cfg` to contain:

```
remote_logging = True
logging_config_class = log_config.LOGGING_CONFIG
remote_log_conn_id = <name of the Azure Blob Storage connection>
```

7. Restart the Airflow webserver and scheduler, and trigger (or wait for) a new task execution.
8. Verify that logs are showing up for newly executed tasks in the bucket you've defined.

3.6.8.4 Writing Logs to Google Cloud Storage

Follow the steps below to enable Google Cloud Storage logging.

To enable this feature, `airflow.cfg` must be configured as in this example:

```
[core]
# Airflow can store logs remotely in AWS S3, Google Cloud Storage or Elastic Search.
# Users must supply an Airflow connection id that provides access to the storage
# location. If remote_logging is set to true, see UPDATING.md for additional
# configuration requirements.
remote_logging = True
remote_base_log_folder = gs://my-bucket/path/to/logs
remote_log_conn_id = MyGCSConn
```

1. Install the `gcp` package first, like so: `pip install 'apache-airflow[gcp]'`.
2. Make sure a Google Cloud Platform connection hook has been defined in Airflow. The hook should have read and write access to the Google Cloud Storage bucket defined above in `remote_base_log_folder`.
3. Restart the Airflow webserver and scheduler, and trigger (or wait for) a new task execution.
4. Verify that logs are showing up for newly executed tasks in the bucket you've defined.
5. Verify that the Google Cloud Storage viewer is working in the UI. Pull up a newly executed task, and verify that you see something like:

```
*** Reading remote log from gs://<bucket where logs should be persisted>/example_bash_
↪operator/run_this_last/2017-10-03T00:00:00/16.log.
[2017-10-03 21:57:50,056] {cli.py:377} INFO - Running on host chrisr-00532
[2017-10-03 21:57:50,093] {base_task_runner.py:115} INFO - Running: ['bash', '-c',
↪'airflow run example_bash_operator run_this_last 2017-10-03T00:00:00 --job_id 47 --
↪raw -sd DAGS_FOLDER/example_dags/example_bash_operator.py']
[2017-10-03 21:57:51,264] {base_task_runner.py:98} INFO - Subtask: [2017-10-03␣
↪21:57:51,263] {__init__.py:45} INFO - Using executor SequentialExecutor
[2017-10-03 21:57:51,306] {base_task_runner.py:98} INFO - Subtask: [2017-10-03␣
↪21:57:51,306] {models.py:186} INFO - Filling up the DagBag from /airflow/dags/
↪example_dags/example_bash_operator.py
```

Note that the path to the remote log file is listed on the first line.

3.6.8.5 Writing Logs to Elasticsearch

Airflow can be configured to read task logs from Elasticsearch and optionally write logs to stdout in standard or json format. These logs can later be collected and forwarded to the Elasticsearch cluster using tools like fluentd, logstash or others.

You can choose to have all task logs from workers output to the highest parent level process, instead of the standard file locations. This allows for some additional flexibility in container environments like Kubernetes, where container stdout is already being logged to the host nodes. From there a log shipping tool can be used to forward them along to Elasticsearch. To use this feature, set the `write_stdout` option in `airflow.cfg`. You can also choose to have the logs output in a JSON format, using the `json_format` option. Airflow uses the standard Python logging module and JSON fields are directly extracted from the LogRecord object. To use this feature, set the `json_fields` option in `airflow.cfg`. Add the fields to the comma-delimited string that you want collected for the logs. These fields are from the LogRecord object in the `logging` module. Documentation on different attributes can be found here.

First, to use the handler, airflow.cfg must be configured as follows:

```
[core]
# Airflow can store logs remotely in AWS S3, Google Cloud Storage or Elastic Search.
# Users must supply an Airflow connection id that provides access to the storage
# location. If remote_logging is set to true, see UPDATING.md for additional
# configuration requirements.
remote_logging = True

[elasticsearch]
log_id_template = {{dag_id}}-{{task_id}}-{{execution_date}}-{{try_number}}
end_of_log_mark = end_of_log
write_stdout =
json_fields =
```

To output task logs to stdout in JSON format, the following config could be used:

```
[core]
# Airflow can store logs remotely in AWS S3, Google Cloud Storage or Elastic Search.
# Users must supply an Airflow connection id that provides access to the storage
# location. If remote_logging is set to true, see UPDATING.md for additional
# configuration requirements.
remote_logging = True

[elasticsearch]
log_id_template = {{dag_id}}-{{task_id}}-{{execution_date}}-{{try_number}}
end_of_log_mark = end_of_log
write_stdout = True
json_format = True
json_fields = asctime, filename, lineno, levelname, message
```

3.6.9 Scaling Out with Celery

`CeleryExecutor` is one of the ways you can scale out the number of workers. For this to work, you need to setup a Celery backend (**RabbitMQ**, **Redis**, ...) and change your `airflow.cfg` to point the executor parameter to `CeleryExecutor` and provide the related Celery settings.

For more information about setting up a Celery broker, refer to the exhaustive Celery documentation on the topic.

Here are a few imperative requirements for your workers:

- `airflow` needs to be installed, and the CLI needs to be in the path
- Airflow configuration settings should be homogeneous across the cluster

- Operators that are executed on the worker need to have their dependencies met in that context. For example, if you use the `HiveOperator`, the hive CLI needs to be installed on that box, or if you use the `MySqlOperator`, the required Python library needs to be available in the `PYTHONPATH` somehow
- The worker needs to have access to its `DAGS_FOLDER`, and you need to synchronize the filesystems by your own means. A common setup would be to store your DAGS_FOLDER in a Git repository and sync it across machines using Chef, Puppet, Ansible, or whatever you use to configure machines in your environment. If all your boxes have a common mount point, having your pipelines files shared there should work as well

To kick off a worker, you need to setup Airflow and kick off the worker subcommand

```
airflow worker
```

Your worker should start picking up tasks as soon as they get fired in its direction.

Note that you can also run "Celery Flower", a web UI built on top of Celery, to monitor your workers. You can use the shortcut command `airflow flower` to start a Flower web server.

Please note that you must have the `flower` python library already installed on your system. The recommend way is to install the airflow celery bundle.

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

Some caveats:

- Make sure to use a database backed result backend
- Make sure to set a visibility timeout in [celery_broker_transport_options] that exceeds the ETA of your longest running task
- Tasks can consume resources. Make sure your worker has enough resources to run *worker_concurrency* tasks
- Queue names are limited to 256 characters, but each broker backend might have its own restrictions

3.6.10 Scaling Out with Dask

`airflow.executors.dask_executor.DaskExecutor` allows you to run Airflow tasks in a Dask Distributed cluster.

Dask clusters can be run on a single machine or on remote networks. For complete details, consult the Distributed documentation.

To create a cluster, first start a Scheduler:

```
# default settings for a local cluster
DASK_HOST=127.0.0.1
DASK_PORT=8786

dask-scheduler --host $DASK_HOST --port $DASK_PORT
```

Next start at least one Worker on any machine that can connect to the host:

```
dask-worker $DASK_HOST:$DASK_PORT
```

Edit your `airflow.cfg` to set your executor to *`airflow.executors.dask_executor.DaskExecutor`* and provide the Dask Scheduler address in the `[dask]` section.

Please note:

- Each Dask worker must be able to import Airflow and any dependencies you require.
- Dask does not support queues. If an Airflow task was created with a queue, a warning will be raised but the task will be submitted to the cluster.

3.6.11 Running Airflow behind a reverse proxy

Airflow can be set up behind a reverse proxy, with the ability to set its endpoint with great flexibility.

For example, you can configure your reverse proxy to get:

```
https://lab.mycompany.com/myorg/airflow/
```

To do so, you need to set the following setting in your *airflow.cfg*:

```
base_url = http://my_host/myorg/airflow
```

Additionally if you use Celery Executor, you can get Flower in */myorg/flower* with:

```
flower_url_prefix = /myorg/flower
```

Your reverse proxy (ex: nginx) should be configured as follow:

- pass the url and http header as it for the Airflow webserver, without any rewrite, for example:

```
server {
  listen 80;
  server_name lab.mycompany.com;

  location /myorg/airflow/ {
      proxy_pass http://localhost:8080;
      proxy_set_header Host $host;
      proxy_redirect off;
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection "upgrade";
  }
}
```

- rewrite the url for the flower endpoint:

```
server {
    listen 80;
    server_name lab.mycompany.com;

    location /myorg/flower/ {
        rewrite ^/myorg/flower/(.*)$ /$1 break;  # remove prefix from http header
        proxy_pass http://localhost:5555;
        proxy_set_header Host $host;
        proxy_redirect off;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
```

To ensure that Airflow generates URLs with the correct scheme when running behind a TLS-terminating proxy, you should configure the proxy to set the *X-Forwarded-Proto* header, and enable the *ProxyFix* middleware in your *airflow.cfg*:

```
enable_proxy_fix = True
```

Note: You should only enable the *ProxyFix* middleware when running Airflow behind a trusted proxy (AWS ELB, nginx, etc.).

3.6.12 Running Airflow with systemd

Airflow can integrate with systemd based systems. This makes watching your daemons easy as *systemd* can take care of restarting a daemon on failures.

In the `scripts/systemd` directory, you can find unit files that have been tested on Redhat based systems. These files can be used as-is by copying them over to `/usr/lib/systemd/system`.

The following **assumptions** have been made while creating these unit files:

1. Airflow runs as the following *user:group* `airflow:airflow`.
2. Airflow runs on a Redhat based system.

If this is not the case, appropriate changes will need to be made.

Please **note** that environment configuration is picked up from `/etc/sysconfig/airflow`.

An example file is supplied within `scripts/systemd`. You can also define configuration at `AIRFLOW_HOME` or `AIRFLOW_CONFIG`.

3.6.13 Running Airflow with upstart

Airflow can integrate with upstart based systems. Upstart automatically starts all airflow services for which you have a corresponding `*.conf` file in `/etc/init` upon system boot. On failure, upstart automatically restarts the process (until it reaches re-spawn limit set in a `*.conf` file).

You can find sample upstart job files in the `scripts/upstart` directory.

The following assumptions have been used while creating these unit files:

1. **Airflow will run as the following *user:group* `airflow:airflow`.** Change `setuid` and `setgid` appropriately in `*.conf` if airflow runs as a different user or group
2. **These files have been tested on Ubuntu 14.04 LTS** You may have to adjust `start on` and `stop on` stanzas to make it work on other upstart systems. Some of the possible options are listed in `scripts/upstart/README`

Modify `*.conf` files as needed and copy to `/etc/init` directory.

You can use `initctl` to manually start, stop, view status of the airflow process that has been integrated with upstart

```
initctl airflow-webserver status
```

3.6.14 Using the Test Mode Configuration

Airflow has a fixed set of "test mode" configuration options. You can load these at any time by calling `airflow.configuration.load_test_config()`. Please **note** that this operation is **not reversible**.

Some options for example, DAG_FOLDER, are loaded before you have a chance to call load_test_config(). In order to eagerly load the test configuration, set test_mode in airflow.cfg:

```
[tests]
unit_test_mode = True
```

Due to Airflow's automatic environment variable expansion (see *Setting Configuration Options*), you can also set the environment variable `AIRFLOW__CORE__UNIT_TEST_MODE` to temporarily overwrite airflow.cfg.

3.6.15 Checking Airflow Health Status

To check the health status of your Airflow instance, you can simply access the endpoint `/health`. It will return a JSON object in which a high-level glance is provided.

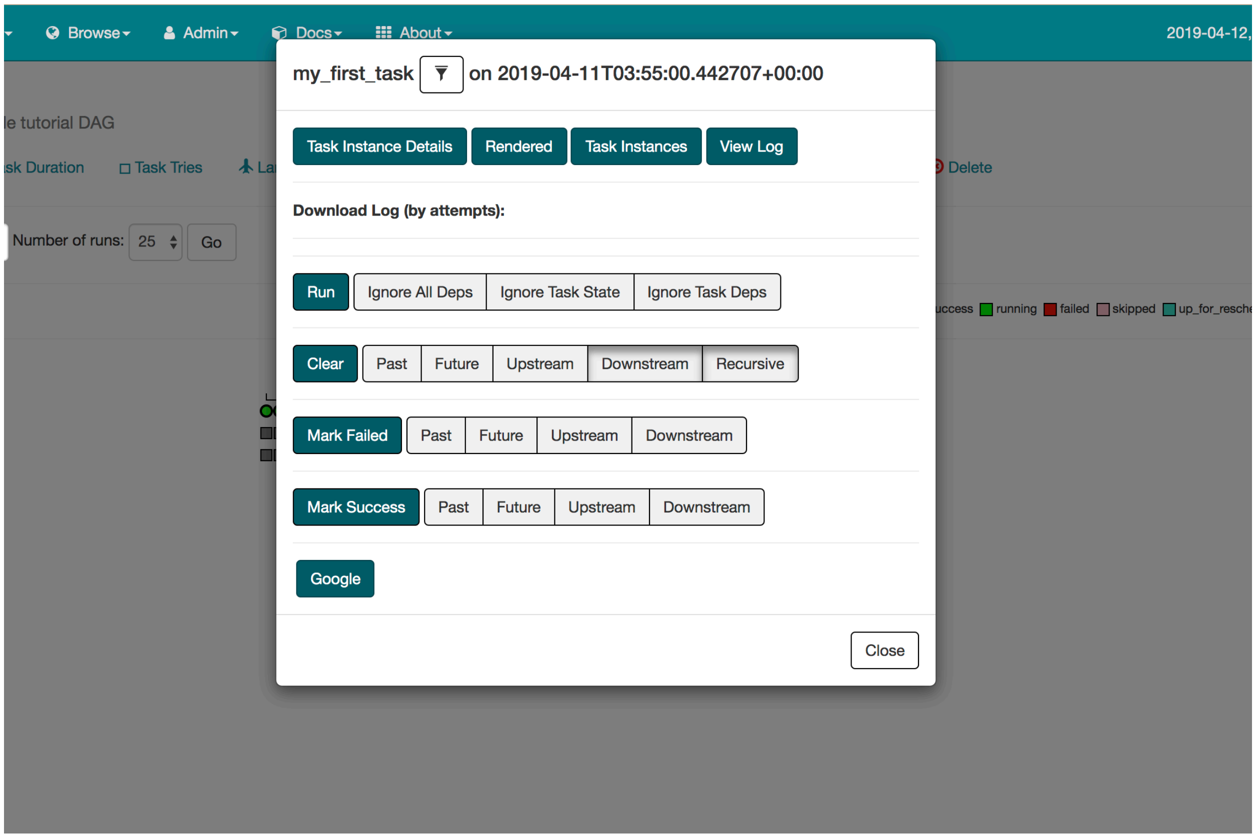
```
{
  "metadatabase":{
    "status":"healthy"
  },
  "scheduler":{
    "status":"healthy",
    "latest_scheduler_heartbeat":"2018-12-26 17:15:11+00:00"
  }
}
```

- The `status` of each component can be either "healthy" or "unhealthy"
  - The status of `metadatabase` depends on whether a valid connection can be initiated with the database
  - The status of `scheduler` depends on when the latest scheduler heartbeat was received
    * If the last heartbeat was received more than 30 seconds (default value) earlier than the current time, the scheduler is considered unhealthy
    * This threshold value can be specified using the option `scheduler_health_check_threshold` within the `scheduler` section in `airflow.cfg`

Please keep in mind that the HTTP response code of `/health` endpoint **should not** be used to determine the health status of the application. The return code is only indicative of the state of the rest call (200 for success).

3.6.16 Define an operator extra link

For each operator, you can define its own extra links that can redirect users to external systems. The extra link buttons will be available on the task page:



The following code shows how to add extra links to an operator:

```
from airflow.models.baseoperator import BaseOperator, BaseOperatorLink
from airflow.utils.decorators import apply_defaults


class GoogleLink(BaseOperatorLink):

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

class MyFirstOperator(BaseOperator):

    operator_extra_link_dict = {
        "Google": GoogleLink(),
    }

    @apply_defaults
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def execute(self, context):
        self.log.info("Hello World!")
```

You can also add a global operator extra link that will be available to all the operators through airflow plugin. Learn more about it in the *plugin example*.

3.6.17 Setup Test Environment using MySQL

By default, Airflow uses SQLite as database backend and `SequentialExecutor` to execute tasks as SQLite does not support multiple connections. Since `SequentialExecutor` runs one instance at a time, some parallel execution logic will not be exercised in this default setup.

To test out the parallel execution setup, we can use MySQL as database backend and `LocalExecutor` as the executor. Checkout the following setups to launch a MySQL database container:

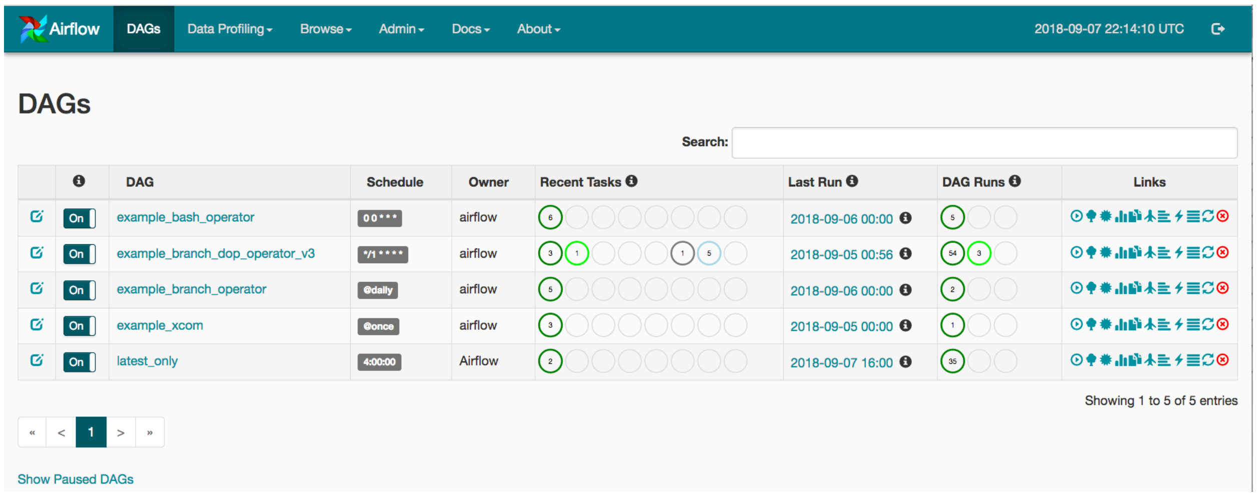
```
# Launch MySQL docker container
docker-compose -f scripts/ci/docker-compose.yml run -p3306:3306 mysql

# Open airflow.cfg and add the following:
# executor = LocalExecutor
vim $AIRFLOW_HOME/airflow.cfg

export AIRFLOW__CORE__SQL_ALCHEMY_CONN=mysql://root@127.0.0.1:3306/AIRFLOW_HOME

airflow initdb
```

3.7 UI / Screenshots

The Airflow UI makes it easy to monitor and troubleshoot your data pipelines. Here's a quick overview of some of the features and visualizations you can find in the Airflow UI.

3.7.1 DAGs View

List of the DAGs in your environment, and a set of shortcuts to useful pages. You can see exactly how many tasks succeeded, failed, or are currently running at a glance.



DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	On	example_bash_operator	0 0 * * *	airflow	6	2018-09-06 00:00	5	
	On	example_branch_dop_operator_v3	*/1 * * * *	airflow	3 1 1 5	2018-09-05 00:56	54 3	
	On	example_branch_operator	@daily	airflow	5	2018-09-06 00:00	2	
	On	example_xcom	@once	airflow	3	2018-09-05 00:00	1	
	On	latest_only	4:00:00	Airflow	2	2018-09-07 16:00	35	

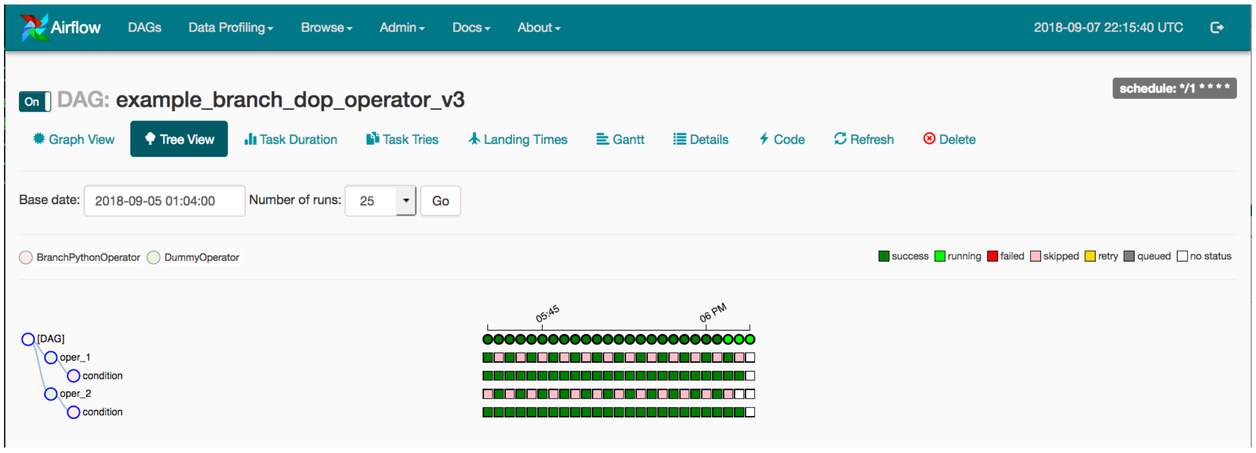
Showing 1 to 5 of 5 entries

« < 1 > »

Show Paused DAGs

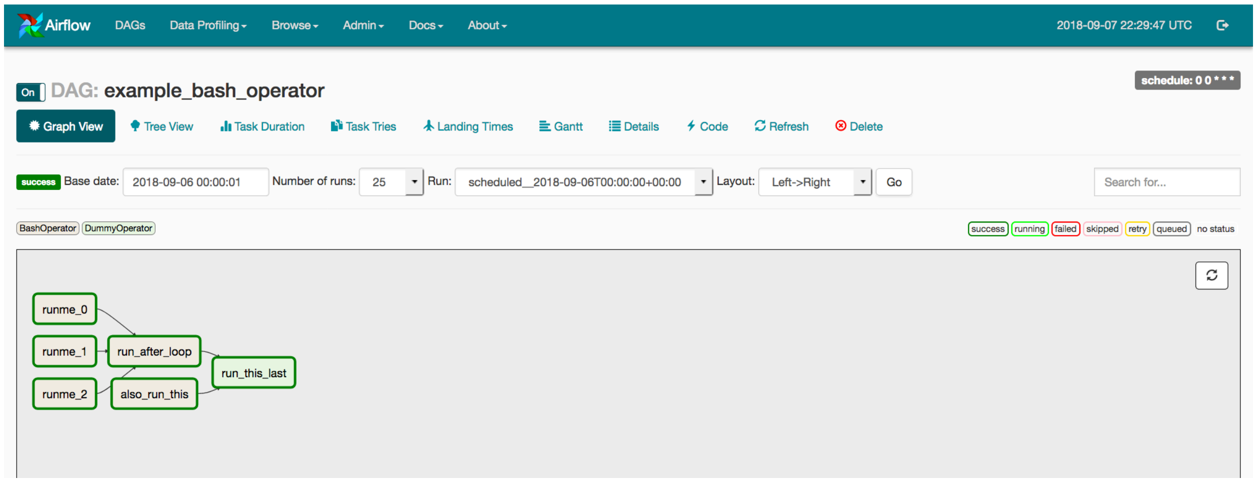
3.7.2 Tree View

A tree representation of the DAG that spans across time. If a pipeline is late, you can quickly see where the different steps are and identify the blocking ones.



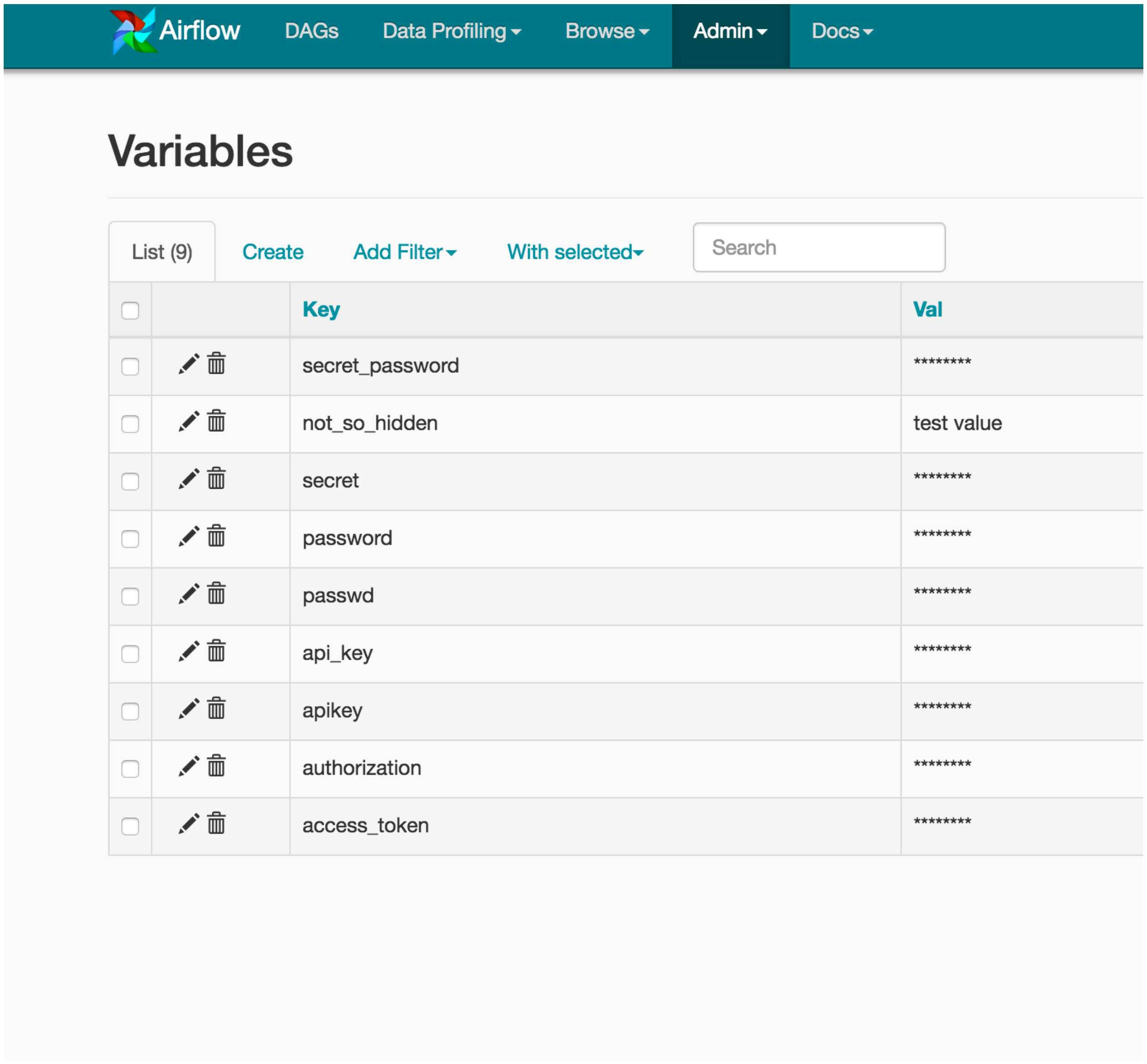
3.7.3 Graph View

The graph view is perhaps the most comprehensive. Visualize your DAG's dependencies and their current status for a specific run.



3.7.4 Variable View

The variable view allows you to list, create, edit or delete the key-value pair of a variable used during jobs. Value of a variable will be hidden if the key contains any words in ('password', 'secret', 'passwd', 'authorization', 'api_key', 'apikey', 'access_token') by default, but can be configured to show in clear-text.

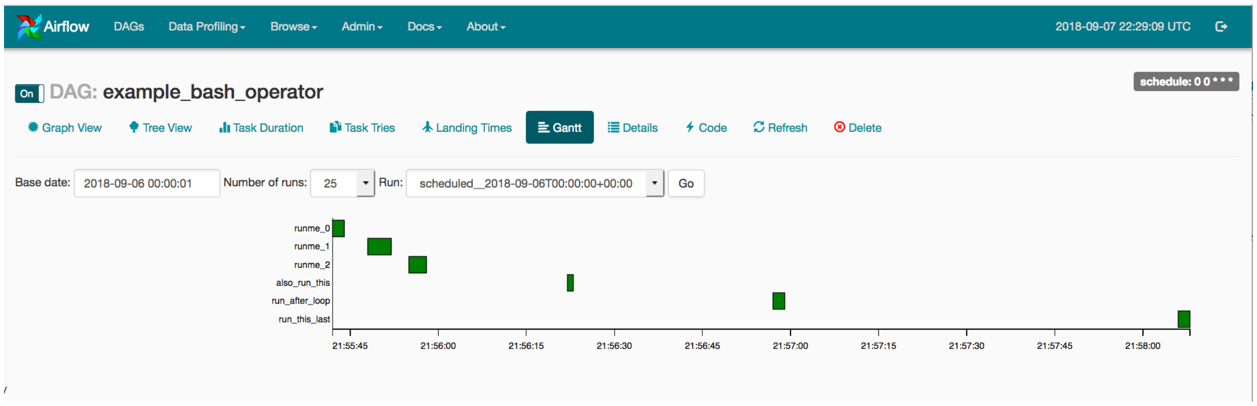


Variables

Key	Val
secret_password	********
not_so_hidden	test value
secret	********
password	********
passwd	********
api_key	********
apikey	********
authorization	********
access_token	********

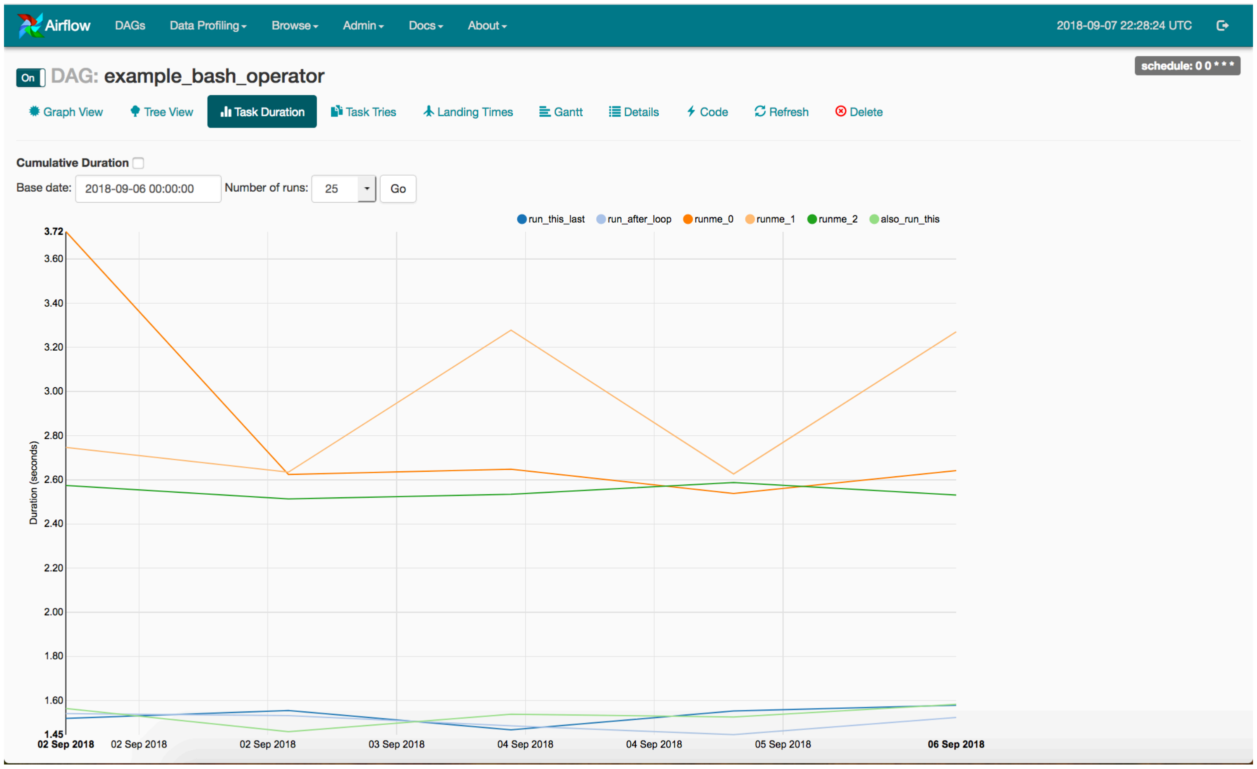
3.7.5 Gantt Chart

The Gantt chart lets you analyse task duration and overlap. You can quickly identify bottlenecks and where the bulk of the time is spent for specific DAG runs.



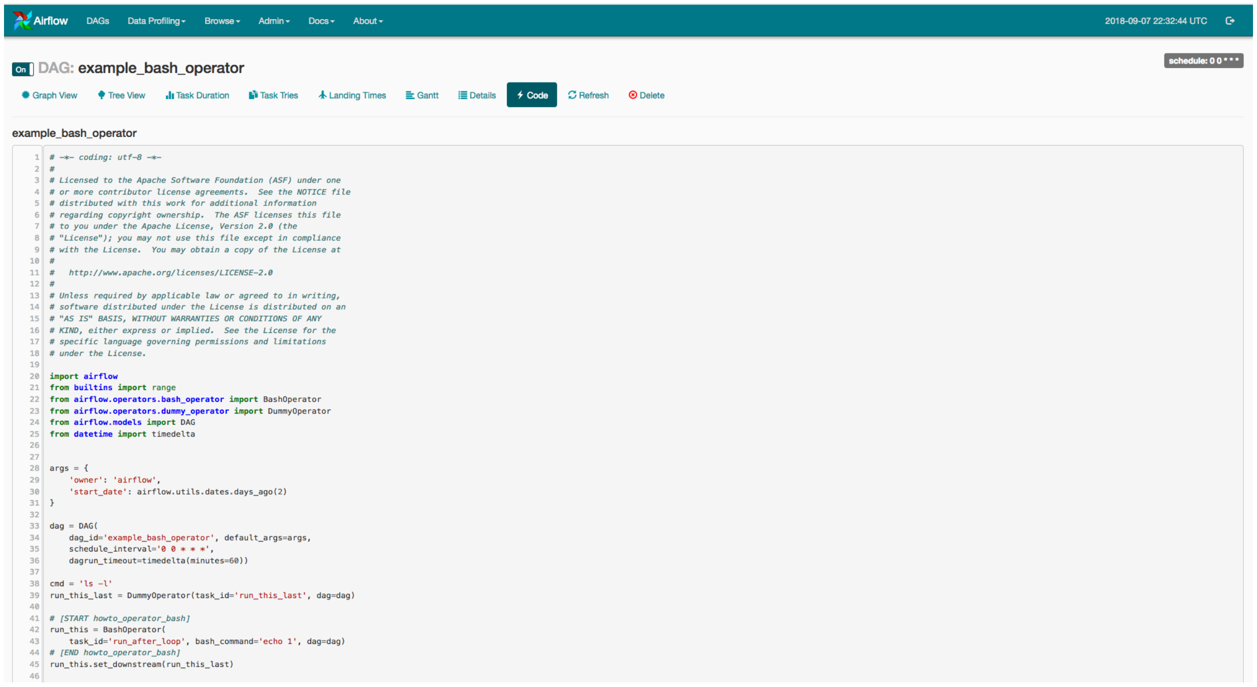
3.7.6 Task Duration

The duration of your different tasks over the past N runs. This view lets you find outliers and quickly understand where the time is spent in your DAG over many runs.



3.7.7 Code View

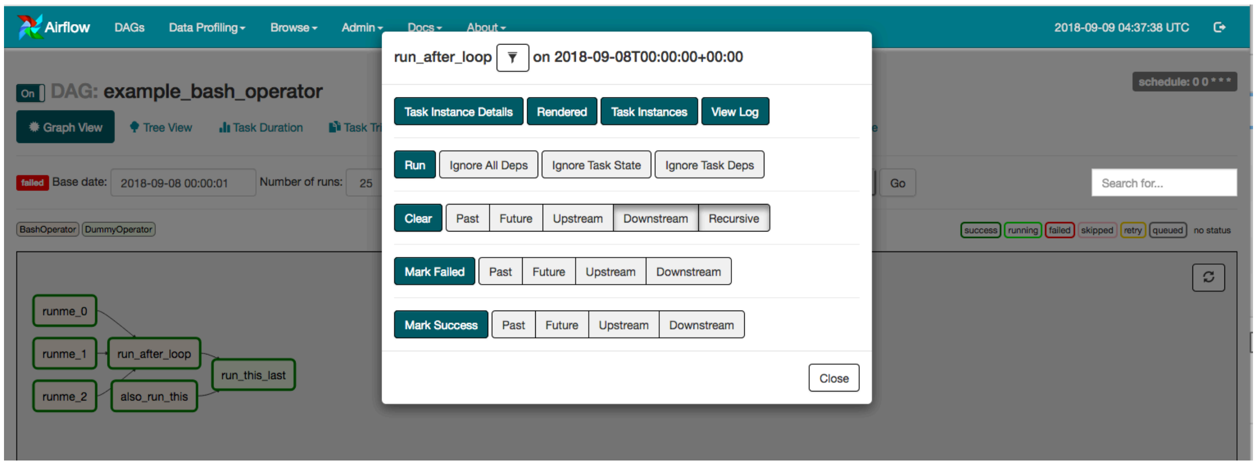
Transparency is everything. While the code for your pipeline is in source control, this is a quick way to get to the code that generates the DAG and provide yet more context.



```
# -*- coding: utf-8 -*-
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.

import airflow
from builtins import range
from airflow.operators.bash_operator import BashOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.models import DAG
from datetime import timedelta


args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2)
}

dag = DAG(
    dag_id='example_bash_operator', default_args=args,
    schedule_interval='0 0 * * *',
    dagrun_timeout=timedelta(minutes=60))

cmd = 'ls -l'
run_this_last = DummyOperator(task_id='run_this_last', dag=dag)

# [START howto_operator_bash]
run_this = BashOperator(
    task_id='run_after_loop', bash_command='echo 1', dag=dag)
# [END howto_operator_bash]
run_this.set_downstream(run_this_last)

```

3.7.8 Task Instance Context Menu

From the pages seen above (tree view, graph view, gantt, …), it is always possible to click on a task instance, and get to this rich context menu that can take you to more detailed metadata, and perform some actions.



3.8 Concepts

The Airflow platform is a tool for describing, executing, and monitoring workflows.

3.8.1 Core Ideas

3.8.1.1 DAGs

In Airflow, a `DAG` – or a Directed Acyclic Graph – is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

For example, a simple DAG could consist of three tasks: A, B, and C. It could say that A has to run successfully before B can run, but C can run anytime. It could say that task A times out after 5 minutes, and B can be restarted up to 5 times in case it fails. It might also say that the workflow will run every night at 10pm, but shouldn't start until a certain date.

In this way, a DAG describes *how* you want to carry out your workflow; but notice that we haven't said anything about *what* we actually want to do! A, B, and C could be anything. Maybe A prepares data for B to analyze while C sends an email. Or perhaps A monitors your location so B can open your garage door while C turns on your house lights. The important thing is that the DAG isn't concerned with what its constituent tasks do; its job is to make sure that whatever they do happens at the right time, or in the right order, or with the right handling of any unexpected issues.

DAGs are defined in standard Python files that are placed in Airflow's `DAG_FOLDER`. Airflow will execute the code in each file to dynamically build the `DAG` objects. You can have as many DAGs as you want, each describing an arbitrary number of tasks. In general, each one should correspond to a single logical workflow.

Note: When searching for DAGs, Airflow only considers python files that contain the strings "airflow" and "DAG" by default. To consider all python files instead, disable the `DAG_DISCOVERY_SAFE_MODE` configuration flag.

Scope

Airflow will load any `DAG` object it can import from a DAGfile. Critically, that means the DAG must appear in `globals()`. Consider the following two DAGs. Only `dag_1` will be loaded; the other one only appears in a local scope.

```
dag_1 = DAG('this_dag_will_be_discovered')

def my_function():
    dag_2 = DAG('but_this_dag_will_not')

my_function()
```

Sometimes this can be put to good use. For example, a common pattern with `SubDagOperator` is to define the subdag inside a function so that Airflow doesn't try to load it as a standalone DAG.

Default Arguments

If a dictionary of `default_args` is passed to a DAG, it will apply them to any of its operators. This makes it easy to apply a common parameter to many operators without having to type it many times.

```
default_args = {
    'start_date': datetime(2016, 1, 1),
    'owner': 'Airflow'
}

dag = DAG('my_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(op.owner) # Airflow
```

Context Manager

Added in Airflow 1.8

DAGs can be used as context managers to automatically assign new operators to that DAG.

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    op = DummyOperator('op')

op.dag is dag # True
```

3.8.1.2 Operators

While DAGs describe *how* to run a workflow, `Operators` determine what actually gets done.

An operator describes a single task in a workflow. Operators are usually (but not always) atomic, meaning they can stand on their own and don't need to share resources with any other operators. The DAG will make sure that operators run in the correct certain order; other than those dependencies, operators generally run independently. In fact, they may run on two completely different machines.

This is a subtle but very important point: in general, if two operators need to share information, like a filename or small amount of data, you should consider combining them into a single operator. If it absolutely can't be avoided, Airflow does have a feature for operator cross-communication called XCom that is described elsewhere in this document.

Airflow provides operators for many common tasks, including:

- *`BashOperator`* - executes a bash command
- *`PythonOperator`* - calls an arbitrary Python function
- *`EmailOperator`* - sends an email
- *`SimpleHttpOperator`* - sends an HTTP request
- *`MySqlOperator`*, *`SqliteOperator`*, *`PostgresOperator`*, *`MsSqlOperator`*, *`OracleOperator`*, *`JdbcOperator`*, etc. - executes a SQL command
- `Sensor` - waits for a certain time, file, database row, S3 key, etc…

In addition to these basic building blocks, there are many more specific operators: *`DockerOperator`*, *`HiveOperator`*, *`S3FileTransformOperator`*, *`PrestoToMySqlTransfer`*, *`SlackAPIOperator`*… you get the idea!

Operators are only loaded by Airflow if they are assigned to a DAG.

See *Using Operators* for how to use Airflow operators.

DAG Assignment

Added in Airflow 1.8

Operators do not have to be assigned to DAGs immediately (previously `dag` was a required argument). However, once an operator is assigned to a DAG, it can not be transferred or unassigned. DAG assignment can be done explicitly when the operator is created, through deferred assignment, or even inferred from other operators.

```
dag = DAG('my_dag', start_date=datetime(2016, 1, 1))

# sets the DAG explicitly
explicit_op = DummyOperator(task_id='op1', dag=dag)

```

(continues on next page)

(continued from previous page)

```
# deferred DAG assignment
deferred_op = DummyOperator(task_id='op2')
deferred_op.dag = dag

# inferred DAG assignment (linked operators must be in the same DAG)
inferred_op = DummyOperator(task_id='op3')
inferred_op.set_upstream(deferred_op)
```

Bitshift Composition

Added in Airflow 1.8

Traditionally, operator relationships are set with the `set_upstream()` and `set_downstream()` methods. In Airflow 1.8, this can be done with the Python bitshift operators >> and <<. The following four statements are all functionally equivalent:

```
op1 >> op2
op1.set_downstream(op2)

op2 << op1
op2.set_upstream(op1)
```

When using the bitshift to compose operators, the relationship is set in the direction that the bitshift operator points. For example, `op1 >> op2` means that `op1` runs first and `op2` runs second. Multiple operators can be composed – keep in mind the chain is executed left-to-right and the rightmost object is always returned. For example:

```
op1 >> op2 >> op3 << op4
```

is equivalent to:

```
op1.set_downstream(op2)
op2.set_downstream(op3)
op3.set_upstream(op4)
```

For convenience, the bitshift operators can also be used with DAGs. For example:

```
dag >> op1 >> op2
```

is equivalent to:

```
op1.dag = dag
op1.set_downstream(op2)
```

We can put this all together to build a simple pipeline:

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    (
        DummyOperator(task_id='dummy_1')
        >> BashOperator(
            task_id='bash_1',
            bash_command='echo "HELLO!"')
        >> PythonOperator(
            task_id='python_1',
            python_callable=lambda: print("GOODBYE!"))
    )
```

Bitshift can also be used with lists. For example:

```
op1 >> [op2, op3]
```

is equivalent to:

```
op1 >> op2
op1 >> op3
```

and equivalent to:

```
op1.set_downstream([op2, op3])
```

3.8.1.3 Tasks

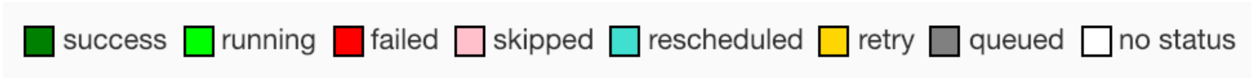
Once an operator is instantiated, it is referred to as a "task". The instantiation defines specific values when calling the abstract operator, and the parameterized task becomes a node in a DAG.

3.8.1.4 Task Instances

A task instance represents a specific run of a task and is characterized as the combination of a DAG, a task, and a point in time. Task instances also have an indicative state, which could be "running", "success", "failed", "skipped", "up for retry", etc.

3.8.1.5 Task Lifecycle

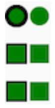
A task goes through various stages from start to completion. In the Airflow UI (graph and tree views), these stages are displayed by a color representing each stage:



The happy flow consists of the following stages:

1. no status (scheduler created empty task instance)
2. queued (scheduler placed a task to run on the queue)
3. running (worker picked up a task and is now running it)
4. success (task completed)

There is also visual difference between scheduled and manually triggered DAGs/tasks:



The DAGs/tasks with a black border are scheduled runs, whereas the non-bordered DAGs/tasks are manually triggered, i.e. by *airflow trigger_dag*.

3.8.1.6 Workflows

You're now familiar with the core building blocks of Airflow. Some of the concepts may sound very similar, but the vocabulary can be conceptualized like this:

- DAG: a description of the order in which work should take place

- Operator: a class that acts as a template for carrying out some work
- Task: a parameterized instance of an operator
- Task Instance: a task that 1) has been assigned to a DAG and 2) has a state associated with a specific run of the DAG

By combining `DAGs` and `Operators` to create `TaskInstances`, you can build complex workflows.

3.8.2 Additional Functionality

In addition to the core Airflow objects, there are a number of more complex features that enable behaviors like limiting simultaneous access to resources, cross-communication, conditional execution, and more.

3.8.2.1 Hooks

Hooks are interfaces to external platforms and databases like Hive, S3, MySQL, Postgres, HDFS, and Pig. Hooks implement a common interface when possible, and act as a building block for operators. They also use the `airflow.models.connection.Connection` model to retrieve hostnames and authentication information. Hooks keep authentication code and information out of pipelines, centralized in the metadata database.

Hooks are also very useful on their own to use in Python scripts, Airflow airflow.operators.PythonOperator, and in interactive environments like iPython or Jupyter Notebook.

3.8.2.2 Pools

Some systems can get overwhelmed when too many processes hit them at the same time. Airflow pools can be used to **limit the execution parallelism** on arbitrary sets of tasks. The list of pools is managed in the UI (`Menu -> Admin -> Pools`) by giving the pools a name and assigning it a number of worker slots. Tasks can then be associated with one of the existing pools by using the `pool` parameter when creating tasks (i.e., instantiating operators).

```
aggregate_db_message_job = BashOperator(
    task_id='aggregate_db_message_job',
    execution_timeout=timedelta(hours=3),
    pool='ep_data_pipeline_db_msg_agg',
    bash_command=aggregate_db_message_job_cmd,
    dag=dag)
aggregate_db_message_job.set_upstream(wait_for_empty_queue)
```

The `pool` parameter can be used in conjunction with `priority_weight` to define priorities in the queue, and which tasks get executed first as slots open up in the pool. The default `priority_weight` is `1`, and can be bumped to any number. When sorting the queue to evaluate which task should be executed next, we use the `priority_weight`, summed up with all of the `priority_weight` values from tasks downstream from this task. You can use this to bump a specific important task and the whole path to that task gets prioritized accordingly.

Tasks will be scheduled as usual while the slots fill up. Once capacity is reached, runnable tasks get queued and their state will show as such in the UI. As slots free up, queued tasks start running based on the `priority_weight` (of the task and its descendants).

Note that by default tasks aren't assigned to any pool and their execution parallelism is only limited to the executor's setting.

To combine Pools with SubDAGs see the *SubDAGs* section.

3.8.2.3 Connections

The connection information to external systems is stored in the Airflow metadata database and managed in the UI (`Menu -> Admin -> Connections`). A `conn_id` is defined there and hostname / login / password / schema information attached to it. Airflow pipelines can simply refer to the centrally managed `conn_id` without having to hard code any of this information anywhere.

Many connections with the same `conn_id` can be defined and when that is the case, and when the **hooks** uses the `get_connection` method from `BaseHook`, Airflow will choose one connection randomly, allowing for some basic load balancing and fault tolerance when used in conjunction with retries.

Airflow also has the ability to reference connections via environment variables from the operating system. Then connection parameters must be saved in URI format.

If connections with the same `conn_id` are defined in both Airflow metadata database and environment variables, only the one in environment variables will be referenced by Airflow (for example, given `conn_id` `postgres_master`, Airflow will search for `AIRFLOW_CONN_POSTGRES_MASTER` in environment variables first and directly reference it if found, before it starts to search in metadata database).

Many hooks have a default `conn_id`, where operators using that hook do not need to supply an explicit connection ID. For example, the default `conn_id` for the *`PostgresHook`* is `postgres_default`.

See *Managing Connections* for how to create and manage connections.

3.8.2.4 Queues

When using the CeleryExecutor, the Celery queues that tasks are sent to can be specified. `queue` is an attribute of BaseOperator, so any task can be assigned to any queue. The default queue for the environment is defined in the `airflow.cfg`'s `celery -> default_queue`. This defines the queue that tasks get assigned to when not specified, as well as which queue Airflow workers listen to when started.

Workers can listen to one or multiple queues of tasks. When a worker is started (using the command `airflow worker`), a set of comma-delimited queue names can be specified (e.g. `airflow worker -q spark`). This worker will then only pick up tasks wired to the specified queue(s).

This can be useful if you need specialized workers, either from a resource perspective (for say very lightweight tasks where one worker could take thousands of tasks without a problem), or from an environment perspective (you want a worker running from within the Spark cluster itself because it needs a very specific environment and security rights).

3.8.2.5 XComs

XComs let tasks exchange messages, allowing more nuanced forms of control and shared state. The name is an abbreviation of "cross-communication". XComs are principally defined by a key, value, and timestamp, but also track attributes like the task/DAG that created the XCom and when it should become visible. Any object that can be pickled can be used as an XCom value, so users should make sure to use objects of appropriate size.

XComs can be "pushed" (sent) or "pulled" (received). When a task pushes an XCom, it makes it generally available to other tasks. Tasks can push XComs at any time by calling the `xcom_push()` method. In addition, if a task returns a value (either from its Operator's `execute()` method, or from a PythonOperator's `python_callable` function), then an XCom containing that value is automatically pushed.

Tasks call `xcom_pull()` to retrieve XComs, optionally applying filters based on criteria like `key`, source `task_ids`, and source `dag_id`. By default, `xcom_pull()` filters for the keys that are automatically given to XComs when they are pushed by being returned from execute functions (as opposed to XComs that are pushed manually).

If `xcom_pull` is passed a single string for `task_ids`, then the most recent XCom value from that task is returned; if a list of `task_ids` is passed, then a corresponding list of XCom values is returned.

```
# inside a PythonOperator called 'pushing_task'
def push_function():
    return value

# inside another PythonOperator where provide_context=True
def pull_function(**context):
    value = context['task_instance'].xcom_pull(task_ids='pushing_task')
```

It is also possible to pull XCom directly in a template, here's an example of what this may look like:

```
SELECT * FROM {{ task_instance.xcom_pull(task_ids='foo', key='table_name') }}
```

Note that XComs are similar to *Variables*, but are specifically designed for inter-task communication rather than global settings.

3.8.2.6 Variables

Variables are a generic way to store and retrieve arbitrary content or settings as a simple key value store within Airflow. Variables can be listed, created, updated and deleted from the UI (`Admin -> Variables`), code or CLI. In addition, json settings files can be bulk uploaded through the UI. While your pipeline code definition and most of your constants and variables should be defined in code and stored in source control, it can be useful to have some variables or configuration items accessible and modifiable through the UI.

```
from airflow.models import Variable
foo = Variable.get("foo")
bar = Variable.get("bar", deserialize_json=True)
baz = Variable.get("baz", default_var=None)
```

The second call assumes `json` content and will be deserialized into `bar`. Note that `Variable` is a sqlalchemy model and can be used as such. The third call uses the `default_var` parameter with the value `None`, which either returns an existing value or `None` if the variable isn't defined. The get function will throw a `KeyError` if the variable doesn't exist and no default is provided.

You can use a variable from a jinja template with the syntax :

```
echo {{ var.value.<variable_name> }}
```

or if you need to deserialize a json object from the variable :

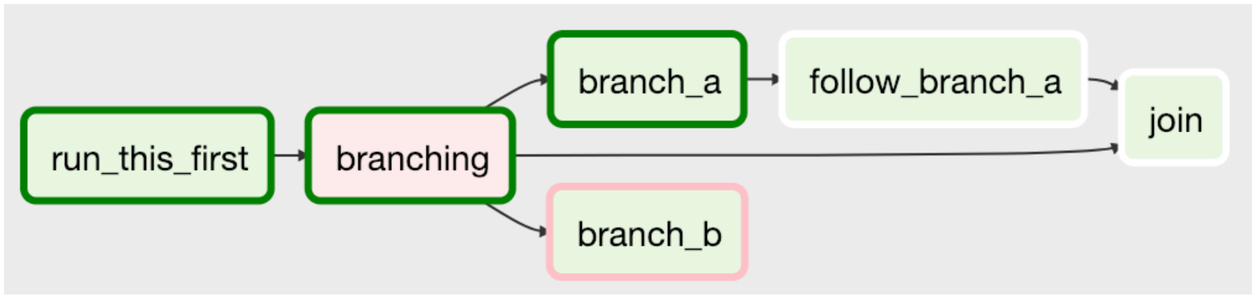
```
echo {{ var.json.<variable_name> }}
```

3.8.2.7 Branching

Sometimes you need a workflow to branch, or only go down a certain path based on an arbitrary condition which is typically related to something that happened in an upstream task. One way to do this is by using the `BranchPythonOperator`.

The `BranchPythonOperator` is much like the PythonOperator except that it expects a `python_callable` that returns a task_id (or list of task_ids). The task_id returned is followed, and all of the other paths are skipped. The task_id returned by the Python function has to reference a task directly downstream from the BranchPythonOperator task.

Note that when a path is a downstream task of the returned task (list), it will not be skipped:



Paths of the branching task are `branch_a`, `join` and `branch_b`. Since `join` is a downstream task of `branch_a`, it will be excluded from the skipped tasks when `branch_a` is returned by the Python callable.

The `BranchPythonOperator` can also be used with XComs allowing branching context to dynamically decide what branch to follow based on previous tasks. For example:

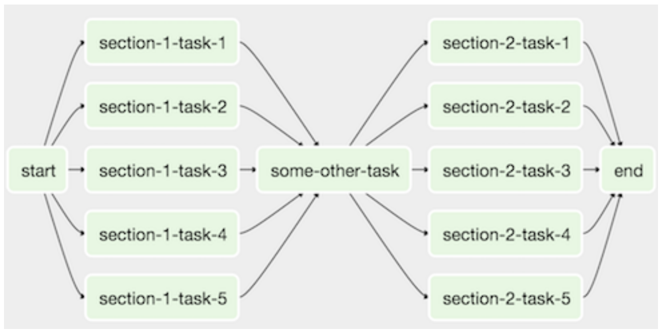
```
def branch_func(**kwargs):
    ti = kwargs['ti']
    xcom_value = int(ti.xcom_pull(task_ids='start_task'))
    if xcom_value >= 5:
        return 'continue_task'
    else:
        return 'stop_task'

start_op = BashOperator(
    task_id='start_task',
    bash_command="echo 5",
    xcom_push=True,
    dag=dag)

branch_op = BranchPythonOperator(
    task_id='branch_task',
    provide_context=True,
    python_callable=branch_func,
    dag=dag)

continue_op = DummyOperator(task_id='continue_task', dag=dag)
stop_op = DummyOperator(task_id='stop_task', dag=dag)

start_op >> branch_op >> [continue_op, stop_op]
```

3.8.2.8 SubDAGs

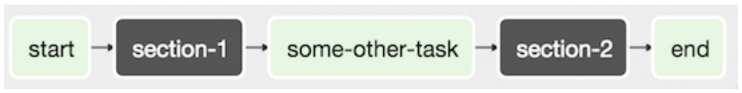
SubDAGs are perfect for repeating patterns. Defining a function that returns a DAG object is a nice design pattern when using Airflow.

Airbnb uses the *stage-check-exchange* pattern when loading data. Data is staged in a temporary table, after which data quality checks are performed against that table. Once the checks all pass the partition is moved into the production table.

As another example, consider the following DAG:



We can combine all of the parallel `task-*` operators into a single SubDAG, so that the resulting DAG resembles the following:



Note that SubDAG operators should contain a factory method that returns a DAG object. This will prevent the SubDAG from being treated like a separate DAG in the main UI. For example:

```
#dags/subdag.py
from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator


# Dag is returned by a factory method
def sub_dag(parent_dag_name, child_dag_name, start_date, schedule_interval):
  dag = DAG(
    '%s.%s' % (parent_dag_name, child_dag_name),
    schedule_interval=schedule_interval,
    start_date=start_date,
  )

  dummy_operator = DummyOperator(
    task_id='dummy_task',
    dag=dag,
  )

  return dag
```

This SubDAG can then be referenced in your main DAG file:

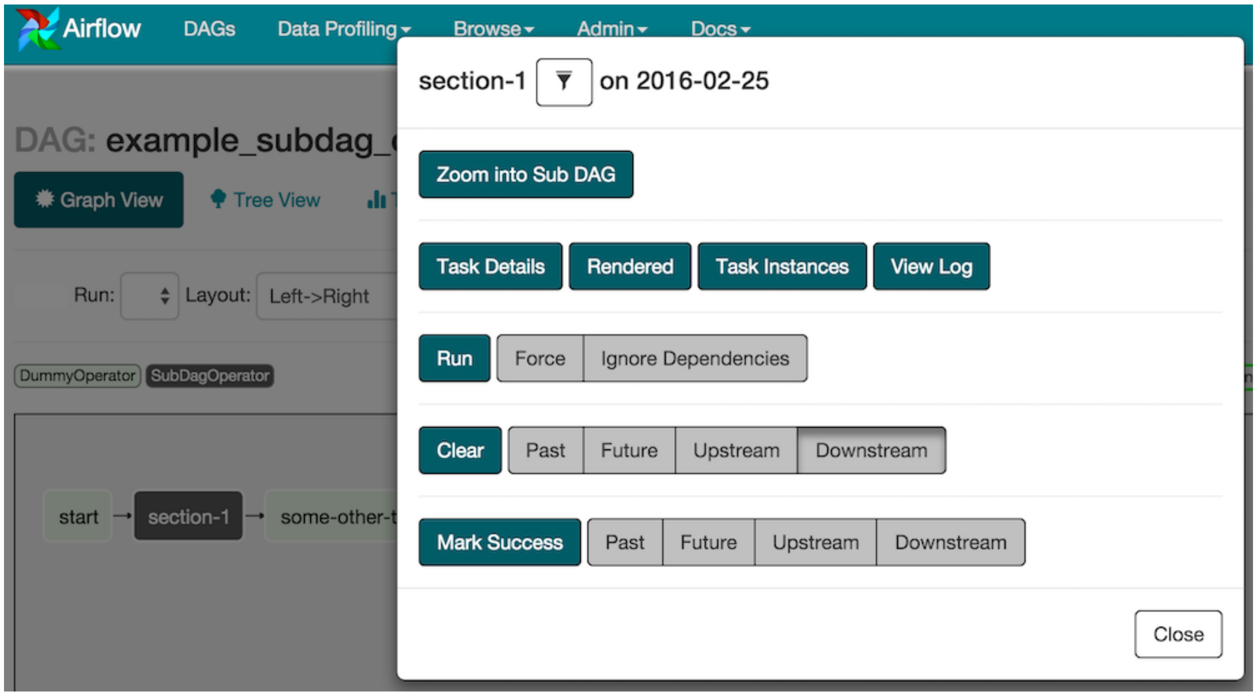
```
# main_dag.py
from datetime import datetime, timedelta
from airflow.models import DAG
from airflow.operators.subdag_operator import SubDagOperator
from dags.subdag import sub_dag


PARENT_DAG_NAME = 'parent_dag'
CHILD_DAG_NAME = 'child_dag'

main_dag = DAG(
  dag_id=PARENT_DAG_NAME,
  schedule_interval=timedelta(hours=1),
  start_date=datetime(2016, 1, 1)
)
```

(continues on next page)

(continued from previous page)

```
sub_dag = SubDagOperator(
  subdag=sub_dag(PARENT_DAG_NAME, CHILD_DAG_NAME, main_dag.start_date,
                 main_dag.schedule_interval),
  task_id=CHILD_DAG_NAME,
  dag=main_dag,
)
```

You can zoom into a SubDagOperator from the graph view of the main DAG to show the tasks contained within the SubDAG:



Some other tips when using SubDAGs:

- by convention, a SubDAG's `dag_id` should be prefixed by its parent and a dot. As in `parent.child`
- share arguments between the main DAG and the SubDAG by passing arguments to the SubDAG operator (as demonstrated above)
- SubDAGs must have a schedule and be enabled. If the SubDAG's schedule is set to `None` or `@once`, the SubDAG will succeed without having done anything
- clearing a SubDagOperator also clears the state of the tasks within
- marking success on a SubDagOperator does not affect the state of the tasks within
- refrain from using `depends_on_past=True` in tasks within the SubDAG as this can be confusing
- it is possible to specify an executor for the SubDAG. It is common to use the SequentialExecutor if you want to run the SubDAG in-process and effectively limit its parallelism to one. Using LocalExecutor can be problematic as it may over-subscribe your worker, running multiple tasks in a single slot

See `airflow/example_dags` for a demonstration.

Note that airflow pool is not honored by SubDagOperator. Hence resources could be consumed by SubdagOperators.

3.8.2.9 SLAs

Service Level Agreements, or time by which a task or DAG should have succeeded, can be set at a task level as a `timedelta`. If one or many instances have not succeeded by that time, an alert email is sent detailing the list of tasks that missed their SLA. The event is also recorded in the database and made available in the web UI under `Browse->SLA Misses` where events can be analyzed and documented.

SLAs can be configured for scheduled tasks by using the *sla* parameter. In addition to sending alerts to the addresses specified in a task's *email* parameter, the *sla_miss_callback* specifies an additional *Callable* object to be invoked when the SLA is not met.

Email Configuration

You can configure the email that is being sent in your `airflow.cfg` by setting a `subject_template` and/or a `html_content_template` in the `email` section.

```
[email]

email_backend = airflow.utils.email.send_email_smtp

subject_template = /path/to/my_subject_template_file
html_content_template = /path/to/my_html_content_template_file
```

To access the task's information you use Jinja Templating in your template files.

For example a `html_content_template` file could look like this:

```
Try {{try_number}} out of {{max_tries + 1}}<br>
Exception:<br>{{exception_html}}<br>
Log: <a href="{{ti.log_url}}">Link</a><br>
Host: {{ti.hostname}}<br>
Log file: {{ti.log_filepath}}<br>
Mark success: <a href="{{ti.mark_success_url}}">Link</a><br>
```

3.8.2.10 Trigger Rules

Though the normal workflow behavior is to trigger tasks when all their directly upstream tasks have succeeded, Airflow allows for more complex dependency settings.

All operators have a `trigger_rule` argument which defines the rule by which the generated task get triggered. The default value for `trigger_rule` is `all_success` and can be defined as "trigger this task when all directly upstream tasks have succeeded". All other rules described here are based on direct parent tasks and are values that can be passed to any operator while creating tasks:

- `all_success`: (default) all parents have succeeded
- `all_failed`: all parents are in a `failed` or `upstream_failed` state
- `all_done`: all parents are done with their execution
- `one_failed`: fires as soon as at least one parent has failed, it does not wait for all parents to be done
- `one_success`: fires as soon as at least one parent succeeds, it does not wait for all parents to be done
- `none_failed`: all parents have not failed (`failed` or `upstream_failed`) i.e. all parents have succeeded or been skipped
- `none_skipped`: no parent is in a `skipped` state, i.e. all parents are in a `success`, `failed`, or `upstream_failed` state

- dummy: dependencies are just for show, trigger at will

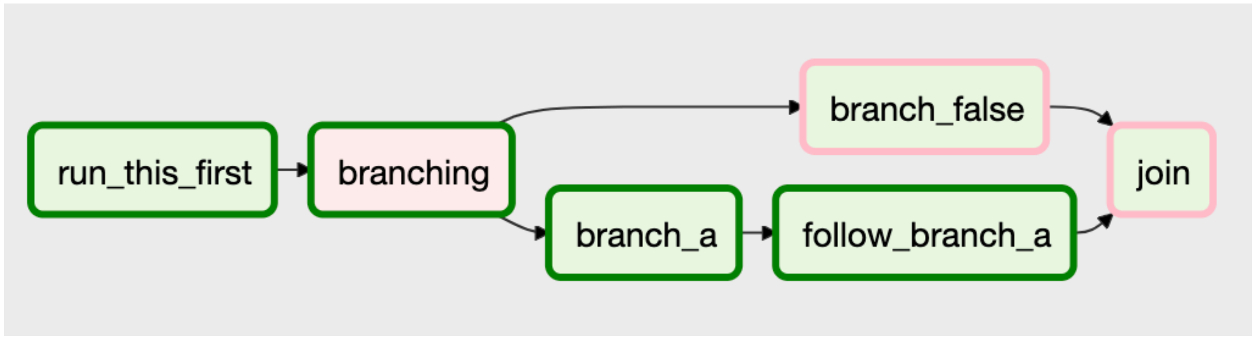
Note that these can be used in conjunction with depends_on_past (boolean) that, when set to True, keeps a task from getting triggered if the previous schedule for the task hasn't succeeded.

One must be aware of the interaction between trigger rules and skipped tasks in schedule level. Skipped tasks will cascade through trigger rules all_success and all_failed but not all_done, one_failed, one_success, none_failed, none_skipped and dummy.

For example, consider the following DAG:

```
#dags/branch_without_trigger.py
import datetime as dt

from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import BranchPythonOperator

dag = DAG(
    dag_id='branch_without_trigger',
    schedule_interval='@once',
    start_date=dt.datetime(2019, 2, 28)
)

run_this_first = DummyOperator(task_id='run_this_first', dag=dag)
branching = BranchPythonOperator(
    task_id='branching', dag=dag,
    python_callable=lambda: 'branch_a'
)

branch_a = DummyOperator(task_id='branch_a', dag=dag)
follow_branch_a = DummyOperator(task_id='follow_branch_a', dag=dag)

branch_false = DummyOperator(task_id='branch_false', dag=dag)

join = DummyOperator(task_id='join', dag=dag)

run_this_first >> branching
branching >> branch_a >> follow_branch_a >> join
branching >> branch_false >> join
```

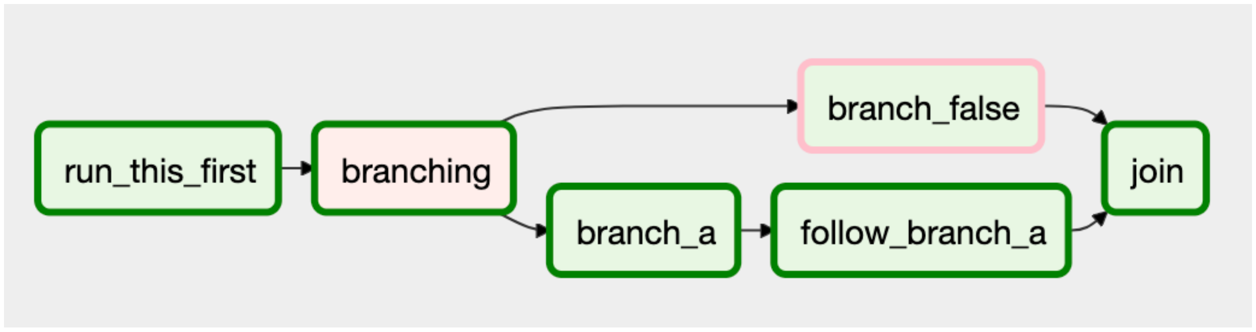
In the case of this DAG, join is downstream of follow_branch_a and branch_false. The join task will show up as skipped because its trigger_rule is set to all_success by default and skipped tasks will cascade through all_success.



By setting trigger_rule to none_failed in join task,

```
#dags/branch_with_trigger.py
...
join = DummyOperator(task_id='join', dag=dag, trigger_rule='none_failed')
...
```

The `join` task will be triggered as soon as `branch_false` has been skipped (a valid completion state) and `follow_branch_a` has succeeded. Because skipped tasks **will not** cascade through `none_failed`.



3.8.2.11 Latest Run Only

Standard workflow behavior involves running a series of tasks for a particular date/time range. Some workflows, however, perform tasks that are independent of run time but need to be run on a schedule, much like a standard cron job. In these cases, backfills or running jobs missed during a pause just wastes CPU cycles.

For situations like this, you can use the `LatestOnlyOperator` to skip tasks that are not being run during the most recent scheduled run for a DAG. The `LatestOnlyOperator` skips all downstream tasks, if the time right now is not between its `execution_time` and the next scheduled `execution_time`.

For example, consider the following DAG:

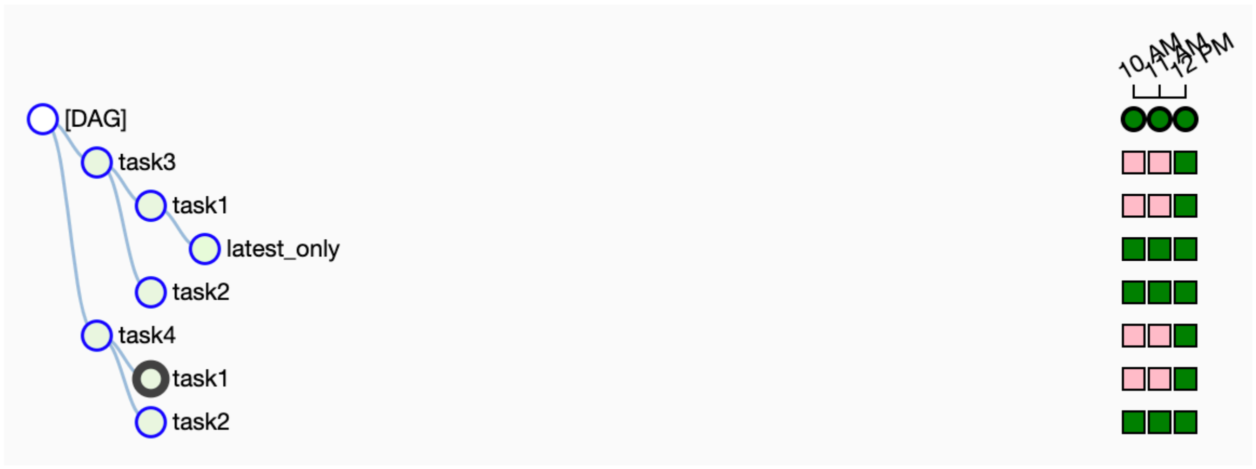
```
#dags/latest_only_with_trigger.py
import datetime as dt

from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.latest_only_operator import LatestOnlyOperator
from airflow.utils.trigger_rule import TriggerRule


dag = DAG(
    dag_id='latest_only_with_trigger',
    schedule_interval=dt.timedelta(hours=1),
    start_date=dt.datetime(2019, 2, 28),
)

latest_only = LatestOnlyOperator(task_id='latest_only', dag=dag)

task1 = DummyOperator(task_id='task1', dag=dag)
task1.set_upstream(latest_only)

task2 = DummyOperator(task_id='task2', dag=dag)

task3 = DummyOperator(task_id='task3', dag=dag)
task3.set_upstream([task1, task2])
```

(continues on next page)

(continued from previous page)

```
task4 = DummyOperator(task_id='task4', dag=dag,
                      trigger_rule=TriggerRule.ALL_DONE)
task4.set_upstream([task1, task2])
```

In the case of this DAG, the `latest_only` task will show up as skipped for all runs except the latest run. `task1` is directly downstream of `latest_only` and will also skip for all runs except the latest. `task2` is entirely independent of `latest_only` and will run in all scheduled periods. `task3` is downstream of `task1` and `task2` and because of the default `trigger_rule` being `all_success` will receive a cascaded skip from `task1`. `task4` is downstream of `task1` and `task2`. It will be first skipped directly by `LatestOnlyOperator`, even its `trigger_rule` is set to `all_done`.



3.8.2.12 Zombies & Undeads

Task instances die all the time, usually as part of their normal life cycle, but sometimes unexpectedly.

Zombie tasks are characterized by the absence of an heartbeat (emitted by the job periodically) and a `running` status in the database. They can occur when a worker node can't reach the database, when Airflow processes are killed externally, or when a node gets rebooted for instance. Zombie killing is performed periodically by the scheduler's process.

Undead processes are characterized by the existence of a process and a matching heartbeat, but Airflow isn't aware of this task as `running` in the database. This mismatch typically occurs as the state of the database is altered, most likely by deleting rows in the "Task Instances" view in the UI. Tasks are instructed to verify their state as part of the heartbeat routine, and terminate themselves upon figuring out that they are in this "undead" state.

3.8.2.13 Cluster Policy

Your local Airflow settings file can define a `policy` function that has the ability to mutate task attributes based on other task or DAG attributes. It receives a single argument as a reference to task objects, and is expected to alter its attributes.

For example, this function could apply a specific queue property when using a specific operator, or enforce a task timeout policy, making sure that no tasks run for more than 48 hours. Here's an example of what this may look like inside your `airflow_settings.py`:

```
def policy(task):
    if task.__class__.__name__ == 'HivePartitionSensor':
        task.queue = "sensor_queue"
    if task.timeout > timedelta(hours=48):
        task.timeout = timedelta(hours=48)
```

3.8.2.14 Documentation & Notes

It's possible to add documentation or notes to your DAGs & task objects that become visible in the web interface ("Graph View" for DAGs, "Task Details" for tasks). There are a set of special task attributes that get rendered as rich content if defined:

attribute	rendered to
doc	monospace
doc_json	json
doc_yaml	yaml
doc_md	markdown
doc_rst	reStructuredText

Please note that for DAGs, doc_md is the only attribute interpreted.

This is especially useful if your tasks are built dynamically from configuration files, it allows you to expose the configuration that led to the related tasks in Airflow.

```
"""
### My great DAG
"""

dag = DAG('my_dag', default_args=default_args)
dag.doc_md = __doc__

t = BashOperator("foo", dag=dag)
t.doc_md = """\
#Title"
Here's a [url](www.airbnb.com)
"""
```

This content will get rendered as markdown respectively in the "Graph View" and "Task Details" pages.

3.8.2.15 Jinja Templating

Airflow leverages the power of Jinja Templating and this can be a powerful tool to use in combination with macros (see the *Macros reference* section).

For example, say you want to pass the execution date as an environment variable to a Bash script using the `BashOperator`.

```
# The execution date as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id='test_env',
    bash_command='/tmp/test.sh ',
    dag=dag,
    env={'EXECUTION_DATE': date})
```

Here, `{{ ds }}` is a macro, and because the `env` parameter of the `BashOperator` is templated with Jinja, the execution date will be available as an environment variable named `EXECUTION_DATE` in your Bash script.

You can use Jinja templating with every parameter that is marked as "templated" in the documentation. Template substitution occurs just before the pre_execute function of your operator is called.

3.8.3 Packaged DAGs

While often you will specify DAGs in a single `.py` file it might sometimes be required to combine a DAG and its dependencies. For example, you might want to combine several DAGs together to version them together or you might want to manage them together or you might need an extra module that is not available by default on the system you are running Airflow on. To allow this you can create a zip file that contains the DAG(s) in the root of the zip file and have the extra modules unpacked in directories.

For instance you can create a zip file that looks like this:

```
my_dag1.py
my_dag2.py
package1/__init__.py
package1/functions.py
```

Airflow will scan the zip file and try to load `my_dag1.py` and `my_dag2.py`. It will not go into subdirectories as these are considered to be potential packages.

In case you would like to add module dependencies to your DAG you basically would do the same, but then it is more suitable to use a virtualenv and pip.

```
virtualenv zip_dag
source zip_dag/bin/activate

mkdir zip_dag_contents
cd zip_dag_contents

pip install --install-option="--install-lib=$PWD" my_useful_package
cp ~/my_dag.py .

zip -r zip_dag.zip *
```

Note: the zip file will be inserted at the beginning of module search list (sys.path) and as such it will be available to any other code that resides within the same interpreter.

Note: packaged dags cannot be used with pickling turned on.

Note: packaged dags cannot contain dynamic libraries (eg. libz.so) these need to be available on the system if a module needs those. In other words only pure python modules can be packaged.

3.8.4 .airflowignore

A `.airflowignore` file specifies the directories or files in `DAG_FOLDER` that Airflow should intentionally ignore. Each line in `.airflowignore` specifies a regular expression pattern, and directories or files whose names (not DAG id) match any of the patterns would be ignored (under the hood, `re.findall()` is used to match the pattern). Overall it works like a `.gitignore` file. Use the `#` character to indicate a comment; all characters on a line following a `#` will be ignored.

`.airflowignore` file should be put in your `DAG_FOLDER`. For example, you can prepare a `.airflowignore` file with contents

```
project_a
tenant_[\d]
```

Then files like "project_a_dag_1.py", "TESTING_project_a.py", "tenant_1.py", "project_a/dag_1.py", and "tenant_1/dag_1.py" in your `DAG_FOLDER` would be ignored (If a directory's name matches any of the patterns, this directory and all its subfolders would not be scanned by Airflow at all. This improves efficiency of DAG finding).

The scope of a `.airflowignore` file is the directory it is in plus all its subfolders. You can also prepare `.airflowignore` file for a subfolder in `DAG_FOLDER` and it would only be applicable for that subfolder.

3.9 Command Line Interface

Airflow has a very rich command line interface that allows for many types of operation on a DAG, starting services, and supporting development and testing.

Content

- *Positional Arguments*
- *Sub-commands:*
  - *backfill*
  - *list_dag_runs*
  - *list_tasks*
  - *clear*
  - *pause*
  - *unpause*
  - *trigger_dag*
  - *delete_dag*
  - *pool*
  - *variables*
  - *kerberos*
  - *render*
  - *run*
  - *initdb*
  - *list_dags*
  - *dag_state*
  - *task_failed_deps*
  - *task_state*
  - *serve_logs*
  - *test*
  - *webserver*
  - *resetdb*

- *upgradedb*
- *scheduler*
- *worker*
- *flower*
- *version*
- *connections*
- *create_user*
- *delete_user*
- *list_users*
- *sync_perm*
- *next_execution*
- *rotate_fernet_key*

```
usage: airflow [-h]
               {backfill,list_dag_runs,list_tasks,clear,pause,unpause,trigger_dag,
↪delete_dag,pool,variables,kerberos,render,run,initdb,list_dags,dag_state,task_
↪failed_deps,task_state,serve_logs,test,webserver,resetdb,upgradedb,scheduler,worker,
↪flower,version,connections,create_user,delete_user,list_users,sync_perm,next_
↪execution,rotate_fernet_key}
               ...
```

3.9.1 Positional Arguments

subcommand	Possible choices: backfill, list_dag_runs, list_tasks, clear, pause, unpause, trigger_dag, delete_dag, pool, variables, kerberos, render, run, initdb, list_dags, dag_state, task_failed_deps, task_state, serve_logs, test, webserver, resetdb, upgradedb, scheduler, worker, flower, version, connections, create_user, delete_user, list_users, sync_perm, next_execution, rotate_fernet_key sub-command help

3.9.2 Sub-commands:

3.9.2.1 backfill

Run subsections of a DAG for a specified date range. If reset_dag_run option is used, backfill will first prompt users whether airflow should clear all the previous dag_run and task_instances within the backfill date range. If rerun_failed_tasks is used, backfill will auto re-run the previous failed task instances within the backfill date range.

```
airflow backfill [-h] [-t TASK_REGEX] [-s START_DATE] [-e END_DATE] [-m] [-l]
                 [-x] [-i] [-I] [-sd SUBDIR] [--pool POOL]
                 [--delay_on_limit DELAY_ON_LIMIT] [-dr] [-v] [-c CONF]
                 [--reset_dagruns] [--rerun_failed_tasks] [-B]
                 dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-t, --task_regex	The regex to filter specific task_ids to backfill (optional)
-s, --start_date	Override start_date YYYY-MM-DD
-e, --end_date	Override end_date YYYY-MM-DD
-m, --mark_success	Mark jobs as succeeded without running them Default: False
-l, --local	Run the task using the LocalExecutor Default: False
-x, --donot_pickle	Do not attempt to pickle the DAG object to send over to the workers, just tell the workers to run their version of the code. Default: False
-i, --ignore_dependencies	Skip upstream tasks, run only the tasks matching the regexp. Only works in conjunction with task_regex Default: False
-I, --ignore_first_depends_on_past	Ignores depends_on_past dependencies for the first set of tasks only (subsequent executions in the backfill DO respect depends_on_past). Default: False
-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"
--pool	Resource pool to use
--delay_on_limit	Amount of time in seconds to wait when the limit on maximum active dag runs (max_active_runs) has been reached before trying to execute a dag run again. Default: 1.0
-dr, --dry_run	Perform a dry run Default: False
-v, --verbose	Make logging output more verbose Default: False
-c, --conf	JSON string that gets pickled into the DagRun's conf attribute
--reset_dagruns	if set, the backfill will delete existing backfill-related DAG runs and start anew with fresh, running DAG runs Default: False
--rerun_failed_tasks	if set, the backfill will auto-rerun all the failed tasks for the backfill date range instead of throwing exceptions Default: False

-B, --run_backwards	if set, the backfill will run tasks from the most recent day first. if there are tasks that depend_on_past this option will throw an exception Default: False

3.9.2.2 list_dag_runs

List dag runs given a DAG id. If state option is given, it will onlysearch for all the dagruns with the given state. If no_backfill option is given, it will filter outall backfill dagruns for given dag id.

```
airflow list_dag_runs [-h] [--no_backfill] [--state STATE] dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

--no_backfill	filter all the backfill dagruns given the dag id Default: False
--state	Only list the dag runs corresponding to the state

3.9.2.3 list_tasks

List the tasks within a DAG

```
airflow list_tasks [-h] [-t] [-sd SUBDIR] dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-t, --tree	Tree view Default: False
-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.4 clear

Clear a set of task instance, as if they never ran

```
airflow clear [-h] [-t TASK_REGEX] [-s START_DATE] [-e END_DATE] [-sd SUBDIR]
              [-u] [-d] [-c] [-f] [-r] [-x] [-xp] [-dx]
              dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-t, --task_regex	The regex to filter specific task_ids to backfill (optional)
-s, --start_date	Override start_date YYYY-MM-DD
-e, --end_date	Override end_date YYYY-MM-DD
-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"
-u, --upstream	Include upstream tasks Default: False
-d, --downstream	Include downstream tasks Default: False
-c, --no_confirm	Do not request confirmation Default: False
-f, --only_failed	Only failed jobs Default: False
-r, --only_running	Only running jobs Default: False
-x, --exclude_subdags	Exclude subdags Default: False
-xp, --exclude_parentdag	Exclude ParentDAGS if the task cleared is a part of a SubDAG Default: False
-dx, --dag_regex	Search dag_id as regex instead of exact string Default: False

3.9.2.5 pause

Pause a DAG

```
airflow pause [-h] [-sd SUBDIR] dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.6 unpause

Resume a paused DAG

```
airflow unpause [-h] [-sd SUBDIR] dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.7 trigger_dag

Trigger a DAG run

```
airflow trigger_dag [-h] [-sd SUBDIR] [-r RUN_ID] [-c CONF] [-e EXEC_DATE]
                    dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"
-r, --run_id	Helps to identify this run

-c, --conf	JSON string that gets pickled into the DagRun's conf attribute
-e, --exec_date	The execution date of the DAG

3.9.2.8 delete_dag

Delete all DB records related to the specified DAG

```
airflow delete_dag [-h] [-y] dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-y, --yes	Do not prompt to confirm reset. Use with care! Default: False

3.9.2.9 pool

CRUD operations on pools

```
airflow pool [-h] [-s NAME SLOT_COUNT POOL_DESCRIPTION] [-g NAME] [-x NAME]
             [-i FILEPATH] [-e FILEPATH]
```

Named Arguments

-s, --set	Set pool slot count and description, respectively
-g, --get	Get pool info
-x, --delete	Delete a pool
-i, --import	Import pool from JSON file
-e, --export	Export pool to JSON file

3.9.2.10 variables

CRUD operations on variables

```
airflow variables [-h] [-s KEY VAL] [-g KEY] [-j] [-d VAL] [-i FILEPATH]
                  [-e FILEPATH] [-x KEY]
```

Named Arguments

-s, --set	Set a variable
-g, --get	Get value of a variable

-j, --json	Deserialize JSON variable Default: False
-d, --default	Default value returned if variable does not exist
-i, --import	Import variables from JSON file
-e, --export	Export variables to JSON file
-x, --delete	Delete a variable

3.9.2.11 kerberos

Start a kerberos ticket renewer

```
airflow kerberos [-h] [-kt [KEYTAB]] [--pid [PID]] [-D] [--stdout STDOUT]
                 [--stderr STDERR] [-l LOG_FILE]
                 [principal]
```

Positional Arguments

principal	kerberos principal

Named Arguments

-kt, --keytab	keytab Default: "airflow.keytab"
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-l, --log-file	Location of the log file

3.9.2.12 render

Render a task instance's template(s)

```
airflow render [-h] [-sd SUBDIR] dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.13 run

Run a single task instance

```
airflow run [-h] [-sd SUBDIR] [-m] [-f] [--pool POOL] [--cfg_path CFG_PATH]
            [-l] [-A] [-i] [-I] [--ship_dag] [-p PICKLE] [-int]
            dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"
-m, --mark_success	Mark jobs as succeeded without running them Default: False
-f, --force	Ignore previous task instance state, rerun regardless if task already succeeded/failed Default: False
--pool	Resource pool to use
--cfg_path	Path to config file to use instead of airflow.cfg
-l, --local	Run the task using the LocalExecutor Default: False
-A, --ignore_all_dependencies	Ignores all non-critical dependencies, including ignore_ti_state and ignore_task_deps Default: False
-i, --ignore_dependencies	Ignore task-specific dependencies, e.g. upstream, depends_on_past, and retry delay dependencies Default: False
-I, --ignore_depends_on_past	Ignore depends_on_past dependencies (but respect upstream dependencies) Default: False

--ship_dag	Pickles (serializes) the DAG and ships it to the worker Default: False
-p, --pickle	Serialized pickle object of the entire dag (used internally)
-int, --interactive	Do not capture standard output and error streams (useful for interactive debugging) Default: False

3.9.2.14 initdb

Initialize the metadata database

```
airflow initdb [-h]
```

3.9.2.15 list_dags

List all the DAGs

```
airflow list_dags [-h] [-sd SUBDIR] [-r]
```

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"
-r, --report	Show DagBag loading report Default: False

3.9.2.16 dag_state

Get the status of a dag run

```
airflow dag_state [-h] [-sd SUBDIR] dag_id execution_date
```

Positional Arguments

dag_id	The id of the dag
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.17 task_failed_deps

Returns the unmet dependencies for a task instance from the perspective of the scheduler. In other words, why a task instance doesn't get scheduled and then queued by the scheduler, and then run by an executor).

```
airflow task_failed_deps [-h] [-sd SUBDIR] dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.18 task_state

Get the status of a task instance

```
airflow task_state [-h] [-sd SUBDIR] dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.19 serve_logs

Serve logs generate by worker

```
airflow serve_logs [-h]
```

3.9.2.20 test

Test a task instance. This will run a task without checking for dependencies or recording its state in the database.

```
airflow test [-h] [-sd SUBDIR] [-dr] [-tp TASK_PARAMS] [-pm]
             dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"
-dr, --dry_run	Perform a dry run Default: False
-tp, --task_params	Sends a JSON params dict to the task
-pm, --post_mortem	Open debugger on uncaught exception Default: False

3.9.2.21 webserver

Start a Airflow webserver instance

```
airflow webserver [-h] [-p PORT] [-w WORKERS]
                  [-k {sync,eventlet,gevent,tornado}] [-t WORKER_TIMEOUT]
                  [-hn HOSTNAME] [--pid [PID]] [-D] [--stdout STDOUT]
                  [--stderr STDERR] [-A ACCESS_LOGFILE] [-E ERROR_LOGFILE]
                  [-l LOG_FILE] [--ssl_cert SSL_CERT] [--ssl_key SSL_KEY] [-d]
```

Named Arguments

-p, --port	The port on which to run the server Default: 8080
-w, --workers	Number of workers to run the webserver on Default: 4
-k, --workerclass	Possible choices: sync, eventlet, gevent, tornado The worker class to use for Gunicorn Default: "sync"

-t, --worker_timeout	The timeout for waiting on webserver workers Default: 120
-hn, --hostname	Set the hostname on which to run the web server Default: “0.0.0.0”
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-A, --access_logfile	The logfile to store the webserver access log. Use ‘-‘ to print to stderr. Default: “-“
-E, --error_logfile	The logfile to store the webserver error log. Use ‘-‘ to print to stderr. Default: “-“
-l, --log-file	Location of the log file
--ssl_cert	Path to the SSL certificate for the webserver
--ssl_key	Path to the key to use with the SSL certificate
-d, --debug	Use the server that ships with Flask in debug mode Default: False

3.9.2.22 resetdb

Burn down and rebuild the metadata database

```
airflow resetdb [-h] [-y]
```

Named Arguments

-y, --yes	Do not prompt to confirm reset. Use with care! Default: False

3.9.2.23 upgradedb

Upgrade the metadata database to latest version

```
airflow upgradedb [-h]
```

3.9.2.24 scheduler

Start a scheduler instance

```
airflow scheduler [-h] [-d DAG_ID] [-sd SUBDIR] [-r RUN_DURATION]
                  [-n NUM_RUNS] [-p] [--pid [PID]] [-D] [--stdout STDOUT]
                  [--stderr STDERR] [-l LOG_FILE]
```

Named Arguments

-d, --dag_id	The id of the dag to run
-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"
-r, --run-duration	Set number of seconds to execute before exiting
-n, --num_runs	Set the number of runs to execute before exiting Default: -1
-p, --do_pickle	Attempt to pickle the DAG object to send over to the workers, instead of letting workers run their version of the code. Default: False
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-l, --log-file	Location of the log file

3.9.2.25 worker

Start a Celery worker node

```
airflow worker [-h] [-p] [-q QUEUES] [-c CONCURRENCY] [-cn CELERY_HOSTNAME]
               [--pid [PID]] [-D] [--stdout STDOUT] [--stderr STDERR]
               [-l LOG_FILE] [-a AUTOSCALE]
```

Named Arguments

-p, --do_pickle	Attempt to pickle the DAG object to send over to the workers, instead of letting workers run their version of the code. Default: False
-q, --queues	Comma delimited list of queues to serve Default: "default"
-c, --concurrency	The number of worker processes Default: 16
-cn, --celery_hostname	Set the hostname of celery worker if you have multiple workers on a single machine.
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground Default: False
--stdout	Redirect stdout to this file

--stderr	Redirect stderr to this file
-l, --log-file	Location of the log file
-a, --autoscale	Minimum and Maximum number of worker to autoscale

3.9.2.26 flower

Start a Celery Flower

```
airflow flower [-h] [-hn HOSTNAME] [-p PORT] [-fc FLOWER_CONF] [-u URL_PREFIX]
               [-ba BASIC_AUTH] [-a BROKER_API] [--pid [PID]] [-D]
               [--stdout STDOUT] [--stderr STDERR] [-l LOG_FILE]
```

Named Arguments

-hn, --hostname	Set the hostname on which to run the server Default: “0.0.0.0”
-p, --port	The port on which to run the server Default: 5555
-fc, --flower_conf	Configuration file for flower
-u, --url_prefix	URL prefix for Flower
-ba, --basic_auth	Securing Flower with Basic Authentication. Accepts user:password pairs separated by a comma. Example: flower_basic_auth = user1:password1,user2:password2
-a, --broker_api	Broker api
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-l, --log-file	Location of the log file

3.9.2.27 version

Show the version

```
airflow version [-h]
```

3.9.2.28 connections

List/Add/Delete connections

```
airflow connections [-h] [-l] [-a] [-d] [--conn_id CONN_ID]
                    [--conn_uri CONN_URI] [--conn_extra CONN_EXTRA]
                    [--conn_type CONN_TYPE] [--conn_host CONN_HOST]
                    [--conn_login CONN_LOGIN] [--conn_password CONN_PASSWORD]
                    [--conn_schema CONN_SCHEMA] [--conn_port CONN_PORT]
```

Named Arguments

-l, --list	List all connections Default: False
-a, --add	Add a connection Default: False
-d, --delete	Delete a connection Default: False
--conn_id	Connection id, required to add/delete a connection
--conn_uri	Connection URI, required to add a connection without conn_type
--conn_extra	Connection *Extra* field, optional when adding a connection
--conn_type	Connection type, required to add a connection without conn_uri
--conn_host	Connection host, optional when adding a connection
--conn_login	Connection login, optional when adding a connection
--conn_password	Connection password, optional when adding a connection
--conn_schema	Connection schema, optional when adding a connection
--conn_port	Connection port, optional when adding a connection

3.9.2.29 create_user

Create an account for the Web UI (FAB-based)

```
airflow create_user [-h] [-r ROLE] [-u USERNAME] [-e EMAIL] [-f FIRSTNAME]
                    [-l LASTNAME] [-p PASSWORD] [--use_random_password]
```

Named Arguments

-r, --role	Role of the user. Existing roles include Admin, User, Op, Viewer, and Public
-u, --username	Username of the user
-e, --email	Email of the user
-f, --firstname	First name of the user
-l, --lastname	Last name of the user
-p, --password	Password of the user
--use_random_password	Do not prompt for password. Use random string instead Default: False

3.9.2.30 delete_user

Delete an account for the Web UI

```
airflow delete_user [-h] [-u USERNAME]
```

Named Arguments

-u, --username	Username of the user

3.9.2.31 list_users

List accounts for the Web UI

```
airflow list_users [-h]
```

3.9.2.32 sync_perm

Update existing role's permissions.

```
airflow sync_perm [-h]
```

3.9.2.33 next_execution

Get the next execution datetime of a DAG.

```
airflow next_execution [-h] [-sd SUBDIR] dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag. Defaults to '[AIRFLOW_HOME]/dags' where [AIRFLOW_HOME] is the value you set for 'AIRFLOW_HOME' config you set in 'airflow.cfg' Default: "[AIRFLOW_HOME]/dags"

3.9.2.34 rotate_fernet_key

Rotate all encrypted connection credentials and variables; see https://airflow.readthedocs.io/en/stable/howto/secure-connections.html#rotating-encryption-keys.

```
airflow rotate_fernet_key [-h]
```

3.10 Scheduling & Triggers

The Airflow scheduler monitors all tasks and all DAGs, and triggers the task instances whose dependencies have been met. Behind the scenes, it spins up a subprocess, which monitors and stays in sync with a folder for all DAG objects it may contain, and periodically (every minute or so) collects DAG parsing results and inspects active tasks to see whether they can be triggered.

The Airflow scheduler is designed to run as a persistent service in an Airflow production environment. To kick it off, all you need to do is execute `airflow scheduler`. It will use the configuration specified in `airflow.cfg`.

Note that if you run a DAG on a `schedule_interval` of one day, the run stamped `2016-01-01` will be triggered soon after `2016-01-01T23:59`. In other words, the job instance is started once the period it covers has ended.

Let's Repeat That The scheduler runs your job one `schedule_interval` AFTER the start date, at the END of the period.

The scheduler starts an instance of the executor specified in the your `airflow.cfg`. If it happens to be the `airflow.contrib.executors.local_executor.LocalExecutor`, tasks will be executed as subprocesses; in the case of *`airflow.executors.celery_executor.CeleryExecutor`*, and *`airflow.executors.dask_executor.DaskExecutor`* tasks are executed remotely.

To start a scheduler, simply run the command:

```
airflow scheduler
```

3.10.1 DAG Runs

A DAG Run is an object representing an instantiation of the DAG in time.

Each DAG may or may not have a schedule, which informs how `DAG Runs` are created. `schedule_interval` is defined as a DAG arguments, and receives preferably a cron expression as a `str`, or a `datetime.timedelta` object. Alternatively, you can also use one of these cron "preset":

preset	meaning	cron
`None`	Don't schedule, use for exclusively "externally triggered" DAGs	
`@once`	Schedule once and only once	
`@hourly`	Run once an hour at the beginning of the hour	`0 * * * *`
`@daily`	Run once a day at midnight	`0 0 * * *`
`@weekly`	Run once a week at midnight on Sunday morning	`0 0 * * 0`
`@monthly`	Run once a month at midnight of the first day of the month	`0 0 1 * *`
`@yearly`	Run once a year at midnight of January 1	`0 0 1 1 *`

Note: Use `schedule_interval=None` and not `schedule_interval='None'` when you don't want to schedule your DAG.

Your DAG will be instantiated for each schedule, while creating a `DAG Run` entry for each schedule.

DAG runs have a state associated to them (running, failed, success) and informs the scheduler on which set of schedules should be evaluated for task submissions. Without the metadata at the DAG run level, the Airflow scheduler would have much more work to do in order to figure out what tasks should be triggered and come to a crawl. It might also create undesired processing when changing the shape of your DAG, by say adding in new tasks.

3.10.2 Backfill and Catchup

An Airflow DAG with a `start_date`, possibly an `end_date`, and a `schedule_interval` defines a series of intervals which the scheduler turn into individual Dag Runs and execute. A key capability of Airflow is that these DAG Runs are atomic, idempotent items, and the scheduler, by default, will examine the lifetime of the DAG (from start to end/now, one interval at a time) and kick off a DAG Run for any interval that has not been run (or has been cleared). This concept is called Catchup.

If your DAG is written to handle its own catchup (IE not limited to the interval, but instead to "Now" for instance.), then you will want to turn catchup off (Either on the DAG itself with `dag.catchup = False`) or by default at the configuration file level with `catchup_by_default = False`. What this will do, is to instruct the scheduler to only create a DAG Run for the most current instance of the DAG interval series.

```
"""
Code that goes along with the Airflow tutorial located at:
https://github.com/apache/airflow/blob/master/airflow/example_dags/tutorial.py
"""
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 12, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5)
}

dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial DAG',
    schedule_interval='@daily',
    catchup=False)
```

In the example above, if the DAG is picked up by the scheduler daemon on 2016-01-02 at 6 AM, (or from the command line), a single DAG Run will be created, with an `execution_date` of 2016-01-01, and the next one will be created just after midnight on the morning of 2016-01-03 with an execution date of 2016-01-02.

If the `dag.catchup` value had been True instead, the scheduler would have created a DAG Run for each completed interval between 2015-12-01 and 2016-01-02 (but not yet one for 2016-01-02, as that interval hasn't completed) and the scheduler will execute them sequentially. This behavior is great for atomic datasets that can easily be split into periods. Turning catchup off is great if your DAG Runs perform backfill internally.

3.10.3 External Triggers

Note that `DAG Runs` can also be created manually through the CLI while running an `airflow trigger_dag` command, where you can define a specific `run_id`. The `DAG Runs` created externally to the scheduler get associated to the trigger's timestamp, and will be displayed in the UI alongside scheduled `DAG runs`.

In addition, you can also manually trigger a `DAG Run` using the web UI (tab "DAGs" -> column "Links" -> button "Trigger Dag").

3.10.4 To Keep in Mind

- The first `DAG Run` is created based on the minimum `start_date` for the tasks in your DAG.
- Subsequent `DAG Runs` are created by the scheduler process, based on your DAG's `schedule_interval`, sequentially.
- When clearing a set of tasks' state in hope of getting them to re-run, it is important to keep in mind the `DAG Run`'s state too as it defines whether the scheduler should look into triggering tasks for that run.

Here are some of the ways you can **unblock tasks**:

- From the UI, you can **clear** (as in delete the status of) individual task instances from the task instances dialog, while defining whether you want to includes the past/future and the upstream/downstream dependencies. Note that a confirmation window comes next and allows you to see the set you are about to clear. You can also clear all task instances associated with the dag.
- The CLI command `airflow clear -h` has lots of options when it comes to clearing task instance states, including specifying date ranges, targeting task_ids by specifying a regular expression, flags for including upstream and downstream relatives, and targeting task instances in specific states (`failed`, or `success`)
- Clearing a task instance will no longer delete the task instance record. Instead it updates max_tries and set the current task instance state to be None.
- Marking task instances as failed can be done through the UI. This can be used to stop running task instances.
- Marking task instances as successful can be done through the UI. This is mostly to fix false negatives, or for instance when the fix has been applied outside of Airflow.
- The `airflow backfill` CLI subcommand has a flag to `--mark_success` and allows selecting subsections of the DAG as well as specifying date ranges.

3.11 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 **hooks**, **operators**, **sensors**, **macros**, **executors** and web **views** get integrated to Airflow's main collections and become available for use.

3.11.1 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
- …

3.11.2 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

3.11.3 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 BaseOperator
    operators = []
    # A list of class(es) derived from BaseSensorOperator
    sensors = []
    # A list of class(es) derived from BaseHook
    hooks = []
    # A list of class(es) derived from BaseExecutor
    executors = []
    # 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 function that validate the statsd stat name, apply changes to the stat name␣
↪if necessary and
    # return the transformed stat name.
    #
    # The function should have the following signature:
    # def func_name(stat_name: str) -> str:
    stat_name_handler = None
    # 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 = []
```

You can derive it by inheritance (please refer to the example below). Please note `name` inside this class must be specified.

After the plugin is imported into Airflow, you can invoke it using statement like

```
from airflow.{type, like "operators", "sensors"}.{name specified inside the plugin␣
↪class} import *
```

When you write your own plugins, make sure you understand them well. There are some essential properties for each type of plugin. For example,

- For `Operator` plugin, an `execute` method is compulsory.
- For `Sensor` plugin, a `poke` method returning a Boolean value is compulsory.

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

3.11.4 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_hook import BaseHook
from airflow.models import BaseOperator
from airflow.models.baseoperator import BaseOperatorLink
from airflow.sensors.base_sensor_operator import BaseSensorOperator
from airflow.executors.base_executor import BaseExecutor

# Will show up under airflow.hooks.test_plugin.PluginHook
class PluginHook(BaseHook):
    pass

# Will show up under airflow.operators.test_plugin.PluginOperator
class PluginOperator(BaseOperator):
    pass

# Will show up under airflow.sensors.test_plugin.PluginSensorOperator
class PluginSensorOperator(BaseSensorOperator):
    pass

# Will show up under airflow.executors.test_plugin.PluginExecutor
class PluginExecutor(BaseExecutor):
    pass

# Will show up under airflow.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')
```

(continues on next page)

(continued from previous page)

```
# 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!")

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"}

# Validate the statsd stat name
def stat_name_dummy_handler(stat_name):
    return stat_name

# A global operator extra link that redirect you to
# task logs stored in S3
class S3LogLink(BaseOperatorLink):
    name = 'S3'

    def get_link(self, operator, dttm):
        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=dttm,
        )


# Defining the plugin class
class AirflowTestPlugin(AirflowPlugin):
    name = "test_plugin"
    operators = [PluginOperator]
    sensors = [PluginSensorOperator]
    hooks = [PluginHook]
    executors = [PluginExecutor]
    macros = [plugin_macro]
    flask_blueprints = [bp]
    appbuilder_views = [v_appbuilder_package]
    appbuilder_menu_items = [appbuilder_mitem]
    stat_name_handler = staticmethod(stat_name_dummy_handler)
    global_operator_extra_links = [S3LogLink(),]
```

3.11.5 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.

3.11.6 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. The structure is determined by *airflow.plugins_manager.AirflowPlugin.name* and the class name of the plugin component with the pattern *airflow.{component}.{name}.{component_class_name}*.

```
# my_package/my_plugin.py
from airflow.plugins_manager import AirflowPlugin
from airflow.models import BaseOperator
from airflow.hooks.base_hook import BaseHook

class MyOperator(BaseOperator):
  pass

class MyHook(BaseHook):
  pass

class MyAirflowPlugin(AirflowPlugin):
  name = 'my_namespace'
  operators = [MyOperator]
  hooks = [MyHook]
```

```
from setuptools import setup

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

This will create a hook, and an operator accessible at:

- *airflow.hooks.my_namespace.MyHook*
- *airflow.operators.my_namespace.MyOperator*

3.12 Security

3.12.1 Reporting Vulnerabilities

⚠ Please do not file Jira issues for security vulnerabilities as they are public! ⚠

The Apache Software Foundation takes security issues very seriously. Apache Airflow specifically offers security features and is responsive to issues around its features. If you have any concern around Airflow Security or believe you have uncovered a vulnerability, we suggest that you get in touch via the e-mail address security@apache.org. In the message, try to provide a description of the issue and ideally a way of reproducing it. The security team will get back to you after assessing the description.

Note that this security address should be used only for undisclosed vulnerabilities. Dealing with fixed issues or general questions on how to use the security features should be handled regularly via the user and the dev lists. Please report any security problems to the project security address before disclosing it publicly.

The ASF Security team's page describes how vulnerability reports are handled, and includes PGP keys if you wish to use that.

3.12.2 Web Authentication

By default, Airflow requires users to specify a password prior to login. You can use the following CLI commands to create an account:

```
# create an admin user
airflow users -c --username admin --firstname Peter --lastname Parker --role Admin --
↪email spiderman@superhero.org
```

It is however possible to switch on authentication by either using one of the supplied backends or creating your own.

Be sure to checkout *Experimental Rest API* for securing the API.

Note: Airflow uses the config parser of Python. This config parser interpolates '%'-signs. Make sure escape any `%` signs in your config file (but not environment variables) as `%%`, otherwise Airflow might leak these passwords on a config parser exception to a log.

3.12.2.1 Password

One of the simplest mechanisms for authentication is requiring users to specify a password before logging in.

Please use command line interface `airflow users --create` to create accounts, or do that in the UI.

3.12.2.2 LDAP

To turn on LDAP authentication configure your `airflow.cfg` as follows. Please note that the example uses an encrypted connection to the ldap server as we do not want passwords be readable on the network level.

Additionally, if you are using Active Directory, and are not explicitly specifying an OU that your users are in, you will need to change `search_scope` to "SUBTREE".

Valid search_scope options can be found in the ldap3 Documentation

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.ldap_auth

[ldap]
# set a connection without encryption: uri = ldap://<your.ldap.server>:<port>
uri = ldaps://<your.ldap.server>:<port>
user_filter = objectClass=*
# in case of Active Directory you would use: user_name_attr = sAMAccountName
user_name_attr = uid
# group_member_attr should be set accordingly with *_filter
# eg :
```

(continues on next page)

(continued from previous page)

```
#     group_member_attr = groupMembership
#     superuser_filter = groupMembership=CN=airflow-super-users...
group_member_attr = memberOf
superuser_filter = memberOf=CN=airflow-super-users,OU=Groups,OU=RWC,OU=US,OU=NORAM,
↪DC=example,DC=com
data_profiler_filter = memberOf=CN=airflow-data-profilers,OU=Groups,OU=RWC,OU=US,
↪OU=NORAM,DC=example,DC=com
bind_user = cn=Manager,dc=example,dc=com
bind_password = insecure
basedn = dc=example,dc=com
cacert = /etc/ca/ldap_ca.crt
# Set search_scope to one of them:  BASE, LEVEL , SUBTREE
# Set search_scope to SUBTREE if using Active Directory, and not specifying an␣
↪Organizational Unit
search_scope = LEVEL

# This option tells ldap3 to ignore schemas that are considered malformed. This␣
↪sometimes comes up
# when using hosted ldap services.
ignore_malformed_schema = False
```

The superuser_filter and data_profiler_filter are optional. If defined, these configurations allow you to specify LDAP groups that users must belong to in order to have superuser (admin) and data-profiler permissions. If undefined, all users will be superusers and data profilers.

3.12.2.3 Roll your own

Airflow uses `flask_login` and exposes a set of hooks in the `airflow.default_login` module. You can alter the content and make it part of the `PYTHONPATH` and configure it as a backend in `airflow.cfg`.

```
[webserver]
authenticate = True
auth_backend = mypackage.auth
```

3.12.3 Kerberos

Airflow has initial support for Kerberos. This means that airflow can renew kerberos tickets for itself and store it in the ticket cache. The hooks and dags can make use of ticket to authenticate against kerberized services.

3.12.3.1 Limitations

Please note that at this time, not all hooks have been adjusted to make use of this functionality. Also it does not integrate kerberos into the web interface and you will have to rely on network level security for now to make sure your service remains secure.

Celery integration has not been tried and tested yet. However, if you generate a key tab for every host and launch a ticket renewer next to every worker it will most likely work.

3.12.3.2 Enabling kerberos

Airflow

To enable kerberos you will need to generate a (service) key tab.

```
# in the kadmin.local or kadmin shell, create the airflow principal
kadmin:  addprinc -randkey airflow/fully.qualified.domain.name@YOUR-REALM.COM

# Create the airflow keytab file that will contain the airflow principal
kadmin:  xst -norandkey -k airflow.keytab airflow/fully.qualified.domain.name
```

Now store this file in a location where the airflow user can read it (chmod 600). And then add the following to your `airflow.cfg`

```
[core]
security = kerberos

[kerberos]
keytab = /etc/airflow/airflow.keytab
reinit_frequency = 3600
principal = airflow
```

Launch the ticket renewer by

```
# run ticket renewer
airflow kerberos
```

Hadoop

If want to use impersonation this needs to be enabled in `core-site.xml` of your hadoop config.

```
<property>
  <name>hadoop.proxyuser.airflow.groups</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.users</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.hosts</name>
  <value>*</value>
</property>
```

Of course if you need to tighten your security replace the asterisk with something more appropriate.

3.12.3.3 Using kerberos authentication

The hive hook has been updated to take advantage of kerberos authentication. To allow your DAGs to use it, simply update the connection details with, for example:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM"}
```

Adjust the principal to your settings. The _HOST part will be replaced by the fully qualified domain name of the server.

You can specify if you would like to use the dag owner as the user for the connection or the user specified in the login section of the connection. For the login user, specify the following as extra:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "login"}
```

For the DAG owner use:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "owner"}
```

and in your DAG, when initializing the HiveOperator, specify:

```
run_as_owner=True
```

To use kerberos authentication, you must install Airflow with the *kerberos* extras group:

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

3.12.4 OAuth Authentication

3.12.4.1 GitHub Enterprise (GHE) Authentication

The GitHub Enterprise authentication backend can be used to authenticate users against an installation of GitHub Enterprise using OAuth2. You can optionally specify a team whitelist (composed of slug cased team names) to restrict login to only members of those teams.

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.github_enterprise_auth

[github_enterprise]
host = github.example.com
client_id = oauth_key_from_github_enterprise
client_secret = oauth_secret_from_github_enterprise
oauth_callback_route = /example/ghe_oauth/callback
allowed_teams = 1, 345, 23
```

Note: If you do not specify a team whitelist, anyone with a valid account on your GHE installation will be able to login to Airflow.

To use GHE authentication, you must install Airflow with the *github_enterprise* extras group:

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

Setting up GHE Authentication

An application must be setup in GHE before you can use the GHE authentication backend. In order to setup an application:

1. Navigate to your GHE profile
2. Select ‘Applications’ from the left hand nav

3. Select the 'Developer Applications' tab
4. Click 'Register new application'
5. Fill in the required information (the 'Authorization callback URL' must be fully qualified e.g. http://airflow.example.com/example/ghe_oauth/callback)
6. Click 'Register application'
7. Copy 'Client ID', 'Client Secret', and your callback route to your airflow.cfg according to the above example

Using GHE Authentication with github.com

It is possible to use GHE authentication with github.com:

1. Create an Oauth App
2. Copy 'Client ID', 'Client Secret' to your airflow.cfg according to the above example
3. Set `host = github.com` and `oauth_callback_route = /oauth/callback` in airflow.cfg

3.12.4.2 Google Authentication

The Google authentication backend can be used to authenticate users against Google using OAuth2. You must specify the email domains to restrict login, separated with a comma, to only members of those domains.

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.google_auth

[google]
client_id = google_client_id
client_secret = google_client_secret
oauth_callback_route = /oauth2callback
domain = example1.com,example2.com
```

To use Google authentication, you must install Airflow with the *google_auth* extras group:

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

Setting up Google Authentication

An application must be setup in the Google API Console before you can use the Google authentication backend. In order to setup an application:

1. Navigate to https://console.developers.google.com/apis/
2. Select 'Credentials' from the left hand nav
3. Click 'Create credentials' and choose 'OAuth client ID'
4. Choose 'Web application'
5. Fill in the required information (the 'Authorized redirect URIs' must be fully qualified e.g. http://airflow.example.com/oauth2callback)
6. Click 'Create'
7. Copy 'Client ID', 'Client Secret', and your redirect URI to your airflow.cfg according to the above example

3.12.5 SSL

SSL can be enabled by providing a certificate and key. Once enabled, be sure to use "https://" in your browser.

```
[webserver]
web_server_ssl_cert = <path to cert>
web_server_ssl_key = <path to key>
```

Enabling SSL will not automatically change the web server port. If you want to use the standard port 443, you'll need to configure that too. Be aware that super user privileges (or cap_net_bind_service on Linux) are required to listen on port 443.

```
# Optionally, set the server to listen on the standard SSL port.
web_server_port = 443
base_url = http://<hostname or IP>:443
```

Enable CeleryExecutor with SSL. Ensure you properly generate client and server certs and keys.

```
[celery]
ssl_active = True
ssl_key = <path to key>
ssl_cert = <path to cert>
ssl_cacert = <path to cacert>
```

3.12.6 Impersonation

Airflow has the ability to impersonate a unix user while running task instances based on the task's `run_as_user` parameter, which takes a user's name.

NOTE: For impersonations to work, Airflow must be run with *sudo* as subtasks are run with *sudo -u* and permissions of files are changed. Furthermore, the unix user needs to exist on the worker. Here is what a simple sudoers file entry could look like to achieve this, assuming as airflow is running as the *airflow* user. Note that this means that the airflow user must be trusted and treated the same way as the root user.

```
airflow ALL=(ALL) NOPASSWD: ALL
```

Subtasks with impersonation will still log to the same folder, except that the files they log to will have permissions changed such that only the unix user can write to it.

3.12.6.1 Default Impersonation

To prevent tasks that don't use impersonation to be run with *sudo* privileges, you can set the `core:default_impersonation` config which sets a default user impersonate if *run_as_user* is not set.

```
[core]
default_impersonation = airflow
```

3.12.7 Flower Authentication

Basic authentication for Celery Flower is supported.

You can specify the details either as an optional argument in the Flower process launching command, or as a configuration item in your `airflow.cfg`. For both cases, please provide *user:password* pairs separated by a comma.

```
airflow flower --basic_auth=user1:password1,user2:password2
```

```
[celery]
flower_basic_auth = user1:password1,user2:password2
```

3.12.8 RBAC UI Security

Security of Airflow Webserver UI is handled by Flask AppBuilder (FAB). Please read its related security document regarding its security model.

3.12.8.1 Default Roles

Airflow ships with a set of roles by default: Admin, User, Op, Viewer, and Public. Only `Admin` users could configure/alter the permissions for other roles. But it is not recommended that `Admin` users alter these default roles in any way by removing or adding permissions to these roles.

Admin

`Admin` users have all possible permissions, including granting or revoking permissions from other users.

Public

`Public` users (anonymous) don't have any permissions.

Viewer

`Viewer` users have limited viewer permissions

airflow/www/security.pyView Source

```
    VIEWER_PERMS = {
        'menu_access',
        'can_index',
        'can_list',
        'can_show',
        'can_chart',
        'can_dag_stats',
        'can_dag_details',
        'can_task_stats',
        'can_code',
        'can_log',
        'can_get_logs_with_metadata',
        'can_tries',
        'can_graph',
        'can_tree',
        'can_task',
        'can_task_instances',
        'can_xcom',
        'can_gantt',
        'can_landing_times',
```

(continues on next page)

(continued from previous page)

```
        'can_duration',
        'can_blocked',
        'can_rendered',
        'can_version',
    }
```

on limited web views

airflow/www/security.pyView Source

```
    VIEWER_VMS = {
        'Airflow',
        'DagModelView',
        'Browse',
        'DAG Runs',
        'DagRunModelView',
        'Task Instances',
        'TaskInstanceModelView',
        'SLA Misses',
        'SlaMissModelView',
        'Jobs',
        'JobModelView',
        'Logs',
        'LogModelView',
        'Docs',
        'Documentation',
        'Github',
        'About',
        'Version',
        'VersionView',
    }
```

User

`User` users have `Viewer` permissions plus additional user permissions

airflow/www/security.pyView Source

```
    USER_PERMS = {
        'can_dagrun_clear',
        'can_run',
        'can_trigger',
        'can_add',
        'can_edit',
        'can_delete',
        'can_paused',
        'can_refresh',
        'can_success',
        'muldelete',
        'set_failed',
        'set_running',
        'set_success',
        'clear',
        'can_clear',
    }
```

on User web views which is the same as Viewer web views.

Op

Op users have User permissions plus additional op permissions

airflow/www/security.pyView Source

```
    OP_PERMS = {
        'can_conf',
        'can_varimport',
    }
```

on User web views plus these additional op web views

airflow/www/security.pyView Source

```
    OP_VMS = {
        'Admin',
        'Configurations',
        'ConfigurationView',
        'Connections',
        'ConnectionModelView',
        'Pools',
        'PoolModelView',
        'Variables',
        'VariableModelView',
        'XComs',
        'XComModelView',
    }
```

3.12.8.2 Custom Roles

DAG Level Role

Admin can create a set of roles which are only allowed to view a certain set of dags. This is called DAG level access. Each dag defined in the dag model table is treated as a View which has two permissions associated with it (can_dag_read and can_dag_edit). There is a special view called all_dags which allows the role to access all the dags. The default Admin, Viewer, User, Op roles can all access all_dags view.

3.13 Time zones

Support for time zones is enabled by default. Airflow stores datetime information in UTC internally and in the database. It allows you to run your DAGs with time zone dependent schedules. At the moment Airflow does not convert them to the end user's time zone in the user interface. There it will always be displayed in UTC. Also templates used in Operators are not converted. Time zone information is exposed and it is up to the writer of DAG what do with it.

This is handy if your users live in more than one time zone and you want to display datetime information according to each user's wall clock.

Even if you are running Airflow in only one time zone it is still good practice to store data in UTC in your database (also before Airflow became time zone aware this was also to recommended or even required setup). The main reason is Daylight Saving Time (DST). Many countries have a system of DST, where clocks are moved forward in spring and backward in autumn. If you're working in local time, you're likely to encounter errors twice a year, when the transitions happen. (The pendulum and pytz documentation discusses these issues in greater detail.) This probably doesn't matter for a simple DAG, but it's a problem if you are in, for example, financial services where you have end of day deadlines to meet.

The time zone is set in `airflow.cfg`. By default it is set to utc, but you change it to use the system's settings or an arbitrary IANA time zone, e.g. *Europe/Amsterdam*. It is dependent on *pendulum*, which is more accurate than *pytz*. Pendulum is installed when you install Airflow.

Please note that the Web UI currently only runs in UTC.

3.13.1 Concepts

3.13.1.1 Naïve and aware datetime objects

Python's datetime.datetime objects have a tzinfo attribute that can be used to store time zone information, represented as an instance of a subclass of datetime.tzinfo. When this attribute is set and describes an offset, a datetime object is aware. Otherwise, it's naive.

You can use timezone.is_localized() and timezone.is_naive() to determine whether datetimes are aware or naive.

Because Airflow uses time-zone-aware datetime objects. If your code creates datetime objects they need to be aware too.

```
from airflow.utils import timezone

now = timezone.utcnow()
a_date = timezone.datetime(2017,1,1)
```

3.13.1.2 Interpretation of naive datetime objects

Although Airflow operates fully time zone aware, it still accepts naive date time objects for *start_dates* and *end_dates* in your DAG definitions. This is mostly in order to preserve backwards compatibility. In case a naive *start_date* or *end_date* is encountered the default time zone is applied. It is applied in such a way that it is assumed that the naive date time is already in the default time zone. In other words if you have a default time zone setting of *Europe/Amsterdam* and create a naive datetime *start_date* of *datetime(2017,1,1)* it is assumed to be a *start_date* of Jan 1, 2017 Amsterdam time.

```
default_args=dict(
    start_date=datetime(2016, 1, 1),
    owner='Airflow'
)

dag = DAG('my_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(op.owner) # Airflow
```

Unfortunately, during DST transitions, some datetimes don't exist or are ambiguous. In such situations, pendulum raises an exception. That's why you should always create aware datetime objects when time zone support is enabled.

In practice, this is rarely an issue. Airflow gives you aware datetime objects in the models and DAGs, and most often, new datetime objects are created from existing ones through timedelta arithmetic. The only datetime that's often created in application code is the current time, and timezone.utcnow() automatically does the right thing.

3.13.1.3 Default time zone

The default time zone is the time zone defined by the *default_timezone* setting under *[core]*. If you just installed Airflow it will be set to *utc*, which is recommended. You can also set it to *system* or an IANA time zone (e.g.'Europe/Amsterdam'). DAGs are also evaluated on Airflow workers, it is therefore important to make sure this setting is equal on all Airflow nodes.

```
[core]
default_timezone = utc
```

3.13.2 Time zone aware DAGs

Creating a time zone aware DAG is quite simple. Just make sure to supply a time zone aware *start_date* using *pendulum*.

```
import pendulum

local_tz = pendulum.timezone("Europe/Amsterdam")

default_args=dict(
    start_date=datetime(2016, 1, 1, tzinfo=local_tz),
    owner='Airflow'
)

dag = DAG('my_tz_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(dag.timezone) # <Timezone [Europe/Amsterdam]>
```

Please note that while it is possible to set a *start_date* and *end_date* for Tasks always the DAG timezone or global timezone (in that order) will be used to calculate the next execution date. Upon first encounter the start date or end date will be converted to UTC using the timezone associated with start_date or end_date, then for calculations this timezone information will be disregarded.

3.13.2.1 Templates

Airflow returns time zone aware datetimes in templates, but does not convert them to local time so they remain in UTC. It is left up to the DAG to handle this.

```
import pendulum

local_tz = pendulum.timezone("Europe/Amsterdam")
local_tz.convert(execution_date)
```

3.13.2.2 Cron schedules

In case you set a cron schedule, Airflow assumes you will always want to run at the exact same time. It will then ignore day light savings time. Thus, if you have a schedule that says run at the end of interval every day at 08:00 GMT+1 it will always run at the end of interval 08:00 GMT+1, regardless if day light savings time is in place.

3.13.2.3 Time deltas

For schedules with time deltas Airflow assumes you always will want to run with the specified interval. So if you specify a timedelta(hours=2) you will always want to run two hours later. In this case day light savings time will be taken into account.

3.14 Experimental Rest API

Airflow exposes an experimental Rest API. It is available through the webserver. Endpoints are available at /api/experimental/. Please note that we expect the endpoint definitions to change.

3.14.1 Endpoints

POST /api/experimental/dags/<DAG_ID>/dag_runs
Creates a dag_run for a given dag id.

Trigger DAG with config, example:

```
curl -X POST \
  http://localhost:8080/api/experimental/dags/<DAG_ID>/dag_runs \
  -H 'Cache-Control: no-cache' \
  -H 'Content-Type: application/json' \
  -d '{"conf":"{\"key\":\"value\"}"}'
```

GET /api/experimental/dags/<DAG_ID>/dag_runs
Returns a list of Dag Runs for a specific DAG ID.

GET /api/experimental/dags/<string:dag_id>/dag_runs/<string:execution_date>
Returns a JSON with a dag_run's public instance variables. The format for the <string:execution_date> is expected to be "YYYY-mm-DDTHH:MM:SS", for example: "2016-11-16T11:34:15".

GET /api/experimental/test
To check REST API server correct work. Return status 'OK'.

GET /api/experimental/dags/<DAG_ID>/tasks/<TASK_ID>
Returns info for a task.

GET /api/experimental/dags/<DAG_ID>/dag_runs/<string:execution_date>/tasks/<TASK_ID>
Returns a JSON with a task instance's public instance variables. The format for the <string:execution_date> is expected to be "YYYY-mm-DDTHH:MM:SS", for example: "2016-11-16T11:34:15".

GET /api/experimental/dags/<DAG_ID>/paused/<string:paused>
'<string:paused>' must be a 'true' to pause a DAG and 'false' to unpause.

GET /api/experimental/latest_runs
Returns the latest DagRun for each DAG formatted for the UI.

GET /api/experimental/pools
Get all pools.

GET /api/experimental/pools/<string:name>
Get pool by a given name.

POST /api/experimental/pools
Create a pool.

DELETE /api/experimental/pools/<string:name>
Delete pool.

3.14.2 CLI

For some functions the cli can use the API. To configure the CLI to use the API when available configure as follows:

```
[cli]
api_client = airflow.api.client.json_client
endpoint_url = http://<WEBSERVER>:<PORT>
```

3.14.3 Authentication

Authentication for the API is handled separately to the Web Authentication. The default is to not require any authentication on the API – i.e. wide open by default. This is not recommended if your Airflow webserver is publicly accessible, and you should probably use the deny all backend:

```
[api]
auth_backend = airflow.api.auth.backend.deny_all
```

Two "real" methods for authentication are currently supported for the API.

To enabled Password authentication, set the following in the configuration:

```
[api]
auth_backend = airflow.contrib.auth.backends.password_auth
```

It's usage is similar to the Password Authentication used for the Web interface.

To enable Kerberos authentication, set the following in the configuration:

```
[api]
auth_backend = airflow.api.auth.backend.kerberos_auth

[kerberos]
keytab = <KEYTAB>
```

The Kerberos service is configured as `airflow/fully.qualified.domainname@REALM`. Make sure this principal exists in the keytab file.

3.15 Integration

- *Azure: Microsoft Azure*
- *AWS: Amazon Web Services*
- *Databricks*
- *GCP: Google Cloud Platform*
- *Qubole*

3.15.1 Azure: Microsoft Azure

Airflow has limited support for Microsoft Azure: interfaces exist only for Azure Blob Storage and Azure Data Lake. Hook, Sensor and Operator for Blob Storage and Azure Data Lake Hook are in contrib section.

3.15.1.1 Azure Blob Storage

All classes communicate via the Window Azure Storage Blob protocol. Make sure that a Airflow connection of type *wasb* exists. Authorization can be done by supplying a login (=Storage account name) and password (=KEY), or login and SAS token in the extra field (see connection *wasb_default* for an example).

`airflow.contrib.hooks.wasb_hook.WasbHook` Interface with Azure Blob Storage.

`airflow.contrib.sensors.wasb_sensor.WasbBlobSensor` Checks if a blob is present on Azure Blob storage.

`airflow.contrib.operators.wasb_delete_blob_operator.WasbDeleteBlobOperator` Deletes blob(s) on Azure Blob Storage.

`airflow.contrib.sensors.wasb_sensor.WasbPrefixSensor` Checks if blobs matching a prefix are present on Azure Blob storage.

airflow.contrib.operators.file_to_wasb.FileToWasbOperator Uploads a local file to a container as a blob.

3.15.1.2 Azure File Share

Cloud variant of a SMB file share. Make sure that a Airflow connection of type *wasb* exists. Authorization can be done by supplying a login (=Storage account name) and password (=Storage account key), or login and SAS token in the extra field (see connection *wasb_default* for an example).

***airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook*:** Interface with Azure File Share.

3.15.1.3 Logging

Airflow can be configured to read and write task logs in Azure Blob Storage. See *Writing Logs to Azure Blob Storage*.

3.15.1.4 Azure CosmosDB

AzureCosmosDBHook communicates via the Azure Cosmos library. Make sure that a Airflow connection of type *azure_cosmos* exists. Authorization can be done by supplying a login (=Endpoint uri), password (=secret key) and extra fields database_name and collection_name to specify the default database and collection to use (see connection *azure_cosmos_default* for an example).

airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook Interface with Azure CosmosDB.

airflow.contrib.operators.azure_cosmos_operator.AzureCosmosInsertDocumentOperator Simple operator to insert document into CosmosDB.

airflow.contrib.sensors.azure_cosmos_sensor.AzureCosmosDocumentSensor Simple sensor to detect document existence in CosmosDB.

3.15.1.5 Azure Data Lake

AzureDataLakeHook communicates via a REST API compatible with WebHDFS. Make sure that a Airflow connection of type *azure_data_lake* exists. Authorization can be done by supplying a login (=Client ID), password (=Client Secret) and extra fields tenant (Tenant) and account_name (Account Name) (see connection *azure_data_lake_default* for an example).

airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook Interface with Azure Data Lake.

airflow.contrib.operators.adls_list_operator.AzureDataLakeStorageListOperator Lists the files located in a specified Azure Data Lake path.

airflow.contrib.operators.adls_to_gcs.AdlsToGoogleCloudStorageOperator Copies files from an Azure Data Lake path to a Google Cloud Storage bucket.

3.15.1.6 Azure Container Instances

Azure Container Instances provides a method to run a docker container without having to worry about managing infrastructure. The AzureContainerInstanceHook requires a service principal. The credentials for this principal can either be defined in the extra field `key_path`, as an environment variable named `AZURE_AUTH_LOCATION`, or by providing a login/password and tenantId in extras.

The AzureContainerRegistryHook requires a host/login/password to be defined in the connection.

airflow.contrib.hooks.azure_container_volume_hook.AzureContainerVolumeHook
Interface with Azure Container Volumes

airflow.contrib.operators.azure_container_instances_operator.AzureContainerInstancesOperat
Start/Monitor a new ACI.

airflow.contrib.hooks.azure_container_instance_hook.AzureContainerInstanceHook
Wrapper around a single ACI.

airflow.contrib.hooks.azure_container_registry_hook.AzureContainerRegistryHook
Interface with ACR

3.15.2 AWS: Amazon Web Services

Airflow has extensive support for Amazon Web Services. But note that the Hooks, Sensors and Operators are in the contrib section.

3.15.2.1 AWS EMR

airflow.contrib.hooks.emr_hook.EmrHook Interface with AWS EMR.

airflow.contrib.operators.emr_add_steps_operator.EmrAddStepsOperator Adds steps to an existing EMR JobFlow.

airflow.contrib.operators.emr_create_job_flow_operator.EmrCreateJobFlowOperator
Creates an EMR JobFlow, reading the config from the EMR connection.

airflow.contrib.operators.emr_terminate_job_flow_operator.EmrTerminateJobFlowOperator
Terminates an EMR JobFlow.

3.15.2.2 AWS S3

airflow.hooks.S3_hook.S3Hook Interface with AWS S3.

airflow.operators.s3_file_transform_operator.S3FileTransformOperator Copies data from a source S3 location to a temporary location on the local filesystem.

airflow.contrib.operators.s3_list_operator.S3ListOperator Lists the files matching a key prefix from a S3 location.

airflow.contrib.operators.s3_to_gcs_operator.S3ToGoogleCloudStorageOperator
Syncs an S3 location with a Google Cloud Storage bucket.

airflow.contrib.operators.s3_to_gcs_transfer_operator.S3ToGoogleCloudStorageTransferOperat
Syncs an S3 bucket with a Google Cloud Storage bucket using the GCP Storage Transfer Service.

airflow.operators.s3_to_hive_operator.S3ToHiveTransfer Moves data from S3 to Hive. The operator downloads a file from S3, stores the file locally before loading it into a Hive table.

3.15.2.3 AWS Batch Service

airflow.contrib.operators.awsbatch_operator.AWSBatchOperator Execute a task on AWS Batch Service.

3.15.2.4 AWS RedShift

airflow.contrib.sensors.aws_redshift_cluster_sensor.AwsRedshiftClusterSensor Waits for a Redshift cluster to reach a specific status.

airflow.contrib.hooks.redshift_hook.RedshiftHook Interact with AWS Redshift, using the boto3 library.

airflow.operators.redshift_to_s3_operator.RedshiftToS3Transfer Executes an unload command to S3 as CSV with or without headers.

airflow.operators.s3_to_redshift_operator.S3ToRedshiftTransfer Executes an copy command from S3 as CSV with or without headers.

3.15.2.5 AWS DynamoDB

airflow.contrib.operators.hive_to_dynamodb.HiveToDynamoDBTransferOperator Moves data from Hive to DynamoDB.

airflow.contrib.hooks.aws_dynamodb_hook.AwsDynamoDBHook Interface with AWS DynamoDB.

3.15.2.6 AWS Lambda

airflow.contrib.hooks.aws_lambda_hook.AwsLambdaHook Interface with AWS Lambda.

3.15.2.7 AWS Kinesis

airflow.contrib.hooks.aws_firehose_hook.AwsFirehoseHook Interface with AWS Kinesis Firehose.

3.15.2.8 Amazon SageMaker

For more instructions on using Amazon SageMaker in Airflow, please see the SageMaker Python SDK README.

airflow.contrib.hooks.sagemaker_hook.SageMakerHook Interface with Amazon SageMaker.

airflow.contrib.operators.sagemaker_training_operator.SageMakerTrainingOperator Create a SageMaker training job.

airflow.contrib.operators.sagemaker_tuning_operator.SageMakerTuningOperator Create a SageMaker tuning job.

airflow.contrib.operators.sagemaker_model_operator.SageMakerModelOperator Create a SageMaker model.

airflow.contrib.operators.sagemaker_transform_operator.SageMakerTransformOperator Create a SageMaker transform job.

airflow.contrib.operators.sagemaker_endpoint_config_operator.SageMakerEndpointConfigOperat Create a SageMaker endpoint config.

airflow.contrib.operators.sagemaker_endpoint_operator.SageMakerEndpointOperator Create a SageMaker endpoint.

3.15.3 Databricks

With contributions from Databricks, Airflow has several operators which enable the submitting and running of jobs to the Databricks platform. Internally the operators talk to the `api/2.0/jobs/runs/submit` endpoint.

`airflow.contrib.operators.databricks_operator.DatabricksSubmitRunOperator`
Submits a Spark job run to Databricks using the api/2.0/jobs/runs/submit API endpoint.

`airflow.contrib.operators.databricks_operator.DatabricksRunNowOperator`

Runs an existing Spark job in Databricks using the api/2.0/jobs/run-now API endpoint.

3.15.4 GCP: Google Cloud Platform

Airflow has extensive support for the Google Cloud Platform. But note that most Hooks and Operators are in the contrib section. Meaning that they have a *beta* status, meaning that they can have breaking changes between minor releases.

See the *GCP connection type* documentation to configure connections to GCP.

3.15.4.1 Logging

Airflow can be configured to read and write task logs in Google Cloud Storage. See *Writing Logs to Google Cloud Storage*.

3.15.4.2 GoogleCloudBaseHook

All hooks is based on `airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook`.

3.15.4.3 BigQuery

`airflow.contrib.operators.bigquery_check_operator.BigQueryCheckOperator` Performs checks against a SQL query that will return a single row with different values.

`airflow.contrib.operators.bigquery_check_operator.BigQueryIntervalCheckOperator`
Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.

`airflow.contrib.operators.bigquery_check_operator.BigQueryValueCheckOperator`
Performs a simple value check using SQL code.

`airflow.contrib.operators.bigquery_get_data.BigQueryGetDataOperator` Fetches the data from a BigQuery table and returns data in a python list

`airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyDatasetOperator`
Creates an empty BigQuery dataset.

`airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyTableOperator`
Creates a new, empty table in the specified BigQuery dataset optionally with schema.

`airflow.contrib.operators.bigquery_operator.BigQueryCreateExternalTableOperator`
Creates a new, external table in the dataset with the data in Google Cloud Storage.

`airflow.contrib.operators.bigquery_operator.BigQueryDeleteDatasetOperator`
Deletes an existing BigQuery dataset.

`airflow.contrib.operators.bigquery_operator.BigQueryOperator` Executes BigQuery SQL queries in a specific BigQuery database.

`airflow.contrib.operators.bigquery_table_delete_operator.BigQueryTableDeleteOperator`
Deletes an existing BigQuery table.

airflow.contrib.operators.bigquery_to_bigquery.BigQueryToBigQueryOperator Copy a BigQuery table to another BigQuery table.

airflow.contrib.operators.bigquery_to_gcs.BigQueryToCloudStorageOperator Transfers a BigQuery table to a Google Cloud Storage bucket

They also use *airflow.contrib.hooks.bigquery_hook.BigQueryHook* to communicate with Google Cloud Platform.

3.15.4.4 Cloud Spanner

airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeleteOperator deletes an existing database from a Google Cloud Spanner instance or returns success if the database is missing.

airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeployOperator creates a new database in a Google Cloud instance or returns success if the database already exists.

airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseQueryOperator executes an arbitrary DML query (INSERT, UPDATE, DELETE).

airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseUpdateOperator updates the structure of a Google Cloud Spanner database.

airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDeleteOperator deletes a Google Cloud Spanner instance.

airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDeployOperator creates a new Google Cloud Spanner instance, or if an instance with the same name exists, updates the instance.

They also use *airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook* to communicate with Google Cloud Platform.

3.15.4.5 Cloud SQL

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceCreateOperator create a new Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseCreateOperator creates a new database inside a Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseDeleteOperator deletes a database from a Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabasePatchOperator updates a database inside a Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDeleteOperator delete a Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceExportOperator exports data from a Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceImportOperator imports data into a Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlInstancePatchOperator patch a Cloud SQL instance.

airflow.contrib.operators.gcp_sql_operator.CloudSqlQueryOperator run query in a Cloud SQL instance.

They also use *airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook* and *airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook* to communicate with Google Cloud Platform.

3.15.4.6 Cloud Bigtable

airflow.contrib.operators.gcp_bigtable_operator.BigtableClusterUpdateOperator
updates the number of nodes in a Google Cloud Bigtable cluster.

airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceCreateOperator
creates a Cloud Bigtable instance.

airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceDeleteOperator
deletes a Google Cloud Bigtable instance.

airflow.contrib.operators.gcp_bigtable_operator.BigtableTableCreateOperator
creates a table in a Google Cloud Bigtable instance.

airflow.contrib.operators.gcp_bigtable_operator.BigtableTableDeleteOperator
deletes a table in a Google Cloud Bigtable instance.

airflow.contrib.operators.gcp_bigtable_operator.BigtableTableWaitForReplicationSensor
(sensor) waits for a table to be fully replicated.

They also use `airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook` to communicate with Google Cloud Platform.

3.15.4.7 Cloud Build

airflow.contrib.operators.gcp_cloud_build_operator.CloudBuildCreateBuildOperator
Starts a build with the specified configuration.

They also use `airflow.contrib.hooks.gcp_cloud_build_hook.CloudBuildHook` to communicate with Google Cloud Platform.

3.15.4.8 Compute Engine

airflow.contrib.operators.gcp_compute_operator.GceInstanceStartOperator start an existing Google Compute Engine instance.

airflow.contrib.operators.gcp_compute_operator.GceInstanceStopOperator stop an existing Google Compute Engine instance.

airflow.contrib.operators.gcp_compute_operator.GceSetMachineTypeOperator change the machine type for a stopped instance.

airflow.contrib.operators.gcp_compute_operator.GceInstanceTemplateCopyOperator
copy the Instance Template, applying specified changes.

airflow.contrib.operators.gcp_compute_operator.GceInstanceGroupManagerUpdateTemplateOperat
patch the Instance Group Manager, replacing source Instance Template URL with the destination one.

The operators have the common base operator `airflow.contrib.operators.gcp_compute_operator.GceBaseOperator`

They also use `airflow.contrib.hooks.gcp_compute_hook.GceHook` to communicate with Google Cloud Platform.

3.15.4.9 Cloud Functions

airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator deploy Google Cloud Function to Google Cloud Platform

airflow.contrib.operators.gcp_function_operator.GcfFunctionDeleteOperator delete Google Cloud Function in Google Cloud Platform

They also use *airflow.contrib.hooks.gcp_function_hook.GcfHook* to communicate with Google Cloud Platform.

3.15.4.10 Cloud DataFlow

airflow.contrib.operators.dataflow_operator.DataFlowJavaOperator launching Cloud Dataflow jobs written in Java.

airflow.contrib.operators.dataflow_operator.DataflowTemplateOperator launching a templated Cloud DataFlow batch job.

airflow.contrib.operators.dataflow_operator.DataFlowPythonOperator launching Cloud Dataflow jobs written in python.

They also use *airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook* to communicate with Google Cloud Platform.

3.15.4.11 Cloud DataProc

airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOperator Create a new cluster on Google Cloud Dataproc.

airflow.contrib.operators.dataproc_operator.DataprocClusterDeleteOperator Delete a cluster on Google Cloud Dataproc.

airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOperator Scale up or down a cluster on Google Cloud Dataproc.

airflow.contrib.operators.dataproc_operator.DataProcHadoopOperator Start a Hadoop Job on a Cloud DataProc cluster.

airflow.contrib.operators.dataproc_operator.DataProcHiveOperator Start a Hive query Job on a Cloud DataProc cluster.

airflow.contrib.operators.dataproc_operator.DataProcPigOperator Start a Pig query Job on a Cloud DataProc cluster.

airflow.contrib.operators.dataproc_operator.DataProcPySparkOperator Start a PySpark Job on a Cloud DataProc cluster.

airflow.contrib.operators.dataproc_operator.DataProcSparkOperator Start a Spark Job on a Cloud DataProc cluster.

airflow.contrib.operators.dataproc_operator.DataProcSparkSqlOperator Start a Spark SQL query Job on a Cloud DataProc cluster.

airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateInlineOpera
Instantiate a WorkflowTemplate Inline on Google Cloud Dataproc.

airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateOperator
Instantiate a WorkflowTemplate on Google Cloud Dataproc.

3.15.4.12 Cloud Datastore

airflow.contrib.operators.datastore_export_operator.DatastoreExportOperator
Export entities from Google Cloud Datastore to Cloud Storage.

airflow.contrib.operators.datastore_import_operator.DatastoreImportOperator
Import entities from Cloud Storage to Google Cloud Datastore.

They also use *airflow.contrib.hooks.datastore_hook.DatastoreHook* to communicate with Google Cloud Platform.

3.15.4.13 Cloud ML Engine

airflow.contrib.operators.mlengine_operator.MLEngineBatchPredictionOperator
Start a Cloud ML Engine batch prediction job.

airflow.contrib.operators.mlengine_operator.MLEngineModelOperator Manages a Cloud ML Engine model.

airflow.contrib.operators.mlengine_operator.MLEngineTrainingOperator Start a Cloud ML Engine training job.

airflow.contrib.operators.mlengine_operator.MLEngineVersionOperator Manages a Cloud ML Engine model version.

They also use *airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook* to communicate with Google Cloud Platform.

3.15.4.14 Cloud Storage

airflow.contrib.operators.file_to_gcs.FileToGoogleCloudStorageOperator Uploads a file to Google Cloud Storage.

airflow.contrib.operators.gcs_acl_operator.GoogleCloudStorageBucketCreateAclEntryOperator
Creates a new ACL entry on the specified bucket.

airflow.contrib.operators.gcs_acl_operator.GoogleCloudStorageObjectCreateAclEntryOperator
Creates a new ACL entry on the specified object.

airflow.contrib.operators.gcs_download_operator.GoogleCloudStorageDownloadOperator
Downloads a file from Google Cloud Storage.

airflow.contrib.operators.gcs_list_operator.GoogleCloudStorageListOperator List all objects from the bucket with the give string prefix and delimiter in name.

airflow.contrib.operators.gcs_operator.GoogleCloudStorageCreateBucketOperator
Creates a new cloud storage bucket.

airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBigQueryOperator Loads files from Google cloud storage into BigQuery.

airflow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator
Copies objects from a bucket to another, with renaming if requested.

airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator Copy data from any MySQL Database to Google cloud storage in JSON format.

airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOperator Copy data from any Microsoft SQL Server Database to Google Cloud Storage in JSON format.

airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectSensor Checks for the existence of a file in Google Cloud Storage.

airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectUpdatedSensor Checks if an object is updated in Google Cloud Storage.

airflow.contrib.sensors.gcs_sensor.GoogleCloudStoragePrefixSensor Checks for the existence of a objects at prefix in Google Cloud Storage.

airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageUploadSessionCompleteSession
Checks for changes in the number of objects at prefix in Google Cloud Storage bucket and returns True if the inactivity period has passed with no increase in the number of objects for situations when many objects are being uploaded to a bucket with no formal success signal.

airflow.contrib.operators.gcs_delete_operator.GoogleCloudStorageDeleteOperator
Deletes objects from a Google Cloud Storage bucket.

They also use *airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook* to communicate with Google Cloud Platform.

3.15.4.15 Transfer Service

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobDeleteOperator
Deletes a transfer job.

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobCreateOperator
Creates a transfer job.

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobUpdateOperator
Updates a transfer job.

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationCancelOperator
Cancels a transfer operation.

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationGetOperator
Gets a transfer operation.

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationPauseOperator
Pauses a transfer operation

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationResumeOperator
Resumes a transfer operation.

airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationsListOperator
Gets a list of transfer operations.

airflow.contrib.operators.gcp_transfer_operator.GoogleCloudStorageToGoogleCloudStorageTrans
Copies objects from a Google Cloud Storage bucket to another bucket.

airflow.contrib.operators.gcp_transfer_operator.S3ToGoogleCloudStorageTransferOperator
Synchronizes an S3 bucket with a Google Cloud Storage bucket.

airflow.contrib.sensors.gcp_transfer_operator.GCPTransferServiceWaitForJobStatusSensor
Waits for at least one operation belonging to the job to have the expected status.

They also use *airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook* to communicate with Google Cloud Platform.

3.15.4.16 Cloud Vision

Cloud Vision Product Search Operators

airflow.contrib.operators.gcp_vision_operator.CloudVisionAddProductToProductSetOperator
Adds a Product to the specified ProductSet.

airflow.contrib.operators.gcp_vision_operator.CloudVisionAnnotateImageOperator
Run image detection and annotation for an image.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductCreateOperator
Creates a new Product resource.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductDeleteOperator
Permanently deletes a product and its reference images.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductGetOperator
Gets information associated with a Product.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetCreateOperator
Creates a new ProductSet resource.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetDeleteOperator
Permanently deletes a ProductSet.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetGetOperator
Gets information associated with a ProductSet.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetUpdateOperator
Makes changes to a ProductSet resource.

airflow.contrib.operators.gcp_vision_operator.CloudVisionProductUpdateOperator
Makes changes to a Product resource.

airflow.contrib.operators.gcp_vision_operator.CloudVisionReferenceImageCreateOperator
Creates a new ReferenceImage resource.

airflow.contrib.operators.gcp_vision_operator.CloudVisionRemoveProductFromProductSetOperat
Removes a Product from the specified ProductSet.

airflow.contrib.operators.gcp_vision_operator.CloudVisionAnnotateImageOperator
Run image detection and annotation for an image.

airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectTextOperator
Run text detection for an image

airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectDocumentTextOperator
Run document text detection for an image

airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectImageLabelsOperator
Run image labels detection for an image

airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectImageSafeSearchOperator
Run safe search detection for an image

They also use *airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook* to communicate with Google Cloud Platform.

3.15.4.17 Cloud Text to Speech

airflow.contrib.operators.gcp_text_to_speech_operator.GcpTextToSpeechSynthesizeOperator
Synthesizes input text into audio file and stores this file to GCS.

They also use *airflow.contrib.hooks.gcp_text_to_speech_hook.GCPTextToSpeechHook* to communicate with Google Cloud Platform.

3.15.4.18 Cloud Speech to Text

airflow.contrib.operators.gcp_speech_to_text_operator.GcpSpeechToTextRecognizeSpeechOperat
Recognizes speech in audio input and returns text.

They also use *airflow.contrib.hooks.gcp_speech_to_text_hook.GCPSpeechToTextHook* to communicate with Google Cloud Platform.

3.15.5 Cloud Speech Translate Operators

airflow.contrib.operators.gcp_translate_speech_operator.GcpTranslateSpeechOperator
Recognizes speech in audio input and translates it.

They also use *airflow.contrib.hooks.gcp_speech_to_text_hook.GCPSpeechToTextHook* and
airflow.contrib.hooks.gcp_translate_hook.CloudTranslateHook to communicate with Google Cloud Platform.

3.15.5.1 Cloud Translate

Cloud Translate Text Operators

airflow.contrib.operators.gcp_translate_operator.CloudTranslateTextOperator
Translate a string or list of strings.

3.15.5.2 Cloud Video Intelligence

airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVide
Performs video annotation, annotating video labels.

airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVide
Performs video annotation, annotating explicit content.

airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVide
Performs video annotation, annotating video shots.

They also use *airflow.contrib.hooks.gcp_video_intelligence_hook.CloudVideoIntelligenceHook* to communicate with Google Cloud Platform.

3.15.5.3 Google Kubernetes Engine

airflow.contrib.operators.gcp_container_operator.GKEClusterCreateOperator
Creates a Kubernetes Cluster in Google Cloud Platform

airflow.contrib.operators.gcp_container_operator.GKEClusterDeleteOperator
Deletes a Kubernetes Cluster in Google Cloud Platform

airflow.contrib.operators.gcp_container_operator.GKEPodOperator Executes a task in a Kubernetes pod in the specified Google Kubernetes Engine cluster

They also use *airflow.contrib.hooks.gcp_container_hook.GKEClusterHook* to communicate with Google Cloud Platform.

3.15.5.4 Google Natural Language

airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageAnalyzeEntities
Finds named entities (currently proper names and common nouns) in the text along with entity types, salience, mentions for each entity, and other properties.

airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageAnalyzeEntitySentiment
Finds entities, similar to AnalyzeEntities in the text and analyzes sentiment associated with each entity and its mentions.

airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageAnalyzeSentiment
Analyzes the sentiment of the provided text.

airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageClassifyTextOperator
Classifies a document into categories.

They also use `airflow.contrib.hooks.gcp_natural_language_operator.CloudNaturalLanguageHook` to communicate with Google Cloud Platform.

3.15.6 Qubole

Apache Airflow has a native operator and hooks to talk to Qubole, which lets you submit your big data jobs directly to Qubole from Apache Airflow.

airflow.contrib.operators.qubole_operator.QuboleOperator Execute tasks (commands) on QDS (https://qubole.com).

airflow.contrib.sensors.qubole_sensor.QubolePartitionSensor Wait for a Hive partition to show up in QHS (Qubole Hive Service) and check for its presence via QDS APIs

airflow.contrib.sensors.qubole_sensor.QuboleFileSensor Wait for a file or folder to be present in cloud storage and check for its presence via QDS APIs

airflow.contrib.operators.qubole_check_operator.QuboleCheckOperator Performs checks against Qubole Commands. `QuboleCheckOperator` expects a command that will be executed on QDS.

airflow.contrib.operators.qubole_check_operator.QuboleValueCheckOperator
Performs a simple value check using Qubole command. By default, each value on the first row of this Qubole command is compared with a pre-defined value

3.16 Metrics

Airflow can be set up to send metrics to StatsD.

3.16.1 Setup

First you must install statsd requirement:

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

Add the following lines to your configuration file e.g. `airflow.cfg`

```
[scheduler]
statsd_on = True
statsd_host = localhost
statsd_port = 8125
statsd_prefix = airflow
```

3.16.2 Counters

Name	Description
<job_name>_start	Number of started <job_name> job, ex. SchedulerJob, LocalTaskJob
<job_name>_end	Number of ended <job_name> job, ex. SchedulerJob, LocalTaskJob
operator_failures_<operator_name>	Operator <operator_name> failures
operator_successes_<operator_name>	Operator <operator_name> successes
ti_failures	Overall task instances failures
ti_successes	Overall task instances successes
zombies_killed	Zombie tasks killed
scheduler_heartbeat	Scheduler heartbeats

3.16.3 Gauges

Name	Description
collect_dags	Seconds taken to scan and import DAGs
dagbag_import_errors	DAG import errors
dagbag_size	DAG bag size
dag_processing.last_runtime.<dag_file>	Seconds spent processing <dag_file> (in most recent iteration)
dag_processing.last_run.seconds_ago.<dag_file>	Seconds since <dag_file> was last processed
executor.open_slots	Number of of open slots on executor
executor.queued_tasks	Number of queued tasks on executor
executor.running_tasks	Number of running tasks on executor
pool.starving_tasks.<pool_name>	Number of starving tasks in the pool

3.16.4 Timers

Name	Description
dagrun.dependency-check.<dag_id>	Seconds taken to check DAG dependencies
dag.<dag_id>.<task_id>.duration	Seconds taken to finish a task
dag.loading-duration.<dag_id>	Seconds taken to load the given DAG
dagrun.duration.success.<dag_id>	Seconds taken for a DagRun to reach success state
dagrun.duration.failed.<dag_id>	Seconds taken for a DagRun to reach failed state
dagrun.schedule_delay.<dag_id>	Seconds of delay between the scheduled DagRun start date and the actual DagRun start date

3.17 Kubernetes

3.17.1 Kubernetes Executor

The kubernetes executor is introduced in Apache Airflow 1.10.0. The Kubernetes executor will create a new pod for every task instance.

Example helm charts are available at `scripts/ci/kubernetes/kube/{airflow,volumes,postgres}.yaml` in the source distribution. The volumes are optional and depend on your configuration. There are two volumes available:

- **Dags**:
  - By storing dags onto persistent disk, it will be made available to all workers
  - Another option is to use `git-sync`. Before starting the container, a git pull of the dags repository will be performed and used throughout the lifecycle of the pod
- **Logs**:
  - By storing logs onto a persistent disk, the files are accessible by workers and the webserver. If you don't configure this, the logs will be lost after the worker pods shuts down
  - Another option is to use S3/GCS/etc to store logs

3.17.2 Kubernetes Operator

```
from airflow.contrib.operators import KubernetesOperator
from airflow.contrib.operators.kubernetes_pod_operator import KubernetesPodOperator
from airflow.contrib.kubernetes.secret import Secret
from airflow.contrib.kubernetes.volume import Volume
from airflow.contrib.kubernetes.volume_mount import VolumeMount
from airflow.contrib.kubernetes.pod import Port


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 = VolumeMount('test-volume',
                            mount_path='/root/mount_file',
                            sub_path=None,
                            read_only=True)
port = Port('http', 80)
configmaps = ['test-configmap-1', 'test-configmap-2']

volume_config= {
    'persistentVolumeClaim':
      {
        'claimName': 'test-volume'
      }
    }
volume = Volume(name='test-volume', configs=volume_config)

affinity = {
    'nodeAffinity': {
      'preferredDuringSchedulingIgnoredDuringExecution': [
        {
          "weight": 1,
          "preference": {
            "matchExpressions": {
              "key": "disktype",
              "operator": "In",
              "values": ["ssd"]
            }
          }
        }
```

(continues on next page)

(continued from previous page)

```
            ]
        },
        "podAffinity": {
          "requiredDuringSchedulingIgnoredDuringExecution": [
            {
              "labelSelector": {
                "matchExpressions": [
                  {
                    "key": "security",
                    "operator": "In",
                    "values": ["S1"]
                  }
                ]
              },
              "topologyKey": "failure-domain.beta.kubernetes.io/zone"
            }
          ]
        },
        "podAntiAffinity": {
          "requiredDuringSchedulingIgnoredDuringExecution": [
            {
              "labelSelector": {
                "matchExpressions": [
                  {
                    "key": "security",
                    "operator": "In",
                    "values": ["S2"]
                  }
                ]
              },
              "topologyKey": "kubernetes.io/hostname"
            }
          ]
        }
}

tolerations = [
    {
        'key': "key",
        'operator': 'Equal',
        'value': 'value'
     }
]

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

(continues on next page)

(continued from previous page)

```
                              hostnetwork=False,
                              tolerations=tolerations,
                              configmaps=configmaps
                              )
```

See *airflow.contrib.operators.kubernetes_pod_operator.KubernetesPodOperator*

3.17.3 Pod Mutation Hook

Your local Airflow settings file can define a `pod_mutation_hook` function that has the ability to mutate pod objects before sending them to the Kubernetes client for scheduling. It receives a single argument as a reference to pod objects, and is expected to alter its attributes.

This could be used, for instance, to add sidecar or init containers to every worker pod launched by KubernetesExecutor or KubernetesPodOperator.

```
def pod_mutation_hook(pod: Pod):
  pod.annotations['airflow.apache.org/launched-by'] = 'Tests'
```

3.18 Lineage

Note: Lineage support is very experimental and subject to change.

Airflow can help track origins of data, what happens to it and where it moves over time. This can aid having audit trails and data governance, but also debugging of data flows.

Airflow tracks data by means of inlets and outlets of the tasks. Let's work from an example and see how it works.

```
from airflow.operators.bash_operator import BashOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.lineage.datasets import File
from airflow.models import DAG
from datetime import timedelta

FILE_CATEGORIES = ["CAT1", "CAT2", "CAT3"]

args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2)
}

dag = DAG(
    dag_id='example_lineage', default_args=args,
    schedule_interval='0 0 * * *',
    dagrun_timeout=timedelta(minutes=60))

f_final = File("/tmp/final")
run_this_last = DummyOperator(task_id='run_this_last', dag=dag,
    inlets={"auto": True},
    outlets={"datasets": [f_final,]})

f_in = File("/tmp/whole_directory/")
```

(continues on next page)

(continued from previous page)

```
outlets = []
for file in FILE_CATEGORIES:
    f_out = File("/tmp/{}/{{{{ execution_date }}}}".format(file))
    outlets.append(f_out)
run_this = BashOperator(
    task_id='run_me_first', bash_command='echo 1', dag=dag,
    inlets={"datasets": [f_in,]},
    outlets={"datasets": outlets}
    )
run_this.set_downstream(run_this_last)
```

Tasks take the parameters *inlets* and *outlets*.

Inlets can be manually defined by the following options:

- by a list of dataset `{"datasets": [dataset1, dataset2]}`
- can be configured to look for outlets from upstream tasks `{"task_ids": ["task_id1", "task_id2"]}`
- can be configured to pick up outlets from direct upstream tasks `{"auto": True}`
- a combination of them

Outlets are defined as list of dataset `{"datasets": [dataset1, dataset2]}`. Any fields for the dataset are templated with the context when the task is being executed.

Note: Operators can add inlets and outlets automatically if the operator supports it.

In the example DAG task *run_me_first* is a BashOperator that takes 3 inlets: *CAT1*, *CAT2*, *CAT3*, that are generated from a list. Note that *execution_date* is a templated field and will be rendered when the task is running.

Note: Behind the scenes Airflow prepares the lineage metadata as part of the *pre_execute* method of a task. When the task has finished execution *post_execute* is called and lineage metadata is pushed into XCOM. Thus if you are creating your own operators that override this method make sure to decorate your method with *prepare_lineage* and *apply_lineage* respectively.

3.18.1 Apache Atlas

Airflow can send its lineage metadata to Apache Atlas. You need to enable the *atlas* backend and configure it properly, e.g. in your `airflow.cfg`:

```
[lineage]
backend = airflow.lineage.backend.atlas.AtlasBackend

[atlas]
username = my_username
password = my_password
host = host
port = 21000
```

Please make sure to have the *atlasclient* package installed.

3.19 Changelog

3.19.1 Airflow 1.10.3, 2019-04-09

3.19.1.1 New Feature

- [AIRFLOW-4232] Add `none_skipped` trigger rule (#5032)
- [AIRFLOW-3971] Add Google Cloud Natural Language operators (#4980)
- [AIRFLOW-4069] Add Opsgenie Alert Hook and Operator (#4903)
- [AIRFLOW-3552] Fix encoding issue in ImapAttachmentToS3Operator (#5040)
- [AIRFLOW-3552] Add ImapAttachmentToS3Operator (#4476)
- [AIRFLOW-1526] Add dingding hook and operator (#4895)
- [AIRFLOW-3490] Add BigQueryHook's Ability to Patch Table/View (#4299)
- [AIRFLOW-3918] Add ssh private-key support to git-sync for KubernetesExecutor (#4777)
- [AIRFLOW-3659] Create Google Cloud Transfer Service Operators (#4792)
- [AIRFLOW-3939] Add Google Cloud Translate operator (#4755)
- [AIRFLOW-3541] Add Avro logical type conversion to bigquery hook (#4553)
- [AIRFLOW-4106] instrument staving tasks in pool (#4927)
- [AIRFLOW-2568] Azure Container Instances operator (#4121)
- [AIRFLOW-4107] instrument executor (#4928)
- [AIRFLOW-4033] record stats of task duration (#4858)
- [AIRFLOW-3892] Create Redis pub sub sensor (#4712)
- [AIRFLOW-4124] add get_table and get_table_location in aws_glue_hook and tests (#4942)
- [AIRFLOW-1262] Adds missing docs for email configuration (#4557)
- [AIRFLOW-3701] Add Google Cloud Vision Product Search operators (#4665)
- [AIRFLOW-3766] Add support for kubernetes annotations (#4589)
- [AIRFLOW-3741] Add extra config to Oracle hook (#4584)
- [AIRFLOW-1262] Allow configuration of email alert subject and body (#2338)
- [AIRFLOW-2985] Operators for S3 object copying/deleting (#3823)
- [AIRFLOW-2993] s3_to_sftp and sftp_to_s3 operators (#3828)
- [AIRFLOW-3799] Add compose method to GoogleCloudStorageHook (#4641)
- [AIRFLOW-3218] add support for poking a whole DAG (#4058)
- [AIRFLOW-3315] Add ImapAttachmentSensor (#4161)
- [AIRFLOW-3556] Add cross join set dependency function (#4356)

3.19.1.2 Improvement

- [AIRFLOW-3823] Exclude branch's downstream tasks from the tasks to skip (#4666)
- [AIRFLOW-3274] Add run_as_user and fs_group options for Kubernetes (#4648)
- [AIRFLOW-4247] Template Region on the DataprocOperators (#5046)

- [AIRFLOW-4008] add envFrom for Kubernetes Executor (#4952)
- [AIRFLOW-3947] Flash msg for no DAG-level access error (#4767)
- [AIRFLOW-3287] Moving database clean-up code into the CoreTest.tearDown() (#4122)
- [AIRFLOW-4058] Name models test file to get automatically picked up (#4901)
- [AIRFLOW-3830] Remove DagBag from /dag_details (#4831)
- [AIRFLOW-3596] Clean up undefined template variables. (#4401)
- [AIRFLOW-3573] Remove DagStat table (#4378)
- [AIRFLOW-3623] Fix bugs in Download task logs (#5005)
- [AIRFLOW-4173] Improve SchedulerJob.process_file() (#4993)
- [AIRFLOW-3540] Warn if old airflow.cfg file is found (#5006)
- [AIRFLOW-4000] Return response when no file (#4822)
- [AIRFLOW-3383] Rotate fernet keys. (#4225)
- [AIRFLOW-3003] Pull the krb5 image instead of building (#3844)
- [AIRFLOW-3862] Check types with mypy. (#4685)
- [AIRFLOW-251] Add option SQL_ALCHEMY_SCHEMA parameter to specify schema for metadata (#4199)
- [AIRFLOW-1814] Temple PythonOperator {op_args,op_kwargs} fields (#4691)
- [AIRFLOW-3730] Standarization use of logs mechanisms (#4556)
- [AIRFLOW-3770] Validation of documentation on CI] (#4593)
- [AIRFLOW-3866] Run docker-compose pull silently in CI (#4688)
- [AIRFLOW-3685] Move licence header check (#4497)
- [AIRFLOW-3670] Add stages to Travis build (#4477)
- [AIRFLOW-3937] KubernetesPodOperator support for envFrom configMapRef and secretRef (#4772)
- [AIRFLOW-3408] Remove outdated info from Systemd Instructions (#4269)
- [AIRFLOW-3202] add missing documentation for AWS hooks/operator (#4048)
- [AIRFLOW-3908] Add more Google Cloud Vision operators (#4791)
- [AIRFLOW-2915] Add example DAG for GoogleCloudStorageToBigQueryOperator (#3763)
- [AIRFLOW-3062] Add Qubole in integration docs (#3946)
- [AIRFLOW-3288] Add SNS integration (#4123)
- [AIRFLOW-3148] Remove unnecessary arg “parameters” in RedshiftToS3Transfer (#3995)
- [AIRFLOW-3049] Add extra operations for Mongo hook (#3890)
- [AIRFLOW-3559] Add missing options to DatadogHook. (#4362)
- [AIRFLOW-1191] Simplify override of spark submit command. (#4360)
- [AIRFLOW-3155] Add ability to filter by a last modified time in GCS Operator (#4008)
- [AIRFLOW-2864] Fix docstrings for SubDagOperator (#3712)
- [AIRFLOW-4062] Improve docs on install extra package commands (#4966)
- [AIRFLOW-3743] Unify different methods of working out AIRFLOW_HOME (#4705)
- [AIRFLOW-4002] Option to open debugger on errors in *airflow test*. (#4828)
- [AIRFLOW-3997] Extend Variable.get so it can return None when var not found (#4819)

- [AIRFLOW-4009] Fix docstring issue in GCSToBQOperator (#4836)
- [AIRFLOW-3980] Unify logger (#4804)
- [AIRFLOW-4076] Correct port type of beeline_default in init_db (#4908)
- [AIRFLOW-4046] Add validations for poke_interval & timeout for Sensor (#4878)
- [AIRFLOW-3744] Abandon the use of obsolete aliases of methods (#4568)
- [AIRFLOW-3865] Add API endpoint to get python code of dag by id (#4687)
- [AIRFLOW-3516] Support to create k8 worker pods in batches (#4434)
- [AIRFLOW-2843] Add flag in ExternalTaskSensor to check if external DAG/task exists (#4547)
- [AIRFLOW-2224] Add support CSV files in MySqlToGoogleCloudStorageOperator (#4738)
- [AIRFLOW-3895] GoogleCloudStorageHook/Op create_bucket takes optional resource params (#4717)
- [AIRFLOW-3950] Improve AirflowSecurityManager.update_admin_perm_view (#4774)
- [AIRFLOW-4006] Make better use of Set in AirflowSecurityManager (#4833)
- [AIRFLOW-3917] Specify alternate kube config file/context when running out of cluster (#4859)
- [AIRFLOW-3911] Change Harvesting DAG parsing results to DEBUG log level (#4729)
- [AIRFLOW-3584] Use ORM DAGs for index view. (#4390)
- [AIRFLOW-2821] Refine Doc “Plugins” (#3664)
- [AIRFLOW-3561] Improve queries (#4368)
- [AIRFLOW-3600] Remove dagbag from trigger (#4407)
- [AIRFLOW-3713] Updated documentation for GCP optional project_id (#4541)
- [AIRFLOW-2767] Upgrade gunicorn to 19.5.0 to avoid moderate-severity CVE (#4795)
- [AIRFLOW-3795] provide_context param is now used (#4735)
- [AIRFLOW-4012] Upgrade tabulate to 0.8.3 (#4838)
- [AIRFLOW-3623] Support download logs by attempts from UI (#4425)
- [AIRFLOW-2715] Use region setting when launching Dataflow templates (#4139)
- [AIRFLOW-3932] Update unit tests and documentation for safe mode flag. (#4760)
- [AIRFLOW-3932] Optionally skip dag discovery heuristic. (#4746)
- [AIRFLOW-3258] K8S executor environment variables section. (#4627)
- [AIRFLOW-3931] set network, subnetwork when launching dataflow template (#4744)
- [AIRFLOW-4095] Add template_fields for S3CopyObjectOperator & S3DeleteObjectsOperator (#4920)
- [AIRFLOW-2798] Remove needless code from models.py
- [AIRFLOW-3731] Constrain mysqlclient to <1.4 (#4558)
- [AIRFLOW-3139] include parameters into log.info in SQL operators, if any (#3986)
- [AIRFLOW-3174] Refine Docstring for SQL Operators & Hooks (#4043)
- [AIRFLOW-3933] Fix various typos (#4747)
- [AIRFLOW-3905] Allow using “parameters” in SqlSensor (#4723)
- [AIRFLOW-2761] Parallelize enqueue in celery executor (#4234)
- [AIRFLOW-3540] Respect environment config when looking up config file. (#4340)
- [AIRFLOW-2156] Parallelize Celery Executor task state fetching (#3830)

- [AIRFLOW-3702] Add backfill option to run backwards (#4676)
- [AIRFLOW-3821] Add replicas logic to GCP SQL example DAG (#4662)
- [AIRFLOW-3547] Fixed Jinja templating in SparkSubmitOperator (#4347)
- [AIRFLOW-3647] Add archives config option to SparkSubmitOperator (#4467)
- [AIRFLOW-3802] Updated documentation for HiveServer2Hook (#4647)
- [AIRFLOW-3817] Corrected task ids returned by BranchPythonOperator to match the dummy operator ids (#4659)
- [AIRFLOW-3782] Clarify docs around celery worker_autoscale in default_airflow.cfg (#4609)
- [AIRFLOW-1945] Add Autoscale config for Celery workers (#3989)
- [AIRFLOW-3590] Change log message of executor exit status (#4616)
- [AIRFLOW-3591] Fix start date, end date, duration for rescheduled tasks (#4502)
- [AIRFLOW-3709] Validate *allowed_states* for ExternalTaskSensor (#4536)
- [AIRFLOW-3522] Add support for sending Slack attachments (#4332)
- [AIRFLOW-3569] Add “Trigger DAG” button in DAG page (/www only) (#4373)
- [AIRFLOW-3569] Add “Trigger DAG” button in DAG page (/www_rbac only) (#4373)
- [AIRFLOW-3044] Dataflow operators accept templated job_name param (#3887)
- [AIRFLOW-3023] Fix docstring datatypes
- [AIRFLOW-2928] Use uuid4 instead of uuid1 (#3779)
- [AIRFLOW-2988] Run specifically python2 for dataflow (#3826)
- [AIRFLOW-3697] Vendorize nvd3 and slugify (#4513)
- [AIRFLOW-3692] Remove ENV variables to avoid GPL (#4506)
- [AIRFLOW-3907] Upgrade flask and set cookie security flags. (#4725)
- [AIRFLOW-3698] Add documentation for AWS Connection (#4514)
- [AIRFLOW-3616][AIRFLOW-1215] Add aliases for schema with underscore (#4523)
- [AIRFLOW-3375] Support returning multiple tasks with BranchPythonOperator (#4215)
- [AIRFLOW-3742] Fix handling of “fallback” for AirflowConfigParsxer.getint/boolean (#4674)
- [AIRFLOW-3742] Respect the *fallback* arg in airflow.configuration.get (#4567)
- [AIRFLOW-3789] Fix flake8 3.7 errors. (#4617)
- [AIRFLOW-3602] Improve ImapHook handling of retrieving no attachments (#4475)
- [AIRFLOW-3631] Update flake8 and fix lint. (#4436)

3.19.1.3 Bug fixes

- [AIRFLOW-4248] Fix ‘FileExistsError’ makedirs race in file_processor_handler (#5047)
- [AIRFLOW-4240] State-changing actions should be POST requests (#5039)
- [AIRFLOW-4246] Flask-Oauthlib needs downstream dependencies pinning due to breaking changes (#5045)
- [AIRFLOW-3887] Downgrade dagre-d3 to 0.4.18 (#4713)
- [AIRFLOW-3419] Fix S3Hook.select_key on Python3 (#4970)
- [AIRFLOW-4127] Correct AzureContainerInstanceHook._get_instance_view’s return (#4945)

- [AIRFLOW-4172] Fix changes for driver class path option in Spark Submit (#4992)
- [AIRFLOW-3615] Preserve case of UNIX socket paths in Connections (#4591)
- [AIRFLOW-3417] ECSOperator: pass platformVersion only for FARGATE launch type (#4256)
- [AIRFLOW-3884] Fixing doc checker, no warnings allowed anymore and fixed the current… (#4702)
- [AIRFLOW-2652] implement / enhance baseOperator deepcopy
- [AIRFLOW-4001] Update docs about how to run tests (#4826)
- [AIRFLOW-3699] Speed up Flake8 (#4515)
- [AIRFLOW-4160] Fix redirecting of 'Trigger Dag' Button in DAG Page (#4982)
- [AIRFLOW-3650] Skip running on mysql for the flaky test (#4457)
- [AIRFLOW-3423] Fix mongo hook to work with anonymous access (#4258)
- [AIRFLOW-3982] Fix race condition in CI test (#4968)
- [AIRFLOW-3982] Update DagRun state based on its own tasks (#4808)
- [AIRFLOW-3737] Kubernetes executor cannot handle long dag/task names (#4636)
- [AIRFLOW-3945] Stop inserting row when permission views unchanged (#4764)
- [AIRFLOW-4123] Add Exception handling for _change_state method in K8 Executor (#4941)
- [AIRFLOW-3771] Minor refactor securityManager (#4594)
- [AIRFLOW-987] pass kerberos cli args keytab and principal to kerberos.run() (#4238)
- [AIRFLOW-3736] Allow int value in SqoopOperator.extra_import_options(#4906)
- [AIRFLOW-4063] Fix exception string in BigQueryHook [2/2] (#4902)
- [AIRFLOW-4063] Fix exception string in BigQueryHook (#4899)
- [AIRFLOW-4037] Log response in SimpleHttpOperator even if the response check fails
- [AIRFLOW-4044] The documentation of *query_params* in *BigQueryOperator* is wrong. (#4876)
- [AIRFLOW-4015] Make missing API endpoints available in classic mode
- [AIRFLOW-3153] Send DAG processing stats to statsd (#4748)
- [AIRFLOW-2966] Catch ApiException in the Kubernetes Executor (#4209)
- [AIRFLOW-4129] Escape HTML in generated tooltips (#4950)
- [AIRFLOW-4070] AirflowException -> log.warning for duplicate task dependencies (#4904)
- [AIRFLOW-4054] Fix assertEqualIgnoreMultipleSpaces util & add tests (#4886)
- [AIRFLOW-3239] Fix test recovery further (#4074)
- [AIRFLOW-4053] Fix KubePodOperator Xcom on Kube 1.13.0 (#4883)
- [AIRFLOW-2961] Refactor tests.BackfillJobTest.test_backfill_examples test (#3811)
- [AIRFLOW-3606] Fix Flake8 test & fix the Flake8 errors introduced since Flake8 test was broken (#4415)
- [AIRFLOW-3543] Fix deletion of DAG with rescheduled tasks (#4646)
- [AIRFLOW-2548] Output plugin import errors to web UI (#3930)
- [AIRFLOW-4019] Fix AWS Athena Sensor object has no attribute 'mode' (#4844)
- [AIRFLOW-3758] Fix circular import in WasbTaskHandler (#4601)
- [AIRFLOW-3706] Fix tooltip max-width by correcting ordering of CSS files (#4947)
- [AIRFLOW-4100] Correctly JSON escape data for tree/graph views (#4921)

- [AIRFLOW-3636] Fix a test introduced in #4425 (#4446)
- [AIRFLOW-3977] Add examples of trigger rules in doc (#4805)
- [AIRFLOW-2511] Fix improper failed session commit handling causing deadlocks (#4769)
- [AIRFLOW-3962] Added graceful handling for creation of dag_run of a dag which doesn't have any task (#4781)
- [AIRFLOW-3881] Correct to_csv row number (#4699)
- [AIRFLOW-3875] Simplify SlackWebhookHook code and change docstring (#4696)
- [AIRFLOW-3733] Don't raise NameError in HQL hook to_csv when no rows returned (#4560)
- [AIRFLOW-3734] Fix hql not run when partition is None (#4561)
- [AIRFLOW-3767] Correct bulk insert function (#4773)
- [AIRFLOW-4087] remove sudo in basetaskrunner on_finish (#4916)
- [AIRFLOW-3768] Escape search parameter in pagination controls (#4911)
- [AIRFLOW-4045] Fix hard-coded URLs in FAB-based UI (#4914)
- [AIRFLOW-3123] Use a stack for DAG context management (#3956)
- [AIRFLOW-3060] DAG context manager fails to exit properly in certain circumstances
- [AIRFLOW-3924] Fix try number in alert emails (#4741)
- [AIRFLOW-4083] Add tests for link generation utils (#4912)
- [AIRFLOW-2190] Send correct HTTP status for base_url not found (#4910)
- [AIRFLOW-4015] Add get_dag_runs GET endpoint to "classic" API (#4884)
- [AIRFLOW-3239] Enable existing CI tests (#4131)
- [AIRFLOW-1390] Update Alembic to 0.9 (#3935)
- [AIRFLOW-3885] Fix race condition in scheduler test (#4737)
- [AIRFLOW-3885] ~10x speed-up of SchedulerJobTest suite (#4730)
- [AIRFLOW-3780] Fix some incorrect when base_url is used (#4643)
- [AIRFLOW-3807] Fix Graph View Highlighting of Tasks (#4653)
- [AIRFLOW-3009] Import Hashable from collection.abc to fix Python 3.7 deprecation warning (#3849)
- [AIRFLOW-2231] Fix relativedelta DAG schedule_interval (#3174)
- [AIRFLOW-2641] Fix MySqlToHiveTransfer to handle MySQL DECIMAL correctly
- [AIRFLOW-3751] Option to allow malformed schemas for LDAP authentication (#4574)
- [AIRFLOW-2888] Add deprecation path for task_runner config change (#4851)
- [AIRFLOW-2930] Fix celery excecutor scheduler crash (#3784)
- [AIRFLOW-2888] Remove shell=True and bash from task launch (#3740)
- [AIRFLOW-3885] ~2.5x speed-up for backfill tests (#4731)
- [AIRFLOW-3885] ~20x speed-up of slowest unit test (#4726)
- [AIRFLOW-2508] Handle non string types in Operators templatized fields (#4292)
- [AIRFLOW-3792] Fix validation in BQ for useLegacySQL & queryParameters (#4626)
- [AIRFLOW-3749] Fix Edit Dag Run page when using RBAC (#4613)
- [AIRFLOW-3801] Fix DagBag collect dags invocation to prevent examples to be loaded (#4677)
- [AIRFLOW-3774] Register blueprints with RBAC web app (#4598)

- [AIRFLOW-3719] Handle StopIteration in CloudWatch logs retrieval (#4516)
- [AIRFLOW-3108] Define get_autocommit method for MsSqlHook (#4525)
- [AIRFLOW-3074] Add relevant ECS options to ECS operator. (#3908)
- [AIRFLOW-3353] Upgrade Redis client (#4834)
- [AIRFLOW-3250] Fix for Redis Hook for not authorised connection calls (#4090)
- [AIRFLOW-2009] Fix dataflow hook connection-id (#4563)
- [AIRFLOW-2190] Fix TypeError when returning 404 (#4596)
- [AIRFLOW-2876] Update Tenacity to 4.12 (#3723)
- [AIRFLOW-3923] Update flask-admin dependency to 1.5.3 to resolve security vulnerabilities from safety (#4739)
- [AIRFLOW-3683] Fix formatting of error message for invalid TriggerRule (#4490)
- [AIRFLOW-2787] Allow is_backfill to handle NULL DagRun.run_id (#3629)
- [AIRFLOW-3780] Fix some incorrect when base_url is used (#4643)
- [AIRFLOW-3639] Fix request creation in Jenkins Operator (#4450)
- [AIRFLOW-3779] Don’t install enum34 backport when not needed (#4620)
- [AIRFLOW-3079] Improve migration scripts to support MSSQL Server (#3964)
- [AIRFLOW-2735] Use equality, not identity, check for detecting AWS Batch failures[]
- [AIRFLOW-2706] AWS Batch Operator should use top-level job state to determine status
- [AIRFLOW-XXX] Fix typo in http_operator.py
- [AIRFLOW-XXX] Solve lodash security warning (#4820)
- [AIRFLOW-XXX] Pin version of tornado pulled in by Celery. (#4815)
- [AIRFLOW-XXX] Upgrade FAB to 1.12.3 (#4694)
- [AIRFLOW-XXX] Pin pinodb dependency (#4704)
- [AIRFLOW-XXX] Pin version of Pip in tests to work around pypa/pip#6163 (#4576)
- [AIRFLOW-XXX] Fix spark submit hook KeyError (#4578)
- [AIRFLOW-XXX] Pin psycopg2 due to breaking change (#5036)
- [AIRFLOW-XXX] Pin Sendgrid dep. (#5031)
- [AIRFLOW-XXX] Fix flaky test - test_execution_unlimited_parallelism (#4988)

3.19.1.4 Misc/Internal

- [AIRFLOW-4144] add description of is_delete_operator_pod (#4943)
- [AIRFLOW-3476][AIRFLOW-3477] Move Kube classes out of models.py (#4443)
- [AIRFLOW-3464] Move SkipMixin out of models.py (#4386)
- [AIRFLOW-3463] Move Log out of models.py (#4639)
- [AIRFLOW-3458] Move connection tests (#4680)
- [AIRFLOW-3461] Move TaskFail out of models.py (#4630)
- [AIRFLOW-3462] Move TaskReschedule out of models.py (#4618)
- [AIRFLOW-3474] Move SlaMiss out of models.py (#4608)
- [AIRFLOW-3475] Move ImportError out of models.py (#4383)

- [AIRFLOW-3459] Move DagPickle to separate file (#4374)
- [AIRFLOW-3925] Don't pull docker-images on pretest (#4740)
- [AIRFLOW-4154] Correct string formatting in jobs.py (#4972)
- [AIRFLOW-3458] Deprecation path for moving models.Connection
- [AIRFLOW-3458] Move models.Connection into separate file (#4335)
- [AIRFLOW-XXX] Remove old/non-test files that nose ignores (#4930)

3.19.1.5 Doc-only changes

- [AIRFLOW-3996] Add view source link to included fragments
- [AIRFLOW-3811] automatic generation of API Reference in docs (#4788)
- [AIRFLOW-3810] Remove duplicate autoclass directive (#4656)
- [AIRFLOW-XXX] Mention that statsd must be installed to gather metrics (#5038)
- [AIRFLOW-XXX] Add contents to cli (#4825)
- [AIRFLOW-XXX] fix check docs failure on CI (#4998)
- [AIRFLOW-XXX] Fix syntax docs errors (#4789)
- [AIRFLOW-XXX] Docs rendering improvement (#4684)
- [AIRFLOW-XXX] Automatically link Jira/GH on doc's changelog page (#4587)
- [AIRFLOW-XXX] Mention Oracle in the Extra Packages documentation (#4987)
- [AIRFLOW-XXX] Drop deprecated sudo option; use default docker compose on Travis. (#4732)
- [AIRFLOW-XXX] Update kubernetes.rst docs (#3875)
- [AIRFLOW-XXX] Improvements to formatted content in documentation (#4835)
- [AIRFLOW-XXX] Add Daniel to committer list (#4961)
- [AIRFLOW-XXX] Add Xiaodong Deng to committers list
- [AIRFLOW-XXX] Add history become ASF top level project (#4757)
- [AIRFLOW-XXX] Move out the examples from integration.rst (#4672)
- [AIRFLOW-XXX] Extract reverse proxy info to a separate file (#4657)
- [AIRFLOW-XXX] Reduction of the number of warnings in the documentation (#4585)
- [AIRFLOW-XXX] Fix GCS Operator docstrings (#4054)
- [AIRFLOW-XXX] Fix Docstrings in Hooks, Sensors & Operators (#4137)
- [AIRFLOW-XXX] Split guide for operators to multiple files (#4814)
- [AIRFLOW-XXX] Split connection guide to multiple files (#4824)
- [AIRFLOW-XXX] Remove almost all warnings from building docs (#4588)
- [AIRFLOW-XXX] Add backreference in docs between operator and integration (#4671)
- [AIRFLOW-XXX] Improve linking to classes (#4655)
- [AIRFLOW-XXX] Mock optional modules when building docs (#4586)
- [AIRFLOW-XXX] Update plugin macros documentation (#4971)
- [AIRFLOW-XXX] Add missing docstring for 'autodetect' in GCS to BQ Operator (#4979)
- [AIRFLOW-XXX] Add missing GCP operators to Docs (#4260)

- [AIRFLOW-XXX] Fixing the issue in Documentation (#3756)
- [AIRFLOW-XXX] Add Hint at user defined macros (#4885)
- [AIRFLOW-XXX] Correct schedule_interval in Scheduler docs (#4157)
- [AIRFLOW-XXX] Improve airflow-jira script to make RelManager's life easier (#4857)
- [AIRFLOW-XXX] Add missing class references to docs (#4644)
- [AIRFLOW-XXX] Fix typo (#4564)
- [AIRFLOW-XXX] Add a doc about fab security (#4595)
- [AIRFLOW-XXX] Speed up DagBagTest cases (#3974)
- [AIRFLOW-XXX] Reduction of the number of warnings in the documentation (#4585)

3.19.2 Airflow 1.10.2, 2019-01-19

3.19.2.1 New features

- [AIRFLOW-2658] Add GCP specific k8s pod operator (#3532)
- [AIRFLOW-2440] Google Cloud SQL import/export operator (#4251)
- [AIRFLOW-3212] Add AwsGlueCatalogPartitionSensor (#4112)
- [AIRFLOW-2750] Add subcommands to delete and list users
- [AIRFLOW-3480] Add GCP Spanner Database Operators (#4353)
- [AIRFLOW-3560] Add DayOfWeek Sensor (#4363)
- [AIRFLOW-3371] BigQueryHook's Ability to Create View (#4213)
- [AIRFLOW-3332] Add method to allow inserting rows into BQ table (#4179)
- [AIRFLOW-3055] add get_dataset and get_datasets_list to bigquery_hook (#3894)
- [AIRFLOW-2887] Added BigQueryCreateEmptyDatasetOperator and create_emty_dataset to bigquery_hook (#3876)
- [AIRFLOW-2758] Add a sensor for MongoDB
- [AIRFLOW-2640] Add Cassandra table sensor
- [AIRFLOW-3398] Google Cloud Spanner instance database query operator (#4314)
- [AIRFLOW-3310] Google Cloud Spanner deploy / delete operators (#4286)
- [AIRFLOW-3406] Implement an Azure CosmosDB operator (#4265)
- [AIRFLOW-3434] Allows creating intermediate dirs in SFTPOperator (#4270)
- [AIRFLOW-3345] Add Google Cloud Storage (GCS) operators for ACL (#4192)
- [AIRFLOW-3266] Add AWS Athena Hook and Operator (#4111)
- [AIRFLOW-3346] Add hook and operator for GCP transfer service (#4189)
- [AIRFLOW-2983] Add prev_ds_nodash and next_ds_nodash macro (#3821)
- [AIRFLOW-3403] Add AWS Athena Sensor (#4244)
- [AIRFLOW-3323] Support HTTP basic authentication for Airflow Flower (#4166)
- [AIRFLOW-3410] Add feature to allow Host Key Change for SSH Op (#4249)
- [AIRFLOW-3275] Add Google Cloud SQL Query operator (#4170)
- [AIRFLOW-2691] Manage JS dependencies via npm

- [AIRFLOW-2795] Oracle to Oracle Transfer Operator (#3639)
- [AIRFLOW-2596] Add Oracle to Azure Datalake Transfer Operator
- [AIRFLOW-3220] Add Instance Group Manager Operators for GCE (#4167)
- [AIRFLOW-2882] Add import and export for pool cli using JSON
- [AIRFLOW-2965] CLI tool to show the next execution datetime (#3834)
- [AIRFLOW-2874] Enables FAB's theme support (#3719)
- [AIRFLOW-3336] Add new TriggerRule for 0 upstream failures (#4182)

3.19.2.2 Improvements

- [AIRFLOW-3680] Consistency update in tests for All GCP-related operators (#4493)
- [AIRFLOW-3675] Use googlapiclient for google apis (#4484)
- [AIRFLOW-3205] Support multipart uploads to GCS (#4084)
- [AIRFLOW-2826] Add GoogleCloudKMSHook (#3677)
- [AIRFLOW-3676] Add required permission to CloudSQL export/import example (#4489)
- [AIRFLOW-3679] Added Google Cloud Base Hook to documentation (#4487)
- [AIRFLOW-3594] Unify different License Header
- [AIRFLOW-3197] Remove invalid parameter KeepJobFlowAliveWhenNoSteps in example DAG (#4404)
- [AIRFLOW-3504] Refine the functionality of "/health" endpoint (#4309)
- [AIRFLOW-3103][AIRFLOW-3147] Update flask-appbuilder (#3937)
- [AIRFLOW-3168] More resillient database use in CI (#4014)
- [AIRFLOW-3076] Remove preloading of MySQL testdata (#3911)
- [AIRFLOW-3035] Allow custom 'job_error_states' in dataproc ops (#3884)
- [AIRFLOW-3246] Make hmsclient optional in airflow.hooks.hive_hooks (#4080)
- [AIRFLOW-3059] Log how many rows are read from Postgres (#3905)
- [AIRFLOW-2463] Make task instance context available for hive queries
- [AIRFLOW-3190] Make flake8 compliant (#4035)
- [AIRFLOW-1998] Implemented DatabricksRunNowOperator for jobs/run-now … (#3813)
- [AIRFLOW-2267] Airflow DAG level access (#3197)
- [AIRFLOW-2359] Add set failed for DagRun and task in tree view (#3255)
- [AIRFLOW-3008] Move Kubernetes example DAGs to contrib
- [AIRFLOW-3402] Support global k8s affinity and toleration configs (#4247)
- [AIRFLOW-3610] Add region param for EMR jobflow creation (#4418)
- [AIRFLOW-3531] Fix test for GCS to GCS Transfer Hook (#4452)
- [AIRFLOW-3531] Add gcs to gcs transfer operator. (#4331)
- [AIRFLOW-3034]: Readme updates : Add Slack & Twitter, remove Gitter
- [AIRFLOW-3028] Update Text & Images in Readme.md
- [AIRFLOW-208] Add badge to show supported Python versions (#3839)
- [AIRFLOW-2238] Update PR tool to push directly to Github

- [AIRFLOW-2238] Flake8 fixes on dev/airflow-pr
- [AIRFLOW-2238] Update PR tool to remove outdated info (#3978)
- [AIRFLOW-3005] Replace 'Airbnb Airflow' with 'Apache Airflow' (#3845)
- [AIRFLOW-3150] Make execution_date templated in TriggerDagRunOperator (#4359)
- [AIRFLOW-1196][AIRFLOW-2399] Add templated field in TriggerDagRunOperator (#4228)
- [AIRFLOW-3340] Placeholder support in connections form (#4185)
- [AIRFLOW-3446] Add Google Cloud BigTable operators (#4354)
- [AIRFLOW-1921] Add support for https and user auth (#2879)
- [AIRFLOW-2770] Read *dags_in_image* config value as a boolean (#4319)
- [AIRFLOW-3022] Add volume mount to KubernetesExecutorConfig (#3855)
- [AIRFLOW-2917] Set AIRFLOW__CORE__SQL_ALCHEMY_CONN only when needed (#3766)
- [AIRFLOW-2712] Pass annotations to KubernetesExecutorConfig
- [AIRFLOW-461] Support autodetected schemas in BigQuery run_load (#3880)
- [AIRFLOW-2997] Support cluster fields in bigquery (#3838)
- [AIRFLOW-2916] Arg *verify* for AwsHook() & S3 sensors/operators (#3764)
- [AIRFLOW-491] Add feature to pass extra api configs to BQ Hook (#3733)
- [AIRFLOW-2889] Fix typos detected by github.com/client9/misspell (#3732)
- [AIRFLOW-850] Add a PythonSensor (#4349)
- [AIRFLOW-2747] Explicit re-schedule of sensors (#3596)
- [AIRFLOW-3392] Add index on dag_id in sla_miss table (#4235)
- [AIRFLOW-3001] Add index 'ti_dag_date' to taskinstance (#3885)
- [AIRFLOW-2861] Add index on log table (#3709)
- [AIRFLOW-3518] Performance fixes for topological_sort of Tasks (#4322)
- [AIRFLOW-3521] Fetch more than 50 items in *airflow-jira compare* script (#4300)
- [AIRFLOW-1919] Add option to query for DAG runs given a DAG ID
- [AIRFLOW-3444] Explicitly set transfer operator description. (#4279)
- [AIRFLOW-3411] Add OpenFaaS hook (#4267)
- [AIRFLOW-2785] Add context manager entry points to mongoHook
- [AIRFLOW-2524] Add SageMaker doc to AWS integration section (#4278)
- [AIRFLOW-3479] Keeps records in Log Table when DAG is deleted (#4287)
- [AIRFLOW-2948] Arg check & better doc - SSHOperator & SFTPOperator (#3793)
- [AIRFLOW-2245] Add remote_host of SSH/SFTP operator as templated field (#3765)
- [AIRFLOW-2670] Update SSH Operator's Hook to respect timeout (#3666)
- [AIRFLOW-3380] Add metrics documentation (#4219)
- [AIRFLOW-3361] Log the task_id in the PendingDeprecationWarning from BaseOperator (#4030)
- [AIRFLOW-3213] Create ADLS to GCS operator (#4134)
- [AIRFLOW-3395] added the REST API endpoints to the doc (#4236)
- [AIRFLOW-3294] Update connections form and integration docs (#4129)

- [AIRFLOW-3236] Create AzureDataLakeStorageListOperator (#4094)
- [AIRFLOW-3062] Add Qubole in integration docs (#3946)
- [AIRFLOW-3306] Disable flask-sqlalchemy modification tracking. (#4146)
- [AIRFLOW-2867] Refactor Code to conform standards (#3714)
- [AIRFLOW-2753] Add dataproc_job_id instance var holding actual DP jobId
- [AIRFLOW-3132] Enable specifying auto_remove option for DockerOperator (#3977)
- [AIRFLOW-2731] Raise psutil restriction to <6.0.0
- [AIRFLOW-3384] Allow higher versions of Sqlalchemy and Jinja2 (#4227)
- [Airflow-2760] Decouple DAG parsing loop from scheduler loop (#3873)
- [AIRFLOW-3004] Add config disabling scheduler cron (#3899)
- [AIRFLOW-3175] Fix docstring format in airflow/jobs.py (#4025)
- [AIRFLOW-3589] Visualize reschedule state in all views (#4408)
- [AIRFLOW-2698] Simplify Kerberos code (#3563)
- [AIRFLOW-2499] Dockerise CI pipeline (#3393)
- [AIRFLOW-3432] Add test for feature "Delete DAG in UI" (#4266)
- [AIRFLOW-3301] Update DockerOperator CI test for PR #3977 (#4138)
- [AIRFLOW-3478] Make sure that the session is closed
- [AIRFLOW-3687] Add missing @apply_defaults decorators (#4498)
- [AIRFLOW-3691] Update notice to 2019 (#4503)
- [AIRFLOW-3689] Update pop-up message when deleting DAG in RBAC UI (#4505)
- [AIRFLOW-2801] Skip test_mark_success_no_kill in PostgreSQL on CI (#3642)
- [AIRFLOW-3693] Replace psycopg2-binary by psycopg2 (#4508)
- [AIRFLOW-3700] Change the lowest allowed version of "requests" (#4517)
- [AIRFLOW-3704] Support SSL Protection When Redis is Used as Broker for CeleryExecutor (#4521)
- [AIRFLOW-3681] All GCP operators have now optional GCP Project ID (#4500)
- [Airflow 2782] Upgrades Dagre D3 version to latest possible
- [Airflow 2783] Implement eslint for JS code check (#3641)
- [AIRFLOW-2805] Display multiple timezones on UI (#3687)
- [AIRFLOW-3302] Small CSS fixes (#4140)
- [Airflow-2766] Respect shared datetime across tabs
- [AIRFLOW-2776] Compress tree view JSON
- [AIRFLOW-2407] Use feature detection for reload() (#3298)
- [AIRFLOW-3452] Removed an unused/dangerous display-none (#4295)
- [AIRFLOW-3348] Update run statistics on dag refresh (#4197)
- [AIRFLOW-3125] Monitor Task Instances creation rates (#3966)

3.19.2.3 Bug fixes

- [AIRFLOW-3191] Fix not being able to specify execution_date when creating dagrun (#4037)
- [AIRFLOW-3657] Fix zendesk integration (#4466)
- [AIRFLOW-3605] Load plugins from entry_points (#4412)
- [AIRFLOW-3646] Rename plugins_manager.py to test_xx to trigger tests (#4464)
- [AIRFLOW-3655] Escape links generated in model views (#4463)
- [AIRFLOW-3662] Add dependency for Enum (#4468)
- [AIRFLOW-3630] Cleanup of GCP Cloud SQL Connection (#4451)
- [AIRFLOW-1837] Respect task start_date when different from dag's (#4010)
- [AIRFLOW-2829] Brush up the CI script for minikube
- [AIRFLOW-3519] Fix example http operator (#4455)
- [AIRFLOW-2811] Fix scheduler_ops_metrics.py to work (#3653)
- [AIRFLOW-2751] add job properties update in hive to druid operator.
- [AIRFLOW-2918] Remove unused imports
- [AIRFLOW-2918] Fix Flake8 violations (#3931)
- [AIRFLOW-2771] Add except type to broad S3Hook try catch clauses
- [AIRFLOW-2918] Fix Flake8 violations (#3772)
- [AIRFLOW-2099] Handle getsource() calls gracefully
- [AIRFLOW-3397] Fix integrety error in rbac AirflowSecurityManager (#4305)
- [AIRFLOW-3281] Fix Kubernetes operator with git-sync (#3770)
- [AIRFLOW-2615] Limit DAGs parsing to once only
- [AIRFLOW-2952] Fix Kubernetes CI (#3922)
- [AIRFLOW-2933] Enable Codecov on Docker-CI Build (#3780)
- [AIRFLOW-2082] Resolve a bug in adding password_auth to api as auth method (#4343)
- [AIRFLOW-3612] Remove incubation/incubator mention (#4419)
- [AIRFLOW-3581] Fix next_ds/prev_ds semantics for manual runs (#4385)
- [AIRFLOW-3527] Update Cloud SQL Proxy to have shorter path for UNIX socket (#4350)
- [AIRFLOW-3316] For gcs_to_bq: add missing init of schema_fields var (#4430)
- [AIRFLOW-3583] Fix AirflowException import (#4389)
- [AIRFLOW-3578] Fix Type Error for BigQueryOperator (#4384)
- [AIRFLOW-2755] Added *kubernetes.worker_dags_folder* configuration (#3612)
- [AIRFLOW-2655] Fix inconsistency of default config of kubernetes worker
- [AIRFLOW-2645][AIRFLOW-2617] Add worker_container_image_pull_policy
- [AIRFLOW-2661] fix config dags_volume_subpath and logs_volume_subpath
- [AIRFLOW-3550] Standardize GKE hook (#4364)
- [AIRFLOW-2863] Fix GKEClusterHook catching wrong exception (#3711)
- [AIRFLOW-2939][AIRFLOW-3568] Fix TypeError in GCSToS3Op & S3ToGCSOp (#4371)
- [AIRFLOW-3327] Add support for location in BigQueryHook (#4324)

- [AIRFLOW-3438] Fix default values in BigQuery Hook & BigQueryOperator (…
- [AIRFLOW-3355] Fix BigQueryCursor.execute to work with Python3 (#4198)
- [AIRFLOW-3447] Add 2 options for ts_nodash Macro (#4323)
- [AIRFLOW-1552] Airflow Filter_by_owner not working with password_auth (#4276)
- [AIRFLOW-3484] Fix Over-logging in the k8s executor (#4296)
- [AIRFLOW-3309] Add MongoDB connection (#4154)
- [AIRFLOW-3414] Fix reload_module in DagFileProcessorAgent (#4253)
- [AIRFLOW-1252] API accept JSON when invoking a trigger dag (#2334)
- [AIRFLOW-3425] Fix setting default scope in hook (#4261)
- [AIRFLOW-3416] Fixes Python 3 compatibility with CloudSqlQueryOperator (#4254)
- [AIRFLOW-3263] Ignore exception when 'run' kills already killed job (#4108)
- [AIRFLOW-3264] URL decoding when parsing URI for connection (#4109)
- [AIRFLOW-3365][AIRFLOW-3366] Allow celery_broker_transport_options to be set with environment variables (#4211)
- [AIRFLOW-2642] fix wrong value git-sync initcontainer env GIT_SYNC_ROOT (#3519)
- [AIRFLOW-3353] Pin redis version (#4195)
- [AIRFLOW-3251] KubernetesPodOperator now uses 'image_pull_secrets' argument when creating Pods (#4188)
- [AIRFLOW-2705] Move class-level moto decorator to method-level
- [AIRFLOW-3233] Fix deletion of DAGs in the UI (#4069)
- [AIRFLOW-2908] Allow retries with KubernetesExecutor. (#3758)
- [AIRFLOW-1561] Fix scheduler to pick up example DAGs without other DAGs (#2635)
- [AIRFLOW-3352] Fix expose_config not honoured on RBAC UI (#4194)
- [AIRFLOW-3592] Fix logs when task is in rescheduled state (#4492)
- [AIRFLOW-3634] Fix GCP Spanner Test (#4440)
- [AIRFLOW-XXX] Fix PythonVirtualenvOperator tests (#3968)
- [AIRFLOW-3239] Fix/refine tests for api/common/experimental/ (#4255)
- [AIRFLOW-2951] Update dag_run table end_date when state change (#3798)
- [AIRFLOW-2756] Fix bug in set DAG run state workflow (#3606)
- [AIRFLOW-3690] Fix bug to set state of a task for manually-triggered DAGs (#4504)
- [AIRFLOW-3319] KubernetsExecutor: Need in try_number in labels if getting them later (#4163)
- [AIRFLOW-3724] Fix the broken refresh button on Graph View in RBAC UI
- [AIRFLOW-3732] Fix issue when trying to edit connection in RBAC UI
- [AIRFLOW-2866] Fix missing CSRF token head when using RBAC UI (#3804)
- [AIRFLOW-3259] Fix internal server error when displaying charts (#4114)
- [AIRFLOW-3271] Fix issue with persistence of RBAC Permissions modified via UI (#4118)
- [AIRFLOW-3141] Handle duration View for missing dag (#3984)
- [AIRFLOW-2766] Respect shared datetime across tabs
- [AIRFLOW-1413] Fix FTPSensor failing on error message with unexpected (#2450)

- [AIRFLOW-3378] KubernetesPodOperator does not delete on timeout failure (#4218)
- [AIRFLOW-3245] Fix list processing in resolve_template_files (#4086)
- [AIRFLOW-2703] Catch transient DB exceptions from scheduler's heartbeat it does not crash (#3650)
- [AIRFLOW-1298] Clear UPSTREAM_FAILED using the clean cli (#3886)

3.19.2.4 Doc-only changes

- [AIRFLOW-XXX] GCP operators documentation clarifications (#4273)
- [AIRFLOW-XXX] Docs: Fix paths to GCS transfer operator (#4479)
- [AIRFLOW-XXX] Add missing GCP operators to Docs (#4260)
- [AIRFLOW-XXX] Fix Docstrings for Operators (#3820)
- [AIRFLOW-XXX] Fix inconsistent comment in example_python_operator.py (#4337)
- [AIRFLOW-XXX] Fix incorrect parameter in SFTPOperator example (#4344)
- [AIRFLOW-XXX] Add missing remote logging field (#4333)
- [AIRFLOW-XXX] Revise template variables documentation (#4172)
- [AIRFLOW-XXX] Fix typo in docstring of gcs_to_bq (#3833)
- [AIRFLOW-XXX] Fix display of SageMaker operators/hook docs (#4263)
- [AIRFLOW-XXX] Better instructions for airflow flower (#4214)
- [AIRFLOW-XXX] Make pip install commands consistent (#3752)
- [AIRFLOW-XXX] Add *BigQueryGetDataOperator* to Integration Docs (#4063)
- [AIRFLOW-XXX] Don't spam test logs with "bad cron expression" messages (#3973)
- [AIRFLOW-XXX] Update committer list based on latest TLP discussion (#4427)
- [AIRFLOW-XXX] Fix incorrect statement in contributing guide (#4104)
- [AIRFLOW-XXX] Fix Broken Link in CONTRIBUTING.md
- [AIRFLOW-XXX] Update Contributing Guide - Git Hooks (#4120)
- [AIRFLOW-3426] Correct Python Version Documentation Reference (#4259)
- [AIRFLOW-2663] Add instructions to install SSH dependencies
- [AIRFLOW-XXX] Clean up installation extra packages table (#3750)
- [AIRFLOW-XXX] Remove redundant space in Kerberos (#3866)
- [AIRFLOW-3086] Add extras group for google auth to setup.py (#3917)
- [AIRFLOW-XXX] Add Kubernetes Dependency in Extra Packages Doc (#4281)
- [AIRFLOW-3696] Add Version info to Airflow Documentation (#4512)
- [AIRFLOW-XXX] Correct Typo in sensor's exception (#4545)
- [AIRFLOW-XXX] Fix a typo of config (#4544)
- [AIRFLOW-XXX] Fix BashOperator Docstring (#4052)
- [AIRFLOW-3018] Fix Minor issues in Documentation
- [AIRFLOW-XXX] Fix Minor issues with Azure Cosmos Operator (#4289)
- [AIRFLOW-3382] Fix incorrect docstring in DatastoreHook (#4222)
- [AIRFLOW-XXX] Fix copy&paste mistake (#4212)

- [AIRFLOW-3260] Correct misleading BigQuery error (#4098)
- [AIRFLOW-XXX] Fix Typo in SFTPOperator docstring (#4016)
- [AIRFLOW-XXX] Fixing the issue in Documentation (#3998)
- [AIRFLOW-XXX] Fix undocumented params in S3_hook
- [AIRFLOW-XXX] Fix SlackWebhookOperator execute method comment (#3963)
- [AIRFLOW-3070] Refine web UI authentication-related docs (#3863)

3.19.3 Airflow 1.10.1, 2018-11-13

3.19.3.1 New features

- [AIRFLOW-2524] Airflow integration with AWS Sagemaker
- [AIRFLOW-2657] Add ability to delete DAG from web ui
- [AIRFLOW-2780] Adds IMAP Hook to interact with a mail server
- [AIRFLOW-2794] Add delete support for Azure blob
- [AIRFLOW-2912] Add operators for Google Cloud Functions
- [AIRFLOW-2974] Add Start/Restart/Terminate methods Databricks Hook
- [AIRFLOW-2989] No Parameter to change bootDiskType for DataprocClusterCreateOperator
- [AIRFLOW-3078] Basic operators for Google Compute Engine
- [AIRFLOW-3147] Update Flask-AppBuilder version
- [AIRFLOW-3231] Basic operators for Google Cloud SQL (deploy / patch / delete)
- [AIRFLOW-3276] Google Cloud SQL database create / patch / delete operators

3.19.3.2 Improvements

- [AIRFLOW-393] Add progress callbacks for FTP downloads
- [AIRFLOW-520] Show Airflow version on web page
- [AIRFLOW-843] Exceptions now available in context during on_failure_callback
- [AIRFLOW-2476] Update tabulate dependency to v0.8.2
- [AIRFLOW-2592] Bump Bleach dependency
- [AIRFLOW-2622] Add "confirm=False" option to SFTPOperator
- [AIRFLOW-2662] support affinity & nodeSelector policies for kubernetes executor/operator
- [AIRFLOW-2709] Improve error handling in Databricks hook
- [AIRFLOW-2723] Update lxml dependency to >= 4.0.
- [AIRFLOW-2763] No precheck mechanism in place during worker initialisation for the connection to metadata database
- [AIRFLOW-2789] Add ability to create single node cluster to DataprocClusterCreateOperator
- [AIRFLOW-2797] Add ability to create Google Dataproc cluster with custom image
- [AIRFLOW-2854] kubernetes_pod_operator add more configuration items
- [AIRFLOW-2855] Need to Check Validity of Cron Expression When Process DAG File/Zip File

- [AIRFLOW-2904] Clean an unnecessary line in airflow/executors/celery_executor.py
- [AIRFLOW-2921] A trivial incorrectness in CeleryExecutor()
- [AIRFLOW-2922] Potential deal-lock bug in CeleryExecutor()
- [AIRFLOW-2932] GoogleCloudStorageHook - allow compression of file
- [AIRFLOW-2949] Syntax Highlight for Single Quote
- [AIRFLOW-2951] dag_run end_date Null after a dag is finished
- [AIRFLOW-2956] Kubernetes tolerations for pod operator
- [AIRFLOW-2997] Support for clustered tables in Bigquery hooks/operators
- [AIRFLOW-3006] Fix error when schedule_interval="None"
- [AIRFLOW-3008] Move Kubernetes related example DAGs to contrib/example_dags
- [AIRFLOW-3025] Allow to specify dns and dns-search parameters for DockerOperator
- [AIRFLOW-3067] (www_rbac) Flask flash messages are not displayed properly (no background color)
- [AIRFLOW-3069] Decode output of S3 file transform operator
- [AIRFLOW-3072] Assign permission get_logs_with_metadata to viewer role
- [AIRFLOW-3090] INFO logs are too verbose
- [AIRFLOW-3103] Update Flask-Login
- [AIRFLOW-3112] Align SFTP hook with SSH hook
- [AIRFLOW-3119] Enable loglevel on celery worker and inherit from airflow.cfg
- [AIRFLOW-3137] Make ProxyFix middleware optional
- [AIRFLOW-3173] Add _cmd options for more password config options
- [AIRFLOW-3177] Change scheduler_heartbeat metric from gauge to counter
- [AIRFLOW-3193] Pin docker requirement version to v3
- [AIRFLOW-3195] Druid Hook: Log ingestion spec and task id
- [AIRFLOW-3197] EMR Hook is missing some parameters to valid on the AWS API
- [AIRFLOW-3232] Make documentation for GCF Functions operator more readable
- [AIRFLOW-3262] Can't get log containing Response when using SimpleHttpOperator
- [AIRFLOW-3265] Add support for "unix_socket" in connection extra for Mysql Hook

3.19.3.3 Doc-only changes

- [AIRFLOW-1441] Tutorial Inconsistencies Between Example Pipeline Definition and Recap
- [AIRFLOW-2682] Add how-to guide(s) for how to use basic operators like BashOperator and PythonOperator
- [AIRFLOW-3104] .airflowignore feature is not mentioned at all in documentation
- [AIRFLOW-3237] Refactor example DAGs
- [AIRFLOW-3187] Update airflow.gif file with a slower version
- [AIRFLOW-3159] Update Airflow documentation on GCP Logging
- [AIRFLOW-3030] Command Line docs incorrect subdir
- [AIRFLOW-2990] Docstrings for Hooks/Operators are in incorrect format
- [AIRFLOW-3127] Celery SSL Documentation is out-dated

- [AIRFLOW-2779] Add license headers to doc files
- [AIRFLOW-2779] Add project version to license

3.19.3.4 Bug fixes

- [AIRFLOW-839] docker_operator.py attempts to log status key without first checking existence
- [AIRFLOW-1104] Concurrency check in scheduler should count queued tasks as well as running
- [AIRFLOW-1163] Add support for x-forwarded-* headers to support access behind AWS ELB
- [AIRFLOW-1195] Cleared tasks in SubDagOperator do not trigger Parent dag_runs
- [AIRFLOW-1508] Skipped state not part of State.task_states
- [AIRFLOW-1762] Use key_file in SSHHook.create_tunnel()
- [AIRFLOW-1837] Differing start_dates on tasks not respected by scheduler.
- [AIRFLOW-1874] Support standard SQL in Check, ValueCheck and IntervalCheck BigQuery operators
- [AIRFLOW-1917] print() from python operators end up with extra new line
- [AIRFLOW-1970] Database cannot be initialized if an invalid fernet key is provided
- [AIRFLOW-2145] Deadlock after clearing a running task
- [AIRFLOW-2216] Cannot specify a profile for AWS Hook to load with s3 config file
- [AIRFLOW-2574] initdb fails when mysql password contains percent sign
- [AIRFLOW-2707] Error accessing log files from web UI
- [AIRFLOW-2716] Replace new Python 3.7 keywords
- [AIRFLOW-2744] RBAC app doesn't integrate plugins (blueprints etc)
- [AIRFLOW-2772] BigQuery hook does not allow specifying both the partition field name and table name at the same time
- [AIRFLOW-2778] Bad Import in collect_dag in DagBag
- [AIRFLOW-2786] Variables view fails to render if a variable has an empty key
- [AIRFLOW-2799] Filtering UI objects by datetime is broken
- [AIRFLOW-2800] Remove airflow/ low-hanging linting errors
- [AIRFLOW-2825] S3ToHiveTransfer operator may not may able to handle GZIP file with uppercase ext in S3
- [AIRFLOW-2848] dag_id is missing in metadata table "job" for LocalTaskJob
- [AIRFLOW-2860] DruidHook: time variable is not updated correctly when checking for timeout
- [AIRFLOW-2865] Race condition between on_success_callback and LocalTaskJob's cleanup
- [AIRFLOW-2893] Stuck dataflow job due to jobName mismatch.
- [AIRFLOW-2895] Prevent scheduler from spamming heartbeats/logs
- [AIRFLOW-2900] Code not visible for Packaged DAGs
- [AIRFLOW-2905] Switch to regional dataflow job service.
- [AIRFLOW-2907] Sendgrid - Attachments - ERROR - Object of type 'bytes' is not JSON serializable
- [AIRFLOW-2938] Invalid 'extra' field in connection can raise an AttributeError when attempting to edit
- [AIRFLOW-2979] Deprecated Celery Option not in Options list
- [AIRFLOW-2981] TypeError in dataflow operators when using GCS jar or py_file

- [AIRFLOW-2984] Cannot convert naive_datetime when task has a naive start_date/end_date
- [AIRFLOW-2994] flatten_results in BigQueryOperator/BigQueryHook should default to None
- [AIRFLOW-3002] ValueError in dataflow operators when using GCS jar or py_file
- [AIRFLOW-3012] Email on sla miss is send only to first address on the list
- [AIRFLOW-3046] ECS Operator mistakenly reports success when task is killed due to EC2 host termination
- [AIRFLOW-3064] No output from *airflow test* due to default logging config
- [AIRFLOW-3072] Only admin can view logs in RBAC UI
- [AIRFLOW-3079] Improve initdb to support MSSQL Server
- [AIRFLOW-3089] Google auth doesn't work under http
- [AIRFLOW-3099] Errors raised when some blocs are missing in airflow.cfg
- [AIRFLOW-3109] Default user permission should contain 'can_clear'
- [AIRFLOW-3111] Confusing comments and instructions for log templates in UPDATING.md and default_airflow.cfg
- [AIRFLOW-3124] Broken webserver debug mode (RBAC)
- [AIRFLOW-3136] Scheduler Failing the Task retries run while processing Executor Events
- [AIRFLOW-3138] Migration cc1e65623dc7 creates issues with postgres
- [AIRFLOW-3161] Log Url link does not link to task instance logs in RBAC UI
- [AIRFLOW-3162] HttpHook fails to parse URL when port is specified
- [AIRFLOW-3183] Potential Bug in utils/dag_processing/DagFileProcessorManager.max_runs_reached()
- [AIRFLOW-3203] Bugs in DockerOperator & Some operator test scripts were named incorrectly
- [AIRFLOW-3238] Dags, removed from the filesystem, are not deactivated on initdb
- [AIRFLOW-3268] Cannot pass SSL dictionary to mysql connection via URL
- [AIRFLOW-3277] Invalid timezone transition handling for cron schedules
- [AIRFLOW-3295] Require encryption in DaskExecutor when certificates are configured.
- [AIRFLOW-3297] EmrStepSensor marks cancelled step as successful

3.19.4 Airflow 1.10.0, 2018-08-03

- [AIRFLOW-2870] Use abstract TaskInstance for migration
- [AIRFLOW-2859] Implement own UtcDateTime (#3708)
- [AIRFLOW-2140] Don't require kubernetes for the SparkSubmit hook
- [AIRFLOW-2869] Remove smart quote from default config
- [AIRFLOW-2857] Fix Read the Docs env
- [AIRFLOW-2817] Force explicit choice on GPL dependency
- [AIRFLOW-2716] Replace async and await py3.7 keywords
- [AIRFLOW-2810] Fix typo in Xcom model timestamp
- [AIRFLOW-2710] Clarify fernet key value in documentation
- [AIRFLOW-2606] Fix DB schema and SQLAlchemy model
- [AIRFLOW-2646] Fix setup.py not to install snakebite on Python3

- [AIRFLOW-2604] Add index to task_fail
- [AIRFLOW-2650] Mark SchedulerJob as succeed when hitting Ctrl-c
- [AIRFLOW-2678] Fix db schema unit test to remove checking fab models
- [AIRFLOW-2624] Fix webserver login as anonymous
- [AIRFLOW-2654] Fix incorret URL on refresh in Graph View of FAB UI
- [AIRFLOW-2668] Handle missing optional cryptography dependency
- [AIRFLOW-2681] Include last dag run of externally triggered DAGs in UI.
- [AIRFLOW-1840] Support back-compat on old celery config
- [AIRFLOW-2612][AIRFLOW-2534] Clean up Hive-related tests
- [AIRFLOW-2608] Implements/Standardize custom exceptions for experimental APIs
- [AIRFLOW-2607] Fix failing TestLocalClient
- [AIRFLOW-2638] dbapi_hook: support REPLACE INTO
- [AIRFLOW-2542][AIRFLOW-1790] Rename AWS Batch Operator queue to job_queue
- [AIRFLOW-2567] Extract result from the kubernetes pod as Xcom
- [AIRFLOW-XXX] Adding REA Group to readme
- [AIRFLOW-2601] Allow user to specify k8s config
- [AIRFLOW-2559] Azure Fileshare hook
- [AIRFLOW-1786] Enforce correct behavior for soft-fail sensors
- [AIRFLOW-2355] Airflow trigger tag parameters in subdag
- [AIRFLOW-2613] Fix Airflow searching .zip bug
- [AIRFLOW-2627] Add a sensor for Cassandra
- [AIRFLOW-2634][AIRFLOW-2534] Remove dependency for impyla
- [AIRFLOW-2611] Fix wrong dag volume mount path for kubernetes executor
- [AIRFLOW-2562] Add Google Kubernetes Engine Operators
- [AIRFLOW-2630] Fix classname in test_sql_sensor.py
- [AIRFLOW-2534] Fix bug in HiveServer2Hook
- [AIRFLOW-2586] Stop getting AIRFLOW_HOME value from config file in bash operator
- [AIRFLOW-2605] Fix autocommit for MySqlHook
- [AIRFLOW-2539][AIRFLOW-2359] Move remaining log config to configuration file
- [AIRFLOW-1656] Tree view dags query changed
- [AIRFLOW-2617] add imagePullPolicy config for kubernetes executor
- [AIRFLOW-2429] Fix security/task/sensors/ti_deps folders flake8 error
- [AIRFLOW-2550] Implements API endpoint to list DAG runs
- [AIRFLOW-2512][AIRFLOW-2522] Use google-auth instead of oauth2client
- [AIRFLOW-2429] Fix operators folder flake8 error
- [AIRFLOW-2585] Fix several bugs in CassandraHook and CassandraToGCSOperator
- [AIRFLOW-2597] Restore original dbapi.run() behavior
- [AIRFLOW-2590] Fix commit in DbApiHook.run() for no-autocommit DB

- [AIRFLOW-1115] fix github oauth api URL
- [AIRFLOW-2587] Add TIMESTAMP type mapping to MySqlToHiveTransfer
- [AIRFLOW-2591][AIRFLOW-2581] Set default value of autocommit to False in DbApiHook.run()
- [AIRFLOW-59] Implement bulk_dump and bulk_load for the Postgres hook
- [AIRFLOW-2533] Fix path to DAG's on kubernetes executor workers
- [AIRFLOW-2581] RFLOW-2581] Fix DbApiHook autocommit
- [AIRFLOW-2578] Add option to use proxies in JiraHook
- [AIRFLOW-2575] Make gcs to gcs operator work with large files
- [AIRFLOW-437] Send TI context in kill zombies
- [AIRFLOW-2566] Change backfill to rerun failed tasks
- [AIRFLOW-1021] Fix double login for new users with LDAP
- [AIRFLOW-XXX] Typo fix
- [AIRFLOW-2561] Fix typo in EmailOperator
- [AIRFLOW-2573] Cast BigQuery TIMESTAMP field to float
- [AIRFLOW-2560] Adding support for internalIpOnly to DataprocClusterCreateOperator
- [AIRFLOW-2565] templatize cluster_label
- [AIRFLOW-83] add mongo hook and operator
- [AIRFLOW-2558] Clear task/dag is clearing all executions
- [AIRFLOW-XXX] Fix doc typos
- [AIRFLOW-2513] Change *bql* to *sql* for BigQuery Hooks & Ops
- [AIRFLOW-2557] Fix pagination for s3
- [AIRFLOW-2545] Eliminate DeprecationWarning
- [AIRFLOW-2500] Fix MySqlToHiveTransfer to transfer unsigned type properly
- [AIRFLOW-2462] Change PasswordUser setter to correct syntax
- [AIRFLOW-2525] Fix a bug introduced by commit dabf1b9
- [AIRFLOW-2553] Add webserver.pid to .gitignore
- [AIRFLOW-1863][AIRFLOW-2529] Add dag run selection widgets to gantt view
- [AIRFLOW-2504] Log username correctly and add extra to search columns
- [AIRFLOW-2551] Encode binary data with base64 standard rather than base64 url
- [AIRFLOW-2537] Add reset-dagrun option to backfill command
- [AIRFLOW-2526] dag_run.conf can override params
- [AIRFLOW-2544][AIRFLOW-1967] Guard against next major release of Celery, Flower
- [AIRFLOW-XXX] Add Yieldr to who is using airflow
- [AIRFLOW-2547] Describe how to run tests using Docker
- [AIRFLOW-2538] Update faq doc on how to reduce airflow scheduler latency
- [AIRFLOW-2529] Improve graph view performance and usability
- [AIRFLOW-2517] backfill support passing key values through CLI
- [AIRFLOW-2532] Support logs_volume_subpath for KubernetesExecutor

- [AIRFLOW-2466] consider task_id in _change_state_for_tis_without_dagrun
- [AIRFLOW-2519] Fix CeleryExecutor with SQLAlchemy
- [AIRFLOW-2402] Fix RBAC task log
- [AIRFLOW-XXX] Add M4U to user list
- [AIRFLOW-2536] docs about how to deal with airflow initdb failure
- [AIRFLOW-2530] KubernetesOperator supports multiple clusters
- [AIRFLOW-1499] Eliminate duplicate and unneeded code
- [AIRFLOW-2521] backfill - make variable name and logging messages more acurate
- [AIRFLOW-2429] Fix hook, macros folder flake8 error
- [Airflow-XXX] add Prime to company list
- [AIRFLOW-2525] Fix PostgresHook.copy_expert to work with "COPY FROM"
- [AIRFLOW-2515] Add dependency on thrift_sasl to hive extra
- [AIRFLOW-2523] Add how-to for managing GCP connections
- [AIRFLOW-2510] Introduce new macros: prev_ds and next_ds
- [AIRFLOW-1730] Unpickle value of XCom queried from DB
- [AIRFLOW-2518] Fix broken ToC links in integration.rst
- [AIRFLOW-1472] Fix SLA misses triggering on skipped tasks.
- [AIRFLOW-2520] CLI - make backfill less verbose
- [AIRFLOW-2107] add time_partitioning to run_query on BigQueryBaseCursor
- [AIRFLOW-1057][AIRFLOW-1380][AIRFLOW-2362][2362] AIRFLOW Update DockerOperator to new API
- [AIRFLOW-2415] Make airflow DAG templating render numbers
- [AIRFLOW-2473] Fix wrong skip condition for TransferTests
- [AIRFLOW-2472] Implement MySqlHook.bulk_dump
- [AIRFLOW-2419] Use default view for subdag operator
- [AIRFLOW-2498] Fix Unexpected argument in SFTP Sensor
- [AIRFLOW-2509] Separate config docs into how-to guides
- [AIRFLOW-2429] Add BaseExecutor back
- [AIRFLOW-2429] Fix dag, example_dags, executors flake8 error
- [AIRFLOW-2502] Change Single triple quotes to double for docstrings
- [AIRFLOW-2503] Fix broken links in CONTRIBUTING.md
- [AIRFLOW-2501] Refer to devel instructions in docs contrib guide
- [AIRFLOW-2429] Fix contrib folder's flake8 errors
- [AIRFLOW-2471] Fix HiveCliHook.load_df to use unused parameters
- [AIRFLOW-2495] Update celery to 4.1.1
- [AIRFLOW-2429] Fix api, bin, config_templates folders flake8 error
- [AIRFLOW-2493] Mark template_fields of all Operators in the API document as "templated"
- [AIRFLOW-2489] Update FlaskAppBuilder to 1.11.1
- [AIRFLOW-2448] Enhance HiveCliHook.load_df to work with datetime

- [AIRFLOW-2487] Enhance druid ingestion hook
- [AIRFLOW-2397] Support affinity policies for Kubernetes executor/operator
- [AIRFLOW-2482] Add test for rewrite method in GCS Hook
- [AIRFLOW-2481] Fix flaky Kubernetes test
- [AIRFLOW-2479] Improve doc FAQ section
- [AIRFLOW-2485] Fix Incorrect logging for Qubole Sensor
- [AIRFLOW-2486] Remove unnecessary slash after port
- [AIRFLOW-2429] Make Airflow flake8 compliant
- [AIRFLOW-2491] Resolve flask version conflict
- [AIRFLOW-2484] Remove duplicate key in MySQL to GCS Op
- [AIRFLOW-2458] Add cassandra-to-gcs operator
- [AIRFLOW-2477] Improve time units for task duration and landing times charts for RBAC UI
- [AIRFLOW-2474] Only import snakebite if using py2
- [AIRFLOW-48] Parse connection uri querystring
- [AIRFLOW-2467][AIRFLOW-2] Update import direct warn message to use the module name
- [AIRFLOW-XXX] Fix order of companies
- [AIRFLOW-2452] Document field_dict must be OrderedDict
- [AIRFLOW-2420] Azure Data Lake Hook
- [AIRFLOW-2213] Add Quoble check operator
- [AIRFLOW-2465] Fix wrong module names in the doc
- [AIRFLOW-1929] Modifying TriggerDagRunOperator to accept execution_date
- [AIRFLOW-2460] Users can now use volume mounts and volumes
- [AIRFLOW-2110][AIRFLOW-2122] Enhance Http Hook
- [AIRFLOW-XXX] Updated contributors list
- [AIRFLOW-2435] Add launch_type to ECSOperator to allow FARGATE
- [AIRFLOW-2451] Remove extra slash ('/') char when using wildcard in gcs_to_gcs operator
- [AIRFLOW-2461] Add support for cluster scaling on dataproc operator
- [AIRFLOW-2376] Fix no hive section error
- [AIRFLOW-2425] Add lineage support
- [AIRFLOW-2430] Extend query batching to additional slow queries
- [AIRFLOW-2453] Add default nil value for kubernetes/git_subpath
- [AIRFLOW-2396] Add support for resources in kubernetes operator
- [AIRFLOW-2169] Encode binary data with base64 before importing to BigQuery
- [AIRFLOW-XXX] Add spotahome in user list
- [AIRFLOW-2457] Update FAB version requirement
- [AIRFLOW-2454][Airflow 2454] Support imagePullPolicy for k8s
- [AIRFLOW-2450] update supported k8s versions to 1.9 and 1.10
- [AIRFLOW-2333] Add Segment Hook and TrackEventOperator

- [AIRFLOW-2442][AIRFLOW-2] Airflow run command leaves database connections open
- [AIRFLOW-2016] assign template_fields for Dataproc Workflow Template sub-classes, not base class
- [AIRFLOW-2446] Add S3ToRedshiftTransfer into the "Integration" doc
- [AIRFLOW-2449] Fix operators.py to run all test cases
- [AIRFLOW-2424] Add dagrun status endpoint and increased k8s test coverage
- [AIRFLOW-2441] Fix bugs in HiveCliHook.load_df
- [AIRFLOW-2358][AIRFLOW-201804] Make the Kubernetes example optional
- [AIRFLOW-2436] Remove cli_logger in initdb
- [AIRFLOW-2444] Remove unused option(include_adhoc) in cli backfill command
- [AIRFLOW-2447] Fix TestHiveMetastoreHook to run all cases
- [AIRFLOW-2445] Allow templating in kubernetes operator
- [AIRFLOW-2086][AIRFLOW-2393] Customize default dagrun number in tree view
- [AIRFLOW-2437] Add PubNub to list of current airflow users
- [AIRFLOW-XXX] Add Quantopian to list of Airflow users
- [AIRFLOW-1978] Add WinRM windows operator and hook
- [AIRFLOW-2427] Add tests to named hive sensor
- [AIRFLOW-2412] Fix HiveCliHook.load_file to address HIVE-10541
- [AIRFLOW-2431] Add the navigation bar color parameter for RBAC UI
- [AIRFLOW-2407] Resolve Python undefined names
- [AIRFLOW-1952] Add the navigation bar color parameter
- [AIRFLOW-2222] Implement GoogleCloudStorageHook.rewrite
- [AIRFLOW-2426] Add Google Cloud Storage Hook tests
- [AIRFLOW-2418] Bump Flask-WTF
- [AIRFLOW-2417] Wait for pod is not running to end task
- [AIRFLOW-1914] Add other charset support to email utils
- [AIRFLOW-XXX] Update README.md with Craig@Work
- [AIRFLOW-1899] Fix Kubernetes tests
- [AIRFLOW-1812] Update logging example
- [AIRFLOW-2313] Add TTL parameters for Dataproc
- [AIRFLOW-2411] add dataproc_jars to templated_fields
- [AIRFLOW-XXX] Add Reddit to Airflow users
- [AIRFLOW-XXX] Fix wrong table header in scheduler.rst
- [AIRFLOW-2409] Supply password as a parameter
- [AIRFLOW-2410][AIRFLOW-75] Set the timezone in the RBAC Web UI
- [AIRFLOW-2394] default cmds and arguments in kubernetes operator
- [AIRFLOW-2406] Add Apache2 License Shield to Readme
- [AIRFLOW-2404] Add additional documentation for unqueued task
- [AIRFLOW-2400] Add Ability to set Environment Variables for K8s

- [AIRFLOW-XXX] Add Twine Labs as an Airflow user
- [AIRFLOW-1853] Show only the desired number of runs in tree view
- [AIRFLOW-2401] Document the use of variables in Jinja template
- [AIRFLOW-2403] Fix License Headers
- [AIRFLOW-1313] Fix license header
- [AIRFLOW-2398] Add BounceX to list of current airflow users
- [AIRFLOW-2363] Fix return type bug in TaskHandler
- [AIRFLOW-2389] Create a pinot db api hook
- [AIRFLOW-2390] Resolve FlaskWTFDeprecationWarning
- [AIRFLOW-1933] Fix some typos
- [AIRFLOW-1960] Add support for secrets in kubernetes operator
- [AIRFLOW-1313] Add vertica_to_mysql operator
- [AIRFLOW-1575] Add AWS Kinesis Firehose Hook for inserting batch records
- [AIRFLOW-2266][AIRFLOW-2343] Remove google-cloud-dataflow dependency
- [AIRFLOW-2370] Implement –use_random_password in create_user
- [AIRFLOW-2348] Strip path prefix from the destination_object when source_object contains a wildcard[]
- [AIRFLOW-2391] Fix to Flask 0.12.2
- [AIRFLOW-2381] Fix the flaky ApiPasswordTests test
- [AIRFLOW-2378] Add Groupon to list of current users
- [AIRFLOW-2382] Fix wrong description for delimiter
- [AIRFLOW-2380] Add support for environment variables in Spark submit operator.
- [AIRFLOW-2377] Improve Sendgrid sender support
- [AIRFLOW-2331] Support init action timeout on dataproc cluster create
- [AIRFLOW-1835] Update docs: Variable file is json
- [AIRFLOW-1781] Make search case-insensitive in LDAP group
- [AIRFLOW-2042] Fix browser menu appearing over the autocomplete menu
- [AIRFLOW-XXX] Remove wheelhouse files from travis not owned by travis
- [AIRFLOW-2336] Use hmsclient in hive_hook
- [AIRFLOW-2041] Correct Syntax in python examples
- [AIRFLOW-74] SubdagOperators can consume all celeryd worker processes
- [AIRFLOW-2369] Fix gcs tests
- [AIRFLOW-2365] Fix autocommit attribute check
- [AIRFLOW-2068] MesosExecutor allows optional Docker image
- [AIRFLOW-1652] Push DatabricksRunSubmitOperator metadata into XCOM
- [AIRFLOW-2234] Enable insert_rows for PrestoHook
- [AIRFLOW-2208][Airflow-22208] Link to same DagRun graph from TaskInstance view
- [AIRFLOW-1153] Allow HiveOperators to take hiveconfs
- [AIRFLOW-775] Fix autocommit settings with Jdbc hook

- [AIRFLOW-2364] Warn when setting autocommit on a connection which does not support it
- [AIRFLOW-2357] Add persistent volume for the logs
- [AIRFLOW-766] Skip conn.commit() when in Auto-commit
- [AIRFLOW-2351] Check for valid default_args start_date
- [AIRFLOW-1433] Set default rbac to initdb
- [AIRFLOW-2270] Handle removed tasks in backfill
- [AIRFLOW-2344] Fix *connections -l* to work with pipe/redirect
- [AIRFLOW-2300] Add S3 Select functionarity to S3ToHiveTransfer
- [AIRFLOW-1314] Cleanup the config
- [AIRFLOW-1314] Polish some of the Kubernetes docs/config
- [AIRFLOW-1314] Improve error handling
- [AIRFLOW-1999] Add per-task GCP service account support
- [AIRFLOW-1314] Rebasing against master
- [AIRFLOW-1314] Small cleanup to address PR comments (#24)
- [AIRFLOW-1314] Add executor_config and tests
- [AIRFLOW-1314] Improve k8s support
- [AIRFLOW-1314] Use VolumeClaim for transporting DAGs
- [AIRFLOW-1314] Create integration testing environment
- [AIRFLOW-1314] Git Mode to pull in DAGs for Kubernetes Executor
- [AIRFLOW-1314] Add support for volume mounts & Secrets in Kubernetes Executor
- [AIRFLOW=1314] Basic Kubernetes Mode
- [AIRFLOW-2326][AIRFLOW-2222] remove contrib.gcs_copy_operator
- [AIRFLOW-2328] Fix empty GCS blob in S3ToGoogleCloudStorageOperator
- [AIRFLOW-2350] Fix grammar in UPDATING.md
- [AIRFLOW-2302] Fix documentation
- [AIRFLOW-2345] pip is not used in this setup.py
- [AIRFLOW-2347] Add Banco de Formaturas to Readme
- [AIRFLOW-2346] Add Investorise as official user of Airflow
- [AIRFLOW-2330] Do not append destination prefix if not given
- [AIRFLOW-2240][DASK] Added TLS/SSL support for the dask-distributed scheduler.
- [AIRFLOW-2309] Fix duration calculation on TaskFail
- [AIRFLOW-2335] fix issue with jdk8 download for ci
- [AIRFLOW-2184] Add druid_checker_operator
- [AIRFLOW-2299] Add S3 Select functionarity to S3FileTransformOperator
- [AIRFLOW-2254] Put header as first row in unload
- [AIRFLOW-610] Respect _cmd option in config before defaults
- [AIRFLOW-2287] Fix incorrect ASF headers
- [AIRFLOW-XXX] Add Zego as an Apache Airflow user

- [AIRFLOW-952] fix save empty extra field in UI
- [AIRFLOW-1325] Add ElasticSearch log handler and reader
- [AIRFLOW-2301] Sync files of an S3 key with a GCS path
- [AIRFLOW-2293] Fix S3FileTransformOperator to work with boto3
- [AIRFLOW-3212][AIRFLOW-2314] Remove only leading slash in GCS path
- [AIRFLOW-1509][AIRFLOW-442] SFTP Sensor
- [AIRFLOW-2291] Add optional params to ML Engine
- [AIRFLOW-1774] Allow consistent templating of arguments in MLEngineBatchPredictionOperator
- [AIRFLOW-2302] Add missing operators and hooks
- [AIRFLOW-2312] Docs Typo Correction: Corresponding
- [AIRFLOW-1623] Trigger on_kill method in operators
- [AIRFLOW-2162] When impersonating another user, pass env variables to sudo
- [AIRFLOW-2304] Update quickstart doc to mention scheduler part
- [AIRFLOW-1633] docker_operator needs a way to set shm_size
- [AIRFLOW-1340] Add S3 to Redshift transfer operator
- [AIRFLOW-2303] Lists the keys inside an S3 bucket
- [AIRFLOW-2209] restore flask_login imports
- [AIRFLOW-2306] Add Bonnier Broadcasting to list of current users
- [AIRFLOW-2305][AIRFLOW-2027] Fix CI failure caused by []
- [AIRFLOW-2281] Add support for Sendgrid categories
- [AIRFLOW-2027] Only trigger sleep in scheduler after all files have parsed
- [AIRFLOW-2256] SparkOperator: Add Client Standalone mode and retry mechanism
- [AIRFLOW-2284] GCS to S3 operator
- [AIRFLOW-2287] Update license notices
- [AIRFLOW-2296] Add Cinimex DataLab to Readme
- [AIRFLOW-2298] Add Kalibrr to who uses airflow
- [AIRFLOW-2292] Fix docstring for S3Hook.get_wildcard_key
- [AIRFLOW-XXX] Update PR template
- [AIRFLOW-XXX] Remove outdated migrations.sql
- [AIRFLOW-2287] Add license header to docs/Makefile
- [AIRFLOW-2286] Add tokopedia to the readme
- [AIRFLOW-2273] Add Discord webhook operator/hook
- [AIRFLOW-2282] Fix grammar in UPDATING.md
- [AIRFLOW-2200] Add snowflake operator with tests
- [AIRFLOW-2178] Add handling on SLA miss errors
- [AIRFLOW-2169] Fix type 'bytes' is not JSON serializable in python3
- [AIRFLOW-2215] Pass environment to subproces.Popen in base_task_runner
- [AIRFLOW-2253] Add Airflow CLI instrumentation

- [AIRFLOW-2274] Fix Dataflow tests
- [AIRFLOW-2269] Add Custom Ink as an Airflow user
- [AIRFLOW-2259] Dataflow Hook Index out of range
- [AIRFLOW-2233] Update updating.md to include the info of hdfs_sensors renaming
- [AIRFLOW-2217] Add Slack webhook operator
- [AIRFLOW-1729] improve dagBag time
- [AIRFLOW-2264] Improve create_user cli help message
- [AIRFLOW-2260] [AIRFLOW-2260] SSHOperator add command template .sh files
- [AIRFLOW-2261] Check config/env for remote base log folder
- [AIRFLOW-2258] Allow import of Parquet-format files into BigQuery
- [AIRFLOW-1430] Include INSTALL instructions to avoid GPL
- [AIRFLOW-1430] Solve GPL dependency
- [AIRFLOW-2251] Add Thinknear as an Airflow user
- [AIRFLOW-2244] bugfix: remove legacy LongText code from models.py
- [AIRFLOW-2247] Fix RedshiftToS3Transfer not to fail with ValueError
- [AIRFLOW-2249] Add side-loading support for Zendesk Hook
- [AIRFLOW-XXX] Add Qplum to Airflow users
- [AIRFLOW-2228] Enhancements in ValueCheckOperator
- [AIRFLOW-1206] Typos
- [AIRFLOW-2060] Update pendulum version to 1.4.4
- [AIRFLOW-2248] Fix wrong param name in RedshiftToS3Transfer doc
- [AIRFLOW-1433][AIRFLOW-85] New Airflow Webserver UI with RBAC support
- [AIRFLOW-1235] Fix webserver's odd behaviour
- [AIRFLOW-1460] Allow restoration of REMOVED TI's
- [airflow-2235] Fix wrong docstrings in two operators
- [AIRFLOW-XXX] Fix chronological order for companies using Airflow
- [AIRFLOW-2124] Upload Python file to a bucket for Dataproc
- [AIRFLOW-2212] Fix ungenerated sensor API reference
- [AIRFLOW-2226] Rename google_cloud_storage_default to google_cloud_default
- [AIRFLOW-2211] Rename hdfs_sensors.py to hdfs_sensor.py for consistency
- [AIRFLOW-2225] Update document to include DruidDbApiHook
- [Airflow-2202] Add filter support in HiveMetastoreHook().max_partition()
- [AIRFLOW-2220] Remove duplicate numeric list entry in security.rst
- [AIRFLOW-XXX] Update tutorial documentation
- [AIRFLOW-2215] Update celery task to preserve environment variables and improve logging on exception
- [AIRFLOW-2185] Use state instead of query param
- [AIRFLOW-2183] Refactor DruidHook to enable sql
- [AIRFLOW-2203] Defer cycle detection

- [AIRFLOW-2203] Remove Useless Commands.
- [AIRFLOW-2203] Cache signature in apply_defaults
- [AIRFLOW-2203] Speed up Operator Resources
- [AIRFLOW-2203] Cache static rules (trigger/weight)
- [AIRFLOW-2203] Store task ids as sets not lists
- [AIRFLOW-2205] Remove unsupported args from JdbcHook doc
- [AIRFLOW-2207] Fix flaky test that uses app.cached_app()
- [AIRFLOW-2206] Remove unsupported args from JdbcOperator doc
- [AIRFLOW-2140] Add Kubernetes scheduler to SparkSubmitOperator
- [AIRFLOW-XXX] Add Xero to list of users
- [AIRFLOW-2204] Fix webserver debug mode
- [AIRFLOW-102] Fix test_complex_template always succeeds
- [AIRFLOW-442] Add SFTPHook
- [AIRFLOW-2169] Add schema to MySqlToGoogleCloudStorageOperator
- [AIRFLOW-2184][AIRFLOW-2138] Google Cloud Storage allow wildcards
- [AIRFLOW-1588] Cast Variable value to string
- [AIRFLOW-2199] Fix invalid reference to logger
- [AIRFLOW-2191] Change scheduler heartbeat logs from info to debug
- [AIRFLOW-2106] SalesForce hook sandbox option
- [AIRFLOW-2197] Silence hostname_callable config error message
- [AIRFLOW-2150] Use lighter call in HiveMetastoreHook().max_partition()
- [AIRFLOW-2186] Change the way logging is carried out in few ops
- [AIRFLOW-2181] Convert password_auth and test_password_endpoints from DOS to UNIX
- [AIRFLOW-2187] Fix Broken Travis CI due to AIRFLOW-2123
- [AIRFLOW-2175] Check that filepath is not None
- [AIRFLOW-2173] Don't check task IDs for concurrency reached check
- [AIRFLOW-2168] Remote logging for Azure Blob Storage
- [AIRFLOW-XXX] Add DocuTAP to list of users
- [AIRFLOW-2176] Change the way logging is carried out in BQ Get Data Operator
- [AIRFLOW-2177] Add mock test for GCS Download op
- [AIRFLOW-2123] Install CI dependencies from setup.py
- [AIRFLOW-2129] Presto hook calls _parse_exception_message but defines _get_pretty_exception_message
- [AIRFLOW-2174] Fix typos and wrongly rendered documents
- [AIRFLOW-2171] Store delegated credentials
- [AIRFLOW-2166] Restore BQ run_query dialect param
- [AIRFLOW-2163] Add HBC Digital to users of airflow
- [AIRFLOW-2065] Fix race-conditions when creating loggers
- [AIRFLOW-2147] Plugin manager: added 'sensors' attribute

- [AIRFLOW-2059] taskinstance query is awful, un-indexed, and does not scale
- [AIRFLOW-2159] Fix a few typos in salesforce_hook
- [AIRFLOW-2132] Add step to initialize database
- [AIRFLOW-2160] Fix bad rowid deserialization
- [AIRFLOW-2161] Add Vevo to list of companies using Airflow
- [AIRFLOW-2149] Add link to apache Beam documentation to create self executing Jar
- [AIRFLOW-2151] Allow getting the session from AwsHook
- [AIRFLOW-2097] tz referenced before assignment
- [AIRFLOW-2152] Add Multiply to list of companies using Airflow
- [AIRFLOW-1551] Add operator to trigger Jenkins job
- [AIRFLOW-2034] Fix mixup between %s and {} when using str.format Convention is to use .format for string formating oustide logging, else use lazy format See comment in related issue https://github.com/apache/airflow/pull/2823/files Identified problematic case using following command line .git/COMMIT_EDITMSG:grep -r ‘%s’./* | grep ‘.format(‘
- [AIRFLOW-2102] Add custom_args to Sendgrid personalizations
- [AIRFLOW-1035][AIRFLOW-1053] import unicode_literals to parse Unicode in HQL
- [AIRFLOW-2127] Keep loggers during DB migrations
- [AIRFLOW-2146] Resolve issues with BQ using DbApiHook methods
- [AIRFLOW-2087] Scheduler Report shows incorrect Total task number
- [AIRFLOW-2139] Remove unncecessary boilerplate to get DataFrame using pandas_gbq
- [AIRFLOW-2125] Using binary package psycopg2-binary
- [AIRFLOW-2142] Include message on mkdir failure
- [AIRFLOW-1615] SSHHook: use port specified by Connection
- [AIRFLOW-2122] Handle boolean values in sshHook
- [AIRFLOW-XXX] Add Tile to the list of users
- [AIRFLOW-2130] Add missing Operators to API Reference docs
- [AIRFLOW-XXX] Add timeout units (seconds)
- [AIRFLOW-2134] Add Alan to the list of companies that use Airflow
- [AIRFLOW-2133] Remove references to GitHub issues in CONTRIBUTING
- [AIRFLOW-2131] Remove confusing AirflowImport docs
- [AIRFLOW-1852] Allow hostname to be overridable.
- [AIRFLOW-2126] Add Bluecore to active users
- [AIRFLOW-1618] Add feature to create GCS bucket
- [AIRFLOW-2108] Fix log indentation in BashOperator
- [AIRFLOW-2115] Fix doc links to PythonHosted
- [AIRFLOW-XXX] Add contributor from Easy company
- [AIRFLOW-1882] Add ignoreUnknownValues option to gcs_to_bq operator
- [AIRFLOW-2089] Add on kill for SparkSubmit in Standalone Cluster

- [AIRFLOW-2113] Address missing DagRun callbacks Given that the handle_callback method belongs to the DAG object, we are able to get the list of task directly with get_task and reduce the communication with the database, making airflow more lightweight.
- [AIRFLOW-2112] Fix svg width for Recent Tasks on UI.
- [AIRFLOW-2116] Set CI Cloudant version to <2.0
- [AIRFLOW-XXX] Add PMC to list of companies using Airflow
- [AIRFLOW-2100] Fix Broken Documentation Links
- [AIRFLOW-1404] Add 'flatten_results' & 'maximum_bytes_billed' to BQ Operator
- [AIRFLOW-800] Initialize valid Google BigQuery Connection
- [AIRFLOW-1319] Fix misleading SparkSubmitOperator and SparkSubmitHook docstring
- [AIRFLOW-1983] Parse environment parameter as template
- [AIRFLOW-2095] Add operator to create External BigQuery Table
- [AIRFLOW-2085] Add SparkJdbc operator
- [AIRFLOW-1002] Add ability to clean all dependencies of removed DAG
- [AIRFLOW-2094] Jinjafied project_id, region & zone in DataProc{*} Operators
- [AIRFLOW-2092] Fixed incorrect parameter in docstring for FTPHook
- [AIRFLOW-XXX] Add SocialCops to Airflow users
- [AIRFLOW-2088] Fix duplicate keys in MySQL to GCS Helper function
- [AIRFLOW-2091] Fix incorrect docstring parameter in BigQuery Hook
- [AIRFLOW-2090] Fix typo in DataStore Hook
- [AIRFLOW-1157] Fix missing pools crashing the scheduler
- [AIRFLOW-713] Jinjafy {EmrCreateJobFlow,EmrAddSteps}Operator attributes
- [AIRFLOW-2083] Docs: Use "its" instead of "it's" where appropriate
- [AIRFLOW-2066] Add operator to create empty BQ table
- [AIRFLOW-XXX] add Karmic to list of companies
- [AIRFLOW-2073] Make FileSensor fail when the file doesn't exist
- [AIRFLOW-2078] Improve task_stats and dag_stats performance
- [AIRFLOW-2080] Use a log-out icon instead of a power button
- [AIRFLOW-2077] Fetch all pages of list_objects_v2 response
- [AIRFLOW-XXX] Add TM to list of companies
- [AIRFLOW-1985] Impersonation fixes for using *run_as_user*
- [AIRFLOW-2018][AIRFLOW-2] Make Sensors backward compatible
- [AIRFLOW-XXX] Fix typo in concepts doc (dag_md)
- [AIRFLOW-2069] Allow Bytes to be uploaded to S3
- [AIRFLOW-2074] Fix log var name in GHE auth
- [AIRFLOW-1927] Convert naive datetimes for TaskInstances
- [AIRFLOW-1760] Password auth for experimental API
- [AIRFLOW-2038] Add missing kubernetes dependency for dev
- [AIRFLOW-2040] Escape special chars in task instance logs URL

- [AIRFLOW-1968][AIRFLOW-1520] Add role_arn and aws_account_id/aws_iam_role support back to aws hook
- [AIRFLOW-2048] Fix task instance failure string formatting
- [AIRFLOW-2046] Fix kerberos error to work with python 3.x
- [AIRFLOW-2063] Add missing docs for GCP
- [AIRFLOW-XXX] Fix typo in docs
- [AIRFLOW-1793] Use docker_url instead of invalid base_url
- [AIRFLOW-2055] Elaborate on slightly ambiguous documentation
- [AIRFLOW-2039] BigQueryOperator supports priority property
- [AIRFLOW-2053] Fix quote character bug in BQ hook
- [AIRFLOW-2057] Add Overstock to list of companies
- [AIRFLOW-XXX] Add Plaid to Airflow users
- [AIRFLOW-2044] Add SparkSubmitOperator to documentation
- [AIRFLOW-2037] Add methods to get Hash values of a GCS object
- [AIRFLOW-2050] Fix Travis permission problem
- [AIRFLOW-2043] Add Intercom to list of companies
- [AIRFLOW-2023] Add debug logging around number of queued files
- [AIRFLOW-XXX] Add Pernod-ricard as a airflow user
- [AIRFLOW-1453] Add 'steps' into template_fields in EmrAddSteps
- [AIRFLOW-2015] Add flag for interactive runs
- [AIRFLOW-1895] Fix primary key integrity for mysql
- [AIRFLOW-2030] Fix KeyError:*i* in DbApiHook for insert
- [AIRFLOW-1943] Add External BigQuery Table feature
- [AIRFLOW-2033] Add Google Cloud Storage List Operator
- [AIRFLOW-2006] Add local log catching to kubernetes operator
- [AIRFLOW-2031] Add missing gcp_conn_id in the example in DataFlow docstrings
- [AIRFLOW-2029] Fix AttributeError in BigQueryPandasConnector
- [AIRFLOW-2028] Add JobTeaser to official users list
- [AIRFLOW-2016] Add support for Dataproc Workflow Templates
- [AIRFLOW-2025] Reduced Logging verbosity
- [AIRFLOW-1267][AIRFLOW-1874] Add dialect parameter to BigQueryHook
- [AIRFLOW-XXX] Fixed a typo
- [AIRFLOW-XXX] Typo node to nodes
- [AIRFLOW-2019] Update DataflowHook for updating Streaming type job
- [AIRFLOW-2017][Airflow 2017] adding query output to PostgresOperator
- [AIRFLOW-1889] Split sensors into separate files
- [AIRFLOW-1950] Optionally pass xcom_pull task_ids
- [AIRFLOW-1755] Allow mount below root
- [AIRFLOW-511][Airflow 511] add success/failure callbacks on dag level

- [AIRFLOW-192] Add weight_rule param to BaseOperator
- [AIRFLOW-2008] Use callable for python column defaults
- [AIRFLOW-1984] Fix to AWS Batch operator
- [AIRFLOW-2000] Support non-main dataflow job class
- [AIRFLOW-2003] Use flask-caching instead of flask-cache
- [AIRFLOW-2002] Do not swallow exception on logging import
- [AIRFLOW-2004] Import flash from flask not flask.login
- [AIRFLOW-1997] Fix GCP operator doc strings
- [AIRFLOW-1996] Update DataflowHook waitfordone for Streaming type job[]
- [AIRFLOW-1995][Airflow 1995] add on_kill method to SqoopOperator
- [AIRFLOW-1770] Allow HiveOperator to take in a file
- [AIRFLOW-1994] Change background color of Scheduled state Task Instances
- [AIRFLOW-1436][AIRFLOW-1475] EmrJobFlowSensor considers Cancelled step as Successful
- [AIRFLOW-1517] Kubernetes operator PR fixes
- [AIRFLOW-1517] addressed PR comments
- [AIRFLOW-1517] started documentation of k8s operator
- [AIRFLOW-1517] Restore authorship of resources
- [AIRFLOW-1517] Remove authorship of resources
- [AIRFLOW-1517] Add minikube for kubernetes integration tests
- [AIRFLOW-1517] Restore authorship of resources
- [AIRFLOW-1517] fixed license issues
- [AIRFLOW-1517] Created more accurate failures for kube cluster issues
- [AIRFLOW-1517] Remove authorship of resources
- [AIRFLOW-1517] Add minikube for kubernetes integration tests
- [AIRFLOW-1988] Change BG color of None state TIs
- [AIRFLOW-790] Clean up TaskInstances without DagRuns
- [AIRFLOW-1949] Fix var upload, str() produces "b'…'" which is not json
- [AIRFLOW-1930] Convert func.now() to timezone.utcnow()
- [AIRFLOW-1688] Support load.time_partitioning in bigquery_hook
- [AIRFLOW-1975] Make TriggerDagRunOperator callback optional
- [AIRFLOW-1480] Render template attributes for ExternalTaskSensor fields
- [AIRFLOW-1958] Add kwargs to send_email
- [AIRFLOW-1976] Fix for missing log/logger attribute FileProcessHandler
- [AIRFLOW-1982] Fix Executor event log formatting
- [AIRFLOW-1971] Propagate hive config on impersonation
- [AIRFLOW-1969] Always use HTTPS URIs for Google OAuth2
- [AIRFLOW-1954] Add DataFlowTemplateOperator
- [AIRFLOW-1963] Add config for HiveOperator mapred_queue

- [AIRFLOW-1946][AIRFLOW-1855] Create a BigQuery Get Data Operator
- [AIRFLOW-1953] Add labels to dataflow operators
- [AIRFLOW-1967] Update Celery to 4.0.2
- [AIRFLOW-1964] Add Upsight to list of Airflow users
- [AIRFLOW-XXX] Changelog for 1.9.0
- [AIRFLOW-1470] Implement BashSensor operator
- [AIRFLOW-XXX] Pin sqlalchemy dependency
- [AIRFLOW-1955] Do not reference unassigned variable
- [AIRFLOW-1957] Add contributor to BalanceHero in Readme
- [AIRFLOW-1517] Restore authorship of secrets and init container
- [AIRFLOW-1517] Remove authorship of secrets and init container
- [AIRFLOW-1935] Add BalanceHero to readme
- [AIRFLOW-1939] add astronomer contributors
- [AIRFLOW-1517] Kubernetes Operator
- [AIRFLOW-1928] Fix @once with catchup=False
- [AIRFLOW-1937] Speed up scheduling by committing in batch
- [AIRFLOW-1821] Enhance default logging config by removing extra loggers
- [AIRFLOW-1904] Correct DAG fileloc to the right filepath
- [AIRFLOW-1909] Update docs with supported versions of MySQL server
- [AIRFLOW-1915] Relax flask-wtf dependency specification
- [AIRFLOW-1920] Update CONTRIBUTING.md to reflect enforced linting rules
- [AIRFLOW-1942] Update Sphinx docs to remove deprecated import structure
- [AIRFLOW-1846][AIRFLOW-1697] Hide Ad Hoc Query behind secure_mode config
- [AIRFLOW-1948] Include details for on_kill failure
- [AIRFLOW-1938] Clean up unused exception
- [AIRFLOW-1932] Add GCP Pub/Sub Pull and Ack
- [AIRFLOW-XXX] Purge coveralls
- [AIRFLOW-XXX] Remove unused coveralls token
- [AIRFLOW-1938] Remove tag version check in setup.py
- [AIRFLOW-1916] Don't upload logs to remote from *run –raw*
- [AIRFLOW-XXX] Fix failing PubSub tests on Python3
- [AIRFLOW-XXX] Upgrade to python 3.5 and disable dask tests
- [AIRFLOW-1913] Add new GCP PubSub operators
- [AIRFLOW-1525] Fix minor LICENSE and NOTICE issues
- [AIRFLOW-1687] fix fernet error without encryption
- [AIRFLOW-1912] airflow.processor should not propagate logging
- [AIRFLOW-1911] Rename celeryd_concurrency
- [AIRFLOW-1885] Fix IndexError in ready_prefix_on_cmdline

- [AIRFLOW-1854] Improve Spark Submit operator for standalone cluster mode
- [AIRFLOW-1908] Fix celery broker options config load
- [AIRFLOW-1907] Pass max_ingestion_time to Druid hook
- [AIRFLOW-1909] Add away to list of users
- [AIRFLOW-1893][AIRFLOW-1901] Propagate PYTHONPATH when using impersonation
- [AIRFLOW-1892] Modify BQ hook to extract data filtered by column
- [AIRFLOW-1829] Support for schema updates in query jobs
- [AIRFLOW-1840] Make celery configuration congruent with Celery 4
- [AIRFLOW-1878] Fix stderr/stdout redirection for tasks
- [AIRFLOW-1897][AIRFLOW-1873] Task Logs for running instance not visible in WebUI
- [AIRFLOW-1896] FIX bleach <> html5lib incompatibility
- [AIRFLOW-1884][AIRFLOW-1059] Reset orphaned task state for external dagruns
- [AIRFLOW-XXX] Fix typo in comment
- [AIRFLOW-1869] Do not emit spurious warning on missing logs
- [AIRFLOW-1888] Add AWS Redshift Cluster Sensor
- [AIRFLOW-1887] Renamed endpoint url variable
- [AIRFLOW-1873] Set TI.try_number to right value depending TI state
- [AIRFLOW-1891] Fix non-ascii typo in default configuration template
- [AIRFLOW-1879] Handle ti log entirely within ti
- [AIRFLOW-1869] Write more error messages into gcs and file logs
- [AIRFLOW-1876] Write subtask id to task log header
- [AIRFLOW-1554] Fix wrong DagFileProcessor termination method call
- [AIRFLOW-342] Do not use amqp, rpc as result backend
- [AIRFLOW-966] Make celery broker_transport_options configurable
- [AIRFLOW-1881] Make operator log in task log
- [AIRFLOW-XXX] Added DataReply to the list of Airflow Users
- [AIRFLOW-1883] Get File Size for objects in Google Cloud Storage
- [AIRFLOW-1872] Set context for all handlers including parents
- [AIRFLOW-1855][AIRFLOW-1866] Add GCS Copy Operator to copy multiple files
- [AIRFLOW-1870] Enable flake8 tests
- [AIRFLOW-1785] Enable Python 3 tests
- [AIRFLOW-1850] Copy cmd before masking
- [AIRFLOW-1665] Reconnect on database errors
- [AIRFLOW-1559] Dispose SQLAlchemy engines on exit
- [AIRFLOW-1559] Close file handles in subprocesses
- [AIRFLOW-1559] Make database pooling optional
- [AIRFLOW-1848][Airflow-1848] Fix DataFlowPythonOperator py_file extension doc comment
- [AIRFLOW-1843] Add Google Cloud Storage Sensor with prefix

- [AIRFLOW-1803] Time zone documentation
- [AIRFLOW-1826] Update views to use timezone aware objects
- [AIRFLOW-1827] Fix api endpoint date parsing
- [AIRFLOW-1806] Use naive datetime when using cron
- [AIRFLOW-1809] Update tests to use timezone aware objects
- [AIRFLOW-1806] Use naive datetime for cron scheduling
- [AIRFLOW-1807] Force use of time zone aware db fields
- [AIRFLOW-1808] Convert all utcnow() to time zone aware
- [AIRFLOW-1804] Add time zone configuration options
- [AIRFLOW-1802] Convert database fields to timezone aware
- [AIRFLOW-XXX] Add dask lock files to excludes
- [AIRFLOW-1790] Add support for AWS Batch operator
- [AIRFLOW-XXX] Update README.md
- [AIRFLOW-1820] Remove timestamp from metric name
- [AIRFLOW-1810] Remove unused mysql import in migrations.
- [AIRFLOW-1838] Properly log collect_dags exception
- [AIRFLOW-1842] Fixed Super class name for the gcs to gcs copy operator
- [AIRFLOW-1845] Modal background now covers long or tall pages
- [AIRFLOW-1229] Add link to Run Id, incl execution_date
- [AIRFLOW-1842] Add gcs to gcs copy operator with renaming if required
- [AIRFLOW-1841] change False to None in operator and hook
- [AIRFLOW-1839] Fix more bugs in S3Hook boto -> boto3 migration
- [AIRFLOW-1830] Support multiple domains in Google authentication backend
- [AIRFLOW-1831] Add driver-classpath spark submit
- [AIRFLOW-1795] Correctly call S3Hook after migration to boto3
- [AIRFLOW-1811] Fix render Druid operator
- [AIRFLOW-1819] Fix slack operator unittest bug
- [AIRFLOW-1805] Allow Slack token to be passed through connection
- [AIRFLOW-1816] Add region param to Dataproc operators
- [AIRFLOW-868] Add postgres_to_gcs operator and unittests
- [AIRFLOW-1613] make mysql_to_gcs_operator py3 compatible
- [AIRFLOW-1817] use boto3 for s3 dependency
- [AIRFLOW-1813] Bug SSH Operator empty buffer
- [AIRFLOW-1801][AIRFLOW-288] Url encode execution dates
- [AIRFLOW-1563] Catch OSError while symlinking the latest log directory
- [AIRFLOW-1794] Remove uses of Exception.message for Python 3
- [AIRFLOW-1799] Fix logging line which raises errors
- [AIRFLOW-1102] Upgrade Gunicorn >=19.4.0

- [AIRFLOW-1756] Fix S3TaskHandler to work with Boto3-based S3Hook
- [AIRFLOW-1797] S3Hook.load_string didn't work on Python3
- [AIRFLOW-646] Add docutils to setup_requires
- [AIRFLOW-1792] Missing intervals DruidOperator
- [AIRFLOW-1789][AIRFLOW-1712] Log SSHOperator stderr to log.warning
- [AIRFLOW-1787] Fix task instance batch clear and set state bugs
- [AIRFLOW-1780] Fix long output lines with unicode from hanging parent
- [AIRFLOW-387] Close SQLAlchemy sessions properly
- [AIRFLOW-1779] Add keepalive packets to ssh hook
- [AIRFLOW-1669] Fix Docker and pin Moto to 1.1.19
- [AIRFLOW-71] Add support for private Docker images
- [AIRFLOW-XXX] Give a clue what the 'ds' variable is
- [AIRFLOW-XXX] Correct typos in the faq docs page
- [AIRFLOW-1571] Add AWS Lambda Hook
- [AIRFLOW-1675] Fix docstrings for API docs
- [AIRFLOW-1712][AIRFLOW-756][AIRFLOW-751] Log SSHOperator output
- [AIRFLOW-1776] Capture stdout and stderr for logging
- [AIRFLOW-1765] Make experimental API securable without needing Kerberos.
- [AIRFLOW-1764] The web interface should not use the experimental API
- [AIRFLOW-1771] Rename heartbeat to avoid confusion
- [AIRFLOW-1769] Add support for templates in VirtualenvOperator
- [AIRFLOW-1763] Fix S3TaskHandler unit tests
- [AIRFLOW-1315] Add Qubole File & Partition Sensors
- [AIRFLOW-1018] Make processor use logging framework
- [AIRFLOW-1695] Add RedshiftHook using boto3
- [AIRFLOW-1706] Fix query error for MSSQL backend
- [AIRFLOW-1711] Use ldap3 dict for group membership
- [AIRFLOW-1723] Make sendgrid a plugin
- [AIRFLOW-1757] Add missing options to SparkSubmitOperator
- [AIRFLOW-1734][Airflow 1734] Sqoop hook/operator enhancements
- [AIRFLOW-1761] Fix type in scheduler.rst
- [AIRFLOW-1731] Set pythonpath for logging
- [AIRFLOW-1641] Handle executor events in the scheduler
- [AIRFLOW-1744] Make sure max_tries can be set
- [AIRFLOW-1732] Improve dataflow hook logging
- [AIRFLOW-1736] Add HotelQuickly to Who Uses Airflow
- [AIRFLOW-1657] Handle failing qubole operator
- [AIRFLOW-1677] Fix typo in example_qubole_operator

- [AIRFLOW-926] Fix JDBC Hook
- [AIRFLOW-1520] Boto3 S3Hook, S3Log
- [AIRFLOW-1716] Fix multiple __init__ def in SimpleDag
- [AIRFLOW-XXX] Fix DateTime in Tree View
- [AIRFLOW-1719] Fix small typo
- [AIRFLOW-1432] Charts label for Y axis not visible
- [AIRFLOW-1743] Verify ldap filters correctly
- [AIRFLOW-1745] Restore default signal disposition
- [AIRFLOW-1741] Correctly hide second chart on task duration page
- [AIRFLOW-1728] Add networkUri, subnet, tags to Dataproc operator
- [AIRFLOW-1726] Add copy_expert psycopg2 method to PostgresHook
- [AIRFLOW-1330] Add conn_type argument to CLI when adding connection
- [AIRFLOW-1698] Remove SCHEDULER_RUNS env var in systemd
- [AIRFLOW-1694] Stop using itertools.izip
- [AIRFLOW-1692] Change test_views filename to support Windows
- [AIRFLOW-1722] Fix typo in scheduler autorestart output filename
- [AIRFLOW-1723] Support sendgrid in email backend
- [AIRFLOW-1718] Set num_retries on Dataproc job request execution
- [AIRFLOW-1727] Add unit tests for DataProcHook
- [AIRFLOW-1631] Fix timing issue in unit test
- [AIRFLOW-1631] Fix local executor unbound parallelism
- [AIRFLOW-1724] Add Fundera to Who uses Airflow?
- [AIRFLOW-1683] Cancel BigQuery job on timeout.
- [AIRFLOW-1714] Fix misspelling: s/seperate/separate/
- [AIRFLOW-1681] Add batch clear in task instance view
- [AIRFLOW-1696] Fix dataproc version label error
- [AIRFLOW-1613] Handle binary field in MySqlToGoogleCloudStorageOperator
- [AIRFLOW-1697] Mode to disable charts endpoint
- [AIRFLOW-1691] Add better Google cloud logging documentation
- [AIRFLOW-1690] Add detail to gcs error messages
- [AIRFLOW-1682] Make S3TaskHandler write to S3 on close
- [AIRFLOW-1634] Adds task_concurrency feature
- [AIRFLOW-1676] Make GCSTaskHandler write to GCS on close
- [AIRFLOW-1678] Fix erroneously repeated word in function docstrings
- [AIRFLOW-1323] Made Dataproc operator parameter names consistent
- [AIRFLOW-1590] fix unused module and variable
- [AIRFLOW-1671] Add @apply_defaults back to gcs download operator
- [AIRFLOW-988] Fix repeating SLA miss callbacks

- [AIRFLOW-1611] Customize logging
- [AIRFLOW-1668] Expose keepalives_idle for Postgres connections
- [AIRFLOW-1658] Kill Druid task on timeout
- [AIRFLOW-1669][AIRFLOW-1368] Fix Docker import
- [AIRFLOW-891] Make webserver clock include date
- [AIRFLOW-1560] Add AWS DynamoDB hook and operator for inserting batch items
- [AIRFLOW-1654] Show tooltips for link icons in DAGs view
- [AIRFLOW-1660] Change webpage width to full-width
- [AIRFLOW-1664] write file as binary instead of str
- [AIRFLOW-1659] Fix invalid obj attribute bug in file_task_handler.py
- [AIRFLOW-1635] Allow creating GCP connection without requiring a JSON file
- [AIRFLOW-1650] Fix custom celery config loading
- [AIRFLOW-1647] Fix Spark-sql hook
- [AIRFLOW-1587] Fix CeleryExecutor import error
- [Airflow-1640][AIRFLOW-1640] Add qubole default connection
- [AIRFLOW-1576] Added region param to Dataproc{*}Operators
- [AIRFLOW-1643] Add healthjump to officially using list
- [AIRFLOW-1626] Add Azri Solutions to Airflow users
- [AIRFLOW-1636] Add AWS and EMR connection type
- [AIRFLOW-1527] Refactor celery config
- [AIRFLOW-1639] Fix Fernet error handling
- [AIRFLOW-1637] Fix Travis CI build status link
- [AIRFLOW-1628] Fix docstring of sqlsensor
- [AIRFLOW-1331] add SparkSubmitOperator option
- [AIRFLOW-1627] Only query pool in SubDAG init when necessary
- [AIRFLOW-1629] Make extra a textarea in edit connections form
- [AIRFLOW-1368] Automatically remove Docker container on exit
- [AIRFLOW-289] Make airflow timezone independent
- [AIRFLOW-1356] Add *–celery_hostname* to *airflow worker*
- [AIRFLOW-1247] Fix ignore_all_dependencies argument ignored
- [AIRFLOW-1621] Add tests for server side paging
- [AIRFLOW-1591] Avoid attribute error when rendering logging filename
- [AIRFLOW-1031] Replace hard-code to DagRun.ID_PREFIX
- [AIRFLOW-1604] Rename logger to log
- [AIRFLOW-1512] Add PythonVirtualenvOperator
- [AIRFLOW-1617] Fix XSS vulnerability in Variable endpoint
- [AIRFLOW-1497] Reset hidden fields when changing connection type
- [AIRFLOW-1619] Add poll_sleep parameter to GCP dataflow operator

- [AIRFLOW-XXX] Remove landscape.io config
- [AIRFLOW-XXX] Remove non working service badges
- [AIRFLOW-1177] Fix Variable.setdefault w/existing JSON
- [AIRFLOW-1600] Fix exception handling in get_fernet
- [AIRFLOW-1614] Replace inspect.stack() with sys._getframe()
- [AIRFLOW-1519] Add server side paging in DAGs list
- [AIRFLOW-1309] Allow hive_to_druid to take tblproperties
- [AIRFLOW-1613] Make MySqlToGoogleCloudStorageOperator compaitible with python3
- [AIRFLOW-1603] add PAYMILL to companies list
- [AIRFLOW-1609] Fix gitignore to ignore all venvs
- [AIRFLOW-1601] Add configurable task cleanup time

3.19.5 Airflow 1.9.0, 2018-01-02

- [AIRFLOW-1525] Fix minor LICENSE and NOTICE issues
- [AIRFLOW-XXX] Bump version to 1.9.0
- [AIRFLOW-1897][AIRFLOW-1873] Task Logs for running instance not visible in WebUI
- [AIRFLOW-XXX] Make sure session is committed
- [AIRFLOW-1896] FIX bleach <> html5lib incompatibility
- [AIRFLOW-XXX] Fix log handler test
- [AIRFLOW-1873] Set TI.try_number to right value depending TI state
- [AIRFLOW-1554] Fix wrong DagFileProcessor termination method call
- [AIRFLOW-1872] Set context for all handlers including parents
- [AIRFLOW-XXX] Add dask lock files to excludes
- [AIRFLOW-1839] Fix more bugs in S3Hook boto -> boto3 migration
- [AIRFLOW-1795] Correctly call S3Hook after migration to boto3
- [AIRFLOW-1813] Bug SSH Operator empty buffer
- [AIRFLOW-1794] Remove uses of Exception.message for Python 3
- [AIRFLOW-1799] Fix logging line which raises errors
- [AIRFLOW-1102] Upgrade Gunicorn >=19.4.0
- [AIRFLOW-1756] Fix S3TaskHandler to work with Boto3-based S3Hook
- [AIRFLOW-1797] S3Hook.load_string didn't work on Python3
- [AIRFLOW-1792] Missing intervals DruidOperator
- [AIRFLOW-1789][AIRFLOW-1712] Log SSHOperator stderr to log.warning
- [AIRFLOW-1669] Fix Docker and pin Moto to 1.1.19
- [AIRFLOW-71] Add support for private Docker images
- [AIRFLOW-1779] Add keepalive packets to ssh hook
- [AIRFLOW-XXX] Give a clue what the 'ds' variable is
- [AIRFLOW-XXX] Correct typos in the faq docs page

- [AIRFLOW-1571] Add AWS Lambda Hook
- [AIRFLOW-1675] Fix docstrings for API docs
- [AIRFLOW-1712][AIRFLOW-756][AIRFLOW-751] Log SSHOperator output
- [AIRFLOW-1776] Capture stdout and stderr for logging
- [AIRFLOW-1765] Make experimental API securable without needing Kerberos.
- [AIRFLOW-1764] The web interface should not use the experimental API
- [AIRFLOW-1634] Adds task_concurrency feature
- [AIRFLOW-1018] Make processor use logging framework
- [AIRFLOW-1695] Add RedshiftHook using boto3
- [AIRFLOW-1706] Fix query error for MSSQL backend
- [AIRFLOW-1711] Use ldap3 dict for group membership
- [AIRFLOW-1757] Add missing options to SparkSubmitOperator
- [AIRFLOW-1734][Airflow 1734] Sqoop hook/operator enhancements
- [AIRFLOW-1731] Set pythonpath for logging
- [AIRFLOW-1641] Handle executor events in the scheduler
- [AIRFLOW-1744] Make sure max_tries can be set
- [AIRFLOW-1330] Add conn_type argument to CLI when adding connection
- [AIRFLOW-926] Fix JDBC Hook
- [AIRFLOW-1520] Boto3 S3Hook, S3Log
- [AIRFLOW-XXX] Fix DateTime in Tree View
- [AIRFLOW-1432] Charts label for Y axis not visible
- [AIRFLOW-1743] Verify ldap filters correctly
- [AIRFLOW-1745] Restore default signal disposition
- [AIRFLOW-1741] Correctly hide second chart on task duration page
- [AIRFLOW-1726] Add copy_expert psycopg2 method to PostgresHook
- [AIRFLOW-1698] Remove SCHEDULER_RUNS env var in systemd
- [AIRFLOW-1694] Stop using itertools.izip
- [AIRFLOW-1692] Change test_views filename to support Windows
- [AIRFLOW-1722] Fix typo in scheduler autorestart output filename
- [AIRFLOW-1691] Add better Google cloud logging documentation
- [AIRFLOW-1690] Add detail to gcs error messages
- [AIRFLOW-1682] Make S3TaskHandler write to S3 on close
- [AIRFLOW-1676] Make GCSTaskHandler write to GCS on close
- [AIRFLOW-1635] Allow creating GCP connection without requiring a JSON file
- [AIRFLOW-1323] Made Dataproc operator parameter names consistent
- [AIRFLOW-1590] fix unused module and variable
- [AIRFLOW-988] Fix repeating SLA miss callbacks
- [AIRFLOW-1611] Customize logging

- [AIRFLOW-1668] Expose keepalives_idle for Postgres connections
- [AIRFLOW-1658] Kill Druid task on timeout
- [AIRFLOW-1669][AIRFLOW-1368] Fix Docker import
- [AIRFLOW-1560] Add AWS DynamoDB hook and operator for inserting batch items
- [AIRFLOW-1654] Show tooltips for link icons in DAGs view
- [AIRFLOW-1660] Change webpage width to full-width
- [AIRFLOW-1664] write file as binary instead of str
- [AIRFLOW-1659] Fix invalid obj attribute bug in file_task_handler.py
- [AIRFLOW-1650] Fix custom celery config loading
- [AIRFLOW-1647] Fix Spark-sql hook
- [AIRFLOW-1587] Fix CeleryExecutor import error
- [AIRFLOW-1636] Add AWS and EMR connection type
- [AIRFLOW-1527] Refactor celery config
- [AIRFLOW-1639] Fix Fernet error handling
- [AIRFLOW-1628] Fix docstring of sqlsensor
- [AIRFLOW-1331] add SparkSubmitOperator option
- [AIRFLOW-1627] Only query pool in SubDAG init when necessary
- [AIRFLOW-1629] Make extra a textarea in edit connections form
- [AIRFLOW-1621] Add tests for server side paging
- [AIRFLOW-1519] Add server side paging in DAGs list
- [AIRFLOW-289] Make airflow timezone independent
- [AIRFLOW-1356] Add *–celery_hostname* to *airflow worker*
- [AIRFLOW-1591] Avoid attribute error when rendering logging filename
- [AIRFLOW-1031] Replace hard-code to DagRun.ID_PREFIX
- [AIRFLOW-1604] Rename logger to log
- [AIRFLOW-1512] Add PythonVirtualenvOperator
- [AIRFLOW-1617] Fix XSS vulnerability in Variable endpoint
- [AIRFLOW-1497] Reset hidden fields when changing connection type
- [AIRFLOW-1177] Fix Variable.setdefault w/existing JSON
- [AIRFLOW-1600] Fix exception handling in get_fernet
- [AIRFLOW-1614] Replace inspect.stack() with sys._getframe()
- [AIRFLOW-1613] Make MySqlToGoogleCloudStorageOperator compaitible with python3
- [AIRFLOW-1609] Fix gitignore to ignore all venvs
- [AIRFLOW-1601] Add configurable task cleanup time
- [AIRFLOW-XXX] Bumping Airflow 1.9.0alpha0 version
- [AIRFLOW-1608] Handle pending job state in GCP Dataflow hook
- [AIRFLOW-1606] Use non static DAG.sync_to_db

- [AIRFLOW-1606][Airflow-1606][AIRFLOW-1605][AIRFLOW-160] DAG.sync_to_db is now a normal method
- [AIRFLOW-1602] LoggingMixin in DAG class
- [AIRFLOW-1593] expose load_string in WasbHook
- [AIRFLOW-1597] Add GameWisp as Airflow user
- [AIRFLOW-1594] Don't install test packages into python root.[]
- [AIRFLOW-1582] Improve logging within Airflow
- [AIRFLOW-1476] add INSTALL instruction for source releases
- [AIRFLOW-XXX] Save username and password in airflow-pr
- [AIRFLOW-1522] Increase text size for var field in variables for MySQL
- [AIRFLOW-950] Missing AWS integrations on documentation::integrations
- [AIRFLOW-XXX] 1.8.2 release notes
- [AIRFLOW-1573] Remove *thrift < 0.10.0* requirement
- [AIRFLOW-1584] Remove insecure /headers endpoint
- [AIRFLOW-1586] Add mapping for date type to mysql_to_gcs operator
- [AIRFLOW-1579] Adds support for jagged rows in Bigquery hook for BQ load jobs
- [AIRFLOW-1577] Add token support to DatabricksHook
- [AIRFLOW-1580] Error in string formating
- [AIRFLOW-1567] Updated docs for Google ML Engine operators/hooks
- [AIRFLOW-1574] add 'to' attribute to templated vars of email operator
- [AIRFLOW-1572] add carbonite to company list
- [AIRFLOW-1568] Fix typo in BigQueryHook
- [AIRFLOW-1493][AIRFLOW-XXXX][WIP] fixed dumb thing
- [AIRFLOW-1567][Airflow-1567] Renamed cloudml hook and operator to mlengine
- [AIRFLOW-1568] Add datastore export/import operators
- [AIRFLOW-1564] Use Jinja2 to render logging filename
- [AIRFLOW-1562] Spark-sql logging contains deadlock
- [AIRFLOW-1556][Airflow 1556] Add support for SQL parameters in BigQueryBaseCursor
- [AIRFLOW-108] Add CreditCards.com to companies list
- [AIRFLOW-1541] Add channel to template fields of slack_operator
- [AIRFLOW-1535] Add service account/scopes in dataproc
- [AIRFLOW-1384] Add to README.md CaDC/ARGO
- [AIRFLOW-1546] add Zymergen 80to org list in README
- [AIRFLOW-1545] Add Nextdoor to companies list
- [AIRFLOW-1544] Add DataFox to companies list
- [AIRFLOW-1529] Add logic supporting quoted newlines in Google BigQuery load jobs
- [AIRFLOW-1521] Fix emplate rendering for BigqueryTableDeleteOperator
- [AIRFLOW-1324] Generalize Druid operator and hook

- [AIRFLOW-1516] Fix error handling getting fernet
- [AIRFLOW-1420][AIRFLOW-1473] Fix deadlock check
- [AIRFLOW-1495] Fix migration on index on job_id
- [AIRFLOW-1483] Making page size consistent in list
- [AIRFLOW-1495] Add TaskInstance index on job_id
- [AIRFLOW-855] Replace PickleType with LargeBinary in XCom
- [AIRFLOW-1505] Document when Jinja substitution occurs
- [AIRFLOW-1504] Log dataproc cluster name
- [AIRFLOW-1239] Fix unicode error for logs in base_task_runner
- [AIRFLOW-1280] Fix Gantt chart height
- [AIRFLOW-1507] Template parameters in file_to_gcs operator
- [AIRFLOW-1452] workaround lock on method
- [AIRFLOW-1385] Make Airflow task logging configurable
- [AIRFLOW-940] Handle error on variable decrypt
- [AIRFLOW-1492] Add gauge for task successes/failures
- [AIRFLOW-1443] Update Airflow configuration documentation
- [AIRFLOW-1486] Unexpected S3 writing log error
- [AIRFLOW-1487] Added links to all companies officially using Airflow
- [AIRFLOW-1489] Fix typo in BigQueryCheckOperator
- [AIRFLOW-1349] Fix backfill to respect limits
- [AIRFLOW-1478] Chart owner column should be sortable
- [AIRFLOW-1397][AIRFLOW-1] No Last Run column data displyed in Airflow UI 1.8.1
- [AIRFLOW-1474] Add dag_id regex feature for *airflow clear* command
- [AIRFLOW-1445] Changing HivePartitionSensor UI color to lighter shade
- [AIRFLOW-1359] Use default_args in Cloud ML eval
- [AIRFLOW-1389] Support createDisposition in BigQueryOperator
- [AIRFLOW-1349] Refactor BackfillJob _execute
- [AIRFLOW-1459] Fixed broken integration .rst formatting
- [AIRFLOW-1448] Revert "Fix cli reading logfile in memory"
- [AIRFLOW-1398] Allow ExternalTaskSensor to wait on multiple runs of a task
- [AIRFLOW-1399] Fix cli reading logfile in memory
- [AIRFLOW-1442] Remove extra space from ignore_all_deps generated command
- [AIRFLOW-1438] Change batch size per query in scheduler
- [AIRFLOW-1439] Add max billing tier for the BQ Hook and Operator
- [AIRFLOW-1437] Modify BigQueryTableDeleteOperator
- [Airflow 1332] Split logs based on try number
- [AIRFLOW-1385] Create abstraction for Airflow task logging
- [AIRFLOW-756][AIRFLOW-751] Replace ssh hook, operator & sftp operator with paramiko based

- [AIRFLOW-1393][[AIRFLOW-1393] Enable Py3 tests in contrib/spark_submit_hook[
- [AIRFLOW-1345] Dont expire TIs on each scheduler loop
- [AIRFLOW-1059] Reset orphaned tasks in batch for scheduler
- [AIRFLOW-1255] Fix SparkSubmitHook output deadlock
- [AIRFLOW-1359] Add Google CloudML utils for model evaluation
- [AIRFLOW-1247] Fix ignore all dependencies argument ignored
- [AIRFLOW-1401] Standardize cloud ml operator arguments
- [AIRFLOW-1394] Add quote_character param to GCS hook and operator
- [AIRFLOW-1402] Cleanup SafeConfigParser DeprecationWarning
- [AIRFLOW-1326][[AIRFLOW-1326][AIRFLOW-1184] Don't split argument array – it's already an array.[
- [AIRFLOW-1384] Add ARGO/CaDC as a Airflow user
- [AIRFLOW-1357] Fix scheduler zip file support
- [AIRFLOW-1382] Add working dir option to DockerOperator
- [AIRFLOW-1388] Add Cloud ML Engine operators to integration doc
- [AIRFLOW-1387] Add unicode string prefix
- [AIRFLOW-1366] Add max_tries to task instance
- [AIRFLOW-1300] Enable table creation with TBLPROPERTIES
- [AIRFLOW-1271] Add Google CloudML Training Operator
- [AIRFLOW-300] Add Google Pubsub hook and operator
- [AIRFLOW-1343] Fix dataproc label format
- [AIRFLOW-1367] Pass Content-ID To reference inline images in an email, we need to be able to add <img src="cid:{}"/> to the HTML. However currently the Content-ID (cid) is not passed, so we need to add it
- [AIRFLOW-1265] Fix celery executor parsing CELERY_SSL_ACTIVE
- [AIRFLOW-1272] Google Cloud ML Batch Prediction Operator
- [AIRFLOW-1352][AIRFLOW-1335] Revert MemoryHandler change ()[]
- [AIRFLOW-1350] Add query_uri param to Hive/SparkSQL DataProc operator
- [AIRFLOW-1334] Check if tasks are backfill on scheduler in a join
- [AIRFLOW-1343] Add Airflow default label to the dataproc operator
- [AIRFLOW-1273] Add Google Cloud ML version and model operators
- [AIRFLOW-1273]AIRFLOW-1273] Add Google Cloud ML version and model operators
- [AIRFLOW-1321] Fix hidden field key to ignore case
- [AIRFLOW-1337] Make log_format key names lowercase
- [AIRFLOW-1338][AIRFLOW-782] Add GCP dataflow hook runner change to UPDATING.md
- [AIRFLOW-801] Remove outdated docstring on BaseOperator
- [AIRFLOW-1344] Fix text encoding bug when reading logs for Python 3.5
- [AIRFLOW-1338] Fix incompatible GCP dataflow hook
- [AIRFLOW-1333] Enable copy function for Google Cloud Storage Hook
- [AIRFLOW-1337] Allow log format customization via airflow.cfg

- [AIRFLOW-1320] Update LetsBonus users in README
- [AIRFLOW-1335] Use MemoryHandler for buffered logging
- [AIRFLOW-1339] Add Drivy to the list of users
- [AIRFLOW-1275] Put 'airflow pool' into API
- [AIRFLOW-1296] Propagate SKIPPED to all downstream tasks
- [AIRFLOW-1317] Fix minor issues in API reference
- [AIRFLOW-1308] Disable nanny usage for Dask
- [AIRFLOW-1172] Support nth weekday of the month cron expression
- [AIRFLOW-936] Add clear/mark success for DAG in the UI
- [AIRFLOW-1294] Backfills can loose tasks to execute
- [AIRFLOW-1299] Support imageVersion in Google Dataproc cluster
- [AIRFLOW-1291] Update NOTICE and LICENSE files to match ASF requirements
- [AIRFLOW-1301] Add New Relic to list of companies
- [AIRFLOW-1289] Removes restriction on number of scheduler threads
- [AIRFLOW-1024] Ignore celery executor errors (#49)
- [AIRFLOW-1265] Fix exception while loading celery configurations
- [AIRFLOW-1290] set docs author to 'Apache Airflow'
- [AIRFLOW-1242] Allowing project_id to have a colon in it.
- [AIRFLOW-1282] Fix known event column sorting
- [AIRFLOW-1166] Speed up _change_state_for_tis_without_dagrun
- [AIRFLOW-1208] Speed-up cli tests
- [AIRFLOW-1192] Some enhancements to qubole_operator
- [AIRFLOW-1281] Sort variables by key field by default
- [AIRFLOW-1277] Forbid KE creation with empty fields
- [AIRFLOW-1276] Forbid event creation with end_data earlier than start_date
- [AIRFLOW-1263] Dynamic height for charts
- [AIRFLOW-1266] Increase width of gantt y axis
- [AIRFLOW-1244] Forbid creation of a pool with empty name
- [AIRFLOW-1274][HTTPSENSOR] Rename parameter params to data
- [AIRFLOW-654] Add SSL Config Option for CeleryExecutor w/ RabbitMQ - Add BROKER_USE_SSL config to give option to send AMQP messages over SSL - Can be set using usual airflow options (e.g. airflow.cfg, env vars, etc.)
- [AIRFLOW-1256] Add United Airlines to readme
- [AIRFLOW-1251] Add eRevalue to Airflow users
- [AIRFLOW-908] Print hostname at the start of cli run
- [AIRFLOW-1237] Fix IN-predicate sqlalchemy warning
- [AIRFLOW-1243] DAGs table has no default entries to show
- [AIRFLOW-1245] Fix random failure in test_trigger_dag_for_date
- [AIRFLOW-1248] Fix wrong conf name for worker timeout

- [AIRFLOW-1197] : SparkSubmitHook on_kill error
- [AIRFLOW-1191] : SparkSubmitHook custom cmd
- [AIRFLOW-1234] Cover utils.operator_helpers with UTs
- [AIRFLOW-1217] Enable Sqoop logging
- [AIRFLOW-645] Support HTTPS connections in HttpHook
- [AIRFLOW-1231] Use flask_wtf.CSRFProtect
- [AIRFLOW-1232] Remove deprecated readfp warning
- [AIRFLOW-1233] Cover utils.json with unit tests
- [AIRFLOW-1227] Remove empty column on the Logs view
- [AIRFLOW-1226] Remove empty column on the Jobs view
- [AIRFLOW-1221] Fix templating bug with DatabricksSubmitRunOperator
- [AIRFLOW-1210] Enable DbApiHook unit tests
- [AIRFLOW-1199] Fix create modal
- [AIRFLOW-1200] Forbid creation of a variable with an empty key
- [AIRFLOW-1207] Enable utils.helpers unit tests
- [AIRFLOW-1213] Add hcatalog parameters to sqoop
- [AIRFLOW-1201] Update deprecated 'nose-parameterized'
- [AIRFLOW-1186] Sort dag.get_task_instances by execution_date
- [AIRFLOW-1203] Pin Google API client version to fix OAuth issue
- [AIRFLOW-1145] Fix closest_date_partition function with before set to True If we're looking for the closest date before, we should take the latest date in the list of date before.
- [AIRFLOW-1180] Fix flask-wtf version for test_csrf_rejection
- [AIRFLOW-993] Update date inference logic
- [AIRFLOW-1170] DbApiHook insert_rows inserts parameters separately
- [AIRFLOW-1041] Do not shadow xcom_push method[]
- [AIRFLOW-860][AIRFLOW-935] Fix plugin executor import cycle and executor selection
- [AIRFLOW-1189] Fix get a DataFrame using BigQueryHook failing
- [AIRFLOW-1184] SparkSubmitHook does not split args
- [AIRFLOW-1182] SparkSubmitOperator template field
- [AIRFLOW-823] Allow specifying execution date in task_info API
- [AIRFLOW-1175] Add Pronto Tools to Airflow user list
- [AIRFLOW-1150] Fix scripts execution in sparksql hook[]
- [AIRFLOW-1141] remove crawl_for_tasks
- [AIRFLOW-1193] Add Checkr to company using Airflow
- [AIRFLOW-1168] Add closing() to all connections and cursors
- [AIRFLOW-1188] Add max_bad_records param to GoogleCloudStorageToBigQueryOperator
- [AIRFLOW-1187][AIRFLOW-1185] Fix PyPi package names in documents
- [AIRFLOW-1185] Fix PyPi URL in templates

- [AIRFLOW-XXX] Updating CHANGELOG, README, and UPDATING after 1.8.1 release
- [AIRFLOW-1181] Add delete and list functionality to gcs_hook
- [AIRFLOW-1179] Fix Pandas 0.2x breaking Google BigQuery change
- [AIRFLOW-1167] Support microseconds in FTPHook modification time
- [AIRFLOW-1173] Add Robinhood to who uses Airflow
- [AIRFLOW-945][AIRFLOW-941] Remove psycopg2 connection workaround
- [AIRFLOW-1140] DatabricksSubmitRunOperator should template the "json" field.
- [AIRFLOW-1160] Update Spark parameters for Mesos
- [AIRFLOW 1149][AIRFLOW-1149] Allow for custom filters in Jinja2 templates
- [AIRFLOW-1036] Randomize exponential backoff
- [AIRFLOW-1155] Add Tails.com to community
- [AIRFLOW-1142] Do not reset orphaned state for backfills
- [AIRFLOW-492] Make sure stat updates cannot fail a task
- [AIRFLOW-1119] Fix unload query so headers are on first row[]
- [AIRFLOW-1089] Add Spark application arguments
- [AIRFLOW-1125] Document encrypted connections
- [AIRFLOW-1122] Increase stroke width in UI
- [AIRFLOW-1138] Add missing licenses to files in scripts directory
- (AIRFLOW-11-38) [AIRFLOW-1136] Capture invalid arguments for Sqoop
- [AIRFLOW-1127] Move license notices to LICENSE
- [AIRFLOW-1118] Add evo.company to Airflow users
- [AIRFLOW-1121][AIRFLOW-1004] Fix *airflow webserver –pid* to write out pid file
- [AIRFLOW-1124] Do not set all tasks to scheduled in backfill
- [AIRFLOW-1120] Update version view to include Apache prefix
- [AIRFLOW-1091] Add script that can compare jira target against merges
- [AIRFLOW-1107] Add support for ftps non-default port
- [AIRFLOW-1000] Rebrand distribution to Apache Airflow
- [AIRFLOW-1094] Run unit tests under contrib in Travis
- [AIRFLOW-1112] Log which pool when pool is full in scheduler
- [AIRFLOW-1106] Add Groupalia/Letsbonus to the ReadMe
- [AIRFLOW-1109] Use kill signal to kill processes and log results
- [AIRFLOW-1074] Don't count queued tasks for concurrency limits
- [AIRFLOW-1095] Make ldap_auth memberOf come from configuration
- [AIRFLOW-1090] Add HBO
- [AIRFLOW-1035] Use binary exponential backoff
- [AIRFLOW-1081] Improve performance of duration chart
- [AIRFLOW-1078] Fix latest_runs endpoint for old flask versions
- [AIRFLOW-1085] Enhance the SparkSubmitOperator

- [AIRFLOW-1050] Do not count up_for_retry as not ready
- [AIRFLOW-1028] Databricks Operator for Airflow
- [AIRFLOW-1075] Security docs cleanup
- [AIRFLOW-1033][AIFRLOW-1033] Fix ti_deps for no schedule dags
- [AIRFLOW-1016] Allow HTTP HEAD request method on HTTPSensor
- [AIRFLOW-970] Load latest_runs on homepage async
- [AIRFLOW-111] Include queued tasks in scheduler concurrency check
- [AIRFLOW-1001] Fix landing times if there is no following schedule
- [AIRFLOW-1065] Add functionality for Azure Blob Storage over wasb://
- [AIRFLOW-947] Improve exceptions for unavailable Presto cluster
- [AIRFLOW-1067] use example.com in examples
- [AIRFLOW-1064] Change default sort to job_id for TaskInstanceModelView
- [AIRFLOW-1030][AIRFLOW-1] Fix hook import for HttpSensor
- [AIRFLOW-1051] Add a test for resetdb to CliTests
- [AIRFLOW-1004][AIRFLOW-276] Fix *airflow webserver -D* to run in background
- [AIRFLOW-1062] Fix DagRun#find to return correct result
- [AIRFLOW-1011] Fix bug in BackfillJob._execute() for SubDAGs
- [AIRFLOW-1038] Specify celery serialization options explicitly
- [AIRFLOW-1054] Fix broken import in test_dag
- [AIRFLOW-1007] Use Jinja sandbox for chart_data endpoint
- [AIRFLOW-719] Fix race condition in ShortCircuit, Branch and LatestOnly
- [AIRFLOW-1043] Fix doc strings of operators
- [AIRFLOW-840] Make ticket renewer python3 compatible
- [AIRFLOW-985] Extend the sqoop operator and hook
- [AIRFLOW-1034] Make it possible to connect to S3 in sigv4 regions
- [AIRFLOW-1045] Make log level configurable via airflow.cfg
- [AIRFLOW-1047] Sanitize strings passed to Markup
- [AIRFLOW-1040] Fix some small typos in comments and docstrings
- [AIRFLOW-1017] get_task_instance shouldn't throw exception when no TI
- [AIRFLOW-1006] Add config_templates to MANIFEST
- [AIRFLOW-999] Add support for Redis database
- [AIRFLOW-1009] Remove SQLOperator from Concepts page
- [AIRFLOW-1006] Move config templates to separate files
- [AIRFLOW-1005] Improve Airflow startup time
- [AIRFLOW-1010] Add convenience script for signing releases
- [AIRFLOW-995] Remove reference to actual Airflow issue
- [AIRFLOW-681] homepage doc link should pointing to apache repo not airbnb repo
- [AIRFLOW-705][AIRFLOW-706] Fix run_command bugs

- [AIRFLOW-990] Fix Py27 unicode logging in DockerOperator
- [AIRFLOW-963] Fix non-rendered code examples
- [AIRFLOW-969] Catch bad python_callable argument
- [AIRFLOW-984] Enable subclassing of SubDagOperator
- [AIRFLOW-997] Update setup.cfg to point to Apache
- [AIRFLOW-994] Add MiNODES to the official airflow user list
- [AIRFLOW-995][AIRFLOW-1] Update GitHub PR Template
- [AIRFLOW-989] Do not mark dag run successful if unfinished tasks
- [AIRFLOW-903] New configuration setting for the default dag view
- [AIRFLOW-979] Add GovTech GDS
- [AIRFLOW-933] Replace eval with literal_eval to prevent RCE
- [AIRFLOW-974] Fix mkdirs race condition
- [AIRFLOW-917] Fix formatting of error message
- [AIRFLOW-770] Refactor BaseHook so env vars are always read
- [AIRFLOW-900] Double trigger should not kill original task instance
- [AIRFLOW-900] Fixes bugs in LocalTaskJob for double run protection
- [AIRFLOW-932][AIRFLOW-932][AIRFLOW-921][AIRFLOW-910] Do not mark tasks removed when backfilling[
- [AIRFLOW-961] run onkill when SIGTERMed
- [AIRFLOW-910] Use parallel task execution for backfills
- [AIRFLOW-967] Wrap strings in native for py2 ldap compatibility
- [AIRFLOW-958] Improve tooltip readability
- AIRFLOW-959 Cleanup and reorganize .gitignore
- AIRFLOW-960 Add .editorconfig file
- [AIRFLOW-931] Do not set QUEUED in TaskInstances
- [AIRFLOW-956] Get docs working on readthedocs.org
- [AIRFLOW-954] Fix configparser ImportError
- [AIRFLOW-941] Use defined parameters for psycopg2
- [AIRFLOW-943] Update Digital First Media in users list
- [AIRFLOW-942] Add mytaxi to Airflow users
- [AIRFLOW-939] add .swp to gitginore
- [AIRFLOW-719] Prevent DAGs from ending prematurely
- [AIRFLOW-938] Use test for True in task_stats queries
- [AIRFLOW-937] Improve performance of task_stats
- [AIRFLOW-933] use ast.literal_eval rather eval because ast.literal_eval does not execute input.
- [AIRFLOW-925] Revert airflow.hooks change that cherry-pick picked
- [AIRFLOW-919] Running tasks with no start date shouldn't break a DAGs UI
- [AIRFLOW-802][AIRFLOW-1] Add spark-submit operator/hook

- [AIRFLOW-725] Use keyring to store credentials for JIRA
- [AIRFLOW-916] Remove deprecated readfp function
- [AIRFLOW-911] Add coloring and timing to tests
- [AIRFLOW-906] Update Code icon from lightning bolt to file
- [AIRFLOW-897] Prevent dagruns from failing with unfinished tasks
- [AIRFLOW-896] Remove unicode to 8-bit conversion in BigQueryOperator
- [AIRFLOW-899] Tasks in SCHEDULED state should be white in the UI instead of black
- [AIRFLOW-895] Address Apache release incompliancies
- [AIRFLOW-893][AIRFLOW-510] Fix crashing webservers when a dagrun has no start date
- [AIRFLOW-880] Make webserver serve logs in a sane way for remote logs
- [AIRFLOW-889] Fix minor error in the docstrings for BaseOperator
- [AIRFLOW-809][AIRFLOW-1] Use __eq__ ColumnOperator When Testing Booleans
- [AIRFLOW-875] Add template to HttpSensor params
- [AIRFLOW-866] Add FTPSensor
- [AIRFLOW-881] Check if SubDagOperator is in DAG context manager
- [AIRFLOW-885] Add change.org to the users list
- [AIRFLOW-836] Use POST and CSRF for state changing endpoints
- [AIRFLOW-862] Fix Unit Tests for DaskExecutor
- [AIRFLOW-887] Support future v0.16
- [AIRFLOW-886] Pass result to post_execute() hook
- [AIRFLOW-871] change logging.warn() into warning()
- [AIRFLOW-882] Remove unnecessary dag>>op assignment in docs
- [AIRFLOW-861] make pickle_info endpoint be login_required
- [AIRFLOW-869] Refactor mark success functionality
- [AIRFLOW-877] Remove .sql template extension from GCS download operator
- [AIRFLOW-826] Add Zendesk hook
- [AIRFLOW-842] do not query the DB with an empty IN clause
- [AIRFLOW-834] change raise StopIteration into return
- [AIRFLOW-832] Let debug server run without SSL
- [AIRFLOW-862] Add DaskExecutor
- [AIRFLOW-858] Configurable database name for DB operators
- [AIRFLOW-863] Example DAGs should have recent start dates
- [AIRFLOW-853] use utf8 encoding for stdout line decode
- [AIRFLOW-857] Use library assert statements instead of conditionals
- [AIRFLOW-856] Make sure execution date is set for local client
- [AIRFLOW-854] Add OKI as Airflow user
- [AIRFLOW-830][AIRFLOW-829][AIRFLOW-88] Reduce Travis log verbosity
- [AIRFLOW-814] Fix Presto*CheckOperator.__init__

- [AIRFLOW-793] Enable compressed loading in S3ToHiveTransfer
- [AIRFLOW-844] Fix cgroups directory creation
- [AIRFLOW-831] Restore import to fix broken tests
- [AIRFLOW-794] Access DAGS_FOLDER and SQL_ALCHEMY_CONN exclusively from settings
- [AIRFLOW-694] Fix config behaviour for empty envvar
- [AIRFLOW-365] Set dag.fileloc explicitly and use for Code view
- [AIRFLOW-781] Allow DataFlowOperators to accept jobs stored in GCS

3.19.6 Airflow 1.8.2, 2017-09-04

- [AIRFLOW-809][AIRFLOW-1] Use __eq__ ColumnOperator When Testing Booleans
- [AIRFLOW-1296] Propagate SKIPPED to all downstream tasks
- Re-enable caching for hadoop components
- Pin Hive and Hadoop to a specific version and create writable warehouse dir
- [AIRFLOW-1308] Disable nanny usage for Dask
- Updating CHANGELOG for 1.8.2rc1
- [AIRFLOW-1294] Backfills can loose tasks to execute
- [AIRFLOW-1291] Update NOTICE and LICENSE files to match ASF requirements
- [AIRFLOW-XXX] Set version to 1.8.2rc1
- [AIRFLOW-1160] Update Spark parameters for Mesos
- [AIRFLOW 1149][AIRFLOW-1149] Allow for custom filters in Jinja2 templates
- [AIRFLOW-1119] Fix unload query so headers are on first row[]
- [AIRFLOW-1089] Add Spark application arguments
- [AIRFLOW-1078] Fix latest_runs endpoint for old flask versions
- [AIRFLOW-1074] Don't count queued tasks for concurrency limits
- [AIRFLOW-1064] Change default sort to job_id for TaskInstanceModelView
- [AIRFLOW-1038] Specify celery serialization options explicitly
- [AIRFLOW-1036] Randomize exponential backoff
- [AIRFLOW-993] Update date inference logic
- [AIRFLOW-1167] Support microseconds in FTPHook modification time
- [AIRFLOW-1179] Fix Pandas 0.2x breaking Google BigQuery change
- [AIRFLOW-1263] Dynamic height for charts
- [AIRFLOW-1266] Increase width of gantt y axis
- [AIRFLOW-1290] set docs author to 'Apache Airflow'
- [AIRFLOW-1282] Fix known event column sorting
- [AIRFLOW-1166] Speed up _change_state_for_tis_without_dagrun
- [AIRFLOW-1192] Some enhancements to qubole_operator
- [AIRFLOW-1281] Sort variables by key field by default
- [AIRFLOW-1244] Forbid creation of a pool with empty name

- [AIRFLOW-1243] DAGs table has no default entries to show
- [AIRFLOW-1227] Remove empty column on the Logs view
- [AIRFLOW-1226] Remove empty column on the Jobs view
- [AIRFLOW-1199] Fix create modal
- [AIRFLOW-1200] Forbid creation of a variable with an empty key
- [AIRFLOW-1186] Sort dag.get_task_instances by execution_date
- [AIRFLOW-1145] Fix closest_date_partition function with before set to True If we're looking for the closest date before, we should take the latest date in the list of date before.
- [AIRFLOW-1180] Fix flask-wtf version for test_csrf_rejection
- [AIRFLOW-1170] DbApiHook insert_rows inserts parameters separately
- [AIRFLOW-1150] Fix scripts execution in sparksql hook[]
- [AIRFLOW-1168] Add closing() to all connections and cursors
- [AIRFLOW-XXX] Updating CHANGELOG, README, and UPDATING after 1.8.1 release

3.19.7 Airflow 1.8.1, 2017-05-09

- [AIRFLOW-1142] SubDAG Tasks Not Executed Even Though All Dependencies Met
- [AIRFLOW-1138] Add licenses to files in scripts directory
- [AIRFLOW-1127] Move license notices to LICENSE instead of NOTICE
- [AIRFLOW-1124] Do not set all task instances to scheduled on backfill
- [AIRFLOW-1120] Update version view to include Apache prefix
- [AIRFLOW-1062] DagRun#find returns wrong result if external_trigger=False is specified
- [AIRFLOW-1054] Fix broken import on test_dag
- [AIRFLOW-1050] Retries ignored - regression
- [AIRFLOW-1033] TypeError: can't compare datetime.datetime to None
- [AIRFLOW-1017] get_task_instance should return None instead of throw an exception for non-existent TIs
- [AIRFLOW-1011] Fix bug in BackfillJob._execute() for SubDAGs
- [AIRFLOW-1004] *airflow webserver -D* runs in foreground
- [AIRFLOW-1001] Landing Time shows "unsupported operand type(s) for -: 'datetime.datetime' and 'NoneType'" on example_subdag_operator
- [AIRFLOW-1000] Rebrand to Apache Airflow instead of Airflow
- [AIRFLOW-989] Clear Task Regression
- [AIRFLOW-974] airflow.util.file mkdir has a race condition
- [AIRFLOW-906] Update Code icon from lightning bolt to file
- [AIRFLOW-858] Configurable database name for DB operators
- [AIRFLOW-853] ssh_execute_operator.py stdout decode default to ASCII
- [AIRFLOW-832] Fix debug server
- [AIRFLOW-817] Trigger dag fails when using CLI + API
- [AIRFLOW-816] Make sure to pull nvd3 from local resources

- [AIRFLOW-815] Add previous/next execution dates to available default variables.
- [AIRFLOW-813] Fix unterminated unit tests in tests.job (tests/job.py)
- [AIRFLOW-812] Scheduler job terminates when there is no dag file
- [AIRFLOW-806] UI should properly ignore DAG doc when it is None
- [AIRFLOW-794] Consistent access to DAGS_FOLDER and SQL_ALCHEMY_CONN
- [AIRFLOW-785] ImportError if cgroupspy is not installed
- [AIRFLOW-784] Cannot install with funcsigs > 1.0.0
- [AIRFLOW-780] The UI no longer shows broken DAGs
- [AIRFLOW-777] dag_is_running is initlialized to True instead of False
- [AIRFLOW-719] Skipped operations make DAG finish prematurely
- [AIRFLOW-694] Empty env vars do not overwrite non-empty config values
- [AIRFLOW-492] Insert into dag_stats table results into failed task while task itself succeeded
- [AIRFLOW-139] Executing VACUUM with PostgresOperator
- [AIRFLOW-111] DAG concurrency is not honored
- [AIRFLOW-88] Improve clarity Travis CI reports

3.19.8 Airflow 1.8.0, 2017-03-12

- [AIRFLOW-900] Double trigger should not kill original task instance
- [AIRFLOW-900] Fixes bugs in LocalTaskJob for double run protection
- [AIRFLOW-932] Do not mark tasks removed when backfilling
- [AIRFLOW-961] run onkill when SIGTERMed
- [AIRFLOW-910] Use parallel task execution for backfills
- [AIRFLOW-967] Wrap strings in native for py2 ldap compatibility
- [AIRFLOW-941] Use defined parameters for psycopg2
- [AIRFLOW-719] Prevent DAGs from ending prematurely
- [AIRFLOW-938] Use test for True in task_stats queries
- [AIRFLOW-937] Improve performance of task_stats
- [AIRFLOW-933] use ast.literal_eval rather eval because ast.literal_eval does not execute input.
- [AIRFLOW-925] Revert airflow.hooks change that cherry-pick picked
- [AIRFLOW-919] Running tasks with no start date shouldn't break a DAGs UI
- [AIRFLOW-802] Add spark-submit operator/hook
- [AIRFLOW-897] Prevent dagruns from failing with unfinished tasks
- [AIRFLOW-861] make pickle_info endpoint be login_required
- [AIRFLOW-853] use utf8 encoding for stdout line decode
- [AIRFLOW-856] Make sure execution date is set for local client
- [AIRFLOW-830][AIRFLOW-829][AIRFLOW-88] Reduce Travis log verbosity
- [AIRFLOW-831] Restore import to fix broken tests
- [AIRFLOW-794] Access DAGS_FOLDER and SQL_ALCHEMY_CONN exclusively from settings

- [AIRFLOW-694] Fix config behaviour for empty envvar
- [AIRFLOW-365] Set dag.fileloc explicitly and use for Code view
- [AIRFLOW-931] Do not set QUEUED in TaskInstances
- [AIRFLOW-899] Tasks in SCHEDULED state should be white in the UI instead of black
- [AIRFLOW-895] Address Apache release incompliancies
- [AIRFLOW-893][AIRFLOW-510] Fix crashing webservers when a dagrun has no start date
- [AIRFLOW-793] Enable compressed loading in S3ToHiveTransfer
- [AIRFLOW-863] Example DAGs should have recent start dates
- [AIRFLOW-869] Refactor mark success functionality
- [AIRFLOW-856] Make sure execution date is set for local client
- [AIRFLOW-814] Fix Presto*CheckOperator.__init__
- [AIRFLOW-844] Fix cgroups directory creation
- [AIRFLOW-816] Use static nvd3 and d3
- [AIRFLOW-821] Fix py3 compatibility
- [AIRFLOW-817] Check for None value of execution_date in endpoint
- [AIRFLOW-822] Close db before exception
- [AIRFLOW-815] Add prev/next execution dates to template variables
- [AIRFLOW-813] Fix unterminated unit tests in SchedulerJobTest
- [AIRFLOW-813] Fix unterminated scheduler unit tests
- [AIRFLOW-806] UI should properly ignore DAG doc when it is None
- [AIRFLOW-812] Fix the scheduler termination bug.
- [AIRFLOW-780] Fix dag import errors no longer working
- [AIRFLOW-783] Fix py3 incompatibility in BaseTaskRunner
- [AIRFLOW-810] Correct down_revision dag_id/state index creation
- [AIRFLOW-807] Improve scheduler performance for large DAGs
- [AIRFLOW-798] Check return_code before forcing termination
- [AIRFLOW-139] Let psycopg2 handle autocommit for PostgresHook
- [AIRFLOW-776] Add missing cgroups devel dependency
- [AIRFLOW-777] Fix expression to check if a DagRun is in running state
- [AIRFLOW-785] Don’t import CgroupTaskRunner at global scope
- [AIRFLOW-784] Pin funcsigs to 1.0.0
- [AIRFLOW-624] Fix setup.py to not import airflow.version as version
- [AIRFLOW-779] Task should fail with specific message when deleted
- [AIRFLOW-778] Fix completely broken MetastorePartitionSensor
- [AIRFLOW-739] Set pickle_info log to debug
- [AIRFLOW-771] Make S3 logs append instead of clobber
- [AIRFLOW-773] Fix flaky datetime addition in api test
- [AIRFLOW-219][AIRFLOW-398] Cgroups + impersonation

- [AIRFLOW-683] Add jira hook, operator and sensor
- [AIRFLOW-762] Add Google DataProc delete operator
- [AIRFLOW-760] Update systemd config
- [AIRFLOW-759] Use previous dag_run to verify depend_on_past
- [AIRFLOW-757] Set child_process_log_directory default more sensible
- [AIRFLOW-692] Open XCom page to super-admins only
- [AIRFLOW-737] Fix HDFS Sensor directory.
- [AIRFLOW-747] Fix retry_delay not honoured
- [AIRFLOW-558] Add Support for dag.catchup=(True|False) Option
- [AIRFLOW-489] Allow specifying execution date in trigger_dag API
- [AIRFLOW-738] Commit deleted xcom items before insert
- [AIRFLOW-729] Add Google Cloud Dataproc cluster creation operator
- [AIRFLOW-728] Add Google BigQuery table sensor
- [AIRFLOW-741] Log to debug instead of info for app.py
- [AIRFLOW-731] Fix period bug for NamedHivePartitionSensor
- [AIRFLOW-740] Pin jinja2 to < 2.9.0
- [AIRFLOW-663] Improve time units for task performance charts
- [AIRFLOW-665] Fix email attachments
- [AIRFLOW-734] Fix SMTP auth regression when not using user/pass
- [AIRFLOW-702] Fix LDAP Regex Bug
- [AIRFLOW-717] Add Cloud Storage updated sensor
- [AIRFLOW-695] Retries do not execute because dagrun is in FAILED state
- [AIRFLOW-673] Add operational metrics test for SchedulerJob
- [AIRFLOW-727] try_number is not increased
- [AIRFLOW-715] A more efficient HDFS Sensor:
- [AIRFLOW-716] Allow AVRO BigQuery load-job without schema
- [AIRFLOW-718] Allow the query URI for DataProc Pig
- Log needs to be part of try/catch block
- [AIRFLOW-721] Descendant process can disappear before termination
- [AIRFLOW-403] Bash operator's kill method leaves underlying processes running
- [AIRFLOW-657] Add AutoCommit Parameter for MSSQL
- [AIRFLOW-641] Improve pull request instructions
- [AIRFLOW-685] Add test for MySqlHook.bulk_load()
- [AIRFLOW-686] Match auth backend config section
- [AIRFLOW-691] Add SSH KeepAlive option to SSH_hook
- [AIRFLOW-709] Use same engine for migrations and reflection
- [AIRFLOW-700] Update to reference to web authentication documentation
- [AIRFLOW-649] Support non-sched DAGs in LatestOnlyOp

- [AIRFLOW-712] Fix AIRFLOW-667 to use proper HTTP error properties
- [AIRFLOW-710] Add OneFineStay as official user
- [AIRFLOW-703][AIRFLOW-1] Stop Xcom being cleared too early
- [AIRFLOW-679] Stop concurrent task instances from running
- [AIRFLOW-704][AIRFLOW-1] Fix invalid syntax in BQ hook
- [AIRFLOW-667] Handle BigQuery 503 error
- [AIRFLOW-680] Disable connection pool for commands
- [AIRFLOW-678] Prevent scheduler from double triggering TIs
- [AIRFLOW-677] Kill task if it fails to heartbeat
- [AIRFLOW-674] Ability to add descriptions for DAGs
- [AIRFLOW-682] Bump MAX_PERIODS to make mark_success work for large DAGs
- Use jdk selector to set required jdk
- [AIRFLOW-647] Restore dag.get_active_runs
- [AIRFLOW-662] Change seasons to months in project description
- [AIRFLOW-656] Add dag/task/date index to xcom table
- [AIRFLOW-658] Improve schema_update_options in GCP
- [AIRFLOW-41] Fix pool oversubscription
- [AIRFLOW-489] Add API Framework
- [AIRFLOW-653] Add some missing endpoint tests
- [AIRFLOW-652] Remove obsolete endpoint
- [AIRFLOW-345] Add contrib ECSOperator
- [AIRFLOW-650] Adding Celect to user list
- [AIRFLOW-510] Filter Paused Dags, show Last Run & Trigger Dag
- [AIRFLOW-643] Improve date handling for sf_hook
- [AIRFLOW-638] Add schema_update_options to GCP ops
- [AIRFLOW-640] Install and enable nose-ignore-docstring
- [AIRFLOW-639]AIRFLOW-639] Alphasort package names
- [AIRFLOW-375] Fix pylint errors
- [AIRFLOW-347] Show empty DAG runs in tree view
- [AIRFLOW-628] Adding SalesforceHook to contrib/hooks
- [AIRFLOW-514] hive hook loads data from pandas DataFrame into hive and infers types
- [AIRFLOW-565] Fixes DockerOperator on Python3.x
- [AIRFLOW-635] Encryption option for S3 hook
- [AIRFLOW-137] Fix max_active_runs on clearing tasks
- [AIRFLOW-343] Fix schema plumbing in HiveServer2Hook
- [AIRFLOW-130] Fix ssh operator macosx
- [AIRFLOW-633] Show TI attributes in TI view
- [AIRFLOW-626][AIRFLOW-1] HTML Content does not show up when sending email with attachment

- [AIRFLOW-533] Set autocommit via set_autocommit
- [AIRFLOW-629] stop pinning lxml
- [AIRFLOW-464] Add setdefault method to Variable
- [AIRFLOW-626][AIRFLOW-1] HTML Content does not show up when sending email with attachment
- [AIRFLOW-591] Add datadog hook & sensor
- [AIRFLOW-561] Add RedshiftToS3Transfer operator
- [AIRFLOW-570] Pass root to date form on gantt
- [AIRFLOW-504] Store fractional seconds in MySQL tables
- [AIRFLOW-623] LDAP attributes not always a list
- [AIRFLOW-611] source_format in BigQueryBaseCursor
- [AIRFLOW-619] Fix exception in Gannt chart
- [AIRFLOW-618] Cast DateTimes to avoid sqllite errors
- [AIRFLOW-422] Add JSON endpoint for task info
- [AIRFLOW-616][AIRFLOW-617] Minor fixes to PR tool UX
- [AIRFLOW-179] Fix DbApiHook with non-ASCII chars
- [AIRFLOW-566] Add timeout while fetching logs
- [AIRFLOW-615] Set graph glyphicon first
- [AIRFLOW-609] Add application_name to PostgresHook
- [AIRFLOW-604] Revert .first() to .one()
- [AIRFLOW-370] Create AirflowConfigException in exceptions.py
- [AIRFLOW-582] Fixes TI.get_dagrun filter (removes start_date)
- [AIRFLOW-568] Fix double task_stats count if a DagRun is active
- [AIRFLOW-585] Fix race condition in backfill execution loop
- [AIRFLOW-580] Prevent landscape warning on .format
- [AIRFLOW-597] Check if content is None, not false-equivalent
- [AIRFLOW-586] test_dag_v1 fails from 0 to 3 a.m.
- [AIRFLOW-453] Add XCom Admin Page
- [AIRFLOW-588] Add Google Cloud Storage Object sensor[]
- [AIRFLOW-592] example_xcom import Error
- [AIRFLOW-587] Fix incorrect scope for Google Auth[]
- [AIRFLOW-589] Add templatable job_name[]
- [AIRFLOW-227] Show running config in config view
- [AIRFLOW-319]AIRFLOW-319] xcom push response in HTTP Operator
- [AIRFLOW-385] Add symlink to latest scheduler log directory
- [AIRFLOW-583] Fix decode error in gcs_to_bq
- [AIRFLOW-96] s3_conn_id using environment variable
- [AIRFLOW-575] Clarify tutorial and FAQ about *schedule_interval* always inheriting from DAG object
- [AIRFLOW-577] Output BigQuery job for improved debugging

- [AIRFLOW-560] Get URI & SQLA engine from Connection
- [AIRFLOW-518] Require DataProfilingMixin for Variables CRUD
- [AIRFLOW-553] Fix load path for filters.js
- [AIRFLOW-554] Add Jinja support to Spark-sql
- [AIRFLOW-550] Make ssl config check empty string safe
- [AIRFLOW-500] Use id for github allowed teams
- [AIRFLOW-556] Add UI PR guidelines
- [AIRFLOW-358][AIRFLOW-430] Add *connections* cli
- [AIRFLOW-548] Load DAGs immediately & continually
- [AIRFLOW-539] Updated BQ hook and BQ operator to support Standard SQL.
- [AIRFLOW-378] Add string casting to params of spark-sql operator
- [AIRFLOW-544] Add Pause/Resume toggle button
- [AIRFLOW-333][AIRFLOW-258] Fix non-module plugin components
- [AIRFLOW-542] Add tooltip to DAGs links icons
- [AIRFLOW-530] Update docs to reflect connection environment var has to be in uppercase
- [AIRFLOW-525] Update template_fields in Qubole Op
- [AIRFLOW-480] Support binary file download from GCS
- [AIRFLOW-198] Implement latest_only_operator
- [AIRFLOW-91] Add SSL config option for the webserver
- [AIRFLOW-191] Fix connection leak with PostgreSQL backend
- [AIRFLOW-512] Fix 'bellow' typo in docs & comments
- [AIRFLOW-509][AIRFLOW-1] Create operator to delete tables in BigQuery
- [AIRFLOW-498] Remove hard-coded gcp project id
- [AIRFLOW-505] Support unicode characters in authors' names
- [AIRFLOW-494] Add per-operator success/failure metrics
- [AIRFLOW-488] Fix test_simple fail
- [AIRFLOW-468] Update Panda requirement to 0.17.1
- [AIRFLOW-159] Add cloud integration section + GCP documentation
- [AIRFLOW-477][AIRFLOW-478] Restructure security section for clarity
- [AIRFLOW-467] Allow defining of project_id in BigQueryHook
- [AIRFLOW-483] Change print to logging statement
- [AIRFLOW-475] make the segment granularity in Druid hook configurable

3.19.9 Airflow 1.7.2

- [AIRFLOW-463] Link Airflow icon to landing page
- [AIRFLOW-149] Task Dependency Engine + Why Isn't My Task Running View
- [AIRFLOW-361] Add default failure handler for the Qubole Operator
- [AIRFLOW-353] Fix dag run status update failure

- [AIRFLOW-447] Store source URIs in Python 3 compatible list
- [AIRFLOW-443] Make module names unique when importing
- [AIRFLOW-444] Add Google authentication backend
- [AIRFLOW-446][AIRFLOW-445] Adds missing dataproc submit options
- [AIRFLOW-431] Add CLI for CRUD operations on pools
- [AIRFLOW-329] Update Dag Overview Page with Better Status Columns
- [AIRFLOW-360] Fix style warnings in models.py
- [AIRFLOW-425] Add white fill for null state tasks in tree view.
- [AIRFLOW-69] Use dag runs in backfill jobs
- [AIRFLOW-415] Make dag_id not found error clearer
- [AIRFLOW-416] Use ordinals in README's company list
- [AIRFLOW-369] Allow setting default DAG orientation
- [AIRFLOW-410] Add 2 Q/A to the FAQ in the docs
- [AIRFLOW-407] Add different colors for some sensors
- [AIRFLOW-414] Improve error message for missing FERNET_KEY
- [AIRFLOW-406] Sphinx/rst fixes
- [AIRFLOW-412] Fix lxml dependency
- [AIRFLOW-413] Fix unset path bug when backfilling via pickle
- [AIRFLOW-78] Airflow clear leaves dag_runs
- [AIRFLOW-402] Remove NamedHivePartitionSensor static check, add docs
- [AIRFLOW-394] Add an option to the Task Duration graph to show cumulative times
- [AIRFLOW-404] Retry download if unpacking fails for hive
- [AIRFLOW-276] Gunicorn rolling restart
- [AIRFLOW-399] Remove dags/testdruid.py
- [AIRFLOW-400] models.py/DAG.set_dag_runs_state() does not correctly set state
- [AIRFLOW-395] Fix colon/equal signs typo for resources in default config
- [AIRFLOW-397] Documentation: Fix typo "instatiating" to "instantiating"
- [AIRFLOW-395] Remove trailing commas from resources in config
- [AIRFLOW-388] Add a new chart for Task_Tries for each DAG
- [AIRFLOW-322] Fix typo in FAQ section
- [AIRFLOW-375] Pylint fixes
- limit scope to user email only AIRFLOW-386
- [AIRFLOW-383] Cleanup example qubole operator dag
- [AIRFLOW-160] Parse DAG files through child processes
- [AIRFLOW-381] Manual UI Dag Run creation: require dag_id field
- [AIRFLOW-373] Enhance CLI variables functionality
- [AIRFLOW-379] Enhance Variables page functionality: import/export variables
- [AIRFLOW-331] modify the LDAP authentication config lines in 'Security' sample codes

- [AIRFLOW-356][AIRFLOW-355][AIRFLOW-354] Replace nobr, enable DAG only exists locally message, change edit DAG icon
- [AIRFLOW-362] Import __future__ division
- [AIRFLOW-359] Pin flask-login to 0.2.11
- [AIRFLOW-261] Add bcc and cc fields to EmailOperator
- [AIRFLOW-348] Fix code style warnings
- [AIRFLOW-349] Add metric for number of zombies killed
- [AIRFLOW-340] Remove unused dependency on Babel
- [AIRFLOW-339]: Ability to pass a flower conf file
- [AIRFLOW-341][operators] Add resource requirement attributes to operators
- [AIRFLOW-335] Fix simple style errors/warnings
- [AIRFLOW-337] Add __repr__ to VariableAccessor and VariableJsonAccessor
- [AIRFLOW-334] Fix using undefined variable
- [AIRFLOW-315] Fix blank lines code style warnings
- [AIRFLOW-306] Add Spark-sql Hook and Operator
- [AIRFLOW-327] Add rename method to the FTPHook
- [AIRFLOW-321] Fix a wrong code example about tests/dags
- [AIRFLOW-316] Always check DB state for Backfill Job execution
- [AIRFLOW-264] Adding workload management for Hive
- [AIRFLOW-297] support exponential backoff option for retry delay
- [AIRFLOW-31][AIRFLOW-200] Add note to updating.md
- [AIRFLOW-307] There is no __neq__ python magic method.
- [AIRFLOW-309] Add requirements of develop dependencies to docs
- [AIRFLOW-307] Rename __neq__ to __ne__ python magic method.
- [AIRFLOW-313] Fix code style for sqoop_hook.py
- [AIRFLOW-311] Fix wrong path in CONTRIBUTING.md
- [AIRFLOW-24] DataFlow Java Operator
- [AIRFLOW-308] Add link to refresh DAG within DAG view header
- [AIRFLOW-314] Fix BigQuery cursor run_table_upsert method
- [AIRFLOW-298] fix incubator diclaimer in docs
- [AIRFLOW-284] HiveServer2Hook fix for cursor scope for get_results
- [AIRFLOW-260] More graceful exit when issues can't be closed
- [AIRFLOW-260] Handle case when no version is found
- [AIRFLOW-228] Handle empty version list in PR tool
- [AIRFLOW-302] Improve default squash commit message
- [AIRFLOW-187] Improve prompt styling
- [AIRFLOW-187] Fix typo in argument name
- [AIRFLOW-187] Move "Close XXX" message to end of squash commit

- [AIRFLOW-247] Add EMR hook, operators and sensors. Add AWS base hook
- [AIRFLOW-301] Fix broken unit test
- [AIRFLOW-100] Add execution_date_fn to ExternalTaskSensor
- [AIRFLOW-282] Remove PR Tool logic that depends on version formatting
- [AIRFLOW-291] Add index for state in TI table
- [AIRFLOW-269] Add some unit tests for PostgreSQL
- [AIRFLOW-296] template_ext is being treated as a string rather than a tuple in qubole operator
- [AIRFLOW-286] Improve FTPHook to implement context manager interface
- [AIRFLOW-243] Create NamedHivePartitionSensor
- [AIRFLOW-246] Improve dag_stats endpoint query
- [AIRFLOW-189] Highlighting of Parent/Child nodes in Graphs
- [ARFLOW-255] Check dagrun timeout when comparing active runs
- [AIRFLOW-281] Add port to mssql_hook
- [AIRFLOW-285] Use Airflow 2.0 style imports for all remaining hooks/operators
- [AIRFLOW-40] Add LDAP group filtering feature.
- [AIRFLOW-277] Multiple deletions does not work in Task Instances view if using SQLite backend
- [AIRFLOW-200] Make hook/operator imports lazy, and print proper exceptions
- [AIRFLOW-283] Make store_to_xcom_key a templated field in GoogleCloudStorageDownloadOperator
- [AIRFLOW-278] Support utf-8 ecoding for SQL
- [AIRFLOW-280] clean up tmp druid table no matter if an ingestion job succeeds or not
- [AIRFLOW-274] Add XCom functionality to GoogleCloudStorageDownloadOperator
- [AIRFLOW-273] Create an svg version of the airflow logo.
- [AIRFLOW-275] Update contributing guidelines
- [AIRFLOW-244] Modify hive operator to inject analysis data
- [AIRFLOW-162] Allow variable to be accessible into templates
- [AIRFLOW-248] Add Apache license header to all files
- [AIRFLOW-263] Remove temp backtick file
- [AIRFLOW-252] Raise Sqlite exceptions when deleting tasks instance in WebUI
- [AIRFLOW-180] Fix timeout behavior for sensors
- [AIRFLOW-262] Simplify commands in MANIFEST.in
- [AIRFLOW-31] Add zope dependency
- [AIRFLOW-6] Remove dependency on Highcharts
- [AIRFLOW-234] make task that aren't *running* self-terminate
- [AIRFLOW-256] Fix test_scheduler_reschedule heartrate
- Add Python 3 compatibility fix
- [AIRFLOW-31] Use standard imports for hooks/operators
- [AIRFLOW-173] Initial implementation of FileSensor
- [AIRFLOW-224] Collect orphaned tasks and reschedule them

- [AIRFLOW-239] Fix tests indentation
- [AIRFLOW-225] Better units for task duration graph
- [AIRFLOW-241] Add testing done section to PR template
- [AIRFLOW-222] Show duration of task instances in ui
- [AIRFLOW-231] Do not eval user input in PrestoHook
- [AIRFLOW-216] Add Sqoop Hook and Operator
- [AIRFLOW-171] Add upgrade notes on email and S3 to 1.7.1.2
- [AIRFLOW-238] Make compatible with flask-admin 1.4.1
- [AIRFLOW-230] [HiveServer2Hook] adding multi statements support
- [AIRFLOW-142] setup_env.sh doesn't download hive tarball if hdp is specified as distro
- [AIRFLOW-223] Make parametrable the IP on which Flower binds to
- [AIRFLOW-218] Added option to enable webserver gunicorn access/err logs
- [AIRFLOW-213] Add "Closes #X" phrase to commit messages
- [AIRFLOW-68] Align start_date with the schedule_interval
- [AIRFLOW-9] Improving docs to meet Apache's standards
- [AIRFLOW-131] Make XCom.clear more selective
- [AIRFLOW-214] Fix occasion of detached taskinstance
- [AIRFLOW-206] Add commit to close PR
- [AIRFLOW-206] Always load local log files if they exist
- [AIRFLOW-211] Fix JIRA "resolve" vs "close" behavior
- [AIRFLOW-64] Add note about relative DAGS_FOLDER
- [AIRFLOW-114] Sort plugins dropdown
- [AIRFLOW-209] Add scheduler tests and improve lineage handling
- [AIRFLOW-207] Improve JIRA auth workflow
- [AIRFLOW-187] Improve PR tool UX
- [AIRFLOW-155] Documentation of Qubole Operator
- Optimize and refactor process_dag
- [AIRFLOW-185] Handle empty versions list
- [AIRFLOW-201] Fix for HiveMetastoreHook + kerberos
- [AIRFLOW-202]: Fixes stray print line
- [AIRFLOW-196] Fix bug that exception is not handled in HttpSensor
- [AIRFLOW-195] : Add toggle support to subdag clearing in the CLI
- [AIRFLOW-23] Support for Google Cloud DataProc
- [AIRFLOW-25] Configuration for Celery always required
- [AIRFLOW-190] Add codecov and remove download count
- [AIRFLOW-168] Correct evaluation of @once schedule
- [AIRFLOW-183] Fetch log from remote when worker returns 4xx/5xx response
- [AIRFLOW-181] Fix failing unpacking of hadoop by redownloading

- [AIRFLOW-176] remove unused formatting key
- [AIRFLOW-167]: Add dag_state option in cli
- [AIRFLOW-178] Fix bug so that zip file is detected in DAG folder
- [AIRFLOW-176] Improve PR Tool JIRA workflow
- AIRFLOW-45: Support Hidden Airflow Variables
- [AIRFLOW-175] Run git-reset before checkout in PR tool
- [AIRFLOW-157] Make PR tool Py3-compat; add JIRA command
- [AIRFLOW-170] Add missing @apply_defaults

3.19.10 Airflow 1.7.1, 2016-05-19

- Fix : Don't treat premature tasks as could_not_run tasks
- AIRFLOW-92 Avoid unneeded upstream_failed session closes apache/airflow#1485
- Add logic to lock DB and avoid race condition
- Handle queued tasks from multiple jobs/executors
- AIRFLOW-52 Warn about overwriting tasks in a DAG
- Fix corner case with joining processes/queues (#1473)
- [AIRFLOW-52] Fix bottlenecks when working with many tasks
- Add columns to toggle extra detail in the connection list view.
- Log the number of errors when importing DAGs
- Log dagbag metrics dupplicate messages in queue into Statsd (#1406)
- Clean up issue template (#1419)
- correct missed arg.foreground to arg.daemon in cli
- Reinstate imports for github enterprise auth
- Use os.execvp instead of subprocess.Popen for the webserver
- Revert from using "–foreground" to "–daemon"
- Implement a Cloudant hook
- Add missing args to *airflow clear*
- Fixed a bug in the scheduler: num_runs used where runs intended
- Add multiprocessing support to the scheduler
- Partial fix to make sure next_run_date cannot be None
- Support list/get/set variables in the CLI
- Properly handle BigQuery booleans in BigQuery hook.
- Added the ability to view XCom variables in webserver
- Change DAG.tasks from a list to a dict
- Add support for zipped dags
- Stop creating hook on instantiating of S3 operator
- User subquery in views to find running DAGs
- Prevent DAGs from being reloaded on every scheduler iteration

- Add a missing word to docs
- Document the parameters of *DbApiHook*
- added oracle operator with existing oracle hook
- Add PyOpenSSL to Google cloud gcp_api.
- Remove executor error unit test
- Add DAG inference, deferral, and context manager
- Don't return error when writing files to Google cloud storage.
- Fix GCS logging for gcp_api.
- Ensure attr is in scope for error message
- Fixing misnamed PULL_REQUEST_TEMPLATE
- Extract non_pooled_task_slot_count into a configuration param
- Update plugins.rst for clarity on the example (#1309)
- Fix s3 logging issue
- Add twitter feed example dag
- Github ISSUE_TEMPLATE & PR_TEMPLATE cleanup
- Reduce logger verbosity
- Adding a PR Template
- Add Lucid to list of users
- Fix usage of asciiart
- Use session instead of outdated main_session for are_dependencies_met
- Fix celery flower port allocation
- Fix for missing edit actions due to flask-admin upgrade
- Fix typo in comment in prioritize_queued method
- Add HipchatOperator
- Include all example dags in backfill unit test
- Make sure skipped jobs are actually skipped
- Fixing a broken example dag, example_skip_dag.py
- Add consistent and thorough signal handling and logging
- Allow Operators to specify SKIPPED status internally
- Update docstring for executor trap unit test
- Doc: explain the usage of Jinja templating for templated params
- Don't schedule runs before the DAG's start_date
- Fix infinite retries with pools, with test
- Fix handling of deadlocked jobs
- Show only Airflow's deprecation warnings
- Set DAG_FOLDER for unit tests
- Missing comma in setup.py
- Deprecate args and kwargs in BaseOperator

- Raise deep scheduler exceptions to force a process restart.
- Change inconsistent example DAG owners
- Fix module path of send_email_smtp in configuration
- added Gentner Lab to list of users
- Increase timeout time for unit test
- Fix reading strings from conf
- CHORE - Remove Trailing Spaces
- Fix SSHExecuteOperator crash when using a custom ssh port
- Add note about airflow components to template
- Rewrite BackfillJob logic for clarity
- Add unit tests
- Fix miscellaneous bugs and clean up code
- Fix logic for determining DagRun states
- Make SchedulerJob not run EVERY queued task
- Improve BackfillJob handling of queued/deadlocked tasks
- Introduce ignore_depends_on_past parameters
- Use Popen with CeleryExecutor
- Rename user table to users to avoid conflict with postgres
- Beware of negative pool slots.
- Add support for calling_format from boto to S3_Hook
- Add pypi meta data and sync version number
- Set dags_are_paused_at_creation's default value to True
- Resurface S3Log class eaten by rebase/push -f
- Add missing session.commit() at end of initdb
- Validate that subdag tasks have pool slots available, and test
- Use urlparse for remote GCS logs, and add unit tests
- Make webserver worker timeout configurable
- Fixed scheduling for @once interval
- Use psycopg2's API for serializing postgres cell values
- Make the provide_session decorator more robust
- update link to Lyft's website
- use num_shards instead of partitions to be consistent with batch ingestion
- Add documentation links to README
- Update docs with separate configuration section
- Fix airflow.utils deprecation warning code being Python 3 incompatible
- Extract dbapi cell serialization into its own method
- Set Postgres autocommit as supported only if server version is < 7.4
- Use refactored utils module in unit test imports

- Add changelog for 1.7.0
- Use LocalExecutor on Travis if possible
- remove unused logging,errno, MiniHiveCluster imports
- remove extra import of logging lib
- Fix required gcloud version
- Refactoring utils into smaller submodules
- Properly measure number of task retry attempts
- Add function to get configuration as dict, plus unit tests
- Merge branch 'master' into hivemeta_sasl
- Add wiki link to README.md
- [hotfix] make email.Utils > email.utils for py3
- Add the missing "Date" header to the warning e-mails
- Add the missing "Date" header to the warning e-mails
- Check name of SubDag class instead of class itself
- [hotfix] removing repo_token from .coveralls.yml
- Set the service_name in coverals.yml
- Fixes #1223
- Update Airflow docs for remote logging
- Add unit tests for trapping Executor errors
- Make sure Executors properly trap errors
- Fix HttpOpSensorTest to use fake resquest session
- Linting
- Add an example on pool usage in the documentation
- Add two methods to bigquery hook's base cursor: run_table_upsert, which adds a table or updates an existing table; and run_grant_dataset_view_access, which grants view access to a given dataset for a given table.
- Tasks references upstream and downstream tasks using strings instead of references
- Fix typos in models.py
- Fix broken links in documentation
- [hotfix] fixing the Scheduler CLI to make dag_id optional
- Update link to Common Pitfalls wiki page in README
- Allow disabling periodic committing when inserting rows with DbApiHook
- added Glassdoor to "who uses airflow"
- Fix typo preventing from launching webserver
- Documentation badge
- Fixing ISSUE_TEMPLATE name to include .md suffix
- Adding an ISSUE_TEMPLATE to ensure that issues are adequately defined
- Linting & debugging
- Refactoring the CLI to be data-driven

- Updating the Bug Reporting protocol in the Contributing.md file
- Fixing the docs
- clean up references to old session
- remove session reference
- resolve conflict
- clear xcom data when task instance starts
- replace main_session with @provide_session
- Add extras to installation.rst
- Changes to Contributing to reflect more closely the current state of development.
- Modifying README to link to the wiki committer list
- docs: fixes a spelling mistake in default config
- Set killMode to 'control-group' for webservice.service
- Set KillMode to 'control-group' for worker.service
- Linting
- Fix WebHdfsSensor
- Adding more licenses to pass checks
- fixing landscape's config
- [hotfix] typo that made it in master
- [hotfix] fixing landscape requirement detection
- Make testing on hive conditional
- Merge remote-tracking branch 'upstream/master' into minicluster
- Update README.md
- Throwing in a few license to pass the build
- Adding a reqs.txt for landscape.io
- Pointing to a reqs file
- Some linting
- Adding a .landscape.yml file
- badge for pypi version
- Add license and ignore for sql and csv
- Use correct connection id
- Use correct table name
- Provide data for ci tests
- new badge for showing staleness of reqs
- removing requirements.txt as it is uni-dimensional
- Make it work on py3
- Remove decode for logging
- Also keep py2 compatible
- More py3 fixes

- Convert to bytes for py3 compat
- Make sure to be py3 compatible
- Use unicodecsv to make it py3 compatible
- Replace tab with spaces Remove unused import
- Merge remote-tracking branch 'upstream/master'
- Support decimal types in MySQL to GCS
- Make sure to write binary as string can be unicode
- Ignore metastore
- More impyla fixes
- Test HivemetaStore if python 2
- Allow users to set hdfs_namenode_principal in HDFSHook config
- Add tests for Hiveserver2 and fix some issues from impyla
- Merge branch 'impyla' into minicluster
- This patch allows for testing of hive operators and hooks. Sasl is used (NoSasl in connection string is not possible). Tests have been adjusted.
- Treat SKIPPED and SUCCESS the same way when evaluating depends_on_past=True
- fix bigquery hook
- version cap for gcp_api
- Fix typo when returning VerticaHook
- Adding fernet key to use it as part of stdout commands
- Adding support for ssl parameters. (picking up from jthomas123)
- more detail in error message.
- make sure paths don't conflict bc of trailing /
- change gcs_hook to self.hook
- refactor remote log read/write and add GCS support
- Only use multipart upload in S3Hook if file is large enough
- Merge branch 'airbnb/master'
- Add GSSAPI SASL to HiveMetaStoreHook.
- Add warning for deprecated setting
- Use kerberos_service_name = 'hive' as standard instead of 'impala'.
- Use GSSAPI instead of KERBEROS and provide backwards compatibility
- ISSUE-1123 Use impyla instead of pyhs2
- set celery_executor to use queue name as exchange

3.20 FAQ

3.20.1 Why isn't my task getting scheduled?

There are very many reasons why your task might not be getting scheduled. Here are some of the common causes:

- Does your script "compile", can the Airflow engine parse it and find your DAG object. To test this, you can run `airflow list_dags` and confirm that your DAG shows up in the list. You can also run `airflow list_tasks foo_dag_id --tree` and confirm that your task shows up in the list as expected. If you use the CeleryExecutor, you may want to confirm that this works both where the scheduler runs as well as where the worker runs.
- Does the file containing your DAG contain the string "airflow" and "DAG" somewhere in the contents? When searching the DAG directory, Airflow ignores files not containing "airflow" and "DAG" in order to prevent the DagBag parsing from importing all python files collocated with user's DAGs.
- Is your `start_date` set properly? The Airflow scheduler triggers the task soon after the `start_date + scheduler_interval` is passed.
- Is your `schedule_interval` set properly? The default `schedule_interval` is one day (`datetime.timedelta(1)`). You must specify a different `schedule_interval` directly to the DAG object you instantiate, not as a `default_param`, as task instances do not override their parent DAG's `schedule_interval`.
- Is your `start_date` beyond where you can see it in the UI? If you set your `start_date` to some time say 3 months ago, you won't be able to see it in the main view in the UI, but you should be able to see it in the `Menu -> Browse ->Task Instances`.
- Are the dependencies for the task met. The task instances directly upstream from the task need to be in a `success` state. Also, if you have set `depends_on_past=True`, the previous task instance needs to have succeeded (except if it is the first run for that task). Also, if `wait_for_downstream=True`, make sure you understand what it means. You can view how these properties are set from the `Task Instance Details` page for your task.
- Are the DagRuns you need created and active? A DagRun represents a specific execution of an entire DAG and has a state (running, success, failed, …). The scheduler creates new DagRun as it moves forward, but never goes back in time to create new ones. The scheduler only evaluates `running` DagRuns to see what task instances it can trigger. Note that clearing tasks instances (from the UI or CLI) does set the state of a DagRun back to running. You can bulk view the list of DagRuns and alter states by clicking on the schedule tag for a DAG.
- Is the `concurrency` parameter of your DAG reached? `concurrency` defines how many `running` task instances a DAG is allowed to have, beyond which point things get queued.
- Is the `max_active_runs` parameter of your DAG reached? `max_active_runs` defines how many `running` concurrent instances of a DAG there are allowed to be.

You may also want to read the Scheduler section of the docs and make sure you fully understand how it proceeds.

3.20.2 How do I trigger tasks based on another task's failure?

Check out the `Trigger Rule` section in the Concepts section of the documentation.

3.20.3 Why are connection passwords still not encrypted in the metadata db after I installed airflow[crypto]?

Check out the `Securing Connections` section in the How-to Guides section of the documentation.

3.20.4 What's the deal with `start_date`?

`start_date` is partly legacy from the pre-DagRun era, but it is still relevant in many ways. When creating a new DAG, you probably want to set a global `start_date` for your tasks using `default_args`. The first DagRun to be created will be based on the `min(start_date)` for all your task. From that point on, the scheduler creates new DagRuns based on your `schedule_interval` and the corresponding task instances run as your dependencies are met. When introducing new tasks to your DAG, you need to pay special attention to `start_date`, and may want to reactivate inactive DagRuns to get the new task onboarded properly.

We recommend against using dynamic values as `start_date`, especially `datetime.now()` as it can be quite confusing. The task is triggered once the period closes, and in theory an `@hourly` DAG would never get to an hour after now as `now()` moves along.

Previously we also recommended using rounded `start_date` in relation to your `schedule_interval`. This meant an `@hourly` would be at `00:00` minutes:seconds, a `@daily` job at midnight, a `@monthly` job on the first of the month. This is no longer required. Airflow will now auto align the `start_date` and the `schedule_interval`, by using the `start_date` as the moment to start looking.

You can use any sensor or a `TimeDeltaSensor` to delay the execution of tasks within the schedule interval. While `schedule_interval` does allow specifying a `datetime.timedelta` object, we recommend using the macros or cron expressions instead, as it enforces this idea of rounded schedules.

When using `depends_on_past=True` it's important to pay special attention to `start_date` as the past dependency is not enforced only on the specific schedule of the `start_date` specified for the task. It's also important to watch DagRun activity status in time when introducing new `depends_on_past=True`, unless you are planning on running a backfill for the new task(s).

Also important to note is that the tasks `start_date`, in the context of a backfill CLI command, get overridden by the backfill's command `start_date`. This allows for a backfill on tasks that have `depends_on_past=True` to actually start, if that wasn't the case, the backfill just wouldn't start.

3.20.5 How can I create DAGs dynamically?

Airflow looks in your `DAGS_FOLDER` for modules that contain `DAG` objects in their global namespace, and adds the objects it finds in the `DagBag`. Knowing this all we need is a way to dynamically assign variable in the global namespace, which is easily done in python using the `globals()` function for the standard library which behaves like a simple dictionary.

```python
def create_dag(dag_id):
    """
    A function returning a DAG object.
    """

    return DAG(dag_id)


for i in range(10):
    dag_id = f'foo_{i}'
    globals()[dag_id] = DAG(dag_id)

    # or better, call a function that returns a DAG object!
    other_dag_id = f'bar_{i}'
    globals()[other_dag_id] = create_dag(other_dag_id)
```

3.20.6 What are all the `airflow run` commands in my process list?

There are many layers of `airflow run` commands, meaning it can call itself.

- Basic `airflow run`: fires up an executor, and tell it to run an `airflow run --local` command. If using Celery, this means it puts a command in the queue for it to run remotely on the worker. If using LocalExecutor, that translates into running it in a subprocess pool.

- Local `airflow run --local`: starts an `airflow run --raw` command (described below) as a subprocess and is in charge of emitting heartbeats, listening for external kill signals and ensures some cleanup takes place if the subprocess fails.
- Raw `airflow run --raw` runs the actual operator's execute method and performs the actual work.

3.20.7 How can my airflow dag run faster?

There are a few variables we could control to improve airflow dag performance:

- `parallelism`: This variable controls the number of task instances that runs simultaneously across the whole Airflow cluster. User could increase the parallelism variable in the `airflow.cfg`.
- `concurrency`: The Airflow scheduler will run no more than `concurrency` task instances for your DAG at any given time. Concurrency is defined in your Airflow DAG. If you do not set the concurrency on your DAG, the scheduler will use the default value from the `dag_concurrency` entry in your `airflow.cfg`.
- `task_concurrency`: This variable controls the number of concurrent running task instances across `dag_runs` per task.
- `max_active_runs`: the Airflow scheduler will run no more than `max_active_runs` DagRuns of your DAG at a given time. If you do not set the `max_active_runs` in your DAG, the scheduler will use the default value from the `max_active_runs_per_dag` entry in your `airflow.cfg`.
- `pool`: This variable controls the number of concurrent running task instances assigned to the pool.

3.20.8 How can we reduce the airflow UI page load time?

If your dag takes long time to load, you could reduce the value of `default_dag_run_display_number` configuration in `airflow.cfg` to a smaller value. This configurable controls the number of dag run to show in UI with default value 25.

3.20.9 How to fix Exception: Global variable explicit_defaults_for_timestamp needs to be on (1)?

This means `explicit_defaults_for_timestamp` is disabled in your mysql server and you need to enable it by:

1. Set `explicit_defaults_for_timestamp = 1` under the mysqld section in your my.cnf file.
2. Restart the Mysql server.

3.20.10 How to reduce airflow dag scheduling latency in production?

- `max_threads`: Scheduler will spawn multiple threads in parallel to schedule dags. This is controlled by `max_threads` with default value of 2. User should increase this value to a larger value(e.g numbers of cpus where scheduler runs - 1) in production.
- `scheduler_heartbeat_sec`: User should consider to increase `scheduler_heartbeat_sec` config to a higher value(e.g 60 secs) which controls how frequent the airflow scheduler gets the heartbeat and updates the job's entry in database.

3.21 Macros reference

Variables and macros can be used in templates (see the *Jinja Templating* section)

The following come for free out of the box with Airflow. Additional custom macros can be added globally through ORM Extensions, or at a DAG level through the `DAG.user_defined_macros` argument.

3.21.1 Default Variables

The Airflow engine passes a few variables by default that are accessible in all templates

Variable	Description
`{{ ds }}`	the execution date as `YYYY-MM-DD`
`{{ ds_nodash }}`	the execution date as `YYYYMMDD`
`{{ prev_ds }}`	the previous execution date as `YYYY-MM-DD` if `{{ ds }}` is `2018-01-08` anc
`{{ prev_ds_nodash }}`	the previous execution date as `YYYYMMDD` if exists, else `None`
`{{ next_ds }}`	the next execution date as `YYYY-MM-DD` if `{{ ds }}` is `2018-01-01` and sc
`{{ next_ds_nodash }}`	the next execution date as `YYYYMMDD` if exists, else `None`
`{{ yesterday_ds }}`	the day before the execution date as `YYYY-MM-DD`
`{{ yesterday_ds_nodash }}`	the day before the execution date as `YYYYMMDD`
`{{ tomorrow_ds }}`	the day after the execution date as `YYYY-MM-DD`
`{{ tomorrow_ds_nodash }}`	the day after the execution date as `YYYYMMDD`
`{{ ts }}`	same as `execution_date.isoformat()`. Example: `2018-01-01T00:0`
`{{ ts_nodash }}`	same as `ts` without `-`, `:` and TimeZone info. Example: `20180101T000000`
`{{ ts_nodash_with_tz }}`	same as `ts` without `-` and `:`. Example: `20180101T000000+0000`
`{{ execution_date }}`	the execution_date (pendulum.Pendulum)
`{{ prev_execution_date }}`	the previous execution date (if available) (pendulum.Pendulum)
`{{ prev_execution_date_success }}`	execution date from prior succesful dag run (if available) (pendulum.Pendulum)
`{{ prev_start_date_success }}`	start date from prior successful dag run (if available) (pendulum.Pendulum)
`{{ next_execution_date }}`	the next execution date (pendulum.Pendulum)
`{{ dag }}`	the DAG object
`{{ task }}`	the Task object
`{{ macros }}`	a reference to the macros package, described below
`{{ task_instance }}`	the task_instance object
`{{ end_date }}`	same as `{{ ds }}`
`{{ latest_date }}`	same as `{{ ds }}`
`{{ ti }}`	same as `{{ task_instance }}`
`{{ params }}`	a reference to the user-defined params dictionary which can be overridden by the di
`{{ var.value.my_var }}`	global defined variables represented as a dictionary
`{{ var.json.my_var.path }}`	global defined variables represented as a dictionary with deserialized JSON object,
`{{ task_instance_key_str }}`	a unique, human-readable key to the task instance formatted `{dag_id}_{task_`
`{{ conf }}`	the full configuration object located at `airflow.configuration.conf` whic
`{{ run_id }}`	the `run_id` of the current DAG run
`{{ dag_run }}`	a reference to the DagRun object
`{{ test_mode }}`	whether the task instance was called using the CLI's test subcommand

Note that you can access the object's attributes and methods with simple dot notation. Here are some examples of what is possible: `{{ task.owner }}`, `{{ task.task_id }}`, `{{ ti.hostname }}`, … Refer to the models documentation for more information on the objects' attributes and methods.

The `var` template variable allows you to access variables defined in Airflow's UI. You can access them as either plain-text or JSON. If you use JSON, you are also able to walk nested structures, such as dictionaries like: `{{ var.json.`

```
my_dict_var.key1 }}
```

3.21.2 Macros

Macros are a way to expose objects to your templates and live under the `macros` namespace in your templates.

A few commonly used libraries and methods are made available.

Variable	Description
`macros.datetime`	The standard lib's `datetime.datetime`
`macros.timedelta`	The standard lib's `datetime.timedelta`
`macros.dateutil`	A reference to the `dateutil` package
`macros.time`	The standard lib's `datetime.time`
`macros.uuid`	The standard lib's `uuid`
`macros.random`	The standard lib's `random`

Some airflow specific macros are also defined:

`airflow.macros.`**`ds_add`**(*ds*, *days*)

Add or subtract days from a YYYY-MM-DD

Parameters

- **`ds`** (*`str`*) – anchor date in `YYYY-MM-DD` format to add to
- **`days`** (*`int`*) – number of days to add to the ds, you can use negative values

```
>>> ds_add('2015-01-01', 5)
'2015-01-06'
>>> ds_add('2015-01-06', -5)
'2015-01-01'
```

`airflow.macros.`**`ds_format`**(*ds*, *input_format*, *output_format*)

Takes an input string and outputs another string as specified in the output format

Parameters

- **`ds`** (*`str`*) – input string which contains a date
- **`input_format`** (*`str`*) – input string format. E.g. %Y-%m-%d
- **`output_format`** (*`str`*) – output string format E.g. %Y-%m-%d

```
>>> ds_format('2015-01-01', "%Y-%m-%d", "%m-%d-%y")
'01-01-15'
>>> ds_format('1/5/2015', "%m/%d/%Y",  "%Y-%m-%d")
'2015-01-05'
```

`airflow.macros.`**`random`**() → x in the interval [0, 1).

`airflow.macros.hive.`**`closest_ds_partition`**(*table*, *ds*, *before=True*, *schema='default'*, *metastore_conn_id='metastore_default'*)

This function finds the date in a list closest to the target date. An optional parameter can be given to get the closest before or after.

Parameters

- **`table`** (*`str`*) – A hive table name
- **`ds`** (*`list[datetime.date]`*) – A datestamp `%Y-%m-%d` e.g. `yyyy-mm-dd`
- **`before`** (*`bool or None`*) – closest before (True), after (False) or either side of ds

Returns The closest date

Return type str or None

```
>>> tbl = 'airflow.static_babynames_partitioned'
>>> closest_ds_partition(tbl, '2015-01-02')
'2015-01-01'
```

airflow.macros.hive.**max_partition**(*table*, *schema='default'*, *field=None*, *filter_map=None*, *metastore_conn_id='metastore_default'*)

Gets the max partition for a table.

Parameters

- **schema** (*str*) – The hive schema the table lives in
- **table** (*str*) – The hive table you are interested in, supports the dot notation as in "my_database.my_table", if a dot is found, the schema param is disregarded
- **metastore_conn_id** (*str*) – The hive connection you are interested in. If your default is set you don't need to use this parameter.
- **filter_map** (*map*) – partition_key:partition_value map used for partition filtering, e.g. {'key1': 'value1', 'key2': 'value2'}. Only partitions matching all partition_key:partition_value pairs will be considered as candidates of max partition.
- **field** (*str*) – the field to get the max value from. If there's only one partition field, this will be inferred

```
>>> max_partition('airflow.static_babynames_partitioned')
'2015-01-01'
```

3.22 API Reference

3.22.1 Operators

Operators allow for generation of certain types of tasks that become nodes in the DAG when instantiated. All operators derive from `BaseOperator` and inherit many attributes and methods that way.

There are 3 main types of operators:

- Operators that performs an **action**, or tell another system to perform an action
- **Transfer** operators move data from one system to another
- **Sensors** are a certain type of operator that will keep running until a certain criterion is met. Examples include a specific file landing in HDFS or S3, a partition appearing in Hive, or a specific time of the day. Sensors are derived from `BaseSensorOperator` and run a poke method at a specified `poke_interval` until it returns `True`.

3.22.1.1 BaseOperator

All operators are derived from `BaseOperator` and acquire much functionality through inheritance. Since this is the core of the engine, it's worth taking the time to understand the parameters of `BaseOperator` to understand the primitive features that can be leveraged in your DAGs.

3.22.1.2 BaseSensorOperator

All sensors are derived from `BaseSensorOperator`. All sensors inherit the `timeout` and `poke_interval` on top of the `BaseOperator` attributes.

3.22.1.3 Operators packages

All operators are in the following packages:

airflow.operators

Submodules

airflow.operators.bash_operator

Module Contents

class airflow.operators.bash_operator.**BashOperator**(*bash_command*, *env=None*, *output_encoding='utf-8'*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Execute a Bash script, command or set of commands.

See also:

For more information on how to use this operator, take a look at the guide: *BashOperator*

If BaseOperator.do_xcom_push is True, the last line written to stdout will also be pushed to an XCom when the bash command completes

Parameters

- **bash_command** (*str*) – The command, set of commands or reference to a bash script (must be '.sh') to be executed. (templated)
- **env** (*dict*) – If env is not None, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process environment, which is the default behavior. (templated)
- **output_encoding** (*str*) – Output encoding of bash command

On execution of this operator the task will be up for retry when exception is raised. However, if a sub-command exits with non-zero value Airflow will not recognize it as failure unless the whole shell exits with a failure. The easiest way of achieving this is to prefix the command with `set -e;` Example:

```
bash_command = "set -e; python3 script.py '{{ next_execution_date }}'"
```

template_fields = ['bash_command', 'env']

template_ext = ['.sh', '.bash']

ui_color = #f0ede4

execute(*self*, *context*)
Execute the bash command in a temporary directory which will be cleaned afterwards

on_kill(*self*)

`airflow.operators.check_operator`

Module Contents

class `airflow.operators.check_operator.`**`CheckOperator`**(*sql:str*, *conn_id:Optional[str]=None*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Performs checks against a db. The `CheckOperator` expects a sql query that will return a single row. Each value on that first row is evaluated using python `bool` casting. If any of the values return `False` the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average.

This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alerts without stopping the progress of the DAG.

Note that this is an abstract class and get_db_hook needs to be defined. Whereas a get_db_hook is hook that gets a single record from an external source.

Parameters **`sql`** (*`str`*) – the sql to be executed. (templated)

`template_fields :Iterable[str] = ['sql']`

`template_ext :Iterable[str] = ['.hql', '.sql']`

`ui_color = #fff7e6`

`execute`(*self*, *context=None*)

`get_db_hook`(*self*)

`airflow.operators.check_operator.`**`_convert_to_float_if_possible`**(*s*)

`A small helper function to convert a string to a numeric value if appropriate`

Parameters **`s`** (*`str`*) – the string to be converted

class `airflow.operators.check_operator.`**`ValueCheckOperator`**(*sql:str*, *pass_value:Any*, *tolerance:Any=None*, *conn_id:Optional[str]=None*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Performs a simple value check using sql code.

Note that this is an abstract class and get_db_hook needs to be defined. Whereas a get_db_hook is hook that gets a single record from an external source.

Parameters **`sql`** (*`str`*) – the sql to be executed. (templated)

`__mapper_args__`

`template_fields` `:Iterable[str] = ['sql', 'pass_value']`

`template_ext` `:Iterable[str] = ['.hql', '.sql']`

`ui_color` `= #fff7e6`

`execute`(*self*, *context=None*)

`_to_float`(*self*, *records*)

`_get_string_matches`(*self*, *records*, *pass_value_conv*)

`_get_numeric_matches`(*self*, *numeric_records*, *numeric_pass_value_conv*)

`get_db_hook`(*self*)

class `airflow.operators.check_operator.`**`IntervalCheckOperator`**(*table:str, metrics_thresholds:Dict[str, int], date_filter_column:Optional[str]='ds', days_back:SupportsAbs[int]=-7, ratio_formula:Optional[str]='max_over_min', ignore_zero:Optional[bool]=True, conn_id:Optional[str]=None, *args, **kwargs*)

Bases: `airflow.models.BaseOperator`

Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.

Note that this is an abstract class and get_db_hook needs to be defined. Whereas a get_db_hook is hook that gets a single record from an external source.

Parameters

- **`table`** (`str`) – the table name
- **`days_back`** (`int`) – number of days between ds and the ds we want to check against. Defaults to 7 days
- **`ratio_formula`** (`str`) – which formula to use to compute the ratio between the two metrics. Assuming cur is the metric of today and ref is the metric to today - days_back.

  max_over_min: computes max(cur, ref) / min(cur, ref) relative_diff: computes abs(cur-ref) / ref

  Default: 'max_over_min'
- **`ignore_zero`** (`bool`) – whether we should ignore zero metrics
- **`metrics_threshold`** (`dict`) – a dictionary of ratios indexed by metrics

`__mapper_args__`

`template_fields` `:Iterable[str] = ['sql1', 'sql2']`

`template_ext` `:Iterable[str] = ['.hql', '.sql']`

`ui_color` `= #fff7e6`

`ratio_formulas`

`execute`(*self*, *context=None*)

`get_db_hook`(*self*)

airflow.operators.dagrun_operator

Module Contents

class airflow.operators.dagrun_operator.**DagRunOrder**(*run_id=None*, *payload=None*)

class airflow.operators.dagrun_operator.**TriggerDagRunOperator**(*trigger_dag_id*, *python_callable=None*, *execution_date=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Triggers a DAG run for a specified `dag_id`

Parameters

- **trigger_dag_id** (*str*) – the dag_id to trigger (templated)
- **python_callable** (*python callable*) – a reference to a python function that will be called while passing it the `context` object and a placeholder object `obj` for your callable to fill and return if you want a DagRun created. This `obj` object contains a `run_id` and `payload` attribute that you can modify in your function. The `run_id` should be a unique identifier for that DAG run, and the payload has to be a picklable object that will be made available to your tasks while executing that DAG run. Your function header should look like `def foo(context, dag_run_obj):`
- **execution_date** (*str or datetime.datetime*) – Execution date for the dag (templated)

template_fields = ['trigger_dag_id', 'execution_date']

ui_color = #ffefeb

execute(*self*, *context*)

`airflow.operators.docker_operator`

Module Contents

class airflow.operators.docker_operator.**DockerOperator**(*image, api_version=None, command=None, cpus=1.0, docker_url='unix://var/run/docker.sock', environment=None, force_pull=False, mem_limit=None, host_tmp_dir=None, network_mode=None, tls_ca_cert=None, tls_client_cert=None, tls_client_key=None, tls_hostname=None, tls_ssl_version=None, tmp_dir='/tmp/airflow', user=None, volumes=None, working_dir=None, xcom_all=False, docker_conn_id=None, dns=None, dns_search=None, auto_remove=False, shm_size=None, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Execute a command inside a docker container.

A temporary directory is created on the host and mounted into a container to allow storing files that together exceed the default disk size of 10GB in a container. The path to the mounted directory can be accessed via the environment variable `AIRFLOW_TMP_DIR`.

If a login to a private registry is required prior to pulling the image, a Docker connection needs to be configured in Airflow and the connection ID be provided with the parameter `docker_conn_id`.

Parameters

- **image** (*str*) – Docker image from which to create the container. If image tag is omitted, "latest" will be used.
- **api_version** (*str*) – Remote API version. Set to `auto` to automatically detect the server's version.
- **auto_remove** (*bool*) – Auto-removal of the container on daemon side when the container's process exits. The default is False.
- **command** (*str or list*) – Command to be run in the container. (templated)
- **cpus** (*float*) – Number of CPUs to assign to the container. This value gets multiplied with 1024. See https://docs.docker.com/engine/reference/run/#cpu-share-constraint
- **dns** (*list[str]*) – Docker custom DNS servers
- **dns_search** (*list[str]*) – Docker custom DNS search domain
- **docker_url** (*str*) – URL of the host running the docker daemon. Default is unix://var/run/docker.sock

- **environment** (*dict*) – Environment variables to set in the container. (templated)
- **force_pull** (*bool*) – Pull the docker image on every run. Default is False.
- **mem_limit** (*float or str*) – Maximum amount of memory the container can use. Either a float value, which represents the limit in bytes, or a string like `128m` or `1g`.
- **host_tmp_dir** (*str*) – Specify the location of the temporary directory on the host which will be mapped to tmp_dir. If not provided defaults to using the standard system temp directory.
- **network_mode** (*str*) – Network mode for the container.
- **tls_ca_cert** (*str*) – Path to a PEM-encoded certificate authority to secure the docker connection.
- **tls_client_cert** (*str*) – Path to the PEM-encoded certificate used to authenticate docker client.
- **tls_client_key** (*str*) – Path to the PEM-encoded key used to authenticate docker client.
- **tls_hostname** (*str or bool*) – Hostname to match against the docker server certificate or False to disable the check.
- **tls_ssl_version** (*str*) – Version of SSL to use when communicating with docker daemon.
- **tmp_dir** (*str*) – Mount point inside the container to a temporary directory created on the host by the operator. The path is also made available via the environment variable `AIRFLOW_TMP_DIR` inside the container.
- **user** (*int or str*) – Default user inside the docker container.
- **volumes** (*list*) – List of volumes to mount into the container, e.g. `['/host/path:/container/path', '/host/path2:/container/path2:ro']`.
- **working_dir** (*str*) – Working directory to set on the container (equivalent to the -w switch the docker client)
- **xcom_all** (*bool*) – Push all the stdout or just the last line. The default is False (last line).
- **docker_conn_id** (*str*) – ID of the Airflow connection to use
- **shm_size** (*int*) – Size of `/dev/shm` in bytes. The size must be greater than 0. If omitted uses system default.

template_fields = ['command', 'environment']

template_ext = ['.sh', '.bash']

get_hook (*self*)

execute (*self*, *context*)

get_command (*self*)

on_kill (*self*)

__get_tls_config (*self*)

airflow.operators.druid_check_operator

Module Contents

class airflow.operators.druid_check_operator.**DruidCheckOperator**(*sql*, *druid_broker_conn_id='druid_broker_d*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.CheckOperator*

Performs checks against Druid. The `DruidCheckOperator` expects a sql query that will return a single row. Each value on that first row is evaluated using python `bool` casting. If any of the values return `False` the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average. This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alerts without stopping the progress of the DAG.

Parameters

- **sql** (*str*) – the sql to be executed
- **druid_broker_conn_id** (*str*) – reference to the druid broker

get_db_hook(*self*)
Return the druid db api hook.

get_first(*self*, *sql*)
Executes the druid sql to druid broker and returns the first resulting row.

Parameters **sql** (*str*) – the sql statement to be executed (str)

execute(*self*, *context=None*)

airflow.operators.dummy_operator

Module Contents

class airflow.operators.dummy_operator.**DummyOperator**(**args*, ***kwargs*)
Bases: *airflow.models.BaseOperator*

Operator that does literally nothing. It can be used to group tasks in a DAG.

ui_color = #e8f7e4

execute(*self*, *context*)

`airflow.operators.email_operator`

Module Contents

class `airflow.operators.email_operator.`**`EmailOperator`**(*to*, *subject*, *html_content*, *files=None*, *cc=None*, *bcc=None*, *mime_subtype='mixed'*, *mime_charset='utf-8'*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Sends an email.

Parameters

- **`to`** (*`list or string (comma or semicolon delimited)`*) – list of emails to send the email to. (templated)
- **`subject`** (*`str`*) – subject line for the email. (templated)
- **`html_content`** (*`str`*) – content of the email, html markup is allowed. (templated)
- **`files`** (*`list`*) – file names to attach in email
- **`cc`** (*`list or string (comma or semicolon delimited)`*) – list of recipients to be added in CC field
- **`bcc`** (*`list or string (comma or semicolon delimited)`*) – list of recipients to be added in BCC field
- **`mime_subtype`** (*`str`*) – MIME sub content type
- **`mime_charset`** (*`str`*) – character set parameter added to the Content-Type header.

`template_fields = ['to', 'subject', 'html_content']`

`template_ext = ['.html']`

`ui_color = #e6faf9`

`execute`(*self*, *context*)

`airflow.operators.generic_transfer`

Module Contents

class `airflow.operators.generic_transfer.`**`GenericTransfer`**(*sql*, *destination_table*, *source_conn_id*, *destination_conn_id*, *preoperator=None*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Moves data from a connection to another, assuming that they both provide the required methods in their respective hooks. The source hook needs to expose a *get_records* method, and the destination a *insert_rows* method.

This is meant to be used on small-ish datasets that fit in memory.

Parameters

- **`sql`** (*`str`*) – SQL query to execute against the source database. (templated)
- **`destination_table`** (*`str`*) – target table. (templated)

- **source_conn_id** (*str*) – source connection
- **destination_conn_id** (*str*) – source connection
- **preoperator** (*str or list[str]*) – sql statement or list of statements to be executed prior to loading the data. (templated)

template_fields = ['sql', 'destination_table', 'preoperator']

template_ext = ['.sql', '.hql']

ui_color = #b0f07c

execute (*self*, *context*)

airflow.operators.hive_operator

Module Contents

class airflow.operators.hive_operator.**HiveOperator**(*hql*, *hive_cli_conn_id='hive_cli_default'*, *schema='default'*, *hiveconfs=None*, *hiveconf_jinja_translate=False*, *script_begin_tag=None*, *run_as_owner=False*, *mapred_queue=None*, *mapred_queue_priority=None*, *mapred_job_name=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Executes hql code or hive script in a specific Hive database.

Parameters

- **hql** (*str*) – the hql to be executed. Note that you may also use a relative path from the dag file of a (template) hive script. (templated)
- **hive_cli_conn_id** (*str*) – reference to the Hive database. (templated)
- **hiveconfs** (*dict*) – if defined, these key value pairs will be passed to hive as `-hiveconf "key"="value"`
- **hiveconf_jinja_translate** (*bool*) – when True, hiveconf-type templating ${var} gets translated into jinja-type templating {{ var }} and ${hiveconf:var} gets translated into jinja-type templating {{ var }}. Note that you may want to use this along with the `DAG(user_defined_macros=myargs)` parameter. View the DAG object documentation for more details.
- **script_begin_tag** (*str*) – If defined, the operator will get rid of the part of the script before the first occurrence of *script_begin_tag*
- **mapred_queue** (*str*) – queue used by the Hadoop CapacityScheduler. (templated)
- **mapred_queue_priority** (*str*) – priority within CapacityScheduler queue. Possible settings include: VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW
- **mapred_job_name** (*str*) – This name will appear in the jobtracker. This can make monitoring easier.

template_fields = ['hql', 'schema', 'hive_cli_conn_id', 'mapred_queue', 'hiveconfs', '

template_ext = ['.hql', '.sql']

ui_color = #f0e4ec

get_hook(*self*)

prepare_template(*self*)

execute(*self*, *context*)

dry_run(*self*)

on_kill(*self*)

airflow.operators.hive_stats_operator

Module Contents

class airflow.operators.hive_stats_operator.**HiveStatsCollectionOperator**(*table*, *partition*, *extra_exprs=None*, *col_blacklist=None*, *assignment_func=None*, *metastore_conn_id='metastore_de*
presto_conn_id='presto_defa
mysql_conn_id='airflow_db
**args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Gathers partition statistics using a dynamically generated Presto query, inserts the stats into a MySql table with this format. Stats overwrite themselves if you rerun the same date/partition.

```
CREATE TABLE hive_stats (
    ds VARCHAR(16),
    table_name VARCHAR(500),
    metric VARCHAR(200),
    value BIGINT
);
```

Parameters

- **table** (*str*) – the source table, in the format `database.table_name`. (templated)
- **partition** (*dict of {col:value}*) – the source partition. (templated)
- **extra_exprs** (*dict*) – dict of expression to run against the table where keys are metric names and values are Presto compatible expressions
- **col_blacklist** (*list*) – list of columns to blacklist, consider blacklisting blobs, large json columns, …
- **assignment_func** (*function*) – a function that receives a column name and a type, and returns a dict of metric names and an Presto expressions. If None is returned, the global defaults are applied. If an empty dictionary is returned, no stats are computed for that column.

template_fields = ['table', 'partition', 'ds', 'dttm']

ui_color = #aff7a6

get_default_exprs (*self*, *col*, *col_type*)

execute (*self*, *context=None*)

airflow.operators.hive_to_druid

Module Contents

airflow.operators.hive_to_druid.**LOAD_CHECK_INTERVAL = 5**

airflow.operators.hive_to_druid.**DEFAULT_TARGET_PARTITION_SIZE = 5000000**

class airflow.operators.hive_to_druid.**HiveToDruidTransfer** (*sql*, *druid_datasource*, *ts_dim*, *metric_spec=None*, *hive_cli_conn_id='hive_cli_default'*, *druid_ingest_conn_id='druid_ingest_default'*, *metastore_conn_id='metastore_default'*, *hadoop_dependency_coordinates=None*, *intervals=None*, *num_shards=-1*, *target_partition_size=-1*, *query_granularity='NONE'*, *segment_granularity='DAY'*, *hive_tblproperties=None*, *job_properties=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from Hive to Druid, [del]note that for now the data is loaded into memory before being pushed to Druid, so this operator should be used for smallish amount of data.[/del]

Parameters

- **sql** (*str*) – SQL query to execute against the Druid database. (templated)
- **druid_datasource** (*str*) – the datasource you want to ingest into in druid
- **ts_dim** (*str*) – the timestamp dimension
- **metric_spec** (*list*) – the metrics you want to define for your data
- **hive_cli_conn_id** (*str*) – the hive connection id
- **druid_ingest_conn_id** (*str*) – the druid ingest connection id
- **metastore_conn_id** (*str*) – the metastore connection id
- **hadoop_dependency_coordinates** (*list[str]*) – list of coordinates to squeeze int the ingest json
- **intervals** (*list*) – list of time intervals that defines segments, this is passed as is to the json object. (templated)
- **hive_tblproperties** (*dict*) – additional properties for tblproperties in hive for the staging table

- **job_properties** (*dict*) – additional properties for job

template_fields = ['sql', 'intervals']

template_ext = ['.sql']

execute (*self*, *context*)

construct_ingest_query (*self*, *static_path*, *columns*)
Builds an ingest query for an HDFS TSV load.

Parameters

- **static_path** (*str*) – The path on hdfs where the data is
- **columns** (*list*) – List of all the columns that are available

airflow.operators.hive_to_mysql

Module Contents

class airflow.operators.hive_to_mysql.**HiveToMySqlTransfer** (*sql*, *mysql_table*, *hiveserver2_conn_id='hiveserver2_default'*, *mysql_conn_id='mysql_default'*, *mysql_preoperator=None*, *mysql_postoperator=None*, *bulk_load=False*, *hive_conf=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from Hive to MySQL, note that for now the data is loaded into memory before being pushed to MySQL, so this operator should be used for smallish amount of data.

Parameters

- **sql** (*str*) – SQL query to execute against Hive server. (templated)
- **mysql_table** (*str*) – target MySQL table, use dot notation to target a specific database. (templated)
- **mysql_conn_id** (*str*) – source mysql connection
- **hiveserver2_conn_id** (*str*) – destination hive connection
- **mysql_preoperator** (*str*) – sql statement to run against mysql prior to import, typically use to truncate of delete in place of the data coming in, allowing the task to be idempotent (running the task twice won't double load data). (templated)
- **mysql_postoperator** (*str*) – sql statement to run against mysql after the import, typically used to move data from staging to production and issue cleanup commands. (templated)
- **bulk_load** (*bool*) – flag to use bulk_load option. This loads mysql directly from a tab-delimited text file using the LOAD DATA LOCAL INFILE command. This option requires an extra connection parameter for the destination MySQL connection: {'local_infile': true}.

template_fields = ['sql', 'mysql_table', 'mysql_preoperator', 'mysql_postoperator']

template_ext = ['.sql']

ui_color = #a0e08c

execute (*self*, *context*)

airflow.operators.hive_to_samba_operator

Module Contents

class airflow.operators.hive_to_samba_operator.**Hive2SambaOperator**(*hql*, *destination_filepath*, *samba_conn_id='samba_default'*, *hiveserver2_conn_id='hiveserver2_de*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Executes hql code in a specific Hive database and loads the results of the query as a csv to a Samba location.

Parameters

- **hql** (*str*) – the hql to be exported. (templated)
- **destination_filepath** (*str*) – the file path to where the file will be pushed onto samba
- **samba_conn_id** (*str*) – reference to the samba destination
- **hiveserver2_conn_id** (*str*) – reference to the hiveserver2 service

template_fields = ['hql', 'destination_filepath']

template_ext = ['.hql', '.sql']

execute(*self*, *context*)

airflow.operators.http_operator

Module Contents

class airflow.operators.http_operator.**SimpleHttpOperator**(*endpoint*, *method='POST'*, *data=None*, *headers=None*, *response_check=None*, *extra_options=None*, *http_conn_id='http_default'*, *log_response=False*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Calls an endpoint on an HTTP system to execute an action

Parameters

- **http_conn_id** (*str*) – The connection to run the operator against
- **endpoint** (*str*) – The relative part of the full url. (templated)
- **method** (*str*) – The HTTP method to use, default = "POST"
- **data** (*For POST/PUT, depends on the content-type parameter, for GET a dictionary of key/value string pairs*) – The data to pass. POST-data in POST/PUT and params in the URL for a GET request. (templated)
- **headers** (*a dictionary of string key/value pairs*) – The HTTP headers to be added to the GET request
- **response_check** (*A lambda or defined function.*) – A check against the 'requests' response object. Returns True for 'pass' and False otherwise.

- **extra_options** (*A dictionary of options, where key is string and value depends on the option that's being modified.*) – Extra options for the 'requests' library, see the 'requests' documentation (options to modify timeout, ssl, etc.)
- **log_response** (*bool*) – Log the response (default: False)

template_fields = ['endpoint', 'data', 'headers']

template_ext = []

ui_color = #f4a460

execute (*self*, *context*)

airflow.operators.jdbc_operator

Module Contents

class airflow.operators.jdbc_operator.**JdbcOperator** (*sql*, *jdbc_conn_id='jdbc_default'*, *autocommit=False*, *parameters=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a database using jdbc driver.

Requires jaydebeapi.

Parameters

- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'*) – the sql code to be executed. (templated)
- **jdbc_conn_id** (*str*) – reference to a predefined database
- **autocommit** (*bool*) – if True, each command is automatically committed. (default value: False)
- **parameters** (*mapping or iterable*) – (optional) the parameters to render the SQL query with.

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #ededed

execute (*self*, *context*)

airflow.operators.latest_only_operator

Module Contents

class airflow.operators.latest_only_operator.**LatestOnlyOperator**

Bases: *airflow.models.BaseOperator*, *airflow.models.SkipMixin*

Allows a workflow to skip tasks that are not running during the most recent schedule interval.

If the task is run outside of the latest schedule interval, all directly downstream tasks will be skipped.

ui_color = #e9ffdb

execute (*self*, *context*)

airflow.operators.mssql_operator

Module Contents

class airflow.operators.mssql_operator.**MsSqlOperator** (*sql*, *mssql_conn_id='mssql_default'*, *parameters=None*, *autocommit=False*, *database=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a specific Microsoft SQL database

Parameters

- **sql** (*str or string pointing to a template file with .sql extension. (templated)*) – the sql code to be executed
- **mssql_conn_id** (*str*) – reference to a specific mssql database
- **parameters** (*mapping or iterable*) – (optional) the parameters to render the SQL query with.
- **autocommit** (*bool*) – if True, each command is automatically committed. (default value: False)
- **database** (*str*) – name of database which overwrite defined one in connection

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #ededed

execute (*self*, *context*)

airflow.operators.mssql_to_hive

Module Contents

class airflow.operators.mssql_to_hive.**MsSqlToHiveTransfer** (*sql*, *hive_table*, *create=True*, *recreate=False*, *partition=None*, *delimiter=chr(1)*, *mssql_conn_id='mssql_default'*, *hive_cli_conn_id='hive_cli_default'*, *tblproperties=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from Microsoft SQL Server to Hive. The operator runs your query against Microsoft SQL Server, stores the file locally before loading it into a Hive table. If the `create` or `recreate` arguments are set to `True`, a `CREATE TABLE` and `DROP TABLE` statements are generated. Hive data types are inferred from the cursor's metadata. Note that the table generated in Hive uses `STORED AS textfile` which isn't the most efficient serialization format. If a large amount of data is loaded and/or if the table gets queried considerably, you may want to use this operator only to stage the data into a temporary table before loading it into its final destination using a `HiveOperator`.

Parameters

- **`sql`** (*`str`*) – SQL query to execute against the Microsoft SQL Server database. (templated)
- **`hive_table`** (*`str`*) – target Hive table, use dot notation to target a specific database. (templated)
- **`create`** (*`bool`*) – whether to create the table if it doesn't exist
- **`recreate`** (*`bool`*) – whether to drop and recreate the table at every execution
- **`partition`** (*`dict`*) – target partition as a dict of partition columns and values. (templated)
- **`delimiter`** (*`str`*) – field delimiter in the file
- **`mssql_conn_id`** (*`str`*) – source Microsoft SQL Server connection
- **`hive_conn_id`** (*`str`*) – destination hive connection
- **`tblproperties`** (*`dict`*) – TBLPROPERTIES of the hive table being created

`template_fields = ['sql', 'partition', 'hive_table']`

`template_ext = ['.sql']`

`ui_color = #a0e08c`

`classmethod type_map`(*cls*, *mssql_type*)

`execute`(*self*, *context*)

`airflow.operators.mysql_operator`

Module Contents

`class` `airflow.operators.mysql_operator.`**`MySqlOperator`**(*sql*, *mysql_conn_id='mysql_default'*, *parameters=None*, *autocommit=False*, *database=None*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Executes sql code in a specific MySQL database

Parameters

- **`sql`** (*`str or list[str]`*) – the sql code to be executed. Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql' (templated)
- **`mysql_conn_id`** (*`str`*) – reference to a specific mysql database
- **`parameters`** (*`mapping or iterable`*) – (optional) the parameters to render the SQL query with.
- **`autocommit`** (*`bool`*) – if True, each command is automatically committed. (default value: False)
- **`database`** (*`str`*) – name of database which overwrite defined one in connection

`template_fields = ['sql']`

`template_ext = ['.sql']`

`ui_color = #ededed`

execute (*self*, *context*)

airflow.operators.mysql_to_hive

Module Contents

class airflow.operators.mysql_to_hive.**MySqlToHiveTransfer** (*sql*, *hive_table*, *create=True*, *recreate=False*, *partition=None*, *delimiter=chr(1)*, *mysql_conn_id='mysql_default'*, *hive_cli_conn_id='hive_cli_default'*, *tblproperties=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Moves data from MySql to Hive. The operator runs your query against MySQL, stores the file locally before loading it into a Hive table. If the `create` or `recreate` arguments are set to `True`, a `CREATE TABLE` and `DROP TABLE` statements are generated. Hive data types are inferred from the cursor's metadata. Note that the table generated in Hive uses `STORED AS textfile` which isn't the most efficient serialization format. If a large amount of data is loaded and/or if the table gets queried considerably, you may want to use this operator only to stage the data into a temporary table before loading it into its final destination using a `HiveOperator`.

Parameters

- **sql** (*str*) – SQL query to execute against the MySQL database. (templated)
- **hive_table** (*str*) – target Hive table, use dot notation to target a specific database. (templated)
- **create** (*bool*) – whether to create the table if it doesn't exist
- **recreate** (*bool*) – whether to drop and recreate the table at every execution
- **partition** (*dict*) – target partition as a dict of partition columns and values. (templated)
- **delimiter** (*str*) – field delimiter in the file
- **mysql_conn_id** (*str*) – source mysql connection
- **hive_conn_id** (*str*) – destination hive connection
- **tblproperties** (*dict*) – TBLPROPERTIES of the hive table being created

template_fields = ['sql', 'partition', 'hive_table']

template_ext = ['.sql']

ui_color = #a0e08c

classmethod type_map (*cls*, *mysql_type*)

execute (*self*, *context*)

airflow.operators.oracle_operator

Module Contents

class airflow.operators.oracle_operator.**OracleOperator**(*sql*, *oracle_conn_id='oracle_default'*, *parameters=None*, *autocommit=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a specific Oracle database

Parameters

- **sql** (*str or list[str]*) – the sql code to be executed. Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql' (templated)
- **oracle_conn_id** (*str*) – reference to a specific Oracle database
- **parameters** (*mapping or iterable*) – (optional) the parameters to render the SQL query with.
- **autocommit** (*bool*) – if True, each command is automatically committed. (default value: False)

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #ededed

execute(*self*, *context*)

airflow.operators.papermill_operator

Module Contents

class airflow.operators.papermill_operator.**NoteBook**

Bases: airflow.lineage.datasets.DataSet

type_name = jupyter_notebook

attributes = ['location', 'parameters']

class airflow.operators.papermill_operator.**PapermillOperator**(*input_nb:str*, *output_nb:str*, *parameters:dict*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes a jupyter notebook through papermill that is annotated with parameters

Parameters

- **input_nb** (*str*) – input notebook (can also be a NoteBook or a File inlet)
- **output_nb** (*str*) – output notebook (can also be a NoteBook or File outlet)
- **parameters** (*dict*) – the notebook parameters to set

execute(*self*, *context*)

airflow.operators.pig_operator

Module Contents

class airflow.operators.pig_operator.**PigOperator**(*pig*, *pig_cli_conn_id='pig_cli_default'*, *pigparams_jinja_translate=False*, *pig_opts=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes pig script.

Parameters

- **pig** (*str*) – the pig latin script to be executed. (templated)
- **pig_cli_conn_id** (*str*) – reference to the Hive database
- **pigparams_jinja_translate** (*bool*) – when True, pig params-type templating ${var} gets translated into jinja-type templating {{ var }}. Note that you may want to use this along with the `DAG(user_defined_macros=myargs)` parameter. View the DAG object documentation for more details.
- **pig_opts** (*str*) – pig options, such as: -x tez, -useHCatalog, …

template_fields = ['pig']

template_ext = ['.pig', '.piglatin']

ui_color = #f0e4ec

get_hook(*self*)

prepare_template(*self*)

execute(*self*, *context*)

on_kill(*self*)

airflow.operators.postgres_operator

Module Contents

class airflow.operators.postgres_operator.**PostgresOperator**(*sql*, *postgres_conn_id='postgres_default'*, *autocommit=False*, *parameters=None*, *database=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a specific Postgres database

Parameters

- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'*) – the sql code to be executed. (templated)
- **postgres_conn_id** (*str*) – reference to a specific postgres database
- **autocommit** (*bool*) – if True, each command is automatically committed. (default value: False)

- **parameters** (*mapping or iterable*) – (optional) the parameters to render the SQL query with.
- **database** (*str*) – name of database which overwrite defined one in connection

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #ededed

execute (*self*, *context*)

airflow.operators.presto_check_operator

Module Contents

class airflow.operators.presto_check_operator.**PrestoCheckOperator** (*sql*, *presto_conn_id='presto_default'*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.CheckOperator*

Performs checks against Presto. The `PrestoCheckOperator` expects a sql query that will return a single row. Each value on that first row is evaluated using python `bool` casting. If any of the values return `False` the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average.

This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alerts without stopping the progress of the DAG.

Parameters

- **sql** (*str*) – the sql to be executed
- **presto_conn_id** (*str*) – reference to the Presto database

get_db_hook (*self*)

class airflow.operators.presto_check_operator.**PrestoValueCheckOperator** (*sql*, *pass_value*, *tol-er-ance=None*, *presto_conn_id='presto_defau*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.ValueCheckOperator*

Performs a simple value check using sql code.

Parameters

- **sql** (*str*) – the sql to be executed
- **presto_conn_id** (*str*) – reference to the Presto database

get_db_hook(*self*)

class airflow.operators.presto_check_operator.**PrestoIntervalCheckOperator**(*table*, *metrics_thresholds*, *date_filter_column='ds'*, *days_back=-7*, *presto_conn_id='presto_c* **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.IntervalCheckOperator*

Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.

Parameters

- **table** (*str*) – the table name
- **days_back** (*int*) – number of days between ds and the ds we want to check against. Defaults to 7 days
- **metrics_threshold** (*dict*) – a dictionary of ratios indexed by metrics
- **presto_conn_id** (*str*) – reference to the Presto database

get_db_hook(*self*)

airflow.operators.presto_to_mysql

Module Contents

class airflow.operators.presto_to_mysql.**PrestoToMySqlTransfer**(*sql*, *mysql_table*, *presto_conn_id='presto_default'*, *mysql_conn_id='mysql_default'*, *mysql_preoperator=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from Presto to MySQL, note that for now the data is loaded into memory before being pushed to MySQL, so this operator should be used for smallish amount of data.

Parameters

- **sql** (*str*) – SQL query to execute against Presto. (templated)
- **mysql_table** (*str*) – target MySQL table, use dot notation to target a specific database. (templated)
- **mysql_conn_id** (*str*) – source mysql connection
- **presto_conn_id** (*str*) – source presto connection

- **mysql_preoperator** (*str*) – sql statement to run against mysql prior to import, typically use to truncate of delete in place of the data coming in, allowing the task to be idempotent (running the task twice won't double load data). (templated)

template_fields = ['sql', 'mysql_table', 'mysql_preoperator']

template_ext = ['.sql']

ui_color = #a0e08c

execute (*self*, *context*)

airflow.operators.python_operator

Module Contents

class airflow.operators.python_operator.**PythonOperator** (*python_callable:Callable*, *op_args:Optional[Iterable]=None*, *op_kwargs:Optional[Dict]=None*, *provide_context:bool=False*, *templates_dict:Optional[Dict]=None*, *templates_exts:Optional[Iterable[str]]=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes a Python callable

See also:

For more information on how to use this operator, take a look at the guide: *PythonOperator*

Parameters

- **python_callable** (*python callable*) – A reference to an object that is callable
- **op_kwargs** (*dict (templated)*) – a dictionary of keyword arguments that will get unpacked in your function
- **op_args** (*list (templated)*) – a list of positional arguments that will get unpacked when calling your callable
- **provide_context** (*bool*) – if set to true, Airflow will pass a set of keyword arguments that can be used in your function. This set of kwargs correspond exactly to what you can use in your jinja templates. For this to work, you need to define ***kwargs* in your function header.
- **templates_dict** (*dict[str]*) – a dictionary where the values are templates that will get templated by the Airflow engine sometime between `__init__` and `execute` takes place and are made available in your callable's context after the template has been applied. (templated)
- **templates_exts** (*list[str]*) – a list of file extensions to resolve while processing templated fields, for examples `['.sql', '.hql']`

template_fields = ['templates_dict', 'op_args', 'op_kwargs']

ui_color = #ffefeb

shallow_copy_attrs = ['python_callable', 'op_kwargs']

execute (*self*, *context*)

execute_callable(*self*)

class airflow.operators.python_operator.**BranchPythonOperator**

Bases: *airflow.operators.python_operator.PythonOperator*, *airflow.models.SkipMixin*

Allows a workflow to "branch" or follow a path following the execution of this task.

It derives the PythonOperator and expects a Python function that returns a single task_id or list of task_ids to follow. The task_id(s) returned should point to a task directly downstream from {self}. All other "branches" or directly downstream tasks are marked with a state of `skipped` so that these paths can't move forward. The `skipped` states are propagated downstream to allow for the DAG state to fill up and the DAG run's state to be inferred.

execute(*self*, *context*)

class airflow.operators.python_operator.**ShortCircuitOperator**

Bases: *airflow.operators.python_operator.PythonOperator*, *airflow.models.SkipMixin*

Allows a workflow to continue only if a condition is met. Otherwise, the workflow "short-circuits" and downstream tasks are skipped.

The ShortCircuitOperator is derived from the PythonOperator. It evaluates a condition and short-circuits the workflow if the condition is False. Any downstream tasks are marked with a state of "skipped". If the condition is True, downstream tasks proceed as normal.

The condition is determined by the result of *python_callable*.

execute(*self*, *context*)

class airflow.operators.python_operator.**PythonVirtualenvOperator**(*python_callable:Callable, requirements:Optional[Iterable[str]]=None, python_version:Optional[str]=None, use_dill:bool=False, system_site_packages:bool=True, op_args:Iterable=None, op_kwargs:Dict=None, provide_context:bool=False, string_args:Optional[Iterable[str]]=Nor templates_dict:Optional[Dict]=None, templates_exts:Optional[Iterable[str]]=Nor *args, **kwargs*)

Bases: *airflow.operators.python_operator.PythonOperator*

Allows one to run a function in a virtualenv that is created and destroyed automatically (with certain caveats).

The function must be defined using def, and not be part of a class. All imports must happen inside the function and no variables outside of the scope may be referenced. A global scope variable named virtualenv_string_args will be available (populated by string_args). In addition, one can pass stuff through op_args and op_kwargs, and one can use a return value. Note that if your virtualenv runs in a different Python major version than Airflow, you cannot use return values, op_args, or op_kwargs. You can use string_args though.

Parameters

- **python_callable** (*function*) – A python function with no references to outside variables, defined with def, which will be run in a virtualenv

- **requirements** (*list[str]*) – A list of requirements as specified in a pip install command
- **python_version** (*str*) – The Python version to run the virtualenv with. Note that both 2 and 2.7 are acceptable forms.
- **use_dill** (*bool*) – Whether to use dill to serialize the args and result (pickle is default). This allow more complex types but requires you to include dill in your requirements.
- **system_site_packages** (*bool*) – Whether to include system_site_packages in your virtualenv. See virtualenv documentation for more information.
- **op_args** – A list of positional arguments to pass to python_callable.
- **op_kwargs** (*dict*) – A dict of keyword arguments to pass to python_callable.
- **provide_context** (*bool*) – if set to true, Airflow will pass a set of keyword arguments that can be used in your function. This set of kwargs correspond exactly to what you can use in your jinja templates. For this to work, you need to define ***kwargs* in your function header.
- **string_args** (*list[str]*) – Strings that are present in the global var virtualenv_string_args, available to python_callable at runtime as a list[str]. Note that args are split by newline.
- **templates_dict** (*dict of str*) – a dictionary where the values are templates that will get templated by the Airflow engine sometime between `__init__` and `execute` takes place and are made available in your callable's context after the template has been applied
- **templates_exts** (*list[str]*) – a list of file extensions to resolve while processing templated fields, for examples `['.sql', '.hql']`

execute_callable(*self*)

_pass_op_args(*self*)

_execute_in_subprocess(*self*, *cmd*)

_write_string_args(*self*, *filename*)

_write_args(*self*, *input_filename*)

_read_result(*self*, *output_filename*)

_write_script(*self*, *script_filename*)

_generate_virtualenv_cmd(*self*, *tmp_dir*)

_generate_pip_install_cmd(*self*, *tmp_dir*)

static _generate_python_cmd(*tmp_dir*, *script_filename*, *input_filename*, *output_filename*, *string_args_filename*)

_generate_python_code(*self*)

airflow.operators.redshift_to_s3_operator

Module Contents

class airflow.operators.redshift_to_s3_operator.**RedshiftToS3Transfer**(*schema, table, s3_bucket, s3_key, redshift_conn_id='redshift_default', aws_conn_id='aws_default', verify=None, unload_options=tuple(), autocommit=False, include_header=False, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Executes an UNLOAD command to s3 as a CSV with headers

Parameters

- **schema** (*str*) – reference to a specific schema in redshift database
- **table** (*str*) – reference to a specific table in redshift database
- **s3_bucket** (*str*) – reference to a specific S3 bucket
- **s3_key** (*str*) – reference to a specific S3 key
- **redshift_conn_id** (*str*) – reference to a specific redshift database
- **aws_conn_id** (*str*) – reference to a specific S3 connection
- **verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **False: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.
- **unload_options** (*list*) – reference to a list of UNLOAD options

template_fields = []

template_ext = []

ui_color = #ededed

execute(*self, context*)

airflow.operators.s3_file_transform_operator

Module Contents

class airflow.operators.s3_file_transform_operator.**S3FileTransformOperator**(*source_s3_key*, *dest_s3_key*, *transform_script=None*, *select_expression=None*, *source_aws_conn_id='a*, *source_verify=None*, *dest_aws_conn_id='aws*, *dest_verify=None*, *replace=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Copies data from a source S3 location to a temporary location on the local filesystem. Runs a transformation on this file as specified by the transformation script and uploads the output to a destination S3 location.

The locations of the source and the destination files in the local filesystem is provided as an first and second arguments to the transformation script. The transformation script is expected to read the data from source, transform it and write the output to the local destination file. The operator then takes over control and uploads the local destination file to S3.

S3 Select is also available to filter the source contents. Users can omit the transformation script if S3 Select expression is specified.

Parameters

- **source_s3_key** (*str*) – The key to be retrieved from S3. (templated)
- **source_aws_conn_id** (*str*) – source s3 connection
- **source_verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **False: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.

  This is also applicable to `dest_verify`.
- **dest_s3_key** (*str*) – The key to be written from S3. (templated)
- **dest_aws_conn_id** (*str*) – destination s3 connection
- **replace** (*bool*) – Replace dest S3 key if it already exists
- **transform_script** (*str*) – location of the executable transformation script
- **select_expression** (*str*) – S3 Select expression

template_fields = ['source_s3_key', 'dest_s3_key']

template_ext = []

ui_color = #f9c915

execute (*self*, *context*)

airflow.operators.s3_to_hive_operator

Module Contents

class airflow.operators.s3_to_hive_operator.**S3ToHiveTransfer** (*s3_key*, *field_dict*, *hive_table*, *delimiter=',   ', create=True*, *recreate=False*, *partition=None*, *headers=False*, *check_headers=False*, *wildcard_match=False*, *aws_conn_id='aws_default'*, *verify=None*, *hive_cli_conn_id='hive_cli_default'*, *input_compressed=False*, *tblproperties=None*, *select_expression=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from S3 to Hive. The operator downloads a file from S3, stores the file locally before loading it into a Hive table. If the `create` or `recreate` arguments are set to `True`, a `CREATE TABLE` and `DROP TABLE` statements are generated. Hive data types are inferred from the cursor's metadata from.

Note that the table generated in Hive uses `STORED AS textfile` which isn't the most efficient serialization format. If a large amount of data is loaded and/or if the tables gets queried considerably, you may want to use this operator only to stage the data into a temporary table before loading it into its final destination using a `HiveOperator`.

Parameters

- **s3_key** (*str*) – The key to be retrieved from S3. (templated)
- **field_dict** (*dict*) – A dictionary of the fields name in the file as keys and their Hive types as values
- **hive_table** (*str*) – target Hive table, use dot notation to target a specific database. (templated)
- **create** (*bool*) – whether to create the table if it doesn't exist
- **recreate** (*bool*) – whether to drop and recreate the table at every execution
- **partition** (*dict*) – target partition as a dict of partition columns and values. (templated)
- **headers** (*bool*) – whether the file contains column names on the first line
- **check_headers** (*bool*) – whether the column names on the first line should be checked against the keys of field_dict
- **wildcard_match** (*bool*) – whether the s3_key should be interpreted as a Unix wildcard pattern
- **delimiter** (*str*) – field delimiter in the file

- **aws_conn_id** (*str*) – source s3 connection
- **verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **False: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.
- **hive_cli_conn_id** (*str*) – destination hive connection
- **input_compressed** (*bool*) – Boolean to determine if file decompression is required to process headers
- **tblproperties** (*dict*) – TBLPROPERTIES of the hive table being created
- **select_expression** (*str*) – S3 Select expression

template_fields = ['s3_key', 'partition', 'hive_table']

template_ext = []

ui_color = #a0e08c

execute(*self*, *context*)

_get_top_row_as_list(*self*, *file_name*)

_match_headers(*self*, *header_list*)

static _delete_top_row_and_compress(*input_file_name*, *output_file_ext*, *dest_dir*)

airflow.operators.s3_to_redshift_operator

Module Contents

class airflow.operators.s3_to_redshift_operator.**S3ToRedshiftTransfer**(*schema*, *table*, *s3_bucket*, *s3_key*, *redshift_conn_id='redshift_default'*, *aws_conn_id='aws_default'*, *verify=None*, *copy_options=tuple()*, *autocommit=False*, *parameters=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes an COPY command to load files from s3 to Redshift

Parameters

- **schema** (*str*) – reference to a specific schema in redshift database
- **table** (*str*) – reference to a specific table in redshift database
- **s3_bucket** (*str*) – reference to a specific S3 bucket
- **s3_key** (*str*) – reference to a specific S3 key
- **redshift_conn_id** (*str*) – reference to a specific redshift database
- **aws_conn_id** (*str*) – reference to a specific S3 connection
- **verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **False: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.
- **copy_options** (*list*) – reference to a list of COPY options

template_fields = []

template_ext = []

ui_color = #ededed

execute (*self*, *context*)

airflow.operators.slack_operator

Module Contents

class airflow.operators.slack_operator.**SlackAPIOperator** (*slack_conn_id=None*, *token=None*, *method=None*, *api_params=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Base Slack Operator The SlackAPIPostOperator is derived from this operator. In the future additional Slack API Operators will be derived from this class as well

Parameters

- **slack_conn_id** (*str*) – Slack connection ID which its password is Slack API token
- **token** (*str*) – Slack API token (https://api.slack.com/web)
- **method** (*str*) – The Slack API Method to Call (https://api.slack.com/methods)
- **api_params** (*dict*) – API Method call parameters (https://api.slack.com/methods)

construct_api_call_params (*self*)

Used by the execute function. Allows templating on the source fields of the api_call_params dict before construction

Override in child classes. Each SlackAPIOperator child class is responsible for having a construct_api_call_params function which sets self.api_call_params with a dict of API call parameters (https://api.slack.com/methods)

execute (*self*, ***kwargs*)

SlackAPIOperator calls will not fail even if the call is not unsuccessful. It should not prevent a DAG from completing in success

class airflow.operators.slack_operator.**SlackAPIPostOperator**(*channel='#general', username='Airflow', text='No message has been set.nHere is a cat video insteadnhttps://www.youtube.com/watch?v=J–aiyznGQ', icon_url='https://raw.githubusercontent.com/ap attachments=None, *args, **kwargs*)

Bases: `airflow.operators.slack_operator.SlackAPIOperator`

Posts messages to a slack channel

Parameters

- **channel** (*str*) – channel in which to post message on slack name (#general) or ID (C12318391). (templated)
- **username** (*str*) – Username that airflow will be posting to Slack as. (templated)
- **text** (*str*) – message to send to slack. (templated)
- **icon_url** (*str*) – url to icon used for this message
- **attachments** (*array of hashes*) – extra formatting details. (templated) - see https://api.slack.com/docs/attachments.

template_fields = ['username', 'text', 'attachments', 'channel']

ui_color = #FFBA40

construct_api_call_params(*self*)

airflow.operators.sqlite_operator

Module Contents

class airflow.operators.sqlite_operator.**SqliteOperator**(*sql, sqlite_conn_id='sqlite_default', parameters=None, *args, **kwargs*)

Bases: `airflow.models.BaseOperator`

Executes sql code in a specific Sqlite database

Parameters

- **sql** (*str or string pointing to a template file. File must have a '.sql' extensions.*) – the sql code to be executed. (templated)
- **sqlite_conn_id** (*str*) – reference to a specific sqlite database
- **parameters** (*mapping or iterable*) – (optional) the parameters to render the SQL query with.

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #cdaaed

execute(*self, context*)

`airflow.operators.subdag_operator`

Module Contents

class `airflow.operators.subdag_operator.`**`SubDagOperator`**(*subdag*, *executor=SequentialExecutor()*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

This runs a sub dag. By convention, a sub dag's dag_id should be prefixed by its parent and a dot. As in *parent.child*.

Parameters

- **`subdag`** (`airflow.models.DAG`) – the DAG object to run as a subdag of the current DAG.
- **`dag`** (`airflow.models.DAG`) – the parent DAG for the subdag.
- **`executor`** (`airflow.executors.base_executor.BaseExecutor`) – the executor for this subdag. Default to use SequentialExecutor. Please find AIRFLOW-74 for more details.

`ui_color = #555`

`ui_fgcolor = #fff`

`execute`(*self*, *context*)

Package Contents

`airflow.operators.`**`_integrate_plugins`**()
`Integrate plugins to the context`

`airflow.sensors`

Submodules

`airflow.sensors.base_sensor_operator`

Module Contents

class `airflow.sensors.base_sensor_operator.`**`BaseSensorOperator`**(*poke_interval=60*, *timeout=60 * 60 * 24 * 7*, *soft_fail=False*, *mode='poke'*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*, *`airflow.models.SkipMixin`*

Sensor operators are derived from this class and inherit these attributes.

Sensor operators keep executing at a time interval and succeed when a criteria is met and fail if and when they time out.

Parameters

- **`soft_fail`** (*`bool`*) – Set to true to mark the task as SKIPPED on failure
- **`poke_interval`** (*`int`*) – Time in seconds that the job should wait in between each tries

- **timeout** (*int*) – Time, in seconds before the task times out and fails.
- **mode** (*str*) – How the sensor operates. Options are: `{ poke | reschedule }`, default is `poke`. When set to `poke` the sensor is taking up a worker slot for its whole execution time and sleeps between pokes. Use this mode if the expected runtime of the sensor is short or if a short poke interval is required. When set to `reschedule` the sensor task frees the worker slot when the criteria is not yet met and it's rescheduled at a later time. Use this mode if the expected time until the criteria is met is. The poke interval should be more than one minute to prevent too much load on the scheduler.

ui_color = #e6f1f2

valid_modes = ['poke', 'reschedule']

reschedule

deps

Adds one additional dependency for all sensor operators that checks if a sensor task instance can be rescheduled.

_validate_input_values(*self*)

poke(*self*, *context*)

Function that the sensors defined while deriving this class should override.

execute(*self*, *context*)

_do_skip_downstream_tasks(*self*, *context*)

airflow.sensors.external_task_sensor

Module Contents

class airflow.sensors.external_task_sensor.**ExternalTaskSensor**(*external_dag_id*, *external_task_id*, *allowed_states=None*, *execution_delta=None*, *execution_date_fn=None*, *check_existence=False*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a different DAG or a task in a different DAG to complete for a specific execution_date

Parameters

- **external_dag_id** (*str*) – The dag_id that contains the task you want to wait for
- **external_task_id** (*str*) – The task_id that contains the task you want to wait for. If `None` the sensor waits for the DAG
- **allowed_states** (*list*) – list of allowed states, default is `['success']`
- **execution_delta** (*datetime.timedelta*) – time difference with the previous execution to look at, the default is the same execution_date as the current task or DAG. For yesterday, use [positive!] datetime.timedelta(days=1). Either execution_delta or execution_date_fn can be passed to ExternalTaskSensor, but not both.

- **execution_date_fn** (*callable*) – function that receives the current execution date and returns the desired execution dates to query. Either execution_delta or execution_date_fn can be passed to ExternalTaskSensor, but not both.
- **check_existence** (*bool*) – Set to *True* to check if the external task exists (when external_task_id is not None) or check if the DAG to wait for exists (when external_task_id is None), and immediately cease waiting if the external task or DAG does not exist (default value: False).

template_fields = ['external_dag_id', 'external_task_id']

ui_color = #19647e

poke (*self*, *context*, *session=None*)

airflow.sensors.hdfs_sensor

Module Contents

class airflow.sensors.hdfs_sensor.**HdfsSensor** (*filepath*, *hdfs_conn_id='hdfs_default'*, *ignored_ext=None*, *ignore_copying=True*, *file_size=None*, *hook=HDFSHook*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a file or folder to land in HDFS

template_fields = ['filepath']

ui_color

static filter_for_filesize (*result*, *size=None*)

Will test the filepath result and test if its size is at least self.filesize

Parameters

- **result** – a list of dicts returned by Snakebite ls
- **size** – the file size in MB a file should be at least to trigger True

Returns (bool) depending on the matching criteria

static filter_for_ignored_ext (*result*, *ignored_ext*, *ignore_copying*)

Will filter if instructed to do so the result to remove matching criteria

Parameters

- **result** (*list[dict]*) – list of dicts returned by Snakebite ls
- **ignored_ext** (*list*) – list of ignored extensions
- **ignore_copying** (*bool*) – shall we ignore ?

Returns list of dicts which were not removed

Return type list[dict]

poke (*self*, *context*)

airflow.sensors.hive_partition_sensor

Module Contents

class airflow.sensors.hive_partition_sensor.**HivePartitionSensor**(*table*, *partition="ds='{{ ds }}'"*, *metastore_conn_id='metastore_default'*, *schema='default'*, *poke_interval=60 * 3*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a partition to show up in Hive.

Note: Because `partition` supports general logical operators, it can be inefficient. Consider using NamedHivePartitionSensor instead if you don't need the full flexibility of HivePartitionSensor.

Parameters

- **table** (*str*) – The name of the table to wait for, supports the dot notation (my_database.my_table)
- **partition** (*str*) – The partition clause to wait for. This is passed as is to the metastore Thrift client `get_partitions_by_filter` method, and apparently supports SQL like notation as in `ds='2015-01-01' AND type='value'` and comparison operators as in `"ds>=2015-01-01"`
- **metastore_conn_id** (*str*) – reference to the metastore thrift service connection id

template_fields = ['schema', 'table', 'partition']

ui_color = #C5CAE9

poke(*self*, *context*)

airflow.sensors.http_sensor

Module Contents

class airflow.sensors.http_sensor.**HttpSensor**(*endpoint*, *http_conn_id='http_default'*, *method='GET'*, *request_params=None*, *headers=None*, *response_check=None*, *provide_context=False*, *extra_options=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Executes a HTTP GET statement and returns False on failure caused by 404 Not Found or *response_check* returning False.

HTTP Error codes other than 404 (like 403) or Connection Refused Error would fail the sensor itself directly (no more poking).

The response check can access the template context by passing `provide_context=True` to the operator:

```
def response_check(response, **context):
    # Can look at context['ti'] etc.
    return True
```

(continues on next page)

(continued from previous page)

```
HttpSensor(task_id='my_http_sensor', ..., provide_context=True, response_
↪check=response_check)
```

Parameters

- **http_conn_id** (*str*) – The connection to run the sensor against
- **method** (*str*) – The HTTP request method to use
- **endpoint** (*str*) – The relative part of the full url
- **request_params** (*a dictionary of string key/value pairs*) – The parameters to be added to the GET url
- **headers** (*a dictionary of string key/value pairs*) – The HTTP headers to be added to the GET request
- **provide_context** (*bool*) – if set to true, Airflow will pass a set of keyword arguments that can be used in your function. This set of kwargs correspond exactly to what you can use in your jinja templates. For this to work, you need to define context in your function header.
- **response_check** (*A lambda or defined function.*) – A check against the 'requests' response object. Returns True for 'pass' and False otherwise.
- **extra_options** (*A dictionary of options, where key is string and value depends on the option that's being modified.*) – Extra options for the 'requests' library, see the 'requests' documentation (options to modify timeout, ssl, etc.)

template_fields = ['endpoint', 'request_params']

poke (*self*, *context*)

airflow.sensors.metastore_partition_sensor

Module Contents

class airflow.sensors.metastore_partition_sensor.**MetastorePartitionSensor** (*table*, *partition_name*, *schema='default'*, *mysql_conn_id='metastor*
**args*, ***kwargs*)

Bases: *airflow.sensors.sql_sensor.SqlSensor*

An alternative to the HivePartitionSensor that talk directly to the MySQL db. This was created as a result of observing sub optimal queries generated by the Metastore thrift service when hitting subpartitioned tables. The Thrift service's queries were written in a way that wouldn't leverage the indexes.

Parameters

- **schema** (*str*) – the schema
- **table** (*str*) – the table

- **partition_name** (*str*) – the partition name, as defined in the PARTITIONS table of the Metastore. Order of the fields does matter. Examples: `ds=2016-01-01` or `ds=2016-01-01/sub=foo` for a sub partitioned table
- **mysql_conn_id** (*str*) – a reference to the MySQL conn_id for the metastore

template_fields = ['partition_name', 'table', 'schema']

ui_color = #8da7be

poke (*self*, *context*)

airflow.sensors.named_hive_partition_sensor

Module Contents

class airflow.sensors.named_hive_partition_sensor.**NamedHivePartitionSensor** (*partition_names*, *metastore_conn_id='metastor*, *poke_interval=60 * 3*, *hook=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a set of partitions to show up in Hive.

Parameters

- **partition_names** (*list[str]*) – List of fully qualified names of the partitions to wait for. A fully qualified name is of the form `schema.table/pk1=pv1/pk2=pv2`, for example, default.users/ds=2016-01-01. This is passed as is to the metastore Thrift client `get_partitions_by_name` method. Note that you cannot use logical or comparison operators as in HivePartitionSensor.
- **metastore_conn_id** (*str*) – reference to the metastore thrift service connection id

template_fields = ['partition_names']

ui_color = #8d99ae

static parse_partition_name (*partition*)

poke_partition (*self*, *partition*)

poke (*self*, *context*)

airflow.sensors.s3_key_sensor

Module Contents

class airflow.sensors.s3_key_sensor.**S3KeySensor** (*bucket_key*, *bucket_name=None*, *wildcard_match=False*, *aws_conn_id='aws_default'*, *verify=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a key (a file-like instance on S3) to be present in a S3 bucket. S3 being a key/value it does not support folders. The path is just a key a resource.

Parameters

- **`bucket_key`** (*`str`*) – The key being waited on. Supports full s3:// style url or relative path from root level. When it's specified as a full s3:// url, please leave bucket_name as *None*.
- **`bucket_name`** (*`str`*) – Name of the S3 bucket. Only needed when `bucket_key` is not provided as a full s3:// url.
- **`wildcard_match`** (*`bool`*) – whether the bucket_key should be interpreted as a Unix wildcard pattern
- **`aws_conn_id`** (*`str`*) – a reference to the s3 connection
- **`verify`** (*`bool or str`*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **`False`: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **`path/to/cert/bundle.pem`: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.

`template_fields = ['bucket_key', 'bucket_name']`

`poke` (*self*, *context*)

`airflow.sensors.s3_prefix_sensor`

Module Contents

class `airflow.sensors.s3_prefix_sensor.`**`S3PrefixSensor`**(*bucket_name*, *prefix*, *delimiter='/'*, *aws_conn_id='aws_default'*, *verify=None*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

Waits for a prefix to exist. A prefix is the first part of a key, thus enabling checking of constructs similar to glob airfl* or SQL LIKE 'airfl%'. There is the possibility to precise a delimiter to indicate the hierarchy or keys, meaning that the match will stop at that delimiter. Current code accepts sane delimiters, i.e. characters that are NOT special characters in the Python regex engine.

Parameters

- **`bucket_name`** (*`str`*) – Name of the S3 bucket
- **`prefix`** (*`str`*) – The prefix being waited on. Relative path from bucket root level.
- **`delimiter`** (*`str`*) – The delimiter intended to show hierarchy. Defaults to '/'.
- **`aws_conn_id`** (*`str`*) – a reference to the s3 connection
- **`verify`** (*`bool or str`*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **`False`: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.

  – **`path/to/cert/bundle.pem`: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.

`template_fields = ['prefix', 'bucket_name']`

`poke`(*self*, *context*)

`airflow.sensors.sql_sensor`

Module Contents

class `airflow.sensors.sql_sensor.`**`SqlSensor`**(*conn_id*, *sql*, *parameters=None*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

Runs a sql statement until a criteria is met. It will keep trying while sql returns no row, or if the first cell in (0, '0', '').

Parameters

- **`conn_id`** (*`str`*) – The connection to run the sensor against
- **`sql`** (*`str`*) – The sql to run. To pass, it needs to return at least one cell that contains a non-zero / empty string value.
- **`parameters`** (*`mapping or iterable`*) – The parameters to render the SQL query with (optional).

`template_fields :Iterable[str] = ['sql']`

`template_ext :Iterable[str] = ['.hql', '.sql']`

`ui_color = #7c7287`

`poke`(*self*, *context*)

`airflow.sensors.time_delta_sensor`

Module Contents

class `airflow.sensors.time_delta_sensor.`**`TimeDeltaSensor`**(*delta*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

Waits for a timedelta after the task's execution_date + schedule_interval. In Airflow, the daily task stamped with `execution_date` 2016-01-01 can only start running on 2016-01-02. The timedelta here represents the time after the execution period has closed.

Parameters **`delta`** (*`datetime.timedelta`*) – time length to wait after execution_date before succeeding

`poke`(*self*, *context*)

`airflow.sensors.time_sensor`

Module Contents

class `airflow.sensors.time_sensor.`**`TimeSensor`**(*target_time*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

Waits until the specified time of the day.

Parameters **target_time** (*datetime.time*) – time after which the job succeeds

poke (*self*, *context*)

airflow.sensors.web_hdfs_sensor

Module Contents

class airflow.sensors.web_hdfs_sensor.**WebHdfsSensor** (*filepath*, *webhdfs_conn_id='webhdfs_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a file or folder to land in HDFS

template_fields = ['filepath']

poke (*self*, *context*)

Package Contents

airflow.sensors.**_integrate_plugins** ()
Integrate plugins to the context

airflow.contrib.operators

Submodules

airflow.contrib.operators.adls_list_operator

Module Contents

class airflow.contrib.operators.adls_list_operator.**AzureDataLakeStorageListOperator** (*path*, *azure_dat*
**args*,
***kwargs*

Bases: *airflow.models.BaseOperator*

List all files from the specified path

This operator returns a python list with the names of files which can be used by *xcom* in the downstream tasks.

Parameters

- **path** (*str*) – The Azure Data Lake path to find the objects. Supports glob strings (templated)
- **azure_data_lake_conn_id** (*str*) – The connection ID to use when connecting to Azure Data Lake Storage.

Example: The following Operator would list all the Parquet files from `folder/output/` folder in the specified ADLS account

```
adls_files = AzureDataLakeStorageListOperator(
    task_id='adls_files',
    path='folder/output/*.parquet',
    azure_data_lake_conn_id='azure_data_lake_default'
)
```

template_fields :Iterable[str] = ['path']

ui_color = #901dd2

execute (*self*, *context*)

airflow.contrib.operators.adls_to_gcs

Module Contents

class airflow.contrib.operators.adls_to_gcs.**AdlsToGoogleCloudStorageOperator** (*src_adls*, *dest_gcs*, *azure_data_lake_co*
google_cloud_storag
del-
e-
gate_to=None,
re-
place=False,
**args*,
***kwargs*)

Bases: *airflow.contrib.operators.adls_list_operator.AzureDataLakeStorageListOperator*

Synchronizes an Azure Data Lake Storage path with a GCS bucket

Parameters

- **src_adls** (*str*) – The Azure Data Lake path to find the objects (templated)
- **dest_gcs** (*str*) – The Google Cloud Storage bucket and prefix to store the objects. (templated)
- **replace** (*bool*) – If true, replaces same-named files in GCS
- **azure_data_lake_conn_id** (*str*) – The connection ID to use when connecting to Azure Data Lake Storage.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google Cloud Storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Examples: The following Operator would copy a single file named `hello/world.avro` from ADLS to the GCS bucket `mybucket`. Its full resulting gcs path will be `gs://mybucket/hello/world.avro`

```
copy_single_file = AdlsToGoogleCloudStorageOperator(
    task_id='copy_single_file',
    src_adls='hello/world.avro',
    dest_gcs='gs://mybucket',
    replace=False,
    azure_data_lake_conn_id='azure_data_lake_default',
```

(continues on next page)

(continued from previous page)

```
    google_cloud_storage_conn_id='google_cloud_default'
)
```

The following Operator would copy all parquet files from ADLS to the GCS bucket `mybucket`.

```
   copy_all_files = AdlsToGoogleCloudStorageOperator(
       task_id='copy_all_files',
       src_adls='*.parquet',
       dest_gcs='gs://mybucket',
       replace=False,
       azure_data_lake_conn_id='azure_data_lake_default',
       google_cloud_storage_conn_id='google_cloud_default'
   )

The following Operator would copy all parquet files from ADLS
path ``/hello/world``to the GCS bucket ``mybucket``. ::

   copy_world_files = AdlsToGoogleCloudStorageOperator(
       task_id='copy_world_files',
       src_adls='hello/world/*.parquet',
       dest_gcs='gs://mybucket',
       replace=False,
       azure_data_lake_conn_id='azure_data_lake_default',
       google_cloud_storage_conn_id='google_cloud_default'
   )
```

`template_fields = ['src_adls', 'dest_gcs']`

`ui_color = #f0eee4`

`execute`(*self*, *context*)

`airflow.contrib.operators.aws_athena_operator`

Module Contents

class `airflow.contrib.operators.aws_athena_operator.`**`AWSAthenaOperator`**(*query*, *database*, *output_location*, *aws_conn_id='aws_default'*, *client_request_token=None*, *query_execution_context=None*, *result_configuration=None*, *sleep_time=30*, *max_tries=None*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

An operator that submit presto query to athena.

Parameters

- **`query`** (*`str`*) – Presto to be run on athena. (templated)

- **database** (*str*) – Database to select. (templated)
- **output_location** (*str*) – s3 path to write the query results into. (templated)
- **aws_conn_id** (*str*) – aws connection to use
- **sleep_time** (*int*) – Time to wait between two consecutive call to check query status on athena
- **max_tries** – Number of times to poll for query state before function exits

ui_color = #44b5e2

template_fields = ['query', 'database', 'output_location']

template_ext = ['.sql']

get_hook (*self*)

execute (*self*, *context*)
Run Presto Query on Athena

on_kill (*self*)
Cancel the submitted athena query

airflow.contrib.operators.aws_sqs_publish_operator

Module Contents

class airflow.contrib.operators.aws_sqs_publish_operator.**SQSPublishOperator** (*sqs_queue*, *mes-sage_content*, *mes-sage_attributes=None*, *de-lay_seconds=0*, *aws_conn_id='aws_d*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Publish message to a SQS queue.

Parameters

- **sqs_queue** (*str*) – The SQS queue url (templated)
- **message_content** (*str*) – The message content (templated)
- **message_attributes** (*dict*) – additional attributes for the message (default: None) For details of the attributes parameter see `botocore.client.SQS.send_message()`
- **delay_seconds** (*int*) – message delay (templated) (default: 1 second)
- **aws_conn_id** (*str*) – AWS connection id (default: aws_default)

template_fields = ['sqs_queue', 'message_content', 'delay_seconds']

ui_color = #6ad3fa

execute (*self*, *context*)
Publish the message to SQS queue

Parameters context (*dict*) – the context object

Returns dict with information about the message sent For details of the returned dict see `botocore.client.SQS.send_message()`

Return type dict

`airflow.contrib.operators.awsbatch_operator`

Module Contents

class `airflow.contrib.operators.awsbatch_operator.`**`AWSBatchOperator`**(*job_name*, *job_definition*, *job_queue*, *overrides*, *max_retries=4200*, *aws_conn_id=None*, *region_name=None*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Execute a job on AWS Batch Service

Parameters

- **`job_name`** (*`str`*) – the name for the job that will run on AWS Batch (templated)
- **`job_definition`** (*`str`*) – the job definition name on AWS Batch
- **`job_queue`** (*`str`*) – the queue name on AWS Batch
- **`overrides`** (*`dict`*) – the same parameter that boto3 will receive on containerOverrides (templated): http://boto3.readthedocs.io/en/latest/reference/services/batch.html#submit_job
- **`max_retries`** (*`int`*) – exponential backoff retries while waiter is not merged, 4200 = 48 hours
- **`aws_conn_id`** (*`str`*) – connection id of AWS credentials / region name. If None, credential boto3 strategy will be used (http://boto3.readthedocs.io/en/latest/guide/configuration.html).
- **`region_name`** (*`str`*) – region name to use in AWS Hook. Override the region_name in connection (if provided)

`ui_color = #c3dae0`

`client`

`arn`

`template_fields = ['job_name', 'overrides']`

`execute`(*self*, *context*)

`_wait_for_task_ended`(*self*)

Try to use a waiter from the below pull request

- https://github.com/boto/botocore/pull/1307

If the waiter is not available apply a exponential backoff

- docs.aws.amazon.com/general/latest/gr/api-retries.html

`_check_success_task`(*self*)

`get_hook`(*self*)

`on_kill`(*self*)

`airflow.contrib.operators.azure_container_instances_operator`

Module Contents

`airflow.contrib.operators.azure_container_instances_operator.`**`Volume`**

`airflow.contrib.operators.azure_container_instances_operator.`**`DEFAULT_ENVIRONMENT_VARIABLES`**

`airflow.contrib.operators.azure_container_instances_operator.`**`DEFAULT_VOLUMES :Sequence[Volu`**

`airflow.contrib.operators.azure_container_instances_operator.`**`DEFAULT_MEMORY_IN_GB = 2.0`**

`airflow.contrib.operators.azure_container_instances_operator.`**`DEFAULT_CPU = 1.0`**

`class` `airflow.contrib.operators.azure_container_instances_operator.`**`AzureContainerInstances`**

Bases: *`airflow.models.BaseOperator`*

Start a container on Azure Container Instances

Parameters

- **`ci_conn_id`** (*`str`*) – connection id of a service principal which will be used to start the container instance
- **`registry_conn_id`** (*`str`*) – connection id of a user which can login to a private docker registry. If None, we assume a public registry
- **`resource_group`** (*`str`*) – name of the resource group wherein this container instance should be started
- **`name`** (*`str`*) – name of this container instance. Please note this name has to be unique in order to run containers in parallel.
- **`image`** (*`str`*) – the docker image to be used
- **`region`** (*`str`*) – the region wherein this container instance should be started

- **environment_variables** (*dict*) – key,value pairs containing environment variables which will be passed to the running container
- **volumes** (*list[<conn_id, account_name, share_name, mount_path, read_only>]*) – list of volumes to be mounted to the container. Currently only Azure Fileshares are supported.
- **memory_in_gb** (*double*) – the amount of memory to allocate to this container
- **cpu** (*double*) – the number of cpus to allocate to this container
- **command** (*str*) – the command to run inside the container

Example

```
>>>  a = AzureContainerInstancesOperator(
            'azure_service_principal',
            'azure_registry_user',
            'my-resource-group',
            'my-container-name-{{ ds }}',
            'myprivateregistry.azurecr.io/my_container:latest',
            'westeurope',
            {'EXECUTION_DATE': '{{ ds }}'},
            [('azure_wasb_conn_id',
              'my_storage_container',
              'my_fileshare',
              '/input-data',
              True),],
            memory_in_gb=14.0,
            cpu=4.0,
            command='python /app/myfile.py',
            task_id='start_container'
        )
```

template_fields = ['name', 'environment_variables']

execute (*self*, *context*)

_monitor_logging (*self*, *ci_hook*, *resource_group*, *name*)

_log_last (*self*, *logs*, *last_line_logged*)

airflow.contrib.operators.azure_cosmos_operator

Module Contents

class airflow.contrib.operators.azure_cosmos_operator.**AzureCosmosInsertDocumentOperator**(*data col- lec- tion doc u- men azu *ar **k*

Bases: *airflow.models.BaseOperator*

Inserts a new document into the specified Cosmos database and collection It will create both the database and collection if they do not already exist

Parameters

- **database_name** (*str*) – The name of the database. (templated)
- **collection_name** (*str*) – The name of the collection. (templated)
- **document** (*dict*) – The document to insert
- **azure_cosmos_conn_id** (*str*) – reference to a CosmosDB connection.

template_fields = ['database_name', 'collection_name']

ui_color = #e4f0e8

execute (*self*, *context*)

airflow.contrib.operators.bigquery_check_operator

This module contains Google BigQuery check operator.

Module Contents

class airflow.contrib.operators.bigquery_check_operator.**BigQueryCheckOperator** (*sql*, *big-query_conn_id='go*, *use_legacy_sql=Tr*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.CheckOperator*

Performs checks against BigQuery. The `BigQueryCheckOperator` expects a sql query that will return a single row. Each value on that first row is evaluated using python `bool` casting. If any of the values return `False` the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average.

This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alerts without stopping the progress of the DAG.

Parameters

- **sql** (*str*) – the sql to be executed
- **bigquery_conn_id** (*str*) – reference to the BigQuery database
- **use_legacy_sql** (*bool*) – Whether to use legacy SQL (true) or standard SQL (false).

template_fields = ['sql']

template_ext = ['.sql']

get_db_hook(*self*)

class airflow.contrib.operators.bigquery_check_operator.**BigQueryValueCheckOperator**(*sql, pass_value, tol- er- ance=None big- query_conn use_legacy_ *args, **kwargs*)

Bases: *airflow.operators.check_operator.ValueCheckOperator*

Performs a simple value check using sql code.

Parameters

- **sql** (*str*) – the sql to be executed
- **use_legacy_sql** (*bool*) – Whether to use legacy SQL (true) or standard SQL (false).

template_fields = ['sql']

template_ext = ['.sql']

get_db_hook(*self*)

class airflow.contrib.operators.bigquery_check_operator.**BigQueryIntervalCheckOperator**(*table, met- rics_th date_f days_b 7, big- query_ use_leg *args, **kwa*

Bases: *airflow.operators.check_operator.IntervalCheckOperator*

Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.

This method constructs a query like so

```
SELECT {metrics_threshold_dict_key} FROM {table}
WHERE {date_filter_column}=<date>
```

Parameters

- **table** (*str*) – the table name
- **days_back** (*int*) – number of days between ds and the ds we want to check against. Defaults to 7 days
- **metrics_threshold** (*dict*) – a dictionary of ratios indexed by metrics, for example 'COUNT(*)': 1.5 would require a 50 percent or less difference between the current day, and the prior days_back.
- **use_legacy_sql** (*bool*) – Whether to use legacy SQL (true) or standard SQL (false).

template_fields = ['table']

get_db_hook (*self*)

airflow.contrib.operators.bigquery_get_data

This module contains a Google BigQuery data operator.

Module Contents

class airflow.contrib.operators.bigquery_get_data.**BigQueryGetDataOperator** (*dataset_id, table_id, max_results='100', selected_fields=None, bigquery_conn_id='google_c*
*delegate_to=None, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Fetches the data from a BigQuery table (alternatively fetch data for selected columns) and returns data in a python list. The number of elements in the returned list will be equal to the number of rows fetched. Each element in the list will again be a list where element would represent the columns values for that row.

Example Result: `[['Tony', '10'], ['Mike', '20'], ['Steve', '15']]`

Note: If you pass fields to `selected_fields` which are in different order than the order of columns already in BQ table, the data will still be in the order of BQ table. For example if the BQ table has 3 columns as `[A,B,C]` and you pass 'B,A' in the `selected_fields` the data would still be of the form `'A,B'`.

Example:

```
get_data = BigQueryGetDataOperator(
    task_id='get_data_from_bq',
    dataset_id='test_dataset',
    table_id='Transaction_partitions',
    max_results='100',
    selected_fields='DATE',
    bigquery_conn_id='airflow-service-account'
)
```

Parameters

- **dataset_id** (*str*) – The dataset ID of the requested table. (templated)
- **table_id** (*str*) – The table ID of the requested table. (templated)
- **max_results** (*str*) – The maximum number of records (rows) to be fetched from the table. (templated)

- **selected_fields** (*str*) – List of fields to return (comma-separated). If unspecified, all fields are returned.
- **bigquery_conn_id** (*str*) – reference to a specific BigQuery hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['dataset_id', 'table_id', 'max_results']

ui_color = #e4f0e8

execute (*self*, *context*)

airflow.contrib.operators.bigquery_operator

This module contains Google BigQuery operators.

Module Contents

class airflow.contrib.operators.bigquery_operator.**BigQueryConsoleLink**

Bases: *airflow.models.baseoperator.BaseOperatorLink*

Helper class for constructing BigQuery link.

name = BigQuery Console

get_link (*self*, *operator*, *dttm*)

class airflow.contrib.operators.bigquery_operator.**BigQueryOperator**(*sql, destination_dataset_table=None, write_disposition='WRITE_EMPTY', allow_large_results=False, flatten_results=None, bigquery_conn_id='google_cloud_defa, delegate_to=None, udf_config=None, use_legacy_sql=True, maximum_billing_tier=None, maximum_bytes_billed=None, create_disposition='CREATE_IF_NEEL, schema_update_options=(), query_params=None, labels=None, priority='INTERACTIVE', time_partitioning=None, api_resource_configs=None, cluster_fields=None, location=None, *args, **kwargs*)

Bases: `airflow.models.baseoperator.BaseOperator`

Executes BigQuery SQL queries in a specific BigQuery database

Parameters

- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'.*) – the sql code to be executed (templated)
- **destination_dataset_table** (*str*) – A dotted `(<project>.|<project>:)<dataset>.<table>` that, if set, will store the results of the query. (templated)
- **write_disposition** (*str*) – Specifies the action that occurs if the destination table already exists. (default: 'WRITE_EMPTY')
- **create_disposition** (*str*) – Specifies whether the job is allowed to create new tables. (default: 'CREATE_IF_NEEDED')
- **allow_large_results** (*bool*) – Whether to allow large results.
- **flatten_results** (*bool*) – If true and query uses legacy SQL dialect, flattens all nested and repeated fields in the query results. `allow_large_results` must be `true` if this is

set to `false`. For standard SQL queries, this flag is ignored and results are never flattened.

- **bigquery_conn_id** (*str*) – reference to a specific BigQuery hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **udf_config** (*list*) – The User Defined Function configuration for the query. See https://cloud.google.com/bigquery/user-defined-functions for details.
- **use_legacy_sql** (*bool*) – Whether to use legacy SQL (true) or standard SQL (false).
- **maximum_billing_tier** (*int*) – Positive integer that serves as a multiplier of the basic price. Defaults to None, in which case it uses the value set in the project.
- **maximum_bytes_billed** (*float*) – Limits the bytes billed for this job. Queries that will have bytes billed beyond this limit will fail (without incurring a charge). If unspecified, this will be set to your project default.
- **api_resource_configs** (*dict*) – a dictionary that contain params 'configuration' applied for Google BigQuery Jobs API: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs for example, {'query': {'useQueryCache': False}}. You could use it if you need to provide some params that are not supported by BigQueryOperator like args.
- **schema_update_options** (*tuple*) – Allows the schema of the destination table to be updated as a side effect of the load job.
- **query_params** (*list*) – a list of dictionary containing query parameter types and values, passed to BigQuery. The structure of dictionary should look like 'queryParameters' in Google BigQuery Jobs API: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs. For example, [{ 'name': 'corpus', 'parameterType': { 'type': 'STRING' }, 'parameterValue': { 'value': 'romeoandjuliet' } }].
- **labels** (*dict*) – a dictionary containing labels for the job/query, passed to BigQuery
- **priority** (*str*) – Specifies a priority for the query. Possible values include INTERACTIVE and BATCH. The default value is INTERACTIVE.
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications.
- **cluster_fields** (*list[str]*) – Request that the result of this query be stored sorted by one or more columns. This is only available in conjunction with time_partitioning. The order of columns given determines the sort order.
- **location** (*str*) – The geographic location of the job. Required except for US and EU. See details at https://cloud.google.com/bigquery/docs/locations#specifying_your_location

template_fields = ['sql', 'destination_dataset_table', 'labels']

template_ext = ['.sql']

ui_color = #e4f0e8

operator_extra_links

execute (*self*, *context*)

on_kill (*self*)

class airflow.contrib.operators.bigquery_operator.**BigQueryCreateEmptyTableOperator**(*dataset_id, table_id, project_id=schema_fie gcs_schema time_partiti bigquery_conn google_clou delegate_to=No labels=None, *args, **kwargs*)

Bases: *airflow.models.baseoperator.BaseOperator*

Creates a new, empty table in the specified BigQuery dataset, optionally with schema.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it. You can also create a table without schema.

Parameters

- **project_id** (*str*) – The project to create the table into. (templated)
- **dataset_id** (*str*) – The dataset to create the table into. (templated)
- **table_id** (*str*) – The Name of the table to be created. (templated)
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

  Example:

```
schema_fields=[{"name": "emp_name", "type": "STRING", "mode":
↪"REQUIRED"},
                {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}]
```

- **gcs_schema_object** (*str*) – Full path to the JSON file containing schema (templated). For example: `gs://test-bucket/dir1/dir2/employee_schema.json`
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications.

  See also:

  https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#timePartitioning
- **bigquery_conn_id** (*str*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*str*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **labels** (*dict*) – a dictionary containing labels for the table, passed to BigQuery

  Example (with schema JSON in GCS):

```
CreateTable = BigQueryCreateEmptyTableOperator(
    task_id='BigQueryCreateEmptyTableOperator_task',
    dataset_id='ODS',
    table_id='Employees',
    project_id='internal-gcp-project',
    gcs_schema_object='gs://schema-bucket/employee_schema.json',
    bigquery_conn_id='airflow-service-account',
    google_cloud_storage_conn_id='airflow-service-account'
)
```

Corresponding Schema file (`employee_schema.json`):

```
[
  {
    "mode": "NULLABLE",
    "name": "emp_name",
    "type": "STRING"
  },
  {
    "mode": "REQUIRED",
    "name": "salary",
    "type": "INTEGER"
  }
]
```

Example (with schema in the DAG):

```
CreateTable = BigQueryCreateEmptyTableOperator(
    task_id='BigQueryCreateEmptyTableOperator_task',
    dataset_id='ODS',
    table_id='Employees',
    project_id='internal-gcp-project',
    schema_fields=[{"name": "emp_name", "type": "STRING", "mode":
↪"REQUIRED"},
                   {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}],
    bigquery_conn_id='airflow-service-account',
    google_cloud_storage_conn_id='airflow-service-account'
)
```

`template_fields = ['dataset_id', 'table_id', 'project_id', 'gcs_schema_object', 'label`

`ui_color = #f0eee4`

`execute`(*self*, *context*)

class airflow.contrib.operators.bigquery_operator.**BigQueryCreateExternalTableOperator**(*bucket, source_ des- ti- na- tion_pr schema schema source_ com- pres- sion='l skip_le field_d ', max_b quote_ al- low_qu al- low_ja big- query_ google_ del- e- gate_to src_fm la- bels=N *args, **kwa*

Bases: `airflow.models.baseoperator.BaseOperator`

Creates a new external table in the dataset with the data in Google Cloud Storage.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it.

Parameters

- **bucket** (*str*) – The bucket to point the external table to. (templated)
- **source_objects** (*list*) – List of Google cloud storage URIs to point table to. (templated) If source_format is 'DATASTORE_BACKUP', the list must only contain a single URI.
- **destination_project_dataset_table** (*str*) – The dotted `(<project>.)<dataset>.<table>` BigQuery table to load data into (templated). If `<project>` is not included, project will be the project defined in the connection json.
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

  Example:

```
schema_fields=[{"name": "emp_name", "type": "STRING", "mode":
↪"REQUIRED"},
                {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}]
```

Should not be set when source_format is 'DATASTORE_BACKUP'.

- **schema_object** (*str*) – If set, a GCS object path pointing to a .json file that contains the schema for the table. (templated)
- **source_format** (*str*) – File format of the data.
- **compression** (*str*) – [Optional] The compression type of the data source. Possible values include GZIP and NONE. The default value is NONE. This setting is ignored for Google Cloud Bigtable, Google Cloud Datastore backups and Avro formats.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **field_delimiter** (*str*) – The delimiter to use for the CSV.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **quote_character** (*str*) – The value that is used to quote data sections in a CSV file.
- **allow_quoted_newlines** (*bool*) – Whether to allow quoted newlines (true) or not (false).
- **allow_jagged_rows** (*bool*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable to CSV, ignored for other formats.
- **bigquery_conn_id** (*str*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*str*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **src_fmt_configs** (*dict*) – configure optional fields specific to the source format
- **labels** (*dict*) – a dictionary containing labels for the table, passed to BigQuery

template_fields = ['bucket', 'source_objects', 'schema_object', 'destination_project_d

ui_color = #f0eee4

execute (*self*, *context*)

class airflow.contrib.operators.bigquery_operator.**BigQueryDeleteDatasetOperator** (*dataset_id*, *project_id=Non* *big-query_conn_id=* *del-e-gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.baseoperator.BaseOperator*

This operator deletes an existing dataset from your Project in Big query. https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets/delete

Parameters

- **project_id** (*str*) – The project id of the dataset.
- **dataset_id** (*str*) – The dataset to be deleted.

Example:

```
delete_temp_data = BigQueryDeleteDatasetOperator(dataset_id = 'temp-dataset',
                                                 project_id = 'temp-project',
                                                 bigquery_conn_id='_my_gcp_conn_',
                                                 task_id='Deletetemp',
                                                 dag=dag)
```

template_fields = ['dataset_id', 'project_id']

ui_color = #f00004

execute(*self*, *context*)

class airflow.contrib.operators.bigquery_operator.**BigQueryCreateEmptyDatasetOperator**(*dataset_ project_ dataset_ big- query_c del- e- gate_to= *args, **kwarg*

Bases: *airflow.models.baseoperator.BaseOperator*

This operator is used to create new dataset for your Project in Big query. https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets#resource

Parameters

- **project_id** (*str*) – The name of the project where we want to create the dataset. Don't need to provide, if projectId in dataset_reference.
- **dataset_id** (*str*) – The id of dataset. Don't need to provide, if datasetId in dataset_reference.
- **dataset_reference** – Dataset reference that could be provided with request body. More info: https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets#resource

template_fields = ['dataset_id', 'project_id']

ui_color = #f0eee4

execute(*self*, *context*)

airflow.contrib.operators.bigquery_table_delete_operator

This module contains Google BigQuery table delete operator.

Module Contents

class airflow.contrib.operators.bigquery_table_delete_operator.**BigQueryTableDeleteOperator**

Bases: *airflow.models.BaseOperator*

Deletes BigQuery tables

Parameters

- **deletion_dataset_table** (*str*) – A dotted (<project>.|<project>:)<dataset>.<table> that indicates which table will be deleted. (templated)
- **bigquery_conn_id** (*str*) – reference to a specific BigQuery hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **ignore_if_missing** (*bool*) – if True, then return success even if the requested table does not exist.

template_fields = ['deletion_dataset_table']

ui_color = #ffd1dc

execute(*self*, *context*)

airflow.contrib.operators.bigquery_to_bigquery

This module contains a Google BigQuery to BigQuery operator.

Module Contents

class airflow.contrib.operators.bigquery_to_bigquery.**BigQueryToBigQueryOperator**(*source_project_d des- ti- na- tion_project_dat write_dispositior cre- ate_disposition= big- query_conn_id= del- e- gate_to=None, la- bels=None, *args, **kwargs*)

Bases: `airflow.models.BaseOperator`

Copies data from one BigQuery table to another.

See also:

For more details about these parameters: https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.copy

Parameters

- **source_project_dataset_tables** (*list|string*) – One or more dotted `(project:|project.)<dataset>.<table>` BigQuery tables to use as the source data. If `<project>` is not included, project will be the project defined in the connection json. Use a list if there are multiple source tables. (templated)
- **destination_project_dataset_table** (*str*) – The destination BigQuery table. Format is: `(project:|project.)<dataset>.<table>` (templated)
- **write_disposition** (*str*) – The write disposition if the table already exists.
- **create_disposition** (*str*) – The create disposition if the table doesn't exist.
- **bigquery_conn_id** (*str*) – reference to a specific BigQuery hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **labels** (*dict*) – a dictionary containing labels for the job/query, passed to BigQuery

template_fields = ['source_project_dataset_tables', 'destination_project_dataset_table

template_ext = ['.sql']

ui_color = #e6f0e4

execute(*self*, *context*)

airflow.contrib.operators.bigquery_to_gcs

This module contains a Google BigQuery to GCS operator.

Module Contents

class airflow.contrib.operators.bigquery_to_gcs.**BigQueryToCloudStorageOperator**(*source_project_da des- ti- na- tion_cloud_storag com- pres- sion='NONE', ex- port_format='CS field_delimiter=', ' print_header=Tru big- query_conn_id=' del- e- gate_to=None, la- bels=None, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Transfers a BigQuery table to a Google Cloud Storage bucket.

See also:

For more details about these parameters: https://cloud.google.com/bigquery/docs/reference/v2/jobs

Parameters

- **source_project_dataset_table** (*str*) – The dotted (<project>.|<project>:)<dataset>.<table> BigQuery table to use as the source data. If <project> is not included, project will be the project defined in the connection json. (templated)
- **destination_cloud_storage_uris** (*list*) – The destination Google Cloud Storage URI (e.g. gs://some-bucket/some-file.txt). (templated) Follows convention defined here: https://cloud.google.com/bigquery/exporting-data-from-bigquery#exportingmultiple
- **compression** (*str*) – Type of compression to use.
- **export_format** (*str*) – File format to export.
- **field_delimiter** (*str*) – The delimiter to use when extracting to a CSV.
- **print_header** (*bool*) – Whether to print a header for a CSV file extract.
- **bigquery_conn_id** (*str*) – reference to a specific BigQuery hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **labels** (*dict*) – a dictionary containing labels for the job/query, passed to BigQuery

template_fields = ['source_project_dataset_table', 'destination_cloud_storage_uris', '

template_ext = ['.sql']

ui_color = #e4e6f0

execute (*self*, *context*)

airflow.contrib.operators.cassandra_to_gcs

Module Contents

class airflow.contrib.operators.cassandra_to_gcs.**CassandraToGoogleCloudStorageOperator** (*cql*, *bucke* *file-* *name* *schen* *ap-* *prox_* *cas-* *san-* *dra_c* *googl* *del-* *e-* *gate_* **args* ***kw*

Bases: *airflow.models.BaseOperator*

Copy data from Cassandra to Google cloud storage in JSON format

Note: Arrays of arrays are not supported.

Parameters

- **cql** (*str*) – The CQL to execute on the Cassandra table.
- **bucket** (*str*) – The bucket to upload to.
- **filename** (*str*) – The filename to use as the object name when uploading to Google cloud storage. A {} should be specified in the filename to allow the operator to inject file numbers in cases where the file is split due to size.
- **schema_filename** (*str*) – If set, the filename to use as the object name when uploading a .json file containing the BigQuery schema fields for the table that was dumped from MySQL.
- **approx_max_file_size_bytes** (*long*) – This operator supports the ability to split large table dumps into multiple files (see notes in the filename param docs above). This param allows developers to specify the file size of the splits. Check https://cloud.google.com/storage/quotas to see the maximum allowed file size for a single object.
- **cassandra_conn_id** (*str*) – Reference to a specific Cassandra hook.
- **google_cloud_storage_conn_id** (*str*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['cql', 'bucket', 'filename', 'schema_filename']

template_ext = ['.cql']

ui_color = #a0e08c

CQL_TYPE_MAP

execute (*self*, *context*)

_query_cassandra (*self*)
Queries cassandra and returns a cursor to the results.

_write_local_data_files (*self*, *cursor*)
Takes a cursor, and writes results to a local file.

Returns A dictionary where keys are filenames to be used as object names in GCS, and values are file handles to local files that contain the data for the GCS objects.

_write_local_schema_file (*self*, *cursor*)
Takes a cursor, and writes the BigQuery schema for the results to a local file system.

Returns A dictionary where key is a filename to be used as an object name in GCS, and values are file handles to local files that contains the BigQuery schema fields in .json format.

_upload_to_gcs (*self*, *files_to_upload*)

classmethod generate_data_dict (*cls*, *names*, *values*)

classmethod convert_value (*cls*, *name*, *value*)

classmethod convert_array_types (*cls*, *name*, *value*)

classmethod convert_user_type (*cls*, *name*, *value*)
Converts a user type to RECORD that contains n fields, where n is the number of attributes. Each element in the user type class will be converted to its corresponding data type in BQ.

classmethod convert_tuple_type (*cls*, *name*, *value*)
Converts a tuple to RECORD that contains n fields, each will be converted to its corresponding data type in bq and will be named 'field_<index>', where index is determined by the order of the tuple elements defined in cassandra.

classmethod convert_map_type (*cls*, *name*, *value*)
Converts a map to a repeated RECORD that contains two fields: 'key' and 'value', each will be converted to its corresponding data type in BQ.

classmethod generate_schema_dict (*cls*, *name*, *type*)

classmethod get_bq_fields (*cls*, *name*, *type*)

classmethod is_simple_type (*cls*, *type*)

classmethod is_array_type (*cls*, *type*)

classmethod is_record_type (*cls*, *type*)

classmethod get_bq_type (*cls*, *type*)

classmethod get_bq_mode (*cls*, *type*)

airflow.contrib.operators.databricks_operator

This module contains Databricks operators.

Module Contents

airflow.contrib.operators.databricks_operator.**XCOM_RUN_ID_KEY = run_id**

airflow.contrib.operators.databricks_operator.**XCOM_RUN_PAGE_URL_KEY = run_page_url** `airflow.contrib.operators.databricks_operator.`**`_deep_string_coerce`**(*content*, *json_path='json'*)

`Coerces content or all values of content if it is a dict to a string. The function will throw if content contains non-string or non-numeric types.`

The reason why we have this function is because the `self.json` field must be a dict with only string values. This is because `render_template` will fail for numerical values.

`airflow.contrib.operators.databricks_operator.`**`_handle_databricks_operator_execution`**(*operator*, *hook*, *log*, *con-text*)

`Handles the Airflow + Databricks lifecycle logic for a Databricks operator`

Parameters

- **`operator`** – Databricks operator being handled
- **`context`** – Airflow context

`class` `airflow.contrib.operators.databricks_operator.`**`DatabricksSubmitRunOperator`**(*json=None*, *spark_jar_task=* *note-book_task=Non* *new_cluster=No* *ex-ist-ing_cluster_id=* *li-braries=None*, *run_name=Non* *time-out_seconds=No* *databricks_conn* *polling_period_* *databricks_retry* *databricks_retry* *do_xcom_push=* ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Submits a Spark job run to Databricks using the api/2.0/jobs/runs/submit API endpoint.

There are two ways to instantiate this operator.

In the first way, you can take the JSON payload that you typically use to call the `api/2.0/jobs/runs/submit` endpoint and pass it directly to our `DatabricksSubmitRunOperator` through the `json` parameter. For example

```
json = {
  'new_cluster': {
    'spark_version': '2.1.0-db3-scala2.11',
    'num_workers': 2
  },
  'notebook_task': {
    'notebook_path': '/Users/airflow@example.com/PrepareData',
  },
}
notebook_run = DatabricksSubmitRunOperator(task_id='notebook_run', json=json)
```

Another way to accomplish the same thing is to use the named parameters of the `DatabricksSubmitRunOperator` directly. Note that there is exactly one named parameter for each top level parameter in the `runs/submit` endpoint. In this method, your code would look like this:

```
new_cluster = {
  'spark_version': '2.1.0-db3-scala2.11',
  'num_workers': 2
}
notebook_task = {
  'notebook_path': '/Users/airflow@example.com/PrepareData',
}
notebook_run = DatabricksSubmitRunOperator(
    task_id='notebook_run',
    new_cluster=new_cluster,
    notebook_task=notebook_task)
```

In the case where both the json parameter **AND** the named parameters are provided, they will be merged together. If there are conflicts during the merge, the named parameters will take precedence and override the top level `json` keys.

Currently the named parameters that `DatabricksSubmitRunOperator` supports are

- `spark_jar_task`
- `notebook_task`
- `new_cluster`
- `existing_cluster_id`
- `libraries`
- `run_name`
- `timeout_seconds`

Parameters

- **`json`** (*dict*) – A JSON object containing API parameters which will be passed directly to the `api/2.0/jobs/runs/submit` endpoint. The other named parameters (i.e. `spark_jar_task`, `notebook_task`..) to this operator will be merged with this json dictionary if they are provided. If there are conflicts during the merge, the named parameters will take precedence and override the top level json keys. (templated)

  See also:

  For more information about templating see *Jinja Templating*. https://docs.databricks.com/api/latest/jobs.html#runs-submit

- **`spark_jar_task`** (*dict*) – The main class and parameters for the JAR task. Note that the actual JAR is specified in the `libraries`. *EITHER* `spark_jar_task` *OR* `notebook_task` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobssparkjartask

- **`notebook_task`** (*dict*) – The notebook path and parameters for the notebook task. *EITHER* `spark_jar_task` *OR* `notebook_task` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobsnotebooktask

- **new_cluster** (*dict*) – Specs for a new cluster on which this task will be run. *EITHER* `new_cluster` *OR* `existing_cluster_id` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobsclusterspecnewcluster

- **existing_cluster_id** (*str*) – ID for existing cluster on which to run this task. *EITHER* `new_cluster` *OR* `existing_cluster_id` should be specified. This field will be templated.
- **libraries** (*list of dicts*) – Libraries which this run will use. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/libraries.html#managedlibrarieslibrary

- **run_name** (*str*) – The run name used for this task. By default this will be set to the Airflow `task_id`. This `task_id` is a required parameter of the superclass `BaseOperator`. This field will be templated.
- **timeout_seconds** (*int32*) – The timeout for this run. By default a value of 0 is used which means to have no timeout. This field will be templated.
- **databricks_conn_id** (*str*) – The name of the Airflow connection to use. By default and in the common case this will be `databricks_default`. To use token based authentication, provide the key `token` in the extra field for the connection.
- **polling_period_seconds** (*int*) – Controls the rate which we poll for the result of this run. By default the operator will poll every 30 seconds.
- **databricks_retry_limit** (*int*) – Amount of times retry if the Databricks backend is unreachable. Its value must be greater than or equal to 1.
- **databricks_retry_delay** (*float*) – Number of seconds to wait between retries (it might be a floating point number).
- **do_xcom_push** (*bool*) – Whether we should push run_id and run_page_url to xcom.

template_fields = ['json']

ui_color = #1CB1C2

ui_fgcolor = #fff

_get_hook (*self*)

execute (*self*, *context*)

on_kill (*self*)

class airflow.contrib.operators.databricks_operator.**DatabricksRunNowOperator** (*job_id*, *json=None*, *note-book_params=None*, *python_params=Nor* *spark_submit_param* *databricks_conn_id=* *polling_period_secon* *databricks_retry_lim* *databricks_retry_del* *do_xcom_push=Fals* ***kwargs*)

Bases: *airflow.models.BaseOperator*

Runs an existing Spark job run to Databricks using the api/2.0/jobs/run-now API endpoint.

There are two ways to instantiate this operator.

In the first way, you can take the JSON payload that you typically use to call the `api/2.0/jobs/run-now` endpoint and pass it directly to our `DatabricksRunNowOperator` through the `json` parameter. For example

```
json = {
  "job_id": 42,
  "notebook_params": {
    "dry-run": "true",
    "oldest-time-to-consider": "1457570074236"
  }
}

notebook_run = DatabricksRunNowOperator(task_id='notebook_run', json=json)
```

Another way to accomplish the same thing is to use the named parameters of the `DatabricksRunNowOperator` directly. Note that there is exactly one named parameter for each top level parameter in the `run-now` endpoint. In this method, your code would look like this:

```
job_id=42

notebook_params = {
    "dry-run": "true",
    "oldest-time-to-consider": "1457570074236"
}

python_params = ["douglas adams", "42"]

spark_submit_params = ["--class", "org.apache.spark.examples.SparkPi"]

notebook_run = DatabricksRunNowOperator(
    job_id=job_id,
    notebook_params=notebook_params,
    python_params=python_params,
    spark_submit_params=spark_submit_params
)
```

In the case where both the json parameter **AND** the named parameters are provided, they will be merged together. If there are conflicts during the merge, the named parameters will take precedence and override the top level `json` keys.

Currently the named parameters that `DatabricksRunNowOperator` supports are

- `job_id`
- `json`
- `notebook_params`
- `python_params`
- `spark_submit_params`

Parameters

- **`job_id`** (*str*) – the job_id of the existing Databricks job. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#run-now

- **json** (*dict*) – A JSON object containing API parameters which will be passed directly to the `api/2.0/jobs/run-now` endpoint. The other named parameters (i.e. `notebook_params`, `spark_submit_params`..) to this operator will be merged with this json dictionary if they are provided. If there are conflicts during the merge, the named parameters will take precedence and override the top level json keys. (templated)

  See also:

  For more information about templating see *Jinja Templating*. https://docs.databricks.com/api/latest/jobs.html#run-now

- **notebook_params** (*dict*) – A dict from keys to values for jobs with notebook task, e.g. "notebook_params": {"name": "john doe", "age": "35"}. The map is passed to the notebook and will be accessible through the dbutils.widgets.get function. See Widgets for more information. If not specified upon run-now, the triggered run will use the job's base parameters. notebook_params cannot be specified in conjunction with jar_params. The json representation of this field (i.e. {"notebook_params":{"name":"john doe","age":"35"}}) cannot exceed 10,000 bytes. This field will be templated.

  See also:

  https://docs.databricks.com/user-guide/notebooks/widgets.html

- **python_params** (*list[str]*) – A list of parameters for jobs with python tasks, e.g. "python_params": ["john doe", "35"]. The parameters will be passed to python file as command line parameters. If specified upon run-now, it would overwrite the parameters specified in job setting. The json representation of this field (i.e. {"python_params":["john doe","35"]}) cannot exceed 10,000 bytes. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#run-now

- **spark_submit_params** (*list[str]*) – A list of parameters for jobs with spark submit task, e.g. "spark_submit_params": ["–class", "org.apache.spark.examples.SparkPi"]. The parameters will be passed to spark-submit script as command line parameters. If specified upon run-now, it would overwrite the parameters specified in job setting. The json representation of this field cannot exceed 10,000 bytes. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#run-now

- **timeout_seconds** (*int32*) – The timeout for this run. By default a value of 0 is used which means to have no timeout. This field will be templated.
- **databricks_conn_id** (*str*) – The name of the Airflow connection to use. By default and in the common case this will be `databricks_default`. To use token based authentication, provide the key `token` in the extra field for the connection.
- **polling_period_seconds** (*int*) – Controls the rate which we poll for the result of this run. By default the operator will poll every 30 seconds.
- **databricks_retry_limit** (*int*) – Amount of times retry if the Databricks backend is unreachable. Its value must be greater than or equal to 1.
- **do_xcom_push** (*bool*) – Whether we should push run_id and run_page_url to xcom.

template_fields = ['json']

ui_color = #1CB1C2

ui_fgcolor = #fff

_get_hook (*self*)

execute(*self*, *context*)

on_kill(*self*)

airflow.contrib.operators.dataflow_operator

This module contains Google Dataflow operators.

Module Contents

class airflow.contrib.operators.dataflow_operator.**DataFlowJavaOperator**(*jar*, *job_name='{{task.task_id}}'*, *dataflow_default_options=No*, *op-tions=None*, *gcp_conn_id='google_cloud_c*, *del-e-gate_to=None*, *poll_sleep=10*, *job_class=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Java Cloud DataFlow batch job. The parameters of the operation will be passed to the job.

Example:

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date':
        (2016, 8, 1),
    'email': ['alex@vanboxel.be'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=30),
    'dataflow_default_options': {
        'project': 'my-gcp-project',
        'zone': 'us-central1-f',
        'stagingLocation': 'gs://bucket/tmp/dataflow/staging/',
    }
}

dag = DAG('test-dag', default_args=default_args)

task = DataFlowJavaOperator(
    gcp_conn_id='gcp_default',
    task_id='normalize-cal',
    jar='{{var.value.gcp_dataflow_base}}pipeline-ingress-cal-normalize-1.0.jar',
    options={
        'autoscalingAlgorithm': 'BASIC',
        'maxNumWorkers': '50',
        'start': '{{ds}}',
```

(continues on next page)

(continued from previous page)

```
        'partitionType': 'DAY'

    },
    dag=dag)
```

See also:

For more detail on job submission have a look at the reference: https://cloud.google.com/dataflow/pipelines/specifying-exec-params

Parameters

- **jar** (*str*) – The reference to a self executing DataFlow jar (templated).
- **job_name** (*str*) – The 'jobName' to use when executing the DataFlow job (templated). This ends up being set in the pipeline options, so any entry with key `'jobName'` in `options` will be overwritten.
- **dataflow_default_options** (*dict*) – Map of default job options.
- **options** (*dict*) – Map of job specific options.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **poll_sleep** (*int*) – The time in seconds to sleep between polling Google Cloud Platform for the dataflow job status while the job is in the JOB_STATE_RUNNING state.
- **job_class** (*str*) – The name of the dataflow job class to be executed, it is often not the main class configured in the dataflow jar file.

`jar`, `options`, and `job_name` are templated so you can use variables in them.

Note that both `dataflow_default_options` and `options` will be merged to specify pipeline execution parameter, and `dataflow_default_options` is expected to save high-level options, for instances, project and zone information, which apply to all dataflow operators in the DAG.

It's a good practice to define dataflow_* parameters in the default_args of the dag like the project, zone and staging location.

```
default_args = {
    'dataflow_default_options': {
        'project': 'my-gcp-project',
        'zone': 'europe-west1-d',
        'stagingLocation': 'gs://my-staging-bucket/staging/'
    }
}
```

You need to pass the path to your dataflow as a file reference with the `jar` parameter, the jar needs to be a self executing jar (see documentation here: https://beam.apache.org/documentation/runners/dataflow/#self-executing-jar). Use `options` to pass on options to your job.

```
t1 = DataFlowJavaOperator(
    task_id='datapflow_example',
    jar='{{var.value.gcp_dataflow_base}}pipeline/build/libs/pipeline-example-1.0.
↪jar',
    options={
        'autoscalingAlgorithm': 'BASIC',
        'maxNumWorkers': '50',
```

(continues on next page)

(continued from previous page)

```
        'start': '{{ds}}',
        'partitionType': 'DAY',
        'labels': {'foo' : 'bar'}
    },
    gcp_conn_id='gcp-airflow-service-account',
    dag=my-dag)
```

template_fields = ['options', 'jar', 'job_name']

ui_color = #0273d4

execute (*self*, *context*)

class airflow.contrib.operators.dataflow_operator.**DataflowTemplateOperator** (*template*, *job_name='{{task.task_* *dataflow_default_optior* *pa-* *ram-* *e-* *ters=None*, *gcp_conn_id='google_c* *del-* *e-* *gate_to=None*, *poll_sleep=10*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Templated Cloud DataFlow batch job. The parameters of the operation will be passed to the job.

Parameters

- **template** (*str*) – The reference to the DataFlow template.
- **job_name** – The 'jobName' to use when executing the DataFlow template (templated).
- **dataflow_default_options** (*dict*) – Map of default job environment options.
- **parameters** (*dict*) – Map of job specific parameters for the template.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **poll_sleep** (*int*) – The time in seconds to sleep between polling Google Cloud Platform for the dataflow job status while the job is in the JOB_STATE_RUNNING state.

It's a good practice to define dataflow_* parameters in the default_args of the dag like the project, zone and staging location.

See also:

https://cloud.google.com/dataflow/docs/reference/rest/v1b3/LaunchTemplateParameters https://cloud.google.com/dataflow/docs/reference/rest/v1b3/RuntimeEnvironment

```
default_args = {
    'dataflow_default_options': {
        'project': 'my-gcp-project',
        'region': 'europe-west1',
        'zone': 'europe-west1-d',
```

(continues on next page)

(continued from previous page)

```
          'tempLocation': 'gs://my-staging-bucket/staging/',
          }
      }
}
```

You need to pass the path to your dataflow template as a file reference with the `template` parameter. Use `parameters` to pass on parameters to your job. Use `environment` to pass on runtime environment variables to your job.

```
t1 = DataflowTemplateOperator(
    task_id='datapflow_example',
    template='{{var.value.gcp_dataflow_base}}',
    parameters={
        'inputFile': "gs://bucket/input/my_input.txt",
        'outputFile': "gs://bucket/output/my_output.txt"
    },
    gcp_conn_id='gcp-airflow-service-account',
    dag=my-dag)
```

`template`, `dataflow_default_options`, `parameters`, and `job_name` are templated so you can use variables in them.

Note that `dataflow_default_options` is expected to save high-level options for project information, which apply to all dataflow operators in the DAG.

See also:

https://cloud.google.com/dataflow/docs/reference/rest/v1b3 /LaunchTemplateParameters https://cloud.google.com/dataflow/docs/reference/rest/v1b3/RuntimeEnvironment For more detail on job template execution have a look at the reference: https://cloud.google.com/dataflow/docs/templates/executing-templates

template_fields = ['parameters', 'dataflow_default_options', 'template', 'job_name']

ui_color = #0273d4

execute (*self*, *context*)

class airflow.contrib.operators.dataflow_operator.**DataFlowPythonOperator** (*py_file*, *job_name='{{task.task_id}*, *py_options=None*, *dataflow_default_options=*, *options=None*, *gcp_conn_id='google_clou*, *delegate_to=None*, *poll_sleep=10*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Launching Cloud Dataflow jobs written in python. Note that both dataflow_default_options and options will be merged to specify pipeline execution parameter, and dataflow_default_options is expected to save high-level options, for instances, project and zone information, which apply to all dataflow operators in the DAG.

See also:

For more detail on job submission have a look at the reference: https://cloud.google.com/dataflow/pipelines/specifying-exec-params

Parameters

- **py_file** (*str*) – Reference to the python dataflow pipeline file.py, e.g., /some/local/file/path/to/your/python/pipeline/file.
- **job_name** (*str*) – The 'job_name' to use when executing the DataFlow job (templated). This ends up being set in the pipeline options, so any entry with key `'jobName'` or `'job_name'` in `options` will be overwritten.
- **py_options** – Additional python options, e.g., ["-m", "-v"].
- **dataflow_default_options** (*dict*) – Map of default job options.
- **options** (*dict*) – Map of job specific options.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **poll_sleep** (*int*) – The time in seconds to sleep between polling Google Cloud Platform for the dataflow job status while the job is in the JOB_STATE_RUNNING state.

template_fields = ['options', 'dataflow_default_options', 'job_name']

execute (*self*, *context*)
Execute the python dataflow job.

class airflow.contrib.operators.dataflow_operator.**GoogleCloudBucketHelper** (*gcp_conn_id='google_clo del- e- gate_to=None*)

GoogleCloudStorageHook helper class to download GCS object.

GCS_PREFIX_LENGTH = 5

google_cloud_to_local (*self*, *file_name*)
Checks whether the file specified by file_name is stored in Google Cloud Storage (GCS), if so, downloads the file and saves it locally. The full path of the saved file will be returned. Otherwise the local file_name will be returned immediately.

Parameters **file_name** (*str*) – The full path of input file.

Returns The full path of local file.

Return type str

airflow.contrib.operators.dataproc_operator

This module contains Google Dataproc operators.

Module Contents

class airflow.contrib.operators.dataproc_operator.**DataprocOperationBaseOperator**(*project_id*, *re-gion='global'*, *gcp_conn_id='g* *del-e-gate_to=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

The base class for operators that poll on a Dataproc Operation.

execute(*self*, *context*)

start(*self*, *context*)
    You are expected to override the method.

class airflow.contrib.operators.dataproc_operator.**DataprocClusterCreateOperator**(*project_id*, *cluster_name*, *num_workers*, *zone=None*, *network_uri=None*, *subnetwork_uri=None*, *internal_ip_only=No* *tags=None*, *storage_bucket=Nor* *init_actions_uris* *init_action_time* *metadata=None*, *custom_image=Nor* *image_version=No* *autoscaling_policy=Non* *properties=None*, *master_machine* *standard-4'*, *master_disk_typ* *standard'*, *master_disk_size=1(* *worker_machin* *standard-4'*, *worker_disk_typ* *standard'*, *worker_disk_siz* *num_preemptibl* *labels=None*, *region='global'*, *service_account=N* *service_account_sc* *idle_delete_ttl=1* *auto_delete_tim* *auto_delete_ttl=* *customer_managed* **args*, ***kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataprocOperationBaseOperator*

Create a new cluster on Google Cloud Dataproc. The operator will wait until the creation is successful or an error occurs in the creation process.

The parameters allow to configure the cluster. Please refer to

https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters

for a detailed explanation on the different parameters. Most of the configuration parameters detailed in the link are available as a parameter to this operator.

Parameters

- **cluster_name** (*str*) – The name of the DataProc cluster to create. (templated)
- **project_id** (*str*) – The ID of the google cloud project in which to create the cluster. (templated)
- **num_workers** (*int*) – The # of workers to spin up. If set to zero will spin up cluster in a single node mode
- **storage_bucket** (*str*) – The storage bucket to use, setting to None lets dataproc generate a custom one for you
- **init_actions_uris** (*list[str]*) – List of GCS uri's containing dataproc initialization scripts
- **init_action_timeout** (*str*) – Amount of time executable scripts in init_actions_uris has to complete
- **metadata** (*dict*) – dict of key-value google compute engine metadata entries to add to all instances
- **image_version** (*str*) – the version of software inside the Dataproc cluster
- **custom_image** (*str*) – custom Dataproc image for more info see https://cloud.google.com/dataproc/docs/guides/dataproc-images
- **autoscaling_policy** (*str*) – The autoscaling policy used by the cluster. Only resource names including projectid and location (region) are valid. Example: `projects/[projectId]/locations/[dataproc_region]/autoscalingPolicies/[policy_id]`
- **properties** (*dict*) – dict of properties to set on config files (e.g. spark-defaults.conf), see https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters#SoftwareConfig
- **master_machine_type** (*str*) – Compute engine machine type to use for the master node
- **master_disk_type** (*str*) – Type of the boot disk for the master node (default is `pd-standard`). Valid values: `pd-ssd` (Persistent Disk Solid State Drive) or `pd-standard` (Persistent Disk Hard Disk Drive).
- **master_disk_size** (*int*) – Disk size for the master node
- **worker_machine_type** (*str*) – Compute engine machine type to use for the worker nodes
- **worker_disk_type** (*str*) – Type of the boot disk for the worker node (default is `pd-standard`). Valid values: `pd-ssd` (Persistent Disk Solid State Drive) or `pd-standard` (Persistent Disk Hard Disk Drive).
- **worker_disk_size** (*int*) – Disk size for the worker nodes
- **num_preemptible_workers** (*int*) – The # of preemptible worker nodes to spin up

- **labels** (*dict*) – dict of labels to add to the cluster
- **zone** (*str*) – The zone where the cluster will be located. Set to None to auto-zone. (templated)
- **network_uri** (*str*) – The network uri to be used for machine communication, cannot be specified with subnetwork_uri
- **subnetwork_uri** (*str*) – The subnetwork uri to be used for machine communication, cannot be specified with network_uri
- **internal_ip_only** (*bool*) – If true, all instances in the cluster will only have internal IP addresses. This can only be enabled for subnetwork enabled networks
- **tags** (*list[str]*) – The GCE tags to add to all instances
- **region** (*str*) – leave as 'global', might become relevant in the future. (templated)
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **service_account** (*str*) – The service account of the dataproc instances.
- **service_account_scopes** (*list[str]*) – The URIs of service account scopes to be included.
- **idle_delete_ttl** (*int*) – The longest duration that cluster would keep alive while staying idle. Passing this threshold will cause cluster to be auto-deleted. A duration in seconds.
- **auto_delete_time** (*datetime.datetime*) – The time when cluster will be auto-deleted.
- **auto_delete_ttl** (*int*) – The life duration of cluster, the cluster will be auto-deleted at the end of this duration. A duration in seconds. (If auto_delete_time is set this parameter will be ignored)
- **customer_managed_key** (*str*) – The customer-managed key used for disk encryption `projects/[PROJECT_STORING_KEYS]/locations/[LOCATION]/keyRings/[KEY_RING_NAME]/cryptoKeys/[KEY_NAME]` # noqa # pylint: disable=line-too-long

template_fields = ['cluster_name', 'project_id', 'zone', 'region']

_get_init_action_timeout(*self*)

_build_gce_cluster_config(*self*, *cluster_data*)

_build_lifecycle_config(*self*, *cluster_data*)

_build_cluster_data(*self*)

start(*self*)

Create a new cluster on Google Cloud Dataproc.

class airflow.contrib.operators.dataproc_operator.**DataprocClusterScaleOperator**(*cluster_name*, *project_id*, *region='global'*, *num_workers=2*, *num_preemptible_* *graceful_decommission* **args*, ***kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataprocOperationBaseOperator*

Scale, up or down, a cluster on Google Cloud Dataproc. The operator will wait until the cluster is re-scaled.

Example:

```
t1 = DataprocClusterScaleOperator(
        task_id='dataproc_scale',
        project_id='my-project',
        cluster_name='cluster-1',
        num_workers=10,
        num_preemptible_workers=10,
        graceful_decommission_timeout='1h',
        dag=dag)
```

See also:

For more detail on about scaling clusters have a look at the reference: https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/scaling-clusters

Parameters

- **cluster_name** (*str*) – The name of the cluster to scale. (templated)
- **project_id** (*str*) – The ID of the google cloud project in which the cluster runs. (templated)
- **region** (*str*) – The region for the dataproc cluster. (templated)
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **num_workers** (*int*) – The new number of workers
- **num_preemptible_workers** (*int*) – The new number of preemptible workers
- **graceful_decommission_timeout** (*str*) – Timeout for graceful YARN decomissioning. Maximum value is 1d
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['cluster_name', 'project_id', 'region']

_build_scale_cluster_data(*self*)

static _get_graceful_decommission_timeout(*timeout*)

start(*self*)

Scale, up or down, a cluster on Google Cloud Dataproc.

class airflow.contrib.operators.dataproc_operator.**DataprocClusterDeleteOperator**(*cluster_name, project_id, region='global', *args, **kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataprocOperationBaseOperator*

Delete a cluster on Google Cloud Dataproc. The operator will wait until the cluster is destroyed.

Parameters

- **cluster_name** (*str*) – The name of the cluster to delete. (templated)
- **project_id** (*str*) – The ID of the google cloud project in which the cluster runs. (templated)

- **region** (*str*) – leave as 'global', might become relevant in the future. (templated)
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['cluster_name', 'project_id', 'region']

start (*self*)
Delete a cluster on Google Cloud Dataproc.

class airflow.contrib.operators.dataproc_operator.**DataProcJobBaseOperator** (*job_name='{{task.task_i cluster_name='cluster-1', dat-aproc_properties=None, dat-aproc_jars=None, gcp_conn_id='google_clo del-e-gate_to=None, re-gion='global', job_error_states=None, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

The base class for operators that launch job on DataProc.

Parameters

- **job_name** (*str*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes.
- **cluster_name** (*str*) – The name of the DataProc cluster.
- **dataproc_properties** (*dict*) – Map for the Hive properties. Ideal to put in default arguments
- **dataproc_jars** (*list*) – HCFS URIs of jar files to add to the CLASSPATH of the Hive server and Hadoop MapReduce (MR) tasks. Can contain Hive SerDes and UDFs. (templated)
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*str*) – The specified region where the dataproc cluster is created.
- **job_error_states** (*set*) – Job states that should be considered error states. Any states in this set will result in an error being raised and failure of the task. Eg, if the `CANCELLED` state should also be considered a task failure, pass in `{'ERROR', 'CANCELLED'}`. Possible values are currently only `'ERROR'` and `'CANCELLED'`, but could change in the future. Defaults to `{'ERROR'}`.

Variables dataproc_job_id (*str*) – The actual "jobId" as submitted to the Dataproc API. This is useful for identifying or linking to the job in the Google Cloud Console Dataproc UI, as the actual "jobId" submitted to the Dataproc API is appended with an 8 character random string.

job_type =

create_job_template (*self*)
Initialize *self.job_template* with default values

execute (*self*, *context*)

on_kill (*self*)
Callback called when the operator is killed. Cancel any running job.

class airflow.contrib.operators.dataproc_operator.**DataProcPigOperator** (*query=None*, *query_uri=None*, *variables=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator*

Start a Pig query Job on a Cloud DataProc cluster. The parameters of the operation will be passed to the cluster.

It's a good practice to define dataproc_* parameters in the default_args of the dag like the cluster name and UDFs.

```
default_args = {
    'cluster_name': 'cluster-1',
    'dataproc_pig_jars': [
        'gs://example/udf/jar/datafu/1.2.0/datafu.jar',
        'gs://example/udf/jar/gpig/1.2/gpig.jar'
    ]
}
```

You can pass a pig script as string or file reference. Use variables to pass on variables for the pig script to be resolved on the cluster or use the parameters to be resolved in the script as template parameters.

Example:

```
t1 = DataProcPigOperator(
        task_id='dataproc_pig',
        query='a_pig_script.pig',
        variables={'out': 'gs://example/output/{{ds}}'},
        dag=dag)
```

See also:

For more detail on about job submission have a look at the reference: https://cloud.google.com/dataproc/reference/rest/v1/projects.regions.jobs

Parameters

- **query** (*str*) – The query or reference to the query file (pg or pig extension). (templated)
- **query_uri** (*str*) – The uri of a pig script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query. (templated)

template_fields = ['query', 'variables', 'job_name', 'cluster_name', 'region', 'datapr

template_ext = ['.pg', '.pig']

ui_color = #0273d4

job_type = pigJob

execute (*self*, *context*)

class airflow.contrib.operators.dataproc_operator.**DataProcHiveOperator**(*query=None, query_uri=None, variables=None, *args, **kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator*

Start a Hive query Job on a Cloud DataProc cluster.

Parameters

- **query** (*str*) – The query or reference to the query file (q extension).
- **query_uri** (*str*) – The uri of a hive script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query.

template_fields = ['query', 'variables', 'job_name', 'cluster_name', 'region', 'datapr

template_ext = ['.q']

ui_color = #0273d4

job_type = hiveJob

execute(*self, context*)

class airflow.contrib.operators.dataproc_operator.**DataProcSparkSqlOperator**(*query=None, query_uri=None, variables=None, *args, **kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator*

Start a Spark SQL query Job on a Cloud DataProc cluster.

Parameters

- **query** (*str*) – The query or reference to the query file (q extension). (templated)
- **query_uri** (*str*) – The uri of a spark sql script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query. (templated)

template_fields = ['query', 'variables', 'job_name', 'cluster_name', 'region', 'datapr

template_ext = ['.q']

ui_color = #0273d4

job_type = sparkSqlJob

execute(*self, context*)

class airflow.contrib.operators.dataproc_operator.**DataProcSparkOperator**(*main_jar=None, main_class=None, arguments=None, archives=None, files=None, *args, **kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator*

Start a Spark Job on a Cloud DataProc cluster.

Parameters

- **main_jar** (*str*) – URI of the job jar provisioned on Cloud Storage. (use this or the main_class, not both together).
- **main_class** (*str*) – Name of the job class. (use this or the main_jar, not both together).
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory

template_fields = ['arguments', 'job_name', 'cluster_name', 'region', 'dataproc_jars']

ui_color = #0273d4

job_type = sparkJob

execute (*self*, *context*)

class airflow.contrib.operators.dataproc_operator.**DataProcHadoopOperator** (*main_jar=None*, *main_class=None*, *arguments=None*, *archives=None*, *files=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator*

Start a Hadoop Job on a Cloud DataProc cluster.

Parameters

- **main_jar** (*str*) – URI of the job jar provisioned on Cloud Storage. (use this or the main_class, not both together).
- **main_class** (*str*) – Name of the job class. (use this or the main_jar, not both together).
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory

template_fields = ['arguments', 'job_name', 'cluster_name', 'region', 'dataproc_jars']

ui_color = #0273d4

job_type = hadoopJob

execute (*self*, *context*)

class airflow.contrib.operators.dataproc_operator.**DataProcPySparkOperator** (*main*, *arguments=None*, *archives=None*, *pyfiles=None*, *files=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator*

Start a PySpark Job on a Cloud DataProc cluster.

Parameters

- **main** (*str*) – [Required] The Hadoop Compatible Filesystem (HCFS) URI of the main Python file to use as the driver. Must be a .py file.
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory
- **pyfiles** (*list*) – List of Python files to pass to the PySpark framework. Supported file types: .py, .egg, and .zip

template_fields = ['arguments', 'job_name', 'cluster_name', 'region', 'dataproc_jars']

ui_color = #0273d4

job_type = pysparkJob

static _generate_temp_filename (*filename*)

_upload_file_temp (*self*, *bucket*, *local_file*)
Upload a local file to a Google Cloud Storage bucket.

execute (*self*, *context*)

class airflow.contrib.operators.dataproc_operator.**DataprocWorkflowTemplateInstantiateOperat**

Bases: *airflow.contrib.operators.dataproc_operator.DataprocOperationBaseOperator*

Instantiate a WorkflowTemplate on Google Cloud Dataproc. The operator will wait until the WorkflowTemplate is finished executing.

See also:

Please refer to: https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiate

Parameters

- **template_id** (*str*) – The id of the template. (templated)
- **project_id** (*str*) – The ID of the google cloud project in which the template runs
- **region** (*str*) – leave as 'global', might become relevant in the future
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['template_id']

start (*self*)
Instantiate a WorkflowTemplate on Google Cloud Dataproc.

class airflow.contrib.operators.dataproc_operator.**DataprocWorkflowTemplateInstantiateInline**

Bases: *airflow.contrib.operators.dataproc_operator.DataprocOperationBaseOperator*

Instantiate a WorkflowTemplate Inline on Google Cloud Dataproc. The operator will wait until the WorkflowTemplate is finished executing.

See also:

Please refer to: https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiateInline

Parameters

- **`template`** (*map*) – The template contents. (templated)
- **`project_id`** (*str*) – The ID of the google cloud project in which the template runs
- **`region`** (*str*) – leave as 'global', might become relevant in the future
- **`gcp_conn_id`** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **`delegate_to`** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

`template_fields = ['template']`

`start`(*self*)

Instantiate a WorkflowTemplate Inline on Google Cloud Dataproc.

`airflow.contrib.operators.datastore_export_operator`

Module Contents

class `airflow.contrib.operators.datastore_export_operator.`**`DatastoreExportOperator`**(*bucket*, *namespace=None*, *datastore_conn_id*
cloud_storag
delegate_to=Non
entity_filter=N
labels=None,
polling_inter
overwrite_existing
**args*,
***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Export entities from Google Cloud Datastore to Cloud Storage

Parameters

- **`bucket`** (*str*) – name of the cloud storage bucket to backup data
- **`namespace`** (*str*) – optional namespace path in the specified Cloud Storage bucket to backup data. If this namespace does not exist in GCS, it will be created.
- **`datastore_conn_id`** (*str*) – the name of the Datastore connection id to use

- **cloud_storage_conn_id** (*str*) – the name of the cloud storage connection id to force-write backup
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **entity_filter** (*dict*) – description of what data from the project is included in the export, refer to https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- **labels** (*dict*) – client-assigned labels for cloud storage
- **polling_interval_in_seconds** (*int*) – number of seconds to wait before polling for execution status again
- **overwrite_existing** (*bool*) – if the storage bucket + namespace is not empty, it will be emptied prior to exports. This enables overwriting existing backups.

execute (*self*, *context*)

airflow.contrib.operators.datastore_import_operator

Module Contents

class airflow.contrib.operators.datastore_import_operator.**DatastoreImportOperator** (*bucket, file, namespace=None, entity_filter=No labels=None, datastore_conn_id delegate_to=Non polling_inter *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Import entities from Cloud Storage to Google Cloud Datastore

Parameters

- **bucket** (*str*) – container in Cloud Storage to store data
- **file** (*str*) – path of the backup metadata file in the specified Cloud Storage bucket. It should have the extension .overall_export_metadata
- **namespace** (*str*) – optional namespace of the backup metadata file in the specified Cloud Storage bucket.
- **entity_filter** (*dict*) – description of what data from the project is included in the export, refer to https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- **labels** (*dict*) – client-assigned labels for cloud storage

- **datastore_conn_id** (*str*) – the name of the connection id to use
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **polling_interval_in_seconds** (*int*) – number of seconds to wait before polling for execution status again

execute (*self*, *context*)

airflow.contrib.operators.dingding_operator

Module Contents

class airflow.contrib.operators.dingding_operator.**DingdingOperator** (*dingding_conn_id='dingding_defaul*
message_type='text', *message=None*, *at_mobiles=None*, *at_all=False*, **args*, ***kwargs*)

Bases: `airflow.operators.bash_operator.BaseOperator`

This operator allows you send Dingding message using Dingding custom bot. Get Dingding token from conn_id.password. And prefer set domain to conn_id.host, if not will use default `https://oapi.dingtalk.com`.

For more detail message in Dingding custom bot

Parameters

- **dingding_conn_id** (*str*) – The name of the Dingding connection to use
- **message_type** (*str*) – Message type you want to send to Dingding, support five type so far including text, link, markdown, actionCard, feedCard
- **message** (*str or dict*) – The message send to Dingding chat group
- **at_mobiles** (*list[str]*) – Remind specific users with this message
- **at_all** (*bool*) – Remind all people in group or not. If True, will overwrite `at_mobiles`

template_fields = ['message']

ui_color = #4ea4d4

execute (*self*, *context*)

airflow.contrib.operators.discord_webhook_operator

Module Contents

class airflow.contrib.operators.discord_webhook_operator.**DiscordWebhookOperator**(*http_conn_id=N web-hook_endpoint= mes-sage=”, user-name=None, avatar_url=Non tts=False, proxy=None, *args, **kwargs*)

Bases: *airflow.operators.http_operator.SimpleHttpOperator*

This operator allows you to post messages to Discord using incoming webhooks. Takes a Discord connection ID with a default relative webhook endpoint. The default endpoint can be overridden using the webhook_endpoint parameter (https://discordapp.com/developers/docs/resources/webhook).

Each Discord webhook can be pre-configured to use a specific username and avatar_url. You can override these defaults in this operator.

Parameters

- **http_conn_id** (*str*) – Http connection ID with host as “https://discord.com/api/” and default webhook endpoint in the extra field in the form of {“webhook_endpoint”: “webhooks/{webhook.id}/{webhook.token}”}
- **webhook_endpoint** (*str*) – Discord webhook endpoint in the form of “webhooks/{webhook.id}/{webhook.token}”
- **message** (*str*) – The message you want to send to your Discord channel (max 2000 characters). (templated)
- **username** (*str*) – Override the default username of the webhook. (templated)
- **avatar_url** (*str*) – Override the default avatar of the webhook
- **tts** (*bool*) – Is a text-to-speech message
- **proxy** (*str*) – Proxy to use to make the Discord webhook call

template_fields = ['username', 'message']

execute(*self, context*)
Call the DiscordWebhookHook to post message

airflow.contrib.operators.druid_operator

Module Contents

class airflow.contrib.operators.druid_operator.**DruidOperator**(*json_index_file, druid_ingest_conn_id='druid_ingest_default', max_ingestion_time=None, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Allows to submit a task directly to druid

Parameters

- **`json_index_file`** (*`str`*) – The filepath to the druid index specification
- **`druid_ingest_conn_id`** (*`str`*) – The connection id of the Druid overlord which accepts index jobs

`template_fields = ['index_spec_str']`

`template_ext = ['.json']`

`execute` (*self*, *context*)

`airflow.contrib.operators.ecs_operator`

Module Contents

class `airflow.contrib.operators.ecs_operator.`**`ECSOperator`**(*task_definition*, *cluster*, *overrides*, *aws_conn_id=None*, *region_name=None*, *launch_type='EC2'*, *group=None*, *placement_constraints=None*, *platform_version='LATEST'*, *network_configuration=None*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Execute a task on AWS EC2 Container Service

Parameters

- **`task_definition`** (*`str`*) – the task definition name on EC2 Container Service
- **`cluster`** (*`str`*) – the cluster name on EC2 Container Service
- **`overrides`** (*`dict`*) – the same parameter that boto3 will receive (templated): http://boto3.readthedocs.org/en/latest/reference/services/ecs.html#ECS.Client.run_task
- **`aws_conn_id`** (*`str`*) – connection id of AWS credentials / region name. If None, credential boto3 strategy will be used (http://boto3.readthedocs.io/en/latest/guide/configuration.html).
- **`region_name`** (*`str`*) – region name to use in AWS Hook. Override the region_name in connection (if provided)
- **`launch_type`** (*`str`*) – the launch type on which to run your task ('EC2' or 'FARGATE')
- **`group`** (*`str`*) – the name of the task group associated with the task
- **`placement_constraints`** (*`list`*) – an array of placement constraint objects to use for the task
- **`platform_version`** (*`str`*) – the platform version on which your task is running
- **`network_configuration`** (*`dict`*) – the network configuration for the task

`ui_color = #f0ede4`

`client`

arn

template_fields = ['overrides']

execute (*self*, *context*)

_wait_for_task_ended (*self*)

_check_success_task (*self*)

get_hook (*self*)

on_kill (*self*)

`airflow.contrib.operators.emr_add_steps_operator`

Module Contents

class airflow.contrib.operators.emr_add_steps_operator.**EmrAddStepsOperator** (*job_flow_id*, *aws_conn_id='s3_defau* *steps=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

An operator that adds steps to an existing EMR job_flow.

Parameters

- **job_flow_id** (`str`) – id of the JobFlow to add steps to. (templated)
- **aws_conn_id** (`str`) – aws connection to uses
- **steps** (`list`) – boto3 style steps to be added to the jobflow. (templated)

template_fields = ['job_flow_id', 'steps']

template_ext = []

ui_color = #f9c915

execute (*self*, *context*)

`airflow.contrib.operators.emr_create_job_flow_operator`

Module Contents

class airflow.contrib.operators.emr_create_job_flow_operator.**EmrCreateJobFlowOperator** (*aws_co* *emr_co* *job_flo* *re-* *gion_n* **args*, ***kwa*

Bases: `airflow.models.BaseOperator`

Creates an EMR JobFlow, reading the config from the EMR connection. A dictionary of JobFlow overrides can be passed that override the config from the connection.

Parameters

- **aws_conn_id** (`str`) – aws connection to uses

- **emr_conn_id** (*str*) – emr connection to use
- **job_flow_overrides** (*dict*) – boto3 style arguments to override emr_connection extra. (templated)

template_fields = ['job_flow_overrides']

template_ext = []

ui_color = #f9c915

execute(*self*, *context*)

airflow.contrib.operators.emr_terminate_job_flow_operator

Module Contents

class airflow.contrib.operators.emr_terminate_job_flow_operator.**EmrTerminateJobFlowOperato**

Bases: *airflow.models.BaseOperator*

Operator to terminate EMR JobFlows.

Parameters

- **job_flow_id** (*str*) – id of the JobFlow to terminate. (templated)
- **aws_conn_id** (*str*) – aws connection to uses

template_fields = ['job_flow_id']

template_ext = []

ui_color = #f9c915

execute(*self*, *context*)

airflow.contrib.operators.file_to_gcs

Module Contents

class airflow.contrib.operators.file_to_gcs.**FileToGoogleCloudStorageOperator**(*src*, *dst*, *bucket*, *google_cloud_storag* *mime_type='applica* *stream'*, *del-* *e-* *gate_to=None*, *gzip=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Uploads a file to Google Cloud Storage. Optionally can compress the file for upload.

Parameters

- **src** (*str*) – Path to the local file. (templated)
- **dst** (*str*) – Destination path within the specified bucket. (templated)
- **bucket** (*str*) – The bucket to upload to. (templated)
- **google_cloud_storage_conn_id** (*str*) – The Airflow connection ID to upload with
- **mime_type** (*str*) – The mime-type string
- **delegate_to** (*str*) – The account to impersonate, if any
- **gzip** (*bool*) – Allows for file to be compressed and uploaded as gzip

template_fields = ['src', 'dst', 'bucket']

execute (*self*, *context*)
Uploads the file to Google cloud storage

airflow.contrib.operators.file_to_wasb

Module Contents

class airflow.contrib.operators.file_to_wasb.**FileToWasbOperator** (*file_path*, *container_name*, *blob_name*, *wasb_conn_id='wasb_default'*, *load_options=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Uploads a file to Azure Blob Storage.

Parameters

- **file_path** (*str*) – Path to the file to load. (templated)
- **container_name** (*str*) – Name of the container. (templated)
- **blob_name** (*str*) – Name of the blob. (templated)
- **wasb_conn_id** (*str*) – Reference to the wasb connection.
- **load_options** (*dict*) – Optional keyword arguments that *WasbHook.load_file()* takes.

template_fields = ['file_path', 'container_name', 'blob_name']

execute (*self*, *context*)
Upload a file to Azure Blob Storage.

airflow.contrib.operators.gcp_bigtable_operator

This module contains Google Cloud Bigtable operators.

Module Contents

class airflow.contrib.operators.gcp_bigtable_operator.**BigtableValidationMixin**
Common class for Cloud Bigtable operators for validating required fields.

REQUIRED_ATTRIBUTES :Iterable[str] = []

_validate_inputs(*self*)

class airflow.contrib.operators.gcp_bigtable_operator.**BigtableInstanceCreateOperator**(*instance_*
main_cl
main_cl
*project_*
*replica_*
*replica_*
in-
stance_d
in-
stance_t
in-
stance_l
clus-
ter_node
clus-
ter_stora
time-
out=Non
**args*,
***kwarg*

Bases: *airflow.models.BaseOperator*, *airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMixin*

Creates a new Cloud Bigtable instance. If the Cloud Bigtable instance with the given ID exists, the operator does not compare its configuration and immediately succeeds. No changes are made to the existing instance.

For more details about instance creation have a look at the reference: https://googleapis.github.io/google-cloud-python/latest/bigtable/instance.html#google.cloud.bigtable.instance.Instance.create

See also:

For more information on how to use this operator, take a look at the guide: *BigtableInstanceCreateOperator*

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance to create.
- **main_cluster_id** (*str*) – The ID for main cluster for the new instance.
- **main_cluster_zone** (*str*) – The zone for main cluster See https://cloud.google.com/bigtable/docs/locations for more details.
- **project_id** (*str*) – Optional, the ID of the GCP project. If set to None or missing, the default project_id from the GCP connection is used.
- **replica_cluster_id** (*str*) – (optional) The ID for replica cluster for the new instance.
- **replica_cluster_zone** (*str*) – (optional) The zone for replica cluster.
- **instance_type** (*enums.IntEnum*) – (optional) The type of the instance.
- **instance_display_name** (*str*) – (optional) Human-readable name of the instance. Defaults to `instance_id`.
- **instance_labels** (*dict*) – (optional) Dictionary of labels to associate with the instance.
- **cluster_nodes** (*int*) – (optional) Number of nodes for cluster.
- **cluster_storage_type** (*enums.IntEnum*) – (optional) The type of storage.
- **timeout** (*int*) – (optional) timeout (in seconds) for instance creation. If None is not specified, Operator will wait indefinitely.

REQUIRED_ATTRIBUTES = ['instance_id', 'main_cluster_id', 'main_cluster_zone']

template_fields = ['project_id', 'instance_id', 'main_cluster_id', 'main_cluster_zone'

execute (*self*, *context*)

class airflow.contrib.operators.gcp_bigtable_operator.**BigtableInstanceDeleteOperator** (*instance_ project_ *args, **kwarg*

Bases: *airflow.models.BaseOperator*, *airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMixin*

Deletes the Cloud Bigtable instance, including its clusters and all related tables.

For more details about deleting instance have a look at the reference: https://googleapis.github.io/google-cloud-python/latest/bigtable/instance.html#google.cloud.bigtable.instance.Instance.delete

See also:

For more information on how to use this operator, take a look at the guide: *BigtableInstanceDeleteOperator*

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance to delete.
- **project_id** (*str*) – Optional, the ID of the GCP project. If set to None or missing, the default project_id from the GCP connection is used.

REQUIRED_ATTRIBUTES = ['instance_id']

template_fields = ['project_id', 'instance_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_bigtable_operator.**BigtableTableCreateOperator** (*instance_id, table_id, project_id=N initial_split_key column_familie *args, **kwargs*)

Bases: *airflow.models.BaseOperator*, *airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMixin*

Creates the table in the Cloud Bigtable instance.

For more details about creating table have a look at the reference: https://googleapis.github.io/google-cloud-python/latest/bigtable/table.html#google.cloud.bigtable.table.Table.create

See also:

For more information on how to use this operator, take a look at the guide: *BigtableTableCreateOperator*

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance that will hold the new table.
- **table_id** (*str*) – The ID of the table to be created.
- **project_id** (*str*) – Optional, the ID of the GCP project. If set to None or missing, the default project_id from the GCP connection is used.

- **initial_split_keys** (*list*) – (Optional) list of row keys in bytes that will be used to initially split the table into several tablets.
- **column_families** (*dict*) – (Optional) A map columns to create. The key is the column_id str and the value is a `google.cloud.bigtable.column_family.GarbageCollectionRule`

REQUIRED_ATTRIBUTES = ['instance_id', 'table_id']

template_fields = ['project_id', 'instance_id', 'table_id']

_compare_column_families (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_bigtable_operator.**BigtableTableDeleteOperator** (*instance_id*, *ta-ble_id*, *project_id=N* *app_profile_* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*, *airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMixin*

Deletes the Cloud Bigtable table.

For more details about deleting table have a look at the reference: https://googleapis.github.io/google-cloud-python/latest/bigtable/table.html#google.cloud.bigtable.table.Table.delete

See also:

For more information on how to use this operator, take a look at the guide: *BigtableTableDeleteOperator*

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance.
- **table_id** (*str*) – The ID of the table to be deleted.
- **project_id** (*str*) – Optional, the ID of the GCP project. If set to None or missing, the default project_id from the GCP connection is used.

Parm app_profile_id Application profile.

REQUIRED_ATTRIBUTES = ['instance_id', 'table_id']

template_fields = ['project_id', 'instance_id', 'table_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_bigtable_operator.**BigtableClusterUpdateOperator** (*instance_i* *clus-ter_id*, *nodes*, *project_id* **args*, ***kwargs*

Bases: *airflow.models.BaseOperator*, *airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMixin*

Updates a Cloud Bigtable cluster.

For more details about updating a Cloud Bigtable cluster, have a look at the reference: https://googleapis.github.io/google-cloud-python/latest/bigtable/cluster.html#google.cloud.bigtable.cluster.Cluster.update

See also:

For more information on how to use this operator, take a look at the guide: *BigtableClusterUpdateOperator*

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance.
- **cluster_id** (*str*) – The ID of the Cloud Bigtable cluster to update.
- **nodes** (*int*) – The desired number of nodes for the Cloud Bigtable cluster.
- **project_id** (*str*) – Optional, the ID of the GCP project.

REQUIRED_ATTRIBUTES = ['instance_id', 'cluster_id', 'nodes']

template_fields = ['project_id', 'instance_id', 'cluster_id', 'nodes']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_bigtable_operator.**BigtableTableWaitForReplicationSenso**

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*, *airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMixin*

Sensor that waits for Cloud Bigtable table to be fully replicated to its clusters. No exception will be raised if the instance or the table does not exist.

For more details about cluster states for a table, have a look at the reference: https://googleapis.github.io/google-cloud-python/latest/bigtable/table.html#google.cloud.bigtable.table.Table.get_cluster_states

See also:

For more information on how to use this operator, take a look at the guide: *BigtableTableWaitForReplicationSensor*

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance.
- **table_id** (*str*) – The ID of the table to check replication status.
- **project_id** (*str*) – Optional, the ID of the GCP project.

REQUIRED_ATTRIBUTES = ['instance_id', 'table_id']

template_fields = ['project_id', 'instance_id', 'table_id']

poke (*self*, *context*)

airflow.contrib.operators.gcp_cloud_build_operator

Operators that integrat with Google Cloud Build service.

Module Contents

airflow.contrib.operators.gcp_cloud_build_operator.**REGEX_REPO_PATH**

class airflow.contrib.operators.gcp_cloud_build_operator.**BuildProcessor**(*body*)

Processes build configurations to add additional functionality to support the use of operators.

The following improvements are made:

- It is required to provide the source and only one type can be given,
- It is possible to provide the source as the URL address instead dict.

Parameters **body** (`dict`) – The request body. See: https://cloud.google.com/cloud-build/docs/api/reference/rest/Shared.Types/Build

_verify_source(*self*)

_reformat_source(*self*)

_reformat_repo_source(*self*)

_reformat_storage_source(*self*)

process_body(*self*)

Processes the body passed in the constructor

Returns the body.

Type dict

static _convert_repo_url_to_dict(*source*)

Convert url to repository in Google Cloud Source to a format supported by the API

Example valid input:

```
https://source.developers.google.com/p/airflow-project/r/airflow-repo#branch-
↪name
```

static _convert_storage_url_to_dict(*storage_url*)

Convert url to object in Google Cloud Storage to a format supported by the API

Example valid input:

```
gs://bucket-name/object-name.tar.gz
```

class airflow.contrib.operators.gcp_cloud_build_operator.**CloudBuildCreateBuildOperator**(*body,
proje
gcp_c
api_v
*args
**kw*

Bases: *airflow.models.BaseOperator*

Starts a build with the specified configuration.

See also:

For more information on how to use this operator, take a look at the guide: *Trigger a build*

Parameters

- **body** (`dict`) – The request body. See: https://cloud.google.com/cloud-build/docs/api/reference/rest/Shared.Types/Build
- **gcp_conn_id** (`str`) – The connection ID to use to connect to Google Cloud Platform.
- **api_version** (`str`) – API version used (for example v1 or v1beta1).

template_fields = ['body', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

execute(*self*, *context*)

airflow.contrib.operators.gcp_compute_operator

This module contains Google Compute Engine operators.

Module Contents

class airflow.contrib.operators.gcp_compute_operator.**GceBaseOperator**(*zone*, *re-source_id*, *project_id=None*, *gcp_conn_id='google_cloud_defa* *api_version='v1'*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Abstract base operator for Google Compute Engine operators to inherit from.

_validate_inputs(*self*)

execute(*self*, *context*)

class airflow.contrib.operators.gcp_compute_operator.**GceInstanceStartOperator**(*zone*, *re-source_id*, *project_id=None*, *gcp_conn_id='goog* *api_version='v1'*, **args*, ***kwargs*)

Bases: `airflow.contrib.operators.gcp_compute_operator.GceBaseOperator`

Starts an instance in Google Compute Engine.

See also:

For more information on how to use this operator, take a look at the guide: *GceInstanceStartOperator*

Parameters

- **zone** (*str*) – Google Cloud Platform zone where the instance exists.
- **resource_id** (*str*) – Name of the Compute Engine instance resource.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.
- **api_version** (*str*) – Optional, API version used (for example v1 - or beta). Defaults to v1.
- **validate_body** – Optional, If set to False, body validation is not performed. Defaults to False.

template_fields = ['project_id', 'zone', 'resource_id', 'gcp_conn_id', 'api_version']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_compute_operator.**GceInstanceStopOperator**(*zone*, *re-source_id*, *project_id=None*, *gcp_conn_id='googl*, *api_version='v1'*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.gcp_compute_operator.GceBaseOperator*

Stops an instance in Google Compute Engine.

See also:

For more information on how to use this operator, take a look at the guide: *GceInstanceStopOperator*

Parameters

- **zone** (*str*) – Google Cloud Platform zone where the instance exists.
- **resource_id** (*str*) – Name of the Compute Engine instance resource.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.
- **api_version** (*str*) – Optional, API version used (for example v1 - or beta). Defaults to v1.
- **validate_body** – Optional, If set to False, body validation is not performed. Defaults to False.

template_fields = ['project_id', 'zone', 'resource_id', 'gcp_conn_id', 'api_version']

execute(*self*, *context*)

airflow.contrib.operators.gcp_compute_operator.**SET_MACHINE_TYPE_VALIDATION_SPECIFICATION**

class airflow.contrib.operators.gcp_compute_operator.**GceSetMachineTypeOperator**(*zone*, *re-source_id*, *body*, *project_id=None*, *gcp_conn_id='go*, *api_version='v1'*, *val-i-date_body=True*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.gcp_compute_operator.GceBaseOperator*

Changes the machine type for a stopped instance to the machine type specified in the request.

See also:

For more information on how to use this operator, take a look at the guide: *GceSetMachineTypeOperator*

Parameters

- **zone** (*str*) – Google Cloud Platform zone where the instance exists.
- **resource_id** (*str*) – Name of the Compute Engine instance resource.
- **body** (*dict*) – Body required by the Compute Engine setMachineType API, as described in https://cloud.google.com/compute/docs/reference/rest/v1/instances/setMachineType#request-body
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.
- **api_version** (*str*) – Optional, API version used (for example v1 - or beta). Defaults to v1.
- **validate_body** (*bool*) – Optional, If set to False, body validation is not performed. Defaults to False.

template_fields = ['project_id', 'zone', 'resource_id', 'gcp_conn_id', 'api_version']

_validate_all_body_fields (*self*)

execute (*self*, *context*)

airflow.contrib.operators.gcp_compute_operator.**GCE_INSTANCE_TEMPLATE_VALIDATION_PATCH_SPECI**

airflow.contrib.operators.gcp_compute_operator.**GCE_INSTANCE_TEMPLATE_FIELDS_TO_SANITIZE =**

class airflow.contrib.operators.gcp_compute_operator.**GceInstanceTemplateCopyOperator** (*resource body_pa project_i re-quest_id gcp_con api_vers val-i-date_bo *args, **kwarg*

Bases: *airflow.contrib.operators.gcp_compute_operator.GceBaseOperator*

Copies the instance template, applying specified changes.

See also:

For more information on how to use this operator, take a look at the guide: *GceInstanceTemplateCopyOperator*

Parameters

- **resource_id** (*str*) – Name of the Instance Template
- **body_patch** (*dict*) – Patch to the body of instanceTemplates object following rfc7386 PATCH semantics. The body_patch content follows https://cloud.google.com/compute/docs/reference/rest/v1/instanceTemplates Name field is required as we need to rename the template, all the other fields are optional. It is important to follow PATCH semantics - arrays are replaced fully, so if you need to update an array you should provide the whole target array as patch element.

- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **request_id** (*str*) – Optional, unique request_id that you might add to achieve full idempotence (for example when client call times out repeating the request with the same request id will not create a new instance template again). It should be in UUID format as defined in RFC 4122.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.
- **api_version** (*str*) – Optional, API version used (for example v1 - or beta). Defaults to v1.
- **validate_body** (*bool*) – Optional, If set to False, body validation is not performed. Defaults to False.

template_fields = ['project_id', 'resource_id', 'request_id', 'gcp_conn_id', 'api_vers

_validate_all_body_fields(*self*)

execute(*self*, *context*)

class airflow.contrib.operators.gcp_compute_operator.**GceInstanceGroupManagerUpdateTemplate**

Bases: *airflow.contrib.operators.gcp_compute_operator.GceBaseOperator*

Patches the Instance Group Manager, replacing source template URL with the destination one. API V1 does not have update/patch operations for Instance Group Manager, so you must use beta or newer API version. Beta is the default.

See also:

For more information on how to use this operator, take a look at the guide: *GceInstanceGroupManagerUpdateTemplateOperator*

Parameters

- **resource_id** (*str*) – Name of the Instance Group Manager
- **zone** (*str*) – Google Cloud Platform zone where the Instance Group Manager exists.
- **source_template** (*str*) – URL of the template to replace.
- **destination_template** (*str*) – URL of the target template.

- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **request_id** (*str*) – Optional, unique request_id that you might add to achieve full idempotence (for example when client call times out repeating the request with the same request id will not create a new instance template again). It should be in UUID format as defined in RFC 4122.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.
- **api_version** (*str*) – Optional, API version used (for example v1 - or beta). Defaults to v1.
- **validate_body** (*bool*) – Optional, If set to False, body validation is not performed. Defaults to False.

template_fields = ['project_id', 'resource_id', 'zone', 'request_id', 'source_template

_possibly_replace_template(*self*, *dictionary:Dict*)

execute(*self*, *context*)

airflow.contrib.operators.gcp_container_operator

This module contains Google Kubernetes Engine operators.

Module Contents

class airflow.contrib.operators.gcp_container_operator.**GKEClusterDeleteOperator**(*project_id*, *name*, *location*, *gcp_conn_id='g* *api_version='v2* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Deletes the cluster, including the Kubernetes endpoint and all worker nodes.

To delete a certain cluster, you must specify the `project_id`, the `name` of the cluster, the `location` that the cluster is in, and the `task_id`.

Operator Creation:

```
operator = GKEClusterDeleteOperator(
            task_id='cluster_delete',
            project_id='my-project',
            location='cluster-location'
            name='cluster-name')
```

See also:

For more detail about deleting clusters have a look at the reference: https://google-cloud-python.readthedocs.io/en/latest/container/gapic/v1/api.html#google.cloud.container_v1.ClusterManagerClient.delete_cluster

Parameters

- **project_id** (*str*) – The Google Developers Console [project ID or project number]
- **name** (*str*) – The name of the resource to delete, in this case cluster name
- **location** (*str*) – The name of the Google Compute Engine zone in which the cluster resides.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **api_version** (*str*) – The api version to use

template_fields = ['project_id', 'gcp_conn_id', 'name', 'location', 'api_version']

_check_input (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_container_operator.**GKEClusterCreateOperator** (*project_id*, *location*, *body=None*, *gcp_conn_id='g* *api_version='v2* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Create a Google Kubernetes Engine Cluster of specified dimensions The operator will wait until the cluster is created.

The **minimum** required to define a cluster to create is:

dict() ::

cluster_def = {'name': 'my-cluster-name', 'initial_node_count': 1}

or

Cluster proto :: from google.cloud.container_v1.types import Cluster

cluster_def = Cluster(name='my-cluster-name', initial_node_count=1)

Operator Creation:

```
operator = GKEClusterCreateOperator(
            task_id='cluster_create',
            project_id='my-project',
            location='my-location'
            body=cluster_def)
```

See also:

For more detail on about creating clusters have a look at the reference: google.cloud.container_v1.types.Cluster

Parameters

- **project_id** (*str*) – The Google Developers Console [project ID or project number]
- **location** (*str*) – The name of the Google Compute Engine zone in which the cluster resides.

- **body** (*dict or google.cloud.container_v1.types.Cluster*) – The Cluster definition to create, can be protobuf or python dict, if dict it must match protobuf message Cluster
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.
- **api_version** (*str*) – The api version to use

template_fields = ['project_id', 'gcp_conn_id', 'location', 'api_version', 'body']

_check_input (*self*)

execute (*self*, *context*)

airflow.contrib.operators.gcp_container_operator.**KUBE_CONFIG_ENV_VAR = KUBECONFIG**

airflow.contrib.operators.gcp_container_operator.**G_APP_CRED = GOOGLE_APPLICATION_CREDENTIAL**

class airflow.contrib.operators.gcp_container_operator.**GKEPodOperator** (*project_id*, *location*, *cluster_name*, *gcp_conn_id='google_cloud_de* **args*, ***kwargs*)

Bases: *airflow.contrib.operators.kubernetes_pod_operator.KubernetesPodOperator*

Executes a task in a Kubernetes pod in the specified Google Kubernetes Engine cluster

This Operator assumes that the system has gcloud installed and either has working default application credentials or has configured a connection id with a service account.

The **minimum** required to define a cluster to create are the variables `task_id`, `project_id`, `location`, `cluster_name`, `name`, `namespace`, and `image`

Operator Creation:

```
operator = GKEPodOperator(task_id='pod_op',
                          project_id='my-project',
                          location='us-central1-a',
                          cluster_name='my-cluster-name',
                          name='task-name',
                          namespace='default',
                          image='perl')
```

See also:

For more detail about application authentication have a look at the reference: https://cloud.google.com/docs/authentication/production#providing_credentials_to_your_application

Parameters

- **project_id** (*str*) – The Google Developers Console project id
- **location** (*str*) – The name of the Google Kubernetes Engine zone in which the cluster resides, e.g. 'us-central1-a'
- **cluster_name** (*str*) – The name of the Google Kubernetes Engine cluster the pod should be spawned in
- **gcp_conn_id** (*str*) – The google cloud connection id to use. This allows for users to specify a service account.

template_fields

execute(*self*, *context*)

_set_env_from_extras(*self*, *extras*)
Sets the environment variable *GOOGLE_APPLICATION_CREDENTIALS* with either:

- The path to the keyfile from the specified connection id
- **A generated file's path if the user specified JSON in the connection id. The** file is assumed to be deleted after the process dies due to how mkstemp() works.

The environment variable is used inside the gcloud command to determine correct service account to use.

_get_field(*self*, *extras*, *field*, *default=None*)
Fetches a field from extras, and returns it. This is some Airflow magic. The google_cloud_platform hook type adds custom UI elements to the hook page, which allow admins to specify service_account, key_path, etc. They get formatted as shown below.

airflow.contrib.operators.gcp_function_operator

This module contains Google Cloud Functions operators.

Module Contents

airflow.contrib.operators.gcp_function_operator.**_validate_available_memory_in_mb**(*value*)

airflow.contrib.operators.gcp_function_operator.**_validate_max_instances**(*value*)

airflow.contrib.operators.gcp_function_operator.**CLOUD_FUNCTION_VALIDATION**

class airflow.contrib.operators.gcp_function_operator.**GcfFunctionDeployOperator**(*location*, *body*, *project_id=Non* *gcp_conn_id='g* *api_version='v1* *zip_path=None*, *val-* *i-* *date_body=Tru* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Creates a function in Google Cloud Functions. If a function with this name already exists, it will be updated.

See also:

For more information on how to use this operator, take a look at the guide: *GcfFunctionDeployOperator*

Parameters

- **location** (*str*) – Google Cloud Platform region where the function should be created.
- **body** (*dict or google.cloud.functions.v1.CloudFunction*) – Body of the Cloud Functions definition. The body must be a Cloud Functions dictionary as described in: https://cloud.google.com/functions/docs/reference/rest/v1/projects.locations.functions . Different API versions require different variants of the Cloud Functions dictionary.
- **project_id** (*str*) – (Optional) Google Cloud Platform project ID where the function should be created.

- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform - default 'google_cloud_default'.
- **api_version** (*str*) – (Optional) API version used (for example v1 - default - or v1beta1).
- **zip_path** (*str*) – Path to zip file containing source code of the function. If the path is set, the sourceUploadUrl should not be specified in the body or it should be empty. Then the zip file will be uploaded using the upload URL generated via generateUploadUrl from the Cloud Functions API.
- **validate_body** (*bool*) – If set to False, body validation is not performed.

template_fields = ['project_id', 'location', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

_validate_all_body_fields(*self*)

_create_new_function(*self*)

_update_function(*self*)

_check_if_function_exists(*self*)

_upload_source_code(*self*)

_set_airflow_version_label(*self*)

execute(*self*, *context*)

airflow.contrib.operators.gcp_function_operator.**GCF_SOURCE_ARCHIVE_URL = sourceArchiveUrl**

airflow.contrib.operators.gcp_function_operator.**GCF_SOURCE_UPLOAD_URL = sourceUploadUrl**

airflow.contrib.operators.gcp_function_operator.**SOURCE_REPOSITORY = sourceRepository**

airflow.contrib.operators.gcp_function_operator.**GCF_ZIP_PATH = zip_path**

class airflow.contrib.operators.gcp_function_operator.**ZipPathPreprocessor**(*body*, *zip_path*)

Pre-processes zip path parameter.

Responsible for checking if the zip path parameter is correctly specified in relation with source_code body fields. Non empty zip path parameter is special because it is mutually exclusive with sourceArchiveUrl and sourceRepository body fields. It is also mutually exclusive with non-empty sourceUploadUrl. The pre-process modifies sourceUploadUrl body field in special way when zip_path is not empty. An extra step is run when execute method is called and sourceUploadUrl field value is set in the body with the value returned by generateUploadUrl Cloud Function API method.

Parameters

- **body** (*dict*) – Body passed to the create/update method calls.
- **zip_path** – path to the zip file containing source code.

upload_function

static _is_present_and_empty(*dictionary*, *field*)

_verify_upload_url_and_no_zip_path(*self*)

_verify_upload_url_and_zip_path(*self*)

_verify_archive_url_and_zip_path(*self*)

should_upload_function(*self*)

Checks if function source should be uploaded.

Return type bool

preprocess_body(*self*)
Modifies sourceUploadUrl body field in special way when zip_path is not empty.

airflow.contrib.operators.gcp_function_operator.**FUNCTION_NAME_PATTERN = ^projects/[^/]+/loc**

airflow.contrib.operators.gcp_function_operator.**FUNCTION_NAME_COMPILED_PATTERN**

class airflow.contrib.operators.gcp_function_operator.**GcfFunctionDeleteOperator**(*name*, *gcp_conn_id='g* *api_version='v1* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Deletes the specified function from Google Cloud Functions.

See also:

For more information on how to use this operator, take a look at the guide: *GcfFunctionDeleteOperator*

Parameters

- **name** (*str*) – A fully-qualified function name, matching the pattern: *^projects/[^/]+/locations/[^/]+/functions/[^/]+$*
- **gcp_conn_id** (*str*) – The connection ID to use to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (for example v1 or v1beta1).

template_fields = ['name', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

execute(*self*, *context*)

airflow.contrib.operators.gcp_natural_language_operator

This module contains Google Cloud Language operators.

Module Contents

class airflow.contrib.operators.gcp_natural_language_operator.**CloudLanguageAnalyzeEntities**

Bases: *airflow.models.BaseOperator*

Finds named entities in the text along with entity types, salience, mentions for each entity, and other properties.

See also:

For more information on how to use this operator, take a look at the guide: *Analyzing Entities*

Parameters

- **document** (*dict or google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*seq[tuple[str, str]]]*) – Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.

template_fields = ['document', 'gcp_conn_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_natural_language_operator.**CloudLanguageAnalyzeEntitySen**

Bases: *airflow.models.BaseOperator*

Finds entities, similar to AnalyzeEntities in the text and analyzes sentiment associated with each entity and its mentions.

See also:

For more information on how to use this operator, take a look at the guide: *Analyzing Entity Sentiment*

Parameters

- **document** (*dict or google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*seq[tuple[str, str]]]*) – Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.

Return type google.cloud.language_v1.types.AnalyzeEntitiesResponse

template_fields = ['document', 'gcp_conn_id']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_natural_language_operator.**CloudLanguageAnalyzeSentiment**

Bases: *airflow.models.BaseOperator*

Analyzes the sentiment of the provided text.

See also:

For more information on how to use this operator, take a look at the guide: *Analyzing Sentiment*

Parameters

- **document** (*dict or google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.

Return type google.cloud.language_v1.types.AnalyzeEntitiesResponse

template_fields = ['document', 'gcp_conn_id']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_natural_language_operator.**CloudLanguageClassifyTextOpe**

Bases: *airflow.models.BaseOperator*

Classifies a document into categories.

See also:

For more information on how to use this operator, take a look at the guide: *Classifying Content*

Parameters

- **document** (*dict or google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **retry** – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – The connection ID to use connecting to Google Cloud Platform.

template_fields = ['document', 'gcp_conn_id']

execute (*self*, *context*)

airflow.contrib.operators.gcp_spanner_operator

This module contains Google Spanner operators.

Module Contents

class airflow.contrib.operators.gcp_spanner_operator.**CloudSpannerInstanceDeployOperator** (*inst con fig- u- ra- tion nod dis- play proj gcp *ar **k*

Bases: *airflow.models.BaseOperator*

Creates a new Cloud Spanner instance, or if an instance with the same instance_id exists in the specified project, updates the Cloud Spanner instance.

Parameters

- **instance_id** (*str*) – Cloud Spanner instance ID.
- **configuration_name** (*str*) – The name of the Cloud Spanner instance configuration defining how the instance will be created. Required for instances that do not yet exist.
- **node_count** (*int*) – (Optional) The number of nodes allocated to the Cloud Spanner instance.
- **display_name** (*str*) – (Optional) The display name for the Cloud Spanner instance in the GCP Console. (Must be between 4 and 30 characters.) If this value is not set in the constructor, the name is the same as the instance ID.
- **project_id** (*str*) – Optional, the ID of the project which owns the Cloud Spanner Database. If set to None or missing, the default project_id from the GCP connection is used.

- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['project_id', 'instance_id', 'configuration_name', 'display_name',

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_spanner_operator.**CloudSpannerInstanceDeleteOperator** (*inst*
proj
gcp
**ar*
***k*

Bases: *airflow.models.BaseOperator*

Deletes a Cloud Spanner instance. If an instance does not exist, no action is taken and the operator succeeds.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSpannerInstanceDeleteOperator*

Parameters

- **instance_id** (*str*) – The Cloud Spanner instance ID.
- **project_id** (*str*) – Optional, the ID of the project that owns the Cloud Spanner Database. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['project_id', 'instance_id', 'gcp_conn_id']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_spanner_operator.**CloudSpannerInstanceDatabaseQueryOpera**

Bases: *airflow.models.BaseOperator*

Executes an arbitrary DML query (INSERT, UPDATE, DELETE).

See also:

For more information on how to use this operator, take a look at the guide: *CloudSpannerInstanceDatabaseQuery-Operator*

Parameters

- **instance_id** (*str*) – The Cloud Spanner instance ID.
- **database_id** (*str*) – The Cloud Spanner database ID.
- **query** (*str or list*) – The query or list of queries to be executed. Can be a path to a SQL file.
- **project_id** (*str*) – Optional, the ID of the project that owns the Cloud Spanner Database. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['project_id', 'instance_id', 'database_id', 'query', 'gcp_conn_id']

template_ext = ['.sql']

_validate_inputs (*self*)

execute (*self*, *context*)

static sanitize_queries (*queries*)
Drops empty query in queries.

Parameters **queries** (*List[str]*) – queries

Return type None

class airflow.contrib.operators.gcp_spanner_operator.**CloudSpannerInstanceDatabaseDeployOpe**

Bases: *airflow.models.BaseOperator*

Creates a new Cloud Spanner database, or if database exists, the operator does nothing.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSpannerInstanceDatabaseDeploy-Operator*

Parameters

- **instance_id** (*str*) – The Cloud Spanner instance ID.
- **database_id** (*str*) – The Cloud Spanner database ID.
- **ddl_statements** (*list[str]*) – The string list containing DDL for the new database.
- **project_id** (*str*) – Optional, the ID of the project that owns the Cloud Spanner Database. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['project_id', 'instance_id', 'database_id', 'ddl_statements', 'gcp_

template_ext = ['.sql']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_spanner_operator.**CloudSpannerInstanceDatabaseUpdateOpe**

Bases: *airflow.models.BaseOperator*

Updates a Cloud Spanner database with the specified DDL statement.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSpannerInstanceDatabaseUpdateOperator*

Parameters

- **instance_id** (*str*) – The Cloud Spanner instance ID.
- **database_id** (*str*) – The Cloud Spanner database ID.
- **ddl_statements** (*list[str]*) – The string list containing DDL to apply to the database.
- **project_id** (*str*) – Optional, the ID of the project that owns the the Cloud Spanner Database. If set to None or missing, the default project_id from the GCP connection is used.
- **operation_id** (*str*) – (Optional) Unique per database operation id that can be specified to implement idempotency check.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['project_id', 'instance_id', 'database_id', 'ddl_statements', 'gcp_

template_ext = ['.sql']

_validate_inputs(*self*)

execute(*self*, *context*)

class airflow.contrib.operators.gcp_spanner_operator.**CloudSpannerInstanceDatabaseDeleteOpe**

Bases: *airflow.models.BaseOperator*

Deletes a Cloud Spanner database.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSpannerInstanceDatabaseDeleteOperator*

Parameters

- **instance_id** (*str*) – Cloud Spanner instance ID.
- **database_id** (*str*) – Cloud Spanner database ID.
- **project_id** (*str*) – Optional, the ID of the project that owns the Cloud Spanner Database. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['project_id', 'instance_id', 'database_id', 'gcp_conn_id']

_validate_inputs(*self*)

execute(*self*, *context*)

airflow.contrib.operators.gcp_speech_to_text_operator

This module contains a Google Speech to Text operator.

Module Contents

class airflow.contrib.operators.gcp_speech_to_text_operator.**GcpSpeechToTextRecognizeSpeech**

Bases: *airflow.models.BaseOperator*

Recognizes speech from audio file and returns it as text.

See also:

For more information on how to use this operator, take a look at the guide: *GcpSpeechToTextRecognizeSpeechOperator*

Parameters

- **config** (*dict or google.cloud.speech_v1.types.RecognitionConfig*) – information to the recognizer that specifies how to process the request. See more: https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/types.html#google.cloud.speech_v1.types.RecognitionConfig
- **audio** (*dict or google.cloud.speech_v1.types.RecognitionAudio*) – audio data to be recognized. See more: https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/types.html#google.cloud.speech_v1.types.RecognitionAudio
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

template_fields = ['audio', 'config', 'project_id', 'gcp_conn_id', 'timeout']

_validate_inputs (*self*)

execute (*self*, *context*)

airflow.contrib.operators.gcp_sql_operator

This module contains Google Cloud SQL operators.

Module Contents

airflow.contrib.operators.gcp_sql_operator.**SETTINGS = settings** airflow.contrib.operators.gcp_sql_operator.**SETTINGS_VERSION = settingsVersion**

airflow.contrib.operators.gcp_sql_operator.**CLOUD_SQL_CREATE_VALIDATION**

airflow.contrib.operators.gcp_sql_operator.**CLOUD_SQL_EXPORT_VALIDATION**

airflow.contrib.operators.gcp_sql_operator.**CLOUD_SQL_IMPORT_VALIDATION**

airflow.contrib.operators.gcp_sql_operator.**CLOUD_SQL_DATABASE_CREATE_VALIDATION**

airflow.contrib.operators.gcp_sql_operator.**CLOUD_SQL_DATABASE_PATCH_VALIDATION**

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlBaseOperator**(*instance*, *project_id=None*, *gcp_conn_id='google_cloud_de*, *api_version='v1beta4'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Abstract base operator for Google Cloud SQL operators to inherit from.

Parameters

- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. f set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).

_validate_inputs(*self*)

_check_if_instance_exists(*self*, *instance*)

_check_if_db_exists(*self*, *db_name*)

execute(*self*, *context*)

static _get_settings_version(*instance*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstanceCreateOperator**(*body*, *in-stance*, *project_id=None*, *gcp_conn_id='g*, *api_version='v1*, *val-i-date_body=Tru*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Creates a new Cloud SQL instance. If an instance with the same name exists, no action will be taken and the operator will succeed.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstanceCreateOperator*

Parameters

- **body** (*dict*) – Body required by the Cloud SQL insert API, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/insert #request-body

- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).
- **validate_body** (*bool*) – True if body should be validated, False otherwise.

template_fields = ['project_id', 'instance', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

_validate_body_fields(*self*)

execute(*self, context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstancePatchOperator**(*body, instance, project_id=None, gcp_conn_id='go api_version='v1b *args, **kwargs*)

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Updates settings of a Cloud SQL instance.

Caution: This is a partial update, so only included values for the settings will be updated.

In the request body, supply the relevant portions of an instance resource, according to the rules of patch semantics. https://cloud.google.com/sql/docs/mysql/admin-api/how-tos/performance#patch

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstancePatchOperator*

Parameters

- **body** (*dict*) – Body required by the Cloud SQL patch API, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/patch#request-body
- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).

template_fields = ['project_id', 'instance', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

execute(*self, context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstanceDeleteOperator**(*instance, project_id=None gcp_conn_id='g api_version='v1 *args, **kwargs*)

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Deletes a Cloud SQL instance.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstanceDeleteOperator*

Parameters

- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).

template_fields = ['project_id', 'instance', 'gcp_conn_id', 'api_version']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstanceDatabaseCreateOperator** (*inst bod proj gcp_ api_ val- i- date *ar **k*

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Creates a new database inside a Cloud SQL instance.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstanceDatabaseCreateOperator*

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/databases/insert#request-body
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).
- **validate_body** (*bool*) – Whether the body should be validated. Defaults to True.

template_fields = ['project_id', 'instance', 'gcp_conn_id', 'api_version']

_validate_inputs (*self*)

_validate_body_fields (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstanceDatabasePatchOperator**(*instan datab body, projec gcp_c api_v val- i- date_ *args **kw*

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Updates a resource containing information about a database inside a Cloud SQL instance using patch semantics. See: https://cloud.google.com/sql/docs/mysql/admin-api/how-tos/performance#patch

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstanceDatabasePatchOperator*

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **database** (*str*) – Name of the database to be updated in the instance.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/databases/patch#request-body
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).
- **validate_body** (*bool*) – Whether the body should be validated. Defaults to True.

template_fields = ['project_id', 'instance', 'database', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

_validate_body_fields(*self*)

execute(*self*, *context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstanceDatabaseDeleteOperator**(*inst data proj gcp_ api_ *ar **k*

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Deletes a database from a Cloud SQL instance.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstanceDatabaseDeleteOperator*

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **database** (*str*) – Name of the database to be deleted in the instance.

- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).

template_fields = ['project_id', 'instance', 'database', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

execute(*self*, *context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstanceExportOperator**(*instance*, *body*, *project_id=Non* *gcp_conn_id='g* *api_version='v1* *val-* *i-* *date_body=Tru* **args*, ***kwargs*)

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Exports data from a Cloud SQL instance to a Cloud Storage bucket as a SQL dump or CSV file.

Note: This operator is idempotent. If executed multiple times with the same export file URI, the export file in GCS will simply be overridden.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstanceImportOperator*

Parameters

- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/export#request-body
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).
- **validate_body** (*bool*) – Whether the body should be validated. Defaults to True.

template_fields = ['project_id', 'instance', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

_validate_body_fields(*self*)

execute(*self*, *context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlInstanceImportOperator**(*instance, body, project_id=Non gcp_conn_id='g api_version='v1 val- i- date_body=Tru *args, **kwargs*)

Bases: *airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator*

Imports data into a Cloud SQL instance from a SQL dump or CSV file in Cloud Storage.

CSV IMPORT:

This operator is NOT idempotent for a CSV import. If the same file is imported multiple times, the imported data will be duplicated in the database. Moreover, if there are any unique constraints the duplicate import may result in an error.

SQL IMPORT:

This operator is idempotent for a SQL import if it was also exported by Cloud SQL. The exported SQL contains 'DROP TABLE IF EXISTS' statements for all tables to be imported.

If the import file was generated in a different way, idempotence is not guaranteed. It has to be ensured on the SQL file level.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlInstanceImportOperator*

Parameters

- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/export#request-body
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1beta4).
- **validate_body** (*bool*) – Whether the body should be validated. Defaults to True.

template_fields = ['project_id', 'instance', 'gcp_conn_id', 'api_version']

_validate_inputs(*self*)

_validate_body_fields(*self*)

execute(*self, context*)

class airflow.contrib.operators.gcp_sql_operator.**CloudSqlQueryOperator**(*sql, autocommit=False, parameters=None, gcp_conn_id='google_cloud_d gcp_cloudsql_conn_id='googl *args, **kwargs*)

Bases: `airflow.models.BaseOperator`

Performs DML or DDL query on an existing Cloud Sql instance. It optionally uses cloud-sql-proxy to establish secure connection with the database.

See also:

For more information on how to use this operator, take a look at the guide: *CloudSqlQueryOperator*

Parameters

- **sql** (*str or list[str]*) – SQL query or list of queries to run (should be DML or DDL query - this operator does not return any data from the database, so it is useless to pass it DQL queries. Note that it is responsibility of the author of the queries to make sure that the queries are idempotent. For example you can use CREATE TABLE IF NOT EXISTS to create a table.
- **parameters** (*mapping or iterable*) – (optional) the parameters to render the SQL query with.
- **autocommit** (*bool*) – if True, each command is automatically committed. (default value: False)
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform for cloud-sql-proxy authentication.
- **gcp_cloudsql_conn_id** (*str*) – The connection ID used to connect to Google Cloud SQL its schema should be gcpcloudsql://. See `CloudSqlDatabaseHook` for details on how to define gcpcloudsql:// connection.

template_fields = ['sql', 'gcp_cloudsql_conn_id', 'gcp_conn_id']

template_ext = ['.sql']

_execute_query(*self*)

execute(*self, context*)

airflow.contrib.operators.gcp_text_to_speech_operator

This module contains a Google Text to Speech operator.

Module Contents

class airflow.contrib.operators.gcp_text_to_speech_operator.**GcpTextToSpeechSynthesizeOperat**

Bases: *airflow.models.BaseOperator*

Synthesizes text to speech and stores it in Google Cloud Storage

See also:

For more information on how to use this operator, take a look at the guide: *GcpTextToSpeechSynthesizeOperator*

Parameters

- **input_data** (*dict or google.cloud.texttospeech_v1.types.SynthesisInput*) – text input to be synthesized. See more: https://googleapis.github.io/google-cloud-python/latest/texttospeech/gapic/v1/types.html#google.cloud.texttospeech_v1.types.SynthesisInput
- **voice** (*dict or google.cloud.texttospeech_v1.types.VoiceSelectionParams*) – configuration of voice to be used in synthesis. See more: https://googleapis.github.io/google-cloud-python/latest/texttospeech/gapic/v1/types.html#google.cloud.texttospeech_v1.types.VoiceSelectionParams
- **audio_config** (*dict or google.cloud.texttospeech_v1.types.AudioConfig*) – configuration of the synthesized audio. See more: https://googleapis.github.io/google-cloud-python/latest/texttospeech/gapic/v1/types.html#google.cloud.texttospeech_v1.types.AudioConfig
- **target_bucket_name** (*str*) – name of the GCS bucket in which output file should be stored
- **target_filename** (*str*) – filename of the output file.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

template_fields = ['input_data', 'voice', 'audio_config', 'project_id', 'gcp_conn_id',

`_validate_inputs` (*self*)

`execute` (*self*, *context*)

`airflow.contrib.operators.gcp_transfer_operator`

This module contains Google Cloud Transfer operators.

Module Contents

`airflow.contrib.operators.gcp_transfer_operator.`**`AwsHook`**

`class` `airflow.contrib.operators.gcp_transfer_operator.`**`TransferJobPreprocessor`** (*body*, *aws_conn_id='aws*

Helper class for preprocess of transfer job body.

`_inject_aws_credentials` (*self*)

`_reformat_date` (*self*, *field_key*)

`_reformat_time` (*self*, *field_key*)

`_reformat_schedule` (*self*)

`process_body` (*self*)

Injects AWS credentials into body if needed and reformats schedule information.

Returns Preprocessed body

Return type dict

`static _convert_date_to_dict` (*field_date*)

Convert native python `datetime.date` object to a format supported by the API

`static _convert_time_to_dict` (*time_object*)

Convert native python `datetime.time` object to a format supported by the API

`class` `airflow.contrib.operators.gcp_transfer_operator.`**`TransferJobValidator`** (*body*)

Helper class for validating transfer job body.

`_verify_data_source` (*self*)

`_restrict_aws_credentials` (*self*)

`validate_body` (*self*)

Validates the body. Checks if body specifies *transferSpec* if yes, then check if AWS credentials are passed correctly and no more than 1 data source was selected.

Raises AirflowException

`class` `airflow.contrib.operators.gcp_transfer_operator.`**`GcpTransferServiceJobCreateOperator`** (*l*

Bases: *`airflow.models.BaseOperator`*

Creates a transfer job that runs periodically.

> **Warning:** This operator is NOT idempotent. If you run it many times, many transfer jobs will be created in the Google Cloud Platform.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceJobCreateOperator*

Parameters

- **body** (*dict*) – (Required) The request body, as described in https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs/create#request-body With three additional improvements:
  - dates can be given in the form `datetime.date`
  - times can be given in the form `datetime.time`
  - credentials to Amazon Web Service should be stored in the connection and indicated by the aws_conn_id parameter
- **aws_conn_id** (*str*) – The connection ID used to retrieve credentials to Amazon Web Service.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1).

template_fields = ['body', 'gcp_conn_id', 'aws_conn_id']

_validate_inputs(*self*)

execute(*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**GcpTransferServiceJobUpdateOperator**(*j*
b
a
g
a
***
***

Bases: *airflow.models.BaseOperator*

Updates a transfer job that runs periodically.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceJobUpdateOperator*

Parameters

- **job_name** (*str*) – (Required) Name of the job to be updated
- **body** (*dict*) – (Required) The request body, as described in https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs/patch#request-body With three additional improvements:
  - dates can be given in the form `datetime.date`
  - times can be given in the form `datetime.time`
  - credentials to Amazon Web Service should be stored in the connection and indicated by the aws_conn_id parameter
- **aws_conn_id** (*str*) – The connection ID used to retrieve credentials to Amazon Web Service.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1).

template_fields = ['job_name', 'body', 'gcp_conn_id', 'aws_conn_id']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**GcpTransferServiceJobDeleteOperator** (*j*
g
a
p
***
***

Bases: `airflow.models.BaseOperator`

Delete a transfer job. This is a soft delete. After a transfer job is deleted, the job and all the transfer executions are subject to garbage collection. Transfer jobs become eligible for garbage collection 30 days after soft delete.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceJobDeleteOperator*

Parameters

- **job_name** (*str*) – (Required) Name of the TRANSFER operation
- **project_id** (*str*) – (Optional) the ID of the project that owns the Transfer Job. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1).

template_fields = ['job_name', 'project_id', 'gcp_conn_id', 'api_version']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**GcpTransferServiceOperationGetOperato**

Bases: `airflow.models.BaseOperator`

Gets the latest state of a long-running operation in Google Storage Transfer Service.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceOperationGetOperator*

Parameters

- **operation_name** (*str*) – (Required) Name of the transfer operation.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1).

template_fields = ['operation_name', 'gcp_conn_id']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**GcpTransferServiceOperationsListOpera**

Bases: *airflow.models.BaseOperator*

Lists long-running operations in Google Storage Transfer Service that match the specified filter.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceOperationsListOperator*

Parameters

- **filter** (*dict*) – (Required) A request filter, as described in https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs/list#body.QUERY_PARAMETERS.filter
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1).

template_fields = ['filter', 'gcp_conn_id']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**GcpTransferServiceOperationPauseOpera**

Bases: *airflow.models.BaseOperator*

Pauses a transfer operation in Google Storage Transfer Service.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceOperationPauseOperator*

Parameters

- **operation_name** (*str*) – (Required) Name of the transfer operation.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1).

template_fields = ['operation_name', 'gcp_conn_id', 'api_version']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**GcpTransferServiceOperationResumeOpe**

Bases: *airflow.models.BaseOperator*

Resumes a transfer operation in Google Storage Transfer Service.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceOperationResume-Operator*

Parameters

- **operation_name** (*str*) – (Required) Name of the transfer operation.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.
- **api_version** (*str*) – API version used (e.g. v1).

template_fields = ['operation_name', 'gcp_conn_id', 'api_version']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**GcpTransferServiceOperationCancelOpe**

Bases: *airflow.models.BaseOperator*

Cancels a transfer operation in Google Storage Transfer Service.

See also:

For more information on how to use this operator, take a look at the guide: *GcpTransferServiceOperationCancel-Operator*

Parameters

- **operation_name** (*str*) – (Required) Name of the transfer operation.
- **api_version** (*str*) – API version used (e.g. v1).
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['operation_name', 'gcp_conn_id', 'api_version']

_validate_inputs (*self*)

execute (*self*, *context*)

class airflow.contrib.operators.gcp_transfer_operator.**S3ToGoogleCloudStorageTransferOperato**

Bases: *airflow.models.BaseOperator*

Synchronizes an S3 bucket with a Google Cloud Storage bucket using the GCP Storage Transfer Service.

Warning: This operator is NOT idempotent. If you run it many times, many transfer jobs will be created in the Google Cloud Platform.

Example:

```
s3_to_gcs_transfer_op = S3ToGoogleCloudStorageTransferOperator(
     task_id='s3_to_gcs_transfer_example',
     s3_bucket='my-s3-bucket',
     project_id='my-gcp-project',
     gcs_bucket='my-gcs-bucket',
     dag=my_dag)
```

Parameters

- **s3_bucket** (*str*) – The S3 bucket where to find the objects. (templated)
- **gcs_bucket** (*str*) – The destination Google Cloud Storage bucket where you want to store the files. (templated)
- **project_id** (*str*) – Optional ID of the Google Cloud Platform Console project that owns the job
- **aws_conn_id** (*str*) – The source S3 connection
- **gcp_conn_id** (*str*) – The destination connection ID to use when connecting to Google Cloud Storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **description** (*str*) – Optional transfer service job description

- **schedule** (*dict*) – Optional transfer service schedule; If not set, run transfer job once as soon as the operator runs The format is described https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs. With two additional improvements:
  - dates they can be passed as `datetime.date`
  - times they can be passed as `datetime.time`
- **object_conditions** (*dict*) – Optional transfer service object conditions; see https://cloud.google.com/storage-transfer/docs/reference/rest/v1/TransferSpec
- **transfer_options** (*dict*) – Optional transfer service transfer options; see https://cloud.google.com/storage-transfer/docs/reference/rest/v1/TransferSpec
- **wait** (*bool*) – Wait for transfer to finish
- **timeout** (*int*) – Time to wait for the operation to end in seconds

template_fields = ['gcp_conn_id', 's3_bucket', 'gcs_bucket', 'description', 'object_co

ui_color = #e09411

execute (*self*, *context*)

_create_body (*self*)

class airflow.contrib.operators.gcp_transfer_operator.**GoogleCloudStorageToGoogleCloudStora**

Bases: *airflow.models.BaseOperator*

Copies objects from a bucket to another using the GCP Storage Transfer Service.

> **Warning:** This operator is NOT idempotent. If you run it many times, many transfer jobs will be created in the Google Cloud Platform.

Example:

```
gcs_to_gcs_transfer_op = GoogleCloudStorageToGoogleCloudStorageTransferOperator(
     task_id='gcs_to_gcs_transfer_example',
     source_bucket='my-source-bucket',
     destination_bucket='my-destination-bucket',
     project_id='my-gcp-project',
     dag=my_dag)
```

Parameters

- **source_bucket** (*str*) – The source Google cloud storage bucket where the object is. (templated)
- **destination_bucket** (*str*) – The destination Google cloud storage bucket where the object should be. (templated)
- **project_id** (*str*) – The ID of the Google Cloud Platform Console project that owns the job
- **gcp_conn_id** (*str*) – Optional connection ID to use when connecting to Google Cloud Storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **description** (*str*) – Optional transfer service job description
- **schedule** (*dict*) – Optional transfer service schedule; If not set, run transfer job once as soon as the operator runs See: https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs. With two additional improvements:
  - dates they can be passed as `datetime.date`
  - times they can be passed as `datetime.time`
- **object_conditions** (*dict*) – Optional transfer service object conditions; see https://cloud.google.com/storage-transfer/docs/reference/rest/v1/TransferSpec#ObjectConditions
- **transfer_options** (*dict*) – Optional transfer service transfer options; see https://cloud.google.com/storage-transfer/docs/reference/rest/v1/TransferSpec#TransferOptions
- **wait** (*bool*) – Wait for transfer to finish; defaults to *True*
- **timeout** (*int*) – Time to wait for the operation to end in seconds

template_fields = ['gcp_conn_id', 'source_bucket', 'destination_bucket', 'description'

ui_color = #e09411

execute (*self*, *context*)

_create_body (*self*)

airflow.contrib.operators.gcp_translate_operator

This module contains Google Translate operators.

Module Contents

class airflow.contrib.operators.gcp_translate_operator.**CloudTranslateTextOperator**(*values*, *tar-get_language*, *for-mat_*, *source_langu*, *model*, *gcp_conn_id*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Translate a string or list of strings.

See also:

For more information on how to use this operator, take a look at the guide: *CloudTranslateTextOperator*

See https://cloud.google.com/translate/docs/translating-text

Execute method returns str or list.

This is a list of dictionaries for each queried value. Each dictionary typically contains three keys (though not all will be present in all cases).

- `detectedSourceLanguage`: The detected language (as an ISO 639-1 language code) of the text.
- `translatedText`: The translation of the text into the target language.
- `input`: The corresponding input value.
- `model`: The model used to translate the text.

If only a single value is passed, then only a single dictionary is set as XCom return value.

Parameters

- **values** (*str or list*) – String or list of strings to translate.
- **target_language** (*str*) – The language to translate results into. This is required by the API and defaults to the target language of the current instance.
- **format** (*str or None*) – (Optional) One of `text` or `html`, to specify if the input text is plain text or HTML.
- **source_language** (*str or None*) – (Optional) The language of the text to be translated.
- **model** (*str or None*) – (Optional) The model used to translate the text, such as `'base'` or `'nmt'`.

template_fields = ['values', 'target_language', 'format_', 'source_language', 'model',

execute(*self*, *context*)

airflow.contrib.operators.gcp_translate_speech_operator

This module contains a Google Cloud Translate Speech operator.

Module Contents

class airflow.contrib.operators.gcp_translate_speech_operator.**GcpTranslateSpeechOperator**(*au co fig ta ge fo m so m pr gc *a ***

Bases: *airflow.models.BaseOperator*

Recognizes speech in audio input and translates it.

Note that it uses the first result from the recognition api response - the one with the highest confidence In order to see other possible results please use *GcpSpeechToTextRecognizeSpeechOperator* and *CloudTranslateTextOperator* separately

See also:

For more information on how to use this operator, take a look at the guide: *GcpTranslateSpeechOperator*

See https://cloud.google.com/translate/docs/translating-text

Execute method returns string object with the translation

This is a list of dictionaries queried value. Dictionary typically contains three keys (though not all will be present in all cases).

- `detectedSourceLanguage`: The detected language (as an ISO 639-1 language code) of the text.
- `translatedText`: The translation of the text into the target language.
- `input`: The corresponding input value.
- `model`: The model used to translate the text.

Dictionary is set as XCom return value.

Parameters

- **audio** (*dict or google.cloud.speech_v1.types.RecognitionAudio*) – audio data to be recognized. See more: https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/types.html#google.cloud.speech_v1.types.RecognitionAudio
- **config** (*dict or google.cloud.speech_v1.types.RecognitionConfig*) – information to the recognizer that specifies how to process the request. See more: https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/types.html#google.cloud.speech_v1.types.RecognitionConfig
- **target_language** (*str*) – The language to translate results into. This is required by the API and defaults to the target language of the current instance. Check the list of available languages here: https://cloud.google.com/translate/docs/languages
- **format** (*str or None*) – (Optional) One of `text` or `html`, to specify if the input text is plain text or HTML.
- **source_language** (*str or None*) – (Optional) The language of the text to be translated.

- **model** (*str or None*) – (Optional) The model used to translate the text, such as `'base'` or `'nmt'`.
- **project_id** (*str*) – Optional, Google Cloud Platform Project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to 'google_cloud_default'.

template_fields = ['target_language', 'format_', 'source_language', 'model', 'project_

execute (*self*, *context*)

airflow.contrib.operators.gcp_video_intelligence_operator

This module contains Google Cloud Vision operators.

Module Contents

class airflow.contrib.operators.gcp_video_intelligence_operator.**CloudVideoIntelligenceDetec**

Bases: *airflow.models.BaseOperator*

Performs video annotation, annotating video labels.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVideoIntelligenceDetectVideoLabelsOperator*.

Parameters

- **input_uri** (*str*) – Input video location. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **input_content** (*bytes*) – The video data bytes. If unset, the input video(s) should be specified via `input_uri`. If set, `input_uri` should be unset.
- **output_uri** (*str*) – Optional, location where the output (in JSON format) should be stored. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **video_context** (*dict or google.cloud.videointelligence_v1.types.VideoContext*) – Optional, Additional video context and/or feature-specific parameters.

- **location** (*str*) – Optional, cloud region where annotation should take place. Supported cloud regions: us-east1, us-west1, europe-west1, asia-east1. If no region is specified, a region will be determined based on video file location.
- **retry** (*google.api_core.retry.Retry*) – Retry object used to determine when/if to retry requests. If None is specified, requests will not be retried.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to `google_cloud_default`.

template_fields = ['input_uri', 'output_uri', 'gcp_conn_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_video_intelligence_operator.**CloudVideoIntelligenceDete**

Bases: *airflow.models.BaseOperator*

Performs video annotation, annotating explicit content.

See also:

For more information on how to use this operator, take a look at the guide: *More information*

Parameters

- **input_uri** (*str*) – Input video location. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **input_content** (*bytes*) – The video data bytes. If unset, the input video(s) should be specified via `input_uri`. If set, `input_uri` should be unset.
- **output_uri** (*str*) – Optional, location where the output (in JSON format) should be stored. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **video_context** (*dict or google.cloud.videointelligence_v1.types.VideoContext*) – Optional, Additional video context and/or feature-specific parameters.
- **location** (*str*) – Optional, cloud region where annotation should take place. Supported cloud regions: us-east1, us-west1, europe-west1, asia-east1. If no region is specified, a region will be determined based on video file location.
- **retry** (*google.api_core.retry.Retry*) – Retry object used to determine when/if to retry requests. If None is specified, requests will not be retried.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to `google_cloud_default`.

template_fields = ['input_uri', 'output_uri', 'gcp_conn_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_video_intelligence_operator.**CloudVideoIntelligenceDetec**

Bases: *airflow.models.BaseOperator*

Performs video annotation, annotating video shots.

See also:

For more information on how to use this operator, take a look at the guide: *More information*

Parameters

- **input_uri** (*str*) – Input video location. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **input_content** (*bytes*) – The video data bytes. If unset, the input video(s) should be specified via `input_uri`. If set, `input_uri` should be unset.
- **output_uri** (*str*) – Optional, location where the output (in JSON format) should be stored. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **video_context** (*dict or google.cloud.videointelligence_v1.types.VideoContext*) – Optional, Additional video context and/or feature-specific parameters.
- **location** (*str*) – Optional, cloud region where annotation should take place. Supported cloud regions: us-east1, us-west1, europe-west1, asia-east1. If no region is specified, a region will be determined based on video file location.
- **retry** (*google.api_core.retry.Retry*) – Retry object used to determine when/if to retry requests. If None is specified, requests will not be retried.
- **gcp_conn_id** (*str*) – Optional, The connection ID used to connect to Google Cloud Platform. Defaults to `google_cloud_default`.

template_fields = ['input_uri', 'output_uri', 'gcp_conn_id']

execute (*self*, *context*)

airflow.contrib.operators.gcp_vision_operator

This module contains a Google Cloud Vision operator.

Module Contents

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductSetCreateOperator**(*pro lo- ca- tion proj prod uct_ retr time out= met data gcp_ *ar **k*

Bases: *airflow.models.BaseOperator*

Creates a new ProductSet resource.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductSetCreateOperator*

Parameters

- **product_set** (*dict or google.cloud.vision_v1.types.ProductSet*) – (Required) The ProductSet to create. If a dict is provided, it must be of the same form as the protobuf message *ProductSet*.
- **location** (*str*) – (Required) The region where the ProductSet should be created. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **project_id** (*str*) – (Optional) The project in which the ProductSet should be created. If set to None or missing, the default project_id from the GCP connection is used.
- **product_set_id** (*str*) – (Optional) A user-supplied resource id for this ProductSet. If set, the server will attempt to use this value as the resource id. If it is already in use, an error is returned with code ALREADY_EXISTS. Must be at most 128 characters long. It cannot contain the character /.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_set_id', 'gcp_conn_id']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductSetGetOperator**(*location, prod-uct_set_ project_ retry=N time-out=Nor meta-data=N gcp_con *args, **kwarg*

Bases: `airflow.models.BaseOperator`

Gets information associated with a ProductSet.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductSetGetOperator*

Parameters

- **location** (*str*) – (Required) The region where the ProductSet is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **product_set_id** (*str*) – (Required) The resource id of this ProductSet.
- **project_id** (*str*) – (Optional) The project in which the ProductSet is located. If set to None or missing, the default *project_id* from the GCP connection is used.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_set_id', 'gcp_conn_id']

execute(*self, context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductSetUpdateOperator**(*pro lo- ca- tion proc uct_ proj up- date retr time out= met data gcp_ *arg **k*

Bases: `airflow.models.BaseOperator`

Makes changes to a *ProductSet* resource. Only display_name can be updated currently.

Note: To locate the *ProductSet* resource, its *name* in the form *projects/PROJECT_ID/locations/LOC_ID/productSets/PRODUCT_SET_ID* is necessary.

You can provide the *name* directly as an attribute of the *product_set* object. However, you can leave it blank and provide *location* and *product_set_id* instead (and optionally *project_id* - if not present, the connection default will be used) and the *name* will be created by the operator itself.

This mechanism exists for your convenience, to allow leaving the *project_id* empty and having Airflow use the connection default *project_id*.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductSetUpdateOperator*

Parameters

- **product_set** (*dict or google.cloud.vision_v1.types.ProductSet*) – (Required) The ProductSet resource which replaces the one on the server. If a dict is provided, it must be of the same form as the protobuf message *ProductSet*.
- **location** (*str*) – (Optional) The region where the ProductSet is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **product_set_id** (*str*) – (Optional) The resource id of this ProductSet.
- **project_id** (*str*) – (Optional) The project in which the ProductSet should be created. If set to None or missing, the default project_id from the GCP connection is used.
- **update_mask** (*dict or google.cloud.vision_v1.types.FieldMask*) – (Optional) The *FieldMask* that specifies which fields to update. If update_mask isn't specified, all mutable fields are to be updated. Valid mask path is display_name. If a dict is provided, it must be of the same form as the protobuf message *FieldMask*.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_set_id', 'gcp_conn_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductSetDeleteOperator** (*loc*
pro
*uct_*
proj
retr
tim
out
met
dat
gcp
**ar*
***k*

Bases: *airflow.models.BaseOperator*

Permanently deletes a *ProductSet*. *Products* and *ReferenceImages* in the *ProductSet* are not deleted. The actual image files are not deleted from Google Cloud Storage.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductSetDeleteOperator*

Parameters

- **location** (*str*) – (Required) The region where the ProductSet is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **product_set_id** (*str*) – (Required) The resource id of this ProductSet.
- **project_id** (*str*) – (Optional) The project in which the ProductSet should be created. If set to None or missing, the default project_id from the GCP connection is used.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_set_id', 'gcp_conn_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductCreateOperator**(*location, prod- uct, project_i prod- uct_id=N retry=No time- out=Non meta- data=No gcp_con *args, **kwarg*

Bases: `airflow.models.BaseOperator`

Creates and returns a new product resource.

Possible errors regarding the *Product* object provided:

- Returns *INVALID_ARGUMENT* if *display_name* is missing or longer than 4096 characters.
- Returns *INVALID_ARGUMENT* if *description* is longer than 4096 characters.
- Returns *INVALID_ARGUMENT* if *product_category* is missing or invalid.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductCreateOperator*

Parameters

- **location** (*str*) – (Required) The region where the Product should be created. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **product** (*dict or google.cloud.vision_v1.types.Product*) – (Required) The product to create. If a dict is provided, it must be of the same form as the protobuf message *Product*.
- **project_id** (*str*) – (Optional) The project in which the Product should be created. If set to None or missing, the default project_id from the GCP connection is used.
- **product_id** (*str*) – (Optional) A user-supplied resource id for this Product. If set, the server will attempt to use this value as the resource id. If it is already in use, an error is returned with code ALREADY_EXISTS. Must be at most 128 characters long. It cannot contain the character /.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_id', 'gcp_conn_id']

execute(*self, context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductGetOperator**(*location*, *prod-uct_id*, *project_id=N*, *retry=None*, *time-out=None*, *meta-data=None*, *gcp_conn_id*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Gets information associated with a *Product*.

Possible errors:

- Returns *NOT_FOUND* if the *Product* does not exist.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductGetOperator*

Parameters

- **location** (*str*) – (Required) The region where the Product is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **product_id** (*str*) – (Required) The resource id of this Product.
- **project_id** (*str*) – (Optional) The project in which the Product is located. If set to None or missing, the default project_id from the GCP connection is used.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_id', 'gcp_conn_id']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductUpdateOperator**(*product*, *lo-ca-tion=No* *prod-uct_id=* *project_* *up-date_ma* *retry=N* *time-out=Nor* *meta-data=N* *gcp_con* **args*, ***kwarg*

Bases: `airflow.models.BaseOperator`

Makes changes to a Product resource. Only the display_name, description, and labels fields can be updated right now.

If labels are updated, the change will not be reflected in queries until the next index time.

Note: To locate the *Product* resource, its *name* in the form *projects/PROJECT_ID/locations/LOC_ID/products/PRODUCT_ID* is necessary.

You can provide the *name* directly as an attribute of the *product* object. However, you can leave it blank and provide *location* and *product_id* instead (and optionally *project_id* - if not present, the connection default will be used) and the *name* will be created by the operator itself.

This mechanism exists for your convenience, to allow leaving the *project_id* empty and having Airflow use the connection default *project_id*.

Possible errors related to the provided *Product*:

- Returns *NOT_FOUND* if the Product does not exist.
- **Returns *INVALID_ARGUMENT* if *display_name* is present in update_mask but is missing from the request** or longer than 4096 characters.
- **Returns *INVALID_ARGUMENT* if *description* is present in update_mask but is longer than 4096** characters.
- Returns *INVALID_ARGUMENT* if *product_category* is present in update_mask.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductUpdateOperator*

Parameters

- **product** (*dict or google.cloud.vision_v1.types.ProductSet*) – (Required) The Product resource which replaces the one on the server. product.name is immutable. If a dict is provided, it must be of the same form as the protobuf message *Product*.
- **location** (*str*) – (Optional) The region where the Product is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **product_id** (*str*) – (Optional) The resource id of this Product.

- **project_id** (*str*) – (Optional) The project in which the Product is located. If set to None or missing, the default project_id from the GCP connection is used.
- **update_mask** (*dict or google.cloud.vision_v1.types.FieldMask*) – (Optional) The *FieldMask* that specifies which fields to update. If update_mask isn't specified, all mutable fields are to be updated. Valid mask paths include product_labels, display_name, and description. If a dict is provided, it must be of the same form as the protobuf message *FieldMask*.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_id', 'gcp_conn_id']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionProductDeleteOperator**(*location, prod-uct_id, project_i retry=No time-out=Non meta-data=No gcp_con *args, **kwarg*

Bases: *airflow.models.BaseOperator*

Permanently deletes a product and its reference images.

Metadata of the product and all its images will be deleted right away, but search queries against ProductSets containing the product may still work until all related caches are refreshed.

Possible errors:

- Returns *NOT_FOUND* if the product does not exist.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionProductDeleteOperator*

Parameters

- **location** (*str*) – (Required) The region where the Product is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **product_id** (*str*) – (Required) The resource id of this Product.
- **project_id** (*str*) – (Optional) The project in which the Product is located. If set to None or missing, the default project_id from the GCP connection is used.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.

- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'project_id', 'product_id', 'gcp_conn_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionAnnotateImageOperator** (*request*, *retry=N* *time-* *out=Non* *gcp_con* **args*, ***kwarg*

Bases: *airflow.models.BaseOperator*

Run image detection and annotation for an image or a batch of images.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionAnnotateImageOperator*

Parameters

- **request** (*list[dict or google.cloud.vision_v1.types.AnnotateImageRequest] for batch or dict or google.cloud.vision_v1.types.AnnotateImageRequest for single image.*) – (Required) Annotation request for image or a batch. If a dict is provided, it must be of the same form as the protobuf message class:*google.cloud.vision_v1.types.AnnotateImageRequest*
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['request', 'gcp_conn_id']

execute (*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionReferenceImageCreateOperato**

Bases: *airflow.models.BaseOperator*

Creates and returns a new ReferenceImage ID resource.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionReferenceImageCreateOperator*

Parameters

- **location** (*str*) – (Required) The region where the Product is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **reference_image** (*dict or google.cloud.vision_v1.types.ReferenceImage*) – (Required) The reference image to create. If an image ID is specified, it is ignored. If a dict is provided, it must be of the same form as the protobuf message `google.cloud.vision_v1.types.ReferenceImage`
- **reference_image_id** (*str*) – (Optional) A user-supplied resource id for the ReferenceImage to be added. If set, the server will attempt to use this value as the resource id. If it is already in use, an error is returned with code ALREADY_EXISTS. Must be at most 128 characters long. It cannot contain the character */*.
- **product_id** (*str*) – (Optional) The resource id of this Product.
- **project_id** (*str*) – (Optional) The project in which the Product is located. If set to None or missing, the default project_id from the GCP connection is used.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

template_fields = ['location', 'reference_image', 'product_id', 'reference_image_id',

execute (*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionAddProductToProductSetOperat**

Bases: *airflow.models.BaseOperator*

Adds a Product to the specified ProductSet. If the Product is already present, no change is made.

One Product can be added to at most 100 ProductSets.

Possible errors:

- Returns *NOT_FOUND* if the Product or the ProductSet doesn't exist.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionAddProductToProductSet-Operator*

Parameters

- **product_set_id** (*str*) – (Required) The resource id for the ProductSet to modify.
- **product_id** (*str*) – (Required) The resource id of this Product.
- **location** – (Required) The region where the ProductSet is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **project_id** (*str*) – (Optional) The project in which the Product is located. If set to None or missing, the default project_id from the GCP connection is used.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

Type str

template_fields = ['location', 'product_set_id', 'product_id', 'project_id', 'gcp_conn

execute (*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionRemoveProductFromProductSet**

Bases: *airflow.models.BaseOperator*

Removes a Product from the specified ProductSet.

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionRemoveProductFromProductSetOperator*

Parameters

- **product_set_id** (*str*) – (Required) The resource id for the ProductSet to modify.
- **product_id** (*str*) – (Required) The resource id of this Product.
- **location** – (Required) The region where the ProductSet is located. Valid regions (as of 2019-02-05) are: us-east1, us-west1, europe-west1, asia-east1
- **project_id** (*str*) – (Optional) The project in which the Product is located. If set to None or missing, the default project_id from the GCP connection is used.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]*) – (Optional) Additional metadata that is provided to the method.
- **gcp_conn_id** (*str*) – (Optional) The connection ID used to connect to Google Cloud Platform.

Type str

template_fields = ['location', 'product_set_id', 'product_id', 'project_id', 'gcp_conn

execute (*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionDetectTextOperator**(*image, max_results=retry=None, time-out=None, lan-guage_hints=web_detectio ad-di-tional_prope gcp_conn_id *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Detects Text in the image

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionDetectTextOperator*

Parameters

- **image** (*dict or google.cloud.vision_v1.types.Image*) – (Required) The image to analyze. See more: https://googleapis.github.io/google-cloud-python/latest/vision/gapic/v1/types.html#google.cloud.vision_v1.types.Image
- **max_results** (*int*) – (Optional) Number of results to return.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – Number of seconds before timing out.
- **language_hints** (*str, list or google.cloud.vision.v1.ImageContext.language_hints:*) – List of languages to use for TEXT_DETECTION. In most cases, an empty value yields the best results since it enables automatic language detection. For languages based on the Latin alphabet, setting language_hints is not needed.
- **web_detection_params** (*dict or google.cloud.vision.v1.ImageContext.web_detection_params*) – Parameters for web detection.
- **additional_properties** (*dict*) – Additional properties to be set on the AnnotateImageRequest. See more: google.cloud.vision_v1.types.AnnotateImageRequest

template_fields = ['image', 'max_results', 'timeout', 'gcp_conn_id']

execute(*self, context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionDetectDocumentTextOperator**(*i*
r
r
t
c
l
g
v
a
a
t
g
***
***

Bases: *airflow.models.BaseOperator*

Detects Document Text in the image

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionDetectDocumentTextOperator*

Parameters

- **image** (*dict or google.cloud.vision_v1.types.Image*) – (Required) The image to analyze. See more: https://googleapis.github.io/google-cloud-python/latest/vision/gapic/v1/types.html#google.cloud.vision_v1.types.Image
- **max_results** (*int*) – Number of results to return.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – Number of seconds before timing out.
- **language_hints** (*str, list or google.cloud.vision.v1.ImageContext.language_hints:*) – List of languages to use for TEXT_DETECTION. In most cases, an empty value yields the best results since it enables automatic language detection. For languages based on the Latin alphabet, setting language_hints is not needed.
- **web_detection_params** (*dict or google.cloud.vision.v1.ImageContext.web_detection_params*) – Parameters for web detection.
- **additional_properties** (*dict*) – Additional properties to be set on the AnnotateImageRequest. See more: https://googleapis.github.io/google-cloud-python/latest/vision/gapic/v1/types.html#google.cloud.vision_v1.types.AnnotateImageRequest

template_fields = ['image', 'max_results', 'timeout', 'gcp_conn_id']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionDetectImageLabelsOperator**(*im*
m
re
tin
ou
ad
di
tio
gc
**a*
****

Bases: `airflow.models.BaseOperator`

Detects Document Text in the image

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionDetectImageLabelsOperator*

Parameters

- **image** (*dict or google.cloud.vision_v1.types.Image*) – (Required) The image to analyze. See more: https://googleapis.github.io/google-cloud-python/latest/vision/gapic/v1/types.html#google.cloud.vision_v1.types.Image
- **max_results** (*int*) – Number of results to return.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – Number of seconds before timing out.
- **additional_properties** (*dict*) – Additional properties to be set on the AnnotateImageRequest. See more: https://googleapis.github.io/google-cloud-python/latest/vision/gapic/v1/types.html#google.cloud.vision_v1.types.AnnotateImageRequest

template_fields = ['image', 'max_results', 'timeout', 'gcp_conn_id']

execute(*self*, *context*)

class airflow.contrib.operators.gcp_vision_operator.**CloudVisionDetectImageSafeSearchOperat**

Bases: `airflow.models.BaseOperator`

Detects Document Text in the image

See also:

For more information on how to use this operator, take a look at the guide: *CloudVisionDetectImageSafeSearchOperator*

Parameters

- **image** (*dict or google.cloud.vision_v1.types.Image*) – (Required) The image to analyze. See more: https://googleapis.github.io/google-cloud-python/latest/vision/gapic/v1/types.html#google.cloud.vision_v1.types.Image
- **max_results** (*int*) – Number of results to return.
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If *None* is specified, requests will not be retried.
- **timeout** (*float*) – Number of seconds before timing out.
- **additional_properties** (*dict*) – Additional properties to be set on the AnnotateImageRequest. See more: https://googleapis.github.io/google-cloud-python/latest/vision/gapic/v1/types.html#google.cloud.vision_v1.types.AnnotateImageRequest

template_fields = ['image', 'max_results', 'timeout', 'gcp_conn_id']

execute(*self*, *context*)

airflow.contrib.operators.gcp_vision_operator.**prepare_additional_parameters**(*additional_properties*, *language_hints*, *web_detection_param*

Creates additional_properties parameter based on language_hints, web_detection_params and additional_properties parameters specified by the user

airflow.contrib.operators.gcs_acl_operator

This module contains Google Cloud Storage ACL entry operator.

Module Contents

class airflow.contrib.operators.gcs_acl_operator.**GoogleCloudStorageBucketCreateAclEntryOper**

Bases: *airflow.models.BaseOperator*

Creates a new ACL entry on the specified bucket.

See also:

For more information on how to use this operator, take a look at the guide: *GoogleCloudStorageBucketCreateAclEntryOperator*

Parameters

- **bucket** (*str*) – Name of a bucket.
- **entity** (*str*) – The entity holding the permission, in one of the following forms: user-userId, user-email, group-groupId, group-email, domain-domain, project-team-projectId, allUsers, allAuthenticatedUsers
- **role** (*str*) – The access permission for the entity. Acceptable values are: “OWNER”, “READER”, “WRITER”.

- **user_project** (*str*) – (Optional) The project to be billed for this request. Required for Requester Pays buckets.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google Cloud Storage.

template_fields = ['bucket', 'entity', 'role', 'user_project']

execute(*self*, *context*)

class airflow.contrib.operators.gcs_acl_operator.**GoogleCloudStorageObjectCreateAclEntryOper**

Bases: *airflow.models.BaseOperator*

Creates a new ACL entry on the specified object.

See also:

For more information on how to use this operator, take a look at the guide: *GoogleCloudStorageObjectCreateAclEntryOperator*

Parameters

- **bucket** (*str*) – Name of a bucket.
- **object_name** (*str*) – Name of the object. For information about how to URL encode object names to be path safe, see: https://cloud.google.com/storage/docs/json_api/#encoding
- **entity** (*str*) – The entity holding the permission, in one of the following forms: user-userId, user-email, group-groupId, group-email, domain-domain, project-team-projectId, allUsers, allAuthenticatedUsers
- **role** (*str*) – The access permission for the entity. Acceptable values are: "OWNER", "READER".
- **user_project** (*str*) – (Optional) The project to be billed for this request. Required for Requester Pays buckets.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google Cloud Storage.

template_fields = ['bucket', 'object_name', 'entity', 'role', 'generation', 'user_proj

execute(*self*, *context*)

airflow.contrib.operators.gcs_delete_operator

This module contains Google Cloud Storage delete operator.

Module Contents

class airflow.contrib.operators.gcs_delete_operator.**GoogleCloudStorageDeleteOperator**(*bucket_r ob- jects:Opt pre- fix:Optio google_c del- e- gate_to:C *args, **kwarg*

Bases: *airflow.models.BaseOperator*

Deletes objects from a Google Cloud Storage bucket, either from an explicit list of object names or all objects matching a prefix.

Parameters

- **bucket_name** (*str*) – The GCS bucket to delete from
- **objects** (*Iterable[str]*) – List of objects to delete. These should be the names of objects in the bucket, not including gs://bucket/
- **prefix** – Prefix of objects to delete. All objects matching this prefix in the bucket will be deleted.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use for Google Cloud Storage
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['bucket_name', 'prefix', 'objects']

execute(*self*, *context*)

airflow.contrib.operators.gcs_download_operator

This module contains Google Cloud Storage download operator.

Module Contents

class airflow.contrib.operators.gcs_download_operator.**GoogleCloudStorageDownloadOperator**(*bu ob je fil na sto go de e- ga *a ***

Bases: *airflow.models.BaseOperator*

Downloads a file from Google Cloud Storage.

Parameters

- **bucket** (*str*) – The Google cloud storage bucket where the object is. (templated)
- **object** (*str*) – The name of the object to download in the Google cloud storage bucket. (templated)
- **filename** (*str*) – The file path on the local file system (where the operator is being executed) that the file should be downloaded to. (templated) If no filename passed, the downloaded data will not be stored on the local file system.
- **store_to_xcom_key** (*str*) – If this param is set, the operator will push the contents of the downloaded file to XCom with the key set in this parameter. If not set, the downloaded data will not be pushed to XCom. (templated)
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['bucket', 'object', 'filename', 'store_to_xcom_key']

ui_color = #f0eee4

execute(*self*, *context*)

airflow.contrib.operators.gcs_list_operator

This module contains a Google Cloud Storage list operator.

Module Contents

class airflow.contrib.operators.gcs_list_operator.**GoogleCloudStorageListOperator**(*bucket*, *prefix=None*, *delimiter=None*, *google_cloud_*
delegate_to=None, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

List all objects from the bucket with the give string prefix and delimiter in name.

This operator returns a python list with the name of objects which can be used by *xcom* in the downstream task.

Parameters

- **bucket** (*str*) – The Google cloud storage bucket to find the objects. (templated)
- **prefix** (*str*) – Prefix string which filters objects whose name begin with this prefix. (templated)

- **delimiter** (*str*) – The delimiter by which you want to filter the objects. (templated) For e.g to lists the CSV files from in a directory in GCS you would use delimiter='.csv'.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example: The following Operator would list all the Avro files from `sales/sales-2017` folder in `data` bucket.

```
GCS_Files = GoogleCloudStorageListOperator(
    task_id='GCS_Files',
    bucket='data',
    prefix='sales/sales-2017/',
    delimiter='.avro',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

template_fields :Iterable[str] = ['bucket', 'prefix', 'delimiter']

ui_color = #f0eee4

execute (*self*, *context*)

airflow.contrib.operators.gcs_operator

This module contains a Google Cloud Storage Bucket operator.

Module Contents

class airflow.contrib.operators.gcs_operator.**GoogleCloudStorageCreateBucketOperator** (*bucket_na re-source=N stor-age_class lo-ca-tion='US', project_id la-bels=Non google_cl del-e-gate_to=N *args, **kwargs*

Bases: *airflow.models.BaseOperator*

Creates a new bucket. Google Cloud Storage uses a flat namespace, so you can't create a bucket with a name that is already in use.

See also:

For more information, see Bucket Naming Guidelines: https://cloud.google.com/storage/docs/bucketnaming.html#requirements

Parameters

- **bucket_name** (*str*) – The name of the bucket. (templated)
- **resource** (*dict*) – An optional dict with parameters for creating the bucket. For information on available parameters, see Cloud Storage API doc: https://cloud.google.com/storage/docs/json_api/v1/buckets/insert
- **storage_class** (*str*) – This defines how objects in the bucket are stored and determines the SLA and the cost of storage (templated). Values include
  - `MULTI_REGIONAL`
  - `REGIONAL`
  - `STANDARD`
  - `NEARLINE`
  - `COLDLINE`.

  If this value is not specified when the bucket is created, it will default to STANDARD.
- **location** (*str*) – The location of the bucket. (templated) Object data for objects in the bucket resides in physical storage within this region. Defaults to US.

  See also:

  https://developers.google.com/storage/docs/bucket-locations
- **project_id** (*str*) – The ID of the GCP Project. (templated)
- **labels** (*dict*) – User-provided labels, in key/value pairs.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

:Example:: The following Operator would create a new bucket `test-bucket` with `MULTI_REGIONAL` storage class in `EU` region

```
CreateBucket = GoogleCloudStorageCreateBucketOperator(
    task_id='CreateNewBucket',
    bucket_name='test-bucket',
    storage_class='MULTI_REGIONAL',
    location='EU',
    labels={'env': 'dev', 'team': 'airflow'},
    google_cloud_storage_conn_id='airflow-service-account'
)
```

template_fields = ['bucket_name', 'storage_class', 'location', 'project_id']

ui_color = #f0eee4

execute (*self*, *context*)

airflow.contrib.operators.gcs_to_bq

This module contains a Google Cloud Storage to BigQuery operator.

Module Contents

class airflow.contrib.operators.gcs_to_bq.**GoogleCloudStorageToBigQueryOperator**(*bucket*, *source_objects*, *des- ti- na- tion_project_data* *schema_fields=N* *schema_object=N* *source_format='C* *com- pres- sion='NONE'*, *cre- ate_disposition='C* *skip_leading_row* *write_disposition=* *field_delimiter=',* *'* *max_bad_records* *quote_character=* *ig- nore_unknown_v* *al- low_quoted_newl* *al- low_jagged_rows* *max_id_key=Non* *big- query_conn_id='g* *google_cloud_stor* *del- e- gate_to=None*, *schema_update_o* *src_fmt_configs=* *ex- ter- nal_table=False*, *time_partitioning=* *clus- ter_fields=None*, *au- tode- tect=False*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Loads files from Google cloud storage into BigQuery.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it.

See also:

For more information on how to use this operator, take a look at the guide: *GoogleCloudStorageToBigQueryOperator*

Parameters

- **bucket** (*str*) – The bucket to load from. (templated)
- **source_objects** (*list[str]*) – List of Google cloud storage URIs to load from. (templated) If source_format is 'DATASTORE_BACKUP', the list must only contain a single URI.
- **destination_project_dataset_table** (*str*) – The dotted `(<project>.|<project>:)<dataset>.<table>` BigQuery table to load data into. If `<project>` is not included, project will be the project defined in the connection json. (templated)
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load Should not be set when source_format is 'DATASTORE_BACKUP'. Parameter must be defined if 'schema_object' is null and autodetect is False.
- **schema_object** (*str*) – If set, a GCS object path pointing to a .json file that contains the schema for the table. (templated) Parameter must be defined if 'schema_fields' is null and autodetect is False.
- **source_format** (*str*) – File format to export.
- **compression** (*str*) – [Optional] The compression type of the data source. Possible values include GZIP and NONE. The default value is NONE. This setting is ignored for Google Cloud Bigtable, Google Cloud Datastore backups and Avro formats.
- **create_disposition** (*str*) – The create disposition if the table doesn't exist.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **write_disposition** (*str*) – The write disposition if the table already exists.
- **field_delimiter** (*str*) – The delimiter to use when loading from a CSV.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **quote_character** (*str*) – The value that is used to quote data sections in a CSV file.
- **ignore_unknown_values** (*bool*) – [Optional] Indicates if BigQuery should allow extra values that are not represented in the table schema. If true, the extra values are ignored. If false, records with extra columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result.
- **allow_quoted_newlines** (*bool*) – Whether to allow quoted newlines (true) or not (false).
- **allow_jagged_rows** (*bool*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable to CSV, ignored for other formats.
- **max_id_key** (*str*) – If set, the name of a column in the BigQuery table that's to be loaded. This will be used to select the MAX value from BigQuery after the load occurs. The results will be returned by the execute() command, which in turn gets stored in XCom for future operators to use. This can be helpful with incremental loads–during future executions, you can pick up from the max ID.
- **bigquery_conn_id** (*str*) – Reference to a specific BigQuery hook.

- **google_cloud_storage_conn_id** (*str*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **schema_update_options** (*list*) – Allows the schema of the destination table to be updated as a side effect of the load job.
- **src_fmt_configs** (*dict*) – configure optional fields specific to the source format
- **external_table** (*bool*) – Flag to specify if the destination table should be a BigQuery external table. Default Value is False.
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications. Note that 'field' is not available in concurrency with dataset.table$partition.
- **cluster_fields** (*list[str]*) – Request that the result of this load be stored sorted by one or more columns. This is only available in conjunction with time_partitioning. The order of columns given determines the sort order. Not applicable for external tables.
- **autodetect** (*bool*) – [Optional] Indicates if we should automatically infer the options and schema for CSV and JSON sources. (Default: `False`). Parameter must be setted to True if 'schema_fields' and 'schema_object' are undefined. It is suggested to set to True if table are create outside of Airflow.

template_fields = ['bucket', 'source_objects', 'schema_object', 'destination_project_d

template_ext = ['.sql']

ui_color = #f0eee4

execute (*self*, *context*)

airflow.contrib.operators.gcs_to_gcs

This module contains a Google Cloud Storage operator.

Module Contents

`airflow.contrib.operators.gcs_to_gcs.`**WILDCARD = ***

class airflow.contrib.operators.gcs_to_gcs.**GoogleCloudStorageToGoogleCloudStorageOperator**(

Bases: *airflow.models.BaseOperator*

Copies objects from a bucket to another, with renaming if requested.

Parameters

- **source_bucket** (*str*) – The source Google cloud storage bucket where the object is. (templated)
- **source_object** (*str*) – The source name of the object to copy in the Google cloud storage bucket. (templated) You can use only one wildcard for objects (filenames) within your bucket. The wildcard can appear inside the object name or at the end of the object name. Appending a wildcard to the bucket name is unsupported.
- **destination_bucket** (*str*) – The destination Google cloud storage bucket where the object should be. If the destination_bucket is None, it defaults to source_bucket. (templated)
- **destination_object** (*str*) – The destination name of the object in the destination Google cloud storage bucket. (templated) If a wildcard is supplied in the source_object argument, this is the prefix that will be prepended to the final destination objects' paths. Note that the source path's part before the wildcard will be removed; if it needs to be retained it should be appended to destination_object. For example, with prefix `foo/*` and destination_object `blah/`, the file `foo/baz` will be copied to `blah/baz`; to retain the prefix write the destination_object as e.g. `blah/foo`, in which case the copied file will be named `blah/foo/baz`.
- **move_object** (*bool*) – When move object is True, the object is moved instead of copied to the new location. This is the equivalent of a mv command as opposed to a cp command.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **last_modified_time** (*datetime.datetime*) – When specified, the objects will be copied or moved, only if they were modified after last_modified_time. If tzinfo has not been set, UTC will be assumed.

Example

The following Operator would copy a single file named `sales/sales-2017/january.avro` in the `data` bucket to the file named `copied_sales/2017/january-backup.avro` in the `data_backup` bucket

```
copy_single_file = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='copy_single_file',
    source_bucket='data',
    source_object='sales/sales-2017/january.avro',
    destination_bucket='data_backup',
    destination_object='copied_sales/2017/january-backup.avro',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

The following Operator would copy all the Avro files from `sales/sales-2017` folder (i.e. with names starting with that prefix) in `data` bucket to the `copied_sales/2017` folder in the `data_backup` bucket.

```
copy_files = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='copy_files',
    source_bucket='data',
    source_object='sales/sales-2017/*.avro',
    destination_bucket='data_backup',
    destination_object='copied_sales/2017/',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

The following Operator would move all the Avro files from `sales/sales-2017` folder (i.e. with names starting with that prefix) in `data` bucket to the same folder in the `data_backup` bucket, deleting the original files in the process.

```
move_files = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='move_files',
    source_bucket='data',
    source_object='sales/sales-2017/*.avro',
    destination_bucket='data_backup',
    move_object=True,
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

template_fields = ['source_bucket', 'source_object', 'destination_bucket', 'destinatio

ui_color = #f0eee4

execute (*self*, *context*)

_copy_single_object (*self*, *hook*, *source_object*, *destination_object*)

airflow.contrib.operators.gcs_to_s3

This module contains Google Cloud Storage to S3 operator.

Module Contents

class airflow.contrib.operators.gcs_to_s3.**GoogleCloudStorageToS3Operator**(*bucket, prefix=None, delimiter=None, google_cloud_storage_conn delegate_to=None, dest_aws_conn_id=None, dest_s3_key=None, dest_verify=None, replace=False, *args, **kwargs*)

Bases: *airflow.contrib.operators.gcs_list_operator.GoogleCloudStorageListOperator*

Synchronizes a Google Cloud Storage bucket with an S3 bucket.

Parameters

- **bucket** (*str*) – The Google Cloud Storage bucket to find the objects. (templated)
- **prefix** (*str*) – Prefix string which filters objects whose name begin with this prefix. (templated)
- **delimiter** (*str*) – The delimiter by which you want to filter the objects. (templated) For e.g to lists the CSV files from in a directory in GCS you would use delimiter='.csv'.
- **google_cloud_storage_conn_id** (*str*) – The connection ID to use when connecting to Google Cloud Storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **dest_aws_conn_id** (*str*) – The destination S3 connection
- **dest_s3_key** (*str*) – The base S3 key to be used to store the files. (templated)
- **dest_verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **False: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.
- **replace** (*bool*) – Whether or not to verify the existence of the files in the destination bucket. By default is set to False If set to True, will upload all the files replacing the existing ones in the destination bucket. If set to False, will upload only the files that are in the origin but not in the destination bucket.

template_fields = ['bucket', 'prefix', 'delimiter', 'dest_s3_key']

ui_color = #f0eee4

execute(*self, context*)

airflow.contrib.operators.grpc_operator

Module Contents

class airflow.contrib.operators.grpc_operator.**GrpcOperator**(*stub_class*, *call_func*, *grpc_conn_id='grpc_default'*, *data=None*, *interceptors=None*, *custom_connection_func=None*, *streaming=False*, *response_callback=None*, *log_response=False*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Calls a gRPC endpoint to execute an action

Parameters

- **stub_class** (*gRPC stub class generated from proto file*) – The stub client to use for this gRPC call
- **call_func** (*gRPC client function name for the endpoint generated from proto file, str*) – The client function name to call the gRPC endpoint
- **grpc_conn_id** (*str*) – The connection to run the operator against
- **data** (*A dict with key value pairs as kwargs of the call_func*) – The data to pass to the rpc call
- **interceptors** (*A list of gRPC interceptor objects, has to be initialized*) – A list of gRPC interceptor objects to be used on the channel
- **custom_connection_func** (*A python function that returns channel object, take in a connection object, can be a partial function*) – The customized connection function to return channel object
- **streaming** (*boolean*) – A flag to indicate if the call is a streaming call
- **response_callback** (*A python function that process the response from gRPC call, takes in response object and context object, context object can be used to perform push xcom or other after task actions*) – The callback function to process the response from gRPC call
- **log_response** (*boolean*) – A flag to indicate if we need to log the response

template_fields = ['stub_class', 'call_func', 'data']

_get_grpc_hook(*self*)

execute(*self*, *context*)

_handle_response(*self*, *response*, *context*)

airflow.contrib.operators.hipchat_operator

Module Contents

class airflow.contrib.operators.hipchat_operator.**HipChatAPIOperator**(*token*, *base_url='https://api.hipchat.com/* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Base HipChat Operator. All derived HipChat operators reference from HipChat's official REST API documentation at https://www.hipchat.com/docs/apiv2. Before using any HipChat API operators you need to get an authentication token at https://www.hipchat.com/docs/apiv2/auth. In the future additional HipChat operators will be derived from this class as well.

Parameters

- **token** (*str*) – HipChat REST API authentication token
- **base_url** (*str*) – HipChat REST API base url.

prepare_request(*self*)

Used by the execute function. Set the request method, url, and body of HipChat's REST API call. Override in child class. Each HipChatAPI child operator is responsible for having a prepare_request method call which sets self.method, self.url, and self.body.

execute(*self*, *context*)

class airflow.contrib.operators.hipchat_operator.**HipChatAPISendRoomNotificationOperator**(*roo* *mes* *sag* *mes* *sag* *colo* *frm* *at-* *tach* *no-* *tify* *carc* **ar* ***k*

Bases: *airflow.contrib.operators.hipchat_operator.HipChatAPIOperator*

Send notification to a specific HipChat room. More info: https://www.hipchat.com/docs/apiv2/method/send_room_notification

Parameters

- **room_id** (*str*) – Room in which to send notification on HipChat. (templated)
- **message** (*str*) – The message body. (templated)
- **frm** (*str*) – Label to be shown in addition to sender's name
- **message_format** (*str*) – How the notification is rendered: html or text
- **color** (*str*) – Background color of the msg: yellow, green, red, purple, gray, or random
- **attach_to** (*str*) – The message id to attach this notification to
- **notify** (*bool*) – Whether this message should trigger a user notification
- **card** (*dict*) – HipChat-defined card object

template_fields = ['token', 'room_id', 'message', 'message_format', 'color', 'frm', 'a

ui_color = #2980b9

prepare_request (*self*)

airflow.contrib.operators.hive_to_dynamodb

Module Contents

class airflow.contrib.operators.hive_to_dynamodb.**HiveToDynamoDBTransferOperator** (*sql, table_name, table_keys, pre_process=No pre_process_arg pre_process_kw region_name=Nor schema='defaul hiveserver2_con aws_conn_id='a *args, **kwargs*)

Bases: `airflow.models.BaseOperator`

Moves data from Hive to DynamoDB, note that for now the data is loaded into memory before being pushed to DynamoDB, so this operator should be used for smallish amount of data.

Parameters

- **sql** (*str*) – SQL query to execute against the hive database. (templated)
- **table_name** (*str*) – target DynamoDB table
- **table_keys** (*list*) – partition key and sort key
- **pre_process** (*function*) – implement pre-processing of source data
- **pre_process_args** (*list*) – list of pre_process function arguments
- **pre_process_kwargs** (*dict*) – dict of pre_process function arguments
- **region_name** (*str*) – aws region name (example: us-east-1)
- **schema** (*str*) – hive database schema
- **hiveserver2_conn_id** (*str*) – source hive connection
- **aws_conn_id** (*str*) – aws connection

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #a0e08c

execute (*self, context*)

airflow.contrib.operators.imap_attachment_to_s3_operator

Module Contents

class airflow.contrib.operators.imap_attachment_to_s3_operator.**ImapAttachmentToS3Operator**(*i
s
i
i
s
i
s
*
**

Bases: *airflow.models.BaseOperator*

Transfers a mail attachment from a mail server into s3 bucket.

Parameters

- **imap_attachment_name** (*str*) – The file name of the mail attachment that you want to transfer.
- **s3_key** (*str*) – The destination file name in the s3 bucket for the attachment.
- **imap_mail_folder** (*str*) – The folder on the mail server to look for the attachment.
- **imap_check_regex** (*bool*) – If set checks the *imap_attachment_name* for a regular expression.
- **s3_overwrite** (*bool*) – If set overwrites the s3 key if already exists.
- **imap_conn_id** (*str*) – The reference to the connection details of the mail server.
- **s3_conn_id** (*str*) – The reference to the s3 connection details.

template_fields = ['imap_attachment_name', 's3_key']

execute(*self*, *context*)

This function executes the transfer from the email server (via imap) into s3.

Parameters **context** (*dict*) – The context while executing.

airflow.contrib.operators.jenkins_job_trigger_operator

Module Contents

airflow.contrib.operators.jenkins_job_trigger_operator.**jenkins_request_with_headers**(*jenkins_se
req*)

We need to get the headers in addition to the body answer to get the location from them This function uses jenkins_request method from python-jenkins library with just the return call changed

Parameters

- **jenkins_server** – The server to query
- **req** – The request to execute

Returns Dict containing the response body (key body) and the headers coming along (headers)

class airflow.contrib.operators.jenkins_job_trigger_operator.**JenkinsJobTriggerOperator**(*jenki job_n pa- ram- e- ters= sleep_ max_ *args **kw*

Bases: `airflow.models.BaseOperator`

Trigger a Jenkins Job and monitor it's execution. This operator depend on python-jenkins library, version >= 0.4.15 to communicate with jenkins server. You'll also need to configure a Jenkins connection in the connections screen.

Parameters

- **jenkins_connection_id** (*str*) – The jenkins connection to use for this job
- **job_name** (*str*) – The name of the job to trigger
- **parameters** (*str*) – The parameters block to provide to jenkins. (templated)
- **sleep_time** (*int*) – How long will the operator sleep between each status request for the job (min 1, default 10)
- **max_try_before_job_appears** (*int*) – The maximum number of requests to make while waiting for the job to appears on jenkins server (default 10)

template_fields = ['parameters']

template_ext = ['.json']

ui_color = #f9ec86

build_job(*self*, *jenkins_server*)

This function makes an API call to Jenkins to trigger a build for 'job_name' It returned a dict with 2 keys : body and headers. headers contains also a dict-like object which can be queried to get the location to poll in the queue.

Parameters **jenkins_server** – The jenkins server where the job should be triggered

Returns Dict containing the response body (key body) and the headers coming along (headers)

poll_job_in_queue(*self*, *location*, *jenkins_server*)

This method poll the jenkins queue until the job is executed. When we trigger a job through an API call, the job is first put in the queue without having a build number assigned. Thus we have to wait the job exit the queue to know its build number. To do so, we have to add /api/json (or /api/xml) to the location returned by the build_job call and poll this file. When a 'executable' block appears in the json, it means the job execution started and the field 'number' then contains the build number.

Parameters

- **location** – Location to poll, returned in the header of the build_job call
- **jenkins_server** – The jenkins server to poll

Returns The build_number corresponding to the triggered job

get_hook(*self*)

execute(*self*, *context*)

`airflow.contrib.operators.jira_operator`

Module Contents

class `airflow.contrib.operators.jira_operator.`**`JiraOperator`**(*jira_conn_id='jira_default', jira_method=None, jira_method_args=None, result_processor=None, get_jira_resource_method=None, *args, **kwargs*)

Bases: *`airflow.models.BaseOperator`*

JiraOperator to interact and perform action on Jira issue tracking system. This operator is designed to use Jira Python SDK: http://jira.readthedocs.io

Parameters

- **`jira_conn_id`** (*`str`*) – reference to a pre-defined Jira Connection
- **`jira_method`** (*`str`*) – method name from Jira Python SDK to be called
- **`jira_method_args`** (*`dict`*) – required method parameters for the jira_method. (templated)
- **`result_processor`** (*`function`*) – function to further process the response from Jira
- **`get_jira_resource_method`** (*`function`*) – function or operator to get jira resource on which the provided jira_method will be executed

`template_fields = ['jira_method_args']`

`execute`(*self, context*)

airflow.contrib.operators.kubernetes_pod_operator

Module Contents

class airflow.contrib.operators.kubernetes_pod_operator.**KubernetesPodOperator**(*namespace, image, name, cmds=None, arguments=None, ports=None, volume_mounts=None volumes=None, env_vars=None, secrets=None, in_cluster=False, cluster_context=None, labels=None, startup_timeout_sec get_logs=True, image_pull_policy='If annotations=None, resources=None, affinity=None, config_file=None, do_xcom_push=Fa node_selectors=No image_pull_secrets=N service_account_name is_delete_operator_ hostnetwork=False, tolerations=None, configmaps=None, security_context=None, pod_runtime_info_ *args*

Execute a task in a Kubernetes Pod

Parameters

- **image** (*str*) – Docker image you wish to launch. Defaults to dockerhub.io, but fully qualified URLS will point to custom repositories
- **namespace** (*str*) – the namespace to run within kubernetes
- **cmds** (*list[str]*) – entrypoint of the container. (templated) The docker images's entrypoint is used if this is not provide.
- **arguments** (*list[str]*) – arguments of the entrypoint. (templated) The docker image's CMD is used if this is not provided.
- **image_pull_policy** (*str*) – Specify a policy to cache or always pull an image
- **image_pull_secrets** (*str*) – Any image pull secrets to be given to the pod. If more than one secret is required, provide a comma separated list: secret_a,secret_b
- **ports** (*list[airflow.kubernetes.pod.Port]*) – ports for launched pod
- **volume_mounts** (*list[airflow.contrib.kubernetes.volume_mount.VolumeMount]*) – volumeMounts for launched pod
- **volumes** (*list[airflow.contrib.kubernetes.volume.Volume]*) – volumes for launched pod. Includes ConfigMaps and PersistentVolumes
- **labels** (*dict*) – labels to apply to the Pod
- **startup_timeout_seconds** (*int*) – timeout in seconds to startup the pod
- **name** (*str*) – name of the task you want to run, will be used to generate a pod id
- **env_vars** (*dict*) – Environment variables initialized in the container. (templated)
- **secrets** (*list[airflow.contrib.kubernetes.secret.Secret]*) – Kubernetes secrets to inject in the container, They can be exposed as environment vars or files in a volume.
- **in_cluster** (*bool*) – run kubernetes client with in_cluster configuration
- **cluster_context** (*str*) – context that points to kubernetes cluster. Ignored when in_cluster is True. If None, current-context is used.
- **get_logs** (*bool*) – get the stdout of the container as logs of the tasks
- **resources** (*dict*) – A dict containing a group of resources requests and limits
- **affinity** (*dict*) – A dict containing a group of affinity scheduling rules
- **node_selectors** (*dict*) – A dict containing a group of scheduling rules
- **config_file** (*str*) – The path to the Kubernetes config file. If not specified, default value is `~/.kube/config`
- **do_xcom_push** (*bool*) – If True, the content of the file /airflow/xcom/return.json in the container will also be pushed to an XCom when the container completes.
- **is_delete_operator_pod** (*bool*) – What to do when the pod reaches its final state, or the execution is interrupted. If False (default): do nothing, If True: delete the pod
- **hostnetwork** (*bool*) – If True enable host networking on the pod
- **tolerations** (*list tolerations*) – A list of kubernetes tolerations
- **configmaps** (*list[str]*) – A list of configmap names objects that we want mount as env variables

- **pod_runtime_info_envs** (*list[PodRuntimeEnv]*) – environment variables about pod runtime information (ip, namespace, nodeName, podName)

template_fields = ['cmds', 'arguments', 'env_vars', 'config_file']

execute(*self*, *context*)

_set_resources(*self*, *resources*)

airflow.contrib.operators.mlengine_operator

Module Contents

airflow.contrib.operators.mlengine_operator.**log**

airflow.contrib.operators.mlengine_operator.**_normalize_mlengine_job_id**(*job_id*)
Replaces invalid MLEngine job_id characters with '_'.

This also adds a leading 'z' in case job_id starts with an invalid character.

Args: job_id: A job_id str that may have invalid characters.

Returns: A valid job_id representation.

class airflow.contrib.operators.mlengine_operator.**MLEngineBatchPredictionOperator**(*project_id, job_id, re-gion, data_format, in-put_paths, out-put_path, model_name ver-sion_name=I uri=None, max_worker run-time_version sig-na-ture_name=I gcp_conn_id del-e-gate_to=Non *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Google Cloud ML Engine prediction job.

NOTE: For model origin, users should consider exactly one from the three options below:

1. Populate `uri` field only, which should be a GCS location that points to a tensorflow savedModel directory.
2. Populate `model_name` field only, which refers to an existing model, and the default version of the model will be used.

3. Populate both `model_name` and `version_name` fields, which refers to a specific version of a specific model.

In options 2 and 3, both model and version name should contain the minimal identifier. For instance, call:

```
MLEngineBatchPredictionOperator(
    ...,
    model_name='my_model',
    version_name='my_version',
    ...)
```

if the desired model version is `projects/my_project/models/my_model/versions/my_version`.

See https://cloud.google.com/ml-engine/reference/rest/v1/projects.jobs for further documentation on the parameters.

Parameters

- **project_id** (*str*) – The Google Cloud project name where the prediction job is submitted. (templated)
- **job_id** (*str*) – A unique id for the prediction job on Google Cloud ML Engine. (templated)
- **data_format** (*str*) – The format of the input data. It will default to 'DATA_FORMAT_UNSPECIFIED' if is not provided or is not one of ["TEXT", "TF_RECORD", "TF_RECORD_GZIP"].
- **input_paths** (*list[str]*) – A list of GCS paths of input data for batch prediction. Accepting wildcard operator `*`, but only at the end. (templated)
- **output_path** (*str*) – The GCS path where the prediction results are written to. (templated)
- **region** (*str*) – The Google Compute Engine region to run the prediction job in. (templated)
- **model_name** (*str*) – The Google Cloud ML Engine model to use for prediction. If version_name is not provided, the default version of this model will be used. Should not be None if version_name is provided. Should be None if uri is provided. (templated)
- **version_name** (*str*) – The Google Cloud ML Engine model version to use for prediction. Should be None if uri is provided. (templated)
- **uri** (*str*) – The GCS path of the saved model to use for prediction. Should be None if model_name is provided. It should be a GCS path pointing to a tensorflow SavedModel. (templated)
- **max_worker_count** (*int*) – The maximum number of workers to be used for parallel processing. Defaults to 10 if not specified.
- **runtime_version** (*str*) – The Google Cloud ML Engine runtime version to use for batch prediction.
- **signature_name** (*str*) – The name of the signature defined in the SavedModel to use for this job.
- **gcp_conn_id** (*str*) – The connection ID used for connection to Google Cloud Platform.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Raises `ValueError`: if a unique model/version origin cannot be determined.

template_fields = ['_project_id', '_job_id', '_region', '_input_paths', '_output_path'

execute (*self*, *context*)

class airflow.contrib.operators.mlengine_operator.**MLEngineModelOperator**(*project_id*, *model*, *op- er- a- tion='create'*, *gcp_conn_id='google_cloud_* *del- e- gate_to=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Operator for managing a Google Cloud ML Engine model.

Parameters

- **project_id** (*str*) – The Google Cloud project name to which MLEngine model belongs. (templated)
- **model** (*dict*) – A dictionary containing the information about the model. If the *operation* is *create*, then the *model* parameter should contain all the information about this model such as *name*.

  If the *operation* is *get*, the *model* parameter should contain the *name* of the model.
- **operation** (*str*) – The operation to perform. Available operations are:
  - `create`: Creates a new model as provided by the *model* parameter.
  - `get`: Gets a particular model where the name is specified in *model*.
- **gcp_conn_id** (*str*) – The connection ID to use when fetching connection info.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['_model']

execute(*self*, *context*)

class airflow.contrib.operators.mlengine_operator.**MLEngineVersionOperator**(*project_id*, *model_name*, *ver- sion_name=None*, *ver- sion=None*, *op- er- a- tion='create'*, *gcp_conn_id='google_clo* *del- e- gate_to=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Operator for managing a Google Cloud ML Engine version.

Parameters

- **project_id** (*str*) – The Google Cloud project name to which MLEngine model belongs.
- **model_name** (*str*) – The name of the Google Cloud ML Engine model that the version belongs to. (templated)
- **version_name** (*str*) – A name to use for the version being operated upon. If not None and the *version* argument is None or does not have a value for the *name* key, then this will be populated in the payload for the *name* key. (templated)
- **version** (*dict*) – A dictionary containing the information about the version. If the *operation* is *create*, *version* should contain all the information about this version such as name, and deploymentUrl. If the *operation* is *get* or *delete*, the *version* parameter should contain the *name* of the version. If it is None, the only *operation* possible would be *list*. (templated)
- **operation** (*str*) – The operation to perform. Available operations are:
  - `create`: Creates a new version in the model specified by *model_name*, in which case the *version* parameter should contain all the information to create that version (e.g. *name*, *deploymentUrl*).
  - `get`: Gets full information of a particular version in the model specified by *model_name*. The name of the version should be specified in the *version* parameter.
  - `list`: Lists all available versions of the model specified by *model_name*.
  - `delete`: Deletes the version specified in *version* parameter from the model specified by *model_name*). The name of the version should be specified in the *version* parameter.
- **gcp_conn_id** (*str*) – The connection ID to use when fetching connection info.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['_model_name', '_version_name', '_version']

execute (*self*, *context*)

class airflow.contrib.operators.mlengine_operator.**MLEngineTrainingOperator** (*project_id, job_id, package_uris, training_python_module, training_args, region, scale_tier=None, master_type=None, runtime_version=None, python_version=None, job_dir=None, gcp_conn_id='google_c delegate_to=None, mode='PRODUCTION', *args, **kwargs*)

Bases: `airflow.models.BaseOperator`

Operator for launching a MLEngine training job.

Parameters

- **`project_id`** (*`str`*) – The Google Cloud project name within which MLEngine training job should run (templated).
- **`job_id`** (*`str`*) – A unique templated id for the submitted Google MLEngine training job. (templated)
- **`package_uris`** (*`str`*) – A list of package locations for MLEngine training job, which should include the main training program + any additional dependencies. (templated)
- **`training_python_module`** (*`str`*) – The Python module name to run within MLEngine training job after installing 'package_uris' packages. (templated)
- **`training_args`** (*`str`*) – A list of templated command line arguments to pass to the MLEngine training program. (templated)
- **`region`** (*`str`*) – The Google Compute Engine region to run the MLEngine training job in (templated).
- **`scale_tier`** (*`str`*) – Resource tier for MLEngine training job. (templated)
- **`master_type`** (*`str`*) – Cloud ML Engine machine name. Must be set when scale_tier is CUSTOM. (templated)
- **`runtime_version`** (*`str`*) – The Google Cloud ML runtime version to use for training. (templated)
- **`python_version`** (*`str`*) – The version of Python used in training. (templated)
- **`job_dir`** (*`str`*) – A Google Cloud Storage path in which to store training outputs and other data needed for training. (templated)
- **`gcp_conn_id`** (*`str`*) – The connection ID to use when fetching connection info.
- **`delegate_to`** (*`str`*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **`mode`** (*`str`*) – Can be one of 'DRY_RUN'/'CLOUD'. In 'DRY_RUN' mode, no real training job will be launched, but the MLEngine training job request will be printed out. In 'CLOUD' mode, a real MLEngine training job creation request will be issued.

`template_fields = ['_project_id', '_job_id', '_package_uris', '_training_python_module`

`execute`(*self*, *context*)

`airflow.contrib.operators.mongo_to_s3`

Module Contents

class `airflow.contrib.operators.mongo_to_s3.`**`MongoToS3Operator`**(*mongo_conn_id*, *s3_conn_id*, *mongo_collection*, *mongo_query*, *s3_bucket*, *s3_key*, *mongo_db=None*, *replace=False*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Mongo -> S3 A more specific baseOperator meant to move data from mongo via pymongo to s3 via boto

things to note .execute() is written to depend on .transform() .transform() is meant to be extended by child classes to perform transformations unique to those operators needs

`template_fields = ['s3_key', 'mongo_query']`

`execute`(*self*, *context*)
Executed by task_instance at runtime

`static _stringify`(*iterable*, *joinable='n'*)
Takes an iterable (pymongo Cursor or Array) containing dictionaries and returns a stringified version using python join

`static transform`(*docs*)

Processes pyMongo cursor and returns an iterable with each element being a JSON serializable dictionary

Base transform() assumes no processing is needed ie. docs is a pyMongo cursor of documents and cursor just needs to be passed through

Override this method for custom transformations

`airflow.contrib.operators.mssql_to_gcs`

Module Contents

class `airflow.contrib.operators.mssql_to_gcs.`**`MsSqlToGoogleCloudStorageOperator`**(*sql*, *bucket*, *file-name*, *schema_filename* *ap-prox_max_file_si* *gzip=False*, *mssql_conn_id='r* *google_cloud_stor* *del-e-gate_to=None*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

Copy data from Microsoft SQL Server to Google Cloud Storage in JSON format.

Parameters

- **`sql`** (*`str`*) – The SQL to execute on the MSSQL table.
- **`bucket`** (*`str`*) – The bucket to upload to.
- **`filename`** (*`str`*) – The filename to use as the object name when uploading to Google Cloud Storage. A {} should be specified in the filename to allow the operator to inject file numbers in cases where the file is split due to size, e.g. filename='data/customers/export_{}.json'.
- **`schema_filename`** (*`str`*) – If set, the filename to use as the object name when uploading a .json file containing the BigQuery schema fields for the table that was dumped from MSSQL.
- **`approx_max_file_size_bytes`** (*`long`*) – This operator supports the ability to split large table dumps into multiple files.
- **`gzip`** (*`bool`*) – Option to compress file for upload (does not apply to schemas).

- **mssql_conn_id** (*str*) – Reference to a specific MSSQL hook.
- **google_cloud_storage_conn_id** (*str*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example: The following operator will export data from the Customers table within the given MSSQL Database and then upload it to the 'mssql-export' GCS bucket (along with a schema file).

```
export_customers = MsSqlToGoogleCloudStorageOperator(
    task_id='export_customers',
    sql='SELECT * FROM dbo.Customers;',
    bucket='mssql-export',
    filename='data/customers/export.json',
    schema_filename='schemas/export.json',
    mssql_conn_id='mssql_default',
    google_cloud_storage_conn_id='google_cloud_default',
    dag=dag
)
```

template_fields = ['sql', 'bucket', 'filename', 'schema_filename']

template_ext = ['.sql']

ui_color = #e0a98c

execute (*self*, *context*)

_query_mssql (*self*)
Queries MSSQL and returns a cursor of results.

Returns mssql cursor

_write_local_data_files (*self*, *cursor*)
Takes a cursor, and writes results to a local file.

Returns A dictionary where keys are filenames to be used as object names in GCS, and values are file handles to local files that contain the data for the GCS objects.

_write_local_schema_file (*self*, *cursor*)
Takes a cursor, and writes the BigQuery schema for the results to a local file system.

Returns A dictionary where key is a filename to be used as an object name in GCS, and values are file handles to local files that contains the BigQuery schema fields in .json format.

_upload_to_gcs (*self*, *files_to_upload*)
Upload all of the file splits (and optionally the schema .json file) to Google cloud storage.

classmethod convert_types (*cls*, *value*)
Takes a value from MSSQL, and converts it to a value that's safe for JSON/Google Cloud Storage/BigQuery.

classmethod type_map (*cls*, *mssql_type*)
Helper function that maps from MSSQL fields to BigQuery fields. Used when a schema_filename is set.

airflow.contrib.operators.mysql_to_gcs

Module Contents

class airflow.contrib.operators.mysql_to_gcs.**MySqlToGoogleCloudStorageOperator**(*sql*, *bucket*, *file-name*, *schema_filename*, *ap-prox_max_file_si* *mysql_conn_id='* *google_cloud_stor* *schema=None*, *del-e-gate_to=None*, *ex-port_format='jso* *field_delimiter=',* *'*, *en-sure_utc=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Copy data from MySQL to Google cloud storage in JSON or CSV format.

The JSON data files generated are newline-delimited to enable them to be loaded into BigQuery. Reference: https://cloud.google.com/bigquery/docs/ loading-data-cloud-storage-json#limitations

Parameters

- **sql** (*str*) – The SQL to execute on the MySQL table.
- **bucket** (*str*) – The bucket to upload to.
- **filename** (*str*) – The filename to use as the object name when uploading to Google cloud storage. A {} should be specified in the filename to allow the operator to inject file numbers in cases where the file is split due to size.
- **schema_filename** (*str*) – If set, the filename to use as the object name when uploading a .json file containing the BigQuery schema fields for the table that was dumped from MySQL.
- **approx_max_file_size_bytes** (*long*) – This operator supports the ability to split large table dumps into multiple files (see notes in the filename param docs above). This param allows developers to specify the file size of the splits. Check https://cloud.google.com/storage/quotas to see the maximum allowed file size for a single object.
- **mysql_conn_id** (*str*) – Reference to a specific MySQL hook.
- **google_cloud_storage_conn_id** (*str*) – Reference to a specific Google cloud storage hook.
- **schema** (*str or list*) – The schema to use, if any. Should be a list of dict or a str. Pass a string if using Jinja template, otherwise, pass a list of dict. Examples could be seen: https://cloud.google.com/bigquery/docs /schemas#specifying_a_json_schema_file
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

- **export_format** (*str*) – Desired format of files to be exported.
- **field_delimiter** (*str*) – The delimiter to be used for CSV files.
- **ensure_utc** (*bool*) – Ensure TIMESTAMP columns exported as UTC. If set to *False*, TIMESTAMP columns will be exported using the MySQL server's default timezone.

template_fields = ['sql', 'bucket', 'filename', 'schema_filename', 'schema']

template_ext = ['.sql']

ui_color = #a0e08c

type_map

execute (*self*, *context*)

_query_mysql (*self*)
Queries mysql and returns a cursor to the results.

_write_local_data_files (*self*, *cursor*)
Takes a cursor, and writes results to a local file.

Returns A dictionary where keys are filenames to be used as object names in GCS, and values are file handles to local files that contain the data for the GCS objects.

_configure_csv_file (*self*, *file_handle*, *schema*)
Configure a csv writer with the file_handle and write schema as headers for the new file.

_write_local_schema_file (*self*, *cursor*)
Takes a cursor, and writes the BigQuery schema in .json format for the results to a local file system.

Returns A dictionary where key is a filename to be used as an object name in GCS, and values are file handles to local files that contains the BigQuery schema fields in .json format.

_upload_to_gcs (*self*, *files_to_upload*)
Upload all of the file splits (and optionally the schema .json file) to Google cloud storage.

classmethod _convert_types (*cls*, *schema*, *col_type_dict*, *row*)

classmethod _convert_type (*cls*, *value*, *schema_type*)
Takes a value from MySQLdb, and converts it to a value that's safe for JSON/Google cloud storage/BigQuery. Dates are converted to UTC seconds. Decimals are converted to floats. Binary type fields are encoded with base64, as imported BYTES data must be base64-encoded according to Bigquery SQL date type documentation: https://cloud.google.com/bigquery/data-types

Parameters

- **value** (*Any*) – MySQLdb column value
- **schema_type** (*str*) – BigQuery data type

_get_col_type_dict (*self*)
Return a dict of column name and column type based on self.schema if not None.

airflow.contrib.operators.opsgenie_alert_operator

Module Contents

class airflow.contrib.operators.opsgenie_alert_operator.**OpsgenieAlertOperator**(*message*, *opsgenie_conn_id='opsg*, *alias=None*, *description=None*, *responders=None*, *visibleTo=None*, *actions=None*, *tags=None*, *details=None*, *entity=None*, *source=None*, *priority=None*, *user=None*, *note=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

This operator allows you to post alerts to Opsgenie. Accepts a connection that has an Opsgenie API key as the connection's password. This operator sets the domain to conn_id.host, and if not set will default to `https://api.opsgenie.com`.

Each Opsgenie API key can be pre-configured to a team integration. You can override these defaults in this operator.

Parameters

- **opsgenie_conn_id** (*str*) – The name of the Opsgenie connection to use
- **message** (*str*) – The Message of the Opsgenie alert (templated)
- **alias** (*str*) – Client-defined identifier of the alert (templated)
- **description** (*str*) – Description field of the alert (templated)
- **responders** (*list[dict]*) – Teams, users, escalations and schedules that the alert will be routed to send notifications.
- **visibleTo** (*list[dict]*) – Teams and users that the alert will become visible to without sending any notification.
- **actions** (*list[str]*) – Custom actions that will be available for the alert.
- **tags** (*list[str]*) – Tags of the alert.

- **details** (*dict*) – Map of key-value pairs to use as custom properties of the alert.
- **entity** (*str*) – Entity field of the alert that is generally used to specify which domain alert is related to. (templated)
- **source** (*str*) – Source field of the alert. Default value is IP address of the incoming request.
- **priority** (*str*) – Priority level of the alert. Default value is P3. (templated)
- **user** (*str*) – Display name of the request owner.
- **note** (*str*) – Additional note that will be added while creating the alert. (templated)

template_fields = ['message', 'alias', 'description', 'entity', 'priority', 'note']

_build_opsgenie_payload (*self*)

Construct the Opsgenie JSON payload. All relevant parameters are combined here to a valid Opsgenie JSON payload.

Returns Opsgenie payload (dict) to send

execute (*self*, *context*)

Call the OpsgenieAlertHook to post message

airflow.contrib.operators.oracle_to_azure_data_lake_transfer

Module Contents

class airflow.contrib.operators.oracle_to_azure_data_lake_transfer.**OracleToAzureDataLakeTra**

Bases: *airflow.models.BaseOperator*

Moves data from Oracle to Azure Data Lake. The operator runs the query against Oracle and stores the file locally before loading it into Azure Data Lake.

Parameters

- **filename** (*str*) – file name to be used by the csv file.
- **azure_data_lake_conn_id** (*str*) – destination azure data lake connection.
- **azure_data_lake_path** (*str*) – destination path in azure data lake to put the file.
- **oracle_conn_id** (*str*) – source Oracle connection.

- **sql** (*str*) – SQL query to execute against the Oracle database. (templated)
- **sql_params** (*str*) – Parameters to use in sql query. (templated)
- **delimiter** (*str*) – field delimiter in the file.
- **encoding** (*str*) – encoding type for the file.
- **quotechar** (*str*) – Character to use in quoting.
- **quoting** (*str*) – Quoting strategy. See unicodecsv quoting for more information.

template_fields = ['filename', 'sql', 'sql_params']

ui_color = #e08c8c

_write_temp_file (*self*, *cursor*, *path_to_save*)

execute (*self*, *context*)

airflow.contrib.operators.oracle_to_oracle_transfer

Module Contents

class airflow.contrib.operators.oracle_to_oracle_transfer.**OracleToOracleTransfer** (*oracle_destina des- ti- na- tion_table*, *or- a- cle_source_co source_sql*, *source_sql_pa rows_chunk=5* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from Oracle to Oracle.

Parameters

- **oracle_destination_conn_id** (*str*) – destination Oracle connection.
- **destination_table** (*str*) – destination table to insert rows.
- **oracle_source_conn_id** (*str*) – source Oracle connection.
- **source_sql** (*str*) – SQL query to execute against the source Oracle database. (templated)
- **source_sql_params** (*dict*) – Parameters to use in sql query. (templated)
- **rows_chunk** (*int*) – number of rows per chunk to commit.

template_fields = ['source_sql', 'source_sql_params']

ui_color = #e08c8c

_execute (*self*, *src_hook*, *dest_hook*, *context*)

execute (*self*, *context*)

airflow.contrib.operators.postgres_to_gcs_operator

Module Contents

class airflow.contrib.operators.postgres_to_gcs_operator.**PostgresToGoogleCloudStorageOperat**

Bases: *airflow.models.BaseOperator*

Copy data from Postgres to Google Cloud Storage in JSON format.

template_fields = ['sql', 'bucket', 'filename', 'schema_filename', 'parameters']

template_ext = ['.sql']

ui_color = #a0e08c

execute(*self*, *context*)

_query_postgres(*self*)
Queries Postgres and returns a cursor to the results.

_write_local_data_files(*self*, *cursor*)
Takes a cursor, and writes results to a local file.

Returns A dictionary where keys are filenames to be used as object names in GCS, and values are file handles to local files that contain the data for the GCS objects.

_write_local_schema_file(*self*, *cursor*)
Takes a cursor, and writes the BigQuery schema for the results to a local file system.

Returns A dictionary where key is a filename to be used as an object name in GCS, and values are file handles to local files that contains the BigQuery schema fields in .json format.

_upload_to_gcs(*self*, *files_to_upload*)
Upload all of the file splits (and optionally the schema .json file) to Google Cloud Storage.

classmethod convert_types(*cls*, *value*)
Takes a value from Postgres, and converts it to a value that's safe for JSON/Google Cloud Storage/BigQuery. Dates are converted to UTC seconds. Decimals are converted to floats. Times are converted to seconds.

classmethod type_map(*cls*, *postgres_type*)
Helper function that maps from Postgres fields to BigQuery fields. Used when a schema_filename is set.

airflow.contrib.operators.pubsub_operator

Module Contents

class airflow.contrib.operators.pubsub_operator.**PubSubTopicCreateOperator**(*project, topic, fail_if_exists=False, gcp_conn_id='google_clo del- e- gate_to=None, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Create a PubSub topic.

By default, if the topic already exists, this operator will not cause the DAG to fail.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic')
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic')
    )
```

The operator can be configured to fail if the topic already exists.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic')
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic',
                                     fail_if_exists=True)
    )
```

Both `project` and `topic` are templated so you can use variables in them.

template_fields = ['project', 'topic']

ui_color = #0273d4

execute(*self, context*)

class airflow.contrib.operators.pubsub_operator.**PubSubSubscriptionCreateOperator**(*topic_project, topic, sub- scrip- tion=None, sub- scrip- tion_project=N ack_deadline_ fail_if_exists= gcp_conn_id= del- e- gate_to=None, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Create a PubSub subscription.

By default, the subscription will be created in `topic_project`. If `subscription_project` is specified and the GCP credentials allow, the Subscription can be created in a different project from its topic.

By default, if the subscription already exists, this operator will not cause the DAG to fail. However, the topic must exist in the project.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription')
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription')
    )
```

The operator can be configured to fail if the subscription already exists.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription')
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription', fail_if_exists=True)
    )
```

Finally, subscription is not required. If not passed, the operator will generated a universally unique identifier for the subscription's name.

```
with DAG('DAG') as dag:
    (
        dag >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic')
    )
```

`topic_project`, `topic`, `subscription`, and `subscription` are templated so you can use variables in them.

template_fields = ['topic_project', 'topic', 'subscription', 'subscription_project']

ui_color = #0273d4

execute (*self*, *context*)

class airflow.contrib.operators.pubsub_operator.**PubSubTopicDeleteOperator** (*project*, *topic*, *fail_if_not_exists=False*, *gcp_conn_id='google_clo* *del-* *e-* *gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Delete a PubSub topic.

By default, if the topic does not exist, this operator will not cause the DAG to fail.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubTopicDeleteOperator(project='my-project',
                                     topic='non_existing_topic')
    )
```

The operator can be configured to fail if the topic does not exist.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='non_existing_topic',
                                     fail_if_not_exists=True)
    )
```

Both `project` and `topic` are templated so you can use variables in them.

template_fields = ['project', 'topic']

ui_color = #cb4335

execute (*self*, *context*)

class airflow.contrib.operators.pubsub_operator.**PubSubSubscriptionDeleteOperator** (*project*, *sub-* *scrip-* *tion*, *fail_if_not_exi* *gcp_conn_id=* *del-* *e-* *gate_to=None,* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Delete a PubSub subscription.

By default, if the subscription does not exist, this operator will not cause the DAG to fail.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubSubscriptionDeleteOperator(project='my-project',
                                            subscription='non-existing')
    )
```

The operator can be configured to fail if the subscription already exists.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubSubscriptionDeleteOperator(
             project='my-project', subscription='non-existing',
             fail_if_not_exists=True)
    )
```

`project`, and `subscription` are templated so you can use variables in them.

template_fields = ['project', 'subscription']

ui_color = #cb4335

execute (*self*, *context*)

class airflow.contrib.operators.pubsub_operator.**PubSubPublishOperator** (*project*, *topic*, *messages*, *gcp_conn_id='google_cloud_de*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Publish messages to a PubSub topic.

Each Task publishes all provided messages to the same topic in a single GCP project. If the topic does not exist, this task will fail.

```
from base64 import b64encode as b64e

m1 = {'data': b64e('Hello, World!'),
      'attributes': {'type': 'greeting'}
     }
m2 = {'data': b64e('Knock, knock')}
m3 = {'attributes': {'foo': ''}}

t1 = PubSubPublishOperator(
    project='my-project',topic='my_topic',
    messages=[m1, m2, m3],
    create_topic=True,
    dag=dag)
```

`project` , `topic`, and `messages` are templated so you can use variables in them.

template_fields = ['project', 'topic', 'messages']

ui_color = #0273d4

execute (*self*, *context*)

airflow.contrib.operators.qubole_check_operator

Module Contents

class airflow.contrib.operators.qubole_check_operator.**QuboleCheckOperator** (*qubole_conn_id='qubole_*
**args*,
***kwargs*)

Bases: *airflow.operators.check_operator.CheckOperator*, *airflow.contrib.operators.qubole_operator.QuboleOperator*

Performs checks against Qubole Commands. `QuboleCheckOperator` expects a command that will be executed on QDS. By default, each value on first row of the result of this Qubole Command is evaluated using python `bool` casting. If any of the values return `False`, the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average.

This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alerts without stopping the progress of the DAG.

Parameters **qubole_conn_id** (*str*) – Connection id which consists of qds auth_token

kwargs:

Arguments specific to Qubole command can be referred from QuboleOperator docs.

results_parser_callable This is an optional parameter to extend the flexibility of parsing the results of Qubole command to the users. This is a python callable which can hold the logic to parse list of rows returned by Qubole command. By default, only the values on first row are used for performing checks. This callable should return a list of records on which the checks have to be performed.

Note: All fields in common with template fields of QuboleOperator and CheckOperator are template-supported.

template_fields

template_ext

ui_fgcolor = #000

execute (*self*, *context=None*)

get_db_hook (*self*)

get_hook(*self*, *context=None*)

__getattribute__(*self*, *name*)

__setattr__(*self*, *name*, *value*)

class airflow.contrib.operators.qubole_check_operator.**QuboleValueCheckOperator**(*pass_value*, *tol-er-ance=None*, *qubole_conn_id=*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.ValueCheckOperator*, *airflow.contrib.operators.qubole_operator.QuboleOperator*

Performs a simple value check using Qubole command. By default, each value on the first row of this Qubole command is compared with a pre-defined value. The check fails and errors out if the output of the command is not within the permissible limit of expected value.

Parameters

- **qubole_conn_id** (*str*) – Connection id which consists of qds auth_token
- **pass_value** (*str or int or float*) – Expected value of the query results.
- **tolerance** (*int or float*) – Defines the permissible pass_value range, for example if tolerance is 2, the Qubole command output can be anything between -2*pass_value and 2*pass_value, without the operator erring out.

kwargs:

Arguments specific to Qubole command can be referred from QuboleOperator docs.

results_parser_callable This is an optional parameter to extend the flexibility of parsing the results of Qubole command to the users. This is a python callable which can hold the logic to parse list of rows returned by Qubole command. By default, only the values on first row are used for performing checks. This callable should return a list of records on which the checks have to be performed.

Note: All fields in common with template fields of QuboleOperator and ValueCheckOperator are template-supported.

template_fields

template_ext

ui_fgcolor = #000

execute(*self*, *context=None*)

get_db_hook(*self*)

get_hook(*self*, *context=None*)

__getattribute__(*self*, *name*)

__setattr__(*self*, *name*, *value*)

airflow.contrib.operators.qubole_check_operator.**get_sql_from_qbol_cmd**(*params*)

airflow.contrib.operators.qubole_check_operator.**handle_airflow_exception**(*airflow_exception*, *hook*)

airflow.contrib.operators.qubole_operator

Module Contents

class airflow.contrib.operators.qubole_operator.**QDSLink**

Bases: *airflow.models.baseoperator.BaseOperatorLink*

name = Go to QDS

get_link(*self*, *operator*, *dttm*)

class airflow.contrib.operators.qubole_operator.**QuboleOperator**(*qubole_conn_id='qubole_default'*, **args*, ***kwargs*)

Bases: *airflow.models.baseoperator.BaseOperator*

Execute tasks (commands) on QDS (https://qubole.com).

Parameters **qubole_conn_id** (*str*) – Connection id which consists of qds auth_token

kwargs:

command_type type of command to be executed, e.g. hivecmd, shellcmd, hadoopcmd

tags array of tags to be assigned with the command

cluster_label cluster label on which the command will be executed

name name to be given to command

notify whether to send email on command completion or not (default is False)

Arguments specific to command types

hivecmd:

query inline query statement

script_location s3 location containing query statement

sample_size size of sample in bytes on which to run query

macros macro values which were used in query

sample_size size of sample in bytes on which to run query

hive-version Specifies the hive version to be used. eg: 0.13,1.2,etc.

prestocmd:

query inline query statement

script_location s3 location containing query statement

macros macro values which were used in query

hadoopcmd:

sub_commnad must be one these ["jar", "s3distcp", "streaming"] followed by 1 or more args

shellcmd:

script inline command with args

script_location s3 location containing query statement

files list of files in s3 bucket as file1,file2 format. These files will be copied into the working directory where the qubole command is being executed.

archives list of archives in s3 bucket as archive1,archive2 format. These will be unarchived into the working directory where the qubole command is being executed

parameters any extra args which need to be passed to script (only when script_location is supplied)

pigcmd:

script inline query statement (latin_statements)

script_location s3 location containing pig query

parameters any extra args which need to be passed to script (only when script_location is supplied

sparkcmd:

program the complete Spark Program in Scala, R, or Python

cmdline spark-submit command line, all required information must be specify in cmdline itself.

sql inline sql query

script_location s3 location containing query statement

language language of the program, Scala, R, or Python

app_id ID of an Spark job server app

arguments spark-submit command line arguments

user_program_arguments arguments that the user program takes in

macros macro values which were used in query

note_id Id of the Notebook to run

dbtapquerycmd:

db_tap_id data store ID of the target database, in Qubole.

query inline query statement

macros macro values which were used in query

dbexportcmd:

mode Can be 1 for Hive export or 2 for HDFS/S3 export

schema Db schema name assumed accordingly by database if not specified

hive_table Name of the hive table

partition_spec partition specification for Hive table.

dbtap_id data store ID of the target database, in Qubole.

db_table name of the db table

db_update_mode allowinsert or updateonly

db_update_keys columns used to determine the uniqueness of rows

export_dir HDFS/S3 location from which data will be exported.

fields_terminated_by hex of the char used as column separator in the dataset

use_customer_cluster To use cluster to run command

customer_cluster_label the label of the cluster to run the command on

additional_options Additional Sqoop options which are needed enclose options in double or single quotes e.g. '–map-column-hive id=int,data=string'

dbimportcmd:

mode 1 (simple), 2 (advance)

hive_table Name of the hive table

schema Db schema name assumed accordingly by database if not specified

hive_serde Output format of the Hive Table

dbtap_id data store ID of the target database, in Qubole.

db_table name of the db table

where_clause where clause, if any

parallelism number of parallel db connections to use for extracting data

extract_query SQL query to extract data from db. $CONDITIONS must be part of the where clause.

boundary_query Query to be used get range of row IDs to be extracted

split_column Column used as row ID to split data into ranges (mode 2)

use_customer_cluster To use cluster to run command

customer_cluster_label the label of the cluster to run the command on

additional_options Additional Sqoop options which are needed enclose options in double or single quotes

`template_fields :Iterable[str] = ['query', 'script_location', 'sub_command', 'script',`

`template_ext :Iterable[str] = ['.txt']`

`ui_color = #3064A1`

`ui_fgcolor = #fff`

`qubole_hook_allowed_args_list = ['command_type', 'qubole_conn_id', 'fetch_logs']`

`operator_extra_links`

`_get_filtered_args` (*self*, *all_kwargs*)

`execute` (*self*, *context*)

`on_kill` (*self*, *ti=None*)

`get_results` (*self*, *ti=None*, *fp=None*, *inline=True*, *delim=None*, *fetch=True*)

`get_log` (*self*, *ti*)

`get_jobs_id` (*self*, *ti*)

`get_hook` (*self*)

`__getattribute__` (*self*, *name*)

`__setattr__` (*self*, *name*, *value*)

airflow.contrib.operators.redis_publish_operator

Module Contents

class airflow.contrib.operators.redis_publish_operator.**RedisPublishOperator**(*channel*, *message*, *redis_conn_id='redis_de*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Publish a message to Redis.

Parameters

- **channel** (*str*) – redis channel to which the message is published (templated)
- **message** (*str*) – the message to publish (templated)
- **redis_conn_id** (*str*) – redis connection to use

template_fields = ['channel', 'message']

execute(*self*, *context*)

Publish the message to Redis channel

Parameters **context** (*dict*) – the context object

airflow.contrib.operators.s3_copy_object_operator

Module Contents

class airflow.contrib.operators.s3_copy_object_operator.**S3CopyObjectOperator**(*source_bucket_key*, *dest_bucket_key*, *source_bucket_name*, *dest_bucket_name=1*, *source_version_id=N*, *aws_conn_id='aws_*, *verify=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Creates a copy of an object that is already stored in S3.

Note: the S3 connection used here needs to have access to both source and destination bucket/key.

Parameters

- **source_bucket_key** (*str*) – The key of the source object. (templated)

  It can be either full s3:// style url or relative path from root level.

  When it's specified as a full s3:// url, please omit source_bucket_name.
- **dest_bucket_key** (*str*) – The key of the object to copy to. (templated)

  The convention to specify *dest_bucket_key* is the same as *source_bucket_key*.

- **source_bucket_name** (*str*) – Name of the S3 bucket where the source object is in. (templated)

  It should be omitted when *source_bucket_key* is provided as a full s3:// url.
- **dest_bucket_name** (*str*) – Name of the S3 bucket to where the object is copied. (templated)

  It should be omitted when *dest_bucket_key* is provided as a full s3:// url.
- **source_version_id** (*str*) – Version ID of the source object (OPTIONAL)
- **aws_conn_id** (*str*) – Connection id of the S3 connection to use
- **verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified.

  You can provide the following values:

  – **False: do not validate SSL certificates. SSL will still be used,** but SSL certificates will not be verified.

  – **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.

template_fields = ['source_bucket_key', 'dest_bucket_key', 'source_bucket_name', 'dest_

execute (*self*, *context*)

airflow.contrib.operators.s3_delete_objects_operator

Module Contents

class airflow.contrib.operators.s3_delete_objects_operator.**S3DeleteObjectsOperator**(*bucket*, *keys*, *aws_conn_*, *verify=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

To enable users to delete single object or multiple objects from a bucket using a single HTTP request.

Users may specify up to 1000 keys to delete.

Parameters

- **bucket** (*str*) – Name of the bucket in which you are going to delete object(s). (templated)
- **keys** (*str or list*) – The key(s) to delete from S3 bucket. (templated)

  When `keys` is a string, it's supposed to be the key name of the single object to delete.

  When `keys` is a list, it's supposed to be the list of the keys to delete.

  You may specify up to 1000 keys.
- **aws_conn_id** (*str*) – Connection id of the S3 connection to use
- **verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified.

  You can provide the following values:

- **False: do not validate SSL certificates. SSL will still be used,** but SSL certificates will not be verified.
- **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.

template_fields = ['keys', 'bucket']

execute (*self*, *context*)

airflow.contrib.operators.s3_list_operator

Module Contents

class airflow.contrib.operators.s3_list_operator.**S3ListOperator** (*bucket*, *prefix=''*, *delimiter=''*, *aws_conn_id='aws_default'*, *verify=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

List all objects from the bucket with the given string prefix in name.

This operator returns a python list with the name of objects which can be used by *xcom* in the downstream task.

Parameters

- **bucket** (*str*) – The S3 bucket where to find the objects. (templated)
- **prefix** (*str*) – Prefix string to filters the objects whose name begin with such prefix. (templated)
- **delimiter** (*str*) – the delimiter marks key hierarchy. (templated)
- **aws_conn_id** (*str*) – The connection ID to use when connecting to S3 storage.
- **verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **False: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.

Example: The following operator would list all the files (excluding subfolders) from the S3 `customers/2018/04/` key in the `data` bucket.

```
s3_file = S3ListOperator(
    task_id='list_3s_files',
    bucket='data',
    prefix='customers/2018/04/',
    delimiter='/',
    aws_conn_id='aws_customers_conn'
)
```

template_fields :Iterable[str] = ['bucket', 'prefix', 'delimiter']

ui_color = #ffd700

execute (*self*, *context*)

airflow.contrib.operators.s3_to_gcs_operator

Module Contents

class airflow.contrib.operators.s3_to_gcs_operator.**S3ToGoogleCloudStorageOperator** (*bucket*, *pre-fix=''*, *de-lim-iter=''*, *aws_conn_id* *ver-ify=None*, *dest_gcs_con* *dest_gcs=No* *del-e-gate_to=Non* *re-place=False*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.s3_list_operator.S3ListOperator*

Synchronizes an S3 key, possibly a prefix, with a Google Cloud Storage destination path.

Parameters

- **bucket** (*str*) – The S3 bucket where to find the objects. (templated)
- **prefix** (*str*) – Prefix string which filters objects whose name begin with such prefix. (templated)
- **delimiter** (*str*) – the delimiter marks key hierarchy. (templated)
- **aws_conn_id** (*str*) – The source S3 connection
- **verify** (*bool or str*) – Whether or not to verify SSL certificates for S3 connection. By default SSL certificates are verified. You can provide the following values:
  - **False: do not validate SSL certificates. SSL will still be used** (unless use_ssl is False), but SSL certificates will not be verified.
  - **path/to/cert/bundle.pem: A filename of the CA cert bundle to uses.** You can specify this argument if you want to use a different CA cert bundle than the one used by botocore.
- **dest_gcs_conn_id** (*str*) – The destination connection ID to use when connecting to Google Cloud Storage.
- **dest_gcs** (*str*) – The destination Google Cloud Storage bucket and prefix where you want to store the files. (templated)
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **replace** (*bool*) – Whether you want to replace existing destination files or not.

Example:

```
s3_to_gcs_op = S3ToGoogleCloudStorageOperator(
     task_id='s3_to_gcs_example',
     bucket='my-s3-bucket',
     prefix='data/customers-201804',
     dest_gcs_conn_id='google_cloud_default',
     dest_gcs='gs://my.gcs.bucket/some/customers/',
     replace=False,
     dag=my-dag)
```

Note that `bucket`, `prefix`, `delimiter` and `dest_gcs` are templated, so you can use variables in them if you wish.

template_fields = ['bucket', 'prefix', 'delimiter', 'dest_gcs']

ui_color = #e09411

execute (*self*, *context*)

static _gcs_object_is_directory (*object*)

airflow.contrib.operators.s3_to_sftp_operator

Module Contents

class airflow.contrib.operators.s3_to_sftp_operator.**S3ToSFTPOperator** (*s3_bucket*, *s3_key*, *sftp_path*, *sftp_conn_id='ssh_default'*, *s3_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

This operator enables the transferring of files from S3 to a SFTP server.

Parameters

- **sftp_conn_id** (*string*) – The sftp connection id. The name or identifier for establishing a connection to the SFTP server.
- **sftp_path** (*string*) – The sftp remote path. This is the specified file path for uploading file to the SFTP server.
- **s3_conn_id** (*string*) – The s3 connection id. The name or identifier for establishing a connection to S3
- **s3_bucket** (*string*) – The targeted s3 bucket. This is the S3 bucket from where the file is downloaded.
- **s3_key** (*string*) – The targeted s3 key. This is the specified file path for downloading the file from S3.

template_fields = ['s3_key', 'sftp_path']

static get_s3_key (*s3_key*)

This parses the correct format for S3 keys regardless of how the S3 url is passed.

execute (*self*, *context*)

airflow.contrib.operators.sagemaker_base_operator

Module Contents

class airflow.contrib.operators.sagemaker_base_operator.**SageMakerBaseOperator**(*config*, *aws_conn_id='aws*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

This is the base operator for all SageMaker operators.

Parameters

- **config** (*dict*) – The configuration necessary to start a training job (templated)
- **aws_conn_id** (*str*) – The AWS connection ID to use.

template_fields = ['config']

template_ext = []

ui_color = #ededed

integer_fields :Iterable[Iterable[str]] = []

parse_integer(*self*, *config*, *field*)

parse_config_integers(*self*)

expand_role(*self*)

preprocess_config(*self*)

execute(*self*, *context*)

airflow.contrib.operators.sagemaker_endpoint_config_operator

Module Contents

class airflow.contrib.operators.sagemaker_endpoint_config_operator.**SageMakerEndpointConfig**

Bases: *airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator*

Create a SageMaker endpoint config.

This operator returns The ARN of the endpoint config created in Amazon SageMaker

Parameters

- **config** (*dict*) – The configuration necessary to create an endpoint config.

  For details of the configuration parameter see `SageMaker.Client.create_endpoint_config()`
- **aws_conn_id** (*str*) – The AWS connection ID to use.

integer_fields = [['ProductionVariants', 'InitialInstanceCount']]

execute(*self*, *context*)

airflow.contrib.operators.sagemaker_endpoint_operator

Module Contents

class airflow.contrib.operators.sagemaker_endpoint_operator.**SageMakerEndpointOperator**(*config, wait_f check_ max_i op- er- a- tion='c *args, **kwa*

Bases: *airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator*

Create a SageMaker endpoint.

This operator returns The ARN of the endpoint created in Amazon SageMaker

Parameters

- **config** (*dict*) – The configuration necessary to create an endpoint.

  If you need to create a SageMaker endpoint based on an existed SageMaker model and an existed SageMaker endpoint config:

  ```
  config = endpoint_configuration;
  ```

  If you need to create all of SageMaker model, SageMaker endpoint-config and SageMaker endpoint:

  ```
  config = {
      'Model': model_configuration,
      'EndpointConfig': endpoint_config_configuration,
      'Endpoint': endpoint_configuration
  }
  ```

  For details of the configuration parameter of model_configuration see `SageMaker.Client.create_model()`

  For details of the configuration parameter of endpoint_config_configuration see `SageMaker.Client.create_endpoint_config()`

  For details of the configuration parameter of endpoint_configuration see `SageMaker.Client.create_endpoint()`

- **aws_conn_id** (*str*) – The AWS connection ID to use.
- **wait_for_completion** (*bool*) – Whether the operator should wait until the endpoint creation finishes.
- **check_interval** (*int*) – If wait is set to True, this is the time interval, in seconds, that this operation waits before polling the status of the endpoint creation.
- **max_ingestion_time** (*int*) – If wait is set to True, this operation fails if the endpoint creation doesn't finish within max_ingestion_time seconds. If you set this parameter to None it never times out.
- **operation** (*str*) – Whether to create an endpoint or update an endpoint. Must be either 'create or 'update'.

create_integer_fields (*self*)

expand_role (*self*)

execute (*self*, *context*)

airflow.contrib.operators.sagemaker_model_operator

Module Contents

class airflow.contrib.operators.sagemaker_model_operator.**SageMakerModelOperator** (*config*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator*

Create a SageMaker model.

This operator returns The ARN of the model created in Amazon SageMaker

Parameters

- **config** (*dict*) – The configuration necessary to create a model.

  For details of the configuration parameter see `SageMaker.Client.create_model()`
- **aws_conn_id** (*str*) – The AWS connection ID to use.

expand_role (*self*)

execute (*self*, *context*)

airflow.contrib.operators.sagemaker_training_operator

Module Contents

class airflow.contrib.operators.sagemaker_training_operator.**SageMakerTrainingOperator** (*config*, *wait_f*, *print_l*, *check_*, *max_i*, **args*, ***kwa*

Bases: *airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator*

Initiate a SageMaker training job.

This operator returns The ARN of the training job created in Amazon SageMaker.

Parameters

- **config** (*dict*) – The configuration necessary to start a training job (templated).

  For details of the configuration parameter see `SageMaker.Client.create_training_job()`
- **aws_conn_id** (*str*) – The AWS connection ID to use.
- **wait_for_completion** (*bool*) – If wait is set to True, the time interval, in seconds, that the operation waits to check the status of the training job.
- **print_log** (*bool*) – if the operator should print the cloudwatch log during training

- **check_interval** (*int*) – if wait is set to be true, this is the time interval in seconds which the operator will check the status of the training job
- **max_ingestion_time** (*int*) – If wait is set to True, the operation fails if the training job doesn't finish within max_ingestion_time seconds. If you set this parameter to None, the operation does not timeout.

integer_fields = [['ResourceConfig', 'InstanceCount'], ['ResourceConfig', 'VolumeSizeI

expand_role (*self*)

execute (*self*, *context*)

airflow.contrib.operators.sagemaker_transform_operator

Module Contents

class airflow.contrib.operators.sagemaker_transform_operator.**SageMakerTransformOperator** (*con*
wai
chec
max
**ar*
***k*

Bases: *airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator*

Initiate a SageMaker transform job.

This operator returns The ARN of the model created in Amazon SageMaker.

Parameters

- **config** (*dict*) – The configuration necessary to start a transform job (templated).

  If you need to create a SageMaker transform job based on an existed SageMaker model:

```
config = transform_config
```

  If you need to create both SageMaker model and SageMaker Transform job:

```
config = {
    'Model': model_config,
    'Transform': transform_config
}
```

  For details of the configuration parameter of transform_config see `SageMaker.Client.create_transform_job()`

  For details of the configuration parameter of model_config, See: `SageMaker.Client.create_model()`

- **aws_conn_id** (*string*) – The AWS connection ID to use.
- **wait_for_completion** (*bool*) – Set to True to wait until the transform job finishes.
- **check_interval** (*int*) – If wait is set to True, the time interval, in seconds, that this operation waits to check the status of the transform job.
- **max_ingestion_time** (*int*) – If wait is set to True, the operation fails if the transform job doesn't finish within max_ingestion_time seconds. If you set this parameter to None, the operation does not timeout.

create_integer_fields (*self*)

expand_role (*self*)

execute (*self*, *context*)

airflow.contrib.operators.sagemaker_tuning_operator

Module Contents

class airflow.contrib.operators.sagemaker_tuning_operator.**SageMakerTuningOperator** (*config*, *wait_for_co*, *check_interv*, *max_ingestio*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator*

Initiate a SageMaker hyperparameter tuning job.

This operator returns The ARN of the tuning job created in Amazon SageMaker.

Parameters

- **config** (*dict*) – The configuration necessary to start a tuning job (templated).

  For details of the configuration parameter see `SageMaker.Client.create_hyper_parameter_tuning_job()`
- **aws_conn_id** (*str*) – The AWS connection ID to use.
- **wait_for_completion** (*bool*) – Set to True to wait until the tuning job finishes.
- **check_interval** (*int*) – If wait is set to True, the time interval, in seconds, that this operation waits to check the status of the tuning job.
- **max_ingestion_time** (*int*) – If wait is set to True, the operation fails if the tuning job doesn't finish within max_ingestion_time seconds. If you set this parameter to None, the operation does not timeout.

integer_fields = [['HyperParameterTuningJobConfig', 'ResourceLimits', 'MaxNumberOfTrai

expand_role (*self*)

execute (*self*, *context*)

airflow.contrib.operators.segment_track_event_operator

Module Contents

class airflow.contrib.operators.segment_track_event_operator.**SegmentTrackEventOperator**(*user_ event prop- er- ties= seg- ment_ seg- ment_ *args **kw*

Bases: *airflow.models.BaseOperator*

Send Track Event to Segment for a specified user_id and event

Parameters

- **user_id** (*str*) – The ID for this user in your database. (templated)
- **event** (*str*) – The name of the event you're tracking. (templated)
- **properties** (*dict*) – A dictionary of properties for the event. (templated)
- **segment_conn_id** (*str*) – The connection ID to use when connecting to Segment.
- **segment_debug_mode** (*bool*) – Determines whether Segment should run in debug mode. Defaults to False

template_fields = ['user_id', 'event', 'properties']

ui_color = #ffd700

execute(*self*, *context*)

airflow.contrib.operators.sftp_operator

Module Contents

class airflow.contrib.operators.sftp_operator.**SFTPOperation**

PUT = put

GET = get

class airflow.contrib.operators.sftp_operator.**SFTPOperator**(*ssh_hook=None, ssh_conn_id=None, remote_host=None, local_filepath=None, remote_filepath=None, operation=SFTPOperation.PUT, confirm=True, create_intermediate_dirs=False, *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

SFTPOperator for transferring files from remote host to local or vice a versa. This operator uses ssh_hook to open sftp transport channel that serve as basis for file transfer.

Parameters

- **ssh_hook** (airflow.contrib.hooks.ssh_hook.SSHHook) – predefined ssh_hook to use for remote execution. Either *ssh_hook* or *ssh_conn_id* needs to be provided.
- **ssh_conn_id** (*str*) – connection id from airflow Connections. *ssh_conn_id* will be ignored if *ssh_hook* is provided.
- **remote_host** (*str*) – remote host to connect (templated) Nullable. If provided, it will replace the *remote_host* which was defined in *ssh_hook* or predefined in the connection of *ssh_conn_id*.
- **local_filepath** (*str*) – local file path to get or put. (templated)
- **remote_filepath** (*str*) – remote file path to get or put. (templated)
- **operation** (*str*) – specify operation 'get' or 'put', defaults to put
- **confirm** (*bool*) – specify if the SFTP operation should be confirmed, defaults to True
- **create_intermediate_dirs** (*bool*) – create missing intermediate directories when copying from remote to local and vice-versa. Default is False.

  Example: The following task would copy `file.txt` to the remote host at `/tmp/tmp1/tmp2/` while creating `tmp`,"tmp1" and `tmp2` if they don't exist. If the parameter is not passed it would error as the directory does not exist.

```
put_file = SFTPOperator(
    task_id="test_sftp",
    ssh_conn_id="ssh_default",
    local_filepath="/tmp/file.txt",
    remote_filepath="/tmp/tmp1/tmp2/file.txt",
    operation="put",
    create_intermediate_dirs=True,
    dag=dag
)
```

template_fields = ['local_filepath', 'remote_filepath', 'remote_host']

execute (*self*, *context*)

airflow.contrib.operators.sftp_operator.**_make_intermediate_dirs** (*sftp_client*, *remote_directory*)

Create all the intermediate directories in a remote host

Parameters

- **sftp_client** – A Paramiko SFTP client.
- **remote_directory** – Absolute Path of the directory containing the file

Returns

`airflow.contrib.operators.sftp_to_s3_operator`

Module Contents

class `airflow.contrib.operators.sftp_to_s3_operator.`**`SFTPToS3Operator`**(*s3_bucket*, *s3_key*, *sftp_path*, *sftp_conn_id='ssh_default'*, *s3_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *`airflow.models.BaseOperator`*

This operator enables the transferring of files from a SFTP server to Amazon S3.

Parameters

- **`sftp_conn_id`** (*`string`*) – The sftp connection id. The name or identifier for establishing a connection to the SFTP server.
- **`sftp_path`** (*`string`*) – The sftp remote path. This is the specified file path for downloading the file from the SFTP server.
- **`s3_conn_id`** (*`string`*) – The s3 connection id. The name or identifier for establishing a connection to S3
- **`s3_bucket`** (*`string`*) – The targeted s3 bucket. This is the S3 bucket to where the file is uploaded.
- **`s3_key`** (*`string`*) – The targeted s3 key. This is the specified path for uploading the file to S3.

`template_fields = ['s3_key', 'sftp_path']`

`static get_s3_key`(*s3_key*)

This parses the correct format for S3 keys regardless of how the S3 url is passed.

`execute`(*self*, *context*)

airflow.contrib.operators.slack_webhook_operator

Module Contents

class airflow.contrib.operators.slack_webhook_operator.**SlackWebhookOperator**(*http_conn_id=None, webhook_token=None, message='', attachments=None, channel=None, username=None, icon_emoji=None, link_names=False, proxy=None, *args, **kwargs*)

Bases: `airflow.operators.http_operator.SimpleHttpOperator`

This operator allows you to post messages to Slack using incoming webhooks. Takes both Slack webhook token directly and connection that has Slack webhook token. If both supplied, http_conn_id will be used as base_url, and webhook_token will be taken as endpoint, the relative path of the url.

Each Slack webhook token can be pre-configured to use a specific channel, username and icon. You can override these defaults in this hook.

Parameters

- **http_conn_id** (*str*) – connection that has Slack webhook token in the extra field
- **webhook_token** (*str*) – Slack webhook token
- **message** (*str*) – The message you want to send on Slack
- **attachments** (*list*) – The attachments to send on Slack. Should be a list of dictionaries representing Slack attachments.
- **channel** (*str*) – The channel the message should be posted to
- **username** (*str*) – The username to post to slack with
- **icon_emoji** (*str*) – The emoji to use as icon for the user posting to Slack
- **link_names** (*bool*) – Whether or not to find and link channel and usernames in your message
- **proxy** (*str*) – Proxy to use to make the Slack webhook call

execute(*self, context*)

Call the SlackWebhookHook to post the provided Slack message

airflow.contrib.operators.snowflake_operator

Module Contents

class airflow.contrib.operators.snowflake_operator.**SnowflakeOperator**(*sql*, *snowflake_conn_id='snowflake_* *parameters=None*, *autocommit=True*, *warehouse=None*, *database=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a Snowflake database

Parameters

- **snowflake_conn_id** (*str*) – reference to specific snowflake connection id
- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'*) – the sql code to be executed. (templated)
- **warehouse** (*str*) – name of warehouse which overwrite defined one in connection
- **database** (*str*) – name of database which overwrite defined one in connection

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #ededed

get_hook(*self*)

execute(*self*, *context*)

airflow.contrib.operators.sns_publish_operator

Module Contents

class airflow.contrib.operators.sns_publish_operator.**SnsPublishOperator**(*target_arn*, *message*, *aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Publish a message to Amazon SNS.

Parameters

- **aws_conn_id** (*str*) – aws connection to use

- **target_arn** (*str*) – either a TopicArn or an EndpointArn
- **message** (*str*) – the default message you want to send (templated)

template_fields = ['message']

template_ext = []

execute(*self*, *context*)

airflow.contrib.operators.spark_jdbc_operator

Module Contents

class airflow.contrib.operators.spark_jdbc_operator.**SparkJDBCOperator**(*spark_app_name='airflow-spark-jdbc', spark_conn_id='spark-default', spark_conf=None, spark_py_files=None, spark_files=None, spark_jars=None, num_executors=None, executor_cores=None, executor_memory=None, driver_memory=None, verbose=False, keytab=None, principal=None, cmd_type='spark_to_jdbc', jdbc_table=None, jdbc_conn_id='jdbc-default', jdbc_driver=None, metastore_table=None, jdbc_truncate=False, save_mode=None, save_format=None, batch_size=None, fetch_size=None, num_partitions=None, partition_column=None, lower_bound=None, upper_bound=None, create_table_column_types=None, *args, **kwargs*)

Bases: `airflow.contrib.operators.spark_submit_operator.SparkSubmitOperator`

This operator extends the SparkSubmitOperator specifically for performing data transfers to/from JDBC-based databases with Apache Spark. As with the SparkSubmitOperator, it assumes that the “spark-submit” binary is available on the PATH.

Parameters

- **spark_app_name** (*str*) – Name of the job (default airflow-spark-jdbc)
- **spark_conn_id** (*str*) – Connection id as configured in Airflow administration
- **spark_conf** (*dict*) – Any additional Spark configuration properties
- **spark_py_files** (*str*) – Additional python files used (.zip, .egg, or .py)
- **spark_files** (*str*) – Additional files to upload to the container running the job
- **spark_jars** (*str*) – Additional jars to upload and add to the driver and executor classpath
- **num_executors** (*int*) – number of executor to run. This should be set so as to manage the number of connections made with the JDBC database
- **executor_cores** (*int*) – Number of cores per executor
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G)
- **driver_memory** (*str*) – Memory allocated to the driver (e.g. 1000M, 2G)
- **verbose** (*bool*) – Whether to pass the verbose flag to spark-submit for debugging
- **keytab** (*str*) – Full path to the file that contains the keytab
- **principal** (*str*) – The name of the kerberos principal used for keytab
- **cmd_type** (*str*) – Which way the data should flow. 2 possible values: spark_to_jdbc: data written by spark from metastore to jdbc jdbc_to_spark: data written by spark from jdbc to metastore
- **jdbc_table** (*str*) – The name of the JDBC table
- **jdbc_conn_id** (*str*) – Connection id used for connection to JDBC database
- **jdbc_driver** (*str*) – Name of the JDBC driver to use for the JDBC connection. This driver (usually a jar) should be passed in the 'jars' parameter
- **metastore_table** (*str*) – The name of the metastore table,
- **jdbc_truncate** (*bool*) – (spark_to_jdbc only) Whether or not Spark should truncate or drop and recreate the JDBC table. This only takes effect if 'save_mode' is set to Overwrite. Also, if the schema is different, Spark cannot truncate, and will drop and recreate
- **save_mode** (*str*) – The Spark save-mode to use (e.g. overwrite, append, etc.)
- **save_format** (*str*) – (jdbc_to_spark-only) The Spark save-format to use (e.g. parquet)
- **batch_size** (*int*) – (spark_to_jdbc only) The size of the batch to insert per round trip to the JDBC database. Defaults to 1000
- **fetch_size** (*int*) – (jdbc_to_spark only) The size of the batch to fetch per round trip from the JDBC database. Default depends on the JDBC driver
- **num_partitions** (*int*) – The maximum number of partitions that can be used by Spark simultaneously, both for spark_to_jdbc and jdbc_to_spark operations. This will also cap the number of JDBC connections that can be opened
- **partition_column** (*str*) – (jdbc_to_spark-only) A numeric column to be used to partition the metastore table by. If specified, you must also specify: num_partitions, lower_bound, upper_bound
- **lower_bound** (*int*) – (jdbc_to_spark-only) Lower bound of the range of the numeric partition column to fetch. If specified, you must also specify: num_partitions, partition_column, upper_bound
- **upper_bound** (*int*) – (jdbc_to_spark-only) Upper bound of the range of the numeric partition column to fetch. If specified, you must also specify: num_partitions, partition_column, lower_bound

- **create_table_column_types** – (spark_to_jdbc-only) The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g: "name CHAR(64), comments VARCHAR(1024)"). The specified types should be valid spark sql data types.

execute(*self*, *context*)
Call the SparkSubmitHook to run the provided spark job

on_kill(*self*)

airflow.contrib.operators.spark_sql_operator

Module Contents

class airflow.contrib.operators.spark_sql_operator.**SparkSqlOperator**(*sql*, *conf=None*, *conn_id='spark_sql_default'*, *total_executor_cores=None*, *executor_cores=None*, *executor_memory=None*, *keytab=None*, *principal=None*, *master='yarn'*, *name='default-name'*, *num_executors=None*, *yarn_queue='default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Execute Spark SQL query

Parameters

- **sql** (*str*) – The SQL query to execute. (templated)
- **conf** (*str (format: PROP=VALUE)*) – arbitrary Spark configuration property
- **conn_id** (*str*) – connection_id string
- **total_executor_cores** (*int*) – (Standalone & Mesos only) Total cores for all executors (Default: all the available cores on the worker)
- **executor_cores** (*int*) – (Standalone & YARN only) Number of cores per executor (Default: 2)
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G) (Default: 1G)
- **keytab** (*str*) – Full path to the file that contains the keytab
- **master** (*str*) – spark://host:port, mesos://host:port, yarn, or local
- **name** (*str*) – Name of the job
- **num_executors** (*int*) – Number of executors to launch

- **verbose** (*bool*) – Whether to pass the verbose flag to spark-sql
- **yarn_queue** (*str*) – The YARN queue to submit to (Default: “default”)

template_fields = ['_sql']

template_ext = ['.sql', '.hql']

execute (*self*, *context*)
Call the SparkSqlHook to run the provided sql query

on_kill (*self*)

`airflow.contrib.operators.spark_submit_operator`

Module Contents

class airflow.contrib.operators.spark_submit_operator.**SparkSubmitOperator**(*application='',
conf=None,
conn_id='spark_default',
files=None,
py_files=None,
archives=None,
driver_class_path=None,
jars=None,
java_class=None,
pack-
ages=None,
ex-
clude_packages=None,
repos-
i-
to-
ries=None,
to-
tal_executor_cores=None
ex-
ecu-
tor_cores=None,
ex-
ecu-
tor_memory=None,
driver_memory=None,
keytab=None,
prin-
ci-
pal=None,
name='airflow-
spark',
num_executors=None,
ap-
pli-
ca-
tion_args=None,
env_vars=None,
ver-
bose=False,
spark_binary='spark-
submit',
*args,
**kwargs*)

Bases: `airflow.models.BaseOperator`

This hook is a wrapper around the spark-submit binary to kick off a spark-submit job. It requires that the "spark-submit" binary is in the PATH or the spark-home is set in the extra on the connection.

Parameters

- **application** (*str*) – The application that submitted as a job, either jar or py file. (templated)
- **conf** (*dict*) – Arbitrary Spark configuration properties (templated)
- **conn_id** (*str*) – The connection id as configured in Airflow administration. When an invalid connection_id is supplied, it will default to yarn.
- **files** (*str*) – Upload additional files to the executor running the job, separated by a comma. Files will be placed in the working directory of each executor. For example, serialized objects. (templated)
- **py_files** (*str*) – Additional python files used by the job, can be .zip, .egg or .py. (templated)
- **jars** (*str*) – Submit additional jars to upload and place them in executor classpath. (templated)
- **driver_class_path** (*str*) – Additional, driver-specific, classpath settings. (templated)
- **java_class** (*str*) – the main class of the Java application
- **packages** (*str*) – Comma-separated list of maven coordinates of jars to include on the driver and executor classpaths. (templated)
- **exclude_packages** (*str*) – Comma-separated list of maven coordinates of jars to exclude while resolving the dependencies provided in 'packages' (templated)
- **repositories** (*str*) – Comma-separated list of additional remote repositories to search for the maven coordinates given with 'packages'
- **total_executor_cores** (*int*) – (Standalone & Mesos only) Total cores for all executors (Default: all the available cores on the worker)
- **executor_cores** (*int*) – (Standalone & YARN only) Number of cores per executor (Default: 2)
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G) (Default: 1G)
- **driver_memory** (*str*) – Memory allocated to the driver (e.g. 1000M, 2G) (Default: 1G)
- **keytab** (*str*) – Full path to the file that contains the keytab (templated)
- **principal** (*str*) – The name of the kerberos principal used for keytab (templated)
- **name** (*str*) – Name of the job (default airflow-spark). (templated)
- **num_executors** (*int*) – Number of executors to launch
- **application_args** (*list*) – Arguments for the application being submitted (templated)
- **env_vars** (*dict*) – Environment variables for spark-submit. It supports yarn and k8s mode too. (templated)
- **verbose** (*bool*) – Whether to pass the verbose flag to spark-submit process for debugging
- **spark_binary** (*string*) – The command to use for spark submit. Some distros may use spark2-submit.

template_fields = ['_application', '_conf', '_files', '_py_files', '_jars', '_driver_c

ui_color

execute (*self*, *context*)
Call the SparkSubmitHook to run the provided spark job

on_kill (*self*)

airflow.contrib.operators.sqoop_operator

This module contains a sqoop 1 operator

Module Contents

class airflow.contrib.operators.sqoop_operator.**SqoopOperator**(*conn_id='sqoop_default', cmd_type='import', table=None, query=None, target_dir=None, append=None, file_type='text', columns=None, num_mappers=None, split_by=None, where=None, export_dir=None, input_null_string=None, input_null_non_string=None, staging_table=None, clear_staging_table=False, enclosed_by=None, escaped_by=None, input_fields_terminated_by=None, input_lines_terminated_by=None, input_optionally_enclosed_by=None, batch=False, direct=False, driver=None, verbose=False, relaxed_isolation=False, properties=None, hcatalog_database=None, hcatalog_table=None, create_hcatalog_table=False, extra_import_options=None, extra_export_options=None, *args, **kwargs*)

Bases: `airflow.models.BaseOperator`

Execute a Sqoop job. Documentation for Apache Sqoop can be found here: https://sqoop.apache.org/docs/1.4.2/SqoopUserGuide.html

template_fields = ['conn_id', 'cmd_type', 'table', 'query', 'target_dir', 'file_type',

ui_color = #7D8CA4

execute(*self*, *context*)
Execute sqoop job

on_kill(*self*)

airflow.contrib.operators.ssh_operator

Module Contents

class airflow.contrib.operators.ssh_operator.**SSHOperator**(*ssh_hook=None*, *ssh_conn_id=None*, *remote_host=None*, *command=None*, *timeout=10*, *environment=None*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

SSHOperator to execute commands on given remote host using the ssh_hook.

Parameters

- **ssh_hook** (`airflow.contrib.hooks.ssh_hook.SSHHook`) – predefined ssh_hook to use for remote execution. Either *ssh_hook* or *ssh_conn_id* needs to be provided.
- **ssh_conn_id** (*str*) – connection id from airflow Connections. *ssh_conn_id* will be ignored if *ssh_hook* is provided.
- **remote_host** (*str*) – remote host to connect (templated) Nullable. If provided, it will replace the *remote_host* which was defined in *ssh_hook* or predefined in the connection of *ssh_conn_id*.
- **command** (*str*) – command to execute on remote host. (templated)
- **timeout** (*int*) – timeout (in seconds) for executing the command.
- **environment** (*dict*) – a dict of shell environment variables. Note that the server will reject them silently if *AcceptEnv* is not set in SSH config.

template_fields = ['command', 'remote_host']

template_ext = ['.sh']

execute(*self*, *context*)

tunnel(*self*)

airflow.contrib.operators.vertica_operator

Module Contents

class airflow.contrib.operators.vertica_operator.**VerticaOperator**(*sql*, *vertica_conn_id='vertica_default'*, **args*, ***kwargs*)

Bases: `airflow.models.BaseOperator`

Executes sql code in a specific Vertica database

Parameters

- **vertica_conn_id** (*str*) – reference to a specific Vertica database
- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'*) – the sql code to be executed. (templated)

template_fields = ['sql']

template_ext = ['.sql']

ui_color = #b4e0ff

execute (*self*, *context*)

airflow.contrib.operators.vertica_to_hive

Module Contents

class airflow.contrib.operators.vertica_to_hive.**VerticaToHiveTransfer** (*sql*, *hive_table*, *create=True*, *recreate=False*, *partition=None*, *delimiter=chr(1)*, *vertica_conn_id='vertica_default'*, *hive_cli_conn_id='hive_cli_def*
**args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from Vertica to Hive. The operator runs your query against Vertica, stores the file locally before loading it into a Hive table. If the `create` or `recreate` arguments are set to `True`, a `CREATE TABLE` and `DROP TABLE` statements are generated. Hive data types are inferred from the cursor's metadata. Note that the table generated in Hive uses `STORED AS textfile` which isn't the most efficient serialization format. If a large amount of data is loaded and/or if the table gets queried considerably, you may want to use this operator only to stage the data into a temporary table before loading it into its final destination using a `HiveOperator`.

Parameters

- **sql** (*str*) – SQL query to execute against the Vertica database. (templated)
- **hive_table** (*str*) – target Hive table, use dot notation to target a specific database. (templated)
- **create** (*bool*) – whether to create the table if it doesn't exist
- **recreate** (*bool*) – whether to drop and recreate the table at every execution
- **partition** (*dict*) – target partition as a dict of partition columns and values. (templated)
- **delimiter** (*str*) – field delimiter in the file
- **vertica_conn_id** (*str*) – source Vertica connection

- **hive_conn_id** (*str*) – destination hive connection

template_fields = ['sql', 'partition', 'hive_table']

template_ext = ['.sql']

ui_color = #b4e0ff

classmethod type_map (*cls*, *vertica_type*)

execute (*self*, *context*)

airflow.contrib.operators.vertica_to_mysql

Module Contents

class airflow.contrib.operators.vertica_to_mysql.**VerticaToMySqlTransfer** (*sql*, *mysql_table*, *ver-tica_conn_id='vertica_defau* *mysql_conn_id='mysql_defa* *mysql_preoperator=None*, *mysql_postoperator=None*, *bulk_load=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from Vertica to MySQL.

Parameters

- **sql** (*str*) – SQL query to execute against the Vertica database. (templated)
- **vertica_conn_id** (*str*) – source Vertica connection
- **mysql_table** (*str*) – target MySQL table, use dot notation to target a specific database. (templated)
- **mysql_conn_id** (*str*) – source mysql connection
- **mysql_preoperator** (*str*) – sql statement to run against MySQL prior to import, typically use to truncate of delete in place of the data coming in, allowing the task to be idempotent (running the task twice won't double load data). (templated)
- **mysql_postoperator** (*str*) – sql statement to run against MySQL after the import, typically used to move data from staging to production and issue cleanup commands. (templated)
- **bulk_load** (*bool*) – flag to use bulk_load option. This loads MySQL directly from a tab-delimited text file using the LOAD DATA LOCAL INFILE command. This option requires an extra connection parameter for the destination MySQL connection: {'local_infile': true}.

template_fields = ['sql', 'mysql_table', 'mysql_preoperator', 'mysql_postoperator']

template_ext = ['.sql']

ui_color = #a0e08c

execute (*self*, *context*)

airflow.contrib.operators.wasb_delete_blob_operator

Module Contents

class airflow.contrib.operators.wasb_delete_blob_operator.**WasbDeleteBlobOperator**(*container_nam*
blob_name,
wasb_conn_id
check_options
is_prefix=Fals
ig-
nore_if_missin
**args*,
***kwargs*)

Bases: *airflow.models.BaseOperator*

Deletes blob(s) on Azure Blob Storage.

Parameters

- **container_name** (*str*) – Name of the container. (templated)
- **blob_name** (*str*) – Name of the blob. (templated)
- **wasb_conn_id** (*str*) – Reference to the wasb connection.
- **check_options** – Optional keyword arguments that *WasbHook.check_for_blob()* takes.
- **is_prefix** (*bool*) – If blob_name is a prefix, delete all files matching prefix.
- **ignore_if_missing** (*bool*) – if True, then return success even if the blob does not exist.

template_fields = ['container_name', 'blob_name']

execute(*self*, *context*)

airflow.contrib.operators.winrm_operator

Module Contents

class airflow.contrib.operators.winrm_operator.**WinRMOperator**(*winrm_hook=None*, *ssh_conn_id=None*, *remote_host=None*, *command=None*, *timeout=10*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

WinRMOperator to execute commands on given remote host using the winrm_hook.

Parameters

- **winrm_hook** (airflow.contrib.hooks.winrm_hook.WinRMHook) – predefined ssh_hook to use for remote execution
- **ssh_conn_id** (*str*) – connection id from airflow Connections
- **remote_host** (*str*) – remote host to connect
- **command** (*str*) – command to execute on remote host. (templated)
- **timeout** (*int*) – timeout for executing the command.

template_fields = ['command']

execute (*self*, *context*)

airflow.contrib.sensors

Submodules

airflow.contrib.sensors.aws_athena_sensor

Module Contents

class airflow.contrib.sensors.aws_athena_sensor.**AthenaSensor** (*query_execution_id*, *max_retires=None*, *aws_conn_id='aws_default'*, *sleep_time=10*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Asks for the state of the Query until it reaches a failure state or success state. If it fails, failing the task.

Parameters

- **query_execution_id** (*str*) – query_execution_id to check the state of
- **max_retires** (*int*) – Number of times to poll for query state before returning the current state, defaults to None
- **aws_conn_id** (*str*) – aws connection to use, defaults to 'aws_default'
- **sleep_time** (*int*) – Time to wait between two consecutive call to check query status on athena, defaults to 10

INTERMEDIATE_STATES = ['QUEUED', 'RUNNING']

FAILURE_STATES = ['FAILED', 'CANCELLED']

SUCCESS_STATES = ['SUCCEEDED']

template_fields = ['query_execution_id']

template_ext = []

ui_color = #66c3ff

poke (*self*, *context*)

get_hook (*self*)

airflow.contrib.sensors.aws_glue_catalog_partition_sensor

Module Contents

class airflow.contrib.sensors.aws_glue_catalog_partition_sensor.**AwsGlueCatalogPartitionSens**

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a partition to show up in AWS Glue Catalog.

Parameters

- **table_name** (*str*) – The name of the table to wait for, supports the dot notation (my_database.my_table)
- **expression** (*str*) – The partition clause to wait for. This is passed as is to the AWS Glue Catalog API's get_partitions function, and supports SQL like notation as in `ds='2015-01-01' AND type='value'` and comparison operators as in `"ds>=2015-01-01"`. See https://docs.aws.amazon.com/glue/latest/dg/aws-glue-api-catalog-partitions.html #aws-glue-api-catalog-partitions-GetPartitions
- **aws_conn_id** (*str*) – ID of the Airflow connection where credentials and extra configuration are stored
- **region_name** (*str*) – Optional aws region name (example: us-east-1). Uses region from connection if not specified.
- **database_name** (*str*) – The name of the catalog database where the partitions reside.
- **poke_interval** (*int*) – Time in seconds that the job should wait in between each tries

template_fields = ['database_name', 'table_name', 'expression']

ui_color = #C5CAE9

poke (*self*, *context*)
Checks for existence of the partition in the AWS Glue Catalog table

get_hook (*self*)
Gets the AwsGlueCatalogHook

`airflow.contrib.sensors.aws_redshift_cluster_sensor`

Module Contents

class airflow.contrib.sensors.aws_redshift_cluster_sensor.**AwsRedshiftClusterSensor**(*cluster_ider tar-get_status= aws_conn_ *args, **kwargs*)

Bases: `airflow.sensors.base_sensor_operator.BaseSensorOperator`

Waits for a Redshift cluster to reach a specific status.

Parameters

- **cluster_identifier** (*str*) – The identifier for the cluster being pinged.
- **target_status** (*str*) – The cluster status desired.

template_fields = ['cluster_identifier', 'target_status']

poke(*self*, *context*)

`airflow.contrib.sensors.aws_sqs_sensor`

Module Contents

class airflow.contrib.sensors.aws_sqs_sensor.**SQSSensor**(*sqs_queue, aws_conn_id='aws_default', max_messages=5, wait_time_seconds=1, *args, **kwargs*)

Bases: `airflow.sensors.base_sensor_operator.BaseSensorOperator`

Get messages from an SQS queue and then deletes the message from the SQS queue. If deletion of messages fails an AirflowException is thrown otherwise, the message is pushed through XCom with the key `message`.

Parameters

- **aws_conn_id** (*str*) – AWS connection id
- **sqs_queue** (*str*) – The SQS queue url (templated)
- **max_messages** (*int*) – The maximum number of messages to retrieve for each poke (templated)
- **wait_time_seconds** (*int*) – The time in seconds to wait for receiving messages (default: 1 second)

template_fields = ['sqs_queue', 'max_messages']

poke(*self*, *context*)

Check for message on subscribed queue and write to xcom the message with key `messages`

Parameters **context** (*dict*) – the context object

Returns `True` if message is available or `False`

airflow.contrib.sensors.azure_cosmos_sensor

Module Contents

class airflow.contrib.sensors.azure_cosmos_sensor.**AzureCosmosDocumentSensor**(*database_name, collection_name, document_id, azure_cosmos_conn_i*

**args, **kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a document which matches the given query in CosmosDB. Example:

```
>>> azure_cosmos_sensor = AzureCosmosDocumentSensor(database_name="somedatabase_
↪name",
...                             collection_name="somecollection_name",
...                             document_id="unique-doc-id",
...                             azure_cosmos_conn_id="azure_cosmos_default",
...                             task_id="azure_cosmos_sensor")
```

template_fields = ['database_name', 'collection_name', 'document_id']

poke(*self, context*)

airflow.contrib.sensors.bash_sensor

Module Contents

class airflow.contrib.sensors.bash_sensor.**BashSensor**(*bash_command, env=None, output_encoding='utf-8', *args, **kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Executes a bash command/script and returns True if and only if the return code is 0.

Parameters

- **bash_command** (*str*) – The command, set of commands or reference to a bash script (must be '.sh') to be executed.
- **env** (*dict*) – If env is not None, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process environment, which is the default behavior. (templated)
- **output_encoding** (*str*) – output encoding of bash command.

template_fields = ['bash_command', 'env']

poke(*self, context*)

Execute the bash command in a temporary directory which will be cleaned afterwards

airflow.contrib.sensors.bigquery_sensor

This module contains a Google Bigquery sensor.

Module Contents

class airflow.contrib.sensors.bigquery_sensor.**BigQueryTableSensor**(*project_id*, *dataset_id*, *table_id*, *big-query_conn_id='google_cloud_defau*, *dele-gate_to=None*, **args*, ***kwargs*)

Bases: `airflow.sensors.base_sensor_operator.BaseSensorOperator`

Checks for the existence of a table in Google Bigquery.

Parameters

- **project_id** (*str*) – The Google cloud project in which to look for the table. The connection supplied to the hook must provide access to the specified project.
- **dataset_id** (*str*) – The name of the dataset in which to look for the table. storage bucket.
- **table_id** (*str*) – The name of the table to check the existence of.
- **bigquery_conn_id** (*str*) – The connection ID to use when connecting to Google BigQuery.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['project_id', 'dataset_id', 'table_id']

ui_color = #f0eee4

poke(*self*, *context*)

airflow.contrib.sensors.cassandra_record_sensor

Module Contents

class airflow.contrib.sensors.cassandra_record_sensor.**CassandraRecordSensor**(*table*, *keys*, *cas-san-dra_conn_id*, **args*, ***kwargs*)

Bases: `airflow.sensors.base_sensor_operator.BaseSensorOperator`

Checks for the existence of a record in a Cassandra cluster.

For example, if you want to wait for a record that has values ‘v1’ and ‘v2’ for each primary keys ‘p1’ and ‘p2’ to be populated in keyspace ‘k’ and table ‘t’, instantiate it as follows:

```
>>> cassandra_sensor = CassandraRecordSensor(table="k.t",
...                                           keys={"p1": "v1", "p2": "v2"},
...                                           cassandra_conn_id="cassandra_default
↪",
...                                           task_id="cassandra_sensor")
```

template_fields = ['table', 'keys']

poke (*self*, *context*)

airflow.contrib.sensors.cassandra_table_sensor

Module Contents

class airflow.contrib.sensors.cassandra_table_sensor.**CassandraTableSensor** (*table*, *cassandra_conn_id*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a table in a Cassandra cluster.

For example, if you want to wait for a table called 't' to be created in a keyspace 'k', instantiate it as follows:

```
>>> cassandra_sensor = CassandraTableSensor(table="k.t",
...                                          cassandra_conn_id="cassandra_default",
...                                          task_id="cassandra_sensor")
```

template_fields = ['table']

poke (*self*, *context*)

airflow.contrib.sensors.celery_queue_sensor

Module Contents

class airflow.contrib.sensors.celery_queue_sensor.**CeleryQueueSensor** (*celery_queue*, *target_task_id=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a Celery queue to be empty. By default, in order to be considered empty, the queue must not have any tasks in the `reserved`, `scheduled` or `active` states.

Parameters

- **celery_queue** (*str*) – The name of the Celery queue to wait for.
- **target_task_id** (*str*) – Task id for checking

_check_task_id (*self*, *context*)

Gets the returned Celery result from the Airflow task ID provided to the sensor, and returns True if the celery result has been finished execution.

Parameters **`context`** (*`dict`*) – Airflow's execution context

Returns True if task has been executed, otherwise False

Return type bool

`poke` (*self*, *context*)

`airflow.contrib.sensors.datadog_sensor`

Module Contents

class `airflow.contrib.sensors.datadog_sensor.`**`DatadogSensor`**(*datadog_conn_id='datadog_default'*, *from_seconds_ago=3600*, *up_to_seconds_from_now=0*, *priority=None*, *sources=None*, *tags=None*, *response_check=None*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

A sensor to listen, with a filter, to datadog event streams and determine if some event was emitted.

Depends on the datadog API, which has to be deployed on the same server where Airflow runs.

Parameters

- **`datadog_conn_id`** – The connection to datadog, containing metadata for api keys.
- **`datadog_conn_id`** – str

`ui_color = #66c3dd`

`poke` (*self*, *context*)

`airflow.contrib.sensors.emr_base_sensor`

Module Contents

class `airflow.contrib.sensors.emr_base_sensor.`**`EmrBaseSensor`**(*aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

Contains general sensor behavior for EMR. Subclasses should implement get_emr_response() and state_from_response() methods. Subclasses should also implement NON_TERMINAL_STATES and FAILED_STATE constants.

`ui_color = #66c3ff`

`poke` (*self*, *context*)

airflow.contrib.sensors.emr_job_flow_sensor

Module Contents

class airflow.contrib.sensors.emr_job_flow_sensor.**EmrJobFlowSensor**(*job_flow_id*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor*

Asks for the state of the JobFlow until it reaches a terminal state. If it fails the sensor errors, failing the task.

Parameters **job_flow_id** (*str*) – job_flow_id to check the state of

NON_TERMINAL_STATES = ['STARTING', 'BOOTSTRAPPING', 'RUNNING', 'WAITING', 'TERMINATING

FAILED_STATE = ['TERMINATED_WITH_ERRORS']

template_fields = ['job_flow_id']

template_ext = []

get_emr_response(*self*)

static state_from_response(*response*)

airflow.contrib.sensors.emr_step_sensor

Module Contents

class airflow.contrib.sensors.emr_step_sensor.**EmrStepSensor**(*job_flow_id*, *step_id*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor*

Asks for the state of the step until it reaches a terminal state. If it fails the sensor errors, failing the task.

Parameters

- **job_flow_id** (*str*) – job_flow_id which contains the step check the state of
- **step_id** (*str*) – step to check the state of

NON_TERMINAL_STATES = ['PENDING', 'RUNNING', 'CONTINUE', 'CANCEL_PENDING']

FAILED_STATE = ['CANCELLED', 'FAILED', 'INTERRUPTED']

template_fields = ['job_flow_id', 'step_id']

template_ext = []

get_emr_response(*self*)

static state_from_response(*response*)

airflow.contrib.sensors.file_sensor

Module Contents

class airflow.contrib.sensors.file_sensor.**FileSensor**(*filepath*, *fs_conn_id='fs_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a file or folder to land in a filesystem.

If the path given is a directory then this sensor will only return true if any files exist inside it (either directly, or within a subdirectory)

Parameters

- **`fs_conn_id`** (*`str`*) – reference to the File (path) connection id
- **`filepath`** – File or folder name (relative to the base path set within the connection)

`template_fields = ['filepath']`

`ui_color = #91818a`

`poke` (*self*, *context*)

`airflow.contrib.sensors.ftp_sensor`

Module Contents

`class` `airflow.contrib.sensors.ftp_sensor.`**`FTPSensor`**(*path*, *ftp_conn_id='ftp_default'*, *fail_on_transient_errors=True*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

Waits for a file or directory to be present on FTP.

`template_fields = ['path']`

Errors that are transient in nature, and where action can be retried

`transient_errors = [421, 425, 426, 434, 450, 451, 452]`

`error_code_pattern`

`_create_hook`(*self*)

Return connection hook.

`_get_error_code`(*self*, *e*)

Extract error code from ftp exception

`poke`(*self*, *context*)

`class` `airflow.contrib.sensors.ftp_sensor.`**`FTPSSensor`**

Bases: *`airflow.contrib.sensors.ftp_sensor.FTPSensor`*

Waits for a file or directory to be present on FTP over SSL.

`_create_hook`(*self*)

Return connection hook.

`airflow.contrib.sensors.gcp_transfer_sensor`

This module contains a Google Cloud Transfer sensor.

Module Contents

class airflow.contrib.sensors.gcp_transfer_sensor.**GCPTransferServiceWaitForJobStatusSensor**

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for at least one operation belonging to the job to have the expected status.

Parameters

- **job_name** (*str*) – The name of the transfer job
- **expected_statuses** (*set[str] or string*) – The expected state of the operation. See: https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferOperations#Status
- **project_id** (*str*) – (Optional) the ID of the project that owns the Transfer Job. If set to None or missing, the default project_id from the GCP connection is used.
- **gcp_conn_id** (*str*) – The connection ID used to connect to Google Cloud Platform.

template_fields = ['job_name']

poke (*self*, *context*)

airflow.contrib.sensors.gcs_sensor

This module contains Google Cloud Storage sensors.

Module Contents

class airflow.contrib.sensors.gcs_sensor.**GoogleCloudStorageObjectSensor** (*bucket*, *ob-ject*, *google_cloud_conn_id='goog*, *del-e-gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a file in Google Cloud Storage.

Parameters

- **bucket** (*str*) – The Google cloud storage bucket where the object is.
- **object** (*str*) – The name of the object to check in the Google cloud storage bucket.
- **google_cloud_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['bucket', 'object']

ui_color = #f0eee4

poke(*self*, *context*)

airflow.contrib.sensors.gcs_sensor.**ts_function**(*context*)

Default callback for the GoogleCloudStorageObjectUpdatedSensor. The default behaviour is check for the object being updated after execution_date + schedule_interval.

class airflow.contrib.sensors.gcs_sensor.**GoogleCloudStorageObjectUpdatedSensor**(*bucket*, *ob-ject*, *ts_func=ts_functi* *google_cloud_con* *del-e-gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks if an object is updated in Google Cloud Storage.

Parameters

- **bucket** (*str*) – The Google cloud storage bucket where the object is.
- **object** (*str*) – The name of the object to download in the Google cloud storage bucket.
- **ts_func** (*function*) – Callback for defining the update condition. The default callback returns execution_date + schedule_interval. The callback takes the context as parameter.
- **google_cloud_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['bucket', 'object']

ui_color = #f0eee4

poke(*self*, *context*)

class airflow.contrib.sensors.gcs_sensor.**GoogleCloudStoragePrefixSensor**(*bucket*, *pre-fix*, *google_cloud_conn_id='goo* *del-e-gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a objects at prefix in Google Cloud Storage bucket.

Parameters

- **bucket** (*str*) – The Google cloud storage bucket where the object is.
- **prefix** (*str*) – The name of the prefix to check in the Google cloud storage bucket.

- **google_cloud_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['bucket', 'prefix']

ui_color = #f0eee4

poke(*self*, *context*)

airflow.contrib.sensors.gcs_sensor.**get_time**()
This is just a wrapper of datetime.datetime.now to simplify mocking in the unittests.

class airflow.contrib.sensors.gcs_sensor.**GoogleCloudStorageUploadSessionCompleteSensor**(*bucke pre- fix, in- ac- tiv- ity_p * 60, min_ pre- vi- ous_r al- low_ googl del- e- gate_ *args **kw*

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for changes in the number of objects at prefix in Google Cloud Storage bucket and returns True if the inactivity period has passed with no increase in the number of objects. Note, it is recommended to use reschedule mode if you expect this sensor to run for hours.

Parameters

- **bucket** (*str*) – The Google cloud storage bucket where the objects are. expected.
- **prefix** – The name of the prefix to check in the Google cloud storage bucket.
- **inactivity_period** (*float*) – The total seconds of inactivity to designate an upload session is over. Note, this mechanism is not real time and this operator may not return until a poke_interval after this period has passed with no additional objects sensed.
- **min_objects** (*int*) – The minimum number of objects needed for upload session to be considered valid.
- **previous_num_objects** (*int*) – The number of objects found during the last poke.
- **inactivity_seconds** (*float*) – The current seconds of the inactivity period.
- **allow_delete** (*bool*) – Should this sensor consider objects being deleted between pokes valid behavior. If true a warning message will be logged when this happens. If false an error will be raised.

- **google_cloud_conn_id** (*str*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

template_fields = ['bucket', 'prefix']

ui_color = #f0eee4

is_bucket_updated (*self*, *current_num_objects*)
Checks whether new objects have been uploaded and the inactivity_period has passed and updates the state of the sensor accordingly.

Parameters **current_num_objects** (*int*) – number of objects in bucket during last poke.

poke (*self*, *context*)

airflow.contrib.sensors.hdfs_sensor

Module Contents

class airflow.contrib.sensors.hdfs_sensor.**HdfsSensorRegex** (*regex*, **args*, ***kwargs*)
Bases: *airflow.sensors.hdfs_sensor.HdfsSensor*

poke (*self*, *context*)
poke matching files in a directory with self.regex

Returns Bool depending on the search criteria

class airflow.contrib.sensors.hdfs_sensor.**HdfsSensorFolder** (*be_empty=False*, **args*, ***kwargs*)
Bases: *airflow.sensors.hdfs_sensor.HdfsSensor*

poke (*self*, *context*)
poke for a non empty directory

Returns Bool depending on the search criteria

airflow.contrib.sensors.imap_attachment_sensor

Module Contents

class airflow.contrib.sensors.imap_attachment_sensor.**ImapAttachmentSensor** (*attachment_name*, *mail_folder='INBOX'*, *check_regex=False*, *conn_id='imap_default'*, **args*, ***kwargs*)
Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a specific attachment on a mail server.

Parameters

- **attachment_name** (*str*) – The name of the attachment that will be checked.
- **check_regex** (*bool*) – If set to True the attachment's name will be parsed as regular expression. Through this you can get a broader set of attachments that it will look for than just only the equality of the attachment name. The default value is False.

- **`mail_folder`** (*`str`*) – The mail folder in where to search for the attachment. The default value is 'INBOX'.
- **`conn_id`** (*`str`*) – The connection to run the sensor against. The default value is 'imap_default'.

`template_fields = ['attachment_name']`

`poke`(*self*, *context*)

Pokes for a mail attachment on the mail server.

Parameters **`context`** (*`dict`*) – The context that is being provided when poking.

Returns True if attachment with the given name is present and False if not.

Return type bool

`airflow.contrib.sensors.jira_sensor`

Module Contents

class `airflow.contrib.sensors.jira_sensor.`**`JiraSensor`**(*jira_conn_id='jira_default'*, *method_name=None*, *method_params=None*, *result_processor=None*, **args*, ***kwargs*)

Bases: *`airflow.sensors.base_sensor_operator.BaseSensorOperator`*

Monitors a jira ticket for any change.

Parameters

- **`jira_conn_id`** (*`str`*) – reference to a pre-defined Jira Connection
- **`method_name`** (*`str`*) – method name from jira-python-sdk to be execute
- **`method_params`** (*`dict`*) – parameters for the method method_name
- **`result_processor`** (*`function`*) – function that return boolean and act as a sensor response

`poke`(*self*, *context*)

class `airflow.contrib.sensors.jira_sensor.`**`JiraTicketSensor`**(*jira_conn_id='jira_default'*, *ticket_id=None*, *field=None*, *expected_value=None*, *field_checker_func=None*, **args*, ***kwargs*)

Bases: *`airflow.contrib.sensors.jira_sensor.JiraSensor`*

Monitors a jira ticket for given change in terms of function.

Parameters

- **`jira_conn_id`** (*`str`*) – reference to a pre-defined Jira Connection
- **`ticket_id`** (*`str`*) – id of the ticket to be monitored
- **`field`** (*`str`*) – field of the ticket to be monitored
- **`expected_value`** (*`str`*) – expected value of the field
- **`result_processor`** (*`function`*) – function that return boolean and act as a sensor response

template_fields = ['ticket_id']

poke (*self*, *context*)

issue_field_checker (*self*, *context*, *issue*)

airflow.contrib.sensors.mongo_sensor

Module Contents

class airflow.contrib.sensors.mongo_sensor.**MongoSensor** (*collection*, *query*, *mongo_conn_id='mongo_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a document which matches the given query in MongoDB. Example:

```
>>> mongo_sensor = MongoSensor(collection="coll",
...                            query={"key": "value"},
...                            mongo_conn_id="mongo_default",
...                            task_id="mongo_sensor")
```

template_fields = ['collection', 'query']

poke (*self*, *context*)

airflow.contrib.sensors.pubsub_sensor

This module contains a Google PubSub sensor.

Module Contents

class airflow.contrib.sensors.pubsub_sensor.**PubSubPullSensor** (*project*, *subscription*, *max_messages=5*, *return_immediately=False*, *ack_messages=False*, *gcp_conn_id='google_cloud_default'*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Pulls messages from a PubSub subscription and passes them through XCom.

This sensor operator will pull up to `max_messages` messages from the specified PubSub subscription. When the subscription returns messages, the poke method's criteria will be fulfilled and the messages will be returned from the operator and passed through XCom for downstream tasks.

If `ack_messages` is set to True, messages will be immediately acknowledged before being returned, otherwise, downstream tasks will be responsible for acknowledging them.

`project` and `subscription` are templated so you can use variables in them.

template_fields = ['project', 'subscription']

ui_color = #ff7f50

execute (*self*, *context*)

Overridden to allow messages to be passed

poke (*self*, *context*)

airflow.contrib.sensors.python_sensor

Module Contents

class airflow.contrib.sensors.python_sensor.**PythonSensor** (*python_callable*, *op_args=None*, *op_kwargs=None*, *provide_context=False*, *templates_dict=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a Python callable to return True.

User could put input argument in templates_dict e.g `templates_dict = {'start_ds': 1970}` and access the argument by calling `kwargs['templates_dict']['start_ds']` in the the callable

Parameters

- **python_callable** (*python callable*) – A reference to an object that is callable
- **op_kwargs** (*dict*) – a dictionary of keyword arguments that will get unpacked in your function
- **op_args** (*list*) – a list of positional arguments that will get unpacked when calling your callable
- **provide_context** (*bool*) – if set to true, Airflow will pass a set of keyword arguments that can be used in your function. This set of kwargs correspond exactly to what you can use in your jinja templates. For this to work, you need to define ***kwargs* in your function header.
- **templates_dict** (*dict of str*) – a dictionary where the values are templates that will get templated by the Airflow engine sometime between `__init__` and `execute` takes place and are made available in your callable's context after the template has been applied.

template_fields = ['templates_dict']

poke (*self*, *context*)

airflow.contrib.sensors.qubole_sensor

Module Contents

class airflow.contrib.sensors.qubole_sensor.**QuboleSensor** (*data*, *qubole_conn_id='qubole_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Base class for all Qubole Sensors

template_fields = ['data', 'qubole_conn_id']

template_ext = ['.txt']

poke (*self*, *context*)

class airflow.contrib.sensors.qubole_sensor.**QuboleFileSensor** (**args*, ***kwargs*)

Bases: *airflow.contrib.sensors.qubole_sensor.QuboleSensor*

Wait for a file or folder to be present in cloud storage and check for its presence via QDS APIs

Parameters

- **qubole_conn_id** (*str*) – Connection id which consists of qds auth_token
- **data** (*a JSON object*) – a JSON object containing payload, whose presence needs to be checked Check this example for sample payload structure.

Note: Both `data` and `qubole_conn_id` fields support templating. You can also use `.txt` files for template-driven use cases.

class airflow.contrib.sensors.qubole_sensor.**QubolePartitionSensor**(**args*, ***kwargs*)

Bases: *airflow.contrib.sensors.qubole_sensor.QuboleSensor*

Wait for a Hive partition to show up in QHS (Qubole Hive Service) and check for its presence via QDS APIs

Parameters

- **qubole_conn_id** (*str*) – Connection id which consists of qds auth_token
- **data** (*a JSON object*) – a JSON object containing payload, whose presence needs to be checked. Check this example for sample payload structure.

Note: Both `data` and `qubole_conn_id` fields support templating. You can also use `.txt` files for template-driven use cases.

airflow.contrib.sensors.redis_key_sensor

Module Contents

class airflow.contrib.sensors.redis_key_sensor.**RedisKeySensor**(*key*, *redis_conn_id*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a key in a Redis

template_fields = ['key']

ui_color = #f0eee4

poke(*self*, *context*)

airflow.contrib.sensors.redis_pub_sub_sensor

Module Contents

class airflow.contrib.sensors.redis_pub_sub_sensor.**RedisPubSubSensor**(*channels*, *redis_conn_id*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Redis sensor for reading a message from pub sub channels

template_fields = ['channels']

ui_color = #f0eee4

poke (*self*, *context*)

Check for message on subscribed channels and write to xcom the message with key `message`

An example of message `{'type': 'message', 'pattern': None, 'channel': b'test', 'data': b'hello'}`

Parameters **context** (*dict*) – the context object

Returns `True` if message (with type 'message') is available or `False` if not

airflow.contrib.sensors.sagemaker_base_sensor

Module Contents

class airflow.contrib.sensors.sagemaker_base_sensor.**SageMakerBaseSensor** (*aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Contains general sensor behavior for SageMaker. Subclasses should implement get_sagemaker_response() and state_from_response() methods. Subclasses should also implement NON_TERMINAL_STATES and FAILED_STATE methods.

ui_color = #ededed

poke (*self*, *context*)

non_terminal_states (*self*)

failed_states (*self*)

get_sagemaker_response (*self*)

get_failed_reason_from_response (*self*, *response*)

state_from_response (*self*, *response*)

airflow.contrib.sensors.sagemaker_endpoint_sensor

Module Contents

class airflow.contrib.sensors.sagemaker_endpoint_sensor.**SageMakerEndpointSensor** (*endpoint_name*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor*

Asks for the state of the endpoint state until it reaches a terminal state. If it fails the sensor errors, the task fails.

Parameters **job_name** (*str*) – job_name of the endpoint instance to check the state of

template_fields = ['endpoint_name']

template_ext = []

non_terminal_states (*self*)

failed_states (*self*)

get_sagemaker_response (*self*)

get_failed_reason_from_response (*self*, *response*)

state_from_response (*self*, *response*)

airflow.contrib.sensors.sagemaker_training_sensor

Module Contents

class airflow.contrib.sensors.sagemaker_training_sensor.**SageMakerTrainingSensor** (*job_name*, *print_log=True*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor*

Asks for the state of the training state until it reaches a terminal state. If it fails the sensor errors, failing the task.

Parameters

- **job_name** (*str*) – name of the SageMaker training job to check the state of
- **print_log** (*bool*) – if the operator should print the cloudwatch log

template_fields = ['job_name']

template_ext = []

init_log_resource (*self*, *hook*)

non_terminal_states (*self*)

failed_states (*self*)

get_sagemaker_response (*self*)

get_failed_reason_from_response (*self*, *response*)

state_from_response (*self*, *response*)

airflow.contrib.sensors.sagemaker_transform_sensor

Module Contents

class airflow.contrib.sensors.sagemaker_transform_sensor.**SageMakerTransformSensor** (*job_name*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor*

Asks for the state of the transform state until it reaches a terminal state. The sensor will error if the job errors, throwing a AirflowException containing the failure reason.

Parameters **job_name** (*string*) – job_name of the transform job instance to check the state of

template_fields = ['job_name']

template_ext = []

non_terminal_states (*self*)

failed_states (*self*)

get_sagemaker_response (*self*)

get_failed_reason_from_response (*self*, *response*)

state_from_response (*self*, *response*)

airflow.contrib.sensors.sagemaker_tuning_sensor

Module Contents

class airflow.contrib.sensors.sagemaker_tuning_sensor.**SageMakerTuningSensor**(*job_name*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor*

Asks for the state of the tuning state until it reaches a terminal state. The sensor will error if the job errors, throwing a AirflowException containing the failure reason.

Parameters **job_name** (*str*) – job_name of the tuning instance to check the state of

template_fields = ['job_name']

template_ext = []

non_terminal_states(*self*)

failed_states(*self*)

get_sagemaker_response(*self*)

get_failed_reason_from_response(*self*, *response*)

state_from_response(*self*, *response*)

airflow.contrib.sensors.sftp_sensor

Module Contents

class airflow.contrib.sensors.sftp_sensor.**SFTPSensor**(*path*, *sftp_conn_id='sftp_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a file or directory to be present on SFTP.

Parameters

- **path** (*str*) – Remote file or directory path
- **sftp_conn_id** (*str*) – The connection to run the sensor against

template_fields = ['path']

poke(*self*, *context*)

airflow.contrib.sensors.wasb_sensor

Module Contents

class airflow.contrib.sensors.wasb_sensor.**WasbBlobSensor**(*container_name*, *blob_name*, *wasb_conn_id='wasb_default'*, *check_options=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a blob to arrive on Azure Blob Storage.

Parameters

- **container_name** (*str*) – Name of the container.
- **blob_name** (*str*) – Name of the blob.
- **wasb_conn_id** (*str*) – Reference to the wasb connection.
- **check_options** (*dict*) – Optional keyword arguments that *WasbHook.check_for_blob()* takes.

template_fields = ['container_name', 'blob_name']

poke (*self*, *context*)

class airflow.contrib.sensors.wasb_sensor.**WasbPrefixSensor** (*container_name*, *prefix*, *wasb_conn_id='wasb_default'*, *check_options=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for blobs matching a prefix to arrive on Azure Blob Storage.

Parameters

- **container_name** (*str*) – Name of the container.
- **prefix** (*str*) – Prefix of the blob.
- **wasb_conn_id** (*str*) – Reference to the wasb connection.
- **check_options** (*dict*) – Optional keyword arguments that *WasbHook.check_for_prefix()* takes.

template_fields = ['container_name', 'prefix']

poke (*self*, *context*)

airflow.contrib.sensors.weekday_sensor

Module Contents

class airflow.contrib.sensors.weekday_sensor.**DayOfWeekSensor** (*week_day*, *use_task_execution_day=False*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits until the first specified day of the week. For example, if the execution day of the task is '2018-12-22' (Saturday) and you pass 'FRIDAY', the task will wait until next Friday.

Example (with single day):

```
weekend_check = DayOfWeekSensor(
    task_id='weekend_check',
    week_day='Saturday',
    use_task_execution_day=True,
    dag=dag)
```

Example (with multiple day using set):

```
weekend_check = DayOfWeekSensor(
    task_id='weekend_check',
    week_day={'Saturday', 'Sunday'},
    use_task_execution_day=True,
    dag=dag)
```

Example (with `WeekDay` enum):

```
# import WeekDay Enum
from airflow.contrib.utils.weekday import WeekDay

weekend_check = DayOfWeekSensor(
    task_id='weekend_check',
    week_day={WeekDay.SATURDAY, WeekDay.SUNDAY},
    use_task_execution_day=True,
    dag=dag)
```

Parameters

- **`week_day`** (*`set or str or airflow.contrib.utils.weekday.WeekDay`*) – Day of the week to check (full name). Optionally, a set of days can also be provided using a set. Example values:
  - `"MONDAY"`,
  - `{"Saturday", "Sunday"}`
  - `{WeekDay.TUESDAY}`
  - `{WeekDay.SATURDAY, WeekDay.SUNDAY}`
- **`use_task_execution_day`** (*`bool`*) – If `True`, uses task's execution day to compare with week_day. Execution Date is Useful for backfilling. If `False`, uses system's day of the week. Useful when you don't want to run anything on weekdays on the system.

`poke`(*self*, *context*)

3.22.2 Hooks

Hooks are interfaces to external platforms and databases, implementing a common interface when possible and acting as building blocks for operators. All hooks are derived from *`BaseHook`*.

3.22.2.1 Hooks packages

All hooks are in the following packages:

`airflow.hooks`

Submodules

`airflow.hooks.S3_hook`

Module Contents

class airflow.hooks.S3_hook.**S3Hook**

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS S3, using the boto3 library.

get_conn(*self*)

static parse_s3_url(*s3url*)

check_for_bucket(*self*, *bucket_name*)

Check if bucket_name exists.

Parameters **bucket_name** (*str*) – the name of the bucket

get_bucket(*self*, *bucket_name*)

Returns a boto3.S3.Bucket object

Parameters **bucket_name** (*str*) – the name of the bucket

create_bucket(*self*, *bucket_name*, *region_name=None*)

Creates an Amazon S3 bucket.

Parameters

- **bucket_name** (*str*) – The name of the bucket
- **region_name** (*str*) – The name of the aws region in which to create the bucket.

check_for_prefix(*self*, *bucket_name*, *prefix*, *delimiter*)

Checks that a prefix exists in a bucket

Parameters

- **bucket_name** (*str*) – the name of the bucket
- **prefix** (*str*) – a key prefix
- **delimiter** (*str*) – the delimiter marks key hierarchy.

list_prefixes(*self*, *bucket_name*, *prefix=''*, *delimiter=''*, *page_size=None*, *max_items=None*)

Lists prefixes in a bucket under prefix

Parameters

- **bucket_name** (*str*) – the name of the bucket
- **prefix** (*str*) – a key prefix
- **delimiter** (*str*) – the delimiter marks key hierarchy.
- **page_size** (*int*) – pagination size
- **max_items** (*int*) – maximum items to return

list_keys(*self*, *bucket_name*, *prefix=''*, *delimiter=''*, *page_size=None*, *max_items=None*)

Lists keys in a bucket under prefix and not containing delimiter

Parameters

- **bucket_name** (*str*) – the name of the bucket

- **prefix** (*str*) – a key prefix
- **delimiter** (*str*) – the delimiter marks key hierarchy.
- **page_size** (*int*) – pagination size
- **max_items** (*int*) – maximum items to return

check_for_key (*self*, *key*, *bucket_name=None*)
Checks if a key exists in a bucket

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored

get_key (*self*, *key*, *bucket_name=None*)
Returns a boto3.s3.Object

Parameters

- **key** (*str*) – the path to the key
- **bucket_name** (*str*) – the name of the bucket

read_key (*self*, *key*, *bucket_name=None*)
Reads a key from S3

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored

select_key (*self*, *key*, *bucket_name=None*, *expression='SELECT * FROM S3Object'*, *expression_type='SQL'*, *input_serialization=None*, *output_serialization=None*)
Reads a key with S3 Select.

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored
- **expression** (*str*) – S3 Select expression
- **expression_type** (*str*) – S3 Select expression type
- **input_serialization** (*dict*) – S3 Select input data serialization format
- **output_serialization** (*dict*) – S3 Select output data serialization format

Returns retrieved subset of original data by S3 Select

Return type str

See also:

For more details about S3 Select parameters: http://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.select_object_content

check_for_wildcard_key (*self*, *wildcard_key*, *bucket_name=None*, *delimiter=''*)
Checks that a key matching a wildcard expression exists in a bucket

Parameters

- **wildcard_key** (*str*) – the path to the key
- **bucket_name** (*str*) – the name of the bucket
- **delimiter** (*str*) – the delimiter marks key hierarchy

get_wildcard_key (*self*, *wildcard_key*, *bucket_name=None*, *delimiter=''*)
Returns a boto3.s3.Object object matching the wildcard expression

Parameters

- **wildcard_key** (*str*) – the path to the key
- **bucket_name** (*str*) – the name of the bucket
- **delimiter** (*str*) – the delimiter marks key hierarchy

load_file (*self*, *filename*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*)
Loads a local file to S3

Parameters

- **filename** (*str*) – name of the file to load.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists. If replace is False and the key exists, an error will be raised.
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

load_string (*self*, *string_data*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*, *encoding='utf-8'*)
Loads a string to S3

This is provided as a convenience to drop a string in S3. It uses the boto infrastructure to ship a file to s3.

Parameters

- **string_data** (*str*) – str to set as content for the key.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

load_bytes (*self*, *bytes_data*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*)
Loads bytes to S3

This is provided as a convenience to drop a string in S3. It uses the boto infrastructure to ship a file to s3.

Parameters

- **bytes_data** (*bytes*) – bytes to set as content for the key.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

load_file_obj (*self*, *file_obj*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*)
Loads a file object to S3

Parameters

- **file_obj** (*file-like object*) – The file-like object to set as the content for the S3 key.

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag that indicates whether to overwrite the key if it already exists.
- **encrypt** (*bool*) – If True, S3 encrypts the file on the server, and the file is stored in encrypted form at rest in S3.

copy_object(*self*, *source_bucket_key*, *dest_bucket_key*, *source_bucket_name=None*, *dest_bucket_name=None*, *source_version_id=None*)

Creates a copy of an object that is already stored in S3.

Note: the S3 connection used here needs to have access to both source and destination bucket/key.

Parameters

- **source_bucket_key** (*str*) – The key of the source object.

  It can be either full s3:// style url or relative path from root level.

  When it's specified as a full s3:// url, please omit source_bucket_name.
- **dest_bucket_key** (*str*) – The key of the object to copy to.

  The convention to specify *dest_bucket_key* is the same as *source_bucket_key*.
- **source_bucket_name** (*str*) – Name of the S3 bucket where the source object is in.

  It should be omitted when *source_bucket_key* is provided as a full s3:// url.
- **dest_bucket_name** (*str*) – Name of the S3 bucket to where the object is copied.

  It should be omitted when *dest_bucket_key* is provided as a full s3:// url.
- **source_version_id** (*str*) – Version ID of the source object (OPTIONAL)

delete_objects(*self*, *bucket*, *keys*)

Parameters

- **bucket** (*str*) – Name of the bucket in which you are going to delete object(s)
- **keys** (*str or list*) – The key(s) to delete from S3 bucket.

  When `keys` is a string, it's supposed to be the key name of the single object to delete.

  When `keys` is a list, it's supposed to be the list of the keys to delete.

airflow.hooks.base_hook

Module Contents

airflow.hooks.base_hook.**CONN_ENV_PREFIX = AIRFLOW_CONN_**

class airflow.hooks.base_hook.**BaseHook**(*source*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

Abstract base class for hooks, hooks are meant as an interface to interact with external systems. MySqlHook, HiveHook, PigHook return object that can handle the connection and interaction to specific instances of these systems, and expose consistent methods to interact with them.

classmethod _get_connections_from_db(*cls*, *conn_id*, *session=None*)

classmethod _get_connection_from_env(*cls*, *conn_id*)

classmethod get_connections(*cls*, *conn_id:str*)

classmethod get_connection(*cls*, *conn_id:str*)

classmethod get_hook(*cls*, *conn_id:str*)

get_conn(*self*)

get_records(*self*, *sql*)

get_pandas_df(*self*, *sql*)

run(*self*, *sql*)

airflow.hooks.dbapi_hook

Module Contents

class airflow.hooks.dbapi_hook.**DbApiHook**(**args*, ***kwargs*)

Bases: *airflow.hooks.base_hook.BaseHook*

Abstract base class for sql hooks.

conn_name_attr :Optional[str]

default_conn_name = default_conn_id

supports_autocommit = False

connector

get_conn(*self*)

Returns a connection object

get_uri(*self*)

get_sqlalchemy_engine(*self*, *engine_kwargs=None*)

get_pandas_df(*self*, *sql*, *parameters=None*)

Executes the sql and returns a pandas dataframe

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

get_records(*self*, *sql*, *parameters=None*)

Executes the sql and returns a set of records.

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

get_first(*self*, *sql*, *parameters=None*)

Executes the sql and returns the first resulting row.

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

run (*self*, *sql*, *autocommit=False*, *parameters=None*)
Runs a command or a list of commands. Pass a list of sql statements to the sql parameter to get them to execute sequentially

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **autocommit** (*bool*) – What to set the connection's autocommit setting to before executing the query.
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

set_autocommit (*self*, *conn*, *autocommit*)
Sets the autocommit flag on the connection

get_autocommit (*self*, *conn*)
Get autocommit setting for the provided connection. Return True if conn.autocommit is set to True. Return False if conn.autocommit is not set or set to False or conn does not support autocommit.

Parameters **conn** (*connection object.*) – Connection to get autocommit setting from.

Returns connection autocommit setting.

Return type bool

get_cursor (*self*)
Returns a cursor

insert_rows (*self*, *table*, *rows*, *target_fields=None*, *commit_every=1000*, *replace=False*)
A generic way to insert a set of tuples into a table, a new transaction is created every commit_every rows

Parameters

- **table** (*str*) – Name of the target table
- **rows** (*iterable of tuples*) – The rows to insert into the table
- **target_fields** (*iterable of strings*) – The names of the columns to fill in the table
- **commit_every** (*int*) – The maximum number of rows to insert in one transaction. Set to 0 to insert all rows in one transaction.
- **replace** (*bool*) – Whether to replace instead of insert

static _serialize_cell (*cell*, *conn=None*)
Returns the SQL literal of the cell as a string.

Parameters

- **cell** (*object*) – The cell to insert into the table
- **conn** (*connection object*) – The database connection

Returns The serialized cell

Return type str

bulk_dump (*self*, *table*, *tmp_file*)
Dumps a database table into a tab-delimited file

Parameters

- **table** (*str*) – The name of the source table
- **tmp_file** (*str*) – The path of the target file

bulk_load (*self*, *table*, *tmp_file*)
Loads a tab-delimited file into a database table

Parameters

- **table** (*str*) – The name of the target table
- **tmp_file** (*str*) – The path of the file to load into the table

airflow.hooks.docker_hook

Module Contents

class airflow.hooks.docker_hook.**DockerHook** (*docker_conn_id='docker_default'*, *base_url=None*, *version=None*, *tls=None*)
Bases: *airflow.hooks.base_hook.BaseHook*, airflow.utils.log.logging_mixin.LoggingMixin

Interact with a private Docker registry.

Parameters **docker_conn_id** (*str*) – ID of the Airflow connection where credentials and extra configuration are stored

get_conn (*self*)

__login (*self*, *client*)

airflow.hooks.druid_hook

Module Contents

class airflow.hooks.druid_hook.**DruidHook** (*druid_ingest_conn_id='druid_ingest_default'*, *timeout=1*, *max_ingestion_time=None*)
Bases: *airflow.hooks.base_hook.BaseHook*

Connection to Druid overlord for ingestion

Parameters

- **druid_ingest_conn_id** (*str*) – The connection id to the Druid overlord machine which accepts index jobs
- **timeout** (*int*) – The interval between polling the Druid job for the status of the ingestion job. Must be greater than or equal to 1
- **max_ingestion_time** (*int*) – The maximum ingestion time before assuming the job failed

get_conn_url (*self*)

submit_indexing_job (*self*, *json_index_spec*)

class airflow.hooks.druid_hook.**DruidDbApiHook** (**args*, ***kwargs*)
Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with Druid broker

This hook is purely for users to query druid broker. For ingestion, please use druidHook.

conn_name_attr = druid_broker_conn_id

default_conn_name = druid_broker_default

supports_autocommit = False

get_conn (*self*)
Establish a connection to druid broker.

get_uri (*self*)
Get the connection uri for druid broker.

e.g: druid://localhost:8082/druid/v2/sql/

set_autocommit (*self*, *conn*, *autocommit*)

get_pandas_df (*self*, *sql*, *parameters=None*)

insert_rows (*self*, *table*, *rows*, *target_fields=None*, *commit_every=1000*)

airflow.hooks.hdfs_hook

Module Contents

airflow.hooks.hdfs_hook.**snakebite_loaded = True**

exception airflow.hooks.hdfs_hook.**HDFSHookException**
Bases: airflow.exceptions.AirflowException

class airflow.hooks.hdfs_hook.**HDFSHook** (*hdfs_conn_id='hdfs_default'*, *proxy_user=None*, *autoconfig=False*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interact with HDFS. This class is a wrapper around the snakebite library.

Parameters

- **hdfs_conn_id** (*str*) – Connection id to fetch connection info
- **proxy_user** (*str*) – effective user for HDFS operations
- **autoconfig** (*bool*) – use snakebite's automatically configured client

get_conn (*self*)
Returns a snakebite HDFSClient object.

airflow.hooks.hive_hooks

Module Contents

airflow.hooks.hive_hooks.**HIVE_QUEUE_PRIORITIES = ['VERY_HIGH', 'HIGH', 'NORMAL', 'LOW', 'VI**

airflow.hooks.hive_hooks.**get_context_from_env_var** ()
Extract context from env variable, e.g. dag_id, task_id and execution_date, so that they can be used inside BashOperator and PythonOperator.

Returns The context of interest.

class airflow.hooks.hive_hooks.**HiveCliHook** (*hive_cli_conn_id='hive_cli_default'*, *run_as=None*, *mapred_queue=None*, *mapred_queue_priority=None*, *mapred_job_name=None*)
Bases: *airflow.hooks.base_hook.BaseHook*

Simple wrapper around the hive CLI.

It also supports the `beeline` a lighter CLI that runs JDBC and is replacing the heavier traditional CLI. To enable `beeline`, set the use_beeline param in the extra field of your connection as in `{ "use_beeline": true }`

Note that you can also set default hive CLI parameters using the `hive_cli_params` to be used in your connection as in `{"hive_cli_params": "-hiveconf mapred.job.tracker=some.jobtracker:444"}` Parameters passed here can be overridden by run_cli's hive_conf param

The extra connection parameter `auth` gets passed as in the `jdbc` connection string as is.

Parameters

- **mapred_queue** (*str*) – queue used by the Hadoop Scheduler (Capacity or Fair)
- **mapred_queue_priority** (*str*) – priority within the job queue. Possible settings include: VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW
- **mapred_job_name** (*str*) – This name will appear in the jobtracker. This can make monitoring easier.

_get_proxy_user (*self*)

This function set the proper proxy_user value in case the user overwtire the default.

_prepare_cli_cmd (*self*)

This function creates the command list from available information

static _prepare_hiveconf (*d*)

This function prepares a list of hiveconf params from a dictionary of key value pairs.

Parameters d (*dict*) –

```
>>> hh = HiveCliHook()
>>> hive_conf = {"hive.exec.dynamic.partition": "true",
... "hive.exec.dynamic.partition.mode": "nonstrict"}
>>> hh._prepare_hiveconf(hive_conf)
["-hiveconf", "hive.exec.dynamic.partition=true", "-hiveconf", "hive.exec.
↪dynamic.partition.mode=nonstrict"]
```

run_cli (*self*, *hql*, *schema=None*, *verbose=True*, *hive_conf=None*)

Run an hql statement using the hive cli. If hive_conf is specified it should be a dict and the entries will be set as key/value pairs in HiveConf

Parameters hive_conf (*dict*) – if specified these key value pairs will be passed to hive as `-hiveconf "key"="value"`. Note that they will be passed after the `hive_cli_params` and thus will override whatever values are specified in the database.

```
>>> hh = HiveCliHook()
>>> result = hh.run_cli("USE airflow;")
>>> ("OK" in result)
True
```

test_hql (*self*, *hql*)

Test an hql statement using the hive cli and EXPLAIN

load_df (*self*, *df*, *table*, *field_dict=None*, *delimiter=',*, *encoding='utf8'*, *pandas_kwargs=None*, ***kwargs*)

Loads a pandas DataFrame into hive.

Hive data types will be inferred if not passed but column names will not be sanitized.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to load into a Hive table
- **table** (*str*) – target Hive table, use dot notation to target a specific database
- **field_dict** (*collections.OrderedDict*) – mapping from column name to hive data type. Note that it must be OrderedDict so as to keep columns' order.

- **delimiter** (*str*) – field delimiter in the file
- **encoding** (*str*) – str encoding to use when writing DataFrame to file
- **pandas_kwargs** (*dict*) – passed to DataFrame.to_csv
- **kwargs** – passed to self.load_file

load_file(*self*, *filepath*, *table*, *delimiter=','*, *field_dict=None*, *create=True*, *overwrite=True*, *partition=None*, *recreate=False*, *tblproperties=None*)

Loads a local file into Hive

Note that the table generated in Hive uses `STORED AS textfile` which isn't the most efficient serialization format. If a large amount of data is loaded and/or if the tables gets queried considerably, you may want to use this operator only to stage the data into a temporary table before loading it into its final destination using a `HiveOperator`.

Parameters

- **filepath** (*str*) – local filepath of the file to load
- **table** (*str*) – target Hive table, use dot notation to target a specific database
- **delimiter** (*str*) – field delimiter in the file
- **field_dict** (*collections.OrderedDict*) – A dictionary of the fields name in the file as keys and their Hive types as values. Note that it must be OrderedDict so as to keep columns' order.
- **create** (*bool*) – whether to create the table if it doesn't exist
- **overwrite** (*bool*) – whether to overwrite the data in table or partition
- **partition** (*dict*) – target partition as a dict of partition columns and values
- **recreate** (*bool*) – whether to drop and recreate the table at every execution
- **tblproperties** (*dict*) – TBLPROPERTIES of the hive table being created

kill(*self*)

class airflow.hooks.hive_hooks.**HiveMetastoreHook**(*metastore_conn_id='metastore_default'*)

Bases: *airflow.hooks.base_hook.BaseHook*

Wrapper to interact with the Hive Metastore

MAX_PART_COUNT = 32767

__getstate__(*self*)

__setstate__(*self*, *d*)

get_metastore_client(*self*)

Returns a Hive thrift client.

_find_valid_server(*self*)

get_conn(*self*)

check_for_partition(*self*, *schema*, *table*, *partition*)

Checks whether a partition exists

Parameters

- **schema** (*str*) – Name of hive schema (database) @table belongs to
- **table** – Name of hive table @partition belongs to

Partition Expression that matches the partitions to check for (eg *a = 'b' AND c = 'd'*)

Return type bool

```
>>> hh = HiveMetastoreHook()
>>> t = 'static_babynames_partitioned'
>>> hh.check_for_partition('airflow', t, "ds='2015-01-01'")
True
```

check_for_named_partition (*self*, *schema*, *table*, *partition_name*)
Checks whether a partition with a given name exists

Parameters

- **schema** (*str*) – Name of hive schema (database) @table belongs to
- **table** – Name of hive table @partition belongs to

Partition Name of the partitions to check for (eg *a=b/c=d*)

Return type bool

```
>>> hh = HiveMetastoreHook()
>>> t = 'static_babynames_partitioned'
>>> hh.check_for_named_partition('airflow', t, "ds=2015-01-01")
True
>>> hh.check_for_named_partition('airflow', t, "ds=xxx")
False
```

get_table (*self*, *table_name*, *db='default'*)
Get a metastore table object

```
>>> hh = HiveMetastoreHook()
>>> t = hh.get_table(db='airflow', table_name='static_babynames')
>>> t.tableName
'static_babynames'
>>> [col.name for col in t.sd.cols]
['state', 'year', 'name', 'gender', 'num']
```

get_tables (*self*, *db*, *pattern='*'*)
Get a metastore table object

get_databases (*self*, *pattern='*'*)
Get a metastore table object

get_partitions (*self*, *schema*, *table_name*, *filter=None*)
Returns a list of all partitions in a table. Works only for tables with less than 32767 (java short max val). For subpartitioned table, the number might easily exceed this.

```
>>> hh = HiveMetastoreHook()
>>> t = 'static_babynames_partitioned'
>>> parts = hh.get_partitions(schema='airflow', table_name=t)
>>> len(parts)
1
>>> parts
[{'ds': '2015-01-01'}]
```

static _get_max_partition_from_part_specs (*part_specs*, *partition_key*, *filter_map*)
Helper method to get max partition of partitions with partition_key from part specs. key:value pair in filter_map will be used to filter out partitions.

Parameters

- **part_specs** (*list*) – list of partition specs.
- **partition_key** (*str*) – partition key name.

- **filter_map** (*map*) – partition_key:partition_value map used for partition filtering, e.g. {'key1': 'value1', 'key2': 'value2'}. Only partitions matching all partition_key:partition_value pairs will be considered as candidates of max partition.

Returns Max partition or None if part_specs is empty.

max_partition (*self*, *schema*, *table_name*, *field=None*, *filter_map=None*)
Returns the maximum value for all partitions with given field in a table. If only one partition key exist in the table, the key will be used as field. filter_map should be a partition_key:partition_value map and will be used to filter out partitions.

Parameters

- **schema** (*str*) – schema name.
- **table_name** (*str*) – table name.
- **field** (*str*) – partition key to get max partition from.
- **filter_map** (*map*) – partition_key:partition_value map used for partition filtering.

```
>>> hh = HiveMetastoreHook()
>>> filter_map = {'ds': '2015-01-01', 'ds': '2014-01-01'}
>>> t = 'static_babynames_partitioned'
>>> hh.max_partition(schema='airflow',        ... table_name=t, field='ds',␣
↪filter_map=filter_map)
'2015-01-01'
```

table_exists (*self*, *table_name*, *db='default'*)
Check if table exists

```
>>> hh = HiveMetastoreHook()
>>> hh.table_exists(db='airflow', table_name='static_babynames')
True
>>> hh.table_exists(db='airflow', table_name='does_not_exist')
False
```

class airflow.hooks.hive_hooks.**HiveServer2Hook** (*hiveserver2_conn_id='hiveserver2_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Wrapper around the pyhive library

Notes: * the default authMechanism is PLAIN, to override it you can specify it in the `extra` of your connection in the UI * the default for run_set_variable_statements is true, if you are using impala you may need to set it to false in the `extra` of your connection in the UI

get_conn (*self*, *schema=None*)
Returns a Hive connection object.

_get_results (*self*, *hql*, *schema='default'*, *fetch_size=None*, *hive_conf=None*)

get_results (*self*, *hql*, *schema='default'*, *fetch_size=None*, *hive_conf=None*)
Get results of the provided hql in target schema.

Parameters

- **hql** (*str or list*) – hql to be executed.
- **schema** (*str*) – target schema, default to 'default'.
- **fetch_size** (*int*) – max size of result to fetch.
- **hive_conf** (*dict*) – hive_conf to execute alone with the hql.

Returns results of hql execution, dict with data (list of results) and header

Return type dict

to_csv(*self*, *hql*, *csv_filepath*, *schema='default'*, *delimiter=','*, *lineterminator='rn'*, *output_header=True*, *fetch_size=1000*, *hive_conf=None*)

Execute hql in target schema and write results to a csv file.

Parameters

- **hql** (`str or list`) – hql to be executed.
- **csv_filepath** (`str`) – filepath of csv to write results into.
- **schema** (`str`) – target schema, default to 'default'.
- **delimiter** (`str`) – delimiter of the csv file, default to ','.
- **lineterminator** (`str`) – lineterminator of the csv file.
- **output_header** (`bool`) – header of the csv file, default to True.
- **fetch_size** (`int`) – number of result rows to write into the csv file, default to 1000.
- **hive_conf** (`dict`) – hive_conf to execute alone with the hql.

get_records(*self*, *hql*, *schema='default'*, *hive_conf=None*)

Get a set of records from a Hive query.

Parameters

- **hql** (`str or list`) – hql to be executed.
- **schema** (`str`) – target schema, default to 'default'.
- **hive_conf** (`dict`) – hive_conf to execute alone with the hql.

Returns result of hive execution

Return type list

```
>>> hh = HiveServer2Hook()
>>> sql = "SELECT * FROM airflow.static_babynames LIMIT 100"
>>> len(hh.get_records(sql))
100
```

get_pandas_df(*self*, *hql*, *schema='default'*)

Get a pandas dataframe from a Hive query

Parameters

- **hql** (`str or list`) – hql to be executed.
- **schema** (`str`) – target schema, default to 'default'.

Returns result of hql execution

Return type DataFrame

```
>>> hh = HiveServer2Hook()
>>> sql = "SELECT * FROM airflow.static_babynames LIMIT 100"
>>> df = hh.get_pandas_df(sql)
>>> len(df.index)
100
```

Returns pandas.DateFrame

airflow.hooks.http_hook

Module Contents

class airflow.hooks.http_hook.**HttpHook**(*method='POST'*, *http_conn_id='http_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interact with HTTP servers.

Parameters

- **http_conn_id** (*str*) – connection that has the base API url i.e https://www.google.com/ and optional authentication credentials. Default headers can also be specified in the Extra field in json format.
- **method** (*str*) – the API method to be called

get_conn(*self*, *headers=None*)
Returns http session for use with requests

Parameters **headers** (*dict*) – additional headers to be passed through as a dictionary

run(*self*, *endpoint*, *data=None*, *headers=None*, *extra_options=None*)
Performs the request

Parameters

- **endpoint** (*str*) – the endpoint to be called i.e. resource/v1/query?
- **data** (*dict*) – payload to be uploaded or request parameters
- **headers** (*dict*) – additional headers to be passed through as a dictionary
- **extra_options** (*dict*) – additional options to be used when executing the request i.e. {'check_response': False} to avoid checking raising exceptions on non 2XX or 3XX status codes

check_response(*self*, *response*)
Checks the status code and raise an AirflowException exception on non 2XX or 3XX status codes

Parameters **response** (*requests.response*) – A requests response object

run_and_check(*self*, *session*, *prepped_request*, *extra_options*)
Grabs extra options like timeout and actually runs the request, checking for the result

Parameters

- **session** (*requests.Session*) – the session to be used to execute the request
- **prepped_request** (*session.prepare_request*) – the prepared request generated in run()
- **extra_options** (*dict*) – additional options to be used when executing the request i.e. {'check_response': False} to avoid checking raising exceptions on non 2XX or 3XX status codes

run_with_advanced_retry(*self*, *_retry_args*, **args*, ***kwargs*)
Runs Hook.run() with a Tenacity decorator attached to it. This is useful for connectors which might be disturbed by intermittent issues and should not instantly fail.

Parameters **_retry_args** (*dict*) – Arguments which define the retry behaviour. See Tenacity documentation at https://github.com/jd/tenacity

:Example:

```
hook = HttpHook(http_conn_id='my_conn',method='GET')
retry_args = dict(
     wait=tenacity.wait_exponential(),
     stop=tenacity.stop_after_attempt(10),
     retry=requests.exceptions.ConnectionError
 )
 hook.run_with_advanced_retry(
         endpoint='v1/test',
         _retry_args=retry_args
     )
```

airflow.hooks.jdbc_hook

Module Contents

class airflow.hooks.jdbc_hook.**JdbcHook**

Bases: *airflow.hooks.dbapi_hook.DbApiHook*

General hook for jdbc db access.

JDBC URL, username and password will be taken from the predefined connection. Note that the whole JDBC URL must be specified in the "host" field in the DB. Raises an airflow error if the given connection id doesn't exist.

conn_name_attr = jdbc_conn_id

default_conn_name = jdbc_default

supports_autocommit = True

get_conn(*self*)

set_autocommit(*self*, *conn*, *autocommit*)

Enable or disable autocommit for the given connection.

Parameters **conn** – The connection

Returns

airflow.hooks.mssql_hook

Module Contents

class airflow.hooks.mssql_hook.**MsSqlHook**(**args*, ***kwargs*)

Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with Microsoft SQL Server.

conn_name_attr = mssql_conn_id

default_conn_name = mssql_default

supports_autocommit = True

get_conn(*self*)

Returns a mssql connection object

set_autocommit(*self*, *conn*, *autocommit*)

get_autocommit(*self*, *conn*)

`airflow.hooks.mysql_hook`

Module Contents

class `airflow.hooks.mysql_hook.`**`MySqlHook`**(**args*, ***kwargs*)

Bases: `airflow.hooks.dbapi_hook.DbApiHook`

Interact with MySQL.

You can specify charset in the extra field of your connection as `{"charset": "utf8"}`. Also you can choose cursor as `{"cursor": "SSCursor"}`. Refer to the MySQLdb.cursors for more details.

Note: For AWS IAM authentication, use iam in the extra connection parameters and set it to true. Leave the password field empty. This will use the the "aws_default" connection to get the temporary token unless you override in extras. extras example: `{"iam":true, "aws_conn_id":"my_aws_conn"}`

`conn_name_attr = mysql_conn_id`

`default_conn_name = mysql_default`

`supports_autocommit = True`

`set_autocommit`(*self*, *conn*, *autocommit*)

MySql connection sets autocommit in a different way.

`get_autocommit`(*self*, *conn*)

MySql connection gets autocommit in a different way.

Parameters **conn** (*`connection object.`*) – connection to get autocommit setting from.

Returns connection autocommit setting

Return type bool

`get_conn`(*self*)

Returns a mysql connection object

`bulk_load`(*self*, *table*, *tmp_file*)

Loads a tab-delimited file into a database table

`bulk_dump`(*self*, *table*, *tmp_file*)

Dumps a database table into a tab-delimited file

`static _serialize_cell`(*cell*, *conn*)

MySQLdb converts an argument to a literal when passing those separately to execute. Hence, this method does nothing.

Parameters

- **cell** (*`object`*) – The cell to insert into the table
- **conn** (*`connection object`*) – The database connection

Returns The same cell

Return type object

`get_iam_token`(*self*, *conn*)

Uses AWSHook to retrieve a temporary password to connect to MySQL Port is required. If none is provided, default 3306 is used

`airflow.hooks.oracle_hook`

Module Contents

class `airflow.hooks.oracle_hook.`**`OracleHook`**

Bases: *`airflow.hooks.dbapi_hook.DbApiHook`*

Interact with Oracle SQL.

`conn_name_attr = oracle_conn_id`

`default_conn_name = oracle_default`

`supports_autocommit = False`

`get_conn`(*self*)

Returns a oracle connection object Optional parameters for using a custom DSN connection (instead of using a server alias from tnsnames.ora) The dsn (data source name) is the TNS entry (from the Oracle names server or tnsnames.ora file) or is a string like the one returned from makedsn().

Parameters

- **`dsn`** – the host address for the Oracle server
- **`service_name`** – the db_unique_name of the database that you are connecting to (CONNECT_DATA part of TNS)

You can set these parameters in the extra fields of your connection as in `{ "dsn":"some.host.address" , "service_name":"some.service.name" }` see more param detail in cx_Oracle.connect

`insert_rows`(*self*, *table*, *rows*, *target_fields=None*, *commit_every=1000*)

A generic way to insert a set of tuples into a table, the whole set of inserts is treated as one transaction Changes from standard DbApiHook implementation:

- Oracle SQL queries in cx_Oracle can not be terminated with a semicolon (*;*)
- Replace NaN values with NULL using *numpy.nan_to_num* (not using *is_nan()* because of input types error for strings)
- Coerce datetime cells to Oracle DATETIME format during insert

Parameters

- **`table`** (*`str`*) – target Oracle table, use dot notation to target a specific database
- **`rows`** (*`iterable of tuples`*) – the rows to insert into the table
- **`target_fields`** (*`iterable of str`*) – the names of the columns to fill in the table
- **`commit_every`** (*`int`*) – the maximum number of rows to insert in one transaction Default 1000, Set greater than 0. Set 1 to insert each row in each single transaction

`bulk_insert_rows`(*self*, *table*, *rows*, *target_fields=None*, *commit_every=5000*)

A performant bulk insert for cx_Oracle that uses prepared statements via *executemany()*. For best performance, pass in *rows* as an iterator.

Parameters

- **`table`** (*`str`*) – target Oracle table, use dot notation to target a specific database
- **`rows`** (*`iterable of tuples`*) – the rows to insert into the table
- **`target_fields`** (*`iterable of str Or None`*) – the names of the columns to fill in the table, default None. If None, each rows should have some order as table columns name

- **commit_every** (*int*) – the maximum number of rows to insert in one transaction Default 5000. Set greater than 0. Set 1 to insert each row in each transaction

airflow.hooks.pig_hook

Module Contents

class airflow.hooks.pig_hook.**PigCliHook**(*pig_cli_conn_id='pig_cli_default'*)

Bases: *airflow.hooks.base_hook.BaseHook*

Simple wrapper around the pig CLI.

Note that you can also set default pig CLI properties using the `pig_properties` to be used in your connection as in `{"pig_properties": "-Dpig.tmpfilecompression=true"}`

run_cli(*self*, *pig*, *pig_opts=None*, *verbose=True*)

Run an pig script using the pig cli

```
>>> ph = PigCliHook()
>>> result = ph.run_cli("ls /;", pig_opts="-x mapreduce")
>>> ("hdfs://" in result)
True
```

kill(*self*)

airflow.hooks.postgres_hook

Module Contents

class airflow.hooks.postgres_hook.**PostgresHook**(**args*, ***kwargs*)

Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with Postgres. You can specify ssl parameters in the extra field of your connection as `{"sslmode": "require", "sslcert": "/path/to/cert.pem", etc}`.

Note: For Redshift, use keepalives_idle in the extra connection parameters and set it to less than 300 seconds.

Note: For AWS IAM authentication, use iam in the extra connection parameters and set it to true. Leave the password field empty. This will use the the "aws_default" connection to get the temporary token unless you override in extras. extras example: `{"iam":true, "aws_conn_id":"my_aws_conn"}` For Redshift, also use redshift in the extra connection parameters and set it to true. The cluster-identifier is extracted from the beginning of the host field, so is optional. It can however be overridden in the extra field. extras example: `{"iam":true, "redshift":true, "cluster-identifier": "my_cluster_id"}`

conn_name_attr = postgres_conn_id

default_conn_name = postgres_default

supports_autocommit = True

get_conn(*self*)

copy_expert(*self*, *sql*, *filename*, *open=open*)

Executes SQL using psycopg2 copy_expert method. Necessary to execute COPY command without access to a superuser.

Note: if this method is called with a "COPY FROM" statement and the specified input file does not exist, it creates an empty file and no data is loaded, but the operation succeeds. So if users want to be aware when the input file does not exist, they have to check its existence by themselves.

bulk_load(*self*, *table*, *tmp_file*)
Loads a tab-delimited file into a database table

bulk_dump(*self*, *table*, *tmp_file*)
Dumps a database table into a tab-delimited file

static _serialize_cell(*cell*, *conn*)
Postgresql will adapt all arguments to the execute() method internally, hence we return cell without any conversion.

See http://initd.org/psycopg/docs/advanced.html#adapting-new-types for more information.

Parameters

- **cell** (`object`) – The cell to insert into the table
- **conn** (`connection object`) – The database connection

Returns The cell

Return type object

get_iam_token(*self*, *conn*)
Uses AWSHook to retrieve a temporary password to connect to Postgres or Redshift. Port is required. If none is provided, default is used for each service

airflow.hooks.presto_hook

Module Contents

exception airflow.hooks.presto_hook.**PrestoException**
Bases: `Exception`

class airflow.hooks.presto_hook.**PrestoHook**
Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with Presto through PyHive!

```
>>> ph = PrestoHook()
>>> sql = "SELECT count(1) AS num FROM airflow.static_babynames"
>>> ph.get_records(sql)
[[340698]]
```

conn_name_attr = presto_conn_id

default_conn_name = presto_default

get_conn(*self*)
Returns a connection object

static _strip_sql(*sql*)

static _get_pretty_exception_message(*e*)
Parses some DatabaseError to provide a better error message

get_records(*self*, *hql*, *parameters=None*)
Get a set of records from Presto

get_first(*self*, *hql*, *parameters=None*)
Returns only the first row, regardless of how many rows the query returns.

get_pandas_df(*self*, *hql*, *parameters=None*)
Get a pandas dataframe from a sql query.

run (*self*, *hql*, *parameters=None*)
Execute the statement against Presto. Can be used to create views.

insert_rows (*self*, *table*, *rows*, *target_fields=None*)
A generic way to insert a set of tuples into a table.

Parameters

- **table** (*str*) – Name of the target table
- **rows** (*iterable of tuples*) – The rows to insert into the table
- **target_fields** (*iterable of strings*) – The names of the columns to fill in the table

airflow.hooks.samba_hook

Module Contents

class airflow.hooks.samba_hook.**SambaHook** (*samba_conn_id*)
Bases: *airflow.hooks.base_hook.BaseHook*

Allows for interaction with an samba server.

get_conn (*self*)

push_from_local (*self*, *destination_filepath*, *local_filepath*)

airflow.hooks.slack_hook

Module Contents

class airflow.hooks.slack_hook.**SlackHook** (*token=None*, *slack_conn_id=None*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interact with Slack, using slackclient library.

__get_token (*self*, *token*, *slack_conn_id*)

call (*self*, *method*, *api_params*)

airflow.hooks.sqlite_hook

Module Contents

class airflow.hooks.sqlite_hook.**SqliteHook**
Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with SQLite.

conn_name_attr = sqlite_conn_id

default_conn_name = sqlite_default

supports_autocommit = False

get_conn (*self*)
Returns a sqlite connection object

airflow.hooks.webhdfs_hook

Module Contents

airflow.hooks.webhdfs_hook.**_kerberos_security_mode**

airflow.hooks.webhdfs_hook.**log**

exception airflow.hooks.webhdfs_hook.**AirflowWebHDFSHookException**
Bases: airflow.exceptions.AirflowException

class airflow.hooks.webhdfs_hook.**WebHDFSHook**(*webhdfs_conn_id='webhdfs_default'*, *proxy_user=None*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interact with HDFS. This class is a wrapper around the hdfscli library.

Parameters

- **webhdfs_conn_id** (*str*) – The connection id for the webhdfs client to connect to.
- **proxy_user** (*str*) – The user used to authenticate.

get_conn(*self*)
Establishes a connection depending on the security mode set via config or environment variable.

Returns a hdfscli InsecureClient or KerberosClient object.

Return type hdfs.InsecureClient or hdfs.ext.kerberos.KerberosClient

_get_client(*self*, *connection*)

check_for_path(*self*, *hdfs_path*)
Check for the existence of a path in HDFS by querying FileStatus.

Parameters **hdfs_path** (*str*) – The path to check.

Returns True if the path exists and False if not.

Return type bool

load_file(*self*, *source*, *destination*, *overwrite=True*, *parallelism=1*, ***kwargs*)
Uploads a file to HDFS.

Parameters

- **source** (*str*) – Local path to file or folder. If it's a folder, all the files inside of it will be uploaded. .. note:: This implies that folders empty of files will not be created remotely.
- **destination** (*str*) – PTarget HDFS path. If it already exists and is a directory, files will be uploaded inside.
- **overwrite** (*bool*) – Overwrite any existing file or directory.
- **parallelism** (*int*) – Number of threads to use for parallelization. A value of *0* (or negative) uses as many threads as there are files.
- ****kwargs** – Keyword arguments forwarded to hdfs.client.Client.upload().

airflow.hooks.zendesk_hook

Module Contents

class airflow.hooks.zendesk_hook.**ZendeskHook**(*zendesk_conn_id*)
Bases: *airflow.hooks.base_hook.BaseHook*

A hook to talk to Zendesk

get_conn (*self*)

__handle_rate_limit_exception (*self*, *rate_limit_exception*)
Sleep for the time specified in the exception. If not specified, wait for 60 seconds.

call (*self*, *path*, *query=None*, *get_all_pages=True*, *side_loading=False*)
Call Zendesk API and return results

Parameters

- **path** – The Zendesk API to call
- **query** – Query parameters
- **get_all_pages** – Accumulate results over all pages before returning. Due to strict rate limiting, this can often timeout. Waits for recommended period between tries after a timeout.
- **side_loading** – Retrieve related records as part of a single request. In order to enable side-loading, add an 'include' query parameter containing a comma-separated list of resources to load. For more information on side-loading see https://developer.zendesk.com/rest_api/docs/core/side_loading

Package Contents

airflow.hooks.**_integrate_plugins**()
Integrate plugins to the context

airflow.contrib.hooks

Submodules

airflow.contrib.hooks.aws_athena_hook

Module Contents

class airflow.contrib.hooks.aws_athena_hook.**AWSAthenaHook** (*aws_conn_id='aws_default'*, *sleep_time=30*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Athena to run, poll queries and return query results

Parameters

- **aws_conn_id** (*str*) – aws connection to use.
- **sleep_time** (*int*) – Time to wait between two consecutive call to check query status on athena

INTERMEDIATE_STATES = ['QUEUED', 'RUNNING']

FAILURE_STATES = ['FAILED', 'CANCELLED']

SUCCESS_STATES = ['SUCCEEDED']

get_conn (*self*)
check if aws conn exists already or create one and return it

Returns boto3 session

run_query (*self*, *query*, *query_context*, *result_configuration*, *client_request_token=None*)
Run Presto query on athena with provided config and return submitted query_execution_id

Parameters

- **query** (*str*) – Presto query to run
- **query_context** (*dict*) – Context in which query need to be run
- **result_configuration** (*dict*) – Dict with path to store results in and config related to encryption
- **client_request_token** (*str*) – Unique token created by user to avoid multiple executions of same query

Returns str

check_query_status (*self*, *query_execution_id*)
Fetch the status of submitted athena query. Returns None or one of valid query states.

Parameters **query_execution_id** (*str*) – Id of submitted athena query

Returns str

get_query_results (*self*, *query_execution_id*)
Fetch submitted athena query results. returns none if query is in intermediate state or failed/cancelled state else dict of query output

Parameters **query_execution_id** (*str*) – Id of submitted athena query

Returns dict

poll_query_status (*self*, *query_execution_id*, *max_tries=None*)
Poll the status of submitted athena query until query state reaches final state. Returns one of the final states

Parameters

- **query_execution_id** (*str*) – Id of submitted athena query
- **max_tries** (*int*) – Number of times to poll for query state before function exits

Returns str

stop_query (*self*, *query_execution_id*)
Cancel the submitted athena query

Parameters **query_execution_id** (*str*) – Id of submitted athena query

Returns dict

airflow.contrib.hooks.aws_dynamodb_hook

Module Contents

class airflow.contrib.hooks.aws_dynamodb_hook.**AwsDynamoDBHook** (*table_keys=None*, *ta-ble_name=None*, *re-gion_name=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS DynamoDB.

Parameters

- **table_keys** (*list*) – partition key and sort key
- **table_name** (*str*) – target DynamoDB table
- **region_name** (*str*) – aws region name (example: us-east-1)

get_conn (*self*)

write_batch_data (*self*, *items*)
Write batch items to dynamodb table with provisioned throughout capacity.

airflow.contrib.hooks.aws_firehose_hook

Module Contents

class airflow.contrib.hooks.aws_firehose_hook.**AwsFirehoseHook** (*delivery_stream*, *region_name=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Kinesis Firehose. :param delivery_stream: Name of the delivery stream :type delivery_stream: str :param region_name: AWS region name (example: us-east-1) :type region_name: str

get_conn (*self*)
Returns AwsHook connection object.

put_records (*self*, *records*)
Write batch records to Kinesis Firehose

airflow.contrib.hooks.aws_glue_catalog_hook

Module Contents

class airflow.contrib.hooks.aws_glue_catalog_hook.**AwsGlueCatalogHook** (*aws_conn_id='aws_default'*, *region_name=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Glue Catalog

Parameters

- **aws_conn_id** (*str*) – ID of the Airflow connection where credentials and extra configuration are stored
- **region_name** (*str*) – aws region name (example: us-east-1)

get_conn (*self*)
Returns glue connection object.

get_partitions (*self*, *database_name*, *table_name*, *expression=''*, *page_size=None*, *max_items=None*)
Retrieves the partition values for a table.

Parameters

- **database_name** (*str*) – The name of the catalog database where the partitions reside.
- **table_name** (*str*) – The name of the partitions' table.

- **expression** (*str*) – An expression filtering the partitions to be returned. Please see official AWS documentation for further information. https://docs.aws.amazon.com/glue/latest/dg/aws-glue-api-catalog-partitions.html#aws-glue-api-catalog-partitions-GetPartitions
- **page_size** (*int*) – pagination size
- **max_items** (*int*) – maximum items to return

Returns set of partition values where each value is a tuple since a partition may be composed of multiple columns. For example: `{('2018-01-01','1'), ('2018-01-01','2')}`

check_for_partition (*self*, *database_name*, *table_name*, *expression*)
Checks whether a partition exists

Parameters

- **database_name** (*str*) – Name of hive database (schema) @table belongs to
- **table_name** (*str*) – Name of hive table @partition belongs to

Expression Expression that matches the partitions to check for (eg *a = 'b' AND c = 'd'*)

Return type bool

```
>>> hook = AwsGlueCatalogHook()
>>> t = 'static_babynames_partitioned'
>>> hook.check_for_partition('airflow', t, "ds='2015-01-01'")
True
```

get_table (*self*, *database_name*, *table_name*)
Get the information of the table

Parameters

- **database_name** (*str*) – Name of hive database (schema) @table belongs to
- **table_name** (*str*) – Name of hive table

Return type dict

```
>>> hook = AwsGlueCatalogHook()
>>> r = hook.get_table('db', 'table_foo')
>>> r['Name'] = 'table_foo'
```

get_table_location (*self*, *database_name*, *table_name*)
Get the physical location of the table

Parameters

- **database_name** (*str*) – Name of hive database (schema) @table belongs to
- **table_name** (*str*) – Name of hive table

Returns str

airflow.contrib.hooks.aws_hook

Module Contents

airflow.contrib.hooks.aws_hook.**_parse_s3_config** (*config_file_name*, *config_format='boto'*, *profile=None*)

Parses a config file for s3 credentials. Can currently parse boto, s3cmd.conf and AWS SDK config formats

Parameters

- **config_file_name** (*str*) – path to the config file
- **config_format** (*str*) – config type. One of "boto", "s3cmd" or "aws". Defaults to "boto"
- **profile** (*str*) – profile name in AWS type config file

class airflow.contrib.hooks.aws_hook.**AwsHook**(*aws_conn_id='aws_default'*, *verify=None*)

Bases: *airflow.hooks.base_hook.BaseHook*

Interact with AWS. This class is a thin wrapper around the boto3 python library.

_get_credentials(*self*, *region_name*)

get_client_type(*self*, *client_type*, *region_name=None*, *config=None*)

get_resource_type(*self*, *resource_type*, *region_name=None*, *config=None*)

get_session(*self*, *region_name=None*)

Get the underlying boto3.session.

get_credentials(*self*, *region_name=None*)

Get the underlying *botocore.Credentials* object.

This contains the following authentication attributes: access_key, secret_key and token.

expand_role(*self*, *role*)

If the IAM role is a role name, get the Amazon Resource Name (ARN) for the role. If IAM role is already an IAM role ARN, no change is made.

Parameters **role** – IAM role name or ARN

Returns IAM role ARN

airflow.contrib.hooks.aws_lambda_hook

Module Contents

class airflow.contrib.hooks.aws_lambda_hook.**AwsLambdaHook**(*function_name*, *region_name=None*, *log_type='None'*, *qualifier='$LATEST'*, *invocation_type='RequestResponse'*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Lambda

Parameters

- **function_name** (*str*) – AWS Lambda Function Name
- **region_name** (*str*) – AWS Region Name (example: us-west-2)
- **log_type** (*str*) – Tail Invocation Request
- **qualifier** (*str*) – AWS Lambda Function Version or Alias Name
- **invocation_type** (*str*) – AWS Lambda Invocation Type (RequestResponse, Event etc)

get_conn(*self*)

invoke_lambda(*self*, *payload*)

Invoke Lambda Function

`airflow.contrib.hooks.aws_sns_hook`

Module Contents

class `airflow.contrib.hooks.aws_sns_hook.`**`AwsSnsHook`**(**args*, ***kwargs*)

Bases: `airflow.contrib.hooks.aws_hook.AwsHook`

Interact with Amazon Simple Notification Service.

`get_conn`(*self*)

Get an SNS connection

`publish_to_target`(*self*, *target_arn*, *message*)

Publish a message to a topic or an endpoint.

Parameters

- **`target_arn`** (`str`) – either a TopicArn or an EndpointArn
- **`message`** – the default message you want to send
- **`message`** – str

`airflow.contrib.hooks.aws_sqs_hook`

Module Contents

class `airflow.contrib.hooks.aws_sqs_hook.`**`SQSHook`**

Bases: `airflow.contrib.hooks.aws_hook.AwsHook`

Get the SQS client using boto3 library

Returns SQS client

Return type botocore.client.SQS

`get_conn`(*self*)

`create_queue`(*self*, *queue_name*, *attributes=None*)

Create queue using connection object

Parameters

- **`queue_name`** (`str`) – name of the queue.
- **`attributes`** (`dict`) – additional attributes for the queue (default: None) For details of the attributes parameter see `botocore.client.SQS.create_queue()`

Returns dict with the information about the queue For details of the returned value see `botocore.client.SQS.create_queue()`

Return type dict

`send_message`(*self*, *queue_url*, *message_body*, *delay_seconds=0*, *message_attributes=None*)

Send message to the queue

Parameters

- **`queue_url`** (`str`) – queue url
- **`message_body`** (`str`) – the contents of the message
- **`delay_seconds`** (`int`) – seconds to delay the message

- **message_attributes** (*dict*) – additional attributes for the message (default: None) For details of the attributes parameter see `botocore.client.SQS.send_message()`

Returns dict with the information about the message sent For details of the returned value see `botocore.client.SQS.send_message()`

Return type dict

airflow.contrib.hooks.azure_container_instance_hook

Module Contents

class `airflow.contrib.hooks.azure_container_instance_hook.`**`AzureContainerInstanceHook`**(*conn_id*

Bases: *airflow.hooks.base_hook.BaseHook*

A hook to communicate with Azure Container Instances.

This hook requires a service principal in order to work. After creating this service principal (Azure Active Directory/App Registrations), you need to fill in the client_id (Application ID) as login, the generated password as password, and tenantId and subscriptionId in the extra's field as a json.

Parameters **conn_id** (*str*) – connection id of a service principal which will be used to start the container instance

get_conn(*self*)

create_or_update(*self*, *resource_group*, *name*, *container_group*)

Create a new container group

Parameters

- **resource_group** (*str*) – the name of the resource group
- **name** (*str*) – the name of the container group
- **container_group** (*azure.mgmt.containerinstance.models.ContainerGroup*) – the properties of the container group

get_state_exitcode_details(*self*, *resource_group*, *name*)

Get the state and exitcode of a container group

Parameters

- **resource_group** (*str*) – the name of the resource group
- **name** (*str*) – the name of the container group

Returns A tuple with the state, exitcode, and details. If the exitcode is unknown 0 is returned.

Return type tuple(state,exitcode,details)

_get_instance_view(*self*, *resource_group*, *name*)

get_messages(*self*, *resource_group*, *name*)

Get the messages of a container group

Parameters

- **resource_group** (*str*) – the name of the resource group
- **name** (*str*) – the name of the container group

Returns A list of the event messages

Return type list[str]

get_logs (*self*, *resource_group*, *name*, *tail=1000*)
Get the tail from logs of a container group

Parameters

- **resource_group** (*str*) – the name of the resource group
- **name** (*str*) – the name of the container group
- **tail** (*int*) – the size of the tail

Returns A list of log messages

Return type list[str]

delete (*self*, *resource_group*, *name*)
Delete a container group

Parameters

- **resource_group** (*str*) – the name of the resource group
- **name** (*str*) – the name of the container group

exists (*self*, *resource_group*, *name*)
Test if a container group exists

Parameters

- **resource_group** (*str*) – the name of the resource group
- **name** (*str*) – the name of the container group

airflow.contrib.hooks.azure_container_registry_hook

Module Contents

class airflow.contrib.hooks.azure_container_registry_hook.**AzureContainerRegistryHook** (*conn_id*
Bases: *airflow.hooks.base_hook.BaseHook*

A hook to communicate with a Azure Container Registry.

Parameters **conn_id** (*str*) – connection id of a service principal which will be used to start the container instance

get_conn (*self*)

airflow.contrib.hooks.azure_container_volume_hook

Module Contents

class airflow.contrib.hooks.azure_container_volume_hook.**AzureContainerVolumeHook** (*wasb_conn_id*
Bases: *airflow.hooks.base_hook.BaseHook*

A hook which wraps an Azure Volume.

Parameters **wasb_conn_id** (*str*) – connection id of a Azure storage account of which file shares should be mounted

get_storagekey (*self*)

get_file_volume (*self*, *mount_name*, *share_name*, *storage_account_name*, *read_only=False*)

airflow.contrib.hooks.azure_cosmos_hook

Module Contents

class airflow.contrib.hooks.azure_cosmos_hook.**AzureCosmosDBHook**(*azure_cosmos_conn_id='azure_cosmos_*

Bases: *airflow.hooks.base_hook.BaseHook*

Interacts with Azure CosmosDB.

login should be the endpoint uri, password should be the master key optionally, you can use the following extras to default these values {"database_name": "<DATABASE_NAME>", "collection_name": "COLLECTION_NAME"}.

Parameters **azure_cosmos_conn_id** (*str*) – Reference to the Azure CosmosDB connection.

get_conn(*self*)
Return a cosmos db client.

__get_database_name(*self*, *database_name=None*)

__get_collection_name(*self*, *collection_name=None*)

does_collection_exist(*self*, *collection_name*, *database_name=None*)
Checks if a collection exists in CosmosDB.

create_collection(*self*, *collection_name*, *database_name=None*)
Creates a new collection in the CosmosDB database.

does_database_exist(*self*, *database_name*)
Checks if a database exists in CosmosDB.

create_database(*self*, *database_name*)
Creates a new database in CosmosDB.

delete_database(*self*, *database_name*)
Deletes an existing database in CosmosDB.

delete_collection(*self*, *collection_name*, *database_name=None*)
Deletes an existing collection in the CosmosDB database.

upsert_document(*self*, *document*, *database_name=None*, *collection_name=None*, *document_id=None*)
Inserts a new document (or updates an existing one) into an existing collection in the CosmosDB database.

insert_documents(*self*, *documents*, *database_name=None*, *collection_name=None*)
Insert a list of new documents into an existing collection in the CosmosDB database.

delete_document(*self*, *document_id*, *database_name=None*, *collection_name=None*)
Delete an existing document out of a collection in the CosmosDB database.

get_document(*self*, *document_id*, *database_name=None*, *collection_name=None*)
Get a document from an existing collection in the CosmosDB database.

get_documents(*self*, *sql_string*, *database_name=None*, *collection_name=None*, *partition_key=None*)
Get a list of documents from an existing collection in the CosmosDB database via SQL query.

airflow.contrib.hooks.azure_cosmos_hook.**get_database_link**(*database_id*)

airflow.contrib.hooks.azure_cosmos_hook.**get_collection_link**(*database_id*, *collection_id*)

airflow.contrib.hooks.azure_cosmos_hook.**get_document_link**(*database_id*, *collection_id*, *document_id*)

airflow.contrib.hooks.azure_data_lake_hook

Module Contents

class airflow.contrib.hooks.azure_data_lake_hook.**AzureDataLakeHook**(*azure_data_lake_conn_id='azure_d*

Bases: *airflow.hooks.base_hook.BaseHook*

Interacts with Azure Data Lake.

Client ID and client secret should be in user and password parameters. Tenant and account name should be extra field as {"tenant": "<TENANT>", "account_name": "ACCOUNT_NAME"}.

Parameters **azure_data_lake_conn_id** (*str*) – Reference to the Azure Data Lake connection.

get_conn(*self*)

Return a AzureDLFileSystem object.

check_for_file(*self*, *file_path*)

Check if a file exists on Azure Data Lake.

Parameters **file_path** (*str*) – Path and name of the file.

Returns True if the file exists, False otherwise.

Return type bool

upload_file(*self*, *local_path*, *remote_path*, *nthreads=64*, *overwrite=True*, *buffersize=4194304*, *blocksize=4194304*)

Upload a file to Azure Data Lake.

Parameters

- **local_path** (*str*) – local path. Can be single file, directory (in which case, upload recursively) or glob pattern. Recursive glob patterns using ** are not supported.
- **remote_path** (*str*) – Remote path to upload to; if multiple files, this is the directory root to write within.
- **nthreads** (*int*) – Number of threads to use. If None, uses the number of cores.
- **overwrite** (*bool*) – Whether to forcibly overwrite existing files/directories. If False and remote path is a directory, will quit regardless if any files would be overwritten or not. If True, only matching filenames are actually overwritten.
- **buffersize** (*int*) – int [2**22] Number of bytes for internal buffer. This block cannot be bigger than a chunk and cannot be smaller than a block.
- **blocksize** (*int*) – int [2**22] Number of bytes for a block. Within each chunk, we write a smaller block for each API call. This block cannot be bigger than a chunk.

download_file(*self*, *local_path*, *remote_path*, *nthreads=64*, *overwrite=True*, *buffersize=4194304*, *blocksize=4194304*)

Download a file from Azure Blob Storage.

Parameters

- **local_path** (*str*) – local path. If downloading a single file, will write to this specific file, unless it is an existing directory, in which case a file is created within it. If downloading multiple files, this is the root directory to write within. Will create directories as required.
- **remote_path** (*str*) – remote path/globstring to use to find remote files. Recursive glob patterns using ** are not supported.
- **nthreads** (*int*) – Number of threads to use. If None, uses the number of cores.

- **overwrite** (*bool*) – Whether to forcibly overwrite existing files/directories. If False and remote path is a directory, will quit regardless if any files would be overwritten or not. If True, only matching filenames are actually overwritten.
- **buffersize** (*int*) – int [2**22] Number of bytes for internal buffer. This block cannot be bigger than a chunk and cannot be smaller than a block.
- **blocksize** (*int*) – int [2**22] Number of bytes for a block. Within each chunk, we write a smaller block for each API call. This block cannot be bigger than a chunk.

list (*self*, *path*)
List files in Azure Data Lake Storage

Parameters **path** (*str*) – full path/globstring to use to list files in ADLS

airflow.contrib.hooks.azure_fileshare_hook

Module Contents

class airflow.contrib.hooks.azure_fileshare_hook.**AzureFileShareHook** (*wasb_conn_id='wasb_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interacts with Azure FileShare Storage.

Additional options passed in the 'extra' field of the connection will be passed to the *FileService()* constructor.

Parameters **wasb_conn_id** (*str*) – Reference to the wasb connection.

get_conn (*self*)
Return the FileService object.

check_for_directory (*self*, *share_name*, *directory_name*, ***kwargs*)
Check if a directory exists on Azure File Share.

Parameters

- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.exists()* takes.

Returns True if the file exists, False otherwise.

Return type bool

check_for_file (*self*, *share_name*, *directory_name*, *file_name*, ***kwargs*)
Check if a file exists on Azure File Share.

Parameters

- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **file_name** (*str*) – Name of the file.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.exists()* takes.

Returns True if the file exists, False otherwise.

Return type bool

list_directories_and_files (*self*, *share_name*, *directory_name=None*, ***kwargs*)
Return the list of directories and files stored on a Azure File Share.

Parameters

- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.list_directories_and_files()* takes.

Returns A list of files and directories

Return type list

create_directory (*self*, *share_name*, *directory_name*, ***kwargs*)
Create a new directory on a Azure File Share.

Parameters

- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.create_directory()* takes.

Returns A list of files and directories

Return type list

get_file (*self*, *file_path*, *share_name*, *directory_name*, *file_name*, ***kwargs*)
Download a file from Azure File Share.

Parameters

- **file_path** (*str*) – Where to store the file.
- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **file_name** (*str*) – Name of the file.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.get_file_to_path()* takes.

get_file_to_stream (*self*, *stream*, *share_name*, *directory_name*, *file_name*, ***kwargs*)
Download a file from Azure File Share.

Parameters

- **stream** (*file-like object*) – A filehandle to store the file to.
- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **file_name** (*str*) – Name of the file.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.get_file_to_stream()* takes.

load_file (*self*, *file_path*, *share_name*, *directory_name*, *file_name*, ***kwargs*)
Upload a file to Azure File Share.

Parameters

- **file_path** (*str*) – Path to the file to load.
- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **file_name** (*str*) – Name of the file.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.create_file_from_path()* takes.

load_string (*self*, *string_data*, *share_name*, *directory_name*, *file_name*, ***kwargs*)
Upload a string to Azure File Share.

Parameters

- **string_data** (*str*) – String to load.
- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **file_name** (*str*) – Name of the file.
- **kwargs** (*object*) – Optional keyword arguments that *FileService.create_file_from_text()* takes.

load_stream (*self*, *stream*, *share_name*, *directory_name*, *file_name*, *count*, ***kwargs*)
Upload a stream to Azure File Share.

Parameters

- **stream** (*file-like*) – Opened file/stream to upload as the file content.
- **share_name** (*str*) – Name of the share.
- **directory_name** (*str*) – Name of the directory.
- **file_name** (*str*) – Name of the file.
- **count** (*int*) – Size of the stream in bytes
- **kwargs** (*object*) – Optional keyword arguments that *FileService.create_file_from_stream()* takes.

airflow.contrib.hooks.bigquery_hook

This module contains a BigQuery Hook, as well as a very basic PEP 249 implementation for BigQuery.

Module Contents

class airflow.contrib.hooks.bigquery_hook.**BigQueryHook** (*bigquery_conn_id='google_cloud_default'*, *delegate_to=None*, *use_legacy_sql=True*, *location=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*, *airflow.hooks.dbapi_hook.DbApiHook*

Interact with BigQuery. This hook uses the Google Cloud Platform connection.

conn_name_attr = bigquery_conn_id

get_conn (*self*)
Returns a BigQuery PEP 249 connection object.

get_service (*self*)
Returns a BigQuery service object.

insert_rows (*self*, *table*, *rows*, *target_fields=None*, *commit_every=1000*)
Insertion is currently unsupported. Theoretically, you could use BigQuery's streaming API to insert rows into a table, but this hasn't been implemented.

get_pandas_df (*self*, *sql*, *parameters=None*, *dialect=None*)
Returns a Pandas DataFrame for the results produced by a BigQuery query. The DbApiHook method must be overridden because Pandas doesn't support PEP 249 connections, except for SQLite. See:

https://github.com/pydata/pandas/blob/master/pandas/io/sql.py#L447 https://github.com/pydata/pandas/issues/6900

Parameters

- **sql** (*str*) – The BigQuery SQL to execute.
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with (not used, leave to override superclass method)
- **dialect** (*str in {'legacy', 'standard'}*) – Dialect of BigQuery SQL – legacy SQL or standard SQL defaults to use *self.use_legacy_sql* if not specified

table_exists(*self*, *project_id*, *dataset_id*, *table_id*)

Checks for the existence of a table in Google BigQuery.

Parameters

- **project_id** (*str*) – The Google cloud project in which to look for the table. The connection supplied to the hook must provide access to the specified project.
- **dataset_id** (*str*) – The name of the dataset in which to look for the table.
- **table_id** (*str*) – The name of the table to check the existence of.

class airflow.contrib.hooks.bigquery_hook.**BigQueryPandasConnector**(*project_id*, *service*, *reauth=False*, *verbose=False*, *dialect='legacy'*)

Bases: `pandas_gbq.gbq.GbqConnector`

This connector behaves identically to GbqConnector (from Pandas), except that it allows the service to be injected, and disables a call to self.get_credentials(). This allows Airflow to use BigQuery with Pandas without forcing a three legged OAuth connection. Instead, we can inject service account credentials into the binding.

class airflow.contrib.hooks.bigquery_hook.**BigQueryConnection**(**args*, ***kwargs*)

BigQuery does not have a notion of a persistent connection. Thus, these objects are small stateless factories for cursors, which do all the real work.

close(*self*)

BigQueryConnection does not have anything to close.

commit(*self*)

BigQueryConnection does not support transactions.

cursor(*self*)

Return a new `Cursor` object using the connection.

rollback(*self*)

class airflow.contrib.hooks.bigquery_hook.**BigQueryBaseCursor**(*service*, *project_id*, *use_legacy_sql=True*, *api_resource_configs=None*, *location=None*, *num_retries=None*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

The BigQuery base cursor contains helper methods to execute queries against BigQuery. The methods can be used directly by operators, in cases where a PEP 249 cursor isn't needed.

create_empty_table(*self*, *project_id*, *dataset_id*, *table_id*, *schema_fields=None*, *time_partitioning=None*, *cluster_fields=None*, *labels=None*, *view=None*, *num_retries=None*)

Creates a new, empty table in the dataset. To create a view, which is defined by a SQL query, parse a dictionary to 'view' kwarg

Parameters

- **project_id** (*str*) – The project to create the table into.
- **dataset_id** (*str*) – The dataset to create the table into.
- **table_id** (*str*) – The Name of the table to be created.
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema
- **labels** (*dict*) – a dictionary containing labels for the table, passed to BigQuery

Example:

```
schema_fields=[{"name": "emp_name", "type": "STRING", "mode": "REQUIRED"},
               {"name": "salary", "type": "INTEGER", "mode": "NULLABLE"}]
```

Parameters

- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications.

  See also:

  https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#timePartitioning
- **cluster_fields** (*list*) – [Optional] The fields used for clustering. Must be specified with time_partitioning, data in the table will be first partitioned and subsequently clustered. https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#clustering.fields
- **view** (*dict*) – [Optional] A dictionary containing definition for the view. If set, it will create a view instead of a table: https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#view

Example:

```
view = {
    "query": "SELECT * FROM `test-project-id.test_dataset_id.test_table_
↪prefix*` LIMIT 1000",
    "useLegacySql": False
}
```

Returns None

create_external_table(*self*, *external_project_dataset_table*, *schema_fields*, *source_uris*, *source_format='CSV'*, *autodetect=False*, *compression='NONE'*, *ignore_unknown_values=False*, *max_bad_records=0*, *skip_leading_rows=0*, *field_delimiter=','*, *quote_character=None*, *allow_quoted_newlines=False*, *allow_jagged_rows=False*, *src_fmt_configs=None*, *labels=None*)

Creates a new external table in the dataset with the data in Google Cloud Storage. See here:

https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#resource

for more details about these parameters.

Parameters

- **external_project_dataset_table** (*str*) – The dotted (<project>.|<project>:)<dataset>.<table>($<partition>) BigQuery table name to create external table. If <project> is not included, project will be the project defined in the connection json.
- **schema_fields** (*list*) – The schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#resource
- **source_uris** (*list*) – The source Google Cloud Storage URI (e.g. gs://some-bucket/some-file.txt). A single wild per-object name can be used.
- **source_format** (*str*) – File format to export.
- **autodetect** (*bool*) – Try to detect schema and format options automatically. Any option specified explicitly will be honored.
- **compression** (*str*) – [Optional] The compression type of the data source. Possible values include GZIP and NONE. The default value is NONE. This setting is ignored for Google Cloud Bigtable, Google Cloud Datastore backups and Avro formats.
- **ignore_unknown_values** (*bool*) – [Optional] Indicates if BigQuery should allow extra values that are not represented in the table schema. If true, the extra values are ignored. If false, records with extra columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **field_delimiter** (*str*) – The delimiter to use when loading from a CSV.
- **quote_character** (*str*) – The value that is used to quote data sections in a CSV file.
- **allow_quoted_newlines** (*bool*) – Whether to allow quoted newlines (true) or not (false).
- **allow_jagged_rows** (*bool*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable when soure_format is CSV.
- **src_fmt_configs** (*dict*) – configure optional fields specific to the source format
- **labels** (*dict*) – a dictionary containing labels for the table, passed to BigQuery

patch_table(*self*, *dataset_id*, *table_id*, *project_id=None*, *description=None*, *expiration_time=None*, *external_data_configuration=None*, *friendly_name=None*, *labels=None*, *schema=None*, *time_partitioning=None*, *view=None*, *require_partition_filter=None*)

Patch information in an existing table. It only updates fileds that are provided in the request object.

Reference: https://cloud.google.com/bigquery/docs/reference/rest/v2/tables/patch

Parameters

- **dataset_id** (*str*) – The dataset containing the table to be patched.
- **table_id** (*str*) – The Name of the table to be patched.
- **project_id** (*str*) – The project containing the table to be patched.
- **description** (*str*) – [Optional] A user-friendly description of this table.
- **expiration_time** (*int*) – [Optional] The time when this table expires, in milliseconds since the epoch.

- **external_data_configuration** (*dict*) – [Optional] A dictionary containing properties of a table stored outside of BigQuery.
- **friendly_name** (*str*) – [Optional] A descriptive name for this table.
- **labels** (*dict*) – [Optional] A dictionary containing labels associated with this table.
- **schema** (*list*) – [Optional] If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema The supported schema modifications and unsupported schema modification are listed here: https://cloud.google.com/bigquery/docs/managing-table-schemas **Example**:

```
schema=[{"name": "emp_name", "type": "STRING", "mode": "REQUIRED"}
↪,
               {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}]
```

- **time_partitioning** (*dict*) – [Optional] A dictionary containing time-based partitioning definition for the table.
- **view** (*dict*) – [Optional] A dictionary containing definition for the view. If set, it will patch a view instead of a table: https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#view **Example**:

```
view = {
    "query": "SELECT * FROM `test-project-id.test_dataset_id.test_
↪table_prefix*` LIMIT 500",
    "useLegacySql": False
}
```

- **require_partition_filter** (*bool*) – [Optional] If true, queries over the this table require a partition filter. If false, queries over the table

run_query (*self*, *sql*, *destination_dataset_table=None*, *write_disposition='WRITE_EMPTY'*, *allow_large_results=False*, *flatten_results=None*, *udf_config=None*, *use_legacy_sql=None*, *maximum_billing_tier=None*, *maximum_bytes_billed=None*, *create_disposition='CREATE_IF_NEEDED'*, *query_params=None*, *labels=None*, *schema_update_options=()*, *priority='INTERACTIVE'*, *time_partitioning=None*, *api_resource_configs=None*, *cluster_fields=None*, *location=None*)

Executes a BigQuery SQL query. Optionally persists results in a BigQuery table. See here:

https://cloud.google.com/bigquery/docs/reference/v2/jobs

For more details about these parameters.

Parameters

- **sql** (*str*) – The BigQuery SQL to execute.
- **destination_dataset_table** (*str*) – The dotted `<dataset>.<table>` BigQuery table to save the query results.
- **write_disposition** (*str*) – What to do if the table already exists in BigQuery.
- **allow_large_results** (*bool*) – Whether to allow large results.
- **flatten_results** (*bool*) – If true and query uses legacy SQL dialect, flattens all nested and repeated fields in the query results. `allowLargeResults` must be true if this is set to false. For standard SQL queries, this flag is ignored and results are never flattened.
- **udf_config** (*list*) – The User Defined Function configuration for the query. See https://cloud.google.com/bigquery/user-defined-functions for details.

- **use_legacy_sql** (*bool*) – Whether to use legacy SQL (true) or standard SQL (false). If *None*, defaults to *self.use_legacy_sql*.
- **api_resource_configs** (*dict*) – a dictionary that contain params 'configuration' applied for Google BigQuery Jobs API: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs for example, {'query': {'useQueryCache': False}}. You could use it if you need to provide some params that are not supported by the BigQueryHook like args.
- **maximum_billing_tier** (*int*) – Positive integer that serves as a multiplier of the basic price.
- **maximum_bytes_billed** (*float*) – Limits the bytes billed for this job. Queries that will have bytes billed beyond this limit will fail (without incurring a charge). If unspecified, this will be set to your project default.
- **create_disposition** (*str*) – Specifies whether the job is allowed to create new tables.
- **query_params** (*list*) – a list of dictionary containing query parameter types and values, passed to BigQuery
- **labels** (*dict*) – a dictionary containing labels for the job/query, passed to BigQuery
- **schema_update_options** (*tuple*) – Allows the schema of the destination table to be updated as a side effect of the query job.
- **priority** (*str*) – Specifies a priority for the query. Possible values include INTERACTIVE and BATCH. The default value is INTERACTIVE.
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications.
- **cluster_fields** (*list[str]*) – Request that the result of this query be stored sorted by one or more columns. This is only available in combination with time_partitioning. The order of columns given determines the sort order.
- **location** (*str*) – The geographic location of the job. Required except for US and EU. See details at https://cloud.google.com/bigquery/docs/locations#specifying_your_location

run_extract(*self*, *source_project_dataset_table*, *destination_cloud_storage_uris*, *compression='NONE'*, *export_format='CSV'*, *field_delimiter=', '*, *print_header=True*, *labels=None*)

Executes a BigQuery extract command to copy data from BigQuery to Google Cloud Storage. See here:

https://cloud.google.com/bigquery/docs/reference/v2/jobs

For more details about these parameters.

Parameters

- **source_project_dataset_table** (*str*) – The dotted `<dataset>.<table>` BigQuery table to use as the source data.
- **destination_cloud_storage_uris** (*list*) – The destination Google Cloud Storage URI (e.g. gs://some-bucket/some-file.txt). Follows convention defined here: https://cloud.google.com/bigquery/exporting-data-from-bigquery#exportingmultiple
- **compression** (*str*) – Type of compression to use.
- **export_format** (*str*) – File format to export.
- **field_delimiter** (*str*) – The delimiter to use when extracting to a CSV.
- **print_header** (*bool*) – Whether to print a header for a CSV file extract.
- **labels** (*dict*) – a dictionary containing labels for the job/query, passed to BigQuery

run_copy(*self*, *source_project_dataset_tables*, *destination_project_dataset_table*, *write_disposition='WRITE_EMPTY'*, *create_disposition='CREATE_IF_NEEDED'*, *labels=None*)

Executes a BigQuery copy command to copy data from one BigQuery table to another. See here:

https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.copy

For more details about these parameters.

Parameters

- **source_project_dataset_tables** (*list|string*) – One or more dotted `(project:|project.)<dataset>.<table>` BigQuery tables to use as the source data. Use a list if there are multiple source tables. If `<project>` is not included, project will be the project defined in the connection json.
- **destination_project_dataset_table** (*str*) – The destination BigQuery table. Format is: `(project:|project.)<dataset>.<table>`
- **write_disposition** (*str*) – The write disposition if the table already exists.
- **create_disposition** (*str*) – The create disposition if the table doesn't exist.
- **labels** (*dict*) – a dictionary containing labels for the job/query, passed to BigQuery

run_load(*self*, *destination_project_dataset_table*, *source_uris*, *schema_fields=None*, *source_format='CSV'*, *create_disposition='CREATE_IF_NEEDED'*, *skip_leading_rows=0*, *write_disposition='WRITE_EMPTY'*, *field_delimiter=','*, *max_bad_records=0*, *quote_character=None*, *ignore_unknown_values=False*, *allow_quoted_newlines=False*, *allow_jagged_rows=False*, *schema_update_options=()*, *src_fmt_configs=None*, *time_partitioning=None*, *cluster_fields=None*, *autodetect=False*)

Executes a BigQuery load command to load data from Google Cloud Storage to BigQuery. See here:

https://cloud.google.com/bigquery/docs/reference/v2/jobs

For more details about these parameters.

Parameters

- **destination_project_dataset_table** (*str*) – The dotted `(<project>.|<project>:)<dataset>.<table>($<partition>)` BigQuery table to load data into. If `<project>` is not included, project will be the project defined in the connection json. If a partition is specified the operator will automatically append the data, create a new partition or create a new DAY partitioned table.
- **schema_fields** (*list*) – The schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load Required if autodetect=False; optional if autodetect=True.
- **autodetect** (*bool*) – Attempt to autodetect the schema for CSV and JSON source files.
- **source_uris** (*list*) – The source Google Cloud Storage URI (e.g. gs://some-bucket/some-file.txt). A single wild per-object name can be used.
- **source_format** (*str*) – File format to export.
- **create_disposition** (*str*) – The create disposition if the table doesn't exist.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **write_disposition** (*str*) – The write disposition if the table already exists.
- **field_delimiter** (*str*) – The delimiter to use when loading from a CSV.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **quote_character** (*str*) – The value that is used to quote data sections in a CSV file.

- **`ignore_unknown_values`** (*`bool`*) – [Optional] Indicates if BigQuery should allow extra values that are not represented in the table schema. If true, the extra values are ignored. If false, records with extra columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result.
- **`allow_quoted_newlines`** (*`bool`*) – Whether to allow quoted newlines (true) or not (false).
- **`allow_jagged_rows`** (*`bool`*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable when soure_format is CSV.
- **`schema_update_options`** (*`tuple`*) – Allows the schema of the destination table to be updated as a side effect of the load job.
- **`src_fmt_configs`** (*`dict`*) – configure optional fields specific to the source format
- **`time_partitioning`** (*`dict`*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications.
- **`cluster_fields`** (*`list[str]`*) – Request that the result of this load be stored sorted by one or more columns. This is only available in combination with time_partitioning. The order of columns given determines the sort order.

`run_with_configuration`(*self*, *configuration*)

Executes a BigQuery SQL query. See here:

https://cloud.google.com/bigquery/docs/reference/v2/jobs

For more details about the configuration parameter.

Parameters **`configuration`** – The configuration parameter maps directly to BigQuery's configuration field in the job object. See https://cloud.google.com/bigquery/docs/reference/v2/jobs for details.

`poll_job_complete`(*self*, *job_id*)

`cancel_query`(*self*)

Cancel all started queries that have not yet completed

`get_schema`(*self*, *dataset_id*, *table_id*)

Get the schema for a given datset.table. see https://cloud.google.com/bigquery/docs/reference/v2/tables#resource

Parameters

- **`dataset_id`** – the dataset ID of the requested table
- **`table_id`** – the table ID of the requested table

Returns a table schema

`get_tabledata`(*self*, *dataset_id*, *table_id*, *max_results=None*, *selected_fields=None*, *page_token=None*, *start_index=None*)

Get the data of a given dataset.table and optionally with selected columns. see https://cloud.google.com/bigquery/docs/reference/v2/tabledata/list

Parameters

- **`dataset_id`** – the dataset ID of the requested table.
- **`table_id`** – the table ID of the requested table.
- **`max_results`** – the maximum results to return.
- **`selected_fields`** – List of fields to return (comma-separated). If unspecified, all fields are returned.

- **page_token** – page token, returned from a previous call, identifying the result set.
- **start_index** – zero based index of the starting row to read.

Returns map containing the requested rows.

run_table_delete (*self*, *deletion_dataset_table*, *ignore_if_missing=False*)
Delete an existing table from the dataset; If the table does not exist, return an error unless ignore_if_missing is set to True.

Parameters

- **deletion_dataset_table** (*str*) – A dotted `(<project>.|<project>:)<dataset>.<table>` that indicates which table will be deleted.
- **ignore_if_missing** (*bool*) – if True, then return success even if the requested table does not exist.

Returns

run_table_upsert (*self*, *dataset_id*, *table_resource*, *project_id=None*)
creates a new, empty table in the dataset; If the table already exists, update the existing table. Since BigQuery does not natively allow table upserts, this is not an atomic operation.

Parameters

- **dataset_id** (*str*) – the dataset to upsert the table into.
- **table_resource** (*dict*) – a table resource. see https://cloud.google.com/bigquery/docs/reference/v2/tables#resource
- **project_id** – the project to upsert the table into. If None, project will be self.project_id.

Returns

run_grant_dataset_view_access (*self*, *source_dataset*, *view_dataset*, *view_table*, *source_project=None*, *view_project=None*)
Grant authorized view access of a dataset to a view table. If this view has already been granted access to the dataset, do nothing. This method is not atomic. Running it may clobber a simultaneous update.

Parameters

- **source_dataset** (*str*) – the source dataset
- **view_dataset** (*str*) – the dataset that the view is in
- **view_table** (*str*) – the table of the view
- **source_project** (*str*) – the project of the source dataset. If None, self.project_id will be used.
- **view_project** (*str*) – the project that the view is in. If None, self.project_id will be used.

Returns the datasets resource of the source dataset.

create_empty_dataset (*self*, *dataset_id=''*, *project_id=''*, *dataset_reference=None*)
Create a new empty dataset: https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets/insert

Parameters

- **project_id** (*str*) – The name of the project where we want to create an empty a dataset. Don't need to provide, if projectId in dataset_reference.
- **dataset_id** (*str*) – The id of dataset. Don't need to provide, if datasetId in dataset_reference.

- **dataset_reference** (*dict*) – Dataset reference that could be provided with request body. More info: https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets#resource

delete_dataset (*self*, *project_id*, *dataset_id*)
Delete a dataset of Big query in your project. :param project_id: The name of the project where we have the dataset . :type project_id: str :param dataset_id: The dataset to be delete. :type dataset_id: str :return:

get_dataset (*self*, *dataset_id*, *project_id=None*)
Method returns dataset_resource if dataset exist and raised 404 error if dataset does not exist

Parameters

- **dataset_id** (*str*) – The BigQuery Dataset ID
- **project_id** (*str*) – The GCP Project ID

Returns

dataset_resource

See also:

For more information, see Dataset Resource content: https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets#resource

get_datasets_list (*self*, *project_id=None*)
Method returns full list of BigQuery datasets in the current project

See also:

For more information, see: https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets/list

Parameters **project_id** (*str*) – Google Cloud Project for which you try to get all datasets

Returns

datasets_list

Example of returned datasets_list:

```
    {
       "kind":"bigquery#dataset",
       "location":"US",
       "id":"your-project:dataset_2_test",
       "datasetReference":{
          "projectId":"your-project",
          "datasetId":"dataset_2_test"
       }
    },
    {
       "kind":"bigquery#dataset",
       "location":"US",
       "id":"your-project:dataset_1_test",
       "datasetReference":{
          "projectId":"your-project",
          "datasetId":"dataset_1_test"
       }
    }
]
```

insert_all (*self*, *project_id*, *dataset_id*, *table_id*, *rows*, *ignore_unknown_values=False*, *skip_invalid_rows=False*, *fail_on_error=False*)
Method to stream data into BigQuery one record at a time without needing to run a load job

See also:

For more information, see: https://cloud.google.com/bigquery/docs/reference/rest/v2/tabledata/insertAll

Parameters

- **project_id** (*str*) – The name of the project where we have the table
- **dataset_id** (*str*) – The name of the dataset where we have the table
- **table_id** (*str*) – The name of the table
- **rows** (*list*) – the rows to insert

Example or rows: rows=[{"json": {"a_key": "a_value_0"}}, {"json": {"a_key": "a_value_1"}}]

Parameters

- **ignore_unknown_values** (*bool*) – [Optional] Accept rows that contain values that do not match the schema. The unknown values are ignored. The default value is false, which treats unknown values as errors.
- **skip_invalid_rows** (*bool*) – [Optional] Insert all valid rows of a request, even if invalid rows exist. The default value is false, which causes the entire request to fail if any invalid rows exist.
- **fail_on_error** (*bool*) – [Optional] Force the task to fail if any errors occur. The default value is false, which indicates the task should not fail even if any insertion errors occur.

class airflow.contrib.hooks.bigquery_hook.**BigQueryCursor**(*service*, *project_id*, *use_legacy_sql=True*, *location=None*, *num_retries=None*)

Bases: *airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor*

A very basic BigQuery PEP 249 cursor implementation. The PyHive PEP 249 implementation was used as a reference:

https://github.com/dropbox/PyHive/blob/master/pyhive/presto.py https://github.com/dropbox/PyHive/blob/master/pyhive/common.py

description

The schema description method is not currently implemented.

rowcount

By default, return -1 to indicate that this is not supported.

arraysize

close(*self*)

By default, do nothing

execute(*self*, *operation*, *parameters=None*)

Executes a BigQuery query, and returns the job ID.

Parameters

- **operation** (*str*) – The query to execute.
- **parameters** (*dict*) – Parameters to substitute into the query.

executemany(*self*, *operation*, *seq_of_parameters*)

Execute a BigQuery query multiple times with different parameters.

Parameters

- **operation** (*str*) – The query to execute.
- **seq_of_parameters** (*list*) – List of dictionary parameters to substitute into the query.

fetchone (*self*)
Fetch the next row of a query result set.

next (*self*)
Helper method for fetchone, which returns the next row from a buffer. If the buffer is empty, attempts to paginate through the result set for the next page, and load it into the buffer.

fetchmany (*self*, *size=None*)
Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available. The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned. An `Error` (or subclass) exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.

fetchall (*self*)
Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples).

get_arraysize (*self*)
Specifies the number of rows to fetch at a time with .fetchmany()

set_arraysize (*self*, *arraysize*)
Specifies the number of rows to fetch at a time with .fetchmany()

setinputsizes (*self*, *sizes*)
Does nothing by default

setoutputsize (*self*, *size*, *column=None*)
Does nothing by default

airflow.contrib.hooks.bigquery_hook.**_bind_parameters** (*operation*, *parameters*)
Helper method that binds parameters to a SQL query.

airflow.contrib.hooks.bigquery_hook.**_escape** (*s*)
Helper method that escapes parameters to a SQL query.

airflow.contrib.hooks.bigquery_hook.**_bq_cast** (*string_field*, *bq_type*)
Helper method that casts a BigQuery row to the appropriate data types.
This is useful because BigQuery returns all fields as strings.

airflow.contrib.hooks.bigquery_hook.**_split_tablename** (*table_input*, *default_project_id*, *var_name=None*)

airflow.contrib.hooks.bigquery_hook.**_cleanse_time_partitioning** (*destination_dataset_table*, *time_partitioning_in*)

airflow.contrib.hooks.bigquery_hook.**_validate_value** (*key*, *value*, *expected_type*)
function to check expected type and raise
error if type is not correct

airflow.contrib.hooks.bigquery_hook.**_api_resource_configs_duplication_check** (*key*, *value*, *config_dict*, *config_dict_name='api_r*

airflow.contrib.hooks.cassandra_hook

Module Contents

class airflow.contrib.hooks.cassandra_hook.**CassandraHook**(*cassandra_conn_id='cassandra_default'*)

Bases: *airflow.hooks.base_hook.BaseHook*, airflow.utils.log.logging_mixin.LoggingMixin

Hook used to interact with Cassandra

Contact points can be specified as a comma-separated string in the 'hosts' field of the connection.

Port can be specified in the port field of the connection.

If SSL is enabled in Cassandra, pass in a dict in the extra field as kwargs for `ssl.wrap_socket()`. For example:

```
{
    'ssl_options' : {
        'ca_certs' : PATH_TO_CA_CERTS
    }
}
```

Default load balancing policy is RoundRobinPolicy. To specify a different LB policy:

```
- DCAwareRoundRobinPolicy
    {
        'load_balancing_policy': 'DCAwareRoundRobinPolicy',
        'load_balancing_policy_args': {
            'local_dc': LOCAL_DC_NAME,                      // optional
            'used_hosts_per_remote_dc': SOME_INT_VALUE,     // optional
        }
     }
- WhiteListRoundRobinPolicy
    {
        'load_balancing_policy': 'WhiteListRoundRobinPolicy',
        'load_balancing_policy_args': {
            'hosts': ['HOST1', 'HOST2', 'HOST3']
        }
    }
- TokenAwarePolicy
    {
        'load_balancing_policy': 'TokenAwarePolicy',
        'load_balancing_policy_args': {
            'child_load_balancing_policy': CHILD_POLICY_NAME, // optional
            'child_load_balancing_policy_args': { ... }       // optional
        }
    }
```

For details of the Cluster config, see cassandra.cluster.

get_conn(*self*)
Returns a cassandra Session object

get_cluster(*self*)

shutdown_cluster(*self*)
Closes all sessions and connections associated with this Cluster.

static get_lb_policy(*policy_name*, *policy_args*)

table_exists(*self*, *table*)
Checks if a table exists in Cassandra

Parameters **table** (*str*) – Target Cassandra table. Use dot notation to target a specific keyspace.

record_exists (*self*, *table*, *keys*)
Checks if a record exists in Cassandra

Parameters

- **table** (*str*) – Target Cassandra table. Use dot notation to target a specific keyspace.
- **keys** (*dict*) – The keys and their values to check the existence.

airflow.contrib.hooks.cloudant_hook

Module Contents

class airflow.contrib.hooks.cloudant_hook.**CloudantHook**(*cloudant_conn_id='cloudant_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interact with Cloudant. This class is a thin wrapper around the cloudant python library.

See also:

the latest documentation here.

Parameters **cloudant_conn_id** (*str*) – The connection id to authenticate and get a session object from cloudant.

get_conn(*self*)
Opens a connection to the cloudant service and closes it automatically if used as context manager.

Note: In the connection form: - 'host' equals the 'Account' (optional) - 'login' equals the 'Username (or API Key)' (required) - 'password' equals the 'Password' (required)

Returns an authorized cloudant session context manager object.

Return type cloudant

_validate_connection(*self*, *conn*)

airflow.contrib.hooks.databricks_hook

Module Contents

airflow.contrib.hooks.databricks_hook.**RESTART_CLUSTER_ENDPOINT = ['POST', 'api/2.0/clusters**

airflow.contrib.hooks.databricks_hook.**START_CLUSTER_ENDPOINT = ['POST', 'api/2.0/clusters/s**

airflow.contrib.hooks.databricks_hook.**TERMINATE_CLUSTER_ENDPOINT = ['POST', 'api/2.0/cluste**

airflow.contrib.hooks.databricks_hook.**RUN_NOW_ENDPOINT = ['POST', 'api/2.0/jobs/run-now']**

airflow.contrib.hooks.databricks_hook.**SUBMIT_RUN_ENDPOINT = ['POST', 'api/2.0/jobs/runs/su**

airflow.contrib.hooks.databricks_hook.**GET_RUN_ENDPOINT = ['GET', 'api/2.0/jobs/runs/get']**

airflow.contrib.hooks.databricks_hook.**CANCEL_RUN_ENDPOINT = ['POST', 'api/2.0/jobs/runs/ca**

airflow.contrib.hooks.databricks_hook.**USER_AGENT_HEADER**

class `airflow.contrib.hooks.databricks_hook.`**`DatabricksHook`**(*databricks_conn_id='databricks_default', timeout_seconds=180, retry_limit=3, retry_delay=1.0*)

Bases: `airflow.hooks.base_hook.BaseHook`

Interact with Databricks.

static `_parse_host`(*host*)

The purpose of this function is to be robust to improper connections settings provided by users, specifically in the host field.

For example – when users supply `https://xx.cloud.databricks.com` as the host, we must strip out the protocol to get the host.:

```
h = DatabricksHook()
assert h._parse_host('https://xx.cloud.databricks.com') ==
↪'xx.cloud.databricks.com'
```

In the case where users supply the correct `xx.cloud.databricks.com` as the host, this function is a no-op.:

```
assert h._parse_host('xx.cloud.databricks.com') == 'xx.cloud.databricks.com'
```

`_do_api_call`(*self, endpoint_info, json*)

Utility function to perform an API call with retries

Parameters

- **`endpoint_info`** (`tuple[string, string]`) – Tuple of method and endpoint
- **`json`** (`dict`) – Parameters for this API call.

Returns If the api call returns a OK status code, this function returns the response in JSON. Otherwise, we throw an AirflowException.

Return type dict

`_log_request_error`(*self, attempt_num, error*)

`run_now`(*self, json*)

Utility function to call the `api/2.0/jobs/run-now` endpoint.

Parameters **`json`** (`dict`) – The data used in the body of the request to the `run-now` endpoint.

Returns the run_id as a string

Return type str

`submit_run`(*self, json*)

Utility function to call the `api/2.0/jobs/runs/submit` endpoint.

Parameters **`json`** (`dict`) – The data used in the body of the request to the `submit` endpoint.

Returns the run_id as a string

Return type str

`get_run_page_url`(*self, run_id*)

`get_run_state`(*self, run_id*)

`cancel_run`(*self, run_id*)

`restart_cluster`(*self, json*)

`start_cluster`(*self, json*)

terminate_cluster(*self*, *json*)

airflow.contrib.hooks.databricks_hook.**_retryable_error**(*exception*)

airflow.contrib.hooks.databricks_hook.**RUN_LIFE_CYCLE_STATES = ['PENDING', 'RUNNING', 'TERMI**

class airflow.contrib.hooks.databricks_hook.**RunState**(*life_cycle_state*, *result_state*, *state_message*)

Utility class for the run state concept of Databricks runs.

is_terminal

is_successful

__eq__(*self*, *other*)

__repr__(*self*)

class airflow.contrib.hooks.databricks_hook.**_TokenAuth**(*token*)

Bases: `requests.auth.AuthBase`

Helper class for requests Auth field. AuthBase requires you to implement the __call__ magic function.

__call__(*self*, *r*)

airflow.contrib.hooks.datadog_hook

Module Contents

class airflow.contrib.hooks.datadog_hook.**DatadogHook**(*datadog_conn_id='datadog_default'*)

Bases: *airflow.hooks.base_hook.BaseHook*, airflow.utils.log.logging_mixin.LoggingMixin

Uses datadog API to send metrics of practically anything measurable, so it's possible to track # of db records inserted/deleted, records read from file and many other useful metrics.

Depends on the datadog API, which has to be deployed on the same server where Airflow runs.

Parameters

- **datadog_conn_id** – The connection to datadog, containing metadata for api keys.
- **datadog_conn_id** – str

validate_response(*self*, *response*)

send_metric(*self*, *metric_name*, *datapoint*, *tags=None*, *type_=None*, *interval=None*)

Sends a single datapoint metric to DataDog

Parameters

- **metric_name** (*str*) – The name of the metric
- **datapoint** (*int or float*) – A single integer or float related to the metric
- **tags** (*list*) – A list of tags associated with the metric
- **type** (*str*) – Type of your metric: gauge, rate, or count
- **interval** (*int*) – If the type of the metric is rate or count, define the corresponding interval

query_metric(*self*, *query*, *from_seconds_ago*, *to_seconds_ago*)

Queries datadog for a specific metric, potentially with some function applied to it and returns the results.

Parameters

- **query** (*str*) – The datadog query to execute (see datadog docs)

- **from_seconds_ago** (*int*) – How many seconds ago to start querying for.
- **to_seconds_ago** (*int*) – Up to how many seconds ago to query for.

post_event (*self*, *title*, *text*, *aggregation_key=None*, *alert_type=None*, *date_happened=None*, *handle=None*, *priority=None*, *related_event_id=None*, *tags=None*, *device_name=None*)

Posts an event to datadog (processing finished, potentially alerts, other issues) Think about this as a means to maintain persistence of alerts, rather than alerting itself.

Parameters

- **title** (*str*) – The title of the event
- **text** (*str*) – The body of the event (more information)
- **aggregation_key** (*str*) – Key that can be used to aggregate this event in a stream
- **alert_type** (*str*) – The alert type for the event, one of ["error", "warning", "info", "success"]
- **date_happened** (*int*) – POSIX timestamp of the event; defaults to now
- **handle** – str
- **priority** (*str*) – Priority to post the event as. ("normal" or "low", defaults to "normal")
- **related_event_id** (*id*) – Post event as a child of the given event
- **tags** (*list[str]*) – List of tags to apply to the event
- **device_name** (*list*) – device_name to post the event with

Handle User to post the event as; defaults to owner of the application key used to submit.

airflow.contrib.hooks.datastore_hook

Module Contents

class airflow.contrib.hooks.datastore_hook.**DatastoreHook** (*datastore_conn_id='google_cloud_default'*, *delegate_to=None*, *api_version='v1'*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Interact with Google Cloud Datastore. This hook uses the Google Cloud Platform connection.

This object is not threads safe. If you want to make multiple requests simultaneously, you will need to create a hook per thread.

Parameters **api_version** (*str*) – The version of the API it is going to connect to.

get_conn (*self*)

Establishes a connection to the Google API.

Returns a Google Cloud Datastore service object.

Return type Resource

allocate_ids (*self*, *partial_keys*)

Allocate IDs for incomplete keys.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/allocateIds

Parameters **partial_keys** (*list*) – a list of partial keys.

Returns a list of full keys.

Return type list

begin_transaction (*self*)
Begins a new transaction.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/beginTransaction

Returns a transaction handle.

Return type str

commit (*self*, *body*)
Commit a transaction, optionally creating, deleting or modifying some entities.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/commit

Parameters **body** (*dict*) – the body of the commit request.

Returns the response body of the commit request.

Return type dict

lookup (*self*, *keys*, *read_consistency=None*, *transaction=None*)
Lookup some entities by key.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/lookup

Parameters

- **keys** (*list*) – the keys to lookup.
- **read_consistency** (*str*) – the read consistency to use. default, strong or eventual. Cannot be used with a transaction.
- **transaction** (*str*) – the transaction to use, if any.

Returns the response body of the lookup request.

Return type dict

rollback (*self*, *transaction*)
Roll back a transaction.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/rollback

Parameters **transaction** (*str*) – the transaction to roll back.

run_query (*self*, *body*)
Run a query for entities.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/runQuery

Parameters **body** (*dict*) – the body of the query request.

Returns the batch of query results.

Return type dict

get_operation(*self*, *name*)
Gets the latest state of a long-running operation.

See also:

https://cloud.google.com/datastore/docs/reference/data/rest/v1/projects.operations/get

Parameters **name** (*str*) – the name of the operation resource.

Returns a resource operation instance.

Return type dict

delete_operation(*self*, *name*)
Deletes the long-running operation.

See also:

https://cloud.google.com/datastore/docs/reference/data/rest/v1/projects.operations/delete

Parameters **name** (*str*) – the name of the operation resource.

Returns none if successful.

Return type dict

poll_operation_until_done(*self*, *name*, *polling_interval_in_seconds*)
Poll backup operation state until it's completed.

Parameters

- **name** (*str*) – the name of the operation resource
- **polling_interval_in_seconds** (*int*) – The number of seconds to wait before calling another request.

Returns a resource operation instance.

Return type dict

export_to_storage_bucket(*self*, *bucket*, *namespace=None*, *entity_filter=None*, *labels=None*)
Export entities from Cloud Datastore to Cloud Storage for backup.

Note: Keep in mind that this requests the Admin API not the Data API.

See also:

https://cloud.google.com/datastore/docs/reference/admin/rest/v1/projects/export

Parameters

- **bucket** (*str*) – The name of the Cloud Storage bucket.
- **namespace** (*str*) – The Cloud Storage namespace path.
- **entity_filter** (*dict*) – Description of what data from the project is included in the export.
- **labels** (*dict of str*) – Client-assigned labels.

Returns a resource operation instance.

Return type dict

import_from_storage_bucket(*self*, *bucket*, *file*, *namespace=None*, *entity_filter=None*, *labels=None*)

Import a backup from Cloud Storage to Cloud Datastore.

Note: Keep in mind that this requests the Admin API not the Data API.

See also:

https://cloud.google.com/datastore/docs/reference/admin/rest/v1/projects/import

Parameters

- **bucket** (*str*) – The name of the Cloud Storage bucket.
- **file** (*str*) – the metadata file written by the projects.export operation.
- **namespace** (*str*) – The Cloud Storage namespace path.
- **entity_filter** (*dict*) – specify which kinds/namespaces are to be imported.
- **labels** (*dict of str*) – Client-assigned labels.

Returns a resource operation instance.

Return type dict

airflow.contrib.hooks.dingding_hook

Module Contents

class airflow.contrib.hooks.dingding_hook.**DingdingHook**(*dingding_conn_id='dingding_default'*, *message_type='text'*, *message=None*, *at_mobiles=None*, *at_all=False*, **args*, ***kwargs*)

Bases: `airflow.hooks.http_hook.HttpHook`

This hook allows you send Dingding message using Dingding custom bot. Get Dingding token from conn_id.password. And prefer set domain to conn_id.host, if not will use default `https://oapi.dingtalk.com`.

For more detail message in Dingding custom bot

Parameters

- **dingding_conn_id** (*str*) – The name of the Dingding connection to use
- **message_type** (*str*) – Message type you want to send to Dingding, support five type so far including text, link, markdown, actionCard, feedCard
- **message** (*str or dict*) – The message send to Dingding chat group
- **at_mobiles** (*list[str]*) – Remind specific users with this message
- **at_all** (*bool*) – Remind all people in group or not. If True, will overwrite `at_mobiles`

_get_endpoint(*self*)

Get Dingding endpoint for sending message.

_build_message(*self*)

Build different type of Dingding message As most commonly used type, text message just need post message content rather than a dict like `{'content': 'message'}`

get_conn (*self*, *headers=None*)
Overwrite HttpHook get_conn because just need base_url and headers and not don't need generic params

Parameters **headers** (*dict*) – additional headers to be passed through as a dictionary

send (*self*)
Send Dingding message

airflow.contrib.hooks.discord_webhook_hook

Module Contents

class airflow.contrib.hooks.discord_webhook_hook.**DiscordWebhookHook** (*http_conn_id=None*, *webhook_endpoint=None*, *message=''*, *username=None*, *avatar_url=None*, *tts=False*, *proxy=None*, **args*, ***kwargs*)

Bases: *airflow.hooks.http_hook.HttpHook*

This hook allows you to post messages to Discord using incoming webhooks. Takes a Discord connection ID with a default relative webhook endpoint. The default endpoint can be overridden using the webhook_endpoint parameter (https://discordapp.com/developers/docs/resources/webhook).

Each Discord webhook can be pre-configured to use a specific username and avatar_url. You can override these defaults in this hook.

Parameters

- **http_conn_id** (*str*) – Http connection ID with host as "https://discord.com/api/" and default webhook endpoint in the extra field in the form of {"webhook_endpoint": "webhooks/{webhook.id}/{webhook.token}"}
- **webhook_endpoint** (*str*) – Discord webhook endpoint in the form of "webhooks/{webhook.id}/{webhook.token}"
- **message** (*str*) – The message you want to send to your Discord channel (max 2000 characters)
- **username** (*str*) – Override the default username of the webhook
- **avatar_url** (*str*) – Override the default avatar of the webhook
- **tts** (*bool*) – Is a text-to-speech message
- **proxy** (*str*) – Proxy to use to make the Discord webhook call

_get_webhook_endpoint (*self*, *http_conn_id*, *webhook_endpoint*)
Given a Discord http_conn_id, return the default webhook endpoint or override if a webhook_endpoint is manually supplied.

Parameters

- **http_conn_id** – The provided connection ID
- **webhook_endpoint** – The manually provided webhook endpoint

Returns Webhook endpoint (str) to use

_build_discord_payload(*self*)
Construct the Discord JSON payload. All relevant parameters are combined here to a valid Discord JSON payload.

Returns Discord payload (str) to send

execute(*self*)
Execute the Discord webhook call

airflow.contrib.hooks.emr_hook

Module Contents

class airflow.contrib.hooks.emr_hook.**EmrHook**(*emr_conn_id=None*, *region_name=None*, **args*, ***kwargs*)
Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS EMR. emr_conn_id is only necessary for using the create_job_flow method.

get_conn(*self*)

create_job_flow(*self*, *job_flow_overrides*)
Creates a job flow using the config from the EMR connection. Keys of the json extra hash may have the arguments of the boto3 run_job_flow method. Overrides for this config may be passed as the job_flow_overrides.

airflow.contrib.hooks.fs_hook

Module Contents

class airflow.contrib.hooks.fs_hook.**FSHook**(*conn_id='fs_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Allows for interaction with an file server.

Connection should have a name and a path specified under extra:

example: Conn Id: fs_test Conn Type: File (path) Host, Schema, Login, Password, Port: empty Extra: {"path": "/tmp"}

get_conn(*self*)

get_path(*self*)

airflow.contrib.hooks.ftp_hook

Module Contents

airflow.contrib.hooks.ftp_hook.**mlsd**(*conn*, *path=''*, *facts=None*)
BACKPORT FROM PYTHON3 FTPLIB.
List a directory in a standardized format by using MLSD command (RFC-3659). If path is omitted the current directory is assumed. "facts" is a list of strings representing the type of information desired (e.g. ["type", "size", "perm"]).

Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary including a variable number of "facts" depending on the server and whether "facts" argument has been provided.

class airflow.contrib.hooks.ftp_hook.**FTPHook**(*ftp_conn_id='ftp_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interact with FTP.

Errors that may occur throughout but should be handled downstream. You can specify mode for data transfers in the extra field of your connection as `{"passive": "true"}`.

`__enter__`(*self*)

`__exit__`(*self*, *exc_type*, *exc_val*, *exc_tb*)

`get_conn`(*self*)
Returns a FTP connection object

`close_conn`(*self*)
Closes the connection. An error will occur if the connection wasn't ever opened.

`describe_directory`(*self*, *path*)
Returns a dictionary of {filename: {attributes}} for all files on the remote system (where the MLSD command is supported).

Parameters **path** (*str*) – full path to the remote directory

`list_directory`(*self*, *path*, *nlst=False*)
Returns a list of files on the remote system.

Parameters **path** (*str*) – full path to the remote directory to list

`create_directory`(*self*, *path*)
Creates a directory on the remote system.

Parameters **path** (*str*) – full path to the remote directory to create

`delete_directory`(*self*, *path*)
Deletes a directory on the remote system.

Parameters **path** (*str*) – full path to the remote directory to delete

`retrieve_file`(*self*, *remote_full_path*, *local_full_path_or_buffer*, *callback=None*)
Transfers the remote file to a local location.

If local_full_path_or_buffer is a string path, the file will be put at that location; if it is a file-like buffer, the file will be written to the buffer but not closed.

Parameters

- **remote_full_path** (*str*) – full path to the remote file
- **local_full_path_or_buffer** (*str or file-like buffer*) – full path to the local file or a file-like buffer
- **callback** (*callable*) – callback which is called each time a block of data is read. if you do not use a callback, these blocks will be written to the file or buffer passed in. if you do pass in a callback, note that writing to a file or buffer will need to be handled inside the callback. [default: output_handle.write()]

:Example:

```
hook = FTPHook(ftp_conn_id='my_conn')

remote_path = '/path/to/remote/file'
local_path = '/path/to/local/file'

# with a custom callback (in this case displaying progress on each read)
def print_progress(percent_progress):
```

(continues on next page)

(continued from previous page)

```
    self.log.info('Percent Downloaded: %s%%' % percent_progress)

total_downloaded = 0
total_file_size = hook.get_size(remote_path)
output_handle = open(local_path, 'wb')
def write_to_file_with_progress(data):
    total_downloaded += len(data)
    output_handle.write(data)
    percent_progress = (total_downloaded / total_file_size) * 100
    print_progress(percent_progress)
hook.retrieve_file(remote_path, None, callback=write_to_file_with_progress)

# without a custom callback data is written to the local_path
hook.retrieve_file(remote_path, local_path)
```

store_file (*self*, *remote_full_path*, *local_full_path_or_buffer*)
Transfers a local file to the remote location.

If local_full_path_or_buffer is a string path, the file will be read from that location; if it is a file-like buffer, the file will be read from the buffer but not closed.

Parameters

- **remote_full_path** (*str*) – full path to the remote file
- **local_full_path_or_buffer** (*str or file-like buffer*) – full path to the local file or a file-like buffer

delete_file (*self*, *path*)
Removes a file on the FTP Server.

Parameters **path** (*str*) – full path to the remote file

rename (*self*, *from_name*, *to_name*)
Rename a file.

Parameters

- **from_name** – rename file from name
- **to_name** – rename file to name

get_mod_time (*self*, *path*)
Returns a datetime object representing the last time the file was modified

Parameters **path** (*string*) – remote file path

get_size (*self*, *path*)
Returns the size of a file (in bytes)

Parameters **path** (*string*) – remote file path

class airflow.contrib.hooks.ftp_hook.**FTPSHook**
Bases: *airflow.contrib.hooks.ftp_hook.FTPHook*

get_conn (*self*)
Returns a FTPS connection object.

airflow.contrib.hooks.gcp_api_base_hook

Module Contents

airflow.contrib.hooks.gcp_api_base_hook.**_DEFAULT_SCOPES = ['https://www.googleapis.com/aut**

airflow.contrib.hooks.gcp_api_base_hook.**_G_APP_CRED_ENV_VAR = GOOGLE_APPLICATION_CREDENTIA**

class airflow.contrib.hooks.gcp_api_base_hook.**GoogleCloudBaseHook**(*gcp_conn_id='google_cloud_default', delegate_to=None*)

Bases: *airflow.hooks.base_hook.BaseHook*

A base hook for Google cloud-related hooks. Google cloud has a shared REST API client that is built in the same way no matter which service you use. This class helps construct and authorize the credentials needed to then call googleapiclient.discovery.build() to actually discover and build a client for a Google cloud service.

The class also contains some miscellaneous helper functions.

All hook derived from this base hook use the 'Google Cloud Platform' connection type. Three ways of authentication are supported:

Default credentials: Only the 'Project Id' is required. You'll need to have set up default credentials, such as by the `GOOGLE_APPLICATION_DEFAULT` environment variable or from the metadata server on Google Compute Engine.

JSON key file: Specify 'Project Id', 'Keyfile Path' and 'Scope'.

Legacy P12 key files are not supported.

JSON data provided in the UI: Specify 'Keyfile JSON'.

Parameters

- **gcp_conn_id** (*str*) – The connection ID to use when fetching connection info.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class _Decorators

A private inner class for keeping all decorator methods.

static provide_gcp_credential_file(*func*)

Function decorator that provides a GOOGLE_APPLICATION_CREDENTIALS environment variable, pointing to file path of a JSON file of service account key.

project_id

_get_credentials(*self*)

Returns the Credentials object for Google API

_get_access_token(*self*)

Returns a valid access token from Google API Credentials

_authorize(*self*)

Returns an authorized HTTP object to be used to build a Google cloud service hook connection.

_get_field(*self, f, default=None*)

Fetches a field from extras, and returns it. This is some Airflow magic. The google_cloud_platform hook type adds custom UI elements to the hook page, which allow admins to specify service_account, key_path, etc. They get formatted as shown below.

static catch_http_exception(*func*)

Function decorator that intercepts HTTP Errors and raises AirflowException with more informative message.

static fallback_to_default_project_id(*func*)
Decorator that provides fallback for Google Cloud Platform project id. If the project is None it will be replaced with the project_id from the service account the Hook is authenticated with. Project id can be specified either via project_id kwarg or via first parameter in positional args.

Parameters **func** – function to wrap

Returns result of the function call

_get_project_id(*self*, *project_id*)
In case project_id is None, overrides it with default project_id from the service account that is authorized.

Parameters **project_id** (*str*) – project id to

Returns the project_id specified or default project id if project_id is None

airflow.contrib.hooks.gcp_bigtable_hook

Module Contents

class airflow.contrib.hooks.gcp_bigtable_hook.**BigtableHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud Bigtable APIs.

All the methods in the hook where project_id is used must be called with keyword arguments rather than positional.

_client

_get_client(*self*, *project_id*)

get_instance(*self*, *instance_id*, *project_id=None*)
Retrieves and returns the specified Cloud Bigtable instance if it exists. Otherwise, returns None.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance.
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the BigTable exists. If set to None or missing, the default project_id from the GCP connection is used.

delete_instance(*self*, *instance_id*, *project_id=None*)
Deletes the specified Cloud Bigtable instance. Raises google.api_core.exceptions.NotFound if the Cloud Bigtable instance does not exist.

Parameters

- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the BigTable exists. If set to None or missing, the default project_id from the GCP connection is used.
- **instance_id** (*str*) – The ID of the Cloud Bigtable instance.

create_instance(*self*, *instance_id*, *main_cluster_id*, *main_cluster_zone*, *project_id=None*, *replica_cluster_id=None*, *replica_cluster_zone=None*, *instance_display_name=None*, *instance_type=enums.Instance.Type.TYPE_UNSPECIFIED*, *instance_labels=None*, *cluster_nodes=None*, *cluster_storage_type=enums.StorageType.STORAGE_TYPE_UNSPECIFIED*, *timeout=None*)
Creates new instance.

Parameters

- **instance_id** (*str*) – The ID for the new instance.

- **main_cluster_id** (*str*) – The ID for main cluster for the new instance.
- **main_cluster_zone** (*str*) – The zone for main cluster. See https://cloud.google.com/bigtable/docs/locations for more details.
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the BigTable exists. If set to None or missing, the default project_id from the GCP connection is used.
- **replica_cluster_id** (*str*) – (optional) The ID for replica cluster for the new instance.
- **replica_cluster_zone** (*str*) – (optional) The zone for replica cluster.
- **instance_type** (*enums.Instance.Type*) – (optional) The type of the instance.
- **instance_display_name** (*str*) – (optional) Human-readable name of the instance. Defaults to `instance_id`.
- **instance_labels** (*dict*) – (optional) Dictionary of labels to associate with the instance.
- **cluster_nodes** (*int*) – (optional) Number of nodes for cluster.
- **cluster_storage_type** (*enums.StorageType*) – (optional) The type of storage.
- **timeout** (*int*) – (optional) timeout (in seconds) for instance creation. If None is not specified, Operator will wait indefinitely.

static create_table(*instance*, *table_id*, *initial_split_keys=None*, *column_families=None*)

Creates the specified Cloud Bigtable table. Raises `google.api_core.exceptions.AlreadyExists` if the table exists.

Parameters

- **instance** (*Instance*) – The Cloud Bigtable instance that owns the table.
- **table_id** (*str*) – The ID of the table to create in Cloud Bigtable.
- **initial_split_keys** (*list*) – (Optional) A list of row keys in bytes to use to initially split the table.
- **column_families** (*dict*) – (Optional) A map of columns to create. The key is the column_id str, and the value is a `google.cloud.bigtable.column_family.GarbageCollectionRule`.

delete_table(*self*, *instance_id*, *table_id*, *project_id=None*)

Deletes the specified table in Cloud Bigtable. Raises google.api_core.exceptions.NotFound if the table does not exist.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Bigtable instance.
- **table_id** (*str*) – The ID of the table in Cloud Bigtable.
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the BigTable exists. If set to None or missing, the default project_id from the GCP connection is used.

static update_cluster(*instance*, *cluster_id*, *nodes*)

Updates number of nodes in the specified Cloud Bigtable cluster. Raises google.api_core.exceptions.NotFound if the cluster does not exist.

Parameters

- **instance** (*Instance*) – The Cloud Bigtable instance that owns the cluster.
- **cluster_id** (*str*) – The ID of the cluster.
- **nodes** (*int*) – The desired number of nodes.

static get_column_families_for_table(*instance*, *table_id*)
Fetches Column Families for the specified table in Cloud Bigtable.

Parameters

- **instance** (*Instance*) – The Cloud Bigtable instance that owns the table.
- **table_id** (*str*) – The ID of the table in Cloud Bigtable to fetch Column Families from.

static get_cluster_states_for_table(*instance*, *table_id*)
Fetches Cluster States for the specified table in Cloud Bigtable. Raises google.api_core.exceptions.NotFound if the table does not exist.

Parameters

- **instance** (*Instance*) – The Cloud Bigtable instance that owns the table.
- **table_id** (*str*) – The ID of the table in Cloud Bigtable to fetch Cluster States from.

airflow.contrib.hooks.gcp_cloud_build_hook

Hook for Google Cloud Build service

Module Contents

airflow.contrib.hooks.gcp_cloud_build_hook.**TIME_TO_SLEEP_IN_SECONDS = 5**

class airflow.contrib.hooks.gcp_cloud_build_hook.**CloudBuildHook**(*api_version='v1'*, *gcp_conn_id='google_cloud_default'*, *delegate_to=None*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for the Google Cloud Build APIs.

All the methods in the hook where project_id is used must be called with keyword arguments rather than positional.

Parameters

- **api_version** (*str*) – API version used (for example v1 or v1beta1).
- **gcp_conn_id** (*str*) – The connection ID to use when fetching connection info.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

_conn

get_conn(*self*)
Retrieves the connection to Cloud Functions.

Returns Google Cloud Build services object.

create_build(*self*, *body*, *project_id=None*)
Starts a build with the specified configuration.

Parameters

- **body** (*dict*) – The request body. See: https://cloud.google.com/cloud-build/docs/api/reference/rest/Shared.Types/Build
- **project_id** (*str*) – Optional, Google Cloud Project project_id where the function belongs. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

`_wait_for_operation_to_complete`(*self*, *operation_name*)
Waits for the named operation to complete - checks status of the asynchronous call.

Parameters **`operation_name`** (*`str`*) – The name of the operation.

Returns The response returned by the operation.

Return type dict

Exception AirflowException in case error is returned.

`airflow.contrib.hooks.gcp_compute_hook`

Module Contents

`airflow.contrib.hooks.gcp_compute_hook.`**`TIME_TO_SLEEP_IN_SECONDS = 1`**

`class` `airflow.contrib.hooks.gcp_compute_hook.`**`GceOperationStatus`**

`PENDING = PENDING`

`RUNNING = RUNNING`

`DONE = DONE`

`class` `airflow.contrib.hooks.gcp_compute_hook.`**`GceHook`**(*api_version='v1'*, *gcp_conn_id='google_cloud_default'*, *delegate_to=None*)
Bases: *`airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook`*

Hook for Google Compute Engine APIs.

All the methods in the hook where project_id is used must be called with keyword arguments rather than positional.

`_conn`

`get_conn`(*self*)
Retrieves connection to Google Compute Engine.

Returns Google Compute Engine services object

Return type dict

`start_instance`(*self*, *zone*, *resource_id*, *project_id=None*)
Starts an existing instance defined by project_id, zone and resource_id. Must be called with keyword arguments rather than positional.

Parameters

- **`zone`** (*`str`*) – Google Cloud Platform zone where the instance exists
- **`resource_id`** (*`str`*) – Name of the Compute Engine instance resource
- **`project_id`** (*`str`*) – Optional, Google Cloud Platform project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

`stop_instance`(*self*, *zone*, *resource_id*, *project_id=None*)
Stops an instance defined by project_id, zone and resource_id Must be called with keyword arguments rather than positional.

Parameters

- **zone** (*str*) – Google Cloud Platform zone where the instance exists
- **resource_id** (*str*) – Name of the Compute Engine instance resource
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

set_machine_type (*self*, *zone*, *resource_id*, *body*, *project_id=None*)

Sets machine type of an instance defined by project_id, zone and resource_id. Must be called with keyword arguments rather than positional.

Parameters

- **zone** (*str*) – Google Cloud Platform zone where the instance exists.
- **resource_id** (*str*) – Name of the Compute Engine instance resource
- **body** (*dict*) – Body required by the Compute Engine setMachineType API, as described in https://cloud.google.com/compute/docs/reference/rest/v1/instances/setMachineType
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

_execute_set_machine_type (*self*, *zone*, *resource_id*, *body*, *project_id*)

get_instance_template (*self*, *resource_id*, *project_id=None*)

Retrieves instance template by project_id and resource_id. Must be called with keyword arguments rather than positional.

Parameters

- **resource_id** (*str*) – Name of the instance template
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.

Returns Instance template representation as object according to https://cloud.google.com/compute/docs/reference/rest/v1/instanceTemplates

Return type dict

insert_instance_template (*self*, *body*, *request_id=None*, *project_id=None*)

Inserts instance template using body specified Must be called with keyword arguments rather than positional.

Parameters

- **body** (*dict*) – Instance template representation as object according to https://cloud.google.com/compute/docs/reference/rest/v1/instanceTemplates
- **request_id** (*str*) – Optional, unique request_id that you might add to achieve full idempotence (for example when client call times out repeating the request with the same request id will not create a new instance template again) It should be in UUID format as defined in RFC 4122
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

get_instance_group_manager(*self*, *zone*, *resource_id*, *project_id=None*)
Retrieves Instance Group Manager by project_id, zone and resource_id. Must be called with keyword arguments rather than positional.

Parameters

- **zone** (*str*) – Google Cloud Platform zone where the Instance Group Manager exists
- **resource_id** (*str*) – Name of the Instance Group Manager
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.

Returns Instance group manager representation as object according to https://cloud.google.com/compute/docs/reference/rest/beta/instanceGroupManagers

Return type dict

patch_instance_group_manager(*self*, *zone*, *resource_id*, *body*, *request_id=None*, *project_id=None*)
Patches Instance Group Manager with the specified body. Must be called with keyword arguments rather than positional.

Parameters

- **zone** (*str*) – Google Cloud Platform zone where the Instance Group Manager exists
- **resource_id** (*str*) – Name of the Instance Group Manager
- **body** (*dict*) – Instance Group Manager representation as json-merge-patch object according to https://cloud.google.com/compute/docs/reference/rest/beta/instanceTemplates/patch
- **request_id** (*str*) – Optional, unique request_id that you might add to achieve full idempotence (for example when client call times out repeating the request with the same request id will not create a new instance template again). It should be in UUID format as defined in RFC 4122
- **project_id** (*str*) – Optional, Google Cloud Platform project ID where the Compute Engine Instance exists. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

_wait_for_operation_to_complete(*self*, *project_id*, *operation_name*, *zone=None*)
Waits for the named operation to complete - checks status of the async call.

Parameters

- **operation_name** (*str*) – name of the operation
- **zone** (*str*) – optional region of the request (might be None for global operations)

Returns None

static _check_zone_operation_status(*service*, *operation_name*, *project_id*, *zone*, *num_retries*)

static _check_global_operation_status(*service*, *operation_name*, *project_id*, *num_retries*)

airflow.contrib.hooks.gcp_container_hook

Module Contents

airflow.contrib.hooks.gcp_container_hook.**OPERATIONAL_POLL_INTERVAL = 15**

class airflow.contrib.hooks.gcp_container_hook.**GKEClusterHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*, *location=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

get_client(*self*)

static _dict_to_proto(*py_dict*, *proto*)

Converts a python dictionary to the proto supplied

Parameters

- **py_dict** (*dict*) – The dictionary to convert
- **proto** (*protobuf*) – The proto object to merge with dictionary

Returns A parsed python dictionary in provided proto format

Raises ParseError: On JSON parsing problems.

wait_for_operation(*self*, *operation*, *project_id=None*)

Given an operation, continuously fetches the status from Google Cloud until either completion or an error occurring

Parameters

- **operation** (*google.cloud.container_V1.gapic.enums.Operation*) – The Operation to wait for
- **project_id** (*str*) – Google Cloud Platform project ID

Returns A new, updated operation fetched from Google Cloud

get_operation(*self*, *operation_name*, *project_id=None*)

Fetches the operation from Google Cloud

Parameters

- **operation_name** (*str*) – Name of operation to fetch
- **project_id** (*str*) – Google Cloud Platform project ID

Returns The new, updated operation from Google Cloud

static _append_label(*cluster_proto*, *key*, *val*)

Append labels to provided Cluster Protobuf

Labels must fit the regex `[a-z]([-a-z0-9]*[a-z0-9])?` (current airflow version string follows semantic versioning spec: x.y.z).

Parameters

- **cluster_proto** (*google.cloud.container_v1.types.Cluster*) – The proto to append resource_label airflow version to
- **key** (*str*) – The key label
- **val** (*str*) –

Returns The cluster proto updated with new label

delete_cluster(*self*, *name*, *project_id=None*, *retry=DEFAULT*, *timeout=DEFAULT*)

Deletes the cluster, including the Kubernetes endpoint and all worker nodes. Firewalls and routes that were configured during cluster creation are also deleted. Other Google Compute Engine resources that might be in use by the cluster (e.g. load balancer resources) will not be deleted if they weren't present at the initial create time.

Parameters

- **name** (*str*) – The name of the cluster to delete
- **project_id** (*str*) – Google Cloud Platform project ID
- **retry** (*google.api_core.retry.Retry*) – Retry object used to determine when/if to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns The full url to the delete operation if successful, else None

create_cluster(*self*, *cluster*, *project_id=None*, *retry=DEFAULT*, *timeout=DEFAULT*)
Creates a cluster, consisting of the specified number and type of Google Compute Engine instances.

Parameters

- **cluster** (*dict or google.cloud.container_v1.types.Cluster*) – A Cluster protobuf or dict. If dict is provided, it must be of the same form as the protobuf message `google.cloud.container_v1.types.Cluster`
- **project_id** (*str*) – Google Cloud Platform project ID
- **retry** (*google.api_core.retry.Retry*) – A retry object (`google.api_core.retry.Retry`) used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns The full url to the new, or existing, cluster

Raises ParseError: On JSON parsing problems when trying to convert dict AirflowException: cluster is not dict type nor Cluster proto type

get_cluster(*self*, *name*, *project_id=None*, *retry=DEFAULT*, *timeout=DEFAULT*)
Gets details of specified cluster

Parameters

- **name** (*str*) – The name of the cluster to retrieve
- **project_id** (*str*) – Google Cloud Platform project ID
- **retry** (*google.api_core.retry.Retry*) – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns google.cloud.container_v1.types.Cluster

airflow.contrib.hooks.gcp_dataflow_hook

Module Contents

`airflow.contrib.hooks.gcp_dataflow_hook.`**DEFAULT_DATAFLOW_LOCATION = us-central1**

class `airflow.contrib.hooks.gcp_dataflow_hook.`**_DataflowJob**(*dataflow*, *project_number*, *name*, *location*, *poll_sleep=10*, *job_id=None*, *num_retries=None*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

_get_job_id_from_name (*self*)

_get_job (*self*)

wait_for_done (*self*)

get (*self*)

class airflow.contrib.hooks.gcp_dataflow_hook.**_Dataflow** (*cmd*)
Bases: airflow.utils.log.logging_mixin.LoggingMixin

_line (*self*, *fd*)

static _extract_job (*line*)

wait_for_done (*self*)

class airflow.contrib.hooks.gcp_dataflow_hook.**DataFlowHook** (*gcp_conn_id='google_cloud_default'*, *delegate_to=None*, *poll_sleep=10*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

get_conn (*self*)
Returns a Google Cloud Dataflow service object.

_start_dataflow (*self*, *variables*, *name*, *command_prefix*, *label_formatter*)

static _set_variables (*variables*)

start_java_dataflow (*self*, *job_name*, *variables*, *dataflow*, *job_class=None*, *append_job_name=True*)

start_template_dataflow (*self*, *job_name*, *variables*, *parameters*, *dataflow_template*, *append_job_name=True*)

start_python_dataflow (*self*, *job_name*, *variables*, *dataflow*, *py_options*, *append_job_name=True*)

static _build_dataflow_job_name (*job_name*, *append_job_name=True*)

static _build_cmd (*variables*, *label_formatter*)

_start_template_dataflow (*self*, *name*, *variables*, *parameters*, *dataflow_template*)

airflow.contrib.hooks.gcp_dataproc_hook

Module Contents

class airflow.contrib.hooks.gcp_dataproc_hook.**_DataProcJob** (*dataproc_api*, *project_id*, *job*, *region='global'*, *job_error_states=None*, *num_retries=None*)
Bases: airflow.utils.log.logging_mixin.LoggingMixin

wait_for_done (*self*)

raise_error (*self*, *message=None*)

get (*self*)

class airflow.contrib.hooks.gcp_dataproc_hook.**_DataProcJobBuilder** (*project_id*, *task_id*, *cluster_name*, *job_type*, *properties*)

add_variables (*self*, *variables*)

add_args (*self*, *args*)

add_query (*self*, *query*)

add_query_uri (*self*, *query_uri*)

add_jar_file_uris (*self*, *jars*)

add_archive_uris (*self*, *archives*)

add_file_uris (*self*, *files*)

add_python_file_uris (*self*, *pyfiles*)

set_main (*self*, *main_jar*, *main_class*)

set_python_main (*self*, *main*)

set_job_name (*self*, *name*)

build (*self*)

class airflow.contrib.hooks.gcp_dataproc_hook.**_DataProcOperation** (*dataproc_api*, *operation*, *num_retries*)

Bases: airflow.utils.log.logging_mixin.LoggingMixin

Continuously polls Dataproc Operation until it completes.

wait_for_done (*self*)

get (*self*)

_check_done (*self*)

_raise_error (*self*)

class airflow.contrib.hooks.gcp_dataproc_hook.**DataProcHook** (*gcp_conn_id='google_cloud_default'*, *delegate_to=None*, *api_version='v1beta2'*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud Dataproc APIs.

get_conn (*self*)

Returns a Google Cloud Dataproc service object.

get_cluster (*self*, *project_id*, *region*, *cluster_name*)

submit (*self*, *project_id*, *job*, *region='global'*, *job_error_states=None*)

create_job_template (*self*, *task_id*, *cluster_name*, *job_type*, *properties*)

wait (*self*, *operation*)

Awaits for Google Cloud Dataproc Operation to complete.

cancel (*self*, *project_id*, *job_id*, *region='global'*)

Cancel a Google Cloud DataProc job. :param project_id: Name of the project the job belongs to :type project_id: str :param job_id: Identifier of the job to cancel :type job_id: int :param region: Region used for the job :type region: str :returns A Job json dictionary representing the canceled job

airflow.contrib.hooks.gcp_function_hook

Module Contents

airflow.contrib.hooks.gcp_function_hook.**TIME_TO_SLEEP_IN_SECONDS = 1**

class airflow.contrib.hooks.gcp_function_hook.**GcfHook**(*api_version*, *gcp_conn_id='google_cloud_default'*, *delegate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for the Google Cloud Functions APIs.

All the methods in the hook where project_id is used must be called with keyword arguments rather than positional.

_conn

static _full_location(*project_id*, *location*)

Retrieve full location of the function in the form of projects/<GCP_PROJECT_ID>/locations/<GCP_LOCATION>

Parameters

- **project_id** (*str*) – The Google Cloud Project project_id where the function belongs.
- **location** (*str*) – The location where the function is created.

Returns

get_conn(*self*)

Retrieves the connection to Cloud Functions.

Returns Google Cloud Function services object.

Return type dict

get_function(*self*, *name*)

Returns the Cloud Function with the given name.

Parameters **name** (*str*) – Name of the function.

Returns A Cloud Functions object representing the function.

Return type dict

create_new_function(*self*, *location*, *body*, *project_id=None*)

Creates a new function in Cloud Function in the location specified in the body.

Parameters

- **location** (*str*) – The location of the function.
- **body** (*dict*) – The body required by the Cloud Functions insert API.
- **project_id** (*str*) – Optional, Google Cloud Project project_id where the function belongs. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

update_function(*self*, *name*, *body*, *update_mask*)

Updates Cloud Functions according to the specified update mask.

Parameters

- **name** (*str*) – The name of the function.
- **body** (*dict*) – The body required by the cloud function patch API.
- **update_mask** (*[str]*) – The update mask - array of fields that should be patched.

Returns None

upload_function_zip(*self*, *location*, *zip_path*, *project_id=None*)

Uploads zip file with sources.

Parameters

- **location** (*str*) – The location where the function is created.

- **zip_path** (*str*) – The path of the valid .zip file to upload.
- **project_id** (*str*) – Optional, Google Cloud Project project_id where the function belongs. If set to None or missing, the default project_id from the GCP connection is used.

Returns The upload URL that was returned by generateUploadUrl method.

delete_function (*self*, *name*)

Deletes the specified Cloud Function.

Parameters **name** (*str*) – The name of the function.

Returns None

_wait_for_operation_to_complete (*self*, *operation_name*)

Waits for the named operation to complete - checks status of the asynchronous call.

Parameters **operation_name** (*str*) – The name of the operation.

Returns The response returned by the operation.

Return type dict

Exception AirflowException in case error is returned.

airflow.contrib.hooks.gcp_kms_hook

Module Contents

airflow.contrib.hooks.gcp_kms_hook.**_b64encode** (*s*)

Base 64 encodes a bytes object to a string

airflow.contrib.hooks.gcp_kms_hook.**_b64decode** (*s*)

Base 64 decodes a string to bytes.

class airflow.contrib.hooks.gcp_kms_hook.**GoogleCloudKMSHook** (*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Interact with Google Cloud KMS. This hook uses the Google Cloud Platform connection.

get_conn (*self*)

Returns a KMS service object.

Return type googleapiclient.discovery.Resource

encrypt (*self*, *key_name*, *plaintext*, *authenticated_data=None*)

Encrypts a plaintext message using Google Cloud KMS.

Parameters

- **key_name** (*str*) – The Resource Name for the key (or key version) to be used for encyption. Of the form `projects/*/locations/*/keyRings/*/cryptoKeys/**`
- **plaintext** (*bytes*) – The message to be encrypted.
- **authenticated_data** (*bytes*) – Optional additional authenticated data that must also be provided to decrypt the message.

Returns The base 64 encoded ciphertext of the original message.

Return type str

decrypt (*self*, *key_name*, *ciphertext*, *authenticated_data=None*)

Decrypts a ciphertext message using Google Cloud KMS.

Parameters

- **key_name** (*str*) – The Resource Name for the key to be used for decyption. Of the form `projects/*/locations/*/keyRings/*/cryptoKeys/**`
- **ciphertext** (*str*) – The message to be decrypted.
- **authenticated_data** (*bytes*) – Any additional authenticated data that was provided when encrypting the message.

Returns The original message.

Return type bytes

airflow.contrib.hooks.gcp_mlengine_hook

Module Contents

`airflow.contrib.hooks.gcp_mlengine_hook.`**`_poll_with_exponential_delay`**(*request*, *max_n*, *is_done_func*, *is_error_func*)

class `airflow.contrib.hooks.gcp_mlengine_hook.`**`MLEngineHook`**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)

Bases: `airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook`

get_conn(*self*)

Returns a Google MLEngine service object.

create_job(*self*, *project_id*, *job*, *use_existing_job_fn=None*)

Launches a MLEngine job and wait for it to reach a terminal state.

Parameters

- **project_id** (*str*) – The Google Cloud project id within which MLEngine job will be launched.
- **job** (*dict*) – MLEngine Job object that should be provided to the MLEngine API, such as:

```
{
  'jobId': 'my_job_id',
  'trainingInput': {
    'scaleTier': 'STANDARD_1',
    ...
  }
}
```

- **use_existing_job_fn** (*function*) – In case that a MLEngine job with the same job_id already exist, this method (if provided) will decide whether we should use this existing job, continue waiting for it to finish and returning the job object. It should accepts a MLEngine job object, and returns a boolean value indicating whether it is OK to reuse the existing job. If 'use_existing_job_fn' is not provided, we by default reuse the existing MLEngine job.

Returns The MLEngine job object if the job successfully reach a terminal state (which might be FAILED or CANCELLED state).

Return type dict

_get_job(*self*, *project_id*, *job_id*)

Gets a MLEngine job based on the job name.

Returns MLEngine job object if succeed.

Return type dict

Raises: googleapiclient.errors.HttpError: if HTTP error is returned from server

`_wait_for_job_done`(*self*, *project_id*, *job_id*, *interval=30*)
Waits for the Job to reach a terminal state.

This method will periodically check the job state until the job reach a terminal state.

Raises: googleapiclient.errors.HttpError: if HTTP error is returned when getting the job

`create_version`(*self*, *project_id*, *model_name*, *version_spec*)
Creates the Version on Google Cloud ML Engine.

Returns the operation if the version was created successfully and raises an error otherwise.

`set_default_version`(*self*, *project_id*, *model_name*, *version_name*)
Sets a version to be the default. Blocks until finished.

`list_versions`(*self*, *project_id*, *model_name*)
Lists all available versions of a model. Blocks until finished.

`delete_version`(*self*, *project_id*, *model_name*, *version_name*)
Deletes the given version of a model. Blocks until finished.

`create_model`(*self*, *project_id*, *model*)
Create a Model. Blocks until finished.

`get_model`(*self*, *project_id*, *model_name*)
Gets a Model. Blocks until finished.

`airflow.contrib.hooks.gcp_natural_language_hook`

Module Contents

class `airflow.contrib.hooks.gcp_natural_language_hook.`**`CloudNaturalLanguageHook`**(*gcp_conn_id='go*
del-
e-
gate_to=None)

Bases: *`airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook`*

Hook for Google Cloud Natural Language Service.

Parameters

- **`gcp_conn_id`** (*`str`*) – The connection ID to use when fetching connection info.
- **`delegate_to`** (*`str`*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

`_conn`

`get_conn`(*self*)
Retrieves connection to Cloud Natural Language service.

Returns Cloud Natural Language service object

Return type google.cloud.language_v1.LanguageServiceClient

`analyze_entities`(*self*, *document*, *encoding_type=None*, *retry=None*, *timeout=None*, *metadata=None*)
Finds named entities in the text along with entity types, salience, mentions for each entity, and other properties.

Parameters

- **document** (*dict or class google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** (*google.api_core.retry.Retry*) – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]]*) – Additional metadata that is provided to the method.

Return type google.cloud.language_v1.types.AnalyzeEntitiesResponse

analyze_entity_sentiment(*self*, *document*, *encoding_type=None*, *retry=None*, *timeout=None*, *metadata=None*)

Finds entities, similar to AnalyzeEntities in the text and analyzes sentiment associated with each entity and its mentions.

Parameters

- **document** (*dict or class google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** (*google.api_core.retry.Retry*) – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]]*) – Additional metadata that is provided to the method.

Return type google.cloud.language_v1.types.AnalyzeEntitiesResponse

analyze_sentiment(*self*, *document*, *encoding_type=None*, *retry=None*, *timeout=None*, *metadata=None*)

Analyzes the sentiment of the provided text.

Parameters

- **document** (*dict or class google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** (*google.api_core.retry.Retry*) – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]]*) – Additional metadata that is provided to the method.

Return type google.cloud.language_v1.types.AnalyzeEntitiesResponse

analyze_syntax(*self*, *document*, *encoding_type=None*, *retry=None*, *timeout=None*, *metadata=None*)
Analyzes the syntax of the text and provides sentence boundaries and tokenization along with part of speech tags, dependency trees, and other properties.

Parameters

- **document** (*dict or class google.cloud.language_v1.types.Document#*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** (*google.api_core.retry.Retry*) – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]]*) – Additional metadata that is provided to the method.

Return type google.cloud.language_v1.types.AnalyzeSyntaxResponse

annotate_text(*self*, *document*, *features*, *encoding_type=None*, *retry=None*, *timeout=None*, *metadata=None*)
A convenience method that provides all the features that analyzeSentiment, analyzeEntities, and analyzeSyntax provide in one call.

Parameters

- **document** (*dict or google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **features** (*dict or google.cloud.language_v1.enums.Features*) – The enabled features. If a dict is provided, it must be of the same form as the protobuf message Features
- **encoding_type** (*google.cloud.language_v1.types.EncodingType*) – The encoding type used by the API to calculate offsets.
- **retry** (*google.api_core.retry.Retry*) – A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]]*) – Additional metadata that is provided to the method.

Return type google.cloud.language_v1.types.AnnotateTextResponse

classify_text(*self*, *document*, *retry=None*, *timeout=None*, *metadata=None*)
Classifies a document into categories.

Parameters

- **document** (*dict or class google.cloud.language_v1.types.Document*) – Input document. If a dict is provided, it must be of the same form as the protobuf message Document
- **retry** (*google.api_core.retry.Retry*) – A retry object used to retry requests. If None is specified, requests will not be retried.

- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*sequence[tuple[str, str]]]*) – Additional metadata that is provided to the method.

Return type google.cloud.language_v1.types.AnalyzeEntitiesResponse

airflow.contrib.hooks.gcp_pubsub_hook

Module Contents

airflow.contrib.hooks.gcp_pubsub_hook.**_format_subscription**(*project*, *subscription*)

airflow.contrib.hooks.gcp_pubsub_hook.**_format_topic**(*project*, *topic*)

exception airflow.contrib.hooks.gcp_pubsub_hook.**PubSubException**
Bases: Exception

class airflow.contrib.hooks.gcp_pubsub_hook.**PubSubHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for accessing Google Pub/Sub.

The GCP project against which actions are applied is determined by the project embedded in the Connection referenced by gcp_conn_id.

get_conn(*self*)
Returns a Pub/Sub service object.

Return type googleapiclient.discovery.Resource

publish(*self*, *project*, *topic*, *messages*)
Publishes messages to a Pub/Sub topic.

Parameters

- **project** (*str*) – the GCP project ID in which to publish
- **topic** (*str*) – the Pub/Sub topic to which to publish; do not include the `projects/{project}/topics/` prefix.
- **messages** (list of PubSub messages; see http://cloud.google.com/pubsub/docs/reference/rest/v1/PubsubMessage) – messages to publish; if the data field in a message is set, it should already be base64 encoded.

create_topic(*self*, *project*, *topic*, *fail_if_exists=False*)
Creates a Pub/Sub topic, if it does not already exist.

Parameters

- **project** (*str*) – the GCP project ID in which to create the topic
- **topic** (*str*) – the Pub/Sub topic name to create; do not include the `projects/{project}/topics/` prefix.
- **fail_if_exists** (*bool*) – if set, raise an exception if the topic already exists

delete_topic(*self*, *project*, *topic*, *fail_if_not_exists=False*)
Deletes a Pub/Sub topic if it exists.

Parameters

- **project** (*str*) – the GCP project ID in which to delete the topic

- **topic** (*str*) – the Pub/Sub topic name to delete; do not include the `projects/{project}/topics/` prefix.
- **fail_if_not_exists** (*bool*) – if set, raise an exception if the topic does not exist

create_subscription(*self*, *topic_project*, *topic*, *subscription=None*, *subscription_project=None*, *ack_deadline_secs=10*, *fail_if_exists=False*)

Creates a Pub/Sub subscription, if it does not already exist.

Parameters

- **topic_project** (*str*) – the GCP project ID of the topic that the subscription will be bound to.
- **topic** (*str*) – the Pub/Sub topic name that the subscription will be bound to create; do not include the `projects/{project}/subscriptions/` prefix.
- **subscription** (*str*) – the Pub/Sub subscription name. If empty, a random name will be generated using the uuid module
- **subscription_project** (*str*) – the GCP project ID where the subscription will be created. If unspecified, `topic_project` will be used.
- **ack_deadline_secs** (*int*) – Number of seconds that a subscriber has to acknowledge each message pulled from the subscription
- **fail_if_exists** (*bool*) – if set, raise an exception if the topic already exists

Returns subscription name which will be the system-generated value if the `subscription` parameter is not supplied

Return type str

delete_subscription(*self*, *project*, *subscription*, *fail_if_not_exists=False*)

Deletes a Pub/Sub subscription, if it exists.

Parameters

- **project** (*str*) – the GCP project ID where the subscription exists
- **subscription** (*str*) – the Pub/Sub subscription name to delete; do not include the `projects/{project}/subscriptions/` prefix.
- **fail_if_not_exists** (*bool*) – if set, raise an exception if the topic does not exist

pull(*self*, *project*, *subscription*, *max_messages*, *return_immediately=False*)

Pulls up to `max_messages` messages from Pub/Sub subscription.

Parameters

- **project** (*str*) – the GCP project ID where the subscription exists
- **subscription** (*str*) – the Pub/Sub subscription name to pull from; do not include the 'projects/{project}/topics/' prefix.
- **max_messages** (*int*) – The maximum number of messages to return from the Pub/Sub API.
- **return_immediately** (*bool*) – If set, the Pub/Sub API will immediately return if no messages are available. Otherwise, the request will block for an undisclosed, but bounded period of time

Returns A list of Pub/Sub ReceivedMessage objects each containing an `ackId` property and a `message` property, which includes the base64-encoded message content. See https://cloud.google.com/pubsub/docs/reference/rest/v1/projects.subscriptions/pull#ReceivedMessage

acknowledge(*self*, *project*, *subscription*, *ack_ids*)

Pulls up to `max_messages` messages from Pub/Sub subscription.

Parameters

- **project** (*str*) – the GCP project name or ID in which to create the topic
- **subscription** (*str*) – the Pub/Sub subscription name to delete; do not include the 'projects/{project}/topics/' prefix.
- **ack_ids** (*list*) – List of ReceivedMessage ackIds from a previous pull response

airflow.contrib.hooks.gcp_spanner_hook

Module Contents

class airflow.contrib.hooks.gcp_spanner_hook.**CloudSpannerHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud Spanner APIs.

All the methods in the hook where project_id is used must be called with keyword arguments rather than positional.

_client

_get_client(*self*, *project_id*)

Provides a client for interacting with the Cloud Spanner API.

Parameters **project_id** (*str*) – The ID of the GCP project.

Returns google.cloud.spanner_v1.client.Client

Return type object

get_instance(*self*, *instance_id*, *project_id=None*)

Gets information about a particular instance.

Parameters

- **project_id** (*str*) – Optional, The ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.
- **instance_id** (*str*) – The ID of the Cloud Spanner instance.

Returns google.cloud.spanner_v1.instance.Instance

Return type object

_apply_to_instance(*self*, *project_id*, *instance_id*, *configuration_name*, *node_count*, *display_name*, *func*)

Invokes a method on a given instance by applying a specified Callable.

Parameters

- **project_id** (*str*) – The ID of the GCP project that owns the Cloud Spanner database.
- **instance_id** (*str*) – The ID of the instance.
- **configuration_name** (*str*) – Name of the instance configuration defining how the instance will be created. Required for instances which do not yet exist.
- **node_count** (*int*) – (Optional) Number of nodes allocated to the instance.
- **display_name** (*str*) – (Optional) The display name for the instance in the Cloud Console UI. (Must be between 4 and 30 characters.) If this value is not set in the constructor, will fall back to the instance ID.

- **func** (*Callable*) – Method of the instance to be called.

create_instance (*self*, *instance_id*, *configuration_name*, *node_count*, *display_name*, *project_id=None*)
Creates a new Cloud Spanner instance.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.
- **configuration_name** (*str*) – The name of the instance configuration defining how the instance will be created. Possible configuration values can be retrieved via https://cloud.google.com/spanner/docs/reference/rest/v1/projects.instanceConfigs/list
- **node_count** (*int*) – (Optional) The number of nodes allocated to the Cloud Spanner instance.
- **display_name** (*str*) – (Optional) The display name for the instance in the GCP Console. Must be between 4 and 30 characters. If this value is not set in the constructor, the name falls back to the instance ID.
- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

update_instance (*self*, *instance_id*, *configuration_name*, *node_count*, *display_name*, *project_id=None*)
Updates an existing Cloud Spanner instance.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.
- **configuration_name** (*str*) – The name of the instance configuration defining how the instance will be created. Possible configuration values can be retrieved via https://cloud.google.com/spanner/docs/reference/rest/v1/projects.instanceConfigs/list
- **node_count** (*int*) – (Optional) The number of nodes allocated to the Cloud Spanner instance.
- **display_name** (*str*) – (Optional) The display name for the instance in the GCP Console. Must be between 4 and 30 characters. If this value is not set in the constructor, the name falls back to the instance ID.
- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

delete_instance (*self*, *instance_id*, *project_id=None*)
Deletes an existing Cloud Spanner instance.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.
- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

get_database (*self*, *instance_id*, *database_id*, *project_id=None*)
Retrieves a database in Cloud Spanner. If the database does not exist in the specified instance, it returns None.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.

- **database_id** (*str*) – The ID of the database in Cloud Spanner.
- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.

Returns Database object or None if database does not exist

Return type google.cloud.spanner_v1.database.Database or None

create_database (*self*, *instance_id*, *database_id*, *ddl_statements*, *project_id=None*)
Creates a new database in Cloud Spanner.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.
- **database_id** (*str*) – The ID of the database to create in Cloud Spanner.
- **ddl_statements** (*list[str]*) – The string list containing DDL for the new database.
- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

update_database (*self*, *instance_id*, *database_id*, *ddl_statements*, *project_id=None*, *operation_id=None*)
Updates DDL of a database in Cloud Spanner.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.
- **database_id** (*str*) – The ID of the database in Cloud Spanner.
- **ddl_statements** (*list[str]*) – The string list containing DDL for the new database.
- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.
- **operation_id** (*str*) – (Optional) The unique per database operation ID that can be specified to implement idempotency check.

Returns None

delete_database (*self*, *instance_id*, *database_id*, *project_id=None*)
Drops a database in Cloud Spanner.

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.
- **database_id** (*str*) – The ID of the database in Cloud Spanner.
- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.

Returns True if everything succeeded

Return type bool

execute_dml (*self*, *instance_id*, *database_id*, *queries*, *project_id=None*)
Executes an arbitrary DML query (INSERT, UPDATE, DELETE).

Parameters

- **instance_id** (*str*) – The ID of the Cloud Spanner instance.
- **database_id** (*str*) – The ID of the database in Cloud Spanner.
- **queries** (*str*) – The queries to execute.

- **project_id** (*str*) – Optional, the ID of the GCP project that owns the Cloud Spanner database. If set to None or missing, the default project_id from the GCP connection is used.

static _execute_sql_in_transaction (*transaction*, *queries*)

airflow.contrib.hooks.gcp_speech_to_text_hook

Module Contents

class airflow.contrib.hooks.gcp_speech_to_text_hook.**GCPSpeechToTextHook** (*gcp_conn_id='google_cloud_ del- e- gate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud Speech API.

Parameters

- **gcp_conn_id** (*str*) – The connection ID to use when fetching connection info.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

_client

get_conn (*self*)

Retrieves connection to Cloud Speech.

Returns Google Cloud Speech client object.

Return type google.cloud.speech_v1.SpeechClient

recognize_speech (*self*, *config*, *audio*, *retry=None*, *timeout=None*)

Recognizes audio input

Parameters

- **config** (*dict or google.cloud.speech_v1.types.RecognitionConfig*) – information to the recognizer that specifies how to process the request. https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/types.html#google.cloud.speech_v1.types.RecognitionConfig
- **audio** (*dict or google.cloud.speech_v1.types.RecognitionAudio*) – audio data to be recognized https://googleapis.github.io/google-cloud-python/latest/speech/gapic/v1/types.html#google.cloud.speech_v1.types.RecognitionAudio
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

airflow.contrib.hooks.gcp_sql_hook

Module Contents

airflow.contrib.hooks.gcp_sql_hook.**UNIX_PATH_MAX = 108**

airflow.contrib.hooks.gcp_sql_hook.**TIME_TO_SLEEP_IN_SECONDS = 1**

class airflow.contrib.hooks.gcp_sql_hook.**CloudSqlOperationStatus**

PENDING = PENDING

RUNNING = RUNNING

DONE = DONE

UNKNOWN = UNKNOWN

class airflow.contrib.hooks.gcp_sql_hook.**CloudSqlHook**(*api_version*, *gcp_conn_id='google_cloud_default'*, *delegate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud SQL APIs.

All the methods in the hook where project_id is used must be called with keyword arguments rather than positional.

_conn

get_conn(*self*)

Retrieves connection to Cloud SQL.

Returns Google Cloud SQL services object.

Return type dict

get_instance(*self*, *instance*, *project_id=None*)

Retrieves a resource containing information about a Cloud SQL instance.

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns A Cloud SQL instance resource.

Return type dict

create_instance(*self*, *body*, *project_id=None*)

Creates a new Cloud SQL instance.

Parameters

- **body** (*dict*) – Body required by the Cloud SQL insert API, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/insert#request-body.
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

patch_instance(*self*, *body*, *instance*, *project_id=None*)

Updates settings of a Cloud SQL instance.

Caution: This is not a partial update, so you must include values for all the settings that you want to retain.

Parameters

- **body** (*dict*) – Body required by the Cloud SQL patch API, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/patch#request-body.
- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.

- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

delete_instance (*self*, *instance*, *project_id=None*)
Deletes a Cloud SQL instance.

Parameters

- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.
- **instance** (*str*) – Cloud SQL instance ID. This does not include the project ID.

Returns None

get_database (*self*, *instance*, *database*, *project_id=None*)
Retrieves a database resource from a Cloud SQL instance.

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **database** (*str*) – Name of the database in the instance.
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns A Cloud SQL database resource, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/databases#resource.

Return type dict

create_database (*self*, *instance*, *body*, *project_id=None*)
Creates a new database inside a Cloud SQL instance.

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/databases/insert#request-body.
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

patch_database (*self*, *instance*, *database*, *body*, *project_id=None*)
Updates a database resource inside a Cloud SQL instance.

This method supports patch semantics. See https://cloud.google.com/sql/docs/mysql/admin-api/how-tos/performance#patch.

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **database** (*str*) – Name of the database to be updated in the instance.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/databases/insert#request-body.
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

delete_database(*self*, *instance*, *database*, *project_id=None*)
Deletes a database from a Cloud SQL instance.

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **database** (*str*) – Name of the database to be deleted in the instance.
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

export_instance(*self*, *instance*, *body*, *project_id=None*)
Exports data from a Cloud SQL instance to a Cloud Storage bucket as a SQL dump or CSV file.

Parameters

- **instance** (*str*) – Database instance ID of the Cloud SQL instance. This does not include the project ID.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/export#request-body
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

import_instance(*self*, *instance*, *body*, *project_id=None*)
Imports data into a Cloud SQL instance from a SQL dump or CSV file in Cloud Storage.

Parameters

- **instance** (*str*) – Database instance ID. This does not include the project ID.
- **body** (*dict*) – The request body, as described in https://cloud.google.com/sql/docs/mysql/admin-api/v1beta4/instances/export#request-body
- **project_id** (*str*) – Project ID of the project that contains the instance. If set to None or missing, the default project_id from the GCP connection is used.

Returns None

_wait_for_operation_to_complete(*self*, *project_id*, *operation_name*)
Waits for the named operation to complete - checks status of the asynchronous call.

Parameters

- **project_id** (*str*) – Project ID of the project that contains the instance.
- **operation_name** (*str*) – Name of the operation.

Returns None

airflow.contrib.hooks.gcp_sql_hook.**CLOUD_SQL_PROXY_DOWNLOAD_URL = https://dl.google.com/cl**

airflow.contrib.hooks.gcp_sql_hook.**CLOUD_SQL_PROXY_VERSION_DOWNLOAD_URL = https://storage.**

airflow.contrib.hooks.gcp_sql_hook.**GCP_CREDENTIALS_KEY_PATH = extra__google_cloud_platform_**

airflow.contrib.hooks.gcp_sql_hook.**GCP_CREDENTIALS_KEYFILE_DICT = extra__google_cloud_plat**

class airflow.contrib.hooks.gcp_sql_hook.**CloudSqlProxyRunner**(*path_prefix*, *instance_specification*, *gcp_conn_id='google_cloud_default'*, *project_id=None*, *sql_proxy_version=None*, *sql_proxy_binary_path=None*)

Bases: airflow.LoggingMixin

Downloads and runs cloud-sql-proxy as subprocess of the Python process.

The cloud-sql-proxy needs to be downloaded and started before we can connect to the Google Cloud SQL instance via database connection. It establishes secure tunnel connection to the database. It authorizes using the GCP credentials that are passed by the configuration.

More details about the proxy can be found here: https://cloud.google.com/sql/docs/mysql/sql-proxy

_build_command_line_parameters(*self*)

static _is_os_64bit()

_download_sql_proxy_if_needed(*self*)

_get_credential_parameters(*self*, *session*)

start_proxy(*self*)
Starts Cloud SQL Proxy.

You have to remember to stop the proxy if you started it!

stop_proxy(*self*)
Stops running proxy.

You should stop the proxy after you stop using it.

get_proxy_version(*self*)
Returns version of the Cloud SQL Proxy.

get_socket_path(*self*)
Retrieves UNIX socket path used by Cloud SQL Proxy.

Returns The dynamically generated path for the socket created by the proxy.

Return type str

airflow.contrib.hooks.gcp_sql_hook.**CONNECTION_URIS**

airflow.contrib.hooks.gcp_sql_hook.**CLOUD_SQL_VALID_DATABASE_TYPES = ['postgres', 'mysql']**

class airflow.contrib.hooks.gcp_sql_hook.**CloudSqlDatabaseHook**(*gcp_cloudsql_conn_id='google_cloud_sql_d* *default_gcp_project_id=None*)

Bases: *airflow.hooks.base_hook.BaseHook*

Serves DB connection configuration for Google Cloud SQL (Connections of *gcpcloudsql://* type).

The hook is a "meta" one. It does not perform an actual connection. It is there to retrieve all the parameters configured in gcpcloudsql:// connection, start/stop Cloud SQL Proxy if needed, dynamically generate Postgres or MySQL connection in the database and return an actual Postgres or MySQL hook. The returned Postgres/MySQL hooks are using direct connection or Cloud SQL Proxy socket/TCP as configured.

Main parameters of the hook are retrieved from the standard URI components:

- **user** - User name to authenticate to the database (from login of the URI).
- **password** - Password to authenticate to the database (from password of the URI).
- **public_ip** - IP to connect to for public connection (from host of the URI).
- **public_port** - Port to connect to for public connection (from port of the URI).

- **database** - Database to connect to (from schema of the URI).

Remaining parameters are retrieved from the extras (URI query parameters):

- **project_id - Optional, Google Cloud Platform project where the Cloud SQL** instance exists. If missing, default project id passed is used.
- **instance** - Name of the instance of the Cloud SQL database instance.
- **location** - The location of the Cloud SQL instance (for example europe-west1).
- **database_type** - The type of the database instance (MySQL or Postgres).
- **use_proxy** - (default False) Whether SQL proxy should be used to connect to Cloud SQL DB.
- **use_ssl** - (default False) Whether SSL should be used to connect to Cloud SQL DB. You cannot use proxy and SSL together.
- **sql_proxy_use_tcp** - (default False) If set to true, TCP is used to connect via proxy, otherwise UNIX sockets are used.
- **sql_proxy_binary_path** - Optional path to Cloud SQL Proxy binary. If the binary is not specified or the binary is not present, it is automatically downloaded.
- **sql_proxy_version** - Specific version of the proxy to download (for example v1.13). If not specified, the latest version is downloaded.
- **sslcert** - Path to client certificate to authenticate when SSL is used.
- **sslkey** - Path to client private key to authenticate when SSL is used.
- **sslrootcert** - Path to server's certificate to authenticate when SSL is used.

Parameters

- **`gcp_cloudsql_conn_id`** (`str`) – URL of the connection
- **`default_gcp_project_id`** (`str`) – Default project id used if project_id not specified in the connection URL

`_conn`

`static _get_bool`(*val*)

`static _check_ssl_file`(*file_to_check*, *name*)

`_validate_inputs`(*self*)

`validate_ssl_certs`(*self*)

`validate_socket_path_length`(*self*)

`static _generate_unique_path`()

`static _quote`(*value*)

`_generate_connection_uri`(*self*)

`_get_instance_socket_name`(*self*)

`_get_sqlproxy_instance_specification`(*self*)

`create_connection`(*self*, *session=None*)

Create connection in the Connection table, according to whether it uses proxy, TCP, UNIX sockets, SSL. Connection ID will be randomly generated.

Parameters **`session`** – Session of the SQL Alchemy ORM (automatically generated with decorator).

`retrieve_connection`(*self*, *session=None*)

Retrieves the dynamically created connection from the Connection table.

Parameters **session** – Session of the SQL Alchemy ORM (automatically generated with decorator).

delete_connection (*self*, *session=None*)
Delete the dynamically created connection from the Connection table.

Parameters **session** – Session of the SQL Alchemy ORM (automatically generated with decorator).

get_sqlproxy_runner (*self*)
Retrieve Cloud SQL Proxy runner. It is used to manage the proxy lifecycle per task.

Returns The Cloud SQL Proxy runner.

Return type *CloudSqlProxyRunner*

get_database_hook (*self*)
Retrieve database hook. This is the actual Postgres or MySQL database hook that uses proxy or connects directly to the Google Cloud SQL database.

cleanup_database_hook (*self*)
Clean up database hook after it was used.

reserve_free_tcp_port (*self*)
Reserve free TCP port to be used by Cloud SQL Proxy

free_reserved_port (*self*)
Free TCP port. Makes it immediately ready to be used by Cloud SQL Proxy.

airflow.contrib.hooks.gcp_text_to_speech_hook

Module Contents

class airflow.contrib.hooks.gcp_text_to_speech_hook.**GCPTextToSpeechHook** (*gcp_conn_id='google_cloud_ del- e- gate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud Text to Speech API.

Parameters

- **gcp_conn_id** (*str*) – The connection ID to use when fetching connection info.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

_client

get_conn (*self*)
Retrieves connection to Cloud Text to Speech.

Returns Google Cloud Text to Speech client object.

Return type google.cloud.texttospeech_v1.TextToSpeechClient

synthesize_speech (*self*, *input_data*, *voice*, *audio_config*, *retry=None*, *timeout=None*)
Synthesizes text input

Parameters

- **input_data** (*dict or google.cloud.texttospeech_v1.types.SynthesisInput*) – text input to be synthesized. See more: https://googleapis.

github.io/google-cloud-python/latest/texttospeech/gapic/v1/types.html#google.cloud.texttospeech_v1.types.SynthesisInput

- **voice** (*dict or google.cloud.texttospeech_v1.types.VoiceSelectionParams*) – configuration of voice to be used in synthesis. See more: https://googleapis.github.io/google-cloud-python/latest/texttospeech/gapic/v1/types.html#google.cloud.texttospeech_v1.types.VoiceSelectionParams
- **audio_config** (*dict or google.cloud.texttospeech_v1.types.AudioConfig*) – configuration of the synthesized audio. See more: https://googleapis.github.io/google-cloud-python/latest/texttospeech/gapic/v1/types.html#google.cloud.texttospeech_v1.types.AudioConfig
- **retry** (*google.api_core.retry.Retry*) – (Optional) A retry object used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – (Optional) The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns SynthesizeSpeechResponse See more: https://googleapis.github.io/google-cloud-python/latest/texttospeech/gapic/v1/types.html#google.cloud.texttospeech_v1.types.SynthesizeSpeechResponse

Return type object

airflow.contrib.hooks.gcp_transfer_hook

Module Contents

airflow.contrib.hooks.gcp_transfer_hook.**TIME_TO_SLEEP_IN_SECONDS = 10**

class airflow.contrib.hooks.gcp_transfer_hook.**GcpTransferJobsStatus**

ENABLED = ENABLED

DISABLED = DISABLED

DELETED = DELETED

class airflow.contrib.hooks.gcp_transfer_hook.**GcpTransferOperationStatus**

IN_PROGRESS = IN_PROGRESS

PAUSED = PAUSED

SUCCESS = SUCCESS

FAILED = FAILED

ABORTED = ABORTED

airflow.contrib.hooks.gcp_transfer_hook.**ACCESS_KEY_ID = accessKeyId**

airflow.contrib.hooks.gcp_transfer_hook.**ALREADY_EXISTING_IN_SINK = overwriteObjectsAlreadyE**

airflow.contrib.hooks.gcp_transfer_hook.**AWS_ACCESS_KEY = awsAccessKey**

airflow.contrib.hooks.gcp_transfer_hook.**AWS_S3_DATA_SOURCE = awsS3DataSource**

airflow.contrib.hooks.gcp_transfer_hook.**BODY = body**

airflow.contrib.hooks.gcp_transfer_hook.**BUCKET_NAME = bucketName**

airflow.contrib.hooks.gcp_transfer_hook.**DAY = day** airflow.contrib.hooks.gcp_transfer_hook.**DESCRIPTION = description**

airflow.contrib.hooks.gcp_transfer_hook.**FILTER = filter**

airflow.contrib.hooks.gcp_transfer_hook.**FILTER_JOB_NAMES = job_names**

airflow.contrib.hooks.gcp_transfer_hook.**FILTER_PROJECT_ID = project_id**

airflow.contrib.hooks.gcp_transfer_hook.**GCS_DATA_SINK = gcsDataSink**

airflow.contrib.hooks.gcp_transfer_hook.**GCS_DATA_SOURCE = gcsDataSource**

airflow.contrib.hooks.gcp_transfer_hook.**HOURS = hours**

airflow.contrib.hooks.gcp_transfer_hook.**HTTP_DATA_SOURCE = httpDataSource**

airflow.contrib.hooks.gcp_transfer_hook.**LIST_URL = list_url**

airflow.contrib.hooks.gcp_transfer_hook.**METADATA = metadata**

airflow.contrib.hooks.gcp_transfer_hook.**MINUTES = minutes**

airflow.contrib.hooks.gcp_transfer_hook.**MONTH = month**

airflow.contrib.hooks.gcp_transfer_hook.**NAME = name**

airflow.contrib.hooks.gcp_transfer_hook.**OBJECT_CONDITIONS = object_conditions**

airflow.contrib.hooks.gcp_transfer_hook.**OPERATIONS = operations**

airflow.contrib.hooks.gcp_transfer_hook.**PROJECT_ID = projectId**

airflow.contrib.hooks.gcp_transfer_hook.**SCHEDULE = schedule**

airflow.contrib.hooks.gcp_transfer_hook.**SCHEDULE_END_DATE = scheduleEndDate**

airflow.contrib.hooks.gcp_transfer_hook.**SCHEDULE_START_DATE = scheduleStartDate**

airflow.contrib.hooks.gcp_transfer_hook.**SECONDS = seconds**

airflow.contrib.hooks.gcp_transfer_hook.**SECRET_ACCESS_KEY = secretAccessKey**

airflow.contrib.hooks.gcp_transfer_hook.**START_TIME_OF_DAY = startTimeOfDay**

airflow.contrib.hooks.gcp_transfer_hook.**STATUS = status**

airflow.contrib.hooks.gcp_transfer_hook.**STATUS1 = status**

airflow.contrib.hooks.gcp_transfer_hook.**TRANSFER_JOB = transfer_job**

airflow.contrib.hooks.gcp_transfer_hook.**TRANSFER_JOB_FIELD_MASK = update_transfer_job_fiel**

airflow.contrib.hooks.gcp_transfer_hook.**TRANSFER_JOBS = transferJobs**

airflow.contrib.hooks.gcp_transfer_hook.**TRANSFER_OPERATIONS = transferOperations**

airflow.contrib.hooks.gcp_transfer_hook.**TRANSFER_OPTIONS = transfer_options**

airflow.contrib.hooks.gcp_transfer_hook.**TRANSFER_SPEC = transferSpec**

airflow.contrib.hooks.gcp_transfer_hook.**YEAR = year**

airflow.contrib.hooks.gcp_transfer_hook.**NEGATIVE_STATUSES**

class airflow.contrib.hooks.gcp_transfer_hook.**GCPTransferServiceHook**(*api_version='v1'*, *gcp_conn_id='google_cloud_defa* *dele-gate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Storage Transfer Service.

_conn

get_conn (*self*)
Retrieves connection to Google Storage Transfer service.

Returns Google Storage Transfer service object

Return type dict

create_transfer_job (*self*, *body*)
Creates a transfer job that runs periodically.

Parameters **body** (*dict*) – (Required) A request body, as described in https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs/patch#request-body

Returns transfer job. See: https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs#TransferJob

Return type dict

get_transfer_job (*self*, *job_name*, *project_id=None*)
Gets the latest state of a long-running operation in Google Storage Transfer Service.

Parameters

- **job_name** (*str*) – (Required) Name of the job to be fetched
- **project_id** (*str*) – (Optional) the ID of the project that owns the Transfer Job. If set to None or missing, the default project_id from the GCP connection is used.

Returns Transfer Job

Return type dict

list_transfer_job (*self*, *filter*)
Lists long-running operations in Google Storage Transfer Service that match the specified filter.

Parameters **filter** (*dict*) – (Required) A request filter, as described in https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs/list#body.QUERY_PARAMETERS.filter

Returns List of Transfer Jobs

Return type list[dict]

update_transfer_job (*self*, *job_name*, *body*)
Updates a transfer job that runs periodically.

Parameters

- **job_name** (*str*) – (Required) Name of the job to be updated
- **body** (*dict*) – A request body, as described in https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs/patch#request-body

Returns If successful, TransferJob.

Return type dict

delete_transfer_job (*self*, *job_name*, *project_id*)
Deletes a transfer job. This is a soft delete. After a transfer job is deleted, the job and all the transfer executions are subject to garbage collection. Transfer jobs become eligible for garbage collection 30 days after soft delete.

Parameters

- **job_name** (*str*) – (Required) Name of the job to be deleted
- **project_id** (*str*) – (Optional) the ID of the project that owns the Transfer Job. If set to None or missing, the default project_id from the GCP connection is used.

Return type None

cancel_transfer_operation(*self*, *operation_name*)
Cancels an transfer operation in Google Storage Transfer Service.

Parameters **operation_name** (*str*) – Name of the transfer operation.

Return type None

get_transfer_operation(*self*, *operation_name*)
Gets an transfer operation in Google Storage Transfer Service.

Parameters **operation_name** (*str*) – (Required) Name of the transfer operation.

Returns transfer operation See: https://cloud.google.com/storage-transfer/docs/reference/rest/v1/Operation

Return type dict

list_transfer_operations(*self*, *filter*)
Gets an transfer operation in Google Storage Transfer Service.

Parameters **filter** (*dict*) – (Required) A request filter, as described in https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs/list#body.QUERY_PARAMETERS.filter With one additional improvement:

- project_id is optional if you have a project id defined in the connection See: *Google Cloud Platform Connection*

Returns transfer operation

Return type list[dict]

pause_transfer_operation(*self*, *operation_name*)
Pauses an transfer operation in Google Storage Transfer Service.

Parameters **operation_name** (*str*) – (Required) Name of the transfer operation.

Return type None

resume_transfer_operation(*self*, *operation_name*)
Resumes an transfer operation in Google Storage Transfer Service.

Parameters **operation_name** (*str*) – (Required) Name of the transfer operation.

Return type None

wait_for_transfer_job(*self*, *job*, *expected_statuses=(GcpTransferOperationStatus.SUCCESS,)*, *timeout=60*)
Waits until the job reaches the expected state.

Parameters

- **job** (*dict*) – Transfer job See: https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferJobs#TransferJob
- **expected_statuses** (*set[str]*) – State that is expected See: https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferOperations#Status
- **timeout** (*time in which the operation must end in seconds*) –

Return type None

_inject_project_id(*self*, *body*, *param_name*, *target_key*)

static operations_contain_expected_statuses(*operations*, *expected_statuses*)
Checks whether the operation list has an operation with the expected status, then returns true If it encounters operations in FAILED or ABORTED state throw `airflow.exceptions.AirflowException`.

Parameters

- **operations** (*list[dict]*) – (Required) List of transfer operations to check.

- **expected_statuses** (*set[str]*) – (Required) status that is expected See: https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferOperations#Status

Returns If there is an operation with the expected state in the operation list, returns true,

Raises airflow.exceptions.AirflowException If it encounters operations with a state in the list,

Return type bool

airflow.contrib.hooks.gcp_translate_hook

Module Contents

class airflow.contrib.hooks.gcp_translate_hook.**CloudTranslateHook**(*gcp_conn_id='google_cloud_default'*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud translate APIs.

_client

get_conn(*self*)

Retrieves connection to Cloud Translate

Returns Google Cloud Translate client object.

Return type Client

translate(*self*, *values*, *target_language*, *format_=None*, *source_language=None*, *model=None*)

Translate a string or list of strings.

See https://cloud.google.com/translate/docs/translating-text

Parameters

- **values** (*str or list*) – String or list of strings to translate.
- **target_language** (*str*) – The language to translate results into. This is required by the API and defaults to the target language of the current instance.
- **format** (*str*) – (Optional) One of `text` or `html`, to specify if the input text is plain text or HTML.
- **source_language** (*str or None*) – (Optional) The language of the text to be translated.
- **model** (*str or None*) – (Optional) The model used to translate the text, such as `'base'` or `'nmt'`.

Return type str or list

Returns

A list of dictionaries for each queried value. Each dictionary typically contains three keys (though not all will be present in all cases)

- `detectedSourceLanguage`: The detected language (as an ISO 639-1 language code) of the text.
- `translatedText`: The translation of the text into the target language.
- `input`: The corresponding input value.
- `model`: The model used to translate the text.

If only a single value is passed, then only a single dictionary will be returned.

Raises `ValueError` if the number of values and translations differ.

airflow.contrib.hooks.gcp_video_intelligence_hook

Module Contents

class airflow.contrib.hooks.gcp_video_intelligence_hook.**CloudVideoIntelligenceHook**(*gcp_conn_i del- e- gate_to=No*

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud Video Intelligence APIs.

Parameters

- **gcp_conn_id** (*str*) – The connection ID to use when fetching connection info.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

_conn

get_conn(*self*)

Returns Gcp Video Intelligence Service client

Return type google.cloud.videointelligence_v1.VideoIntelligenceServiceClient

annotate_video(*self*, *input_uri=None*, *input_content=None*, *features=None*, *video_context=None*, *output_uri=None*, *location=None*, *retry=None*, *timeout=None*, *metadata=None*)

Performs video annotation.

Parameters

- **input_uri** (*str*) – Input video location. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **input_content** (*bytes*) – The video data bytes. If unset, the input video(s) should be specified via `input_uri`. If set, `input_uri` should be unset.
- **features** (*list[google.cloud.videointelligence_v1.VideoIntelligenceServiceClient.enums.Feature]*) – Requested video annotation features.
- **output_uri** (*str*) – Optional, location where the output (in JSON format) should be stored. Currently, only Google Cloud Storage URIs are supported, which must be specified in the following format: `gs://bucket-id/object-id`.
- **video_context** (*dict or google.cloud.videointelligence_v1.types.VideoContext*) – Optional, Additional video context and/or feature-specific parameters.
- **location** (*str*) – Optional, cloud region where annotation should take place. Supported cloud regions: us-east1, us-west1, europe-west1, asia-east1. If no region is specified, a region will be determined based on video file location.
- **retry** (*google.api_core.retry.Retry*) – Retry object used to determine when/if to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – Optional, The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.
- **metadata** (*seq[tuple[str, str]]*) – Optional, Additional metadata that is provided to the method.

`airflow.contrib.hooks.gcp_vision_hook`

Module Contents

class `airflow.contrib.hooks.gcp_vision_hook.`**`NameDeterminer`**(*label*, *id_label*, *get_path*)

Class used for checking if the entity has the 'name' attribute set.

- If so, no action is taken.
- If not, and the name can be constructed from other parameters provided, it is created and filled in the entity.
- If both the entity's 'name' attribute is set and the name can be constructed from other parameters provided:
  - If they are the same - no action is taken
  - if they are different - an exception is thrown.

`get_entity_with_name`(*self*, *entity*, *entity_id*, *location*, *project_id*)

`_raise_ex_unable_to_determine_name`(*self*)

`_raise_ex_different_names`(*self*, *constructed_name*, *explicit_name*)

class `airflow.contrib.hooks.gcp_vision_hook.`**`CloudVisionHook`**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)

Bases: *`airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook`*

Hook for Google Cloud Vision APIs.

`_client`

`product_name_determiner`

`product_set_name_determiner`

`get_conn`(*self*)

Retrieves connection to Cloud Vision.

Returns Google Cloud Vision client object.

Return type google.cloud.vision_v1.ProductSearchClient

`annotator_client`(*self*)

static **`_check_for_error`**(*response*)

`create_product_set`(*self*, *location*, *product_set*, *project_id=None*, *product_set_id=None*, *retry=None*, *timeout=None*, *metadata=None*)

For the documentation see: *`CloudVisionProductSetCreateOperator`*

`get_product_set`(*self*, *location*, *product_set_id*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)

For the documentation see: *`CloudVisionProductSetGetOperator`*

`update_product_set`(*self*, *product_set*, *location=None*, *product_set_id=None*, *update_mask=None*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)

For the documentation see: *`CloudVisionProductSetUpdateOperator`*

`delete_product_set`(*self*, *location*, *product_set_id*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)

For the documentation see: *`CloudVisionProductSetDeleteOperator`*

`create_product`(*self*, *location*, *product*, *project_id=None*, *product_id=None*, *retry=None*, *timeout=None*, *metadata=None*)

For the documentation see: *`CloudVisionProductCreateOperator`*

`get_product`(*self*, *location*, *product_id*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)

For the documentation see: *`CloudVisionProductGetOperator`*

update_product (*self*, *product*, *location=None*, *product_id=None*, *update_mask=None*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)
For the documentation see: *CloudVisionProductUpdateOperator*

delete_product (*self*, *location*, *product_id*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)
For the documentation see: *CloudVisionProductDeleteOperator*

create_reference_image (*self*, *location*, *product_id*, *reference_image*, *reference_image_id=None*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)
For the documentation see: *CloudVisionReferenceImageCreateOperator*

delete_reference_image (*self*, *location*, *product_id*, *reference_image_id*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)
For the documentation see: *CloudVisionReferenceImageCreateOperator*

add_product_to_product_set (*self*, *product_set_id*, *product_id*, *location=None*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)
For the documentation see: *CloudVisionAddProductToProductSetOperator*

remove_product_from_product_set (*self*, *product_set_id*, *product_id*, *location=None*, *project_id=None*, *retry=None*, *timeout=None*, *metadata=None*)
For the documentation see: *CloudVisionRemoveProductFromProductSetOperator*

annotate_image (*self*, *request*, *retry=None*, *timeout=None*)
For the documentation see: CloudVisionAnnotateImage

batch_annotate_images (*self*, *requests*, *retry=None*, *timeout=None*)
For the documentation see: CloudVisionAnnotateImage

text_detection (*self*, *image*, *max_results=None*, *retry=None*, *timeout=None*, *additional_properties=None*)
For the documentation see: *CloudVisionDetectTextOperator*

document_text_detection (*self*, *image*, *max_results=None*, *retry=None*, *timeout=None*, *additional_properties=None*)
For the documentation see: *CloudVisionDetectDocumentTextOperator*

label_detection (*self*, *image*, *max_results=None*, *retry=None*, *timeout=None*, *additional_properties=None*)
For the documentation see: *CloudVisionDetectImageLabelsOperator*

safe_search_detection (*self*, *image*, *max_results=None*, *retry=None*, *timeout=None*, *additional_properties=None*)
For the documentation see: *CloudVisionDetectImageSafeSearchOperator*

static _get_autogenerated_id (*response*)

airflow.contrib.hooks.gcs_hook

This module contains a Google Cloud Storage hook.

Module Contents

class airflow.contrib.hooks.gcs_hook.**GoogleCloudStorageHook** (*google_cloud_storage_conn_id='google_cloud_*
delegate_to=None)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Interact with Google Cloud Storage. This hook uses the Google Cloud Platform connection.

_conn

get_conn (*self*)
Returns a Google Cloud Storage service object.

copy (*self*, *source_bucket*, *source_object*, *destination_bucket=None*, *destination_object=None*)
Copies an object from a bucket to another, with renaming if requested.

destination_bucket or destination_object can be omitted, in which case source bucket/object is used, but not both.

Parameters

- **source_bucket** (*str*) – The bucket of the object to copy from.
- **source_object** (*str*) – The object to copy.
- **destination_bucket** (*str*) – The destination of the object to copied to. Can be omitted; then the same bucket is used.
- **destination_object** (*str*) – The (renamed) path of the object if given. Can be omitted; then the same name is used.

rewrite (*self*, *source_bucket*, *source_object*, *destination_bucket*, *destination_object=None*)
Has the same functionality as copy, except that will work on files over 5 TB, as well as when copying between locations and/or storage classes.

destination_object can be omitted, in which case source_object is used.

Parameters

- **source_bucket** (*str*) – The bucket of the object to copy from.
- **source_object** (*str*) – The object to copy.
- **destination_bucket** (*str*) – The destination of the object to copied to.
- **destination_object** (*str*) – The (renamed) path of the object if given. Can be omitted; then the same name is used.

download (*self*, *bucket_name*, *object_name*, *filename=None*)
Get a file from Google Cloud Storage.

Parameters

- **bucket_name** (*str*) – The bucket to fetch from.
- **object_name** (*str*) – The object to fetch.
- **filename** (*str*) – If set, a local file path where the file should be written to.

upload (*self*, *bucket_name*, *object_name*, *filename*, *mime_type='application/octet-stream'*, *gzip=False*)
Uploads a local file to Google Cloud Storage.

Parameters

- **bucket_name** (*str*) – The bucket to upload to.
- **object_name** (*str*) – The object name to set when uploading the local file.
- **filename** (*str*) – The local file path to the file to be uploaded.
- **mime_type** (*str*) – The MIME type to set when uploading the file.
- **gzip** (*bool*) – Option to compress file for upload

exists (*self*, *bucket_name*, *object_name*)
Checks for the existence of a file in Google Cloud Storage.

Parameters

- **bucket_name** (*str*) – The Google cloud storage bucket where the object is.

- **object_name** (*str*) – The name of the blob_name to check in the Google cloud storage bucket.

is_updated_after (*self*, *bucket_name*, *object_name*, *ts*)
Checks if an blob_name is updated in Google Cloud Storage.

Parameters

- **bucket_name** (*str*) – The Google cloud storage bucket where the object is.
- **object_name** (*str*) – The name of the object to check in the Google cloud storage bucket.
- **ts** (*datetime.datetime*) – The timestamp to check against.

delete (*self*, *bucket_name*, *object_name*)
Deletes an object from the bucket.

Parameters

- **bucket_name** (*str*) – name of the bucket, where the object resides
- **object_name** (*str*) – name of the object to delete

list (*self*, *bucket_name*, *versions=None*, *max_results=None*, *prefix=None*, *delimiter=None*)
List all objects from the bucket with the give string prefix in name

Parameters

- **bucket_name** (*str*) – bucket name
- **versions** (*bool*) – if true, list all versions of the objects
- **max_results** (*int*) – max count of items to return in a single page of responses
- **prefix** (*str*) – prefix string which filters objects whose name begin with this prefix
- **delimiter** (*str*) – filters objects based on the delimiter (for e.g '.csv')

Returns a stream of object names matching the filtering criteria

get_size (*self*, *bucket_name*, *object_name*)
Gets the size of a file in Google Cloud Storage.

Parameters

- **bucket_name** (*str*) – The Google cloud storage bucket where the blob_name is.
- **object_name** (*str*) – The name of the object to check in the Google cloud storage bucket_name.

get_crc32c (*self*, *bucket_name*, *object_name*)
Gets the CRC32c checksum of an object in Google Cloud Storage.

Parameters

- **bucket_name** (*str*) – The Google cloud storage bucket where the blob_name is.
- **object_name** (*str*) – The name of the object to check in the Google cloud storage bucket_name.

get_md5hash (*self*, *bucket_name*, *object_name*)
Gets the MD5 hash of an object in Google Cloud Storage.

Parameters

- **bucket_name** (*str*) – The Google cloud storage bucket where the blob_name is.
- **object_name** (*str*) – The name of the object to check in the Google cloud storage bucket_name.

create_bucket (*self*, *bucket_name*, *resource=None*, *storage_class='MULTI_REGIONAL'*, *location='US'*, *project_id=None*, *labels=None*)

Creates a new bucket. Google Cloud Storage uses a flat namespace, so you can't create a bucket with a name that is already in use.

See also:

For more information, see Bucket Naming Guidelines: https://cloud.google.com/storage/docs/bucketnaming.html#requirements

Parameters

- **bucket_name** (*str*) – The name of the bucket.
- **resource** (*dict*) – An optional dict with parameters for creating the bucket. For information on available parameters, see Cloud Storage API doc: https://cloud.google.com/storage/docs/json_api/v1/buckets/insert
- **storage_class** (*str*) – This defines how objects in the bucket are stored and determines the SLA and the cost of storage. Values include
  - `MULTI_REGIONAL`
  - `REGIONAL`
  - `STANDARD`
  - `NEARLINE`
  - `COLDLINE`.

  If this value is not specified when the bucket is created, it will default to STANDARD.
- **location** (*str*) – The location of the bucket. Object data for objects in the bucket resides in physical storage within this region. Defaults to US.

  See also:

  https://developers.google.com/storage/docs/bucket-locations
- **project_id** (*str*) – The ID of the GCP Project.
- **labels** (*dict*) – User-provided labels, in key/value pairs.

Returns If successful, it returns the `id` of the bucket.

insert_bucket_acl (*self*, *bucket_name*, *entity*, *role*, *user_project=None*)

Creates a new ACL entry on the specified bucket_name. See: https://cloud.google.com/storage/docs/json_api/v1/bucketAccessControls/insert

Parameters

- **bucket_name** (*str*) – Name of a bucket_name.
- **entity** (*str*) – The entity holding the permission, in one of the following forms: user-userId, user-email, group-groupId, group-email, domain-domain, project-team-projectId, allUsers, allAuthenticatedUsers. See: https://cloud.google.com/storage/docs/access-control/lists#scopes
- **role** (*str*) – The access permission for the entity. Acceptable values are: "OWNER", "READER", "WRITER".
- **user_project** (*str*) – (Optional) The project to be billed for this request. Required for Requester Pays buckets.

insert_object_acl (*self*, *bucket_name*, *object_name*, *entity*, *role*, *user_project=None*)

Creates a new ACL entry on the specified object. See: https://cloud.google.com/storage/docs/json_api/v1/objectAccessControls/insert

Parameters

- **bucket_name** (*str*) – Name of a bucket_name.
- **object_name** (*str*) – Name of the object. For information about how to URL encode object names to be path safe, see: https://cloud.google.com/storage/docs/json_api/#encoding
- **entity** (*str*) – The entity holding the permission, in one of the following forms: user-userId, user-email, group-groupId, group-email, domain-domain, project-team-projectId, allUsers, allAuthenticatedUsers See: https://cloud.google.com/storage/docs/access-control/lists#scopes
- **role** (*str*) – The access permission for the entity. Acceptable values are: "OWNER", "READER".
- **user_project** (*str*) – (Optional) The project to be billed for this request. Required for Requester Pays buckets.

compose (*self*, *bucket_name*, *source_objects*, *destination_object*)

Composes a list of existing object into a new object in the same storage bucket_name

Currently it only supports up to 32 objects that can be concatenated in a single operation

https://cloud.google.com/storage/docs/json_api/v1/objects/compose

Parameters

- **bucket_name** (*str*) – The name of the bucket containing the source objects. This is also the same bucket to store the composed destination object.
- **source_objects** (*list*) – The list of source objects that will be composed into a single object.
- **destination_object** (*str*) – The path of the object if given.

airflow.contrib.hooks.gcs_hook.**_parse_gcs_url** (*gsurl*)

Given a Google Cloud Storage URL (gs://<bucket>/<blob>), returns a tuple containing the corresponding bucket and blob.

airflow.contrib.hooks.grpc_hook

Module Contents

class airflow.contrib.hooks.grpc_hook.**GrpcHook** (*grpc_conn_id*, *interceptors=None*, *custom_connection_func=None*)

Bases: *airflow.hooks.base_hook.BaseHook*

General interaction with gRPC servers.

get_conn (*self*)

run (*self*, *stub_class*, *call_func*, *streaming=False*, *data={}*)

_get_field (*self*, *field_name*, *default=None*)

Fetches a field from extras, and returns it. This is some Airflow magic. The grpc hook type adds custom UI elements to the hook page, which allow admins to specify scopes, credential pem files, etc. They get formatted as shown below.

airflow.contrib.hooks.imap_hook

Module Contents

class airflow.contrib.hooks.imap_hook.**ImapHook**(*imap_conn_id='imap_default'*)

Bases: *airflow.hooks.base_hook.BaseHook*

This hook connects to a mail server by using the imap protocol.

Parameters **imap_conn_id** (*str*) – The connection id that contains the information used to authenticate the client.

__enter__(*self*)

__exit__(*self*, *exc_type*, *exc_val*, *exc_tb*)

has_mail_attachment(*self*, *name*, *mail_folder='INBOX'*, *check_regex=False*)

Checks the mail folder for mails containing attachments with the given name.

Parameters

- **name** (*str*) – The name of the attachment that will be searched for.
- **mail_folder** (*str*) – The mail folder where to look at.
- **check_regex** (*bool*) – Checks the name for a regular expression.

Returns True if there is an attachment with the given name and False if not.

Return type bool

retrieve_mail_attachments(*self*, *name*, *mail_folder='INBOX'*, *check_regex=False*, *latest_only=False*, *not_found_mode='raise'*)

Retrieves mail's attachments in the mail folder by its name.

Parameters

- **name** (*str*) – The name of the attachment that will be downloaded.
- **mail_folder** (*str*) – The mail folder where to look at.
- **check_regex** (*bool*) – Checks the name for a regular expression.
- **latest_only** (*bool*) – If set to True it will only retrieve the first matched attachment.
- **not_found_mode** (*str*) – Specify what should happen if no attachment has been found. Supported values are 'raise', 'warn' and 'ignore'. If it is set to 'raise' it will raise an exception, if set to 'warn' it will only print a warning and if set to 'ignore' it won't notify you at all.

Returns a list of tuple each containing the attachment filename and its payload.

Return type a list of tuple

download_mail_attachments(*self*, *name*, *local_output_directory*, *mail_folder='INBOX'*, *check_regex=False*, *latest_only=False*, *not_found_mode='raise'*)

Downloads mail's attachments in the mail folder by its name to the local directory.

Parameters

- **name** (*str*) – The name of the attachment that will be downloaded.
- **local_output_directory** (*str*) – The output directory on the local machine where the files will be downloaded to.
- **mail_folder** (*str*) – The mail folder where to look at.
- **check_regex** (*bool*) – Checks the name for a regular expression.
- **latest_only** (*bool*) – If set to True it will only download the first matched attachment.

- **`not_found_mode`** (*`str`*) – Specify what should happen if no attachment has been found. Supported values are 'raise', 'warn' and 'ignore'. If it is set to 'raise' it will raise an exception, if set to 'warn' it will only print a warning and if set to 'ignore' it won't notify you at all.

`_handle_not_found_mode`(*self*, *not_found_mode*)

`_retrieve_mails_attachments_by_name`(*self*, *name*, *mail_folder*, *check_regex*, *latest_only*)

`_list_mail_ids_desc`(*self*)

`_fetch_mail_body`(*self*, *mail_id*)

`_check_mail_body`(*self*, *response_mail_body*, *name*, *check_regex*, *latest_only*)

`_create_files`(*self*, *mail_attachments*, *local_output_directory*)

`_is_symlink`(*self*, *name*)

`_is_escaping_current_directory`(*self*, *name*)

`_correct_path`(*self*, *name*, *local_output_directory*)

`_create_file`(*self*, *name*, *payload*, *local_output_directory*)

class `airflow.contrib.hooks.imap_hook.`**`Mail`**(*mail_body*)

Bases: `airflow.LoggingMixin`

This class simplifies working with mails returned by the imaplib client.

Parameters **`mail_body`** (*`str`*) – The mail body of a mail received from imaplib client.

`has_attachments`(*self*)

Checks the mail for a attachments.

Returns True if it has attachments and False if not.

Return type bool

`get_attachments_by_name`(*self*, *name*, *check_regex*, *find_first=False*)

Gets all attachments by name for the mail.

Parameters

- **`name`** (*`str`*) – The name of the attachment to look for.
- **`check_regex`** (*`bool`*) – Checks the name for a regular expression.
- **`find_first`** (*`bool`*) – If set to True it will only find the first match and then quit.

Returns a list of tuples each containing name and payload where the attachments name matches the given name.

Return type list of tuple

class `airflow.contrib.hooks.imap_hook.`**`MailPart`**(*part*)

This class is a wrapper for a Mail object's part and gives it more features.

Parameters **`part`** (*`any`*) – The mail part in a Mail object.

`is_attachment`(*self*)

Checks if the part is a valid mail attachment.

Returns True if it is an attachment and False if not.

Return type bool

`has_matching_name`(*self*, *name*)

Checks if the given name matches the part's name.

Parameters **`name`** (*`str`*) – The name to look for.

Returns True if it matches the name (including regular expression).

Return type tuple

has_equal_name (*self*, *name*)
Checks if the given name is equal to the part's name.

Parameters **name** (*str*) – The name to look for.

Returns True if it is equal to the given name.

Return type bool

get_file (*self*)
Gets the file including name and payload.

Returns the part's name and payload.

Return type tuple

airflow.contrib.hooks.jenkins_hook

Module Contents

class airflow.contrib.hooks.jenkins_hook.**JenkinsHook** (*conn_id='jenkins_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Hook to manage connection to jenkins server

get_jenkins_server (*self*)

airflow.contrib.hooks.jira_hook

Module Contents

class airflow.contrib.hooks.jira_hook.**JiraHook** (*jira_conn_id='jira_default'*, *proxies=None*)
Bases: *airflow.hooks.base_hook.BaseHook*

Jira interaction hook, a Wrapper around JIRA Python SDK.

Parameters **jira_conn_id** (*str*) – reference to a pre-defined Jira Connection

get_conn (*self*)

airflow.contrib.hooks.mongo_hook

Module Contents

class airflow.contrib.hooks.mongo_hook.**MongoHook** (*conn_id='mongo_default'*, **args*, ***kwargs*)
Bases: *airflow.hooks.base_hook.BaseHook*

PyMongo Wrapper to Interact With Mongo Database Mongo Connection Documentation https://docs.mongodb.com/manual/reference/connection-string/index.html You can specify connection string options in extra field of your connection https://docs.mongodb.com/manual/reference/connection-string/index.html#connection-string-options

If you want use DNS seedlist, set *srv* to True.

ex. {"srv": true, "replicaSet": "test", "ssl": true, "connectTimeoutMS": 30000}

conn_type = mongo

__enter__(*self*)

__exit__(*self*, *exc_type*, *exc_val*, *exc_tb*)

get_conn(*self*)
Fetches PyMongo Client

close_conn(*self*)

get_collection(*self*, *mongo_collection*, *mongo_db=None*)
Fetches a mongo collection object for querying.

Uses connection schema as DB unless specified.

aggregate(*self*, *mongo_collection*, *aggregate_query*, *mongo_db=None*, ***kwargs*)
Runs an aggregation pipeline and returns the results https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.aggregate https://api.mongodb.com/python/current/examples/aggregation.html

find(*self*, *mongo_collection*, *query*, *find_one=False*, *mongo_db=None*, ***kwargs*)
Runs a mongo find query and returns the results https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.find

insert_one(*self*, *mongo_collection*, *doc*, *mongo_db=None*, ***kwargs*)
Inserts a single document into a mongo collection https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.insert_one

insert_many(*self*, *mongo_collection*, *docs*, *mongo_db=None*, ***kwargs*)
Inserts many docs into a mongo collection. https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.insert_many

update_one(*self*, *mongo_collection*, *filter_doc*, *update_doc*, *mongo_db=None*, ***kwargs*)
Updates a single document in a mongo collection. https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.update_one

Parameters

- **mongo_collection** (`str`) – The name of the collection to update.
- **filter_doc** (`dict`) – A query that matches the documents to update.
- **update_doc** (`dict`) – The modifications to apply.
- **mongo_db** (`str`) – The name of the database to use. Can be omitted; then the database from the connection string is used.

update_many(*self*, *mongo_collection*, *filter_doc*, *update_doc*, *mongo_db=None*, ***kwargs*)
Updates one or more documents in a mongo collection. https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.update_many

Parameters

- **mongo_collection** (`str`) – The name of the collection to update.
- **filter_doc** (`dict`) – A query that matches the documents to update.
- **update_doc** (`dict`) – The modifications to apply.
- **mongo_db** (`str`) – The name of the database to use. Can be omitted; then the database from the connection string is used.

replace_one(*self*, *mongo_collection*, *doc*, *filter_doc=None*, *mongo_db=None*, ***kwargs*)
Replaces a single document in a mongo collection. https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.replace_one

Note: If no `filter_doc` is given, it is assumed that the replacement document contain the `_id` field which is then used as filters.

Parameters

- **`mongo_collection`** (*`str`*) – The name of the collection to update.
- **`doc`** (*`dict`*) – The new document.
- **`filter_doc`** (*`dict`*) – A query that matches the documents to replace. Can be omitted; then the _id field from doc will be used.
- **`mongo_db`** (*`str`*) – The name of the database to use. Can be omitted; then the database from the connection string is used.

`replace_many`(*self*, *mongo_collection*, *docs*, *filter_docs=None*, *mongo_db=None*, *upsert=False*, *collation=None*, ***kwargs*)

Replaces many documents in a mongo collection.

Uses bulk_write with multiple ReplaceOne operations https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.bulk_write

Note: If no `filter_docs``are given, it is assumed that all replacement documents contain the ``_id` field which are then used as filters.

Parameters

- **`mongo_collection`** (*`str`*) – The name of the collection to update.
- **`docs`** (*`list[dict]`*) – The new documents.
- **`filter_docs`** (*`list[dict]`*) – A list of queries that match the documents to replace. Can be omitted; then the _id fields from docs will be used.
- **`mongo_db`** (*`str`*) – The name of the database to use. Can be omitted; then the database from the connection string is used.
- **`upsert`** (*`bool`*) – If `True`, perform an insert if no documents match the filters for the replace operation.
- **`collation`** (*`pymongo.collation.Collation`*) – An instance of `Collation`. This option is only supported on MongoDB 3.4 and above.

`delete_one`(*self*, *mongo_collection*, *filter_doc*, *mongo_db=None*, ***kwargs*)

Deletes a single document in a mongo collection. https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.delete_one

Parameters

- **`mongo_collection`** (*`str`*) – The name of the collection to delete from.
- **`filter_doc`** (*`dict`*) – A query that matches the document to delete.
- **`mongo_db`** (*`str`*) – The name of the database to use. Can be omitted; then the database from the connection string is used.

`delete_many`(*self*, *mongo_collection*, *filter_doc*, *mongo_db=None*, ***kwargs*)

Deletes one or more documents in a mongo collection. https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.delete_many

Parameters

- **mongo_collection** (*str*) – The name of the collection to delete from.
- **filter_doc** (*dict*) – A query that matches the documents to delete.
- **mongo_db** (*str*) – The name of the database to use. Can be omitted; then the database from the connection string is used.

airflow.contrib.hooks.openfaas_hook

Module Contents

airflow.contrib.hooks.openfaas_hook.**OK_STATUS_CODE = 202**

class airflow.contrib.hooks.openfaas_hook.**OpenFaasHook**(*function_name=None, conn_id='open_faas_default', *args, **kwargs*)

Bases: *airflow.hooks.base_hook.BaseHook*

Interact with Openfaas to query, deploy, invoke and update function

Parameters

- **function_name** – Name of the function, Defaults to None
- **conn_id** (*str*) – openfass connection to use, Defaults to open_faas_default for example host : http://openfaas.faas.com, Conn Type : Http

GET_FUNCTION = /system/function/

INVOKE_ASYNC_FUNCTION = /async-function/

DEPLOY_FUNCTION = /system/functions

UPDATE_FUNCTION = /system/functions

get_conn(*self*)

deploy_function(*self, overwrite_function_if_exist, body*)

invoke_async_function(*self, body*)

update_function(*self, body*)

does_function_exist(*self*)

airflow.contrib.hooks.opsgenie_alert_hook

Module Contents

class airflow.contrib.hooks.opsgenie_alert_hook.**OpsgenieAlertHook**(*opsgenie_conn_id='opsgenie_default', *args, **kwargs*)

Bases: *airflow.hooks.http_hook.HttpHook*

This hook allows you to post alerts to Opsgenie. Accepts a connection that has an Opsgenie API key as the connection's password. This hook sets the domain to conn_id.host, and if not set will default to `https://api.opsgenie.com`.

Each Opsgenie API key can be pre-configured to a team integration. You can override these defaults in this hook.

Parameters **opsgenie_conn_id** (*str*) – The name of the Opsgenie connection to use

_get_api_key(*self*)
Get Opsgenie api_key for creating alert

get_conn (*self*, *headers=None*)
Overwrite HttpHook get_conn because this hook just needs base_url and headers, and does not need generic params

Parameters headers (*dict*) – additional headers to be passed through as a dictionary

execute (*self*, *payload={}*)
Execute the Opsgenie Alert call

Parameters payload (*dict*) – Opsgenie API Create Alert payload values See https://docs.opsgenie.com/docs/alert-api#section-create-alert

airflow.contrib.hooks.pinot_hook

Module Contents

class airflow.contrib.hooks.pinot_hook.**PinotDbApiHook** (**args*, ***kwargs*)
Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Connect to pinot db(https://github.com/linkedin/pinot) to issue pql

conn_name_attr = pinot_broker_conn_id

default_conn_name = pinot_broker_default

supports_autocommit = False

get_conn (*self*)
Establish a connection to pinot broker through pinot dbqpi.

get_uri (*self*)
Get the connection uri for pinot broker.

e.g: http://localhost:9000/pql

get_records (*self*, *sql*)
Executes the sql and returns a set of records.

Parameters sql (*str*) – the sql statement to be executed (str) or a list of sql statements to execute

get_first (*self*, *sql*)
Executes the sql and returns the first resulting row.

Parameters sql (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute

set_autocommit (*self*, *conn*, *autocommit*)

get_pandas_df (*self*, *sql*, *parameters=None*)

insert_rows (*self*, *table*, *rows*, *target_fields=None*, *commit_every=1000*)

airflow.contrib.hooks.qubole_check_hook

Module Contents

airflow.contrib.hooks.qubole_check_hook.**COL_DELIM =**

airflow.contrib.hooks.qubole_check_hook.**ROW_DELIM =**

airflow.contrib.hooks.qubole_check_hook.**isint** (*value*)

airflow.contrib.hooks.qubole_check_hook.**isfloat** (*value*) airflow.contrib.hooks.qubole_check_hook.**isbool**(*value*)

airflow.contrib.hooks.qubole_check_hook.**parse_first_row**(*row_list*)

class airflow.contrib.hooks.qubole_check_hook.**QuboleCheckHook**(*context*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.qubole_hook.QuboleHook*

static handle_failure_retry(*context*)

get_first(*self*, *sql*)

get_query_results(*self*)

airflow.contrib.hooks.qubole_hook

Module Contents

airflow.contrib.hooks.qubole_hook.**COMMAND_CLASSES**

airflow.contrib.hooks.qubole_hook.**POSITIONAL_ARGS**

airflow.contrib.hooks.qubole_hook.**flatten_list**(*list_of_lists*)

airflow.contrib.hooks.qubole_hook.**filter_options**(*options*)

airflow.contrib.hooks.qubole_hook.**get_options_list**(*command_class*)

airflow.contrib.hooks.qubole_hook.**build_command_args**()

class airflow.contrib.hooks.qubole_hook.**QuboleHook**(**args*, ***kwargs*)

Bases: *airflow.hooks.base_hook.BaseHook*

static handle_failure_retry(*context*)

execute(*self*, *context*)

kill(*self*, *ti*)

Kill (cancel) a Qubole command :param ti: Task Instance of the dag, used to determine the Quboles command id :return: response from Qubole

get_results(*self*, *ti=None*, *fp=None*, *inline=True*, *delim=None*, *fetch=True*)

Get results (or just s3 locations) of a command from Qubole and save into a file :param ti: Task Instance of the dag, used to determine the Quboles command id :param fp: Optional file pointer, will create one and return if None passed :param inline: True to download actual results, False to get s3 locations only :param delim: Replaces the CTL-A chars with the given delim, defaults to ',' :param fetch: when inline is True, get results directly from s3 (if large) :return: file location containing actual results or s3 locations of results

get_log(*self*, *ti*)

Get Logs of a command from Qubole :param ti: Task Instance of the dag, used to determine the Quboles command id :return: command log as text

get_jobs_id(*self*, *ti*)

Get jobs associated with a Qubole commands :param ti: Task Instance of the dag, used to determine the Quboles command id :return: Job information associated with command

get_extra_links(*self*, *operator*, *dttm*)

Get link to qubole command result page.

Parameters

- **operator** – operator
- **dttm** – datetime

Returns url link

create_cmd_args (*self*, *context*)

airflow.contrib.hooks.redis_hook

RedisHook module

Module Contents

class airflow.contrib.hooks.redis_hook.**RedisHook** (*redis_conn_id='redis_default'*)

Bases: *airflow.hooks.base_hook.BaseHook*

Wrapper for connection to interact with Redis in-memory data structure store

get_conn (*self*)

Returns a Redis connection.

airflow.contrib.hooks.redshift_hook

Module Contents

class airflow.contrib.hooks.redshift_hook.**RedshiftHook**

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Redshift, using the boto3 library

get_conn (*self*)

cluster_status (*self*, *cluster_identifier*)

Return status of a cluster

Parameters **cluster_identifier** (*str*) – unique identifier of a cluster

delete_cluster (*self*, *cluster_identifier*, *skip_final_cluster_snapshot=True*, *final_cluster_snapshot_identifier=''*)

Delete a cluster and optionally create a snapshot

Parameters

- **cluster_identifier** (*str*) – unique identifier of a cluster
- **skip_final_cluster_snapshot** (*bool*) – determines cluster snapshot creation
- **final_cluster_snapshot_identifier** (*str*) – name of final cluster snapshot

describe_cluster_snapshots (*self*, *cluster_identifier*)

Gets a list of snapshots for a cluster

Parameters **cluster_identifier** (*str*) – unique identifier of a cluster

restore_from_cluster_snapshot (*self*, *cluster_identifier*, *snapshot_identifier*)

Restores a cluster from its snapshot

Parameters

- **cluster_identifier** (*str*) – unique identifier of a cluster
- **snapshot_identifier** (*str*) – unique identifier for a snapshot of a cluster

create_cluster_snapshot (*self*, *snapshot_identifier*, *cluster_identifier*)

Creates a snapshot of a cluster

Parameters

- **snapshot_identifier** (*str*) – unique identifier for a snapshot of a cluster
- **cluster_identifier** (*str*) – unique identifier of a cluster

airflow.contrib.hooks.sagemaker_hook

Module Contents

class airflow.contrib.hooks.sagemaker_hook.**LogState**

STARTING = 1

WAIT_IN_PROGRESS = 2

TAILING = 3

JOB_COMPLETE = 4

COMPLETE = 5

airflow.contrib.hooks.sagemaker_hook.**Position**

airflow.contrib.hooks.sagemaker_hook.**argmin**(*arr*, *f*)

Return the index, i, in arr that minimizes f(arr[i])

airflow.contrib.hooks.sagemaker_hook.**secondary_training_status_changed**(*current_job_description*, *prev_job_description*)

Returns true if training job's secondary status message has changed.

Parameters

- **current_job_description** (*dict*) – Current job description, returned from DescribeTrainingJob call.
- **prev_job_description** (*dict*) – Previous job description, returned from DescribeTrainingJob call.

Returns Whether the secondary status message of a training job changed or not.

airflow.contrib.hooks.sagemaker_hook.**secondary_training_status_message**(*job_description*, *prev_description*)

Returns a string contains start time and the secondary training job status message.

Parameters

- **job_description** (*dict*) – Returned response from DescribeTrainingJob call
- **prev_description** (*dict*) – Previous job description from DescribeTrainingJob call

Returns Job status string to be printed.

class airflow.contrib.hooks.sagemaker_hook.**SageMakerHook**(**args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with Amazon SageMaker.

non_terminal_states

endpoint_non_terminal_states

failed_states

tar_and_s3_upload(*self*, *path*, *key*, *bucket*)

Tar the local file or directory and upload to s3

Parameters

- **path** (*str*) – local file or directory
- **key** (*str*) – s3 key
- **bucket** (*str*) – s3 bucket

Returns None

configure_s3_resources (*self*, *config*)
Extract the S3 operations from the configuration and execute them.

Parameters **config** (*dict*) – config of SageMaker operation

Return type dict

check_s3_url (*self*, *s3url*)
Check if an S3 URL exists

Parameters **s3url** (*str*) – S3 url

Return type bool

check_training_config (*self*, *training_config*)
Check if a training configuration is valid

Parameters **training_config** (*dict*) – training_config

Returns None

check_tuning_config (*self*, *tuning_config*)
Check if a tuning configuration is valid

Parameters **tuning_config** (*dict*) – tuning_config

Returns None

get_conn (*self*)
Establish an AWS connection for SageMaker

Return type `SageMaker.Client`

get_log_conn (*self*)
Establish an AWS connection for retrieving logs during training

Return type CloudWatchLogs.Client

log_stream (*self*, *log_group*, *stream_name*, *start_time=0*, *skip=0*)
A generator for log items in a single stream. This will yield all the items that are available at the current moment.

Parameters

- **log_group** (*str*) – The name of the log group.
- **stream_name** (*str*) – The name of the specific stream.
- **start_time** (*int*) – The time stamp value to start reading the logs from (default: 0).
- **skip** (*int*) – The number of log entries to skip at the start (default: 0). This is for when there are multiple entries at the same timestamp.

Return type dict

Returns

A CloudWatch log event with the following key-value pairs:
'timestamp' (int): The time in milliseconds of the event.
'message' (str): The log event data.

'ingestionTime' (int): The time in milliseconds the event was ingested.

multi_stream_iter(*self*, *log_group*, *streams*, *positions=None*)
Iterate over the available events coming from a set of log streams in a single log group interleaving the events from each stream so they're yielded in timestamp order.

Parameters

- **log_group** (*str*) – The name of the log group.
- **streams** (*list*) – A list of the log stream names. The position of the stream in this list is the stream number.
- **positions** (*list*) – A list of pairs of (timestamp, skip) which represents the last record read from each stream.

Returns A tuple of (stream number, cloudwatch log event).

create_training_job(*self*, *config*, *wait_for_completion=True*, *print_log=True*, *check_interval=30*, *max_ingestion_time=None*)
Create a training job

Parameters

- **config** (*dict*) – the config for training
- **wait_for_completion** (*bool*) – if the program should keep running until job finishes
- **check_interval** (*int*) – the time interval in seconds which the operator will check the status of any SageMaker job
- **max_ingestion_time** (*int*) – the maximum ingestion time in seconds. Any Sage-Maker jobs that run longer than this will fail. Setting this to None implies no timeout for any SageMaker job.

Returns A response to training job creation

create_tuning_job(*self*, *config*, *wait_for_completion=True*, *check_interval=30*, *max_ingestion_time=None*)
Create a tuning job

Parameters

- **config** (*dict*) – the config for tuning
- **wait_for_completion** (*bool*) – if the program should keep running until job finishes
- **check_interval** (*int*) – the time interval in seconds which the operator will check the status of any SageMaker job
- **max_ingestion_time** (*int*) – the maximum ingestion time in seconds. Any Sage-Maker jobs that run longer than this will fail. Setting this to None implies no timeout for any SageMaker job.

Returns A response to tuning job creation

create_transform_job(*self*, *config*, *wait_for_completion=True*, *check_interval=30*, *max_ingestion_time=None*)
Create a transform job

Parameters

- **config** (*dict*) – the config for transform job
- **wait_for_completion** (*bool*) – if the program should keep running until job finishes
- **check_interval** (*int*) – the time interval in seconds which the operator will check the status of any SageMaker job

- **max_ingestion_time** (*int*) – the maximum ingestion time in seconds. Any SageMaker jobs that run longer than this will fail. Setting this to None implies no timeout for any SageMaker job.

Returns A response to transform job creation

create_model(*self*, *config*)
Create a model job

Parameters **config** (*dict*) – the config for model

Returns A response to model creation

create_endpoint_config(*self*, *config*)
Create an endpoint config

Parameters **config** (*dict*) – the config for endpoint-config

Returns A response to endpoint config creation

create_endpoint(*self*, *config*, *wait_for_completion=True*, *check_interval=30*, *max_ingestion_time=None*)
Create an endpoint

Parameters

- **config** (*dict*) – the config for endpoint
- **wait_for_completion** (*bool*) – if the program should keep running until job finishes
- **check_interval** (*int*) – the time interval in seconds which the operator will check the status of any SageMaker job
- **max_ingestion_time** (*int*) – the maximum ingestion time in seconds. Any SageMaker jobs that run longer than this will fail. Setting this to None implies no timeout for any SageMaker job.

Returns A response to endpoint creation

update_endpoint(*self*, *config*, *wait_for_completion=True*, *check_interval=30*, *max_ingestion_time=None*)
Update an endpoint

Parameters

- **config** (*dict*) – the config for endpoint
- **wait_for_completion** (*bool*) – if the program should keep running until job finishes
- **check_interval** (*int*) – the time interval in seconds which the operator will check the status of any SageMaker job
- **max_ingestion_time** (*int*) – the maximum ingestion time in seconds. Any SageMaker jobs that run longer than this will fail. Setting this to None implies no timeout for any SageMaker job.

Returns A response to endpoint update

describe_training_job(*self*, *name*)
Return the training job info associated with the name

Parameters **name** (*str*) – the name of the training job

Returns A dict contains all the training job info

describe_training_job_with_log(*self*, *job_name*, *positions*, *stream_names*, *instance_count*, *state*, *last_description*, *last_describe_job_call*)
Return the training job info associated with job_name and print CloudWatch logs

describe_tuning_job(*self*, *name*)
Return the tuning job info associated with the name

Parameters **name** (*string*) – the name of the tuning job

Returns A dict contains all the tuning job info

describe_model(*self*, *name*)
Return the SageMaker model info associated with the name

Parameters **name** (*string*) – the name of the SageMaker model

Returns A dict contains all the model info

describe_transform_job(*self*, *name*)
Return the transform job info associated with the name

Parameters **name** (*string*) – the name of the transform job

Returns A dict contains all the transform job info

describe_endpoint_config(*self*, *name*)
Return the endpoint config info associated with the name

Parameters **name** (*string*) – the name of the endpoint config

Returns A dict contains all the endpoint config info

describe_endpoint(*self*, *name*)

Parameters **name** (*string*) – the name of the endpoint

Returns A dict contains all the endpoint info

check_status(*self*, *job_name*, *key*, *describe_function*, *check_interval*, *max_ingestion_time*, *non_terminal_states=None*)
Check status of a SageMaker job

Parameters

- **job_name** (*str*) – name of the job to check status
- **key** (*str*) – the key of the response dict that points to the state
- **describe_function** (*python callable*) – the function used to retrieve the status
- **args** – the arguments for the function
- **check_interval** (*int*) – the time interval in seconds which the operator will check the status of any SageMaker job
- **max_ingestion_time** (*int*) – the maximum ingestion time in seconds. Any Sage-Maker jobs that run longer than this will fail. Setting this to None implies no timeout for any SageMaker job.
- **non_terminal_states** (*set*) – the set of nonterminal states

Returns response of describe call after job is done

check_training_status_with_log(*self*, *job_name*, *non_terminal_states*, *failed_states*, *wait_for_completion*, *check_interval*, *max_ingestion_time*)
Display the logs for a given training job, optionally tailing them until the job is complete.

Parameters

- **job_name** (*str*) – name of the training job to check status and display logs for
- **non_terminal_states** (*set*) – the set of non_terminal states
- **failed_states** (*set*) – the set of failed states

- **wait_for_completion** (*bool*) – Whether to keep looking for new log entries until the job completes
- **check_interval** (*int*) – The interval in seconds between polling for new log entries and job completion
- **max_ingestion_time** (*int*) – the maximum ingestion time in seconds. Any SageMaker jobs that run longer than this will fail. Setting this to None implies no timeout for any SageMaker job.

Returns None

airflow.contrib.hooks.salesforce_hook

This module contains a Salesforce Hook which allows you to connect to your Salesforce instance, retrieve data from it, and write that data to a file for other uses.

Note: this hook also relies on the simple_salesforce package: https://github.com/simple-salesforce/simple-salesforce

Module Contents

class airflow.contrib.hooks.salesforce_hook.**SalesforceHook**(*conn_id*)

Bases: *airflow.hooks.base_hook.BaseHook*

get_conn(*self*)

Sign into Salesforce, only if we are not already signed in.

make_query(*self*, *query*)

Make a query to Salesforce.

Parameters **query** (*str*) – The query to make to Salesforce.

Returns The query result.

Return type dict

describe_object(*self*, *obj*)

Get the description of an object from Salesforce. This description is the object's schema and some extra metadata that Salesforce stores for each object.

Parameters **obj** (*str*) – The name of the Salesforce object that we are getting a description of.

Returns the description of the Salesforce object.

Return type dict

get_available_fields(*self*, *obj*)

Get a list of all available fields for an object.

Parameters **obj** (*str*) – The name of the Salesforce object that we are getting a description of.

Returns the names of the fields.

Return type list of str

get_object_from_salesforce(*self*, *obj*, *fields*)

Get all instances of the *object* from Salesforce. For each model, only get the fields specified in fields.

All we really do underneath the hood is run: SELECT <fields> FROM <obj>;

Parameters

- **obj** (*str*) – The object name to get from Salesforce.
- **fields** (*iterable*) – The fields to get from the object.

Returns all instances of the object from Salesforce.

Return type dict

classmethod _to_timestamp(*cls*, *column*)
Convert a column of a dataframe to UNIX timestamps if applicable

Parameters **column** (*pd.Series*) – A Series object representing a column of a dataframe.

Returns a new series that maintains the same index as the original

Return type pd.Series

write_object_to_file(*self*, *query_results*, *filename*, *fmt='csv'*, *coerce_to_timestamp=False*, *record_time_added=False*)
Write query results to file.

Acceptable formats are:

- **csv:** comma-separated-values file. This is the default format.
- **json:** JSON array. Each element in the array is a different row.
- **ndjson:** JSON array but each element is new-line delimited instead of comma delimited like in *json*

This requires a significant amount of cleanup. Pandas doesn't handle output to CSV and json in a uniform way. This is especially painful for datetime types. Pandas wants to write them as strings in CSV, but as millisecond Unix timestamps.

By default, this function will try and leave all values as they are represented in Salesforce. You use the *coerce_to_timestamp* flag to force all datetimes to become Unix timestamps (UTC). This is can be greatly beneficial as it will make all of your datetime fields look the same, and makes it easier to work with in other database environments

Parameters

- **query_results** (*list of dict*) – the results from a SQL query
- **filename** (*str*) – the name of the file where the data should be dumped to
- **fmt** (*str*) – the format you want the output in. Default: 'csv'
- **coerce_to_timestamp** (*bool*) – True if you want all datetime fields to be converted into Unix timestamps. False if you want them to be left in the same format as they were in Salesforce. Leaving the value as False will result in datetimes being strings. Default: False
- **record_time_added** (*bool*) – True if you want to add a Unix timestamp field to the resulting data that marks when the data was fetched from Salesforce. Default: False

Returns the dataframe that gets written to the file.

Return type pd.Dataframe

airflow.contrib.hooks.segment_hook

This module contains a Segment Hook which allows you to connect to your Segment account, retrieve data from it or write to that file.

NOTE: this hook also relies on the Segment analytics package: https://github.com/segmentio/analytics-python

Module Contents

class airflow.contrib.hooks.segment_hook.**SegmentHook**(*segment_conn_id='segment_default'*, *segment_debug_mode=False*, **args*, ***kwargs*)

Bases: *airflow.hooks.base_hook.BaseHook*

get_conn(*self*)

on_error(*self*, *error*, *items*)
Handles error callbacks when using Segment with segment_debug_mode set to True

airflow.contrib.hooks.sftp_hook

Module Contents

class airflow.contrib.hooks.sftp_hook.**SFTPHook**(*ftp_conn_id='sftp_default'*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.ssh_hook.SSHHook*

This hook is inherited from SSH hook. Please refer to SSH hook for the input arguments.

Interact with SFTP. Aims to be interchangeable with FTPHook.

:Pitfalls:

```
- In contrast with FTPHook describe_directory only returns size, type and
  modify. It doesn't return unix.owner, unix.mode, perm, unix.group and
  unique.
- retrieve_file and store_file only take a local full path and not a
   buffer.
- If no mode is passed to create_directory it will be created with 777
  permissions.
```

Errors that may occur throughout but should be handled downstream.

get_conn(*self*)
Returns an SFTP connection object

close_conn(*self*)
Closes the connection. An error will occur if the connection wasnt ever opened.

describe_directory(*self*, *path*)
Returns a dictionary of {filename: {attributes}} for all files on the remote system (where the MLSD command is supported). :param path: full path to the remote directory :type path: str

list_directory(*self*, *path*)
Returns a list of files on the remote system. :param path: full path to the remote directory to list :type path: str

create_directory(*self*, *path*, *mode=777*)
Creates a directory on the remote system. :param path: full path to the remote directory to create :type path: str :param mode: int representation of octal mode for directory

delete_directory(*self*, *path*)
Deletes a directory on the remote system. :param path: full path to the remote directory to delete :type path: str

retrieve_file(*self*, *remote_full_path*, *local_full_path*)
Transfers the remote file to a local location. If local_full_path is a string path, the file will be put at that location

:param remote_full_path: full path to the remote file :type remote_full_path: str :param local_full_path: full path to the local file :type local_full_path: str

store_file (*self*, *remote_full_path*, *local_full_path*)
Transfers a local file to the remote location. If local_full_path_or_buffer is a string path, the file will be read from that location :param remote_full_path: full path to the remote file :type remote_full_path: str :param local_full_path: full path to the local file :type local_full_path: str

delete_file (*self*, *path*)
Removes a file on the FTP Server :param path: full path to the remote file :type path: str

get_mod_time (*self*, *path*)

airflow.contrib.hooks.slack_webhook_hook

Module Contents

class airflow.contrib.hooks.slack_webhook_hook.**SlackWebhookHook** (*http_conn_id=None*, *webhook_token=None*, *message=”*, *attachments=None*, *channel=None*, *username=None*, *icon_emoji=None*, *link_names=False*, *proxy=None*, **args*, ***kwargs*)

Bases: *airflow.hooks.http_hook.HttpHook*

This hook allows you to post messages to Slack using incoming webhooks. Takes both Slack webhook token directly and connection that has Slack webhook token. If both supplied, http_conn_id will be used as base_url, and webhook_token will be taken as endpoint, the relative path of the url.

Each Slack webhook token can be pre-configured to use a specific channel, username and icon. You can override these defaults in this hook.

Parameters

- **http_conn_id** (*str*) – connection that has Slack webhook token in the extra field
- **webhook_token** (*str*) – Slack webhook token
- **message** (*str*) – The message you want to send on Slack
- **attachments** (*list*) – The attachments to send on Slack. Should be a list of dictionaries representing Slack attachments.
- **channel** (*str*) – The channel the message should be posted to
- **username** (*str*) – The username to post to slack with
- **icon_emoji** (*str*) – The emoji to use as icon for the user posting to Slack
- **link_names** (*bool*) – Whether or not to find and link channel and usernames in your message
- **proxy** (*str*) – Proxy to use to make the Slack webhook call

_get_token (*self*, *token*, *http_conn_id*)
Given either a manually set token or a conn_id, return the webhook_token to use :param token: The manually provided token :type token: str :param http_conn_id: The conn_id provided :type http_conn_id: str :return: webhook_token (str) to use

_build_slack_message (*self*)
Construct the Slack message. All relevant parameters are combined here to a valid Slack json message :return: Slack message (str) to send

execute (*self*)
Remote Popen (actually execute the slack webhook call)

airflow.contrib.hooks.snowflake_hook

Module Contents

class airflow.contrib.hooks.snowflake_hook.**SnowflakeHook** (**args*, ***kwargs*)
Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with Snowflake.

get_sqlalchemy_engine() depends on snowflake-sqlalchemy

conn_name_attr = snowflake_conn_id

default_conn_name = snowflake_default

supports_autocommit = True

_get_conn_params (*self*)
one method to fetch connection params as a dict used in get_uri() and get_connection()

get_uri (*self*)
override DbApiHook get_uri method for get_sqlalchemy_engine()

get_conn (*self*)
Returns a snowflake.connection object

_get_aws_credentials (*self*)
returns aws_access_key_id, aws_secret_access_key from extra

intended to be used by external import and export statements

set_autocommit (*self*, *conn*, *autocommit*)

airflow.contrib.hooks.spark_jdbc_hook

Module Contents

class airflow.contrib.hooks.spark_jdbc_hook.**SparkJDBCHook**(*spark_app_name='airflow-spark-jdbc', spark_conn_id='spark-default', spark_conf=None, spark_py_files=None, spark_files=None, spark_jars=None, num_executors=None, executor_cores=None, executor_memory=None, driver_memory=None, verbose=False, principal=None, keytab=None, cmd_type='spark_to_jdbc', jdbc_table=None, jdbc_conn_id='jdbc-default', jdbc_driver=None, metastore_table=None, jdbc_truncate=False, save_mode=None, save_format=None, batch_size=None, fetch_size=None, num_partitions=None, partition_column=None, lower_bound=None, upper_bound=None, create_table_column_types=None, *args, **kwargs*)

Bases: `airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook`

This hook extends the SparkSubmitHook specifically for performing data transfers to/from JDBC-based databases with Apache Spark.

Parameters

- **spark_app_name** (*str*) – Name of the job (default airflow-spark-jdbc)
- **spark_conn_id** (*str*) – Connection id as configured in Airflow administration
- **spark_conf** (*dict*) – Any additional Spark configuration properties
- **spark_py_files** (*str*) – Additional python files used (.zip, .egg, or .py)
- **spark_files** (*str*) – Additional files to upload to the container running the job
- **spark_jars** (*str*) – Additional jars to upload and add to the driver and executor classpath
- **num_executors** (*int*) – number of executor to run. This should be set so as to manage the number of connections made with the JDBC database

- **executor_cores** (*int*) – Number of cores per executor
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G)
- **driver_memory** (*str*) – Memory allocated to the driver (e.g. 1000M, 2G)
- **verbose** (*bool*) – Whether to pass the verbose flag to spark-submit for debugging
- **keytab** (*str*) – Full path to the file that contains the keytab
- **principal** (*str*) – The name of the kerberos principal used for keytab
- **cmd_type** (*str*) – Which way the data should flow. 2 possible values: spark_to_jdbc: data written by spark from metastore to jdbc jdbc_to_spark: data written by spark from jdbc to metastore
- **jdbc_table** (*str*) – The name of the JDBC table
- **jdbc_conn_id** (*str*) – Connection id used for connection to JDBC database
- **jdbc_driver** (*str*) – Name of the JDBC driver to use for the JDBC connection. This driver (usually a jar) should be passed in the 'jars' parameter
- **metastore_table** (*str*) – The name of the metastore table,
- **jdbc_truncate** (*bool*) – (spark_to_jdbc only) Whether or not Spark should truncate or drop and recreate the JDBC table. This only takes effect if 'save_mode' is set to Overwrite. Also, if the schema is different, Spark cannot truncate, and will drop and recreate
- **save_mode** (*str*) – The Spark save-mode to use (e.g. overwrite, append, etc.)
- **save_format** (*str*) – (jdbc_to_spark-only) The Spark save-format to use (e.g. parquet)
- **batch_size** (*int*) – (spark_to_jdbc only) The size of the batch to insert per round trip to the JDBC database. Defaults to 1000
- **fetch_size** (*int*) – (jdbc_to_spark only) The size of the batch to fetch per round trip from the JDBC database. Default depends on the JDBC driver
- **num_partitions** (*int*) – The maximum number of partitions that can be used by Spark simultaneously, both for spark_to_jdbc and jdbc_to_spark operations. This will also cap the number of JDBC connections that can be opened
- **partition_column** (*str*) – (jdbc_to_spark-only) A numeric column to be used to partition the metastore table by. If specified, you must also specify: num_partitions, lower_bound, upper_bound
- **lower_bound** (*int*) – (jdbc_to_spark-only) Lower bound of the range of the numeric partition column to fetch. If specified, you must also specify: num_partitions, partition_column, upper_bound
- **upper_bound** (*int*) – (jdbc_to_spark-only) Upper bound of the range of the numeric partition column to fetch. If specified, you must also specify: num_partitions, partition_column, lower_bound
- **create_table_column_types** – (spark_to_jdbc-only) The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g: "name CHAR(64), comments VARCHAR(1024)"). The specified types should be valid spark sql data types.

_resolve_jdbc_connection(*self*)

_build_jdbc_application_arguments(*self*, *jdbc_conn*)

submit_jdbc_job(*self*)

get_conn(*self*)

`airflow.contrib.hooks.spark_jdbc_script`

Module Contents

`airflow.contrib.hooks.spark_jdbc_script.`**`set_common_options`**(*spark_source, url='localhost:5432', jdbc_table='default.default', user='root', password='root', driver='driver'*)

`airflow.contrib.hooks.spark_jdbc_script.`**`spark_write_to_jdbc`**(*spark, url, user, password, metastore_table, jdbc_table, driver, truncate, save_mode, batch_size, num_partitions, create_table_column_types*)

`airflow.contrib.hooks.spark_jdbc_script.`**`spark_read_from_jdbc`**(*spark, url, user, password, metastore_table, jdbc_table, driver, save_mode, save_format, fetch_size, num_partitions, partition_column, lower_bound, upper_bound*)

`airflow.contrib.hooks.spark_jdbc_script.`**`parser`**

`airflow.contrib.hooks.spark_sql_hook`

Module Contents

`class` `airflow.contrib.hooks.spark_sql_hook.`**`SparkSqlHook`**(*sql, conf=None, conn_id='spark_sql_default', total_executor_cores=None, executor_cores=None, executor_memory=None, keytab=None, principal=None, master='yarn', name='default-name', num_executors=None, verbose=True, yarn_queue='default'*)

Bases: *`airflow.hooks.base_hook.BaseHook`*

This hook is a wrapper around the spark-sql binary. It requires that the "spark-sql" binary is in the PATH.

Parameters

- **`sql`** (*`str`*) – The SQL query to execute

- **conf** (*str (format: PROP=VALUE)*) – arbitrary Spark configuration property
- **conn_id** (*str*) – connection_id string
- **total_executor_cores** (*int*) – (Standalone & Mesos only) Total cores for all executors (Default: all the available cores on the worker)
- **executor_cores** (*int*) – (Standalone & YARN only) Number of cores per executor (Default: 2)
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G) (Default: 1G)
- **keytab** (*str*) – Full path to the file that contains the keytab
- **master** (*str*) – spark://host:port, mesos://host:port, yarn, or local
- **name** (*str*) – Name of the job.
- **num_executors** (*int*) – Number of executors to launch
- **verbose** (*bool*) – Whether to pass the verbose flag to spark-sql
- **yarn_queue** (*str*) – The YARN queue to submit to (Default: "default")

get_conn(*self*)

_prepare_command(*self*, *cmd*)
Construct the spark-sql command to execute. Verbose output is enabled as default.

Parameters **cmd** (*str*) – command to append to the spark-sql command

Returns full command to be executed

run_query(*self*, *cmd=''*, ***kwargs*)
Remote Popen (actually execute the Spark-sql query)

Parameters

- **cmd** – command to remotely execute
- **kwargs** – extra arguments to Popen (see subprocess.Popen)

kill(*self*)

airflow.contrib.hooks.spark_submit_hook

Module Contents

class airflow.contrib.hooks.spark_submit_hook.**SparkSubmitHook**(*conf=None, conn_id='spark_default', files=None, py_files=None, archives=None, driver_class_path=None, jars=None, java_class=None, packages=None, exclude_packages=None, repositories=None, total_executor_cores=None, executor_cores=None, executor_memory=None, driver_memory=None, keytab=None, principal=None, name='default-name', num_executors=None, application_args=None, env_vars=None, verbose=False, spark_binary='spark-submit'*)

Bases: *airflow.hooks.base_hook.BaseHook*, airflow.utils.log.logging_mixin.LoggingMixin

This hook is a wrapper around the spark-submit binary to kick off a spark-submit job. It requires that the "spark-submit" binary is in the PATH or the spark_home to be supplied.

Parameters

- **conf** (*dict*) – Arbitrary Spark configuration properties
- **conn_id** (*str*) – The connection id as configured in Airflow administration. When an invalid connection_id is supplied, it will default to yarn.
- **files** (*str*) – Upload additional files to the executor running the job, separated by a comma. Files will be placed in the working directory of each executor. For example, serialized objects.
- **py_files** (*str*) – Additional python files used by the job, can be .zip, .egg or .py.
- **driver_class_path** (*str*) – Additional, driver-specific, classpath settings.
- **jars** (*str*) – Submit additional jars to upload and place them in executor classpath.
- **java_class** (*str*) – the main class of the Java application

- **packages** (*str*) – Comma-separated list of maven coordinates of jars to include on the driver and executor classpaths
- **exclude_packages** (*str*) – Comma-separated list of maven coordinates of jars to exclude while resolving the dependencies provided in 'packages'
- **repositories** (*str*) – Comma-separated list of additional remote repositories to search for the maven coordinates given with 'packages'
- **total_executor_cores** (*int*) – (Standalone & Mesos only) Total cores for all executors (Default: all the available cores on the worker)
- **executor_cores** (*int*) – (Standalone, YARN and Kubernetes only) Number of cores per executor (Default: 2)
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G) (Default: 1G)
- **driver_memory** (*str*) – Memory allocated to the driver (e.g. 1000M, 2G) (Default: 1G)
- **keytab** (*str*) – Full path to the file that contains the keytab
- **principal** (*str*) – The name of the kerberos principal used for keytab
- **name** (*str*) – Name of the job (default airflow-spark)
- **num_executors** (*int*) – Number of executors to launch
- **application_args** (*list*) – Arguments for the application being submitted
- **env_vars** (*dict*) – Environment variables for spark-submit. It supports yarn and k8s mode too.
- **verbose** (*bool*) – Whether to pass the verbose flag to spark-submit process for debugging
- **spark_binary** (*str*) – The command to use for spark submit. Some distros may use spark2-submit.

Param archives: Archives that spark should unzip (and possibly tag with #ALIAS) into the application working directory.

_resolve_should_track_driver_status (*self*)
Determines whether or not this hook should poll the spark driver status through subsequent spark-submit status requests after the initial spark-submit request :return: if the driver status should be tracked

_resolve_connection (*self*)

get_conn (*self*)

_get_spark_binary_path (*self*)

_build_spark_submit_command (*self*, *application*)
Construct the spark-submit command to execute. :param application: command to append to the spark-submit command :type application: str :return: full command to be executed

_build_track_driver_status_command (*self*)
Construct the command to poll the driver status.

Returns full command to be executed

submit (*self*, *application=''*, ***kwargs*)
Remote Popen to execute the spark-submit job

Parameters

- **application** (*str*) – Submitted application, jar or py file
- **kwargs** – extra arguments to Popen (see subprocess.Popen)

`_process_spark_submit_log`(*self*, *itr*)
Processes the log files and extracts useful information out of it.

If the deploy-mode is 'client', log the output of the submit command as those are the output logs of the Spark worker directly.

Remark: If the driver needs to be tracked for its status, the log-level of the spark deploy needs to be at least INFO (log4j.logger.org.apache.spark.deploy=INFO)

Parameters **`itr`** – An iterator which iterates over the input of the subprocess

`_process_spark_status_log`(*self*, *itr*)
parses the logs of the spark driver status query process

Parameters **`itr`** – An iterator which iterates over the input of the subprocess

`_start_driver_status_tracking`(*self*)
Polls the driver based on self._driver_id to get the status. Finish successfully when the status is FINISHED. Finish failed when the status is ERROR/UNKNOWN/KILLED/FAILED.

Possible status:

SUBMITTED Submitted but not yet scheduled on a worker

RUNNING Has been allocated to a worker to run

FINISHED Previously ran and exited cleanly

RELAUNCHING Exited non-zero or due to worker failure, but has not yet started running again

UNKNOWN The status of the driver is temporarily not known due to master failure recovery

KILLED A user manually killed this driver

FAILED The driver exited non-zero and was not supervised

ERROR Unable to run or restart due to an unrecoverable error (e.g. missing jar file)

`_build_spark_driver_kill_command`(*self*)
Construct the spark-submit command to kill a driver. :return: full command to kill a driver

`on_kill`(*self*)

`airflow.contrib.hooks.sqoop_hook`

This module contains a sqoop 1.x hook

Module Contents

`class` `airflow.contrib.hooks.sqoop_hook.`**`SqoopHook`**(*conn_id='sqoop_default', verbose=False, num_mappers=None, hcatalog_database=None, hcatalog_table=None, properties=None*)

Bases: *`airflow.hooks.base_hook.BaseHook`*

This hook is a wrapper around the sqoop 1 binary. To be able to use the hook it is required that "sqoop" is in the PATH.

Additional arguments that can be passed via the 'extra' JSON field of the sqoop connection:

- `job_tracker`: Job tracker local|jobtracker:port.
- `namenode`: Namenode.
- `lib_jars`: Comma separated jar files to include in the classpath.

- `files`: Comma separated files to be copied to the map reduce cluster.
- **`archives`: Comma separated archives to be unarchived on the compute** machines.
- `password_file`: Path to file containing the password.

Parameters

- **`conn_id`** (*`str`*) – Reference to the sqoop connection.
- **`verbose`** (*`bool`*) – Set sqoop to verbose.
- **`num_mappers`** (*`int`*) – Number of map tasks to import in parallel.
- **`properties`** (*`dict`*) – Properties to set via the -D argument

`get_conn`(*self*)

`cmd_mask_password`(*self*, *cmd_orig*)

`Popen`(*self*, *cmd*, ***kwargs*)
Remote Popen

Parameters

- **`cmd`** – command to remotely execute
- **`kwargs`** – extra arguments to Popen (see subprocess.Popen)

Returns handle to subprocess

`_prepare_command`(*self*, *export=False*)

`static _get_export_format_argument`(*file_type='text'*)

`_import_cmd`(*self*, *target_dir*, *append*, *file_type*, *split_by*, *direct*, *driver*, *extra_import_options*)

`import_table`(*self*, *table*, *target_dir=None*, *append=False*, *file_type='text'*, *columns=None*, *split_by=None*, *where=None*, *direct=False*, *driver=None*, *extra_import_options=None*)
Imports table from remote location to target dir. Arguments are copies of direct sqoop command line arguments

Parameters

- **`table`** – Table to read
- **`target_dir`** – HDFS destination dir
- **`append`** – Append data to an existing dataset in HDFS
- **`file_type`** – “avro”, “sequence”, “text” or “parquet”. Imports data to into the specified format. Defaults to text.
- **`columns`** – <col,col,col…> Columns to import from table
- **`split_by`** – Column of the table used to split work units
- **`where`** – WHERE clause to use during import
- **`direct`** – Use direct connector if exists for the database
- **`driver`** – Manually specify JDBC driver class to use
- **`extra_import_options`** – Extra import options to pass as dict. If a key doesn’t have a value, just pass an empty string to it. Don’t include prefix of – for sqoop options.

`import_query`(*self*, *query*, *target_dir*, *append=False*, *file_type='text'*, *split_by=None*, *direct=None*, *driver=None*, *extra_import_options=None*)
Imports a specific query from the rdbms to hdfs

Parameters

- **query** – Free format query to run
- **target_dir** – HDFS destination dir
- **append** – Append data to an existing dataset in HDFS
- **file_type** – "avro", "sequence", "text" or "parquet" Imports data to hdfs into the specified format. Defaults to text.
- **split_by** – Column of the table used to split work units
- **direct** – Use direct import fast path
- **driver** – Manually specify JDBC driver class to use
- **extra_import_options** – Extra import options to pass as dict. If a key doesn't have a value, just pass an empty string to it. Don't include prefix of – for sqoop options.

_export_cmd(*self*, *table*, *export_dir*, *input_null_string*, *input_null_non_string*, *staging_table*, *clear_staging_table*, *enclosed_by*, *escaped_by*, *input_fields_terminated_by*, *input_lines_terminated_by*, *input_optionally_enclosed_by*, *batch*, *relaxed_isolation*, *extra_export_options*)

export_table(*self*, *table*, *export_dir*, *input_null_string*, *input_null_non_string*, *staging_table*, *clear_staging_table*, *enclosed_by*, *escaped_by*, *input_fields_terminated_by*, *input_lines_terminated_by*, *input_optionally_enclosed_by*, *batch*, *relaxed_isolation*, *extra_export_options=None*)

Exports Hive table to remote location. Arguments are copies of direct sqoop command line Arguments

Parameters

- **table** – Table remote destination
- **export_dir** – Hive table to export
- **input_null_string** – The string to be interpreted as null for string columns
- **input_null_non_string** – The string to be interpreted as null for non-string columns
- **staging_table** – The table in which data will be staged before being inserted into the destination table
- **clear_staging_table** – Indicate that any data present in the staging table can be deleted
- **enclosed_by** – Sets a required field enclosing character
- **escaped_by** – Sets the escape character
- **input_fields_terminated_by** – Sets the field separator character
- **input_lines_terminated_by** – Sets the end-of-line character
- **input_optionally_enclosed_by** – Sets a field enclosing character
- **batch** – Use batch mode for underlying statement execution
- **relaxed_isolation** – Transaction isolation to read uncommitted for the mappers
- **extra_export_options** – Extra export options to pass as dict. If a key doesn't have a value, just pass an empty string to it. Don't include prefix of – for sqoop options.

`airflow.contrib.hooks.ssh_hook`

Module Contents

class `airflow.contrib.hooks.ssh_hook.`**`SSHHook`**(*ssh_conn_id=None*, *remote_host=None*, *username=None*, *password=None*, *key_file=None*, *port=None*, *timeout=10*, *keepalive_interval=30*)

Bases: *`airflow.hooks.base_hook.BaseHook`*

Hook for ssh remote execution using Paramiko. ref: https://github.com/paramiko/paramiko This hook also lets you create ssh tunnel and serve as basis for SFTP file transfer

Parameters

- **ssh_conn_id** (*str*) – connection id from airflow Connections from where all the required parameters can be fetched like username, password or key_file. Thought the priority is given to the param passed during init
- **remote_host** (*str*) – remote host to connect
- **username** (*str*) – username to connect to the remote_host
- **password** (*str*) – password of the username to connect to the remote_host
- **key_file** (*str*) – key file to use to connect to the remote_host.
- **port** (*int*) – port of remote host to connect (Default is paramiko SSH_PORT)
- **timeout** (*int*) – timeout for the attempt to connect to the remote_host.
- **keepalive_interval** (*int*) – send a keepalive packet to remote host every keepalive_interval seconds

get_conn(*self*)

Opens a ssh connection to the remote host.

Return type paramiko.client.SSHClient

__enter__(*self*)

__exit__(*self*, *exc_type*, *exc_val*, *exc_tb*)

get_tunnel(*self*, *remote_port*, *remote_host='localhost'*, *local_port=None*)

Creates a tunnel between two hosts. Like ssh -L <LOCAL_PORT>:host:<REMOTE_PORT>.

Parameters

- **remote_port** (*int*) – The remote port to create a tunnel to
- **remote_host** (*str*) – The remote host to create a tunnel to (default localhost)
- **local_port** (*int*) – The local port to attach the tunnel to

Returns sshtunnel.SSHTunnelForwarder object

create_tunnel(*self*, *local_port*, *remote_port=None*, *remote_host='localhost'*)

`airflow.contrib.hooks.vertica_hook`

Module Contents

class `airflow.contrib.hooks.vertica_hook.`**`VerticaHook`**

Bases: *`airflow.hooks.dbapi_hook.DbApiHook`*

Interact with Vertica.

conn_name_attr = vertica_conn_id

default_conn_name = vertica_default

supports_autocommit = True

get_conn (*self*)
Returns verticaql connection object

airflow.contrib.hooks.wasb_hook

Module Contents

class airflow.contrib.hooks.wasb_hook.**WasbHook** (*wasb_conn_id='wasb_default'*)
Bases: *airflow.hooks.base_hook.BaseHook*

Interacts with Azure Blob Storage through the wasb:// protocol.

Additional options passed in the 'extra' field of the connection will be passed to the *BlockBlockService()* constructor. For example, authenticate using a SAS token by adding {"sas_token": "YOUR_TOKEN"}.

Parameters **wasb_conn_id** (*str*) – Reference to the wasb connection.

get_conn (*self*)
Return the BlockBlobService object.

check_for_blob (*self*, *container_name*, *blob_name*, ***kwargs*)
Check if a blob exists on Azure Blob Storage.

Parameters

- **container_name** (*str*) – Name of the container.
- **blob_name** (*str*) – Name of the blob.
- **kwargs** (*object*) – Optional keyword arguments that *BlockBlobService.exists()* takes.

Returns True if the blob exists, False otherwise.

Return type bool

check_for_prefix (*self*, *container_name*, *prefix*, ***kwargs*)
Check if a prefix exists on Azure Blob storage.

Parameters

- **container_name** (*str*) – Name of the container.
- **prefix** (*str*) – Prefix of the blob.
- **kwargs** (*object*) – Optional keyword arguments that *BlockBlobService.list_blobs()* takes.

Returns True if blobs matching the prefix exist, False otherwise.

Return type bool

load_file (*self*, *file_path*, *container_name*, *blob_name*, ***kwargs*)
Upload a file to Azure Blob Storage.

Parameters

- **file_path** (*str*) – Path to the file to load.
- **container_name** (*str*) – Name of the container.
- **blob_name** (*str*) – Name of the blob.

- **kwargs** (*object*) – Optional keyword arguments that *BlockBlobService.create_blob_from_path()* takes.

load_string (*self*, *string_data*, *container_name*, *blob_name*, ***kwargs*)
Upload a string to Azure Blob Storage.

Parameters

- **string_data** (*str*) – String to load.
- **container_name** (*str*) – Name of the container.
- **blob_name** (*str*) – Name of the blob.
- **kwargs** (*object*) – Optional keyword arguments that *BlockBlobService.create_blob_from_text()* takes.

get_file (*self*, *file_path*, *container_name*, *blob_name*, ***kwargs*)
Download a file from Azure Blob Storage.

Parameters

- **file_path** (*str*) – Path to the file to download.
- **container_name** (*str*) – Name of the container.
- **blob_name** (*str*) – Name of the blob.
- **kwargs** (*object*) – Optional keyword arguments that *BlockBlobService.create_blob_from_path()* takes.

read_file (*self*, *container_name*, *blob_name*, ***kwargs*)
Read a file from Azure Blob Storage and return as a string.

Parameters

- **container_name** (*str*) – Name of the container.
- **blob_name** (*str*) – Name of the blob.
- **kwargs** (*object*) – Optional keyword arguments that *BlockBlobService.create_blob_from_path()* takes.

delete_file (*self*, *container_name*, *blob_name*, *is_prefix=False*, *ignore_if_missing=False*, ***kwargs*)
Delete a file from Azure Blob Storage.

Parameters

- **container_name** (*str*) – Name of the container.
- **blob_name** (*str*) – Name of the blob.
- **is_prefix** (*bool*) – If blob_name is a prefix, delete all matching files
- **ignore_if_missing** (*bool*) – if True, then return success even if the blob does not exist.
- **kwargs** (*object*) – Optional keyword arguments that *BlockBlobService.create_blob_from_path()* takes.

airflow.contrib.hooks.winrm_hook

Module Contents

class airflow.contrib.hooks.winrm_hook.**WinRMHook**(*ssh_conn_id=None, endpoint=None, remote_host=None, remote_port=5985, transport='plaintext', username=None, password=None, service='HTTP', keytab=None, ca_trust_path=None, cert_pem=None, cert_key_pem=None, server_cert_validation='validate', kerberos_delegation=False, read_timeout_sec=30, operation_timeout_sec=20, kerberos_hostname_override=None, message_encryption='auto', credssp_disable_tlsv1_2=False, send_cbt=True*)

Bases: *airflow.hooks.base_hook.BaseHook*

Hook for winrm remote execution using pywinrm.

Seealso https://github.com/diyan/pywinrm/blob/master/winrm/protocol.py

Parameters

- **ssh_conn_id** (*str*) – connection id from airflow Connections from where all the required parameters can be fetched like username and password. Thought the priority is given to the param passed during init
- **endpoint** (*str*) – When not set, endpoint will be constructed like this: 'http://{remote_host}:{remote_port}/wsman'
- **remote_host** (*str*) – Remote host to connect to. Ignored if *endpoint* is set.
- **remote_port** (*int*) – Remote port to connect to. Ignored if *endpoint* is set.
- **transport** (*str*) – transport type, one of 'plaintext' (default), 'kerberos', 'ssl', 'ntlm', 'credssp'
- **username** (*str*) – username to connect to the remote_host
- **password** (*str*) – password of the username to connect to the remote_host
- **service** (*str*) – the service name, default is HTTP
- **keytab** (*str*) – the path to a keytab file if you are using one
- **ca_trust_path** (*str*) – Certification Authority trust path
- **cert_pem** (*str*) – client authentication certificate file path in PEM format
- **cert_key_pem** (*str*) – client authentication certificate key file path in PEM format
- **server_cert_validation** (*str*) – whether server certificate should be validated on Python versions that support it; one of 'validate' (default), 'ignore'
- **kerberos_delegation** (*bool*) – if True, TGT is sent to target server to allow multiple hops
- **read_timeout_sec** (*int*) – maximum seconds to wait before an HTTP connect/read times out (default 30). This value should be slightly higher than operation_timeout_sec, as the server can block *at least* that long.

- **operation_timeout_sec** (*int*) – maximum allowed time in seconds for any single wsman HTTP operation (default 20). Note that operation timeouts while receiving output (the only wsman operation that should take any significant time, and where these timeouts are expected) will be silently retried indefinitely.
- **kerberos_hostname_override** (*str*) – the hostname to use for the kerberos exchange (defaults to the hostname in the endpoint URL)
- **message_encryption** (*str*) – Will encrypt the WinRM messages if set and the transport auth supports message encryption. (Default 'auto')
- **credssp_disable_tlsv1_2** (*bool*) – Whether to disable TLSv1.2 support and work with older protocols like TLSv1.0, default is False
- **send_cbt** (*bool*) – Will send the channel bindings over a HTTPS channel (Default: True)

get_conn (*self*)

3.22.3 Executors

Executors are the mechanism by which task instances get run. All executors are derived from `BaseExecutor`.

3.22.3.1 Executors packages

All executors are in the following packages:

airflow.executors

Submodules

airflow.executors.base_executor

Module Contents

airflow.executors.base_executor.**PARALLELISM**

class airflow.executors.base_executor.**BaseExecutor** (*parallelism=PARALLELISM*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

start (*self*)

Executors may need to get things started. For example LocalExecutor starts N workers.

queue_command (*self*, *simple_task_instance*, *command*, *priority=1*, *queue=None*)

queue_task_instance (*self*, *task_instance*, *mark_success=False*, *pickle_id=None*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *pool=None*, *cfg_path=None*)

has_task (*self*, *task_instance*)

Checks if a task is either queued or running in this executor

Parameters **task_instance** – TaskInstance

Returns True if the task is known to this executor

sync (*self*)

Sync will get called periodically by the heartbeat method. Executors should override this to perform gather statuses.

heartbeat (*self*)

trigger_tasks (*self*, *open_slots*)
Trigger tasks

Parameters **open_slots** – Number of open slots

Returns

change_state (*self*, *key*, *state*)

fail (*self*, *key*)

success (*self*, *key*)

get_event_buffer (*self*, *dag_ids=None*)
Returns and flush the event buffer. In case dag_ids is specified it will only return and flush events for the given dag_ids. Otherwise it returns and flushes all

Parameters **dag_ids** – to dag_ids to return events for, if None returns all

Returns a dict of events

execute_async (*self*, *key*, *command*, *queue=None*, *executor_config=None*)
This method will execute the command asynchronously.

end (*self*)
This method is called when the caller is done submitting job and wants to wait synchronously for the job submitted previously to be all done.

terminate (*self*)
This method is called when the daemon receives a SIGTERM

airflow.executors.celery_executor

Module Contents

airflow.executors.celery_executor.**CELERY_FETCH_ERR_MSG_HEADER = Error fetching Celery task**

airflow.executors.celery_executor.**CELERY_SEND_ERR_MSG_HEADER = Error sending Celery task**
To start the celery worker, run the command: airflow worker

airflow.executors.celery_executor.**celery_configuration**

airflow.executors.celery_executor.**app**

airflow.executors.celery_executor.**execute_command** (*command_to_exec*)

class airflow.executors.celery_executor.**ExceptionWithTraceback** (*exception*, *exception_traceback*)
Wrapper class used to propagate exceptions to parent processes from subprocesses.

Parameters

- **exception** (*Exception*) – The exception to wrap
- **exception_traceback** (*str*) – The stacktrace to wrap

airflow.executors.celery_executor.**fetch_celery_task_state** (*celery_task*)
Fetch and return the state of the given celery task. The scope of this function is global so that it can be called by subprocesses in the pool.

Parameters **celery_task** (*tuple(str, celery.result.AsyncResult)*) – a tuple of the Celery task key and the async Celery object used to fetch the task's state

Returns a tuple of the Celery task key and the Celery state of the task

Return type tuple[str, str]

airflow.executors.celery_executor.**send_task_to_executor**(*task_tuple*)

class airflow.executors.celery_executor.**CeleryExecutor**
Bases: *airflow.executors.base_executor.BaseExecutor*

CeleryExecutor is recommended for production use of Airflow. It allows distributing the execution of task instances to multiple worker nodes.

Celery is a simple, flexible and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

start(*self*)

_num_tasks_per_send_process(*self*, *to_send_count*)
How many Celery tasks should each worker process send.

Returns Number of tasks that should be sent per process

Return type int

_num_tasks_per_fetch_process(*self*)
How many Celery tasks should be sent to each worker process.

Returns Number of tasks that should be used per process

Return type int

trigger_tasks(*self*, *open_slots*)
Overwrite trigger_tasks function from BaseExecutor

Parameters **open_slots** – Number of open slots

Returns

sync(*self*)

end(*self*, *synchronous=False*)

airflow.executors.dask_executor

Module Contents

class airflow.executors.dask_executor.**DaskExecutor**(*cluster_address=None*)
Bases: *airflow.executors.base_executor.BaseExecutor*

DaskExecutor submits tasks to a Dask Distributed cluster.

start(*self*)

execute_async(*self*, *key*, *command*, *queue=None*, *executor_config=None*)

_process_future(*self*, *future*)

sync(*self*)

end(*self*)

terminate(*self*)

airflow.executors.kubernetes_executor

Module Contents

class airflow.executors.kubernetes_executor.**KubernetesExecutorConfig**(*image=None, image_pull_policy=None, request_memory=None, request_cpu=None, limit_memory=None, limit_cpu=None, gcp_service_account_key=None, node_selectors=None, affinity=None, annotations=None, volumes=None, volume_mounts=None, tolerations=None*)

__repr__(*self*)

static from_dict(*obj*)

as_dict(*self*)

class airflow.executors.kubernetes_executor.**KubeConfig**

core_section = core

kubernetes_section = kubernetes

_validate(*self*)

class airflow.executors.kubernetes_executor.**KubernetesJobWatcher**(*namespace, watcher_queue, resource_version, worker_uuid, kube_config*)

Bases: `multiprocessing.Process`, `airflow.utils.log.logging_mixin.LoggingMixin`, `object`

run(*self*)

_run(*self, kube_client, resource_version, worker_uuid, kube_config*)

process_error(*self, event*)

process_status(*self, pod_id, status, labels, resource_version*)

class `airflow.executors.kubernetes_executor.`**`AirflowKubernetesScheduler`**(*kube_config*, *task_queue*, *result_queue*, *kube_client*, *worker_uuid*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

`_make_kube_watcher`(*self*)

`_health_check_kube_watcher`(*self*)

`run_next`(*self*, *next_job*)
The run_next command will check the task_queue for any un-run jobs. It will then create a unique job-id, launch that job in the cluster, and store relevant info in the current_jobs map so we can track the job's status

`delete_pod`(*self*, *pod_id*)

`sync`(*self*)
The sync function checks the status of all currently running kubernetes jobs. If a job is completed, it's status is placed in the result queue to be sent back to the scheduler.

Returns

`process_watcher_task`(*self*, *task*)

static `_strip_unsafe_kubernetes_special_chars`(*string*)
Kubernetes only supports lowercase alphanumeric characters and "-" and "." in the pod name However, there are special rules about how "-" and "." can be used so let's only keep alphanumeric chars see here for detail: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/

Parameters **`string`** – The requested Pod name

Returns `str` Pod name stripped of any unsafe characters

static `_make_safe_pod_id`(*safe_dag_id*, *safe_task_id*, *safe_uuid*)
Kubernetes pod names must be <= 253 chars and must pass the following regex for validation "^[a-z0-9]([-a-z0-9]*[a-z0-9])?(.[a-z0-9]([-a-z0-9]*[a-z0-9])?)*$"

Parameters

- **`safe_dag_id`** – a dag_id with only alphanumeric characters
- **`safe_task_id`** – a task_id with only alphanumeric characters
- **`random_uuid`** – a uuid

Returns `str` valid Pod name of appropriate length

static `_make_safe_label_value`(*string*)
Valid label values must be 63 characters or less and must be empty or begin and end with an alphanumeric character ([a-z0-9A-Z]) with dashes (-), underscores (_), dots (.), and alphanumerics between.

If the label value is then greater than 63 chars once made safe, or differs in any way from the original value sent to this function, then we need to truncate to 53chars, and append it with a unique hash.

static `_create_pod_id`(*dag_id*, *task_id*)

static `_label_safe_datestring_to_datetime`(*string*)
Kubernetes doesn't permit ":" in labels. ISO datetime format uses ":" but not "_", let's replace ":" with "_"

Parameters **`string`** – str

Returns datetime.datetime object

static `_datetime_to_label_safe_datestring`(*datetime_obj*)
Kubernetes doesn't like ":" in labels, since ISO datetime format uses ":" but not "_" let's replace ":" with "_" :param datetime_obj: datetime.datetime object :return: ISO-like string representing the datetime

_labels_to_key (*self*, *labels*)

terminate (*self*)

class airflow.executors.kubernetes_executor.**KubernetesExecutor**
Bases: *airflow.executors.base_executor.BaseExecutor*, airflow.utils.log.logging_mixin.LoggingMixin

clear_not_launched_queued_tasks (*self*, *session=None*)
If the airflow scheduler restarts with pending "Queued" tasks, the tasks may or may not have been launched Thus, on starting up the scheduler let's check every "Queued" task to see if it has been launched (ie: if there is a corresponding pod on kubernetes)

If it has been launched then do nothing, otherwise reset the state to "None" so the task will be rescheduled

This will not be necessary in a future version of airflow in which there is proper support for State.LAUNCHED

_inject_secrets (*self*)

start (*self*)

execute_async (*self*, *key*, *command*, *queue=None*, *executor_config=None*)

sync (*self*)

_change_state (*self*, *key*, *state*, *pod_id*)

end (*self*)

airflow.executors.local_executor

LocalExecutor runs tasks by spawning processes in a controlled fashion in different modes. Given that BaseExecutor has the option to receive a *parallelism* parameter to limit the number of process spawned, when this parameter is *0* the number of processes that LocalExecutor can spawn is unlimited.

The following strategies are implemented: 1. Unlimited Parallelism (self.parallelism == 0): In this strategy, LocalExecutor will spawn a process every time *execute_async* is called, that is, every task submitted to the LocalExecutor will be executed in its own process. Once the task is executed and the result stored in the *result_queue*, the process terminates. There is no need for a *task_queue* in this approach, since as soon as a task is received a new process will be allocated to the task. Processes used in this strategy are of class LocalWorker.

2. Limited Parallelism (self.parallelism > 0): In this strategy, the LocalExecutor spawns the number of processes equal to the value of *self.parallelism* at *start* time, using a *task_queue* to coordinate the ingestion of tasks and the work distribution among the workers, which will take a task as soon as they are ready. During the lifecycle of the LocalExecutor, the worker processes are running waiting for tasks, once the LocalExecutor receives the call to shutdown the executor a poison token is sent to the workers to terminate them. Processes used in this strategy are of class QueuedLocalWorker.

Arguably, *SequentialExecutor* could be thought as a LocalExecutor with limited parallelism of just 1 worker, i.e. *self.parallelism = 1*. This option could lead to the unification of the executor implementations, running locally, into just one *LocalExecutor* with multiple modes.

Module Contents

class airflow.executors.local_executor.**LocalWorker** (*result_queue*)
Bases: multiprocessing.Process, airflow.utils.log.logging_mixin.LoggingMixin

LocalWorker Process implementation to run airflow commands. Executes the given command and puts the result into a result queue when done, terminating execution.

execute_work (*self*, *key*, *command*)
Executes command received and stores result state in queue. :param key: the key to identify the TI :type key: tuple(dag_id, task_id, execution_date) :param command: the command to execute :type command: str

run (*self*)

class airflow.executors.local_executor.**QueuedLocalWorker** (*task_queue*, *result_queue*)

Bases: *airflow.executors.local_executor.LocalWorker*

LocalWorker implementation that is waiting for tasks from a queue and will continue executing commands as they become available in the queue. It will terminate execution once the poison token is found.

run (*self*)

class airflow.executors.local_executor.**LocalExecutor**

Bases: *airflow.executors.base_executor.BaseExecutor*

LocalExecutor executes tasks locally in parallel. It uses the multiprocessing Python library and queues to parallelize the execution of tasks.

class _UnlimitedParallelism (*executor*)

Implements LocalExecutor with unlimited parallelism, starting one process per each command to execute.

start (*self*)

execute_async (*self*, *key*, *command*)

Parameters

- **key** (*tuple(dag_id, task_id, execution_date)*) – the key to identify the TI
- **command** (*str*) – the command to execute

sync (*self*)

end (*self*)

class _LimitedParallelism (*executor*)

Implements LocalExecutor with limited parallelism using a task queue to coordinate work distribution.

start (*self*)

execute_async (*self*, *key*, *command*)

Parameters

- **key** (*tuple(dag_id, task_id, execution_date)*) – the key to identify the TI
- **command** (*str*) – the command to execute

sync (*self*)

end (*self*)

start (*self*)

execute_async (*self*, *key*, *command*, *queue=None*, *executor_config=None*)

sync (*self*)

end (*self*)

airflow.executors.sequential_executor

Module Contents

class airflow.executors.sequential_executor.**SequentialExecutor**

Bases: *airflow.executors.base_executor.BaseExecutor*

This executor will only run one task instance at a time, can be used for debugging. It is also the only executor that can be used with sqlite since sqlite doesn't support multiple connections.

Since we want airflow to work out of the box, it defaults to this SequentialExecutor alongside sqlite as you first install it.

execute_async (*self*, *key*, *command*, *queue=None*, *executor_config=None*)

sync (*self*)

end (*self*)

Package Contents

class airflow.executors.**LoggingMixin** (*context=None*)

Convenience super-class to have a logger configured with the class name

logger

log

_set_context (*self*, *context*)

exception airflow.executors.**AirflowException**

Bases: Exception

Base class for all Airflow's errors. Each custom exception should be derived from this class

status_code = 500

class airflow.executors.**BaseExecutor** (*parallelism=PARALLELISM*)

Bases: airflow.utils.log.logging_mixin.LoggingMixin

start (*self*)

Executors may need to get things started. For example LocalExecutor starts N workers.

queue_command (*self*, *simple_task_instance*, *command*, *priority=1*, *queue=None*)

queue_task_instance (*self*, *task_instance*, *mark_success=False*, *pickle_id=None*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *pool=None*, *cfg_path=None*)

has_task (*self*, *task_instance*)

Checks if a task is either queued or running in this executor

Parameters **task_instance** – TaskInstance

Returns True if the task is known to this executor

sync (*self*)

Sync will get called periodically by the heartbeat method. Executors should override this to perform gather statuses.

heartbeat (*self*)

trigger_tasks (*self*, *open_slots*)

Trigger tasks

Parameters **open_slots** – Number of open slots

Returns

change_state (*self*, *key*, *state*)

fail (*self*, *key*)

success (*self*, *key*)

get_event_buffer (*self*, *dag_ids=None*)

Returns and flush the event buffer. In case dag_ids is specified it will only return and flush events for the given dag_ids. Otherwise it returns and flushes all

Parameters **`dag_ids`** – to dag_ids to return events for, if None returns all

Returns a dict of events

`execute_async` (*self*, *key*, *command*, *queue=None*, *executor_config=None*)
This method will execute the command asynchronously.

`end` (*self*)
This method is called when the caller is done submitting job and wants to wait synchronously for the job submitted previously to be all done.

`terminate` (*self*)
This method is called when the daemon receives a SIGTERM

`class` `airflow.executors.`**`LocalExecutor`**
Bases: *`airflow.executors.base_executor.BaseExecutor`*

LocalExecutor executes tasks locally in parallel. It uses the multiprocessing Python library and queues to parallelize the execution of tasks.

`class _UnlimitedParallelism` (*executor*)
Implements LocalExecutor with unlimited parallelism, starting one process per each command to execute.

`start` (*self*)

`execute_async` (*self*, *key*, *command*)

Parameters

- **`key`** (*`tuple(dag_id, task_id, execution_date)`*) – the key to identify the TI
- **`command`** (*`str`*) – the command to execute

`sync` (*self*)

`end` (*self*)

`class _LimitedParallelism` (*executor*)
Implements LocalExecutor with limited parallelism using a task queue to coordinate work distribution.

`start` (*self*)

`execute_async` (*self*, *key*, *command*)

Parameters

- **`key`** (*`tuple(dag_id, task_id, execution_date)`*) – the key to identify the TI
- **`command`** (*`str`*) – the command to execute

`sync` (*self*)

`end` (*self*)

`start` (*self*)

`execute_async` (*self*, *key*, *command*, *queue=None*, *executor_config=None*)

`sync` (*self*)

`end` (*self*)

`class` `airflow.executors.`**`SequentialExecutor`**
Bases: *`airflow.executors.base_executor.BaseExecutor`*

This executor will only run one task instance at a time, can be used for debugging. It is also the only executor that can be used with sqlite since sqlite doesn't support multiple connections.

Since we want airflow to work out of the box, it defaults to this SequentialExecutor alongside sqlite as you first install it.

`execute_async` (*self*, *key*, *command*, *queue=None*, *executor_config=None*)

sync(*self*)

end(*self*)

airflow.executors.**DEFAULT_EXECUTOR**

airflow.executors.**_integrate_plugins**()
Integrate plugins to the context.

airflow.executors.**get_default_executor**()
Creates a new instance of the configured executor if none exists and returns it

class airflow.executors.**Executors**

LocalExecutor = LocalExecutor

SequentialExecutor = SequentialExecutor

CeleryExecutor = CeleryExecutor

DaskExecutor = DaskExecutor

KubernetesExecutor = KubernetesExecutor

airflow.executors.**_get_executor**(*executor_name*)
Creates a new instance of the named executor.
In case the executor name is not know in airflow,
look for it in the plugins

3.22.4 Models

Models are built on top of the SQLAlchemy ORM Base class, and instances are persisted in the database.

3.22.4.1 airflow.models

Submodules

airflow.models.base

Module Contents

airflow.models.base.**SQL_ALCHEMY_SCHEMA**

airflow.models.base.**metadata**

airflow.models.base.**Base :Any**

airflow.models.base.**ID_LEN = 250**

`airflow.models.baseoperator`

Module Contents

class airflow.models.baseoperator.**BaseOperator**(*task_id:str, owner:str=configuration.conf.get('operators', 'DEFAULT_OWNER'), email:Optional[str]=None, email_on_retry:bool=True, email_on_failure:bool=True, retries:int=0, retry_delay:timedelta=timedelta(seconds=300), retry_exponential_backoff:bool=False, max_retry_delay:Optional[datetime]=None, start_date:Optional[datetime]=None, end_date:Optional[datetime]=None, schedule_interval=None, depends_on_past:bool=False, wait_for_downstream:bool=False, dag:Optional[DAG]=None, params:Optional[Dict]=None, default_args:Optional[Dict]=None, priority_weight:int=1, weight_rule:str=WeightRule.DOWNSTREAM, queue:str=configuration.conf.get('celery', 'default_queue'), pool:Optional[str]=None, sla:Optional[timedelta]=None, execution_timeout:Optional[timedelta]=None, on_failure_callback:Optional[Callable]=None, on_success_callback:Optional[Callable]=None, on_retry_callback:Optional[Callable]=None, trigger_rule:str=TriggerRule.ALL_SUCCESS, resources:Optional[Dict]=None, run_as_user:Optional[str]=None, task_concurrency:Optional[int]=None, executor_config:Optional[Dict]=None, do_xcom_push:bool=True, inlets:Optional[Dict]=None, outlets:Optional[Dict]=None, *args, **kwargs*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

Abstract base class for all operators. Since operators create objects that become nodes in the dag, BaseOperator contains many recursive methods for dag crawling behavior. To derive this class, you are expected to override the constructor as well as the 'execute' method.

Operators derived from this class should perform or trigger certain tasks synchronously (wait for completion). Example of operators could be an operator that runs a Pig job (PigOperator), a sensor operator that waits for a partition to land in Hive (HiveSensorOperator), or one that moves data from Hive to MySQL (Hive2MySqlOperator). Instances of these operators (tasks) target specific operations, running specific scripts, functions or data transfers.

This class is abstract and shouldn't be instantiated. Instantiating a class derived from this one results in the creation of a task object, which ultimately becomes a node in DAG objects. Task dependencies should be set by using the set_upstream and/or set_downstream methods.

Parameters

- **task_id** (*str*) – a unique, meaningful id for the task
- **owner** (*str*) – the owner of the task, using the unix username is recommended
- **retries** (*int*) – the number of retries that should be performed before failing the task
- **retry_delay** (*datetime.timedelta*) – delay between retries
- **retry_exponential_backoff** (*bool*) – allow progressive longer waits between retries by using exponential backoff algorithm on retry delay (delay will be converted into seconds)
- **max_retry_delay** (*datetime.timedelta*) – maximum delay interval between retries
- **start_date** (*datetime.datetime*) – The `start_date` for the task, determines the `execution_date` for the first task instance. The best practice is to have the start_date rounded to your DAG's `schedule_interval`. Daily jobs have their start_date some day at 00:00:00, hourly jobs have their start_date at 00:00 of a specific hour. Note that Airflow simply looks at the latest `execution_date` and adds the `schedule_interval` to determine the next `execution_date`. It is also very important to note that different tasks' dependencies need to line up in time. If task A depends on task B and their start_date are offset in a way that their execution_date don't line up, A's dependencies will never be met. If you are looking to delay a task, for example running a daily task at 2AM, look into the `TimeSensor` and `TimeDeltaSensor`. We advise against using dynamic `start_date` and recommend using fixed ones. Read the FAQ entry about start_date for more information.
- **end_date** (*datetime.datetime*) – if specified, the scheduler won't go beyond this date
- **depends_on_past** (*bool*) – when set to true, task instances will run sequentially while relying on the previous task's schedule to succeed. The task instance for the start_date is allowed to run.
- **wait_for_downstream** (*bool*) – when set to true, an instance of task X will wait for tasks immediately downstream of the previous instance of task X to finish successfully before it runs. This is useful if the different instances of a task X alter the same asset, and this asset is used by tasks downstream of task X. Note that depends_on_past is forced to True wherever wait_for_downstream is used.
- **queue** (*str*) – which queue to target when running this job. Not all executors implement queue management, the CeleryExecutor does support targeting specific queues.
- **dag** (`airflow.models.DAG`) – a reference to the dag the task is attached to (if any)
- **priority_weight** (*int*) – priority weight of this task against other task. This allows the executor to trigger higher priority tasks before others when things get backed up. Set priority_weight as a higher number for more important tasks.
- **weight_rule** (*str*) – weighting method used for the effective total priority weight of the task. Options are: `{ downstream | upstream | absolute }` default is `downstream` When set to `downstream` the effective weight of the task is the aggregate sum of all downstream descendants. As a result, upstream tasks will have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and desire to have all upstream tasks to complete for all runs before each dag can continue processing downstream tasks. When set to `upstream` the effective weight is the aggregate sum of all upstream ancestors. This is the opposite where downtream tasks have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and prefer to have each dag complete before starting upstream tasks of other dags. When set to `absolute`, the effective weight is the exact `priority_weight` specified without additional weighting. You may

want to do this when you know exactly what priority weight each task should have. Additionally, when set to `absolute`, there is bonus effect of significantly speeding up the task creation process as for very large DAGS. Options can be set as string or using the constants defined in the static class `airflow.utils.WeightRule`

- **pool** (*str*) – the slot pool this task should run in, slot pools are a way to limit concurrency for certain tasks
- **sla** (*datetime.timedelta*) – time by which the job is expected to succeed. Note that this represents the `timedelta` after the period is closed. For example if you set an SLA of 1 hour, the scheduler would send an email soon after 1:00AM on the `2016-01-02` if the `2016-01-01` instance has not succeeded yet. The scheduler pays special attention for jobs with an SLA and sends alert emails for sla misses. SLA misses are also recorded in the database for future reference. All tasks that share the same SLA time get bundled in a single email, sent soon after that time. SLA notification are sent once and only once for each task instance.
- **execution_timeout** (*datetime.timedelta*) – max time allowed for the execution of this task instance, if it goes beyond it will raise and fail.
- **on_failure_callback** (*callable*) – a function to be called when a task instance of this task fails. a context dictionary is passed as a single parameter to this function. Context contains references to related objects to the task instance and is documented under the macros section of the API.
- **on_retry_callback** (*callable*) – much like the `on_failure_callback` except that it is executed when retries occur.
- **on_success_callback** (*callable*) – much like the `on_failure_callback` except that it is executed when the task succeeds.
- **trigger_rule** (*str*) – defines the rule by which dependencies are applied for the task to get triggered. Options are: `{ all_success | all_failed | all_done | one_success | one_failed | none_failed | none_skipped | dummy}` default is `all_success`. Options can be set as string or using the constants defined in the static class `airflow.utils.TriggerRule`
- **resources** (*dict*) – A map of resource parameter names (the argument names of the Resources constructor) to their values.
- **run_as_user** (*str*) – unix username to impersonate while running the task
- **task_concurrency** (*int*) – When set, a task will be able to limit the concurrent runs across execution_dates
- **executor_config** (*dict*) – Additional task-level configuration parameters that are interpreted by a specific executor. Parameters are namespaced by the name of executor.

  Example: to run this task in a specific docker container through the KubernetesExecutor

  ```
  MyOperator(...,
      executor_config={
      "KubernetesExecutor":
          {"image": "myCustomDockerImage"}
          }
  )
  ```

- **do_xcom_push** (*bool*) – if True, an XCom is pushed containing the Operator's result

template_fields :Iterable[str] = []

template_ext :Iterable[str] = []

ui_color = #fff

ui_fgcolor = #000

_base_operator_shallow_copy_attrs = ['user_defined_macros', 'user_defined_filters', 'p

shallow_copy_attrs :Iterable[str] = []

operator_extra_links :Iterable[BaseOperatorLink] = []

dag
: Returns the Operator's DAG if set, otherwise raises an error

dag_id

deps
: Returns the list of dependencies for the operator. These differ from execution context dependencies in that they are specific to tasks and can be extended/overridden by subclasses.

schedule_interval
: The schedule interval of the DAG always wins over individual tasks so that tasks within a DAG always line up. The task still needs a schedule_interval as it may not be attached to a DAG.

priority_weight_total

upstream_list
: @property: list of tasks directly upstream

upstream_task_ids

downstream_list
: @property: list of tasks directly downstream

downstream_task_ids

task_type

__eq__(*self*, *other*)

__ne__(*self*, *other*)

__lt__(*self*, *other*)

__hash__(*self*)

__rshift__(*self*, *other*)
: Implements Self >> Other == self.set_downstream(other)

  If "Other" is a DAG, the DAG is assigned to the Operator.

__lshift__(*self*, *other*)
: Implements Self << Other == self.set_upstream(other)

  If "Other" is a DAG, the DAG is assigned to the Operator.

__rrshift__(*self*, *other*)
: Called for [DAG] >> [Operator] because DAGs don't have __rshift__ operators.

__rlshift__(*self*, *other*)
: Called for [DAG] << [Operator] because DAGs don't have __lshift__ operators.

has_dag(*self*)
: Returns True if the Operator has been assigned to a DAG.

operator_extra_link_dict(*self*)

global_operator_extra_link_dict(*self*)

pre_execute(*self*, *context*)
: This hook is triggered right before self.execute() is called.

`execute` (*self*, *context*)
: This is the main method to derive when creating an operator. Context is the same dictionary used as when rendering jinja templates.

  Refer to get_template_context for more context.

`post_execute` (*self*, *context*, *result=None*)
: This hook is triggered right after self.execute() is called. It is passed the execution context and any results returned by the operator.

`on_kill` (*self*)
: Override this method to cleanup subprocesses when a task instance gets killed. Any use of the threading, subprocess or multiprocessing module within an operator needs to be cleaned up or it will leave ghost processes behind.

`__deepcopy__` (*self*, *memo*)
: Hack sorting double chained task lists by task_id to avoid hitting max_depth on deepcopy operations.

`__getstate__` (*self*)

`__setstate__` (*self*, *state*)

`render_template_from_field` (*self*, *attr*, *content*, *context*, *jinja_env*)
: Renders a template from a field. If the field is a string, it will simply render the string and return the result. If it is a collection or nested set of collections, it will traverse the structure and render all elements in it. If the field has another type, it will return it as it is.

`render_template` (*self*, *attr*, *content*, *context*)
: Renders a template either from a file or directly in a field, and returns the rendered result.

`get_template_env` (*self*)

`prepare_template` (*self*)
: Hook that is triggered after the templated fields get replaced by their content. If you need your operator to alter the content of the file before the template is rendered, it should override this method to do so.

`resolve_template_files` (*self*)

`clear` (*self*, *start_date=None*, *end_date=None*, *upstream=False*, *downstream=False*, *session=None*)
: Clears the state of task instances associated with the task, following the parameters specified.

`get_task_instances` (*self*, *start_date=None*, *end_date=None*, *session=None*)
: Get a set of task instance related to this task for a specific date range.

`get_flat_relative_ids` (*self*, *upstream=False*, *found_descendants=None*)
: Get a flat list of relatives' ids, either upstream or downstream.

`get_flat_relatives` (*self*, *upstream=False*)
: Get a flat list of relatives, either upstream or downstream.

`run` (*self*, *start_date=None*, *end_date=None*, *ignore_first_depends_on_past=False*, *ignore_ti_state=False*, *mark_success=False*)
: Run a set of task instances for a date range.

`dry_run` (*self*)

`get_direct_relative_ids` (*self*, *upstream=False*)
: Get the direct relative ids to the current task, upstream or downstream.

`get_direct_relatives` (*self*, *upstream=False*)
: Get the direct relatives to the current task, upstream or downstream.

`__repr__` (*self*)

`add_only_new` (*self*, *item_set*, *item*)

`_set_relatives` (*self*, *task_or_task_list*, *upstream=False*)

set_downstream(*self*, *task_or_task_list*)
Set a task or a task list to be directly downstream from the current task.

set_upstream(*self*, *task_or_task_list*)
Set a task or a task list to be directly upstream from the current task.

xcom_push(*self*, *context*, *key*, *value*, *execution_date=None*)
See TaskInstance.xcom_push()

xcom_pull(*self*, *context*, *task_ids=None*, *dag_id=None*, *key=XCOM_RETURN_KEY*, *include_prior_dates=None*)
See TaskInstance.xcom_pull()

extra_links(*self*)

get_extra_links(*self*, *dttm*, *link_name*)
For an operator, gets the URL that the external links specified in *extra_links* should point to. :raise ValueError: The error message of a ValueError will be passed on through to the fronted to show up as a tooltip on the disabled link :param dttm: The datetime parsed execution date for the URL being searched for :param link_name: The name of the link we're looking for the URL for. Should be one of the options specified in *extra_links* :return: A URL

class airflow.models.baseoperator.**BaseOperatorLink**
Abstract base class that defines how we get an operator link.

name
Name of the link. This will be the button name on the task UI.

Returns link name

get_link(*self*, *operator:BaseOperator*, *dttm:datetime*)
Link to external system.

Parameters

- **operator** – airflow operator
- **dttm** – datetime

Returns link to external system

airflow.models.connection

Module Contents

airflow.models.connection.**parse_netloc_to_hostname**(*uri_parts*)

class airflow.models.connection.**Connection**(*conn_id=None*, *conn_type=None*, *host=None*, *login=None*, *password=None*, *schema=None*, *port=None*, *extra=None*, *uri=None*)
Bases: *airflow.models.base.Base*, airflow.LoggingMixin

Placeholder to store information about different database instances connection information. The idea here is that scripts use references to database instances (conn_id) instead of hard coding hostname, logins and passwords when using operators or hooks.

__tablename__ = connection

id

conn_id

conn_type

host

schema

login

_password

port

is_encrypted

is_extra_encrypted

_extra

_types = [['docker', 'Docker Registry'], ['fs', 'File (path)'], ['ftp', 'FTP'], ['goog

password

extra

extra_dejson
Returns the extra property by deserializing json.

parse_from_uri (*self*, *uri*)

get_password (*self*)

set_password (*self*, *value*)

get_extra (*self*)

set_extra (*self*, *value*)

rotate_fernet_key (*self*)

get_hook (*self*)

__repr__ (*self*)

debug_info (*self*)

airflow.models.crypto

Module Contents

exception airflow.models.crypto.**InvalidFernetToken**
Bases: Exception

class airflow.models.crypto.**NullFernet**
A "Null" encryptor class that doesn't encrypt or decrypt but that presents a similar interface to Fernet.

The purpose of this is to make the rest of the code not have to know the difference, and to only display the message once, not 20 times when *airflow initdb* is ran.

is_encrypted = False

decrpyt (*self*, *b*)

encrypt (*self*, *b*)

airflow.models.crypto.**_fernet**

airflow.models.crypto.**get_fernet** ()
Deferred load of Fernet key.
This function could fail either because Cryptography is not installed or because the Fernet key is invalid.

Returns Fernet object

Raises airflow.exceptions.AirflowException if there's a problem trying to load Fernet

`airflow.models.dag`

Module Contents

`airflow.models.dag.`**`ScheduleInterval`**

`airflow.models.dag.`**`get_last_dagrun`**(*dag_id*, *session*, *include_externally_triggered=False*)

`Returns the last dag run for a dag, None if there was none.`
`Last dag run can be any type of run eg. scheduled or backfilled.`
`Overridden DagRuns are ignored.`

`class` `airflow.models.dag.`**`DAG`**(*dag_id:str*, *description:str=''*, *schedule_interval:Optional[ScheduleInterval]=timedelta(days=1)*, *start_date:Optional[datetime]=None*, *end_date:Optional[datetime]=None*, *full_filepath:Optional[str]=None*, *template_searchpath:Optional[Union[str*, *Iterable[str]]]=None*, *template_undefined:Type[jinja2.Undefined]=jinja2.Undefined*, *user_defined_macros:Optional[Dict]=None*, *user_defined_filters:Optional[Dict]=None*, *default_args:Optional[Dict]=None*, *concurrency:int=configuration.conf.getint('core'*, *'dag_concurrency')*, *max_active_runs:int=configuration.conf.getint('core'*, *'max_active_runs_per_dag')*, *dagrun_timeout:Optional[timedelta]=None*, *sla_miss_callback:Optional[Callable]=None*, *default_view:Optional[str]=None*, *orientation:str=configuration.conf.get('webserver'*, *'dag_orientation')*, *catchup:bool=configuration.conf.getboolean('scheduler'*, *'catchup_by_default')*, *on_success_callback:Optional[Callable]=None*, *on_failure_callback:Optional[Callable]=None*, *doc_md:Optional[str]=None*, *params:Optional[Dict]=None*, *access_control:Optional[Dict]=None*)

Bases: `airflow.dag.base_dag.BaseDag`, `airflow.utils.log.logging_mixin.LoggingMixin`

A dag (directed acyclic graph) is a collection of tasks with directional dependencies. A dag also has a schedule, a start date and an end date (optional). For each schedule, (say daily or hourly), the DAG needs to run each individual tasks as their dependencies are met. Certain tasks have the property of depending on their own past, meaning that they can't run until their previous schedule (and upstream tasks) are completed.

DAGs essentially act as namespaces for tasks. A task_id can only be added once to a DAG.

Parameters

- **`dag_id`** (*`str`*) – The id of the DAG
- **`description`** (*`str`*) – The description for the DAG to e.g. be shown on the webserver
- **`schedule_interval`** (*`datetime.timedelta or dateutil.relativedelta.relativedelta or str that acts as a cron expression`*) – Defines how often that DAG runs, this timedelta object gets added to your latest task instance's execution_date to figure out the next schedule
- **`start_date`** (*`datetime.datetime`*) – The timestamp from which the scheduler will attempt to backfill
- **`end_date`** (*`datetime.datetime`*) – A date beyond which your DAG won't run, leave to None for open ended scheduling

- **template_searchpath** (*str or list[str]*) – This list of folders (non relative) defines where jinja will look for your templates. Order matters. Note that jinja/airflow includes the path of your DAG file by default
- **template_undefined** (*jinja2.Undefined*) – Template undefined type.
- **user_defined_macros** (*dict*) – a dictionary of macros that will be exposed in your jinja templates. For example, passing `dict(foo='bar')` to this argument allows you to `{{ foo }}` in all jinja templates related to this DAG. Note that you can pass any type of object here.
- **user_defined_filters** (*dict*) – a dictionary of filters that will be exposed in your jinja templates. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to `{{ 'world' | hello }}` in all jinja templates related to this DAG.
- **default_args** (*dict*) – A dictionary of default parameters to be used as constructor keyword parameters when initialising operators. Note that operators have the same hook, and precede those defined here, meaning that if your dict contains *'depends_on_past': True* here and *'depends_on_past': False* in the operator's call *default_args*, the actual value will be *False*.
- **params** (*dict*) – a dictionary of DAG level parameters that are made accessible in templates, namespaced under *params*. These params can be overridden at the task level.
- **concurrency** (*int*) – the number of task instances allowed to run concurrently
- **max_active_runs** (*int*) – maximum number of active DAG runs, beyond this number of DAG runs in a running state, the scheduler won't create new active DAG runs
- **dagrun_timeout** (*datetime.timedelta*) – specify how long a DagRun should be up before timing out / failing, so that new DagRuns can be created. The timeout is only enforced for scheduled DagRuns, and only once the # of active DagRuns == max_active_runs.
- **sla_miss_callback** (*types.FunctionType*) – specify a function to call when reporting SLA timeouts.
- **default_view** (*str*) – Specify DAG default view (tree, graph, duration, gantt, landing_times)
- **orientation** (*str*) – Specify DAG orientation in graph view (LR, TB, RL, BT)
- **catchup** (*bool*) – Perform scheduler catchup (or only run latest)? Defaults to True
- **on_failure_callback** (*callable*) – A function to be called when a DagRun of this dag fails. A context dictionary is passed as a single parameter to this function.
- **on_success_callback** (*callable*) – Much like the `on_failure_callback` except that it is executed when the dag succeeds.
- **access_control** (*dict*) – Specify optional DAG-level permissions, e.g., "{'role1': {'can_dag_read'}, 'role2': {'can_dag_read', 'can_dag_edit'}}"

dag_id

full_filepath

concurrency

access_control

description

pickle_id

tasks

task_ids

`filepath`
File location of where the dag object is instantiated

`folder`
Folder location of where the dag object is instantiated

`owner`
Return list of all owners found in DAG tasks.

Returns Comma separated list of owners in DAG tasks

Return type str

`concurrency_reached`
Returns a boolean indicating whether the concurrency limit for this DAG has been reached

`is_paused`
Returns a boolean indicating whether this DAG is paused

`latest_execution_date`
Returns the latest date for which at least one dag run exists

`subdags`
Returns a list of the subdag objects associated to this DAG

`roots`

`__repr__`(*self*)

`__eq__`(*self*, *other*)

`__ne__`(*self*, *other*)

`__lt__`(*self*, *other*)

`__hash__`(*self*)

`__enter__`(*self*)

`__exit__`(*self*, *_type*, *_value*, *_tb*)

`get_default_view`(*self*)
This is only there for backward compatible jinja2 templates

`date_range`(*self*, *start_date*, *num=None*, *end_date=timezone.utcnow()*)

`is_fixed_time_schedule`(*self*)
Figures out if the DAG schedule has a fixed time (e.g. 3 AM).

Returns True if the schedule has a fixed time, False if not.

`following_schedule`(*self*, *dttm*)
Calculates the following schedule for this dag in UTC.

Parameters **`dttm`** – utc datetime

Returns utc datetime

`previous_schedule`(*self*, *dttm*)
Calculates the previous schedule for this dag in UTC

Parameters **`dttm`** – utc datetime

Returns utc datetime

`get_run_dates`(*self*, *start_date*, *end_date=None*)
Returns a list of dates between the interval received as parameter using this dag's schedule interval. Returned dates can be used for execution dates.

Parameters

- **start_date** (*datetime*) – the start date of the interval
- **end_date** (*datetime*) – the end date of the interval, defaults to timezone.utcnow()

Returns a list of dates within the interval following the dag's schedule

Return type list

normalize_schedule(*self*, *dttm*)
Returns dttm + interval unless dttm is first interval then it returns dttm

get_last_dagrun(*self*, *session=None*, *include_externally_triggered=False*)

_get_concurrency_reached(*self*, *session=None*)

_get_is_paused(*self*, *session=None*)

handle_callback(*self*, *dagrun*, *success=True*, *reason=None*, *session=None*)
Triggers the appropriate callback depending on the value of success, namely the on_failure_callback or on_success_callback. This method gets the context of a single TaskInstance part of this DagRun and passes that to the callable along with a 'reason', primarily to differentiate DagRun failures.

Parameters

- **dagrun** – DagRun object
- **success** – Flag to specify if failure or success callback should be called
- **reason** – Completion reason
- **session** – Database session

get_active_runs(*self*)
Returns a list of dag run execution dates currently running

Returns List of execution dates

get_num_active_runs(*self*, *external_trigger=None*, *session=None*)
Returns the number of active "running" dag runs

Parameters

- **external_trigger** (*bool*) – True for externally triggered active dag runs
- **session** –

Returns number greater than 0 for active dag runs

get_dagrun(*self*, *execution_date*, *session=None*)
Returns the dag run for a given execution date if it exists, otherwise none.

Parameters

- **execution_date** – The execution date of the DagRun to find.
- **session** –

Returns The DagRun if found, otherwise None.

_get_latest_execution_date(*self*, *session=None*)

resolve_template_files(*self*)

get_template_env(*self*)
Returns a jinja2 Environment while taking into account the DAGs template_searchpath, user_defined_macros and user_defined_filters

set_dependency(*self*, *upstream_task_id*, *downstream_task_id*)
Simple utility method to set dependency between two tasks that already have been added to the DAG using add_task()

get_task_instances (*self*, *start_date=None*, *end_date=None*, *state=None*, *session=None*)

topological_sort (*self*)
Sorts tasks in topographical order, such that a task comes after any of its upstream dependencies.

Heavily inspired by: http://blog.jupo.org/2012/04/06/topological-sorting-acyclic-directed-graphs/

Returns list of tasks in topological order

set_dag_runs_state (*self*, *state=State.RUNNING*, *session=None*, *start_date=None*, *end_date=None*)

clear (*self*, *start_date=None*, *end_date=None*, *only_failed=False*, *only_running=False*, *confirm_prompt=False*, *include_subdags=True*, *include_parentdag=True*, *reset_dag_runs=True*, *dry_run=False*, *session=None*, *get_tis=False*)
Clears a set of task instances associated with the current dag for a specified date range.

classmethod clear_dags (*cls*, *dags*, *start_date=None*, *end_date=None*, *only_failed=False*, *only_running=False*, *confirm_prompt=False*, *include_subdags=True*, *include_parentdag=False*, *reset_dag_runs=True*, *dry_run=False*)

__deepcopy__ (*self*, *memo*)

sub_dag (*self*, *task_regex*, *include_downstream=False*, *include_upstream=True*)
Returns a subset of the current dag as a deep copy of the current dag based on a regex that should match one or many tasks, and includes upstream and downstream neighbours based on the flag passed.

has_task (*self*, *task_id*)

get_task (*self*, *task_id*)

pickle_info (*self*)

pickle (*self*, *session=None*)

tree_view (*self*)
Shows an ascii tree representation of the DAG

add_task (*self*, *task*)
Add a task to the DAG

Parameters **task** (*task*) – the task you want to add

add_tasks (*self*, *tasks*)
Add a list of tasks to the DAG

Parameters **tasks** (*list of tasks*) – a lit of tasks you want to add

run (*self*, *start_date=None*, *end_date=None*, *mark_success=False*, *local=False*, *executor=None*, *donot_pickle=configuration.conf.getboolean('core', 'donot_pickle')*, *ignore_task_deps=False*, *ignore_first_depends_on_past=False*, *pool=None*, *delay_on_limit_secs=1.0*, *verbose=False*, *conf=None*, *rerun_failed_tasks=False*, *run_backwards=False*)
Runs the DAG.

Parameters

- **start_date** (*datetime.datetime*) – the start date of the range to run
- **end_date** (*datetime.datetime*) – the end date of the range to run
- **mark_success** (*bool*) – True to mark jobs as succeeded without running them
- **local** (*bool*) – True to run the tasks using the LocalExecutor
- **executor** (*airflow.executor.BaseExecutor*) – The executor instance to run the tasks
- **donot_pickle** (*bool*) – True to avoid pickling DAG object and send to workers
- **ignore_task_deps** (*bool*) – True to skip upstream tasks

- **`ignore_first_depends_on_past`** (*`bool`*) – True to ignore depends_on_past dependencies for the first set of tasks only
- **`pool`** (*`str`*) – Resource pool to use
- **`delay_on_limit_secs`** (*`float`*) – Time in seconds to wait before next attempt to run dag run when max_active_runs limit has been reached
- **`verbose`** (*`bool`*) – Make logging output more verbose
- **`conf`** (*`dict`*) – user defined dictionary passed from CLI
- **`rerun_failed_tasks`** –
- **`run_backwards`** –

Type bool

Type bool

`cli`(*self*)
Exposes a CLI specific to this DAG

`create_dagrun`(*self*, *run_id*, *state*, *execution_date=None*, *start_date=None*, *external_trigger=False*, *conf=None*, *session=None*)
Creates a dag run from this dag including the tasks associated with this dag. Returns the dag run.

Parameters

- **`run_id`** (*`str`*) – defines the the run id for this dag run
- **`execution_date`** (*`datetime.datetime`*) – the execution date of this dag run
- **`state`** (*`airflow.utils.state.State`*) – the state of the dag run
- **`start_date`** (*`datetime`*) – the date this dag run should be evaluated
- **`external_trigger`** (*`bool`*) – whether this dag run is externally triggered
- **`session`** (*`sqlalchemy.orm.session.Session`*) – database session

`sync_to_db`(*self*, *owner=None*, *sync_time=None*, *session=None*)
Save attributes about this DAG to the DB. Note that this method can be called for both DAGs and SubDAGs. A SubDag is actually a SubDagOperator.

Parameters

- **`dag`** (`airflow.models.DAG`) – the DAG object to save to the DB
- **`sync_time`** (*`datetime`*) – The time that the DAG should be marked as sync'ed

Returns None

`static deactivate_unknown_dags`(*active_dag_ids*, *session=None*)
Given a list of known DAGs, deactivate any other DAGs that are marked as active in the ORM

Parameters **`active_dag_ids`** (*`list[unicode]`*) – list of DAG IDs that are active

Returns None

`static deactivate_stale_dags`(*expiration_date*, *session=None*)
Deactivate any DAGs that were last touched by the scheduler before the expiration date. These DAGs were likely deleted.

Parameters **`expiration_date`** (*`datetime`*) – set inactive DAGs that were touched before this time

Returns None

`static get_num_task_instances`(*dag_id*, *task_ids=None*, *states=None*, *session=None*)
Returns the number of task instances in the given DAG.

Parameters

- **session** – ORM session
- **dag_id** (*unicode*) – ID of the DAG to get the task concurrency of
- **task_ids** (*list[unicode]*) – A list of valid task IDs for the given DAG
- **states** (*list[state]*) – A list of states to filter by if supplied

Returns The number of running tasks

Return type int

test_cycle(*self*)
Check to see if there are any cycles in the DAG. Returns False if no cycle found, otherwise raises exception.

_test_cycle_helper(*self*, *visit_map*, *task_id*)
Checks if a cycle exists from the input task using DFS traversal

class airflow.models.dag.**DagModel**
Bases: *airflow.models.base.Base*

__tablename__ = dag
These items are stored in the database for state related information

dag_id

is_paused_at_creation

is_paused

is_subdag

is_active

last_scheduler_run

last_pickled

last_expired

scheduler_lock

pickle_id

fileloc

owners

description

default_view

schedule_interval

timezone

safe_dag_id

__repr__(*self*)

static get_dagmodel(*dag_id*, *session=None*)

classmethod get_current(*cls*, *dag_id*, *session=None*)

get_default_view(*self*)

get_last_dagrun(*self*, *session=None*, *include_externally_triggered=False*)

get_dag(*self*)

create_dagrun (*self*, *run_id*, *state*, *execution_date*, *start_date=None*, *external_trigger=False*, *conf=None*, *session=None*)

Creates a dag run from this dag including the tasks associated with this dag. Returns the dag run.

Parameters

- **run_id** (*str*) – defines the the run id for this dag run
- **execution_date** (*datetime.datetime*) – the execution date of this dag run
- **state** (*airflow.utils.state.State*) – the state of the dag run
- **start_date** (*datetime.datetime*) – the date this dag run should be evaluated
- **external_trigger** (*bool*) – whether this dag run is externally triggered
- **session** (*sqlalchemy.orm.session.Session*) – database session

set_is_paused (*self*, *is_paused:bool*, *including_subdags:bool=True*, *session=None*)

Pause/Un-pause a DAG.

Parameters

- **is_paused** – Is the DAG paused
- **including_subdags** – whether to include the DAG's subdags
- **session** – session

airflow.models.dagbag

Module Contents

class airflow.models.dagbag.**DagBag** (*dag_folder=None*, *executor=None*, *include_examples=configuration.conf.getboolean('core', 'LOAD_EXAMPLES')*, *safe_mode=configuration.conf.getboolean('core', 'DAG_DISCOVERY_SAFE_MODE')*)

Bases: airflow.dag.base_dag.BaseDagBag, airflow.utils.log.logging_mixin.LoggingMixin

A dagbag is a collection of dags, parsed out of a folder tree and has high level configuration settings, like what database to use as a backend and what executor to use to fire off tasks. This makes it easier to run distinct environments for say production and development, tests, or for different teams or security profiles. What would have been system level settings are now dagbag level so that one system can run multiple, independent settings sets.

Parameters

- **dag_folder** (*unicode*) – the folder to scan to find DAGs
- **executor** – the executor to use when executing task instances in this DagBag
- **include_examples** (*bool*) – whether to include the examples that ship with airflow or not
- **has_logged** – an instance boolean that gets flipped from False to True after a file has been skipped. This is to prevent overloading the user with logging messages about skipped files. Therefore only once per DagBag is a file logged being skipped.

CYCLE_NEW = 0

CYCLE_IN_PROGRESS = 1

CYCLE_DONE = 2

dag_ids

size (*self*)

Returns the amount of dags contained in this dagbag

get_dag (*self*, *dag_id*)
Gets the DAG out of the dictionary, and refreshes it if expired

process_file (*self*, *filepath*, *only_if_updated=True*, *safe_mode=True*)
Given a path to a python module or zip file, this method imports the module and look for dag objects within it.

kill_zombies (*self*, *zombies*, *session=None*)
Fail given zombie tasks, which are tasks that haven't had a heartbeat for too long, in the current DagBag.

Parameters

- **zombies** (*airflow.utils.dag_processing.SimpleTaskInstance*) – zombie task instances to kill.
- **session** (*sqlalchemy.orm.session.Session*) – DB session.

bag_dag (*self*, *dag*, *parent_dag*, *root_dag*)
Adds the DAG into the bag, recurses into sub dags. Throws AirflowDagCycleException if a cycle is detected in this dag or its subdags

collect_dags (*self*, *dag_folder=None*, *only_if_updated=True*, *include_examples=configuration.conf.getboolean('core', 'LOAD_EXAMPLES')*, *safe_mode=configuration.conf.getboolean('core', 'DAG_DISCOVERY_SAFE_MODE')*)
Given a file path or a folder, this method looks for python modules, imports them and adds them to the dagbag collection.

Note that if a `.airflowignore` file is found while processing the directory, it will behave much like a `.gitignore`, ignoring files that match any of the regex patterns specified in the file.

Note: The patterns in .airflowignore are treated as un-anchored regexes, not shell-like glob patterns.

dagbag_report (*self*)
Prints a report around DagBag loading stats

airflow.models.dagpickle

Module Contents

class airflow.models.dagpickle.**DagPickle** (*dag*)
Bases: *airflow.models.base.Base*

Dags can originate from different places (user repos, master repo, ...) and also get executed in different places (different executors). This object represents a version of a DAG and becomes a source of truth for a BackfillJob execution. A pickle is a native python serialized object, and in this case gets stored in the database for the duration of the job.

The executors pick up the DagPickle id and read the dag definition from the database.

id

pickle

created_dttm

pickle_hash

__tablename__ = dag_pickle

airflow.models.dagrun

Module Contents

class airflow.models.dagrun.**DagRun**

Bases: *airflow.models.base.Base*, airflow.utils.log.logging_mixin.LoggingMixin

DagRun describes an instance of a Dag. It can be created by the scheduler (for regular runs) or by an external trigger

__tablename__ = dag_run

ID_PREFIX = scheduled__

ID_FORMAT_PREFIX

id

dag_id

execution_date

start_date

end_date

_state

run_id

external_trigger

conf

dag

__table_args__

state

is_backfill

__repr__(*self*)

get_state(*self*)

set_state(*self*, *state*)

classmethod id_for_date(*cls*, *date*, *prefix=ID_FORMAT_PREFIX*)

refresh_from_db(*self*, *session=None*)

Reloads the current dagrun from the database :param session: database session

static find(*dag_id=None*, *run_id=None*, *execution_date=None*, *state=None*, *external_trigger=None*, *no_backfills=False*, *session=None*)

Returns a set of dag runs for the given search criteria.

Parameters

- **dag_id** (*int, list*) – the dag_id to find dag runs for
- **run_id** (*str*) – defines the the run id for this dag run
- **execution_date** (*datetime.datetime*) – the execution date
- **state** (*str*) – the state of the dag run
- **external_trigger** (*bool*) – whether this dag run is externally triggered
- **no_backfills** (*bool*) – return no backfills (True), return all (False). Defaults to False
- **session** (*sqlalchemy.orm.session.Session*) – database session

get_task_instances (*self*, *state=None*, *session=None*)
Returns the task instances for this dag run

get_task_instance (*self*, *task_id*, *session=None*)
Returns the task instance specified by task_id for this dag run

Parameters **task_id** – the task id

get_dag (*self*)
Returns the Dag associated with this DagRun.

Returns DAG

get_previous_dagrun (*self*, *state:str=None*, *session:Session=None*)
The previous DagRun, if there is one

get_previous_scheduled_dagrun (*self*, *session=None*)
The previous, SCHEDULED DagRun, if there is one

update_state (*self*, *session=None*)
Determines the overall state of the DagRun based on the state of its TaskInstances.

Returns State

_emit_duration_stats_for_finished_state (*self*)

verify_integrity (*self*, *session=None*)
Verifies the DagRun by checking for removed tasks or tasks that are not in the database yet. It will set state to removed or add the task if required.

static get_run (*session*, *dag_id*, *execution_date*)

Parameters

- **dag_id** (*unicode*) – DAG ID
- **execution_date** (*datetime*) – execution date

Returns DagRun corresponding to the given dag_id and execution date if one exists. None otherwise.

Return type *airflow.models.DagRun*

classmethod get_latest_runs (*cls*, *session*)
Returns the latest DagRun for each DAG.

airflow.models.errors

Module Contents

class airflow.models.errors.**ImportError**
Bases: *airflow.models.base.Base*

__tablename__ = import_error

id

timestamp

filename

stacktrace

airflow.models.kubernetes

Module Contents

class airflow.models.kubernetes.**KubeResourceVersion**

Bases: *airflow.models.base.Base*

__tablename__ = kube_resource_version

one_row_id

resource_version

static get_current_resource_version(*session=None*)

static checkpoint_resource_version(*resource_version*, *session=None*)

static reset_resource_version(*session=None*)

class airflow.models.kubernetes.**KubeWorkerIdentifier**

Bases: *airflow.models.base.Base*

__tablename__ = kube_worker_uuid

one_row_id

worker_uuid

static get_or_create_current_kube_worker_uuid(*session=None*)

static checkpoint_kube_worker_uuid(*worker_uuid*, *session=None*)

airflow.models.log

Module Contents

class airflow.models.log.**Log**(*event*, *task_instance*, *owner=None*, *extra=None*, ***kwargs*)

Bases: *airflow.models.base.Base*

Used to actively log events to the database

__tablename__ = log

id

dttm

dag_id

task_id

event

execution_date

owner

extra

__table_args__

`airflow.models.pool`

Module Contents

class `airflow.models.pool.`**`Pool`**
Bases: *`airflow.models.base.Base`*

`__tablename__ = slot_pool`

`id`

`pool`

`slots`

`description`

`default_pool_name = not_pooled`

`__repr__`(*self*)

`static default_pool_open_slots`(*session*)

`to_json`(*self*)

`used_slots`(*self*, *session*)
Returns the number of slots used at the moment

`queued_slots`(*self*, *session*)
Returns the number of slots used at the moment

`open_slots`(*self*, *session*)
Returns the number of slots open at the moment

`airflow.models.skipmixin`

Module Contents

class `airflow.models.skipmixin.`**`SkipMixin`**
Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

`skip`(*self*, *dag_run*, *execution_date*, *tasks*, *session=None*)
Sets tasks instances to skipped from the same dag run.

Parameters

- **`dag_run`** – the DagRun for which to set the tasks to skipped
- **`execution_date`** – execution_date
- **`tasks`** – tasks to skip (not task_ids)
- **`session`** – db session to use

`airflow.models.slamiss`

Module Contents

class `airflow.models.slamiss.`**`SlaMiss`**
Bases: *`airflow.models.base.Base`*

Model that stores a history of the SLA that have been missed. It is used to keep track of SLA failures over time and to avoid double triggering alert emails.

`__tablename__ = sla_miss`

`task_id`

`dag_id`

`execution_date`

`email_sent`

`timestamp`

`description`

`notification_sent`

`__table_args__`

`__repr__`(*self*)

`airflow.models.taskfail`

Module Contents

class `airflow.models.taskfail.`**`TaskFail`**(*task*, *execution_date*, *start_date*, *end_date*)

Bases: *`airflow.models.base.Base`*

TaskFail tracks the failed run durations of each task instance.

`__tablename__ = task_fail`

`id`

`task_id`

`dag_id`

`execution_date`

`start_date`

`end_date`

`duration`

`__table_args__`

`airflow.models.taskinstance`

Module Contents

`airflow.models.taskinstance.`**`clear_task_instances`**(*tis*, *session*, *activate_dag_runs=True*, *dag=None*)

`Clears a set of task instances, but makes sure the running ones get killed.`

Parameters

- **`tis`** – a list of task instances
- **`session`** – current session
- **`activate_dag_runs`** – flag to check for active dag run

- **dag** – DAG object

class airflow.models.taskinstance.**TaskInstance**(*task*, *execution_date*, *state=None*)

Bases: *airflow.models.base.Base*, airflow.utils.log.logging_mixin.LoggingMixin

Task instances store the state of a task instance. This table is the authority and single source of truth around what tasks have run and the state they are in.

The SqlAlchemy model doesn't have a SqlAlchemy foreign key to the task or dag model deliberately to have more control over transactions.

Database transactions on this table should insure double triggers and any confusion around what task instances are or aren't ready to run even while multiple schedulers may be firing task instances.

__tablename__ = task_instance

task_id

dag_id

execution_date

start_date

end_date

duration

state

_try_number

max_tries

hostname

unixname

job_id

pool

queue

priority_weight

operator

queued_dttm

pid

executor_config

__table_args__

try_number

Return the try number that this task number will be when it is actually run.

If the TI is currently running, this will match the column in the databse, in all othercases this will be incremenetd

next_try_number

log_filepath

log_url

mark_success_url

key

Returns a tuple that identifies the task instance uniquely

is_premature
Returns whether a task is in UP_FOR_RETRY state and its retry interval has elapsed.

previous_ti
The task instance for the task that ran before this task instance.

previous_ti_success
The ti from prior succesful dag run for this task, by execution date.

previous_execution_date_success
The execution date from property previous_ti_success.

previous_start_date_success
The start date from property previous_ti_success.

init_on_load(*self*)
Initialize the attributes that aren't stored in the DB.

command(*self*, *mark_success=False*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)
Returns a command that can be executed anywhere where airflow is installed. This command is part of the message sent to executors by the orchestrator.

command_as_list(*self*, *mark_success=False*, *ignore_all_deps=False*, *ignore_task_deps=False*, *ignore_depends_on_past=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)
Returns a command that can be executed anywhere where airflow is installed. This command is part of the message sent to executors by the orchestrator.

static generate_command(*dag_id*, *task_id*, *execution_date*, *mark_success=False*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *file_path=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)
Generates the shell command required to execute this task instance.

Parameters

- **dag_id** (*unicode*) – DAG ID
- **task_id** (*unicode*) – Task ID
- **execution_date** (*datetime.datetime*) – Execution date for the task
- **mark_success** (*bool*) – Whether to mark the task as successful
- **ignore_all_deps** (*bool*) – Ignore all ignorable dependencies. Overrides the other ignore_* parameters.
- **ignore_depends_on_past** (*bool*) – Ignore depends_on_past parameter of DAGs (e.g. for Backfills)
- **ignore_task_deps** (*bool*) – Ignore task-specific dependencies such as depends_on_past and trigger rule
- **ignore_ti_state** (*bool*) – Ignore the task instance's previous failure/success
- **local** (*bool*) – Whether to run the task locally
- **pickle_id** (*unicode*) – If the DAG was serialized to the DB, the ID associated with the pickled DAG
- **file_path** – path to the file containing the DAG definition
- **raw** – raw mode (needs more details)

- **`job_id`** – job ID (needs more details)
- **`pool`** (*`unicode`*) – the Airflow pool that the task should run in
- **`cfg_path`** (*`str`*) – the Path to the configuration file

Returns shell command that can be used to run the task instance

`current_state`(*self*, *session=None*)
Get the very latest state from the database, if a session is passed, we use and looking up the state becomes part of the session, otherwise a new session is used.

`error`(*self*, *session=None*)
Forces the task instance's state to FAILED in the database.

`refresh_from_db`(*self*, *session=None*, *lock_for_update=False*)
Refreshes the task instance from the database based on the primary key

Parameters **`lock_for_update`** – if True, indicates that the database should lock the TaskInstance (issuing a FOR UPDATE clause) until the session is committed.

`clear_xcom_data`(*self*, *session=None*)
Clears all XCom data from the database for the task instance

`set_state`(*self*, *state*, *session=None*, *commit=True*)

`are_dependents_done`(*self*, *session=None*)
Checks whether the dependents of this task instance have all succeeded. This is meant to be used by wait_for_downstream.

This is useful when you do not want to start processing the next schedule of a task until the dependents are done. For instance, if the task DROPs and recreates a table.

`_get_previous_ti`(*self*, *state:str=None*, *session:Session=None*)

`are_dependencies_met`(*self*, *dep_context=None*, *session=None*, *verbose=False*)
Returns whether or not all the conditions are met for this task instance to be run given the context for the dependencies (e.g. a task instance being force run from the UI will ignore some dependencies).

Parameters

- **`dep_context`** (*`DepContext`*) – The execution context that determines the dependencies that should be evaluated.
- **`session`** (*`sqlalchemy.orm.session.Session`*) – database session
- **`verbose`** (*`bool`*) – whether log details on failed dependencies on info or debug log level

`get_failed_dep_statuses`(*self*, *dep_context=None*, *session=None*)

`__repr__`(*self*)

`next_retry_datetime`(*self*)
Get datetime of the next retry if the task instance fails. For exponential backoff, retry_delay is used as base and will be converted to seconds.

`ready_for_retry`(*self*)
Checks on whether the task instance is in the right state and timeframe to be retried.

`pool_full`(*self*, *session*)
Returns a boolean as to whether the slot pool has room for this task to run

`get_dagrun`(*self*, *session*)
Returns the DagRun for this TaskInstance

Parameters **`session`** –

Returns DagRun

_check_and_change_state_before_execution(*self*, *verbose=True*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *mark_success=False*, *test_mode=False*, *job_id=None*, *pool=None*, *session=None*)

Checks dependencies and then sets state to RUNNING if they are met. Returns True if and only if state is set to RUNNING, which implies that task should be executed, in preparation for _run_raw_task

Parameters

- **verbose** (*bool*) – whether to turn on more verbose logging
- **ignore_all_deps** (*bool*) – Ignore all of the non-critical dependencies, just runs
- **ignore_depends_on_past** (*bool*) – Ignore depends_on_past DAG attribute
- **ignore_task_deps** (*bool*) – Don't check the dependencies of this TI's task
- **ignore_ti_state** (*bool*) – Disregards previous task instance state
- **mark_success** (*bool*) – Don't run the task, mark its state as success
- **test_mode** (*bool*) – Doesn't record success or failure in the DB
- **pool** (*str*) – specifies the pool to use to run the task instance

Returns whether the state was changed to running or not

Return type bool

_run_raw_task(*self*, *mark_success=False*, *test_mode=False*, *job_id=None*, *pool=None*, *session=None*)

Immediately runs the task (without checking or changing db state before execution) and then sets the appropriate final state after completion and runs any post-execute callbacks. Meant to be called only after another function changes the state to running.

Parameters

- **mark_success** (*bool*) – Don't run the task, mark its state as success
- **test_mode** (*bool*) – Doesn't record success or failure in the DB
- **pool** (*str*) – specifies the pool to use to run the task instance

run(*self*, *verbose=True*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *mark_success=False*, *test_mode=False*, *job_id=None*, *pool=None*, *session=None*)

dry_run(*self*)

_handle_reschedule(*self*, *actual_start_date*, *reschedule_exception*, *test_mode=False*, *context=None*, *session=None*)

handle_failure(*self*, *error*, *test_mode=False*, *context=None*, *session=None*)

is_eligible_to_retry(*self*)

Is task instance is eligible for retry

get_template_context(*self*, *session=None*)

overwrite_params_with_dag_run_conf(*self*, *params*, *dag_run*)

render_templates(*self*)

email_alert(*self*, *exception*)

set_duration(*self*)

xcom_push(*self*, *key*, *value*, *execution_date=None*)

Make an XCom available for tasks to pull.

Parameters

- **key** (*str*) – A key for the XCom
- **value** (*any pickleable object*) – A value for the XCom. The value is pickled and stored in the database.
- **execution_date** (*datetime*) – if provided, the XCom will not be visible until this date. This can be used, for example, to send a message to a task on a future date without it being immediately visible.

xcom_pull(*self*, *task_ids=None*, *dag_id=None*, *key=XCOM_RETURN_KEY*, *include_prior_dates=False*)
Pull XComs that optionally meet certain criteria.

The default value for *key* limits the search to XComs that were returned by other tasks (as opposed to those that were pushed manually). To remove this filter, pass key=None (or any desired value).

If a single task_id string is provided, the result is the value of the most recent matching XCom from that task_id. If multiple task_ids are provided, a tuple of matching values is returned. None is returned whenever no matches are found.

Parameters

- **key** (*str*) – A key for the XCom. If provided, only XComs with matching keys will be returned. The default key is 'return_value', also available as a constant XCOM_RETURN_KEY. This key is automatically given to XComs returned by tasks (as opposed to being pushed manually). To remove the filter, pass key=None.
- **task_ids** (*str or iterable of strings (representing task_ids)*) – Only XComs from tasks with matching ids will be pulled. Can pass None to remove the filter.
- **dag_id** (*str*) – If provided, only pulls XComs from this DAG. If None (default), the DAG of the calling task is used.
- **include_prior_dates** (*bool*) – If False, only XComs from the current execution_date are returned. If True, XComs from previous dates are returned as well.

get_num_running_task_instances(*self*, *session*)

init_run_context(*self*, *raw=False*)
Sets the log context.

airflow.models.taskreschedule

Module Contents

class airflow.models.taskreschedule.**TaskReschedule**(*task*, *execution_date*, *try_number*, *start_date*, *end_date*, *reschedule_date*)

Bases: *airflow.models.base.Base*

TaskReschedule tracks rescheduled task instances.

__tablename__ = task_reschedule

id

task_id

dag_id

execution_date

try_number

start_date

end_date

duration

reschedule_date

__table_args__

static find_for_task_instance (*task_instance*, *session*)
Returns all task reschedules for the task instance and try number, in ascending order.

Parameters **task_instance** (airflow.models.TaskInstance) – the task instance to find task reschedules for

airflow.models.variable

Module Contents

class airflow.models.variable.**Variable**
Bases: *airflow.models.base.Base*, airflow.utils.log.logging_mixin.LoggingMixin

__tablename__ = variable

__NO_DEFAULT_SENTINEL

id

key

_val

is_encrypted

val

__repr__ (*self*)

get_val (*self*)

set_val (*self*, *value*)

classmethod setdefault (*cls*, *key*, *default*, *deserialize_json=False*)
Like a Python builtin dict object, setdefault returns the current value for a key, and if it isn't there, stores the default value and returns it.

Parameters

- **key** (*str*) – Dict key for this Variable
- **default** (*Mixed*) – Default value to set and return if the variable isn't already in the DB
- **deserialize_json** – Store this as a JSON encoded value in the DB and un-encode it when retrieving a value

Returns Mixed

classmethod get (*cls*, *key:str*, *default_var:Any=__NO_DEFAULT_SENTINEL*, *deserialize_json:bool=False*, *session=None*)

classmethod set (*cls*, *key:str*, *value:Any*, *serialize_json:bool=False*, *session=None*)

classmethod delete (*cls*, *key*, *session=None*)

rotate_fernet_key (*self*)

airflow.models.xcom

Module Contents

airflow.models.xcom.**MAX_XCOM_SIZE = 49344**

airflow.models.xcom.**XCOM_RETURN_KEY = return_value**

class airflow.models.xcom.**XCom**

Bases: *airflow.models.base.Base*, airflow.utils.log.logging_mixin.LoggingMixin

Base class for XCom objects.

__tablename__ = xcom

id

key

value

timestamp

execution_date

task_id

dag_id

__table_args__

TODO: “pickling” has been deprecated and JSON is preferred. “pickling” will be removed in Airflow 2.0.

init_on_load(*self*)

__repr__(*self*)

classmethod set(*cls*, *key*, *value*, *execution_date*, *task_id*, *dag_id*, *session=None*)

Store an XCom value. TODO: “pickling” has been deprecated and JSON is preferred. “pickling” will be removed in Airflow 2.0.

Returns None

classmethod get_one(*cls*, *execution_date*, *key=None*, *task_id=None*, *dag_id=None*, *include_prior_dates=False*, *session=None*)

Retrieve an XCom value, optionally meeting certain criteria. TODO: “pickling” has been deprecated and JSON is preferred. “pickling” will be removed in Airflow 2.0.

Returns XCom value

classmethod get_many(*cls*, *execution_date*, *key=None*, *task_ids=None*, *dag_ids=None*, *include_prior_dates=False*, *limit=100*, *session=None*)

Retrieve an XCom value, optionally meeting certain criteria TODO: “pickling” has been deprecated and JSON is preferred. “pickling” will be removed in Airflow 2.0.

classmethod delete(*cls*, *xcoms*, *session=None*)

Package Contents

airflow.models.**Base :Any**

airflow.models.**ID_LEN = 250**

class airflow.models.**BaseOperator**(*task_id:str, owner:str=configuration.conf.get('operators', 'DEFAULT_OWNER'), email:Optional[str]=None, email_on_retry:bool=True, email_on_failure:bool=True, retries:int=0, retry_delay:timedelta=timedelta(seconds=300), retry_exponential_backoff:bool=False, max_retry_delay:Optional[datetime]=None, start_date:Optional[datetime]=None, end_date:Optional[datetime]=None, schedule_interval=None, depends_on_past:bool=False, wait_for_downstream:bool=False, dag:Optional[DAG]=None, params:Optional[Dict]=None, default_args:Optional[Dict]=None, priority_weight:int=1, weight_rule:str=WeightRule.DOWNSTREAM, queue:str=configuration.conf.get('celery', 'default_queue'), pool:Optional[str]=None, sla:Optional[timedelta]=None, execution_timeout:Optional[timedelta]=None, on_failure_callback:Optional[Callable]=None, on_success_callback:Optional[Callable]=None, on_retry_callback:Optional[Callable]=None, trigger_rule:str=TriggerRule.ALL_SUCCESS, resources:Optional[Dict]=None, run_as_user:Optional[str]=None, task_concurrency:Optional[int]=None, executor_config:Optional[Dict]=None, do_xcom_push:bool=True, inlets:Optional[Dict]=None, outlets:Optional[Dict]=None, *args, **kwargs*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

Abstract base class for all operators. Since operators create objects that become nodes in the dag, BaseOperator contains many recursive methods for dag crawling behavior. To derive this class, you are expected to override the constructor as well as the 'execute' method.

Operators derived from this class should perform or trigger certain tasks synchronously (wait for completion). Example of operators could be an operator that runs a Pig job (PigOperator), a sensor operator that waits for a partition to land in Hive (HiveSensorOperator), or one that moves data from Hive to MySQL (Hive2MySqlOperator). Instances of these operators (tasks) target specific operations, running specific scripts, functions or data transfers.

This class is abstract and shouldn't be instantiated. Instantiating a class derived from this one results in the creation of a task object, which ultimately becomes a node in DAG objects. Task dependencies should be set by using the set_upstream and/or set_downstream methods.

Parameters

- **task_id** (*str*) – a unique, meaningful id for the task
- **owner** (*str*) – the owner of the task, using the unix username is recommended
- **retries** (*int*) – the number of retries that should be performed before failing the task
- **retry_delay** (*datetime.timedelta*) – delay between retries
- **retry_exponential_backoff** (*bool*) – allow progressive longer waits between retries by using exponential backoff algorithm on retry delay (delay will be converted into seconds)
- **max_retry_delay** (*datetime.timedelta*) – maximum delay interval between retries
- **start_date** (*datetime.datetime*) – The `start_date` for the task, determines the `execution_date` for the first task instance. The best practice is to have the start_date

rounded to your DAG's `schedule_interval`. Daily jobs have their start_date some day at 00:00:00, hourly jobs have their start_date at 00:00 of a specific hour. Note that Airflow simply looks at the latest `execution_date` and adds the `schedule_interval` to determine the next `execution_date`. It is also very important to note that different tasks' dependencies need to line up in time. If task A depends on task B and their start_date are offset in a way that their execution_date don't line up, A's dependencies will never be met. If you are looking to delay a task, for example running a daily task at 2AM, look into the `TimeSensor` and `TimeDeltaSensor`. We advise against using dynamic `start_date` and recommend using fixed ones. Read the FAQ entry about start_date for more information.

- **end_date** (*datetime.datetime*) – if specified, the scheduler won't go beyond this date
- **depends_on_past** (*bool*) – when set to true, task instances will run sequentially while relying on the previous task's schedule to succeed. The task instance for the start_date is allowed to run.
- **wait_for_downstream** (*bool*) – when set to true, an instance of task X will wait for tasks immediately downstream of the previous instance of task X to finish successfully before it runs. This is useful if the different instances of a task X alter the same asset, and this asset is used by tasks downstream of task X. Note that depends_on_past is forced to True wherever wait_for_downstream is used.
- **queue** (*str*) – which queue to target when running this job. Not all executors implement queue management, the CeleryExecutor does support targeting specific queues.
- **dag** (`airflow.models.DAG`) – a reference to the dag the task is attached to (if any)
- **priority_weight** (*int*) – priority weight of this task against other task. This allows the executor to trigger higher priority tasks before others when things get backed up. Set priority_weight as a higher number for more important tasks.
- **weight_rule** (*str*) – weighting method used for the effective total priority weight of the task. Options are: `{ downstream | upstream | absolute }` default is `downstream` When set to `downstream` the effective weight of the task is the aggregate sum of all downstream descendants. As a result, upstream tasks will have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and desire to have all upstream tasks to complete for all runs before each dag can continue processing downstream tasks. When set to `upstream` the effective weight is the aggregate sum of all upstream ancestors. This is the opposite where downtream tasks have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and prefer to have each dag complete before starting upstream tasks of other dags. When set to `absolute`, the effective weight is the exact `priority_weight` specified without additional weighting. You may want to do this when you know exactly what priority weight each task should have. Additionally, when set to `absolute`, there is bonus effect of significantly speeding up the task creation process as for very large DAGS. Options can be set as string or using the constants defined in the static class `airflow.utils.WeightRule`
- **pool** (*str*) – the slot pool this task should run in, slot pools are a way to limit concurrency for certain tasks
- **sla** (*datetime.timedelta*) – time by which the job is expected to succeed. Note that this represents the `timedelta` after the period is closed. For example if you set an SLA of 1 hour, the scheduler would send an email soon after 1:00AM on the `2016-01-02` if the `2016-01-01` instance has not succeeded yet. The scheduler pays special attention for jobs with an SLA and sends alert emails for sla misses. SLA misses are also recorded in the database for future reference. All tasks that share the same SLA time get bundled in a single email, sent soon after that time. SLA notification are sent once and only once for each task instance.

- **execution_timeout** (*datetime.timedelta*) – max time allowed for the execution of this task instance, if it goes beyond it will raise and fail.
- **on_failure_callback** (*callable*) – a function to be called when a task instance of this task fails. a context dictionary is passed as a single parameter to this function. Context contains references to related objects to the task instance and is documented under the macros section of the API.
- **on_retry_callback** (*callable*) – much like the `on_failure_callback` except that it is executed when retries occur.
- **on_success_callback** (*callable*) – much like the `on_failure_callback` except that it is executed when the task succeeds.
- **trigger_rule** (*str*) – defines the rule by which dependencies are applied for the task to get triggered. Options are: `{ all_success | all_failed | all_done | one_success | one_failed | none_failed | none_skipped | dummy}` default is `all_success`. Options can be set as string or using the constants defined in the static class `airflow.utils.TriggerRule`
- **resources** (*dict*) – A map of resource parameter names (the argument names of the Resources constructor) to their values.
- **run_as_user** (*str*) – unix username to impersonate while running the task
- **task_concurrency** (*int*) – When set, a task will be able to limit the concurrent runs across execution_dates
- **executor_config** (*dict*) – Additional task-level configuration parameters that are interpreted by a specific executor. Parameters are namespaced by the name of executor.

  Example: to run this task in a specific docker container through the KubernetesExecutor

  ```
  MyOperator(...,
      executor_config={
      "KubernetesExecutor":
          {"image": "myCustomDockerImage"}
          }
  )
  ```

- **do_xcom_push** (*bool*) – if True, an XCom is pushed containing the Operator's result

template_fields :Iterable[str] = []

template_ext :Iterable[str] = []

ui_color = #fff

ui_fgcolor = #000

_base_operator_shallow_copy_attrs = ['user_defined_macros', 'user_defined_filters', 'p

shallow_copy_attrs :Iterable[str] = []

operator_extra_links :Iterable[BaseOperatorLink] = []

dag
  Returns the Operator's DAG if set, otherwise raises an error

dag_id

deps
  Returns the list of dependencies for the operator. These differ from execution context dependencies in that they are specific to tasks and can be extended/overridden by subclasses.

schedule_interval
The schedule interval of the DAG always wins over individual tasks so that tasks within a DAG always line up. The task still needs a schedule_interval as it may not be attached to a DAG.

priority_weight_total

upstream_list
@property: list of tasks directly upstream

upstream_task_ids

downstream_list
@property: list of tasks directly downstream

downstream_task_ids

task_type

__eq__ (*self*, *other*)

__ne__ (*self*, *other*)

__lt__ (*self*, *other*)

__hash__ (*self*)

__rshift__ (*self*, *other*)
Implements Self >> Other == self.set_downstream(other)

If "Other" is a DAG, the DAG is assigned to the Operator.

__lshift__ (*self*, *other*)
Implements Self << Other == self.set_upstream(other)

If "Other" is a DAG, the DAG is assigned to the Operator.

__rrshift__ (*self*, *other*)
Called for [DAG] >> [Operator] because DAGs don't have __rshift__ operators.

__rlshift__ (*self*, *other*)
Called for [DAG] << [Operator] because DAGs don't have __lshift__ operators.

has_dag (*self*)
Returns True if the Operator has been assigned to a DAG.

operator_extra_link_dict (*self*)

global_operator_extra_link_dict (*self*)

pre_execute (*self*, *context*)
This hook is triggered right before self.execute() is called.

execute (*self*, *context*)
This is the main method to derive when creating an operator. Context is the same dictionary used as when rendering jinja templates.

Refer to get_template_context for more context.

post_execute (*self*, *context*, *result=None*)
This hook is triggered right after self.execute() is called. It is passed the execution context and any results returned by the operator.

on_kill (*self*)
Override this method to cleanup subprocesses when a task instance gets killed. Any use of the threading, subprocess or multiprocessing module within an operator needs to be cleaned up or it will leave ghost processes behind.

`__deepcopy__` (*self*, *memo*)
Hack sorting double chained task lists by task_id to avoid hitting max_depth on deepcopy operations.

`__getstate__` (*self*)

`__setstate__` (*self*, *state*)

`render_template_from_field` (*self*, *attr*, *content*, *context*, *jinja_env*)
Renders a template from a field. If the field is a string, it will simply render the string and return the result. If it is a collection or nested set of collections, it will traverse the structure and render all elements in it. If the field has another type, it will return it as it is.

`render_template` (*self*, *attr*, *content*, *context*)
Renders a template either from a file or directly in a field, and returns the rendered result.

`get_template_env` (*self*)

`prepare_template` (*self*)
Hook that is triggered after the templated fields get replaced by their content. If you need your operator to alter the content of the file before the template is rendered, it should override this method to do so.

`resolve_template_files` (*self*)

`clear` (*self*, *start_date=None*, *end_date=None*, *upstream=False*, *downstream=False*, *session=None*)
Clears the state of task instances associated with the task, following the parameters specified.

`get_task_instances` (*self*, *start_date=None*, *end_date=None*, *session=None*)
Get a set of task instance related to this task for a specific date range.

`get_flat_relative_ids` (*self*, *upstream=False*, *found_descendants=None*)
Get a flat list of relatives' ids, either upstream or downstream.

`get_flat_relatives` (*self*, *upstream=False*)
Get a flat list of relatives, either upstream or downstream.

`run` (*self*, *start_date=None*, *end_date=None*, *ignore_first_depends_on_past=False*, *ignore_ti_state=False*, *mark_success=False*)
Run a set of task instances for a date range.

`dry_run` (*self*)

`get_direct_relative_ids` (*self*, *upstream=False*)
Get the direct relative ids to the current task, upstream or downstream.

`get_direct_relatives` (*self*, *upstream=False*)
Get the direct relatives to the current task, upstream or downstream.

`__repr__` (*self*)

`add_only_new` (*self*, *item_set*, *item*)

`_set_relatives` (*self*, *task_or_task_list*, *upstream=False*)

`set_downstream` (*self*, *task_or_task_list*)
Set a task or a task list to be directly downstream from the current task.

`set_upstream` (*self*, *task_or_task_list*)
Set a task or a task list to be directly upstream from the current task.

`xcom_push` (*self*, *context*, *key*, *value*, *execution_date=None*)
See TaskInstance.xcom_push()

`xcom_pull` (*self*, *context*, *task_ids=None*, *dag_id=None*, *key=XCOM_RETURN_KEY*, *include_prior_dates=None*)
See TaskInstance.xcom_pull()

`extra_links` (*self*)

get_extra_links(*self*, *dttm*, *link_name*)
For an operator, gets the URL that the external links specified in *extra_links* should point to. :raise ValueError: The error message of a ValueError will be passed on through to the fronted to show up as a tooltip on the disabled link :param dttm: The datetime parsed execution date for the URL being searched for :param link_name: The name of the link we're looking for the URL for. Should be one of the options specified in *extra_links* :return: A URL

class airflow.models.**Connection**(*conn_id=None*, *conn_type=None*, *host=None*, *login=None*, *password=None*, *schema=None*, *port=None*, *extra=None*, *uri=None*)
Bases: *airflow.models.base.Base*, airflow.LoggingMixin

Placeholder to store information about different database instances connection information. The idea here is that scripts use references to database instances (conn_id) instead of hard coding hostname, logins and passwords when using operators or hooks.

__tablename__ = connection

id

conn_id

conn_type

host

schema

login

_password

port

is_encrypted

is_extra_encrypted

_extra

_types = [['docker', 'Docker Registry'], ['fs', 'File (path)'], ['ftp', 'FTP'], ['goog

password

extra

extra_dejson
Returns the extra property by deserializing json.

parse_from_uri(*self*, *uri*)

get_password(*self*)

set_password(*self*, *value*)

get_extra(*self*)

set_extra(*self*, *value*)

rotate_fernet_key(*self*)

get_hook(*self*)

__repr__(*self*)

debug_info(*self*)

class airflow.models.**DAG**(*dag_id:str, description:str=", schedule_interval:Optional[ScheduleInterval]=timedelta(days=1), start_date:Optional[datetime]=None, end_date:Optional[datetime]=None, full_filepath:Optional[str]=None, template_searchpath:Optional[Union[str, Iterable[str]]]=None, template_undefined:Type[jinja2.Undefined]=jinja2.Undefined, user_defined_macros:Optional[Dict]=None, user_defined_filters:Optional[Dict]=None, default_args:Optional[Dict]=None, concurrency:int=configuration.conf.getint('core', 'dag_concurrency'), max_active_runs:int=configuration.conf.getint('core', 'max_active_runs_per_dag'), dagrun_timeout:Optional[timedelta]=None, sla_miss_callback:Optional[Callable]=None, default_view:Optional[str]=None, orientation:str=configuration.conf.get('webserver', 'dag_orientation'), catchup:bool=configuration.conf.getboolean('scheduler', 'catchup_by_default'), on_success_callback:Optional[Callable]=None, on_failure_callback:Optional[Callable]=None, doc_md:Optional[str]=None, params:Optional[Dict]=None, access_control:Optional[Dict]=None*)

Bases: `airflow.dag.base_dag.BaseDag`, `airflow.utils.log.logging_mixin.LoggingMixin`

A dag (directed acyclic graph) is a collection of tasks with directional dependencies. A dag also has a schedule, a start date and an end date (optional). For each schedule, (say daily or hourly), the DAG needs to run each individual tasks as their dependencies are met. Certain tasks have the property of depending on their own past, meaning that they can't run until their previous schedule (and upstream tasks) are completed.

DAGs essentially act as namespaces for tasks. A task_id can only be added once to a DAG.

Parameters

- **dag_id** (*str*) – The id of the DAG
- **description** (*str*) – The description for the DAG to e.g. be shown on the webserver
- **schedule_interval** (*datetime.timedelta or dateutil.relativedelta.relativedelta or str that acts as a cron expression*) – Defines how often that DAG runs, this timedelta object gets added to your latest task instance's execution_date to figure out the next schedule
- **start_date** (*datetime.datetime*) – The timestamp from which the scheduler will attempt to backfill
- **end_date** (*datetime.datetime*) – A date beyond which your DAG won't run, leave to None for open ended scheduling
- **template_searchpath** (*str or list[str]*) – This list of folders (non relative) defines where jinja will look for your templates. Order matters. Note that jinja/airflow includes the path of your DAG file by default
- **template_undefined** (*jinja2.Undefined*) – Template undefined type.
- **user_defined_macros** (*dict*) – a dictionary of macros that will be exposed in your jinja templates. For example, passing `dict(foo='bar')` to this argument allows you to `{{ foo }}` in all jinja templates related to this DAG. Note that you can pass any type of object here.
- **user_defined_filters** (*dict*) – a dictionary of filters that will be exposed in your jinja templates. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to `{{ 'world' | hello }}` in all jinja templates related to this DAG.

- **default_args** (*dict*) – A dictionary of default parameters to be used as constructor keyword parameters when initialising operators. Note that operators have the same hook, and precede those defined here, meaning that if your dict contains *'depends_on_past': True* here and *'depends_on_past': False* in the operator's call *default_args*, the actual value will be *False*.
- **params** (*dict*) – a dictionary of DAG level parameters that are made accessible in templates, namespaced under *params*. These params can be overridden at the task level.
- **concurrency** (*int*) – the number of task instances allowed to run concurrently
- **max_active_runs** (*int*) – maximum number of active DAG runs, beyond this number of DAG runs in a running state, the scheduler won't create new active DAG runs
- **dagrun_timeout** (*datetime.timedelta*) – specify how long a DagRun should be up before timing out / failing, so that new DagRuns can be created. The timeout is only enforced for scheduled DagRuns, and only once the # of active DagRuns == max_active_runs.
- **sla_miss_callback** (*types.FunctionType*) – specify a function to call when reporting SLA timeouts.
- **default_view** (*str*) – Specify DAG default view (tree, graph, duration, gantt, landing_times)
- **orientation** (*str*) – Specify DAG orientation in graph view (LR, TB, RL, BT)
- **catchup** (*bool*) – Perform scheduler catchup (or only run latest)? Defaults to True
- **on_failure_callback** (*callable*) – A function to be called when a DagRun of this dag fails. A context dictionary is passed as a single parameter to this function.
- **on_success_callback** (*callable*) – Much like the `on_failure_callback` except that it is executed when the dag succeeds.
- **access_control** (*dict*) – Specify optional DAG-level permissions, e.g., "{'role1': {'can_dag_read'}, 'role2': {'can_dag_read', 'can_dag_edit'}}"

dag_id

full_filepath

concurrency

access_control

description

pickle_id

tasks

task_ids

filepath
File location of where the dag object is instantiated

folder
Folder location of where the dag object is instantiated

owner
Return list of all owners found in DAG tasks.

Returns Comma separated list of owners in DAG tasks

Return type str

concurrency_reached
Returns a boolean indicating whether the concurrency limit for this DAG has been reached

is_paused
Returns a boolean indicating whether this DAG is paused

latest_execution_date
Returns the latest date for which at least one dag run exists

subdags
Returns a list of the subdag objects associated to this DAG

roots

__repr__(*self*)

__eq__(*self*, *other*)

__ne__(*self*, *other*)

__lt__(*self*, *other*)

__hash__(*self*)

__enter__(*self*)

__exit__(*self*, *_type*, *_value*, *_tb*)

get_default_view(*self*)
This is only there for backward compatible jinja2 templates

date_range(*self*, *start_date*, *num=None*, *end_date=timezone.utcnow()*)

is_fixed_time_schedule(*self*)
Figures out if the DAG schedule has a fixed time (e.g. 3 AM).

Returns True if the schedule has a fixed time, False if not.

following_schedule(*self*, *dttm*)
Calculates the following schedule for this dag in UTC.

Parameters **dttm** – utc datetime

Returns utc datetime

previous_schedule(*self*, *dttm*)
Calculates the previous schedule for this dag in UTC

Parameters **dttm** – utc datetime

Returns utc datetime

get_run_dates(*self*, *start_date*, *end_date=None*)
Returns a list of dates between the interval received as parameter using this dag's schedule interval. Returned dates can be used for execution dates.

Parameters

- **start_date** (*datetime*) – the start date of the interval
- **end_date** (*datetime*) – the end date of the interval, defaults to timezone.utcnow()

Returns a list of dates within the interval following the dag's schedule

Return type list

normalize_schedule(*self*, *dttm*)
Returns dttm + interval unless dttm is first interval then it returns dttm

get_last_dagrun(*self*, *session=None*, *include_externally_triggered=False*)

_get_concurrency_reached(*self*, *session=None*)

_get_is_paused(*self*, *session=None*)

handle_callback (*self*, *dagrun*, *success=True*, *reason=None*, *session=None*)
Triggers the appropriate callback depending on the value of success, namely the on_failure_callback or on_success_callback. This method gets the context of a single TaskInstance part of this DagRun and passes that to the callable along with a 'reason', primarily to differentiate DagRun failures.

Parameters

- **dagrun** – DagRun object
- **success** – Flag to specify if failure or success callback should be called
- **reason** – Completion reason
- **session** – Database session

get_active_runs (*self*)
Returns a list of dag run execution dates currently running

Returns List of execution dates

get_num_active_runs (*self*, *external_trigger=None*, *session=None*)
Returns the number of active "running" dag runs

Parameters

- **external_trigger** (*bool*) – True for externally triggered active dag runs
- **session** –

Returns number greater than 0 for active dag runs

get_dagrun (*self*, *execution_date*, *session=None*)
Returns the dag run for a given execution date if it exists, otherwise none.

Parameters

- **execution_date** – The execution date of the DagRun to find.
- **session** –

Returns The DagRun if found, otherwise None.

_get_latest_execution_date (*self*, *session=None*)

resolve_template_files (*self*)

get_template_env (*self*)
Returns a jinja2 Environment while taking into account the DAGs template_searchpath, user_defined_macros and user_defined_filters

set_dependency (*self*, *upstream_task_id*, *downstream_task_id*)
Simple utility method to set dependency between two tasks that already have been added to the DAG using add_task()

get_task_instances (*self*, *start_date=None*, *end_date=None*, *state=None*, *session=None*)

topological_sort (*self*)
Sorts tasks in topographical order, such that a task comes after any of its upstream dependencies.

Heavily inspired by: http://blog.jupo.org/2012/04/06/topological-sorting-acyclic-directed-graphs/

Returns list of tasks in topological order

set_dag_runs_state (*self*, *state=State.RUNNING*, *session=None*, *start_date=None*, *end_date=None*)

clear (*self*, *start_date=None*, *end_date=None*, *only_failed=False*, *only_running=False*, *confirm_prompt=False*, *include_subdags=True*, *include_parentdag=True*, *reset_dag_runs=True*, *dry_run=False*, *session=None*, *get_tis=False*)
Clears a set of task instances associated with the current dag for a specified date range.

classmethod clear_dags(*cls*, *dags*, *start_date=None*, *end_date=None*, *only_failed=False*, *only_running=False*, *confirm_prompt=False*, *include_subdags=True*, *include_parentdag=False*, *reset_dag_runs=True*, *dry_run=False*)

__deepcopy__(*self*, *memo*)

sub_dag(*self*, *task_regex*, *include_downstream=False*, *include_upstream=True*)
Returns a subset of the current dag as a deep copy of the current dag based on a regex that should match one or many tasks, and includes upstream and downstream neighbours based on the flag passed.

has_task(*self*, *task_id*)

get_task(*self*, *task_id*)

pickle_info(*self*)

pickle(*self*, *session=None*)

tree_view(*self*)
Shows an ascii tree representation of the DAG

add_task(*self*, *task*)
Add a task to the DAG

Parameters **task** (`task`) – the task you want to add

add_tasks(*self*, *tasks*)
Add a list of tasks to the DAG

Parameters **tasks** (`list of tasks`) – a lit of tasks you want to add

run(*self*, *start_date=None*, *end_date=None*, *mark_success=False*, *local=False*, *executor=None*, *donot_pickle=configuration.conf.getboolean('core', 'donot_pickle')*, *ignore_task_deps=False*, *ignore_first_depends_on_past=False*, *pool=None*, *delay_on_limit_secs=1.0*, *verbose=False*, *conf=None*, *rerun_failed_tasks=False*, *run_backwards=False*)
Runs the DAG.

Parameters

- **start_date** (`datetime.datetime`) – the start date of the range to run
- **end_date** (`datetime.datetime`) – the end date of the range to run
- **mark_success** (`bool`) – True to mark jobs as succeeded without running them
- **local** (`bool`) – True to run the tasks using the LocalExecutor
- **executor** (`airflow.executor.BaseExecutor`) – The executor instance to run the tasks
- **donot_pickle** (`bool`) – True to avoid pickling DAG object and send to workers
- **ignore_task_deps** (`bool`) – True to skip upstream tasks
- **ignore_first_depends_on_past** (`bool`) – True to ignore depends_on_past dependencies for the first set of tasks only
- **pool** (`str`) – Resource pool to use
- **delay_on_limit_secs** (`float`) – Time in seconds to wait before next attempt to run dag run when max_active_runs limit has been reached
- **verbose** (`bool`) – Make logging output more verbose
- **conf** (`dict`) – user defined dictionary passed from CLI
- **rerun_failed_tasks** –
- **run_backwards** –

Type bool

Type bool

`cli`(*self*)
Exposes a CLI specific to this DAG

`create_dagrun`(*self*, *run_id*, *state*, *execution_date=None*, *start_date=None*, *external_trigger=False*, *conf=None*, *session=None*)
Creates a dag run from this dag including the tasks associated with this dag. Returns the dag run.

Parameters

- **`run_id`** (`str`) – defines the the run id for this dag run
- **`execution_date`** (`datetime.datetime`) – the execution date of this dag run
- **`state`** (`airflow.utils.state.State`) – the state of the dag run
- **`start_date`** (`datetime`) – the date this dag run should be evaluated
- **`external_trigger`** (`bool`) – whether this dag run is externally triggered
- **`session`** (`sqlalchemy.orm.session.Session`) – database session

`sync_to_db`(*self*, *owner=None*, *sync_time=None*, *session=None*)
Save attributes about this DAG to the DB. Note that this method can be called for both DAGs and SubDAGs. A SubDag is actually a SubDagOperator.

Parameters

- **`dag`** (`airflow.models.DAG`) – the DAG object to save to the DB
- **`sync_time`** (`datetime`) – The time that the DAG should be marked as sync'ed

Returns None

`static deactivate_unknown_dags`(*active_dag_ids*, *session=None*)
Given a list of known DAGs, deactivate any other DAGs that are marked as active in the ORM

Parameters **`active_dag_ids`** (`list[unicode]`) – list of DAG IDs that are active

Returns None

`static deactivate_stale_dags`(*expiration_date*, *session=None*)
Deactivate any DAGs that were last touched by the scheduler before the expiration date. These DAGs were likely deleted.

Parameters **`expiration_date`** (`datetime`) – set inactive DAGs that were touched before this time

Returns None

`static get_num_task_instances`(*dag_id*, *task_ids=None*, *states=None*, *session=None*)
Returns the number of task instances in the given DAG.

Parameters

- **`session`** – ORM session
- **`dag_id`** (`unicode`) – ID of the DAG to get the task concurrency of
- **`task_ids`** (`list[unicode]`) – A list of valid task IDs for the given DAG
- **`states`** (`list[state]`) – A list of states to filter by if supplied

Returns The number of running tasks

Return type int

`test_cycle`(*self*)
Check to see if there are any cycles in the DAG. Returns False if no cycle found, otherwise raises exception.

`_test_cycle_helper`(*self*, *visit_map*, *task_id*)
Checks if a cycle exists from the input task using DFS traversal

`class` `airflow.models.`**`DagModel`**
Bases: *`airflow.models.base.Base`*

`__tablename__ = dag`
These items are stored in the database for state related information

`dag_id`

`is_paused_at_creation`

`is_paused`

`is_subdag`

`is_active`

`last_scheduler_run`

`last_pickled`

`last_expired`

`scheduler_lock`

`pickle_id`

`fileloc`

`owners`

`description`

`default_view`

`schedule_interval`

`timezone`

`safe_dag_id`

`__repr__`(*self*)

`static get_dagmodel`(*dag_id*, *session=None*)

`classmethod get_current`(*cls*, *dag_id*, *session=None*)

`get_default_view`(*self*)

`get_last_dagrun`(*self*, *session=None*, *include_externally_triggered=False*)

`get_dag`(*self*)

`create_dagrun`(*self*, *run_id*, *state*, *execution_date*, *start_date=None*, *external_trigger=False*, *conf=None*, *session=None*)
Creates a dag run from this dag including the tasks associated with this dag. Returns the dag run.

Parameters

- **`run_id`** (*`str`*) – defines the the run id for this dag run
- **`execution_date`** (*`datetime.datetime`*) – the execution date of this dag run
- **`state`** (*`airflow.utils.state.State`*) – the state of the dag run
- **`start_date`** (*`datetime.datetime`*) – the date this dag run should be evaluated
- **`external_trigger`** (*`bool`*) – whether this dag run is externally triggered
- **`session`** (*`sqlalchemy.orm.session.Session`*) – database session

set_is_paused(*self*, *is_paused:bool*, *including_subdags:bool=True*, *session=None*)
Pause/Un-pause a DAG.

Parameters

- **is_paused** – Is the DAG paused
- **including_subdags** – whether to include the DAG's subdags
- **session** – session

class airflow.models.**DagBag**(*dag_folder=None*, *executor=None*, *include_examples=configuration.conf.getboolean('core', 'LOAD_EXAMPLES')*, *safe_mode=configuration.conf.getboolean('core', 'DAG_DISCOVERY_SAFE_MODE')*)
Bases: `airflow.dag.base_dag.BaseDagBag`, `airflow.utils.log.logging_mixin.LoggingMixin`

A dagbag is a collection of dags, parsed out of a folder tree and has high level configuration settings, like what database to use as a backend and what executor to use to fire off tasks. This makes it easier to run distinct environments for say production and development, tests, or for different teams or security profiles. What would have been system level settings are now dagbag level so that one system can run multiple, independent settings sets.

Parameters

- **dag_folder** (*unicode*) – the folder to scan to find DAGs
- **executor** – the executor to use when executing task instances in this DagBag
- **include_examples** (*bool*) – whether to include the examples that ship with airflow or not
- **has_logged** – an instance boolean that gets flipped from False to True after a file has been skipped. This is to prevent overloading the user with logging messages about skipped files. Therefore only once per DagBag is a file logged being skipped.

CYCLE_NEW = 0

CYCLE_IN_PROGRESS = 1

CYCLE_DONE = 2

dag_ids

size(*self*)

Returns the amount of dags contained in this dagbag

get_dag(*self*, *dag_id*)
Gets the DAG out of the dictionary, and refreshes it if expired

process_file(*self*, *filepath*, *only_if_updated=True*, *safe_mode=True*)
Given a path to a python module or zip file, this method imports the module and look for dag objects within it.

kill_zombies(*self*, *zombies*, *session=None*)
Fail given zombie tasks, which are tasks that haven't had a heartbeat for too long, in the current DagBag.

Parameters

- **zombies** (*airflow.utils.dag_processing.SimpleTaskInstance*) – zombie task instances to kill.
- **session** (*sqlalchemy.orm.session.Session*) – DB session.

bag_dag(*self*, *dag*, *parent_dag*, *root_dag*)
Adds the DAG into the bag, recurses into sub dags. Throws AirflowDagCycleException if a cycle is detected in this dag or its subdags

collect_dags(*self*, *dag_folder=None*, *only_if_updated=True*, *include_examples=configuration.conf.getboolean('core', 'LOAD_EXAMPLES')*, *safe_mode=configuration.conf.getboolean('core', 'DAG_DISCOVERY_SAFE_MODE')*)

Given a file path or a folder, this method looks for python modules, imports them and adds them to the dagbag collection.

Note that if a `.airflowignore` file is found while processing the directory, it will behave much like a `.gitignore`, ignoring files that match any of the regex patterns specified in the file.

Note: The patterns in .airflowignore are treated as un-anchored regexes, not shell-like glob patterns.

dagbag_report(*self*)

Prints a report around DagBag loading stats

class airflow.models.**DagPickle**(*dag*)

Bases: *airflow.models.base.Base*

Dags can originate from different places (user repos, master repo, ...) and also get executed in different places (different executors). This object represents a version of a DAG and becomes a source of truth for a BackfillJob execution. A pickle is a native python serialized object, and in this case gets stored in the database for the duration of the job.

The executors pick up the DagPickle id and read the dag definition from the database.

id

pickle

created_dttm

pickle_hash

__tablename__ = dag_pickle

class airflow.models.**DagRun**

Bases: *airflow.models.base.Base*, `airflow.utils.log.logging_mixin.LoggingMixin`

DagRun describes an instance of a Dag. It can be created by the scheduler (for regular runs) or by an external trigger

__tablename__ = dag_run

ID_PREFIX = scheduled__

ID_FORMAT_PREFIX

id

dag_id

execution_date

start_date

end_date

_state

run_id

external_trigger

conf

dag

__table_args__

state

is_backfill

`__repr__`(*self*)

`get_state`(*self*)

`set_state`(*self*, *state*)

`classmethod id_for_date`(*cls*, *date*, *prefix=ID_FORMAT_PREFIX*)

`refresh_from_db`(*self*, *session=None*)
Reloads the current dagrun from the database :param session: database session

`static find`(*dag_id=None*, *run_id=None*, *execution_date=None*, *state=None*, *external_trigger=None*, *no_backfills=False*, *session=None*)
Returns a set of dag runs for the given search criteria.

Parameters

- **`dag_id`** (`int, list`) – the dag_id to find dag runs for
- **`run_id`** (`str`) – defines the the run id for this dag run
- **`execution_date`** (`datetime.datetime`) – the execution date
- **`state`** (`str`) – the state of the dag run
- **`external_trigger`** (`bool`) – whether this dag run is externally triggered
- **`no_backfills`** (`bool`) – return no backfills (True), return all (False). Defaults to False
- **`session`** (`sqlalchemy.orm.session.Session`) – database session

`get_task_instances`(*self*, *state=None*, *session=None*)
Returns the task instances for this dag run

`get_task_instance`(*self*, *task_id*, *session=None*)
Returns the task instance specified by task_id for this dag run

Parameters **`task_id`** – the task id

`get_dag`(*self*)
Returns the Dag associated with this DagRun.

Returns DAG

`get_previous_dagrun`(*self*, *state:str=None*, *session:Session=None*)
The previous DagRun, if there is one

`get_previous_scheduled_dagrun`(*self*, *session=None*)
The previous, SCHEDULED DagRun, if there is one

`update_state`(*self*, *session=None*)
Determines the overall state of the DagRun based on the state of its TaskInstances.

Returns State

`_emit_duration_stats_for_finished_state`(*self*)

`verify_integrity`(*self*, *session=None*)
Verifies the DagRun by checking for removed tasks or tasks that are not in the database yet. It will set state to removed or add the task if required.

`static get_run`(*session*, *dag_id*, *execution_date*)

Parameters

- **`dag_id`** (`unicode`) – DAG ID
- **`execution_date`** (`datetime`) – execution date

Returns DagRun corresponding to the given dag_id and execution date if one exists. None otherwise.

Return type *airflow.models.DagRun*

classmethod get_latest_runs(*cls*, *session*)
Returns the latest DagRun for each DAG.

class airflow.models.**KubeWorkerIdentifier**
Bases: *airflow.models.base.Base*

__tablename__ = kube_worker_uuid

one_row_id

worker_uuid

static get_or_create_current_kube_worker_uuid(*session=None*)

static checkpoint_kube_worker_uuid(*worker_uuid*, *session=None*)

class airflow.models.**KubeResourceVersion**
Bases: *airflow.models.base.Base*

__tablename__ = kube_resource_version

one_row_id

resource_version

static get_current_resource_version(*session=None*)

static checkpoint_resource_version(*resource_version*, *session=None*)

static reset_resource_version(*session=None*)

class airflow.models.**Log**(*event*, *task_instance*, *owner=None*, *extra=None*, ***kwargs*)
Bases: *airflow.models.base.Base*

Used to actively log events to the database

__tablename__ = log

id

dttm

dag_id

task_id

event

execution_date

owner

extra

__table_args__

class airflow.models.**Pool**
Bases: *airflow.models.base.Base*

__tablename__ = slot_pool

id

pool

slots

description

default_pool_name = not_pooled

__repr__(*self*)

static default_pool_open_slots(*session*)

to_json(*self*)

used_slots(*self*, *session*)
Returns the number of slots used at the moment

queued_slots(*self*, *session*)
Returns the number of slots used at the moment

open_slots(*self*, *session*)
Returns the number of slots open at the moment

class airflow.models.**TaskFail**(*task*, *execution_date*, *start_date*, *end_date*)
Bases: *airflow.models.base.Base*

TaskFail tracks the failed run durations of each task instance.

__tablename__ = task_fail

id

task_id

dag_id

execution_date

start_date

end_date

duration

__table_args__

class airflow.models.**SkipMixin**
Bases: airflow.utils.log.logging_mixin.LoggingMixin

skip(*self*, *dag_run*, *execution_date*, *tasks*, *session=None*)
Sets tasks instances to skipped from the same dag run.

Parameters

- **dag_run** – the DagRun for which to set the tasks to skipped
- **execution_date** – execution_date
- **tasks** – tasks to skip (not task_ids)
- **session** – db session to use

class airflow.models.**SlaMiss**
Bases: *airflow.models.base.Base*

Model that stores a history of the SLA that have been missed. It is used to keep track of SLA failures over time and to avoid double triggering alert emails.

__tablename__ = sla_miss

task_id

dag_id

execution_date

email_sent

timestamp

description

notification_sent

`__table_args__`

`__repr__`(*self*)

`airflow.models.`**`clear_task_instances`**(*tis*, *session*, *activate_dag_runs=True*, *dag=None*)
`Clears a set of task instances, but makes sure the running ones get killed.`

Parameters

- **`tis`** – a list of task instances
- **`session`** – current session
- **`activate_dag_runs`** – flag to check for active dag run
- **`dag`** – DAG object

`class` `airflow.models.`**`TaskInstance`**(*task*, *execution_date*, *state=None*)

Bases: *`airflow.models.base.Base`*, `airflow.utils.log.logging_mixin.LoggingMixin`

Task instances store the state of a task instance. This table is the authority and single source of truth around what tasks have run and the state they are in.

The SqlAlchemy model doesn't have a SqlAlchemy foreign key to the task or dag model deliberately to have more control over transactions.

Database transactions on this table should insure double triggers and any confusion around what task instances are or aren't ready to run even while multiple schedulers may be firing task instances.

`__tablename__ = task_instance`

`task_id`

`dag_id`

`execution_date`

`start_date`

`end_date`

`duration`

`state`

`_try_number`

`max_tries`

`hostname`

`unixname`

`job_id`

`pool`

`queue`

`priority_weight`

`operator`

`queued_dttm`

`pid`

`executor_config`

`__table_args__`

try_number
Return the try number that this task number will be when it is actually run.

If the TI is currently running, this will match the column in the databse, in all othercases this will be incremenetd

next_try_number

log_filepath

log_url

mark_success_url

key
Returns a tuple that identifies the task instance uniquely

is_premature
Returns whether a task is in UP_FOR_RETRY state and its retry interval has elapsed.

previous_ti
The task instance for the task that ran before this task instance.

previous_ti_success
The ti from prior succesful dag run for this task, by execution date.

previous_execution_date_success
The execution date from property previous_ti_success.

previous_start_date_success
The start date from property previous_ti_success.

init_on_load(*self*)
Initialize the attributes that aren't stored in the DB.

command(*self*, *mark_success=False*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)
Returns a command that can be executed anywhere where airflow is installed. This command is part of the message sent to executors by the orchestrator.

command_as_list(*self*, *mark_success=False*, *ignore_all_deps=False*, *ignore_task_deps=False*, *ignore_depends_on_past=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)
Returns a command that can be executed anywhere where airflow is installed. This command is part of the message sent to executors by the orchestrator.

static generate_command(*dag_id*, *task_id*, *execution_date*, *mark_success=False*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *file_path=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)
Generates the shell command required to execute this task instance.

Parameters

- **dag_id** (*unicode*) – DAG ID
- **task_id** (*unicode*) – Task ID
- **execution_date** (*datetime.datetime*) – Execution date for the task
- **mark_success** (*bool*) – Whether to mark the task as successful
- **ignore_all_deps** (*bool*) – Ignore all ignorable dependencies. Overrides the other ignore_* parameters.

- **`ignore_depends_on_past`** (*`bool`*) – Ignore depends_on_past parameter of DAGs (e.g. for Backfills)
- **`ignore_task_deps`** (*`bool`*) – Ignore task-specific dependencies such as depends_on_past and trigger rule
- **`ignore_ti_state`** (*`bool`*) – Ignore the task instance's previous failure/success
- **`local`** (*`bool`*) – Whether to run the task locally
- **`pickle_id`** (*`unicode`*) – If the DAG was serialized to the DB, the ID associated with the pickled DAG
- **`file_path`** – path to the file containing the DAG definition
- **`raw`** – raw mode (needs more details)
- **`job_id`** – job ID (needs more details)
- **`pool`** (*`unicode`*) – the Airflow pool that the task should run in
- **`cfg_path`** (*`str`*) – the Path to the configuration file

Returns shell command that can be used to run the task instance

`current_state`(*self*, *session=None*)
Get the very latest state from the database, if a session is passed, we use and looking up the state becomes part of the session, otherwise a new session is used.

`error`(*self*, *session=None*)
Forces the task instance's state to FAILED in the database.

`refresh_from_db`(*self*, *session=None*, *lock_for_update=False*)
Refreshes the task instance from the database based on the primary key

Parameters **`lock_for_update`** – if True, indicates that the database should lock the TaskInstance (issuing a FOR UPDATE clause) until the session is committed.

`clear_xcom_data`(*self*, *session=None*)
Clears all XCom data from the database for the task instance

`set_state`(*self*, *state*, *session=None*, *commit=True*)

`are_dependents_done`(*self*, *session=None*)
Checks whether the dependents of this task instance have all succeeded. This is meant to be used by wait_for_downstream.

This is useful when you do not want to start processing the next schedule of a task until the dependents are done. For instance, if the task DROPs and recreates a table.

`_get_previous_ti`(*self*, *state:str=None*, *session:Session=None*)

`are_dependencies_met`(*self*, *dep_context=None*, *session=None*, *verbose=False*)
Returns whether or not all the conditions are met for this task instance to be run given the context for the dependencies (e.g. a task instance being force run from the UI will ignore some dependencies).

Parameters

- **`dep_context`** (*`DepContext`*) – The execution context that determines the dependencies that should be evaluated.
- **`session`** (*`sqlalchemy.orm.session.Session`*) – database session
- **`verbose`** (*`bool`*) – whether log details on failed dependencies on info or debug log level

`get_failed_dep_statuses`(*self*, *dep_context=None*, *session=None*)

`__repr__`(*self*)

next_retry_datetime(*self*)
Get datetime of the next retry if the task instance fails. For exponential backoff, retry_delay is used as base and will be converted to seconds.

ready_for_retry(*self*)
Checks on whether the task instance is in the right state and timeframe to be retried.

pool_full(*self*, *session*)
Returns a boolean as to whether the slot pool has room for this task to run

get_dagrun(*self*, *session*)
Returns the DagRun for this TaskInstance

Parameters **session** –

Returns DagRun

_check_and_change_state_before_execution(*self*, *verbose=True*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *mark_success=False*, *test_mode=False*, *job_id=None*, *pool=None*, *session=None*)
Checks dependencies and then sets state to RUNNING if they are met. Returns True if and only if state is set to RUNNING, which implies that task should be executed, in preparation for _run_raw_task

Parameters

- **verbose** (*bool*) – whether to turn on more verbose logging
- **ignore_all_deps** (*bool*) – Ignore all of the non-critical dependencies, just runs
- **ignore_depends_on_past** (*bool*) – Ignore depends_on_past DAG attribute
- **ignore_task_deps** (*bool*) – Don't check the dependencies of this TI's task
- **ignore_ti_state** (*bool*) – Disregards previous task instance state
- **mark_success** (*bool*) – Don't run the task, mark its state as success
- **test_mode** (*bool*) – Doesn't record success or failure in the DB
- **pool** (*str*) – specifies the pool to use to run the task instance

Returns whether the state was changed to running or not

Return type bool

_run_raw_task(*self*, *mark_success=False*, *test_mode=False*, *job_id=None*, *pool=None*, *session=None*)
Immediately runs the task (without checking or changing db state before execution) and then sets the appropriate final state after completion and runs any post-execute callbacks. Meant to be called only after another function changes the state to running.

Parameters

- **mark_success** (*bool*) – Don't run the task, mark its state as success
- **test_mode** (*bool*) – Doesn't record success or failure in the DB
- **pool** (*str*) – specifies the pool to use to run the task instance

run(*self*, *verbose=True*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *mark_success=False*, *test_mode=False*, *job_id=None*, *pool=None*, *session=None*)

dry_run(*self*)

_handle_reschedule(*self*, *actual_start_date*, *reschedule_exception*, *test_mode=False*, *context=None*, *session=None*)

handle_failure(*self*, *error*, *test_mode=False*, *context=None*, *session=None*)

is_eligible_to_retry(*self*)
Is task instance is eligible for retry

get_template_context(*self*, *session=None*)

overwrite_params_with_dag_run_conf(*self*, *params*, *dag_run*)

render_templates(*self*)

email_alert(*self*, *exception*)

set_duration(*self*)

xcom_push(*self*, *key*, *value*, *execution_date=None*)
Make an XCom available for tasks to pull.

Parameters

- **key** (*str*) – A key for the XCom
- **value** (*any pickleable object*) – A value for the XCom. The value is pickled and stored in the database.
- **execution_date** (*datetime*) – if provided, the XCom will not be visible until this date. This can be used, for example, to send a message to a task on a future date without it being immediately visible.

xcom_pull(*self*, *task_ids=None*, *dag_id=None*, *key=XCOM_RETURN_KEY*, *include_prior_dates=False*)
Pull XComs that optionally meet certain criteria.

The default value for *key* limits the search to XComs that were returned by other tasks (as opposed to those that were pushed manually). To remove this filter, pass key=None (or any desired value).

If a single task_id string is provided, the result is the value of the most recent matching XCom from that task_id. If multiple task_ids are provided, a tuple of matching values is returned. None is returned whenever no matches are found.

Parameters

- **key** (*str*) – A key for the XCom. If provided, only XComs with matching keys will be returned. The default key is 'return_value', also available as a constant XCOM_RETURN_KEY. This key is automatically given to XComs returned by tasks (as opposed to being pushed manually). To remove the filter, pass key=None.
- **task_ids** (*str or iterable of strings (representing task_ids)*) – Only XComs from tasks with matching ids will be pulled. Can pass None to remove the filter.
- **dag_id** (*str*) – If provided, only pulls XComs from this DAG. If None (default), the DAG of the calling task is used.
- **include_prior_dates** (*bool*) – If False, only XComs from the current execution_date are returned. If True, XComs from previous dates are returned as well.

get_num_running_task_instances(*self*, *session*)

init_run_context(*self*, *raw=False*)
Sets the log context.

class airflow.models.**TaskReschedule**(*task*, *execution_date*, *try_number*, *start_date*, *end_date*, *reschedule_date*)
Bases: *airflow.models.base.Base*

TaskReschedule tracks rescheduled task instances.

__tablename__ = task_reschedule

id

task_id

dag_id

execution_date

try_number

start_date

end_date

duration

reschedule_date

__table_args__

static find_for_task_instance(*task_instance*, *session*)
Returns all task reschedules for the task instance and try number, in ascending order.

Parameters **task_instance** (`airflow.models.TaskInstance`) – the task instance to find task reschedules for

class airflow.models.**Variable**
Bases: *airflow.models.base.Base*, airflow.utils.log.logging_mixin.LoggingMixin

__tablename__ = variable

__NO_DEFAULT_SENTINEL

id

key

_val

is_encrypted

val

__repr__(*self*)

get_val(*self*)

set_val(*self*, *value*)

classmethod setdefault(*cls*, *key*, *default*, *deserialize_json=False*)
Like a Python builtin dict object, setdefault returns the current value for a key, and if it isn't there, stores the default value and returns it.

Parameters

- **key** (*str*) – Dict key for this Variable
- **default** (*Mixed*) – Default value to set and return if the variable isn't already in the DB
- **deserialize_json** – Store this as a JSON encoded value in the DB and un-encode it when retrieving a value

Returns Mixed

classmethod get(*cls*, *key:str*, *default_var:Any=__NO_DEFAULT_SENTINEL*, *deserialize_json:bool=False*, *session=None*)

classmethod set(*cls*, *key:str*, *value:Any*, *serialize_json:bool=False*, *session=None*)

classmethod delete(*cls*, *key*, *session=None*)

rotate_fernet_key(*self*)

class airflow.models.**XCom**

Bases: *airflow.models.base.Base*, airflow.utils.log.logging_mixin.LoggingMixin

Base class for XCom objects.

__tablename__ = xcom

id

key

value

timestamp

execution_date

task_id

dag_id

__table_args__

TODO: "pickling" has been deprecated and JSON is preferred. "pickling" will be removed in Airflow 2.0.

init_on_load(*self*)

__repr__(*self*)

classmethod set(*cls*, *key*, *value*, *execution_date*, *task_id*, *dag_id*, *session=None*)

Store an XCom value. TODO: "pickling" has been deprecated and JSON is preferred. "pickling" will be removed in Airflow 2.0.

Returns None

classmethod get_one(*cls*, *execution_date*, *key=None*, *task_id=None*, *dag_id=None*, *include_prior_dates=False*, *session=None*)

Retrieve an XCom value, optionally meeting certain criteria. TODO: "pickling" has been deprecated and JSON is preferred. "pickling" will be removed in Airflow 2.0.

Returns XCom value

classmethod get_many(*cls*, *execution_date*, *key=None*, *task_ids=None*, *dag_ids=None*, *include_prior_dates=False*, *limit=100*, *session=None*)

Retrieve an XCom value, optionally meeting certain criteria TODO: "pickling" has been deprecated and JSON is preferred. "pickling" will be removed in Airflow 2.0.

classmethod delete(*cls*, *xcoms*, *session=None*)

airflow.models.**XCOM_RETURN_KEY = return_value**

3.22.5 Core and community package

Formerly the core code was maintained by the original creators - Airbnb. The code that was in the contrib package was supported by the community. The project was passed to the Apache community and currently the entire code is maintained by the community, so now the division has no justification, and it is only due to historical reasons. Currently, all new classes are added only to the contrib package.

PYTHON MODULE INDEX

a

airflow.contrib.hooks, 535
airflow.contrib.hooks.aws_athena_hook, 535
airflow.contrib.hooks.aws_dynamodb_hook, 536
airflow.contrib.hooks.aws_firehose_hook, 537
airflow.contrib.hooks.aws_glue_catalog_hook, 537
airflow.contrib.hooks.aws_hook, 538
airflow.contrib.hooks.aws_lambda_hook, 539
airflow.contrib.hooks.aws_sns_hook, 540
airflow.contrib.hooks.aws_sqs_hook, 540
airflow.contrib.hooks.azure_container_instance_hook, 541
airflow.contrib.hooks.azure_container_registry_hook, 542
airflow.contrib.hooks.azure_container_volume_hook, 542
airflow.contrib.hooks.azure_cosmos_hook, 543
airflow.contrib.hooks.azure_data_lake_hook, 544
airflow.contrib.hooks.azure_fileshare_hook, 545
airflow.contrib.hooks.bigquery_hook, 547
airflow.contrib.hooks.cassandra_hook, 559
airflow.contrib.hooks.cloudant_hook, 560
airflow.contrib.hooks.databricks_hook, 560
airflow.contrib.hooks.datadog_hook, 562
airflow.contrib.hooks.datastore_hook, 563
airflow.contrib.hooks.dingding_hook, 566
airflow.contrib.hooks.discord_webhook_hook, 567
airflow.contrib.hooks.emr_hook, 568
airflow.contrib.hooks.fs_hook, 568
airflow.contrib.hooks.ftp_hook, 568
airflow.contrib.hooks.gcp_api_base_hook, 571
airflow.contrib.hooks.gcp_bigtable_hook, 572
airflow.contrib.hooks.gcp_cloud_build_hook, 574
airflow.contrib.hooks.gcp_compute_hook, 575
airflow.contrib.hooks.gcp_container_hook, 577
airflow.contrib.hooks.gcp_dataflow_hook, 579
airflow.contrib.hooks.gcp_dataproc_hook, 580
airflow.contrib.hooks.gcp_function_hook, 581
airflow.contrib.hooks.gcp_kms_hook, 583
airflow.contrib.hooks.gcp_mlengine_hook, 584
airflow.contrib.hooks.gcp_natural_language_hook, 585
airflow.contrib.hooks.gcp_pubsub_hook, 588
airflow.contrib.hooks.gcp_spanner_hook, 590
airflow.contrib.hooks.gcp_speech_to_text_hook, 593
airflow.contrib.hooks.gcp_sql_hook, 593
airflow.contrib.hooks.gcp_text_to_speech_hook, 599
airflow.contrib.hooks.gcp_transfer_hook, 600
airflow.contrib.hooks.gcp_translate_hook, 604
airflow.contrib.hooks.gcp_video_intelligence_hook, 605
airflow.contrib.hooks.gcp_vision_hook, 606
airflow.contrib.hooks.gcs_hook, 607
airflow.contrib.hooks.grpc_hook, 611
airflow.contrib.hooks.imap_hook, 612
airflow.contrib.hooks.jenkins_hook, 614

airflow.contrib.hooks.jira_hook, 614
airflow.contrib.hooks.mongo_hook, 614
airflow.contrib.hooks.openfaas_hook, 617
airflow.contrib.hooks.opsgenie_alert_hook, 617
airflow.contrib.hooks.pinot_hook, 618
airflow.contrib.hooks.qubole_check_hook, 618
airflow.contrib.hooks.qubole_hook, 619
airflow.contrib.hooks.redis_hook, 620
airflow.contrib.hooks.redshift_hook, 620
airflow.contrib.hooks.sagemaker_hook, 621
airflow.contrib.hooks.salesforce_hook, 626
airflow.contrib.hooks.segment_hook, 627
airflow.contrib.hooks.sftp_hook, 628
airflow.contrib.hooks.slack_webhook_hook, 629
airflow.contrib.hooks.snowflake_hook, 630
airflow.contrib.hooks.spark_jdbc_hook, 631
airflow.contrib.hooks.spark_jdbc_script, 633
airflow.contrib.hooks.spark_sql_hook, 633
airflow.contrib.hooks.spark_submit_hook, 635
airflow.contrib.hooks.sqoop_hook, 637
airflow.contrib.hooks.ssh_hook, 640
airflow.contrib.hooks.vertica_hook, 640
airflow.contrib.hooks.wasb_hook, 641
airflow.contrib.hooks.winrm_hook, 643
airflow.contrib.operators, 313
airflow.contrib.operators.adls_list_operator, 313
airflow.contrib.operators.adls_to_gcs, 314
airflow.contrib.operators.aws_athena_operator, 315
airflow.contrib.operators.aws_sqs_publish_operator, 316
airflow.contrib.operators.awsbatch_operator, 317
airflow.contrib.operators.azure_container_instances_operator, 318
airflow.contrib.operators.azure_cosmos_operator, 319
airflow.contrib.operators.bigquery_check_operator, 320
airflow.contrib.operators.bigquery_get_data, 322
airflow.contrib.operators.bigquery_operator,
323
airflow.contrib.operators.bigquery_table_delete_ope
330
airflow.contrib.operators.bigquery_to_bigquery, 331
airflow.contrib.operators.bigquery_to_gcs, 332
airflow.contrib.operators.cassandra_to_gcs, 334
airflow.contrib.operators.databricks_operator, 335
airflow.contrib.operators.dataflow_operator, 341
airflow.contrib.operators.dataproc_operator, 345
airflow.contrib.operators.datastore_export_operator
356
airflow.contrib.operators.datastore_import_operator
357
airflow.contrib.operators.dingding_operator, 358
airflow.contrib.operators.discord_webhook_operator, 359
airflow.contrib.operators.druid_operator, 359
airflow.contrib.operators.ecs_operator, 360
airflow.contrib.operators.emr_add_steps_operator, 361
airflow.contrib.operators.emr_create_job_flow_opera
361
airflow.contrib.operators.emr_terminate_job_flow_op
362
airflow.contrib.operators.file_to_gcs, 362
airflow.contrib.operators.file_to_wasb, 363
airflow.contrib.operators.gcp_bigtable_operator, 363
airflow.contrib.operators.gcp_cloud_build_operator, 367
airflow.contrib.operators.gcp_compute_operator, 369
airflow.contrib.operators.gcp_container_operator, 373
airflow.contrib.operators.gcp_function_operator, 376
airflow.contrib.operators.gcp_natural_language_oper
378
airflow.contrib.operators.gcp_spanner_operator, 381
airflow.contrib.operators.gcp_speech_to_text_operat
384
airflow.contrib.operators.gcp_sql_operator,

385
airflow.contrib.operators.gcp_text_to_speech_operator, 392
airflow.contrib.operators.gcp_transfer_operator, 394
airflow.contrib.operators.gcp_translate_operator, 401
airflow.contrib.operators.gcp_translate_speech_operator, 402
airflow.contrib.operators.gcp_video_intelligence_operator, 404
airflow.contrib.operators.gcp_vision_operator, 406
airflow.contrib.operators.gcs_acl_operator, 422
airflow.contrib.operators.gcs_delete_operator, 423
airflow.contrib.operators.gcs_download_operator, 424
airflow.contrib.operators.gcs_list_operator, 425
airflow.contrib.operators.gcs_operator, 426
airflow.contrib.operators.gcs_to_bq, 427
airflow.contrib.operators.gcs_to_gcs, 430
airflow.contrib.operators.gcs_to_s3, 432
airflow.contrib.operators.grpc_operator, 434
airflow.contrib.operators.hipchat_operator, 435
airflow.contrib.operators.hive_to_dynamodb, 436
airflow.contrib.operators.imap_attachment_to_s3_operator, 437
airflow.contrib.operators.jenkins_job_trigger_operator, 437
airflow.contrib.operators.jira_operator, 439
airflow.contrib.operators.kubernetes_pod_operator, 441
airflow.contrib.operators.mlengine_operator, 443
airflow.contrib.operators.mongo_to_s3, 447
airflow.contrib.operators.mssql_to_gcs, 448
airflow.contrib.operators.mysql_to_gcs, 450
airflow.contrib.operators.opsgenie_alert_operator, 452
airflow.contrib.operators.oracle_to_azure_data_lake_transfer, 453
airflow.contrib.operators.oracle_to_oracle_transfer, 454
airflow.contrib.operators.postgres_to_gcs_operator, 455
airflow.contrib.operators.pubsub_operator, 456
airflow.contrib.operators.qubole_check_operator, 460
airflow.contrib.operators.qubole_operator, 462
airflow.contrib.operators.redis_publish_operator, 465
airflow.contrib.operators.s3_copy_object_operator, 465
airflow.contrib.operators.s3_delete_objects_operato
466
airflow.contrib.operators.s3_list_operator, 467
airflow.contrib.operators.s3_to_gcs_operator, 468
airflow.contrib.operators.s3_to_sftp_operator, 469
airflow.contrib.operators.sagemaker_base_operator, 470
airflow.contrib.operators.sagemaker_endpoint_confi
470
airflow.contrib.operators.sagemaker_endpoint_operat
471
airflow.contrib.operators.sagemaker_model_operator, 472
airflow.contrib.operators.sagemaker_training_operat
472
airflow.contrib.operators.sagemaker_transform_opera
473
airflow.contrib.operators.sagemaker_tuning_operator
474
airflow.contrib.operators.segment_track_event_opera
475
airflow.contrib.operators.sftp_operator, 475
airflow.contrib.operators.sftp_to_s3_operator, 477
airflow.contrib.operators.slack_webhook_operator, 478
airflow.contrib.operators.snowflake_operator, 479
airflow.contrib.operators.sns_publish_operator, 479
airflow.contrib.operators.spark_jdbc_operator, 481
airflow.contrib.operators.spark_sql_operator, 483
airflow.contrib.operators.spark_submit_operator, 485
airflow.contrib.operators.sqoop_operator,

487
airflow.contrib.operators.ssh_operator, 488
airflow.contrib.operators.vertica_operator, 488
airflow.contrib.operators.vertica_to_hive, 489
airflow.contrib.operators.vertica_to_mysql, 490
airflow.contrib.operators.wasb_delete_blob_operator, 491
airflow.contrib.operators.winrm_operator, 491
airflow.contrib.sensors, 492
airflow.contrib.sensors.aws_athena_sensor, 492
airflow.contrib.sensors.aws_glue_catalog_partition_sensor, 493
airflow.contrib.sensors.aws_redshift_cluster_sensor, 494
airflow.contrib.sensors.aws_sqs_sensor, 494
airflow.contrib.sensors.azure_cosmos_sensor, 495
airflow.contrib.sensors.bash_sensor, 495
airflow.contrib.sensors.bigquery_sensor, 496
airflow.contrib.sensors.cassandra_record_sensor, 496
airflow.contrib.sensors.cassandra_table_sensor, 497
airflow.contrib.sensors.celery_queue_sensor, 497
airflow.contrib.sensors.datadog_sensor, 498
airflow.contrib.sensors.emr_base_sensor, 498
airflow.contrib.sensors.emr_job_flow_sensor, 499
airflow.contrib.sensors.emr_step_sensor, 499
airflow.contrib.sensors.file_sensor, 499
airflow.contrib.sensors.ftp_sensor, 500
airflow.contrib.sensors.gcp_transfer_sensor, 500
airflow.contrib.sensors.gcs_sensor, 501
airflow.contrib.sensors.hdfs_sensor, 504
airflow.contrib.sensors.imap_attachment_sensor, 504
airflow.contrib.sensors.jira_sensor, 505
airflow.contrib.sensors.mongo_sensor, 506
airflow.contrib.sensors.pubsub_sensor, 506
airflow.contrib.sensors.python_sensor, 507
airflow.contrib.sensors.qubole_sensor, 507
airflow.contrib.sensors.redis_key_sensor, 508
airflow.contrib.sensors.redis_pub_sub_sensor, 508
airflow.contrib.sensors.sagemaker_base_sensor, 509
airflow.contrib.sensors.sagemaker_endpoint_sensor, 509
airflow.contrib.sensors.sagemaker_training_sensor, 510
airflow.contrib.sensors.sagemaker_transform_sensor, 510
airflow.contrib.sensors.sagemaker_tuning_sensor, 511
airflow.contrib.sensors.sftp_sensor, 511
airflow.contrib.sensors.wasb_sensor, 511
airflow.contrib.sensors.weekday_sensor, 512
airflow.executors, 644
airflow.executors.base_executor, 644
airflow.executors.celery_executor, 645
airflow.executors.dask_executor, 646
airflow.executors.kubernetes_executor, 647
airflow.executors.local_executor, 649
airflow.executors.sequential_executor, 650
airflow.hooks, 514
airflow.hooks.base_hook, 517
airflow.hooks.dbapi_hook, 518
airflow.hooks.docker_hook, 520
airflow.hooks.druid_hook, 520
airflow.hooks.hdfs_hook, 521
airflow.hooks.hive_hooks, 521
airflow.hooks.http_hook, 527
airflow.hooks.jdbc_hook, 528
airflow.hooks.mssql_hook, 528
airflow.hooks.mysql_hook, 529
airflow.hooks.oracle_hook, 530
airflow.hooks.pig_hook, 531
airflow.hooks.postgres_hook, 531
airflow.hooks.presto_hook, 532
airflow.hooks.S3_hook, 514
airflow.hooks.samba_hook, 533
airflow.hooks.slack_hook, 533
airflow.hooks.sqlite_hook, 533
airflow.hooks.webhdfs_hook, 534
airflow.hooks.zendesk_hook, 534
airflow.macros, 273
airflow.models, 653

airflow.models.base, 653
airflow.models.baseoperator, 654
airflow.models.connection, 659
airflow.models.crypto, 660
airflow.models.dag, 661
airflow.models.dagbag, 668
airflow.models.dagpickle, 669
airflow.models.dagrun, 670
airflow.models.errors, 671
airflow.models.kubernetes, 672
airflow.models.log, 672
airflow.models.pool, 673
airflow.models.skipmixin, 673
airflow.models.slamiss, 673
airflow.models.taskfail, 674
airflow.models.taskinstance, 674
airflow.models.taskreschedule, 679
airflow.models.variable, 680
airflow.models.xcom, 681
airflow.operators, 275
airflow.operators.bash_operator, 275
airflow.operators.check_operator, 276
airflow.operators.dagrun_operator, 278
airflow.operators.docker_operator, 279
airflow.operators.druid_check_operator, 281
airflow.operators.dummy_operator, 281
airflow.operators.email_operator, 282
airflow.operators.generic_transfer, 282
airflow.operators.hive_operator, 283
airflow.operators.hive_stats_operator, 284
airflow.operators.hive_to_druid, 285
airflow.operators.hive_to_mysql, 286
airflow.operators.hive_to_samba_operator, 287
airflow.operators.http_operator, 287
airflow.operators.jdbc_operator, 288
airflow.operators.latest_only_operator, 288
airflow.operators.mssql_operator, 289
airflow.operators.mssql_to_hive, 289
airflow.operators.mysql_operator, 290
airflow.operators.mysql_to_hive, 291
airflow.operators.oracle_operator, 292
airflow.operators.papermill_operator, 292
airflow.operators.pig_operator, 293
airflow.operators.postgres_operator, 293
airflow.operators.presto_check_operator, 294
airflow.operators.presto_to_mysql, 295
airflow.operators.python_operator, 296
airflow.operators.redshift_to_s3_operator, 299
airflow.operators.s3_file_transform_operator, 300
airflow.operators.s3_to_hive_operator, 301
airflow.operators.s3_to_redshift_operator, 302
airflow.operators.slack_operator, 303
airflow.operators.sqlite_operator, 304
airflow.operators.subdag_operator, 305
airflow.sensors, 305
airflow.sensors.base_sensor_operator, 305
airflow.sensors.external_task_sensor, 306
airflow.sensors.hdfs_sensor, 307
airflow.sensors.hive_partition_sensor, 308
airflow.sensors.http_sensor, 308
airflow.sensors.metastore_partition_sensor, 309
airflow.sensors.named_hive_partition_sensor, 310
airflow.sensors.s3_key_sensor, 310
airflow.sensors.s3_prefix_sensor, 311
airflow.sensors.sql_sensor, 312
airflow.sensors.time_delta_sensor, 312
airflow.sensors.time_sensor, 312
airflow.sensors.web_hdfs_sensor, 313

HTTP ROUTING TABLE

/api

GET /api/experimental/dags/<DAG_ID>/dag_runs, 181
GET /api/experimental/dags/<DAG_ID>/dag_runs/<string:execution_date>/tasks/<TASK_ID>, 181
GET /api/experimental/dags/<DAG_ID>/paused/<string:paused>, 181
GET /api/experimental/dags/<DAG_ID>/tasks/<TASK_ID>, 181
GET /api/experimental/dags/<string:dag_id>/dag_runs/<string:execution_date>, 181
GET /api/experimental/latest_runs, 181
GET /api/experimental/pools, 181
GET /api/experimental/pools/<string:name>, 181
GET /api/experimental/test, 181
POST /api/experimental/dags/<DAG_ID>/dag_runs, 181
POST /api/experimental/pools, 181
DELETE /api/experimental/pools/<string:name>, 181

INDEX

Non-alphabetical

__call__() (*airflow.contrib.hooks.databricks_hook._TokenAuth method*), 562
__deepcopy__() (*airflow.models.BaseOperator method*), 685
__deepcopy__() (*airflow.models.baseoperator.BaseOperator method*), 658
__deepcopy__() (*airflow.models.DAG method*), 692
__deepcopy__() (*airflow.models.dag.DAG method*), 665
__enter__() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569
__enter__() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 612
__enter__() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
__enter__() (*airflow.contrib.hooks.ssh_hook.SSHHook method*), 640
__enter__() (*airflow.models.DAG method*), 690
__enter__() (*airflow.models.dag.DAG method*), 663
__eq__() (*airflow.contrib.hooks.databricks_hook.RunState method*), 562
__eq__() (*airflow.models.BaseOperator method*), 685
__eq__() (*airflow.models.baseoperator.BaseOperator method*), 657
__eq__() (*airflow.models.DAG method*), 690
__eq__() (*airflow.models.dag.DAG method*), 663
__exit__() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569
__exit__() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 612
__exit__() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
__exit__() (*airflow.contrib.hooks.ssh_hook.SSHHook method*), 640
__exit__() (*airflow.models.DAG method*), 690
__exit__() (*airflow.models.dag.DAG method*), 663
__get_collection_name() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
__get_database_name() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
__get_tls_config() (*airflow.operators.docker_operator.DockerOperator method*), 280
__get_token() (*airflow.hooks.slack_hook.SlackHook method*), 533
__getattribute__() (*airflow.contrib.operators.qubole_check_operator.QuboleCheckOperato method*), 461
__getattribute__() (*airflow.contrib.operators.qubole_check_operator.QuboleValueCheckOp method*), 461
__getattribute__() (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464
__getstate__() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 523
__getstate__() (*airflow.models.BaseOperator method*), 686
__getstate__() (*airflow.models.baseoperator.BaseOperator method*), 658
__handle_rate_limit_exception() (*airflow.hooks.zendesk_hook.ZendeskHook method*), 535
__hash__() (*airflow.models.BaseOperator method*), 685
__hash__() (*airflow.models.baseoperator.BaseOperator method*), 657
__hash__() (*airflow.models.DAG method*), 690
__hash__() (*airflow.models.dag.DAG method*), 663
__login() (*airflow.hooks.docker_hook.DockerHook method*), 520
__lshift__() (*airflow.models.BaseOperator method*), 685
__lshift__() (*airflow.models.baseoperator.BaseOperator method*), 657
__lt__() (*airflow.models.BaseOperator method*), 685
__lt__() (*airflow.models.baseoperator.BaseOperator method*), 657

`__lt__()` (*airflow.models.DAG method*), 690
`__lt__()` (*airflow.models.dag.DAG method*), 663
`__mapper_args__` (*airflow.operators.check_operator.IntervalCheckOperator attribute*), 277
`__mapper_args__` (*airflow.operators.check_operator.ValueCheckOperator attribute*), 277
`__ne__()` (*airflow.models.BaseOperator method*), 685
`__ne__()` (*airflow.models.baseoperator.BaseOperator method*), 657
`__ne__()` (*airflow.models.DAG method*), 690
`__ne__()` (*airflow.models.dag.DAG method*), 663
`__NO_DEFAULT_SENTINEL` (*airflow.models.Variable attribute*), 705
`__NO_DEFAULT_SENTINEL` (*airflow.models.variable.Variable attribute*), 680
`__repr__()` (*airflow.contrib.hooks.databricks_hook.RunState method*), 562
`__repr__()` (*airflow.executors.kubernetes_executor.KubernetesExecutorConfig method*), 647
`__repr__()` (*airflow.models.BaseOperator method*), 686
`__repr__()` (*airflow.models.baseoperator.BaseOperator method*), 658
`__repr__()` (*airflow.models.Connection method*), 687
`__repr__()` (*airflow.models.connection.Connection method*), 660
`__repr__()` (*airflow.models.DAG method*), 690
`__repr__()` (*airflow.models.dag.DAG method*), 663
`__repr__()` (*airflow.models.dag.DagModel method*), 667
`__repr__()` (*airflow.models.DagModel method*), 694
`__repr__()` (*airflow.models.DagRun method*), 696
`__repr__()` (*airflow.models.dagrun.DagRun method*), 670
`__repr__()` (*airflow.models.Pool method*), 698
`__repr__()` (*airflow.models.pool.Pool method*), 673
`__repr__()` (*airflow.models.SlaMiss method*), 700
`__repr__()` (*airflow.models.slamiss.SlaMiss method*), 674
`__repr__()` (*airflow.models.TaskInstance method*), 702
`__repr__()` (*airflow.models.taskinstance.TaskInstance method*), 677
`__repr__()` (*airflow.models.Variable method*), 705
`__repr__()` (*airflow.models.variable.Variable method*), 680
`__repr__()` (*airflow.models.XCom method*), 706
`__repr__()` (*airflow.models.xcom.XCom method*), 681
`__rlshift__()` (*airflow.models.BaseOperator method*), 685
`__rlshift__()` (*airflow.models.baseoperator.BaseOperator method*), 657
`__rrshift__()` (*airflow.models.BaseOperator method*), 685
`__rrshift__()` (*airflow.models.baseoperator.BaseOperator method*), 657
`__rshift__()` (*airflow.models.BaseOperator method*), 685
`__rshift__()` (*airflow.models.baseoperator.BaseOperator method*), 657
`__setattr__()` (*airflow.contrib.operators.qubole_check_operator.QuboleCheckOperato method*), 461
`__setattr__()` (*airflow.contrib.operators.qubole_check_operator.QuboleValueCheckOp method*), 461
`__setattr__()` (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464
`__setstate__()` (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 523
`__setstate__()` (*airflow.models.BaseOperator method*), 686
`__setstate__()` (*airflow.models.baseoperator.BaseOperator method*), 658
`__table_args__` (*airflow.models.DagRun attribute*), 696
`__table_args__` (*airflow.models.dagrun.DagRun attribute*), 670
`__table_args__` (*airflow.models.Log attribute*), 698
`__table_args__` (*airflow.models.log.Log attribute*), 672
`__table_args__` (*airflow.models.SlaMiss attribute*), 700
`__table_args__` (*airflow.models.slamiss.SlaMiss attribute*), 674
`__table_args__` (*airflow.models.TaskFail attribute*), 699
`__table_args__` (*airflow.models.taskfail.TaskFail attribute*), 674
`__table_args__` (*airflow.models.TaskInstance attribute*), 700
`__table_args__` (*airflow.models.taskinstance.TaskInstance attribute*), 675
`__table_args__` (*airflow.models.TaskReschedule attribute*), 705
`__table_args__` (*airflow.models.taskreschedule.TaskReschedule attribute*), 680
`__table_args__` (*airflow.models.XCom attribute*), 706
`__table_args__` (*airflow.models.xcom.XCom attribute*), 681

__tablename__ (*airflow.models.Connection attribute*), 687
__tablename__ (*airflow.models.connection.Connection attribute*), 659
__tablename__ (*airflow.models.dag.DagModel attribute*), 667
__tablename__ (*airflow.models.DagModel attribute*), 694
__tablename__ (*airflow.models.DagPickle attribute*), 696
__tablename__ (*airflow.models.dagpickle.DagPickle attribute*), 669
__tablename__ (*airflow.models.DagRun attribute*), 696
__tablename__ (*airflow.models.dagrun.DagRun attribute*), 670
__tablename__ (*airflow.models.errors.ImportError attribute*), 671
__tablename__ (*airflow.models.KubeResourceVersion attribute*), 698
__tablename__ (*airflow.models.kubernetes.KubeResourceVersion attribute*), 672
__tablename__ (*airflow.models.kubernetes.KubeWorkerIdentifier attribute*), 672
__tablename__ (*airflow.models.KubeWorkerIdentifier attribute*), 698
__tablename__ (*airflow.models.Log attribute*), 698
__tablename__ (*airflow.models.log.Log attribute*), 672
__tablename__ (*airflow.models.Pool attribute*), 698
__tablename__ (*airflow.models.pool.Pool attribute*), 673
__tablename__ (*airflow.models.SlaMiss attribute*), 699
__tablename__ (*airflow.models.slamiss.SlaMiss attribute*), 674
__tablename__ (*airflow.models.TaskFail attribute*), 699
__tablename__ (*airflow.models.taskfail.TaskFail attribute*), 674
__tablename__ (*airflow.models.TaskInstance attribute*), 700
__tablename__ (*airflow.models.taskinstance.TaskInstance attribute*), 675
__tablename__ (*airflow.models.TaskReschedule attribute*), 704
__tablename__ (*airflow.models.taskreschedule.TaskReschedule attribute*), 679
__tablename__ (*airflow.models.Variable attribute*), 705
__tablename__ (*airflow.models.variable.Variable attribute*), 680
__tablename__ (*airflow.models.XCom attribute*), 706
__tablename__ (*airflow.models.xcom.XCom attribute*), 681
_api_resource_configs_duplication_check() (*in module airflow.contrib.hooks.bigquery_hook*), 558
_append_label() (*airflow.contrib.hooks.gcp_container_hook.GKEClusterHook static method*), 578
_apply_to_instance() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 590
_authorize() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBa method*), 571
_b64decode() (*in module airflow.contrib.hooks.gcp_kms_hook*), 583
_b64encode() (*in module airflow.contrib.hooks.gcp_kms_hook*), 583
_base_operator_shallow_copy_attrs (*airflow.models.BaseOperator attribute*), 684
_base_operator_shallow_copy_attrs (*airflow.models.baseoperator.BaseOperator attribute*), 657
_bind_parameters() (*in module airflow.contrib.hooks.bigquery_hook*), 558
_bq_cast() (*in module airflow.contrib.hooks.bigquery_hook*), 558
_build_cluster_data() (*airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOp method*), 349
_build_cmd() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook static method*), 580
_build_command_line_parameters() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner method*), 597
_build_dataflow_job_name() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook static method*), 580
_build_discord_payload() (*airflow.contrib.hooks.discord_webhook_hook.DiscordWebhookHook method*), 568
_build_gce_cluster_config() (*airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOp method*), 349
_build_jdbc_application_arguments() (*airflow.contrib.hooks.spark_jdbc_hook.SparkJDBCHook method*), 632
_build_lifecycle_config() (*airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOp method*), 349
_build_message() (*airflow.contrib.hooks.dingding_hook.DingdingHook method*), 566
_build_opsgenie_payload() (*air-*

flow.contrib.operators.opsgenie_alert_operator.OpsgenieAlertOperator method), 453
_build_scale_cluster_data() (*airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOperator method*), 350
_build_slack_message() (*airflow.contrib.hooks.slack_webhook_hook.SlackWebhookHook method*), 630
_build_spark_driver_kill_command() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 637
_build_spark_submit_command() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
_build_track_driver_status_command() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
_change_state() (*airflow.executors.kubernetes_executor.KubernetesExecutor method*), 649
_check_and_change_state_before_execution() (*airflow.models.TaskInstance method*), 703
_check_and_change_state_before_execution() (*airflow.models.taskinstance.TaskInstance method*), 677
_check_done() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcOperation method*), 581
_check_for_error() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook static method*), 606
_check_global_operation_status() (*airflow.contrib.hooks.gcp_compute_hook.GceHook static method*), 577
_check_if_db_exists() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator method*), 386
_check_if_function_exists() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator method*), 377
_check_if_instance_exists() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator method*), 386
_check_input() (*airflow.contrib.operators.gcp_container_operator.GKEClusterCreateOperator method*), 375
_check_input() (*airflow.contrib.operators.gcp_container_operator.GKEClusterDeleteOperator method*), 374
_check_mail_body() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
_check_ssl_file() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook static method*), 598
_check_success_task() (*airflow.contrib.operators.awsbatch_operator.AWSBatchOperator method*), 317
_check_success_task() (*airflow.contrib.operators.ecs_operator.ECSOperator method*), 361
_check_task_id() (*airflow.contrib.sensors.celery_queue_sensor.CeleryQueueSensor method*), 497
_check_zone_operation_status() (*airflow.contrib.hooks.gcp_compute_hook.GceHook static method*), 577
_cleanse_time_partitioning() (*in module airflow.contrib.hooks.bigquery_hook*), 558
_client (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook attribute*), 572
_client (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook attribute*), 590
_client (*airflow.contrib.hooks.gcp_speech_to_text_hook.GCPSpeechToText attribute*), 593
_client (*airflow.contrib.hooks.gcp_text_to_speech_hook.GCPTextToSpeech attribute*), 599
_client (*airflow.contrib.hooks.gcp_translate_hook.CloudTranslateHook attribute*), 604
_client (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook attribute*), 606
_compare_column_families() (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableCreateOp method*), 366
_configure_csv_file() (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOp method*), 451
_conn (*airflow.contrib.hooks.gcp_cloud_build_hook.CloudBuildHook attribute*), 574
_conn (*airflow.contrib.hooks.gcp_compute_hook.GceHook attribute*), 575
_conn (*airflow.contrib.hooks.gcp_function_hook.GcfHook attribute*), 582
_conn (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLang attribute*), 585
_conn (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook attribute*), 598
_conn (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook attribute*), 594
_conn (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook attribute*), 601
_conn (*airflow.contrib.hooks.gcp_video_intelligence_hook.CloudVideoIntellig attribute*), 605
_conn (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook attribute*), 607
_convert_date_to_dict() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobPreprocess static method*), 394

`_convert_repo_url_to_dict()` (*airflow.contrib.operators.gcp_cloud_build_operator.BuildProcessor static method*), 368
`_convert_storage_url_to_dict()` (*airflow.contrib.operators.gcp_cloud_build_operator.BuildProcessor static method*), 368
`_convert_time_to_dict()` (*airflow.contrib.operators.gcp_transfer_operator.TransferJobPreprocessor static method*), 394
`_convert_to_float_if_possible()` (*in module airflow.operators.check_operator*), 276
`_convert_type()` (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator class method*), 451
`_convert_types()` (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator class method*), 451
`_copy_single_object()` (*airflow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator method*), 432
`_correct_path()` (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
`_create_body()` (*airflow.contrib.operators.gcp_transfer_operator.GoogleCloudStorageToGoogleCloudStorageTransferOperator method*), 401
`_create_body()` (*airflow.contrib.operators.gcp_transfer_operator.S3ToGoogleCloudStorageTransferOperator method*), 400
`_create_file()` (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
`_create_files()` (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
`_create_hook()` (*airflow.contrib.sensors.ftp_sensor.FTPSensor method*), 500
`_create_hook()` (*airflow.contrib.sensors.ftp_sensor.FTPSSensor method*), 500
`_create_new_function()` (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator method*), 377
`_create_pod_id()` (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler static method*), 648
`_Dataflow` (*class in airflow.contrib.hooks.gcp_dataflow_hook*), 580
`_DataflowJob` (*class in airflow.contrib.hooks.gcp_dataflow_hook*), 579
`_DataProcJob` (*class in airflow.contrib.hooks.gcp_dataproc_hook*), 580
`_DataProcJobBuilder` (*class in airflow.contrib.hooks.gcp_dataproc_hook*), 580
`_DataProcOperation` (*class in airflow.contrib.hooks.gcp_dataproc_hook*), 581
`_datetime_to_label_safe_datestring()` (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler static method*), 648
`_deep_string_coerce()` (*in module airflow.contrib.operators.databricks_operator*), 335
`_DEFAULT_SCOPES` (*in module airflow.contrib.hooks.gcp_api_base_hook*), 571
`_delete_top_row_and_compress()` (*airflow.operators.s3_to_hive_operator.S3ToHiveTransfer static method*), 302
`_dict_to_proto()` (*airflow.contrib.hooks.gcp_container_hook.GKEClusterHook static method*), 578
`_do_api_call()` (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
`_do_skip_downstream_tasks()` (*airflow.sensors.base_sensor_operator.BaseSensorOperator method*), 306
`_download_sql_proxy_if_needed()` (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner method*), 597
`_emit_duration_stats_for_finished_state()` (*airflow.models.DagRun method*), 697
`_emit_duration_stats_for_finished_state()` (*airflow.models.dagrun.DagRun method*), 671
`_escape()` (*in module airflow.contrib.hooks.bigquery_hook*), 558
`_execute()` (*airflow.contrib.operators.oracle_to_oracle_transfer.OracleTo method*), 454
`_execute_in_subprocess()` (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
`_execute_query()` (*airflow.contrib.operators.gcp_sql_operator.CloudSqlQueryOperator method*), 392
`_execute_set_machine_type()` (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 576
`_execute_sql_in_transaction()` (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook static method*), 593
`_export_cmd()` (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 639
`_extra` (*airflow.models.Connection attribute*), 687
`_extra` (*airflow.models.connection.Connection attribute*), 660
`_extract_job()` (*airflow.contrib.hooks.gcp_dataflow_hook._Dataflow

static method), 580
_fernet (*in module airflow.models.crypto*), 660
_fetch_mail_body() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
_find_valid_server() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 523
_format_subscription() (*in module airflow.contrib.hooks.gcp_pubsub_hook*), 588
_format_topic() (*in module airflow.contrib.hooks.gcp_pubsub_hook*), 588
_full_location() (*airflow.contrib.hooks.gcp_function_hook.GcfHook static method*), 582
_G_APP_CRED_ENV_VAR (*in module airflow.contrib.hooks.gcp_api_base_hook*), 571
_gcs_object_is_directory() (*airflow.contrib.operators.s3_to_gcs_operator.S3ToGoogleCloudStorageOperator static method*), 469
_generate_connection_uri() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598
_generate_pip_install_cmd() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
_generate_python_cmd() (*airflow.operators.python_operator.PythonVirtualenvOperator static method*), 298
_generate_python_code() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
_generate_temp_filename() (*airflow.contrib.operators.dataproc_operator.DataProcPySparkOperator static method*), 355
_generate_unique_path() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook static method*), 598
_generate_virtualenv_cmd() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
_get_access_token() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook method*), 571
_get_api_key() (*airflow.contrib.hooks.opsgenie_alert_hook.OpsgenieAlertHook method*), 617
_get_autogenerated_id() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook static method*), 607
_get_aws_credentials() (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook method*), 630
_get_bool() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook static method*), 598
_get_client() (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook method*), 572
_get_client() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 590
_get_client() (*airflow.hooks.webhdfs_hook.WebHDFSHook method*), 534
_get_col_type_dict() (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOp method*), 451
_get_concurrency_reached() (*airflow.models.DAG method*), 690
_get_concurrency_reached() (*airflow.models.dag.DAG method*), 664
_get_conn_params() (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook method*), 630
_get_connection_from_env() (*airflow.hooks.base_hook.BaseHook class method*), 517
_get_connections_from_db() (*airflow.hooks.base_hook.BaseHook class method*), 517
_get_credential_parameters() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner method*), 597
_get_credentials() (*airflow.contrib.hooks.aws_hook.AwsHook method*), 539
_get_credentials() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook method*), 571
_get_endpoint() (*airflow.contrib.hooks.dingding_hook.DingdingHook method*), 566
_get_error_code() (*airflow.contrib.sensors.ftp_sensor.FTPSensor method*), 500
_get_executor() (*in module airflow.executors*), 653
_get_export_format_argument() (*airflow.contrib.hooks.sqoop_hook.SqoopHook static method*), 638
_get_field() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBa method*), 571
_get_field() (*airflow.contrib.hooks.grpc_hook.GrpcHook method*), 611
_get_field() (*airflow.contrib.operators.gcp_container_operator.GKEPod method*), 376
_get_filtered_args() (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464

_get_graceful_decommission_timeout() (*airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOperator static method*), 350
_get_grpc_hook() (*airflow.contrib.operators.grpc_operator.GrpcOperator method*), 434
_get_hook() (*airflow.contrib.operators.databricks_operator.DatabricksRunNowOperator method*), 340
_get_hook() (*airflow.contrib.operators.databricks_operator.DatabricksSubmitRunOperator method*), 338
_get_init_action_timeout() (*airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOperator method*), 349
_get_instance_socket_name() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598
_get_instance_view() (*airflow.contrib.hooks.azure_container_instance_hook.AzureContainerInstanceHook method*), 541
_get_is_paused() (*airflow.models.DAG method*), 690
_get_is_paused() (*airflow.models.dag.DAG method*), 664
_get_job() (*airflow.contrib.hooks.gcp_dataflow_hook._DataflowJob method*), 580
_get_job() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 584
_get_job_id_from_name() (*airflow.contrib.hooks.gcp_dataflow_hook._DataflowJob method*), 579
_get_latest_execution_date() (*airflow.models.DAG method*), 691
_get_latest_execution_date() (*airflow.models.dag.DAG method*), 664
_get_max_partition_from_part_specs() (*airflow.hooks.hive_hooks.HiveMetastoreHook static method*), 524
_get_numeric_matches() (*airflow.operators.check_operator.ValueCheckOperator method*), 277
_get_pretty_exception_message() (*airflow.hooks.presto_hook.PrestoHook static method*), 532
_get_previous_ti() (*airflow.models.TaskInstance method*), 702
_get_previous_ti() (*airflow.models.taskinstance.TaskInstance method*), 677
_get_project_id() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook method*), 572
_get_proxy_user() (*airflow.hooks.hive_hooks.HiveCliHook method*), 522
_get_results() (*airflow.hooks.hive_hooks.HiveServer2Hook method*), 525
_get_settings_version() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator static method*), 386
_get_spark_binary_path() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
_get_sqlproxy_instance_specification() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598
_get_string_matches() (*airflow.operators.check_operator.ValueCheckOperator method*), 277
_get_token() (*airflow.contrib.hooks.slack_webhook_hook.SlackWebhook method*), 629
_get_top_row_as_list() (*airflow.operators.s3_to_hive_operator.S3ToHiveTransfer method*), 302
_get_webhook_endpoint() (*airflow.contrib.hooks.discord_webhook_hook.DiscordWebhookHook method*), 567
_handle_databricks_operator_execution() (*in module airflow.contrib.operators.databricks_operator*), 336
_handle_not_found_mode() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
_handle_reschedule() (*airflow.models.TaskInstance method*), 703
_handle_reschedule() (*airflow.models.taskinstance.TaskInstance method*), 678
_handle_response() (*airflow.contrib.operators.grpc_operator.GrpcOperator method*), 434
_health_check_kube_watcher() (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler method*), 648
_import_cmd() (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 638
_inject_aws_credentials() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobPreprocess method*), 394
_inject_project_id() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 603
_inject_secrets() (*airflow.executors.kubernetes_executor.KubernetesExecutor method*), 649
_integrate_plugins() (*in module air-*

flow.executors), 653
`_integrate_plugins()` (*in module airflow.hooks*), 535
`_integrate_plugins()` (*in module airflow.operators*), 305
`_integrate_plugins()` (*in module airflow.sensors*), 313
`_is_escaping_current_directory()` (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
`_is_os_64bit()` (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner static method*), 597
`_is_present_and_empty()` (*airflow.contrib.operators.gcp_function_operator.ZipPathPreprocessor static method*), 377
`_is_symlink()` (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
`_kerberos_security_mode` (*in module airflow.hooks.webhdfs_hook*), 534
`_label_safe_datestring_to_datetime()` (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler static method*), 648
`_labels_to_key()` (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler method*), 649
`_line()` (*airflow.contrib.hooks.gcp_dataflow_hook._Dataflow method*), 580
`_list_mail_ids_desc()` (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
`_log_last()` (*airflow.contrib.operators.azure_container_instances_operator.AzureContainerInstancesOperator method*), 319
`_log_request_error()` (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
`_make_intermediate_dirs()` (*in module airflow.contrib.operators.sftp_operator*), 476
`_make_kube_watcher()` (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler method*), 648
`_make_safe_label_value()` (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler static method*), 648
`_make_safe_pod_id()` (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler static method*), 648
`_match_headers()` (*airflow.operators.s3_to_hive_operator.S3ToHiveTransfer method*), 302
`_monitor_logging()` (*airflow.contrib.operators.azure_container_instances_operator.AzureContainerInstancesOperator method*), 319
`_normalize_mlengine_job_id()` (*in module airflow.contrib.operators.mlengine_operator*), 443
`_num_tasks_per_fetch_process()` (*airflow.executors.celery_executor.CeleryExecutor method*), 646
`_num_tasks_per_send_process()` (*airflow.executors.celery_executor.CeleryExecutor method*), 646
`_parse_gcs_url()` (*in module airflow.contrib.hooks.gcs_hook*), 611
`_parse_host()` (*airflow.contrib.hooks.databricks_hook.DatabricksHook static method*), 561
`_parse_s3_config()` (*in module airflow.contrib.hooks.aws_hook*), 538
`_pass_op_args()` (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
`_password` (*airflow.models.Connection attribute*), 687
`_password` (*airflow.models.connection.Connection attribute*), 660
`_poll_with_exponential_delay()` (*in module airflow.contrib.hooks.gcp_mlengine_hook*), 584
`_possibly_replace_template()` (*airflow.contrib.operators.gcp_compute_operator.GceInstanceGroupMa method*), 373
`_prepare_cli_cmd()` (*airflow.hooks.hive_hooks.HiveCliHook method*), 522
`_prepare_command()` (*airflow.contrib.hooks.spark_sql_hook.SparkSqlHook method*), 634
`_prepare_command()` (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 638
`_prepare_hiveconf()` (*airflow.hooks.hive_hooks.HiveCliHook static method*), 522
`_process_future()` (*airflow.executors.dask_executor.DaskExecutor method*), 646
`_process_spark_status_log()` (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 637
`_process_spark_submit_log()` (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
`_query_cassandra()` (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS method*), 335
`_query_mssql()` (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOp method*), 449
`_query_mysql()` (*air-*

flow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator method), 451
_query_postgres() (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleCloudStorageOperator method*), 455
_quote() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook static method*), 598
_raise_error() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcOperation method*), 581
_raise_ex_different_names() (*airflow.contrib.hooks.gcp_vision_hook.NameDeterminer method*), 606
_raise_ex_unable_to_determine_name() (*airflow.contrib.hooks.gcp_vision_hook.NameDeterminer method*), 606
_read_result() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
_reformat_date() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobPreprocessor method*), 394
_reformat_repo_source() (*airflow.contrib.operators.gcp_cloud_build_operator.BuildProcessor method*), 368
_reformat_schedule() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobPreprocessor method*), 394
_reformat_source() (*airflow.contrib.operators.gcp_cloud_build_operator.BuildProcessor method*), 368
_reformat_storage_source() (*airflow.contrib.operators.gcp_cloud_build_operator.BuildProcessor method*), 368
_reformat_time() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobPreprocessor method*), 394
_resolve_connection() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
_resolve_jdbc_connection() (*airflow.contrib.hooks.spark_jdbc_hook.SparkJDBCHook method*), 632
_resolve_should_track_driver_status() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
_restrict_aws_credentials() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobValidator method*), 394
_retrieve_mails_attachments_by_name() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 613
_retryable_error() (*in module airflow.contrib.hooks.databricks_hook*), 562
_run() (*airflow.executors.kubernetes_executor.KubernetesJobWatcher method*), 647
_run_raw_task() (*airflow.models.TaskInstance method*), 703
_run_raw_task() (*airflow.models.taskinstance.TaskInstance method*), 678
_serialize_cell() (*airflow.hooks.dbapi_hook.DbApiHook static method*), 519
_serialize_cell() (*airflow.hooks.mysql_hook.MySqlHook static method*), 529
_serialize_cell() (*airflow.hooks.postgres_hook.PostgresHook static method*), 532
_set_airflow_version_label() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOp method*), 377
_set_context() (*airflow.executors.LoggingMixin method*), 651
_set_env_from_extras() (*airflow.contrib.operators.gcp_container_operator.GKEPodOperator method*), 376
_set_relatives() (*airflow.models.BaseOperator method*), 686
_set_relatives() (*airflow.models.baseoperator.BaseOperator method*), 658
_set_resources() (*airflow.contrib.operators.kubernetes_pod_operator.KubernetesPodOper method*), 443
_set_variables() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook static method*), 580
_split_tablename() (*in module airflow.contrib.hooks.bigquery_hook*), 558
_start_dataflow() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook method*), 580
_start_driver_status_tracking() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 637
_start_template_dataflow() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook method*), 580
_state (*airflow.models.DagRun attribute*), 696
_state (*airflow.models.dagrun.DagRun attribute*), 670
_stringify() (*airflow.contrib.operators.mongo_to_s3.MongoToS3Operat static method*), 448
_strip_sql() (*airflow.hooks.presto_hook.PrestoHook static method*), 532
_strip_unsafe_kubernetes_special_chars() (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler*

static method), 648
_test_cycle_helper() (*airflow.models.DAG method*), 693
_test_cycle_helper() (*airflow.models.dag.DAG method*), 667
_to_float() (*airflow.operators.check_operator.ValueCheckOperator method*), 277
_to_timestamp() (*airflow.contrib.hooks.salesforce_hook.SalesforceHook class method*), 627
_TokenAuth (*class in airflow.contrib.hooks.databricks_hook*), 562
_try_number (*airflow.models.TaskInstance attribute*), 700
_try_number (*airflow.models.taskinstance.TaskInstance attribute*), 675
_types (*airflow.models.Connection attribute*), 687
_types (*airflow.models.connection.Connection attribute*), 660
_update_function() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator method*), 377
_upload_file_temp() (*airflow.contrib.operators.dataproc_operator.DataProcPySparkOperator method*), 355
_upload_source_code() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator method*), 377
_upload_to_gcs() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator method*), 335
_upload_to_gcs() (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOperator method*), 449
_upload_to_gcs() (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator method*), 451
_upload_to_gcs() (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleCloudStorageOperator method*), 455
_val (*airflow.models.Variable attribute*), 705
_val (*airflow.models.variable.Variable attribute*), 680
_validate() (*airflow.executors.kubernetes_executor.KubeConfig method*), 647
_validate_all_body_fields() (*airflow.contrib.operators.gcp_compute_operator.GceInstanceTemplateCopyOperator method*), 372
_validate_all_body_fields() (*airflow.contrib.operators.gcp_compute_operator.GceSetMachineTypeOperator method*), 371
_validate_all_body_fields() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator method*), 377
_validate_available_memory_in_mb() (*in module airflow.contrib.operators.gcp_function_operator*), 376
_validate_body_fields() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceCreateOpe method*), 387
_validate_body_fields() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseC method*), 388
_validate_body_fields() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseI method*), 389
_validate_body_fields() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceExportOpe method*), 390
_validate_body_fields() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceImportOpe method*), 391
_validate_connection() (*airflow.contrib.hooks.cloudant_hook.CloudantHook method*), 560
_validate_input_values() (*airflow.sensors.base_sensor_operator.BaseSensorOperator method*), 306
_validate_inputs() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598
_validate_inputs() (*airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMix method*), 363
_validate_inputs() (*airflow.contrib.operators.gcp_cloud_build_operator.CloudBuildCreateI method*), 368
_validate_inputs() (*airflow.contrib.operators.gcp_compute_operator.GceBaseOperator method*), 369
_validate_inputs() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeleteOpe method*), 378
_validate_inputs() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOp method*), 377
_validate_inputs() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance method*), 384
_validate_inputs() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance method*), 383
_validate_inputs() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance method*), 383
_validate_inputs() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance method*), 384

_validate_inputs() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeleteOperator method*), 382

_validate_inputs() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeployOperator method*), 382

_validate_inputs() (*airflow.contrib.operators.gcp_speech_to_text_operator.GcpSpeechToTextRecognizeSpeechOperator method*), 385

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator method*), 386

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceCreateOperator method*), 387

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseCreateOperator method*), 388

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseDeleteOperator method*), 390

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabasePatchOperator method*), 389

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceExportOperator method*), 390

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceImportOperator method*), 391

_validate_inputs() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstancePatchOperator method*), 387

_validate_inputs() (*airflow.contrib.operators.gcp_text_to_speech_operator.GcpTextToSpeechSynthesizeOperator method*), 393

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobCreateOperator method*), 395

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobDeleteOperator method*), 396

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobUpdateOperator method*), 396

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationCancelOperator method*), 398

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationGetOperator method*), 396

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationPauseOperator method*), 397

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOp method*), 398

_validate_inputs() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOp method*), 397

_validate_max_instances() (*in module airflow.contrib.operators.gcp_function_operator*), 376

_validate_value() (*in module airflow.contrib.hooks.bigquery_hook*), 558

_verify_archive_url_and_zip_path() (*airflow.contrib.operators.gcp_function_operator.ZipPathPreprocessor method*), 377

_verify_data_source() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobValidator method*), 394

_verify_source() (*airflow.contrib.operators.gcp_cloud_build_operator.BuildProcessor method*), 368

_verify_upload_url_and_no_zip_path() (*airflow.contrib.operators.gcp_function_operator.ZipPathPreprocess method*), 377

_verify_upload_url_and_zip_path() (*airflow.contrib.operators.gcp_function_operator.ZipPathPreprocessor method*), 377

_wait_for_job_done() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 585

_wait_for_operation_to_complete() (*airflow.contrib.hooks.gcp_cloud_build_hook.CloudBuildHook method*), 574

_wait_for_operation_to_complete() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 577

_wait_for_operation_to_complete() (*airflow.contrib.hooks.gcp_function_hook.GcfHook method*), 582

_wait_for_operation_to_complete() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 596

_wait_for_task_ended() (*airflow.contrib.operators.awsbatch_operator.AWSBatchOperator method*), 317

_wait_for_task_ended() (*airflow.contrib.operators.ecs_operator.ECSOperator method*), 361

_write_args() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298

_write_local_data_files() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS method*), 335

_write_local_data_files() (*air-*

flow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOperator method), 449
_write_local_data_files() (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator method*), 451
_write_local_data_files() (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleCloudStorageOperator method*), 455
_write_local_schema_file() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator method*), 335
_write_local_schema_file() (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOperator method*), 449
_write_local_schema_file() (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator method*), 451
_write_local_schema_file() (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleCloudStorageOperator method*), 455
_write_script() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
_write_string_args() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
_write_temp_file() (*airflow.contrib.operators.oracle_to_azure_data_lake_transfer.OracleToAzureDataLakeTransfer method*), 454

A

ABORTED (*airflow.contrib.hooks.gcp_transfer_hook.GcpTransferOperationStatus attribute*), 600
access_control (*airflow.models.DAG attribute*), 689
access_control (*airflow.models.dag.DAG attribute*), 662
ACCESS_KEY_ID (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
acknowledge() (*airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method*), 589
add_archive_uris() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
add_args() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
add_file_uris() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
add_jar_file_uris() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
add_only_new() (*airflow.models.BaseOperator method*), 686
add_only_new() (*airflow.models.baseoperator.BaseOperator method*), 658
add_product_to_product_set() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
add_python_file_uris() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
add_query() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
add_query_uri() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
add_task() (*airflow.models.DAG method*), 692
add_task() (*airflow.models.dag.DAG method*), 665
add_tasks() (*airflow.models.DAG method*), 692
add_tasks() (*airflow.models.dag.DAG method*), 665
add_labels() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 580
AdlsToGoogleCloudStorageOperator (*class in airflow.contrib.operators.adls_to_gcs*), 314
aggregate() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
airflow.contrib.hooks (*module*), 535
airflow.contrib.hooks.aws_athena_hook (*module*), 535
airflow.contrib.hooks.aws_dynamodb_hook (*module*), 536
airflow.contrib.hooks.aws_firehose_hook (*module*), 537
airflow.contrib.hooks.aws_glue_catalog_hook (*module*), 537
airflow.contrib.hooks.aws_hook (*module*), 538
airflow.contrib.hooks.aws_lambda_hook (*module*), 539
airflow.contrib.hooks.aws_sns_hook (*module*), 540
airflow.contrib.hooks.aws_sqs_hook (*module*), 540
airflow.contrib.hooks.azure_container_instance_hoo (*module*), 541
airflow.contrib.hooks.azure_container_registry_hoo (*module*), 542
airflow.contrib.hooks.azure_container_volume_hook (*module*), 542
airflow.contrib.hooks.azure_cosmos_hook (*module*), 543
airflow.contrib.hooks.azure_data_lake_hook (*module*), 544
airflow.contrib.hooks.azure_fileshare_hook (*module*), 545

airflow.contrib.hooks.bigquery_hook (*module*), 547
airflow.contrib.hooks.cassandra_hook (*module*), 559
airflow.contrib.hooks.cloudant_hook (*module*), 560
airflow.contrib.hooks.databricks_hook (*module*), 560
airflow.contrib.hooks.datadog_hook (*module*), 562
airflow.contrib.hooks.datastore_hook (*module*), 563
airflow.contrib.hooks.dingding_hook (*module*), 566
airflow.contrib.hooks.discord_webhook_hook (*module*), 567
airflow.contrib.hooks.emr_hook (*module*), 568
airflow.contrib.hooks.fs_hook (*module*), 568
airflow.contrib.hooks.ftp_hook (*module*), 568
airflow.contrib.hooks.gcp_api_base_hook (*module*), 571
airflow.contrib.hooks.gcp_bigtable_hook (*module*), 572
airflow.contrib.hooks.gcp_cloud_build_hook (*module*), 574
airflow.contrib.hooks.gcp_compute_hook (*module*), 575
airflow.contrib.hooks.gcp_container_hook (*module*), 577
airflow.contrib.hooks.gcp_dataflow_hook (*module*), 579
airflow.contrib.hooks.gcp_dataproc_hook (*module*), 580
airflow.contrib.hooks.gcp_function_hook (*module*), 581
airflow.contrib.hooks.gcp_kms_hook (*module*), 583
airflow.contrib.hooks.gcp_mlengine_hook (*module*), 584
airflow.contrib.hooks.gcp_natural_language_hook (*module*), 585
airflow.contrib.hooks.gcp_pubsub_hook (*module*), 588
airflow.contrib.hooks.gcp_spanner_hook (*module*), 590
airflow.contrib.hooks.gcp_speech_to_text_hook (*module*), 593
airflow.contrib.hooks.gcp_sql_hook (*module*), 593
airflow.contrib.hooks.gcp_text_to_speech_hook (*module*), 599
airflow.contrib.hooks.gcp_transfer_hook (*module*), 600
airflow.contrib.hooks.gcp_translate_hook (*module*), 604
airflow.contrib.hooks.gcp_video_intelligence_hook (*module*), 605
airflow.contrib.hooks.gcp_vision_hook (*module*), 606
airflow.contrib.hooks.gcs_hook (*module*), 607
airflow.contrib.hooks.grpc_hook (*module*), 611
airflow.contrib.hooks.imap_hook (*module*), 612
airflow.contrib.hooks.jenkins_hook (*module*), 614
airflow.contrib.hooks.jira_hook (*module*), 614
airflow.contrib.hooks.mongo_hook (*module*), 614
airflow.contrib.hooks.openfaas_hook (*module*), 617
airflow.contrib.hooks.opsgenie_alert_hook (*module*), 617
airflow.contrib.hooks.pinot_hook (*module*), 618
airflow.contrib.hooks.qubole_check_hook (*module*), 618
airflow.contrib.hooks.qubole_hook (*module*), 619
airflow.contrib.hooks.redis_hook (*module*), 620
airflow.contrib.hooks.redshift_hook (*module*), 620
airflow.contrib.hooks.sagemaker_hook (*module*), 621
airflow.contrib.hooks.salesforce_hook (*module*), 626
airflow.contrib.hooks.segment_hook (*module*), 627
airflow.contrib.hooks.sftp_hook (*module*), 628
airflow.contrib.hooks.slack_webhook_hook (*module*), 629
airflow.contrib.hooks.snowflake_hook (*module*), 630
airflow.contrib.hooks.spark_jdbc_hook (*module*), 631
airflow.contrib.hooks.spark_jdbc_script (*module*), 633
airflow.contrib.hooks.spark_sql_hook (*module*), 633
airflow.contrib.hooks.spark_submit_hook (*module*), 635

airflow.contrib.hooks.sqoop_hook (*module*), 637
airflow.contrib.hooks.ssh_hook (*module*), 640
airflow.contrib.hooks.vertica_hook (*module*), 640
airflow.contrib.hooks.wasb_hook (*module*), 641
airflow.contrib.hooks.winrm_hook (*module*), 643
airflow.contrib.operators (*module*), 313
airflow.contrib.operators.adls_list_operator (*module*), 313
airflow.contrib.operators.adls_to_gcs (*module*), 314
airflow.contrib.operators.aws_athena_operator (*module*), 315
airflow.contrib.operators.aws_sqs_publish_operator (*module*), 316
airflow.contrib.operators.awsbatch_operator (*module*), 317
airflow.contrib.operators.azure_container_instances_operator (*module*), 318
airflow.contrib.operators.azure_cosmos_operator (*module*), 319
airflow.contrib.operators.bigquery_check_operator (*module*), 320
airflow.contrib.operators.bigquery_get_data (*module*), 322
airflow.contrib.operators.bigquery_operator (*module*), 323
airflow.contrib.operators.bigquery_table_delete_operator (*module*), 330
airflow.contrib.operators.bigquery_to_bigquery (*module*), 331
airflow.contrib.operators.bigquery_to_gcs (*module*), 332
airflow.contrib.operators.cassandra_to_gcs (*module*), 334
airflow.contrib.operators.databricks_operator (*module*), 335
airflow.contrib.operators.dataflow_operator (*module*), 341
airflow.contrib.operators.dataproc_operator (*module*), 345
airflow.contrib.operators.datastore_export_operator (*module*), 356
airflow.contrib.operators.datastore_import_operator (*module*), 357
airflow.contrib.operators.dingding_operator (*module*), 358
airflow.contrib.operators.discord_webhook_operator (*module*), 359
airflow.contrib.operators.druid_operator (*module*), 359
airflow.contrib.operators.ecs_operator (*module*), 360
airflow.contrib.operators.emr_add_steps_operator (*module*), 361
airflow.contrib.operators.emr_create_job_flow_opera (*module*), 361
airflow.contrib.operators.emr_terminate_job_flow_op (*module*), 362
airflow.contrib.operators.file_to_gcs (*module*), 362
airflow.contrib.operators.file_to_wasb (*module*), 363
airflow.contrib.operators.gcp_bigtable_operator (*module*), 363
airflow.contrib.operators.gcp_cloud_build_operator (*module*), 367
airflow.contrib.operators.gcp_compute_operator (*module*), 369
airflow.contrib.operators.gcp_container_operator (*module*), 373
airflow.contrib.operators.gcp_function_operator (*module*), 376
airflow.contrib.operators.gcp_natural_language_oper (*module*), 378
airflow.contrib.operators.gcp_spanner_operator (*module*), 381
airflow.contrib.operators.gcp_speech_to_text_operat (*module*), 384
airflow.contrib.operators.gcp_sql_operator (*module*), 385
airflow.contrib.operators.gcp_text_to_speech_operat (*module*), 392
airflow.contrib.operators.gcp_transfer_operator (*module*), 394
airflow.contrib.operators.gcp_translate_operator (*module*), 401
airflow.contrib.operators.gcp_translate_speech_oper (*module*), 402
airflow.contrib.operators.gcp_video_intelligence_op (*module*), 404
airflow.contrib.operators.gcp_vision_operator (*module*), 406
airflow.contrib.operators.gcs_acl_operator (*module*), 422
airflow.contrib.operators.gcs_delete_operator (*module*), 423
airflow.contrib.operators.gcs_download_operator (*module*), 424
airflow.contrib.operators.gcs_list_operator (*module*), 425
airflow.contrib.operators.gcs_operator (*module*), 426
airflow.contrib.operators.gcs_to_bq

(*module*), 427
airflow.contrib.operators.gcs_to_gcs (*module*), 430
airflow.contrib.operators.gcs_to_s3 (*module*), 432
airflow.contrib.operators.grpc_operator (*module*), 434
airflow.contrib.operators.hipchat_operator (*module*), 435
airflow.contrib.operators.hive_to_dynamodb (*module*), 436
airflow.contrib.operators.imap_attachment_to_s3_operator (*module*), 437
airflow.contrib.operators.jenkins_job_trigger_operator (*module*), 437
airflow.contrib.operators.jira_operator (*module*), 439
airflow.contrib.operators.kubernetes_pod_operator (*module*), 441
airflow.contrib.operators.mlengine_operator (*module*), 443
airflow.contrib.operators.mongo_to_s3 (*module*), 447
airflow.contrib.operators.mssql_to_gcs (*module*), 448
airflow.contrib.operators.mysql_to_gcs (*module*), 450
airflow.contrib.operators.opsgenie_alert_operator (*module*), 452
airflow.contrib.operators.oracle_to_azure_data_lake_transfer (*module*), 453
airflow.contrib.operators.oracle_to_oracle_transfer (*module*), 454
airflow.contrib.operators.postgres_to_gcs_operator (*module*), 455
airflow.contrib.operators.pubsub_operator (*module*), 456
airflow.contrib.operators.qubole_check_operator (*module*), 460
airflow.contrib.operators.qubole_operator (*module*), 462
airflow.contrib.operators.redis_publish_operator (*module*), 465
airflow.contrib.operators.s3_copy_object_operator (*module*), 465
airflow.contrib.operators.s3_delete_objects_operator (*module*), 466
airflow.contrib.operators.s3_list_operator (*module*), 467
airflow.contrib.operators.s3_to_gcs_operator (*module*), 468
airflow.contrib.operators.s3_to_sftp_operator (*module*), 469
airflow.contrib.operators.sagemaker_base_operator (*module*), 470
airflow.contrib.operators.sagemaker_endpoint_confi
(*module*), 470
airflow.contrib.operators.sagemaker_endpoint_operat
(*module*), 471
airflow.contrib.operators.sagemaker_model_operator (*module*), 472
airflow.contrib.operators.sagemaker_training_operat
(*module*), 472
airflow.contrib.operators.sagemaker_transform_opera
(*module*), 473
airflow.contrib.operators.sagemaker_tuning_operato
(*module*), 474
airflow.contrib.operators.segment_track_event_opera
(*module*), 475
airflow.contrib.operators.sftp_operator (*module*), 475
airflow.contrib.operators.sftp_to_s3_operator (*module*), 477
airflow.contrib.operators.slack_webhook_operator (*module*), 478
airflow.contrib.operators.snowflake_operator (*module*), 479
airflow.contrib.operators.sns_publish_operator (*module*), 479
airflow.contrib.operators.spark_jdbc_operator (*module*), 481
airflow.contrib.operators.spark_sql_operator (*module*), 483
airflow.contrib.operators.spark_submit_operator (*module*), 485
airflow.contrib.operators.sqoop_operator (*module*), 487
airflow.contrib.operators.ssh_operator (*module*), 488
airflow.contrib.operators.vertica_operator (*module*), 488
airflow.contrib.operators.vertica_to_hive (*module*), 489
airflow.contrib.operators.vertica_to_mysql (*module*), 490
airflow.contrib.operators.wasb_delete_blob_operato
(*module*), 491
airflow.contrib.operators.winrm_operator (*module*), 491
airflow.contrib.sensors (*module*), 492
airflow.contrib.sensors.aws_athena_sensor (*module*), 492
airflow.contrib.sensors.aws_glue_catalog_partition_
(*module*), 493
airflow.contrib.sensors.aws_redshift_cluster_senso
(*module*), 494
airflow.contrib.sensors.aws_sqs_sensor (*module*), 494

airflow.contrib.sensors.azure_cosmos_sensor (*module*), 495
airflow.contrib.sensors.bash_sensor (*module*), 495
airflow.contrib.sensors.bigquery_sensor (*module*), 496
airflow.contrib.sensors.cassandra_record_sensor (*module*), 496
airflow.contrib.sensors.cassandra_table_sensor (*module*), 497
airflow.contrib.sensors.celery_queue_sensor (*module*), 497
airflow.contrib.sensors.datadog_sensor (*module*), 498
airflow.contrib.sensors.emr_base_sensor (*module*), 498
airflow.contrib.sensors.emr_job_flow_sensor (*module*), 499
airflow.contrib.sensors.emr_step_sensor (*module*), 499
airflow.contrib.sensors.file_sensor (*module*), 499
airflow.contrib.sensors.ftp_sensor (*module*), 500
airflow.contrib.sensors.gcp_transfer_sensor (*module*), 500
airflow.contrib.sensors.gcs_sensor (*module*), 501
airflow.contrib.sensors.hdfs_sensor (*module*), 504
airflow.contrib.sensors.imap_attachment_sensor (*module*), 504
airflow.contrib.sensors.jira_sensor (*module*), 505
airflow.contrib.sensors.mongo_sensor (*module*), 506
airflow.contrib.sensors.pubsub_sensor (*module*), 506
airflow.contrib.sensors.python_sensor (*module*), 507
airflow.contrib.sensors.qubole_sensor (*module*), 507
airflow.contrib.sensors.redis_key_sensor (*module*), 508
airflow.contrib.sensors.redis_pub_sub_sensor (*module*), 508
airflow.contrib.sensors.sagemaker_base_sensor (*module*), 509
airflow.contrib.sensors.sagemaker_endpoint_sensor (*module*), 509
airflow.contrib.sensors.sagemaker_training_sensor (*module*), 510
airflow.contrib.sensors.sagemaker_transform_sensor (*module*), 510
airflow.contrib.sensors.sagemaker_tuning_sensor (*module*), 511
airflow.contrib.sensors.sftp_sensor (*module*), 511
airflow.contrib.sensors.wasb_sensor (*module*), 511
airflow.contrib.sensors.weekday_sensor (*module*), 512
AirflowException, 651
airflow.executors (*module*), 644
airflow.executors.base_executor (*module*), 644
airflow.executors.celery_executor (*module*), 645
airflow.executors.dask_executor (*module*), 646
airflow.executors.kubernetes_executor (*module*), 647
airflow.executors.local_executor (*module*), 649
airflow.executors.sequential_executor (*module*), 650
airflow.hooks (*module*), 514
airflow.hooks.base_hook (*module*), 517
airflow.hooks.dbapi_hook (*module*), 518
airflow.hooks.docker_hook (*module*), 520
airflow.hooks.druid_hook (*module*), 520
airflow.hooks.hdfs_hook (*module*), 521
airflow.hooks.hive_hooks (*module*), 521
airflow.hooks.http_hook (*module*), 527
airflow.hooks.jdbc_hook (*module*), 528
airflow.hooks.mssql_hook (*module*), 528
airflow.hooks.mysql_hook (*module*), 529
airflow.hooks.oracle_hook (*module*), 530
airflow.hooks.pig_hook (*module*), 531
airflow.hooks.postgres_hook (*module*), 531
airflow.hooks.presto_hook (*module*), 532
airflow.hooks.S3_hook (*module*), 514
airflow.hooks.samba_hook (*module*), 533
airflow.hooks.slack_hook (*module*), 533
airflow.hooks.sqlite_hook (*module*), 533
airflow.hooks.webhdfs_hook (*module*), 534
airflow.hooks.zendesk_hook (*module*), 534
AirflowKubernetesScheduler (*class in airflow.executors.kubernetes_executor*), 647
airflow.macros (*module*), 273
airflow.models (*module*), 653
airflow.models.base (*module*), 653
airflow.models.baseoperator (*module*), 654
airflow.models.connection (*module*), 659
airflow.models.crypto (*module*), 660
airflow.models.dag (*module*), 661
airflow.models.dagbag (*module*), 668
airflow.models.dagpickle (*module*), 669

airflow.models.dagrun (*module*), 670
airflow.models.errors (*module*), 671
airflow.models.kubernetes (*module*), 672
airflow.models.log (*module*), 672
airflow.models.pool (*module*), 673
airflow.models.skipmixin (*module*), 673
airflow.models.slamiss (*module*), 673
airflow.models.taskfail (*module*), 674
airflow.models.taskinstance (*module*), 674
airflow.models.taskreschedule (*module*), 679
airflow.models.variable (*module*), 680
airflow.models.xcom (*module*), 681
airflow.operators (*module*), 275
airflow.operators.bash_operator (*module*), 275
airflow.operators.check_operator (*module*), 276
airflow.operators.dagrun_operator (*module*), 278
airflow.operators.docker_operator (*module*), 279
airflow.operators.druid_check_operator (*module*), 281
airflow.operators.dummy_operator (*module*), 281
airflow.operators.email_operator (*module*), 282
airflow.operators.generic_transfer (*module*), 282
airflow.operators.hive_operator (*module*), 283
airflow.operators.hive_stats_operator (*module*), 284
airflow.operators.hive_to_druid (*module*), 285
airflow.operators.hive_to_mysql (*module*), 286
airflow.operators.hive_to_samba_operator (*module*), 287
airflow.operators.http_operator (*module*), 287
airflow.operators.jdbc_operator (*module*), 288
airflow.operators.latest_only_operator (*module*), 288
airflow.operators.mssql_operator (*module*), 289
airflow.operators.mssql_to_hive (*module*), 289
airflow.operators.mysql_operator (*module*), 290
airflow.operators.mysql_to_hive (*module*), 291
airflow.operators.oracle_operator (*module*), 292
airflow.operators.papermill_operator (*module*), 292
airflow.operators.pig_operator (*module*), 293
airflow.operators.postgres_operator (*module*), 293
airflow.operators.presto_check_operator (*module*), 294
airflow.operators.presto_to_mysql (*module*), 295
airflow.operators.python_operator (*module*), 296
airflow.operators.redshift_to_s3_operator (*module*), 299
airflow.operators.s3_file_transform_operator (*module*), 300
airflow.operators.s3_to_hive_operator (*module*), 301
airflow.operators.s3_to_redshift_operator (*module*), 302
airflow.operators.slack_operator (*module*), 303
airflow.operators.sqlite_operator (*module*), 304
airflow.operators.subdag_operator (*module*), 305
airflow.sensors (*module*), 305
airflow.sensors.base_sensor_operator (*module*), 305
airflow.sensors.external_task_sensor (*module*), 306
airflow.sensors.hdfs_sensor (*module*), 307
airflow.sensors.hive_partition_sensor (*module*), 308
airflow.sensors.http_sensor (*module*), 308
airflow.sensors.metastore_partition_sensor (*module*), 309
airflow.sensors.named_hive_partition_sensor (*module*), 310
airflow.sensors.s3_key_sensor (*module*), 310
airflow.sensors.s3_prefix_sensor (*module*), 311
airflow.sensors.sql_sensor (*module*), 312
airflow.sensors.time_delta_sensor (*module*), 312
airflow.sensors.time_sensor (*module*), 312
airflow.sensors.web_hdfs_sensor (*module*), 313
AirflowWebHDFSHookException, 534
allocate_ids() (*airflow.contrib.hooks.datastore_hook.DatastoreHook*

method), 563
ALREADY_EXISTING_IN_SINK (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
analyze_entities() (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLanguageHook method*), 585
analyze_entity_sentiment() (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLanguageHook method*), 586
analyze_sentiment() (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLanguageHook method*), 586
analyze_syntax() (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLanguageHook method*), 587
annotate_image() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
annotate_text() (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLanguageHook method*), 587
annotate_video() (*airflow.contrib.hooks.gcp_video_intelligence_hook.CloudVideoIntelligenceHook method*), 605
annotator_client() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606
app (*in module airflow.executors.celery_executor*), 645
are_dependencies_met() (*airflow.models.TaskInstance method*), 702
are_dependencies_met() (*airflow.models.taskinstance.TaskInstance method*), 677
are_dependents_done() (*airflow.models.TaskInstance method*), 702
are_dependents_done() (*airflow.models.taskinstance.TaskInstance method*), 677
argmin() (*in module airflow.contrib.hooks.sagemaker_hook*), 621
arn (*airflow.contrib.operators.awsbatch_operator.AWSBatchOperator attribute*), 317
arn (*airflow.contrib.operators.ecs_operator.ECSOperator attribute*), 360
arraysize (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor attribute*), 557
as_dict() (*airflow.executors.kubernetes_executor.KubernetesExecutorConfig method*), 647
AthenaSensor (*class in airflow.contrib.sensors.aws_athena_sensor*), 492
attributes (*airflow.operators.papermill_operator.NoteBook attribute*), 292
AWS_ACCESS_KEY (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
AWS_S3_DATA_SOURCE (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
AWSAthenaHook (*class in airflow.contrib.hooks.aws_athena_hook*), 535
AWSAthenaOperator (*class in airflow.contrib.operators.aws_athena_operator*), 315
AWSBatchOperator (*class in airflow.contrib.operators.awsbatch_operator*), 317
AwsDynamoDBHook (*class in airflow.contrib.hooks.aws_dynamodb_hook*), 536
AwsFirehoseHook (*class in airflow.contrib.hooks.aws_firehose_hook*), 537
AwsGlueCatalogHook (*class in airflow.contrib.hooks.aws_glue_catalog_hook*), 537
AwsGlueCatalogPartitionSensor (*class in airflow.contrib.sensors.aws_glue_catalog_partition_sensor*), 493
AwsHook (*class in airflow.contrib.hooks.aws_hook*), 539
AwsHook (*in module airflow.contrib.operators.gcp_transfer_operator*), 394
AwsLambdaHook (*class in airflow.contrib.hooks.aws_lambda_hook*), 539
AwsRedshiftClusterSensor (*class in airflow.contrib.sensors.aws_redshift_cluster_sensor*), 494
AwsSnsHook (*class in airflow.contrib.hooks.aws_sns_hook*), 540
AzureContainerInstanceHook (*class in airflow.contrib.hooks.azure_container_instance_hook*), 541
AzureContainerInstancesOperator (*class in airflow.contrib.operators.azure_container_instances_operator*), 318
AzureContainerRegistryHook (*class in airflow.contrib.hooks.azure_container_registry_hook*), 542
AzureContainerVolumeHook (*class in airflow.contrib.hooks.azure_container_volume_hook*), 542
AzureCosmosDBHook (*class in airflow.contrib.hooks.azure_cosmos_hook*), 543
AzureCosmosDocumentSensor (*class in airflow.contrib.sensors.azure_cosmos_sensor*), 495
AzureCosmosInsertDocumentOperator (*class in airflow.contrib.operators.azure_cosmos_operator*), 319
AzureDataLakeHook (*class in air-*

flow.contrib.hooks.azure_data_lake_hook), 544
AzureDataLakeStorageListOperator (*class in airflow.contrib.operators.adls_list_operator*), 313
AzureFileShareHook (*class in airflow.contrib.hooks.azure_fileshare_hook*), 545

B

bag_dag() (*airflow.models.DagBag method*), 695
bag_dag() (*airflow.models.dagbag.DagBag method*), 669
Base (*in module airflow.models*), 681
Base (*in module airflow.models.base*), 653
BaseExecutor (*class in airflow.executors*), 651
BaseExecutor (*class in airflow.executors.base_executor*), 644
BaseHook (*class in airflow.hooks.base_hook*), 517
BaseOperator (*class in airflow.models*), 681
BaseOperator (*class in airflow.models.baseoperator*), 654
BaseOperatorLink (*class in airflow.models.baseoperator*), 659
BaseSensorOperator (*class in airflow.sensors.base_sensor_operator*), 305
BashOperator (*class in airflow.operators.bash_operator*), 275
BashSensor (*class in airflow.contrib.sensors.bash_sensor*), 495
batch_annotate_images() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
begin_transaction() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 564
BigQueryBaseCursor (*class in airflow.contrib.hooks.bigquery_hook*), 548
BigQueryCheckOperator (*class in airflow.contrib.operators.bigquery_check_operator*), 320
BigQueryConnection (*class in airflow.contrib.hooks.bigquery_hook*), 548
BigQueryConsoleLink (*class in airflow.contrib.operators.bigquery_operator*), 323
BigQueryCreateEmptyDatasetOperator (*class in airflow.contrib.operators.bigquery_operator*), 330
BigQueryCreateEmptyTableOperator (*class in airflow.contrib.operators.bigquery_operator*), 325
BigQueryCreateExternalTableOperator (*class in airflow.contrib.operators.bigquery_operator*), 327
BigQueryCursor (*class in airflow.contrib.hooks.bigquery_hook*), 557
BigQueryDeleteDatasetOperator (*class in airflow.contrib.operators.bigquery_operator*), 329
BigQueryGetDataOperator (*class in airflow.contrib.operators.bigquery_get_data*), 322
BigQueryHook (*class in airflow.contrib.hooks.bigquery_hook*), 547
BigQueryIntervalCheckOperator (*class in airflow.contrib.operators.bigquery_check_operator*), 321
BigQueryOperator (*class in airflow.contrib.operators.bigquery_operator*), 323
BigQueryPandasConnector (*class in airflow.contrib.hooks.bigquery_hook*), 548
BigQueryTableDeleteOperator (*class in airflow.contrib.operators.bigquery_table_delete_operator*), 331
BigQueryTableSensor (*class in airflow.contrib.sensors.bigquery_sensor*), 496
BigQueryToBigQueryOperator (*class in airflow.contrib.operators.bigquery_to_bigquery*), 332
BigQueryToCloudStorageOperator (*class in airflow.contrib.operators.bigquery_to_gcs*), 333
BigQueryValueCheckOperator (*class in airflow.contrib.operators.bigquery_check_operator*), 321
BigtableClusterUpdateOperator (*class in airflow.contrib.operators.gcp_bigtable_operator*), 366
BigtableHook (*class in airflow.contrib.hooks.gcp_bigtable_hook*), 572
BigtableInstanceCreateOperator (*class in airflow.contrib.operators.gcp_bigtable_operator*), 364
BigtableInstanceDeleteOperator (*class in airflow.contrib.operators.gcp_bigtable_operator*), 365
BigtableTableCreateOperator (*class in airflow.contrib.operators.gcp_bigtable_operator*), 365
BigtableTableDeleteOperator (*class in airflow.contrib.operators.gcp_bigtable_operator*), 366
BigtableTableWaitForReplicationSensor (*class in airflow.contrib.operators.gcp_bigtable_operator*), 367
BigtableValidationMixin (*class in airflow.contrib.operators.gcp_bigtable_operator*), 363

BODY (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
BranchPythonOperator (*class in airflow.operators.python_operator*), 297
BUCKET_NAME (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
build() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
build_command_args() (*in module airflow.contrib.hooks.qubole_hook*), 619
build_job() (*airflow.contrib.operators.jenkins_job_trigger_operator.JenkinsJobTriggerOperator method*), 438
BuildProcessor (*class in airflow.contrib.operators.gcp_cloud_build_operator*), 367
bulk_dump() (*airflow.hooks.dbapi_hook.DbApiHook method*), 519
bulk_dump() (*airflow.hooks.mysql_hook.MySqlHook method*), 529
bulk_dump() (*airflow.hooks.postgres_hook.PostgresHook method*), 532
bulk_insert_rows() (*airflow.hooks.oracle_hook.OracleHook method*), 530
bulk_load() (*airflow.hooks.dbapi_hook.DbApiHook method*), 519
bulk_load() (*airflow.hooks.mysql_hook.MySqlHook method*), 529
bulk_load() (*airflow.hooks.postgres_hook.PostgresHook method*), 531

C

call() (*airflow.hooks.slack_hook.SlackHook method*), 533
call() (*airflow.hooks.zendesk_hook.ZendeskHook method*), 535
cancel() (*airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook method*), 581
cancel_query() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 554
cancel_run() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
CANCEL_RUN_ENDPOINT (*in module airflow.contrib.hooks.databricks_hook*), 560
cancel_transfer_operation() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 603
CassandraHook (*class in airflow.contrib.hooks.cassandra_hook*), 559
CassandraRecordSensor (*class in airflow.contrib.sensors.cassandra_record_sensor*), 496
CassandraTableSensor (*class in airflow.contrib.sensors.cassandra_table_sensor*), 497
CassandraToGoogleCloudStorageOperator (*class in airflow.contrib.operators.cassandra_to_gcs*), 334
catch_http_exception() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook static method*), 571
celery_configuration (*in module airflow.executors.celery_executor*), 645
CELERY_FETCH_ERR_MSG_HEADER (*in module airflow.executors.celery_executor*), 645
CELERY_SEND_ERR_MSG_HEADER (*in module airflow.executors.celery_executor*), 645
CeleryExecutor (*airflow.executors.Executors attribute*), 653
CeleryExecutor (*class in airflow.executors.celery_executor*), 646
CeleryQueueSensor (*class in airflow.contrib.sensors.celery_queue_sensor*), 497
change_state() (*airflow.executors.base_executor.BaseExecutor method*), 645
change_state() (*airflow.executors.BaseExecutor method*), 651
check_for_blob() (*airflow.contrib.hooks.wasb_hook.WasbHook method*), 641
check_for_bucket() (*airflow.hooks.S3_hook.S3Hook method*), 514
check_for_directory() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 545
check_for_file() (*airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook method*), 544
check_for_file() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 545
check_for_key() (*airflow.hooks.S3_hook.S3Hook method*), 515
check_for_named_partition() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 524
check_for_partition() (*airflow.contrib.hooks.aws_glue_catalog_hook.AwsGlueCatalogHook method*), 538
check_for_partition() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 523
check_for_path() (*airflow.hooks.webhdfs_hook.WebHDFSHook*

method), 534
check_for_prefix() (*airflow.contrib.hooks.wasb_hook.WasbHook method*), 641
check_for_prefix() (*airflow.hooks.S3_hook.S3Hook method*), 514
check_for_wildcard_key() (*airflow.hooks.S3_hook.S3Hook method*), 515
check_query_status() (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook method*), 536
check_response() (*airflow.hooks.http_hook.HttpHook method*), 527
check_s3_url() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 622
check_status() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 625
check_training_config() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 622
check_training_status_with_log() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 625
check_tuning_config() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 622
CheckOperator (*class in airflow.operators.check_operator*), 276
checkpoint_kube_worker_uuid() (*airflow.models.kubernetes.KubeWorkerIdentifier static method*), 672
checkpoint_kube_worker_uuid() (*airflow.models.KubeWorkerIdentifier static method*), 698
checkpoint_resource_version() (*airflow.models.KubeResourceVersion static method*), 698
checkpoint_resource_version() (*airflow.models.kubernetes.KubeResourceVersion static method*), 672
classify_text() (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLanguageHook method*), 587
cleanup_database_hook() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 599
clear() (*airflow.models.BaseOperator method*), 686
clear() (*airflow.models.baseoperator.BaseOperator method*), 658
clear() (*airflow.models.DAG method*), 691
clear() (*airflow.models.dag.DAG method*), 665
clear_dags() (*airflow.models.DAG class method*), 691
clear_dags() (*airflow.models.dag.DAG class method*), 665
clear_not_launched_queued_tasks() (*airflow.executors.kubernetes_executor.KubernetesExecutor method*), 649
clear_task_instances() (*in module airflow.models*), 700
clear_task_instances() (*in module airflow.models.taskinstance*), 674
clear_xcom_data() (*airflow.models.TaskInstance method*), 702
clear_xcom_data() (*airflow.models.taskinstance.TaskInstance method*), 677
cli() (*airflow.models.DAG method*), 693
cli() (*airflow.models.dag.DAG method*), 666
client (*airflow.contrib.operators.awsbatch_operator.AWSBatchOperator attribute*), 317
client (*airflow.contrib.operators.ecs_operator.ECSOperator attribute*), 360
close() (*airflow.contrib.hooks.bigquery_hook.BigQueryConnection method*), 548
close() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 557
close_conn() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569
close_conn() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
close_conn() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 628
closest_ds_partition() (*in module airflow.macros.hive*), 273
CLOUD_FUNCTION_VALIDATION (*in module airflow.contrib.operators.gcp_function_operator*), 376
CLOUD_SQL_CREATE_VALIDATION (*in module airflow.contrib.operators.gcp_sql_operator*), 386
CLOUD_SQL_DATABASE_CREATE_VALIDATION (*in module airflow.contrib.operators.gcp_sql_operator*), 386
CLOUD_SQL_DATABASE_PATCH_VALIDATION (*in module airflow.contrib.operators.gcp_sql_operator*), 386
CLOUD_SQL_EXPORT_VALIDATION (*in module airflow.contrib.operators.gcp_sql_operator*), 386
CLOUD_SQL_IMPORT_VALIDATION (*in module airflow.contrib.operators.gcp_sql_operator*), 386
CLOUD_SQL_PROXY_DOWNLOAD_URL (*in module airflow.contrib.hooks.gcp_sql_hook*), 596
CLOUD_SQL_PROXY_VERSION_DOWNLOAD_URL (*in module airflow.contrib.hooks.gcp_sql_hook*), 596
CLOUD_SQL_VALID_DATABASE_TYPES (*in module*

airflow.contrib.hooks.gcp_sql_hook), 597
CloudantHook (*class in airflow.contrib.hooks.cloudant_hook*), 560
CloudBuildCreateBuildOperator (*class in airflow.contrib.operators.gcp_cloud_build_operator*), 368
CloudBuildHook (*class in airflow.contrib.hooks.gcp_cloud_build_hook*), 574
CloudLanguageAnalyzeEntitiesOperator (*class in airflow.contrib.operators.gcp_natural_language_operator*), 378
CloudLanguageAnalyzeEntitySentimentOperator (*class in airflow.contrib.operators.gcp_natural_language_operator*), 379
CloudLanguageAnalyzeSentimentOperator (*class in airflow.contrib.operators.gcp_natural_language_operator*), 380
CloudLanguageClassifyTextOperator (*class in airflow.contrib.operators.gcp_natural_language_operator*), 380
CloudNaturalLanguageHook (*class in airflow.contrib.hooks.gcp_natural_language_hook*), 585
CloudSpannerHook (*class in airflow.contrib.hooks.gcp_spanner_hook*), 590
CloudSpannerInstanceDatabaseDeleteOperator (*class in airflow.contrib.operators.gcp_spanner_operator*), 384
CloudSpannerInstanceDatabaseDeployOperator (*class in airflow.contrib.operators.gcp_spanner_operator*), 383
CloudSpannerInstanceDatabaseQueryOperator (*class in airflow.contrib.operators.gcp_spanner_operator*), 382
CloudSpannerInstanceDatabaseUpdateOperator (*class in airflow.contrib.operators.gcp_spanner_operator*), 383
CloudSpannerInstanceDeleteOperator (*class in airflow.contrib.operators.gcp_spanner_operator*), 382
CloudSpannerInstanceDeployOperator (*class in airflow.contrib.operators.gcp_spanner_operator*), 381
CloudSqlBaseOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 386
CloudSqlDatabaseHook (*class in airflow.contrib.hooks.gcp_sql_hook*), 597
CloudSqlHook (*class in airflow.contrib.hooks.gcp_sql_hook*), 594
CloudSqlInstanceCreateOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 386
CloudSqlInstanceDatabaseCreateOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 388
CloudSqlInstanceDatabaseDeleteOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 389
CloudSqlInstanceDatabasePatchOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 388
CloudSqlInstanceDeleteOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 387
CloudSqlInstanceExportOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 390
CloudSqlInstanceImportOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 390
CloudSqlInstancePatchOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 387
CloudSqlOperationStatus (*class in airflow.contrib.hooks.gcp_sql_hook*), 593
CloudSqlProxyRunner (*class in airflow.contrib.hooks.gcp_sql_hook*), 596
CloudSqlQueryOperator (*class in airflow.contrib.operators.gcp_sql_operator*), 391
CloudTranslateHook (*class in airflow.contrib.hooks.gcp_translate_hook*), 604
CloudTranslateTextOperator (*class in airflow.contrib.operators.gcp_translate_operator*), 402
CloudVideoIntelligenceDetectVideoExplicitContentOpe
(*class in airflow.contrib.operators.gcp_video_intelligence_operator*), 405
CloudVideoIntelligenceDetectVideoLabelsOperator (*class in airflow.contrib.operators.gcp_video_intelligence_operator*), 404
CloudVideoIntelligenceDetectVideoShotsOperator (*class in airflow.contrib.operators.gcp_video_intelligence_operator*), 406
CloudVideoIntelligenceHook (*class in airflow.contrib.hooks.gcp_video_intelligence_hook*), 605
CloudVisionAddProductToProductSetOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 416
CloudVisionAnnotateImageOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 415
CloudVisionDetectDocumentTextOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 419
CloudVisionDetectImageLabelsOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 420
CloudVisionDetectImageSafeSearchOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 421
CloudVisionDetectTextOperator (*class in air-*

flow.contrib.operators.gcp_vision_operator), 418
CloudVisionHook (*class in airflow.contrib.hooks.gcp_vision_hook*), 606
CloudVisionProductCreateOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 410
CloudVisionProductDeleteOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 414
CloudVisionProductGetOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 411
CloudVisionProductSetCreateOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 407
CloudVisionProductSetDeleteOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 410
CloudVisionProductSetGetOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 407
CloudVisionProductSetUpdateOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 408
CloudVisionProductUpdateOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 412
CloudVisionReferenceImageCreateOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 415
CloudVisionRemoveProductFromProductSetOperator (*class in airflow.contrib.operators.gcp_vision_operator*), 417
cluster_status() (*airflow.contrib.hooks.redshift_hook.RedshiftHook method*), 620
cmd_mask_password() (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 638
COL_DELIM (*in module airflow.contrib.hooks.qubole_check_hook*), 618
collect_dags() (*airflow.models.DagBag method*), 695
collect_dags() (*airflow.models.dagbag.DagBag method*), 669
command() (*airflow.models.TaskInstance method*), 701
command() (*airflow.models.taskinstance.TaskInstance method*), 676
command_as_list() (*airflow.models.TaskInstance method*), 701
command_as_list() (*airflow.models.taskinstance.TaskInstance method*), 676
COMMAND_CLASSES (*in module airflow.contrib.hooks.qubole_hook*), 619
commit() (*airflow.contrib.hooks.bigquery_hook.BigQueryConnection method*), 548
commit() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 564
COMPLETE (*airflow.contrib.hooks.sagemaker_hook.LogState attribute*), 621
compose() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 611
concurrency (*airflow.models.DAG attribute*), 689
concurrency (*airflow.models.dag.DAG attribute*), 662
concurrency_reached (*airflow.models.DAG attribute*), 689
concurrency_reached (*airflow.models.dag.DAG attribute*), 663
conf (*airflow.models.DagRun attribute*), 696
conf (*airflow.models.dagrun.DagRun attribute*), 670
configure_s3_resources() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 622
CONN_ENV_PREFIX (*in module airflow.hooks.base_hook*), 517
conn_id (*airflow.models.Connection attribute*), 687
conn_id (*airflow.models.connection.Connection attribute*), 659
conn_name_attr (*airflow.contrib.hooks.bigquery_hook.BigQueryHook attribute*), 547
conn_name_attr (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook attribute*), 618
conn_name_attr (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook attribute*), 630
conn_name_attr (*airflow.contrib.hooks.vertica_hook.VerticaHook attribute*), 641
conn_name_attr (*airflow.hooks.dbapi_hook.DbApiHook attribute*), 518
conn_name_attr (*airflow.hooks.druid_hook.DruidDbApiHook attribute*), 520
conn_name_attr (*airflow.hooks.jdbc_hook.JdbcHook attribute*), 528
conn_name_attr (*airflow.hooks.mssql_hook.MsSqlHook attribute*), 528
conn_name_attr (*airflow.hooks.mysql_hook.MySqlHook attribute*), 529
conn_name_attr (*airflow.hooks.oracle_hook.OracleHook attribute*), 530
conn_name_attr (*air-

flow.hooks.postgres_hook.PostgresHook attribute), 531

conn_name_attr (*airflow.hooks.presto_hook.PrestoHook attribute*), 532

conn_name_attr (*airflow.hooks.sqlite_hook.SqliteHook attribute*), 533

conn_type (*airflow.contrib.hooks.mongo_hook.MongoHook attribute*), 614

conn_type (*airflow.models.Connection attribute*), 687

conn_type (*airflow.models.connection.Connection attribute*), 659

Connection (*class in airflow.models*), 687

Connection (*class in airflow.models.connection*), 659

CONNECTION_URIS (*in module airflow.contrib.hooks.gcp_sql_hook*), 597

connector (*airflow.hooks.dbapi_hook.DbApiHook attribute*), 518

construct_api_call_params() (*airflow.operators.slack_operator.SlackAPIOperator method*), 303

construct_api_call_params() (*airflow.operators.slack_operator.SlackAPIPostOperator method*), 304

construct_ingest_query() (*airflow.operators.hive_to_druid.HiveToDruidTransfer method*), 286

convert_array_types() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator class method*), 335

convert_map_type() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator class method*), 335

convert_tuple_type() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator class method*), 335

convert_types() (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOperator class method*), 449

convert_types() (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleCloudStorageOperator class method*), 455

convert_user_type() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator class method*), 335

convert_value() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator class method*), 335

copy() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 608

copy_expert() (*airflow.hooks.postgres_hook.PostgresHook method*), 531

copy_object() (*airflow.hooks.S3_hook.S3Hook method*), 517

core_section (*airflow.executors.kubernetes_executor.KubeConfig attribute*), 647

CQL_TYPE_MAP (*airflow.contrib.operators.cassandra_to_gcs.CassandraToG attribute*), 334

create_bucket() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 609

create_bucket() (*airflow.hooks.S3_hook.S3Hook method*), 514

create_build() (*airflow.contrib.hooks.gcp_cloud_build_hook.CloudBuildHook method*), 574

create_cluster() (*airflow.contrib.hooks.gcp_container_hook.GKEClusterHook method*), 579

create_cluster_snapshot() (*airflow.contrib.hooks.redshift_hook.RedshiftHook method*), 620

create_cmd_args() (*airflow.contrib.hooks.qubole_hook.QuboleHook method*), 619

create_collection() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543

create_connection() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598

create_dagrun() (*airflow.models.DAG method*), 693

create_dagrun() (*airflow.models.dag.DAG method*), 666

create_dagrun() (*airflow.models.dag.DagModel method*), 667

create_dagrun() (*airflow.models.DagModel method*), 694

create_database() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543

create_database() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 592

create_database() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 595

create_directory() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 546

create_directory() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569

create_directory() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 628

create_empty_dataset() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 555
create_empty_table() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 548
create_endpoint() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 624
create_endpoint_config() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 624
create_external_table() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 549
create_instance() (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook method*), 572
create_instance() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 591
create_instance() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 594
create_integer_fields() (*airflow.contrib.operators.sagemaker_endpoint_operator.SageMakerEndpointOperator method*), 472
create_integer_fields() (*airflow.contrib.operators.sagemaker_transform_operator.SageMakerTransformOperator method*), 473
create_job() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 584
create_job_flow() (*airflow.contrib.hooks.emr_hook.EmrHook method*), 568
create_job_template() (*airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook method*), 581
create_job_template() (*airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator method*), 351
create_model() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 585
create_model() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 624
create_new_function() (*airflow.contrib.hooks.gcp_function_hook.GcfHook method*), 582
create_or_update() (*airflow.contrib.hooks.azure_container_instance_hook.AzureContainerInstanceHook method*), 541
create_product() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606
create_product_set() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606
create_queue() (*airflow.contrib.hooks.aws_sqs_hook.SQSHook method*), 540
create_reference_image() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
create_subscription() (*airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method*), 589
create_table() (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook static method*), 573
create_topic() (*airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method*), 588
create_training_job() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 623
create_transfer_job() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 602
create_transform_job() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 623
create_tuning_job() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 623
create_tunnel() (*airflow.contrib.hooks.ssh_hook.SSHHook method*), 640
create_version() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 585
created_dttm (*airflow.models.DagPickle attribute*), 696
created_dttm (*airflow.models.dagpickle.DagPickle attribute*), 669
current_state() (*airflow.models.TaskInstance method*), 702
current_state() (*airflow.models.taskinstance.TaskInstance method*), 677
cursor() (*airflow.contrib.hooks.bigquery_hook.BigQueryConnection method*), 548
CYCLE_DONE (*airflow.models.DagBag attribute*), 695
CYCLE_DONE (*airflow.models.dagbag.DagBag attribute*), 668
CYCLE_IN_PROGRESS (*airflow.models.DagBag attribute*), 695

CYCLE_IN_PROGRESS (*airflow.models.dagbag.DagBag attribute*), 668
CYCLE_NEW (*airflow.models.DagBag attribute*), 695
CYCLE_NEW (*airflow.models.dagbag.DagBag attribute*), 668

D

dag (*airflow.models.BaseOperator attribute*), 684
dag (*airflow.models.baseoperator.BaseOperator attribute*), 657
dag (*airflow.models.DagRun attribute*), 696
dag (*airflow.models.dagrun.DagRun attribute*), 670
DAG (*class in airflow.models*), 687
DAG (*class in airflow.models.dag*), 661
dag_id (*airflow.models.BaseOperator attribute*), 684
dag_id (*airflow.models.baseoperator.BaseOperator attribute*), 657
dag_id (*airflow.models.DAG attribute*), 689
dag_id (*airflow.models.dag.DAG attribute*), 662
dag_id (*airflow.models.dag.DagModel attribute*), 667
dag_id (*airflow.models.DagModel attribute*), 694
dag_id (*airflow.models.DagRun attribute*), 696
dag_id (*airflow.models.dagrun.DagRun attribute*), 670
dag_id (*airflow.models.Log attribute*), 698
dag_id (*airflow.models.log.Log attribute*), 672
dag_id (*airflow.models.SlaMiss attribute*), 699
dag_id (*airflow.models.slamiss.SlaMiss attribute*), 674
dag_id (*airflow.models.TaskFail attribute*), 699
dag_id (*airflow.models.taskfail.TaskFail attribute*), 674
dag_id (*airflow.models.TaskInstance attribute*), 700
dag_id (*airflow.models.taskinstance.TaskInstance attribute*), 675
dag_id (*airflow.models.TaskReschedule attribute*), 705
dag_id (*airflow.models.taskreschedule.TaskReschedule attribute*), 679
dag_id (*airflow.models.XCom attribute*), 706
dag_id (*airflow.models.xcom.XCom attribute*), 681
dag_ids (*airflow.models.DagBag attribute*), 695
dag_ids (*airflow.models.dagbag.DagBag attribute*), 668
DagBag (*class in airflow.models*), 695
DagBag (*class in airflow.models.dagbag*), 668
dagbag_report() (*airflow.models.DagBag method*), 696
dagbag_report() (*airflow.models.dagbag.DagBag method*), 669
DagModel (*class in airflow.models*), 694
DagModel (*class in airflow.models.dag*), 667
DagPickle (*class in airflow.models*), 696
DagPickle (*class in airflow.models.dagpickle*), 669
DagRun (*class in airflow.models*), 696
DagRun (*class in airflow.models.dagrun*), 670
DagRunOrder (*class in airflow.operators.dagrun_operator*), 278
DaskExecutor (*airflow.executors.Executors attribute*), 653
DaskExecutor (*class in airflow.executors.dask_executor*), 646
DatabricksHook (*class in airflow.contrib.hooks.databricks_hook*), 560
DatabricksRunNowOperator (*class in airflow.contrib.operators.databricks_operator*), 338
DatabricksSubmitRunOperator (*class in airflow.contrib.operators.databricks_operator*), 336
DatadogHook (*class in airflow.contrib.hooks.datadog_hook*), 562
DatadogSensor (*class in airflow.contrib.sensors.datadog_sensor*), 498
DataFlowHook (*class in airflow.contrib.hooks.gcp_dataflow_hook*), 580
DataFlowJavaOperator (*class in airflow.contrib.operators.dataflow_operator*), 341
DataFlowPythonOperator (*class in airflow.contrib.operators.dataflow_operator*), 344
DataflowTemplateOperator (*class in airflow.contrib.operators.dataflow_operator*), 343
DataprocClusterCreateOperator (*class in airflow.contrib.operators.dataproc_operator*), 346
DataprocClusterDeleteOperator (*class in airflow.contrib.operators.dataproc_operator*), 350
DataprocClusterScaleOperator (*class in airflow.contrib.operators.dataproc_operator*), 349
DataProcHadoopOperator (*class in airflow.contrib.operators.dataproc_operator*), 354
DataProcHiveOperator (*class in airflow.contrib.operators.dataproc_operator*), 352
DataProcHook (*class in airflow.contrib.hooks.gcp_dataproc_hook*), 581
DataProcJobBaseOperator (*class in airflow.contrib.operators.dataproc_operator*), 351
DataprocOperationBaseOperator (*class in airflow.contrib.operators.dataproc_operator*), 346
DataProcPigOperator (*class in airflow.contrib.operators.dataproc_operator*), 352
DataProcPySparkOperator (*class in airflow.contrib.operators.dataproc_operator*), 354
DataProcSparkOperator (*class in airflow.contrib.operators.dataproc_operator*),

353
DataProcSparkSqlOperator (*class in airflow.contrib.operators.dataproc_operator*), 353
DataprocWorkflowTemplateInstantiateInlineOperator (*class in airflow.contrib.operators.dataproc_operator*), 355
DataprocWorkflowTemplateInstantiateOperator (*class in airflow.contrib.operators.dataproc_operator*), 355
DatastoreExportOperator (*class in airflow.contrib.operators.datastore_export_operator*), 356
DatastoreHook (*class in airflow.contrib.hooks.datastore_hook*), 563
DatastoreImportOperator (*class in airflow.contrib.operators.datastore_import_operator*), 357
date_range() (*airflow.models.DAG method*), 690
date_range() (*airflow.models.dag.DAG method*), 663
DAY (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
DayOfWeekSensor (*class in airflow.contrib.sensors.weekday_sensor*), 512
DbApiHook (*class in airflow.hooks.dbapi_hook*), 518
deactivate_stale_dags() (*airflow.models.DAG static method*), 693
deactivate_stale_dags() (*airflow.models.dag.DAG static method*), 666
deactivate_unknown_dags() (*airflow.models.DAG static method*), 693
deactivate_unknown_dags() (*airflow.models.dag.DAG static method*), 666
debug_info() (*airflow.models.Connection method*), 687
debug_info() (*airflow.models.connection.Connection method*), 660
decrpyt() (*airflow.models.crypto.NullFernet method*), 660
decrypt() (*airflow.contrib.hooks.gcp_kms_hook.GoogleCloudKMSHook method*), 583
default_conn_name (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook attribute*), 618
default_conn_name (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook attribute*), 630
default_conn_name (*airflow.contrib.hooks.vertica_hook.VerticaHook attribute*), 641
default_conn_name (*airflow.hooks.dbapi_hook.DbApiHook attribute*), 518
default_conn_name (*airflow.hooks.druid_hook.DruidDbApiHook attribute*), 520
default_conn_name (*airflow.hooks.jdbc_hook.JdbcHook attribute*), 528
default_conn_name (*airflow.hooks.mssql_hook.MsSqlHook attribute*), 528
default_conn_name (*airflow.hooks.mysql_hook.MySqlHook attribute*), 529
default_conn_name (*airflow.hooks.oracle_hook.OracleHook attribute*), 530
default_conn_name (*airflow.hooks.postgres_hook.PostgresHook attribute*), 531
default_conn_name (*airflow.hooks.presto_hook.PrestoHook attribute*), 532
default_conn_name (*airflow.hooks.sqlite_hook.SqliteHook attribute*), 533
DEFAULT_CPU (*in module airflow.contrib.operators.azure_container_instances_operator*), 318
DEFAULT_DATAFLOW_LOCATION (*in module airflow.contrib.hooks.gcp_dataflow_hook*), 579
DEFAULT_ENVIRONMENT_VARIABLES (*in module airflow.contrib.operators.azure_container_instances_operator*), 318
DEFAULT_EXECUTOR (*in module airflow.executors*), 653
DEFAULT_MEMORY_IN_GB (*in module airflow.contrib.operators.azure_container_instances_operator*), 318
default_pool_name (*airflow.models.Pool attribute*), 698
default_pool_name (*airflow.models.pool.Pool attribute*), 673
default_pool_open_slots() (*airflow.models.Pool static method*), 698
default_pool_open_slots() (*airflow.models.pool.Pool static method*), 673
DEFAULT_TARGET_PARTITION_SIZE (*in module airflow.operators.hive_to_druid*), 285
default_view (*airflow.models.dag.DagModel attribute*), 667
default_view (*airflow.models.DagModel attribute*), 694
DEFAULT_VOLUMES (*in module airflow.contrib.operators.azure_container_instances_operator*), 318
delete() (*airflow.contrib.hooks.azure_container_instance_hook.AzureCont*

method), 542
delete() (airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method), 609
delete() (airflow.models.Variable class method), 705
delete() (airflow.models.variable.Variable class method), 680
delete() (airflow.models.XCom class method), 706
delete() (airflow.models.xcom.XCom class method), 681
delete_cluster() (airflow.contrib.hooks.gcp_container_hook.GKEClusterHook method), 578
delete_cluster() (airflow.contrib.hooks.redshift_hook.RedshiftHook method), 620
delete_collection() (airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method), 543
delete_connection() (airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method), 599
delete_database() (airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method), 543
delete_database() (airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method), 592
delete_database() (airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method), 595
delete_dataset() (airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method), 556
delete_directory() (airflow.contrib.hooks.ftp_hook.FTPHook method), 569
delete_directory() (airflow.contrib.hooks.sftp_hook.SFTPHook method), 628
delete_document() (airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method), 543
delete_file() (airflow.contrib.hooks.ftp_hook.FTPHook method), 570
delete_file() (airflow.contrib.hooks.sftp_hook.SFTPHook method), 629
delete_file() (airflow.contrib.hooks.wasb_hook.WasbHook method), 642
delete_function() (airflow.contrib.hooks.gcp_function_hook.GcfHook method), 583
delete_instance() (airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook method), 572
delete_instance() (airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method), 591
delete_instance() (airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method), 595
delete_many() (airflow.contrib.hooks.mongo_hook.MongoHook method), 616
delete_objects() (airflow.hooks.S3_hook.S3Hook method), 517
delete_one() (airflow.contrib.hooks.mongo_hook.MongoHook method), 616
delete_operation() (airflow.contrib.hooks.datastore_hook.DatastoreHook method), 565
delete_pod() (airflow.executors.kubernetes_executor.AirflowKubernetesSc method), 648
delete_product() (airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method), 607
delete_product_set() (airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method), 606
delete_reference_image() (airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method), 607
delete_subscription() (airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method), 589
delete_table() (airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook method), 573
delete_topic() (airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method), 588
delete_transfer_job() (airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method), 602
delete_version() (airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method), 585
DELETED (airflow.contrib.hooks.gcp_transfer_hook.GcpTransferJobsStatus attribute), 600
DEPLOY_FUNCTION (airflow.contrib.hooks.openfaas_hook.OpenFaasHook attribute), 617
deploy_function() (airflow.contrib.hooks.openfaas_hook.OpenFaasHook method), 617
deps (airflow.models.BaseOperator attribute), 684

deps (*airflow.models.baseoperator.BaseOperator attribute*), 657
deps (*airflow.sensors.base_sensor_operator.BaseSensorOperator attribute*), 306
describe_cluster_snapshots() (*airflow.contrib.hooks.redshift_hook.RedshiftHook method*), 620
describe_directory() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569
describe_directory() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 628
describe_endpoint() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 625
describe_endpoint_config() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 625
describe_model() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 625
describe_object() (*airflow.contrib.hooks.salesforce_hook.SalesforceHook method*), 626
describe_training_job() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 624
describe_training_job_with_log() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 624
describe_transform_job() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 625
describe_tuning_job() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 624
description (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor attribute*), 557
description (*airflow.models.DAG attribute*), 689
description (*airflow.models.dag.DAG attribute*), 662
description (*airflow.models.dag.DagModel attribute*), 667
description (*airflow.models.DagModel attribute*), 694
description (*airflow.models.Pool attribute*), 698
description (*airflow.models.pool.Pool attribute*), 673
description (*airflow.models.SlaMiss attribute*), 699
description (*airflow.models.slamiss.SlaMiss attribute*), 674
DESCRIPTION (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600
DingdingHook (*class in airflow.contrib.hooks.dingding_hook*), 566
DingdingOperator (*class in airflow.contrib.operators.dingding_operator*), 358
DISABLED (*airflow.contrib.hooks.gcp_transfer_hook.GcpTransferJobsStatus attribute*), 600
DiscordWebhookHook (*class in airflow.contrib.hooks.discord_webhook_hook*), 567
DiscordWebhookOperator (*class in airflow.contrib.operators.discord_webhook_operator*), 359
DockerHook (*class in airflow.hooks.docker_hook*), 520
DockerOperator (*class in airflow.operators.docker_operator*), 279
document_text_detection() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
does_collection_exist() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
does_database_exist() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
does_function_exist() (*airflow.contrib.hooks.openfaas_hook.OpenFaasHook method*), 617
DONE (*airflow.contrib.hooks.gcp_compute_hook.GceOperationStatus attribute*), 575
DONE (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlOperationStatus attribute*), 594
download() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 608
download_file() (*airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook method*), 544
download_mail_attachments() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 612
downstream_list (*airflow.models.BaseOperator attribute*), 685
downstream_list (*airflow.models.baseoperator.BaseOperator attribute*), 657
downstream_task_ids (*airflow.models.BaseOperator attribute*), 685
downstream_task_ids (*airflow.models.baseoperator.BaseOperator attribute*), 657
DruidCheckOperator (*class in airflow.operators.druid_check_operator*), 281
DruidDbApiHook (*class in airflow.hooks.druid_hook*), 520
DruidHook (*class in airflow.hooks.druid_hook*), 520
DruidOperator (*class in airflow.contrib.operators.druid_operator*), 359

dry_run() (airflow.models.BaseOperator method), 686
dry_run() (airflow.models.baseoperator.BaseOperator method), 658
dry_run() (airflow.models.TaskInstance method), 703
dry_run() (airflow.models.taskinstance.TaskInstance method), 678
dry_run() (airflow.operators.hive_operator.HiveOperator method), 284
ds_add() (in module airflow.macros), 273
ds_format() (in module airflow.macros), 273
dttm (airflow.models.Log attribute), 698
dttm (airflow.models.log.Log attribute), 672
DummyOperator (class in airflow.operators.dummy_operator), 281
duration (airflow.models.TaskFail attribute), 699
duration (airflow.models.taskfail.TaskFail attribute), 674
duration (airflow.models.TaskInstance attribute), 700
duration (airflow.models.taskinstance.TaskInstance attribute), 675
duration (airflow.models.TaskReschedule attribute), 705
duration (airflow.models.taskreschedule.TaskReschedule attribute), 680

E

ECSOperator (class in airflow.contrib.operators.ecs_operator), 360
email_alert() (airflow.models.TaskInstance method), 704
email_alert() (airflow.models.taskinstance.TaskInstance method), 678
email_sent (airflow.models.SlaMiss attribute), 699
email_sent (airflow.models.slamiss.SlaMiss attribute), 674
EmailOperator (class in airflow.operators.email_operator), 282
EmrAddStepsOperator (class in airflow.contrib.operators.emr_add_steps_operator), 361
EmrBaseSensor (class in airflow.contrib.sensors.emr_base_sensor), 498
EmrCreateJobFlowOperator (class in airflow.contrib.operators.emr_create_job_flow_operator), 361
EmrHook (class in airflow.contrib.hooks.emr_hook), 568
EmrJobFlowSensor (class in airflow.contrib.sensors.emr_job_flow_sensor), 499
EmrStepSensor (class in airflow.contrib.sensors.emr_step_sensor), 499
EmrTerminateJobFlowOperator (class in airflow.contrib.operators.emr_terminate_job_flow_operator), 362
ENABLED (airflow.contrib.hooks.gcp_transfer_hook.GcpTransferJobsStatus attribute), 600
encrypt() (airflow.contrib.hooks.gcp_kms_hook.GoogleCloudKMSHook method), 583
encrypt() (airflow.models.crypto.NullFernet method), 660
end() (airflow.executors.base_executor.BaseExecutor method), 645
end() (airflow.executors.BaseExecutor method), 652
end() (airflow.executors.celery_executor.CeleryExecutor method), 646
end() (airflow.executors.dask_executor.DaskExecutor method), 646
end() (airflow.executors.kubernetes_executor.KubernetesExecutor method), 649
end() (airflow.executors.local_executor.LocalExecutor method), 650
end() (airflow.executors.local_executor.LocalExecutor._LimitedParallelism method), 650
end() (airflow.executors.local_executor.LocalExecutor._UnlimitedParallelism method), 650
end() (airflow.executors.LocalExecutor method), 652
end() (airflow.executors.LocalExecutor._LimitedParallelism method), 652
end() (airflow.executors.LocalExecutor._UnlimitedParallelism method), 652
end() (airflow.executors.sequential_executor.SequentialExecutor method), 651
end() (airflow.executors.SequentialExecutor method), 653
end_date (airflow.models.DagRun attribute), 696
end_date (airflow.models.dagrun.DagRun attribute), 670
end_date (airflow.models.TaskFail attribute), 699
end_date (airflow.models.taskfail.TaskFail attribute), 674
end_date (airflow.models.TaskInstance attribute), 700
end_date (airflow.models.taskinstance.TaskInstance attribute), 675
end_date (airflow.models.TaskReschedule attribute), 705
end_date (airflow.models.taskreschedule.TaskReschedule attribute), 679
endpoint_non_terminal_states (airflow.contrib.hooks.sagemaker_hook.SageMakerHook attribute), 621
error() (airflow.models.TaskInstance method), 702
error() (airflow.models.taskinstance.TaskInstance method), 677
error_code_pattern (airflow.contrib.sensors.ftp_sensor.FTPSensor attribute), 500
event (airflow.models.Log attribute), 698
event (airflow.models.log.Log attribute), 672

ExceptionWithTraceback (*class in airflow.executors.celery_executor*), 645
execute() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 557
execute() (*airflow.contrib.hooks.discord_webhook_hook.DiscordWebhookHook method*), 568
execute() (*airflow.contrib.hooks.opsgenie_alert_hook.OpsgenieAlertHook method*), 618
execute() (*airflow.contrib.hooks.qubole_hook.QuboleHook method*), 619
execute() (*airflow.contrib.hooks.slack_webhook_hook.SlackWebhookHook method*), 630
execute() (*airflow.contrib.operators.adls_list_operator.AzureDataLakeStorageListOperator method*), 314
execute() (*airflow.contrib.operators.adls_to_gcs.AdlsToGoogleCloudStorageOperator method*), 315
execute() (*airflow.contrib.operators.aws_athena_operator.AWSAthenaOperator method*), 316
execute() (*airflow.contrib.operators.aws_sqs_publish_operator.SQSPublishOperator method*), 316
execute() (*airflow.contrib.operators.awsbatch_operator.AWSBatchOperator method*), 317
execute() (*airflow.contrib.operators.azure_container_instances_operator.AzureContainerInstancesOperator method*), 319
execute() (*airflow.contrib.operators.azure_cosmos_operator.AzureCosmosInsertDocumentOperator method*), 320
execute() (*airflow.contrib.operators.bigquery_get_data.BigQueryGetDataOperator method*), 323
execute() (*airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyDatasetOperator method*), 330
execute() (*airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyTableOperator method*), 327
execute() (*airflow.contrib.operators.bigquery_operator.BigQueryCreateExternalTableOperator method*), 329
execute() (*airflow.contrib.operators.bigquery_operator.BigQueryDeleteDatasetOperator method*), 330
execute() (*airflow.contrib.operators.bigquery_operator.BigQueryOperator method*), 325
execute() (*airflow.contrib.operators.bigquery_table_delete_operator.BigQueryTableDeleteOperator method*), 331
execute() (*airflow.contrib.operators.bigquery_to_bigquery.BigQueryToBigQueryOperator method*), 332
execute() (*airflow.contrib.operators.bigquery_to_gcs.BigQueryToCloudStorageOperator method*), 334
execute() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator method*), 335
execute() (*airflow.contrib.operators.databricks_operator.DatabricksRunNowOperator method*), 340
execute() (*airflow.contrib.operators.databricks_operator.DatabricksSubmitRunOperator method*), 338
execute() (*airflow.contrib.operators.dataflow_operator.DataFlowJavaOperator method*), 343
execute() (*airflow.contrib.operators.dataflow_operator.DataFlowPythonOperator method*), 345
execute() (*airflow.contrib.operators.dataflow_operator.DataflowTemplate method*), 344
execute() (*airflow.contrib.operators.dataproc_operator.DataProcHadoop method*), 354
execute() (*airflow.contrib.operators.dataproc_operator.DataProcHiveOpe method*), 353
execute() (*airflow.contrib.operators.dataproc_operator.DataProcJobBase method*), 352
execute() (*airflow.contrib.operators.dataproc_operator.DataprocOperation method*), 346
execute() (*airflow.contrib.operators.dataproc_operator.DataProcPigOpera method*), 352
execute() (*airflow.contrib.operators.dataproc_operator.DataProcPySpark method*), 355
execute() (*airflow.contrib.operators.dataproc_operator.DataProcSparkOp method*), 354
execute() (*airflow.contrib.operators.dataproc_operator.DataProcSparkSql method*), 353
execute() (*airflow.contrib.operators.datastore_export_operator.DatastoreE method*), 357
execute() (*airflow.contrib.operators.datastore_import_operator.DatastoreI method*), 358
execute() (*airflow.contrib.operators.dingding_operator.DingdingOperator method*), 358
execute() (*airflow.contrib.operators.discord_webhook_operator.DiscordW method*), 359
execute() (*airflow.contrib.operators.druid_operator.DruidOperator method*), 360
execute() (*airflow.contrib.operators.ecs_operator.ECSOperator method*), 361
execute() (*airflow.contrib.operators.emr_add_steps_operator.EmrAddStep method*), 361
execute() (*airflow.contrib.operators.emr_create_job_flow_operator.EmrC method*), 362
execute() (*airflow.contrib.operators.emr_terminate_job_flow_operator.Em method*), 362
execute() (*airflow.contrib.operators.file_to_gcs.FileToGoogleCloudStorage method*), 363
execute() (*airflow.contrib.operators.file_to_wasb.FileToWasbOperator method*), 363
execute() (*airflow.contrib.operators.gcp_bigtable_operator.BigtableCluster method*), 367
execute() (*airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanc method*), 365
execute() (*airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanc method*), 365
execute() (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableC method*), 366
execute() (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableD method*), 366
execute() (*airflow.contrib.operators.gcp_cloud_build_operator.CloudBuild method*), 369
execute() (*airflow.contrib.operators.gcp_compute_operator.GceBaseOpera method*), 369

execute() (*airflow.contrib.operators.gcp_compute_operator.GceInstanceGroupManagerUpdateTemplateOperator method*), 373
execute() (*airflow.contrib.operators.gcp_compute_operator.GceInstanceStartOperator method*), 370
execute() (*airflow.contrib.operators.gcp_compute_operator.GceInstanceStopOperator method*), 370
execute() (*airflow.contrib.operators.gcp_compute_operator.GceInstanceTemplateCopyOperator method*), 372
execute() (*airflow.contrib.operators.gcp_compute_operator.GceSetMachineTypeOperator method*), 371
execute() (*airflow.contrib.operators.gcp_container_operator.GKEClusterCreateOperator method*), 375
execute() (*airflow.contrib.operators.gcp_container_operator.GKEClusterDeleteOperator method*), 374
execute() (*airflow.contrib.operators.gcp_container_operator.GKEPodOperator method*), 376
execute() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeleteOperator method*), 378
execute() (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator method*), 377
execute() (*airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageAnalyzeEntitiesOperator method*), 379
execute() (*airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageAnalyzeEntitySentimentOperator method*), 380
execute() (*airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageAnalyzeSentimentOperator method*), 380
execute() (*airflow.contrib.operators.gcp_natural_language_operator.CloudLanguageClassifyTextOperator method*), 381
execute() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeleteOperator method*), 384
execute() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeployOperator method*), 383
execute() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseQueryOperator method*), 383
execute() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseUpdateOperator method*), 384
execute() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDeleteOperator method*), 382
execute() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDeployOperator method*), 382
execute() (*airflow.contrib.operators.gcp_speech_to_text_operator.GcpSpeechToTextRecognizeSpeechOperator method*), 385
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlBaseOperator method*), 386
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceCreateOperator method*), 387
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseCreateOperator method*), 388
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseDeleteOperator method*), 390
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabasePatchOperator method*), 389
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDeleteOperator method*), 388
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceEx method*), 390
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceIm method*), 391
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstancePa method*), 387
execute() (*airflow.contrib.operators.gcp_sql_operator.CloudSqlQueryOper method*), 392
execute() (*airflow.contrib.operators.gcp_text_to_speech_operator.GcpText method*), 394
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 395
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 396
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 396
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 398
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 396
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 397
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 398
execute() (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferSe method*), 397
execute() (*airflow.contrib.operators.gcp_transfer_operator.GoogleCloudSt method*), 401
execute() (*airflow.contrib.operators.gcp_transfer_operator.S3ToGoogleClo method*), 400
execute() (*airflow.contrib.operators.gcp_translate_operator.CloudTranslat method*), 402
execute() (*airflow.contrib.operators.gcp_translate_speech_operator.GcpTr method*), 404
execute() (*airflow.contrib.operators.gcp_video_intelligence_operator.Clou method*), 405
execute() (*airflow.contrib.operators.gcp_video_intelligence_operator.Clou method*), 405
execute() (*airflow.contrib.operators.gcp_video_intelligence_operator.Clou method*), 406
execute() (*airflow.contrib.operators.gcp_vision_operator.CloudVisionAdd method*), 417
execute() (*airflow.contrib.operators.gcp_vision_operator.CloudVisionAnn method*), 415
execute() (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDete method*), 420
execute() (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDete method*), 421
execute() (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDete method*), 422
execute() (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDete method*), 419
execute() (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProd method*), 411

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductDeleteOperator method*), 415

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductGetOperator method*), 412

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetCreateOperator method*), 407

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetDeleteOperator method*), 410

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetGetOperator method*), 408

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetUpdateOperator method*), 410

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductUpdateOperator method*), 414

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionReferenceImageCreateOperator method*), 416

`execute()` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionRemoveProductFromProductSetOperator method*), 418

`execute()` (*airflow.contrib.operators.gcs_acl_operator.GoogleCloudStorageBucketCreateAclEntryOperator method*), 423

`execute()` (*airflow.contrib.operators.gcs_acl_operator.GoogleCloudStorageObjectCreateAclEntryOperator method*), 423

`execute()` (*airflow.contrib.operators.gcs_delete_operator.GoogleCloudStorageDeleteOperator method*), 424

`execute()` (*airflow.contrib.operators.gcs_download_operator.GoogleCloudStorageDownloadOperator method*), 425

`execute()` (*airflow.contrib.operators.gcs_list_operator.GoogleCloudStorageListOperator method*), 426

`execute()` (*airflow.contrib.operators.gcs_operator.GoogleCloudStorageCreateBucketOperator method*), 427

`execute()` (*airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBigQueryOperator method*), 430

`execute()` (*airflow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator method*), 432

`execute()` (*airflow.contrib.operators.gcs_to_s3.GoogleCloudStorageToS3Operator method*), 433

`execute()` (*airflow.contrib.operators.grpc_operator.GrpcOperator method*), 434

`execute()` (*airflow.contrib.operators.hipchat_operator.HipChatAPIOperator method*), 435

`execute()` (*airflow.contrib.operators.hive_to_dynamodb.HiveToDynamoDBTransferOperator method*), 436

`execute()` (*airflow.contrib.operators.imap_attachment_to_s3_operator.ImapAttachmentToS3Operator method*), 437

`execute()` (*airflow.contrib.operators.jenkins_job_trigger_operator.JenkinsJobTriggerOperator method*), 438

`execute()` (*airflow.contrib.operators.jira_operator.JiraOperator method*), 439

`execute()` (*airflow.contrib.operators.kubernetes_pod_operator.KubernetesPodOperator method*), 443

`execute()` (*airflow.contrib.operators.mlengine_operator.MLEngineBatchPredictionOperator method*), 444

`execute()` (*airflow.contrib.operators.mlengine_operator.MLEngineModelOperator method*), 445

`execute()` (*airflow.contrib.operators.mlengine_operator.MLEngineTraining method*), 447

`execute()` (*airflow.contrib.operators.mlengine_operator.MLEngineVersion method*), 446

`execute()` (*airflow.contrib.operators.mongo_to_s3.MongoToS3Operator method*), 448

`execute()` (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudSto method*), 449

`execute()` (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudSt method*), 451

`execute()` (*airflow.contrib.operators.opsgenie_alert_operator.OpsgenieAler method*), 453

`execute()` (*airflow.contrib.operators.oracle_to_azure_data_lake_transfer.O method*), 454

`execute()` (*airflow.contrib.operators.oracle_to_oracle_transfer.OracleToOr method*), 454

`execute()` (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresToG method*), 455

`execute()` (*airflow.contrib.operators.pubsub_operator.PubSubPublishOpera method*), 460

`execute()` (*airflow.contrib.operators.pubsub_operator.PubSubSubscriptionC method*), 458

`execute()` (*airflow.contrib.operators.pubsub_operator.PubSubSubscriptionD method*), 459

`execute()` (*airflow.contrib.operators.pubsub_operator.PubSubTopicCreateO method*), 456

`execute()` (*airflow.contrib.operators.pubsub_operator.PubSubTopicDeleteO method*), 458

`execute()` (*airflow.contrib.operators.qubole_check_operator.QuboleCheckO method*), 460

`execute()` (*airflow.contrib.operators.qubole_check_operator.QuboleValueC method*), 461

`execute()` (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464

`execute()` (*airflow.contrib.operators.redis_publish_operator.RedisPublishO method*), 465

`execute()` (*airflow.contrib.operators.s3_copy_object_operator.S3CopyObje method*), 466

`execute()` (*airflow.contrib.operators.s3_delete_objects_operator.S3DeleteO method*), 467

`execute()` (*airflow.contrib.operators.s3_list_operator.S3ListOperator method*), 468

`execute()` (*airflow.contrib.operators.s3_to_gcs_operator.S3ToGoogleCloud method*), 469

`execute()` (*airflow.contrib.operators.s3_to_sftp_operator.S3ToSFTPOperat method*), 469

`execute()` (*airflow.contrib.operators.sagemaker_base_operator.SageMaker method*), 470

`execute()` (*airflow.contrib.operators.sagemaker_endpoint_config_operator method*), 470

`execute()` (*airflow.contrib.operators.sagemaker_endpoint_operator.SageM method*), 472

`execute()` (*airflow.contrib.operators.sagemaker_model_operator.SageMak method*), 472

execute() (*airflow.contrib.operators.sagemaker_training_operator.SageMakerTrainingOperator method*), 473
execute() (*airflow.contrib.operators.sagemaker_transform_operator.SageMakerTransformOperator method*), 474
execute() (*airflow.contrib.operators.sagemaker_tuning_operator.SageMakerTuningOperator method*), 474
execute() (*airflow.contrib.operators.segment_track_event_operator.SegmentTrackEventOperator method*), 475
execute() (*airflow.contrib.operators.sftp_operator.SFTPOperator method*), 476
execute() (*airflow.contrib.operators.sftp_to_s3_operator.SFTPToS3Operator method*), 477
execute() (*airflow.contrib.operators.slack_webhook_operator.SlackWebhookOperator method*), 478
execute() (*airflow.contrib.operators.snowflake_operator.SnowflakeOperator method*), 479
execute() (*airflow.contrib.operators.sns_publish_operator.SnsPublishOperator method*), 480
execute() (*airflow.contrib.operators.spark_jdbc_operator.SparkJDBCOperator method*), 483
execute() (*airflow.contrib.operators.spark_sql_operator.SparkSqlOperator method*), 484
execute() (*airflow.contrib.operators.spark_submit_operator.SparkSubmitOperator method*), 486
execute() (*airflow.contrib.operators.sqoop_operator.SqoopOperator method*), 488
execute() (*airflow.contrib.operators.ssh_operator.SSHOperator method*), 488
execute() (*airflow.contrib.operators.vertica_operator.VerticaOperator method*), 489
execute() (*airflow.contrib.operators.vertica_to_hive.VerticaToHiveTransfer method*), 490
execute() (*airflow.contrib.operators.vertica_to_mysql.VerticaToMySqlTransfer method*), 490
execute() (*airflow.contrib.operators.wasb_delete_blob_operator.WasbDeleteBlobOperator method*), 491
execute() (*airflow.contrib.operators.winrm_operator.WinRMOperator method*), 491
execute() (*airflow.contrib.sensors.pubsub_sensor.PubSubPullSensor method*), 506
execute() (*airflow.models.BaseOperator method*), 685
execute() (*airflow.models.baseoperator.BaseOperator method*), 657
execute() (*airflow.operators.bash_operator.BashOperator method*), 275
execute() (*airflow.operators.check_operator.CheckOperator method*), 276
execute() (*airflow.operators.check_operator.IntervalCheckOperator method*), 277
execute() (*airflow.operators.check_operator.ValueCheckOperator method*), 277
execute() (*airflow.operators.dagrun_operator.TriggerDagRunOperator method*), 278
execute() (*airflow.operators.docker_operator.DockerOperator method*), 280
execute() (*airflow.operators.druid_check_operator.DruidCheckOperator method*), 281
execute() (*airflow.operators.dummy_operator.DummyOperator method*), 281
execute() (*airflow.operators.email_operator.EmailOperator method*), 282
execute() (*airflow.operators.generic_transfer.GenericTransfer method*), 283
execute() (*airflow.operators.hive_operator.HiveOperator method*), 284
execute() (*airflow.operators.hive_stats_operator.HiveStatsCollectionOpera method*), 285
execute() (*airflow.operators.hive_to_druid.HiveToDruidTransfer method*), 286
execute() (*airflow.operators.hive_to_mysql.HiveToMySqlTransfer method*), 286
execute() (*airflow.operators.hive_to_samba_operator.Hive2SambaOperat method*), 287
execute() (*airflow.operators.http_operator.SimpleHttpOperator method*), 288
execute() (*airflow.operators.jdbc_operator.JdbcOperator method*), 288
execute() (*airflow.operators.latest_only_operator.LatestOnlyOperator method*), 288
execute() (*airflow.operators.mssql_operator.MsSqlOperator method*), 289
execute() (*airflow.operators.mssql_to_hive.MsSqlToHiveTransfer method*), 290
execute() (*airflow.operators.mysql_operator.MySqlOperator method*), 291
execute() (*airflow.operators.mysql_to_hive.MySqlToHiveTransfer method*), 291
execute() (*airflow.operators.oracle_operator.OracleOperator method*), 292
execute() (*airflow.operators.papermill_operator.PapermillOperator method*), 292
execute() (*airflow.operators.pig_operator.PigOperator method*), 293
execute() (*airflow.operators.postgres_operator.PostgresOperator method*), 294
execute() (*airflow.operators.presto_to_mysql.PrestoToMySqlTransfer method*), 296
execute() (*airflow.operators.python_operator.BranchPythonOperator method*), 297
execute() (*airflow.operators.python_operator.PythonOperator method*), 296
execute() (*airflow.operators.python_operator.ShortCircuitOperator method*), 297
execute() (*airflow.operators.redshift_to_s3_operator.RedshiftToS3Transfe method*), 299
execute() (*airflow.operators.s3_file_transform_operator.S3FileTransform method*), 300
execute() (*airflow.operators.s3_to_hive_operator.S3ToHiveTransfer*

method), 302
execute() (*airflow.operators.s3_to_redshift_operator.S3ToRedshiftTransfer method*), 303
execute() (*airflow.operators.slack_operator.SlackAPIOperator method*), 303
execute() (*airflow.operators.sqlite_operator.SqliteOperator method*), 304
execute() (*airflow.operators.subdag_operator.SubDagOperator method*), 305
execute() (*airflow.sensors.base_sensor_operator.BaseSensorOperator method*), 306
execute_async() (*airflow.executors.base_executor.BaseExecutor method*), 645
execute_async() (*airflow.executors.BaseExecutor method*), 652
execute_async() (*airflow.executors.dask_executor.DaskExecutor method*), 646
execute_async() (*airflow.executors.kubernetes_executor.KubernetesExecutor method*), 649
execute_async() (*airflow.executors.local_executor.LocalExecutor method*), 650
execute_async() (*airflow.executors.local_executor.LocalExecutor._LimitedParallelism method*), 650
execute_async() (*airflow.executors.local_executor.LocalExecutor._UnlimitedParallelism method*), 650
execute_async() (*airflow.executors.LocalExecutor method*), 652
execute_async() (*airflow.executors.LocalExecutor._LimitedParallelism method*), 652
execute_async() (*airflow.executors.LocalExecutor._UnlimitedParallelism method*), 652
execute_async() (*airflow.executors.sequential_executor.SequentialExecutor method*), 651
execute_async() (*airflow.executors.SequentialExecutor method*), 652
execute_callable() (*airflow.operators.python_operator.PythonOperator method*), 296
execute_callable() (*airflow.operators.python_operator.PythonVirtualenvOperator method*), 298
execute_command() (*in module airflow.executors.celery_executor*), 645
execute_dml() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 592
execute_work() (*airflow.executors.local_executor.LocalWorker method*), 649
executemany() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 557
execution_date (*airflow.models.DagRun attribute*), 696
execution_date (*airflow.models.dagrun.DagRun attribute*), 670
execution_date (*airflow.models.Log attribute*), 698
execution_date (*airflow.models.log.Log attribute*), 672
execution_date (*airflow.models.SlaMiss attribute*), 699
execution_date (*airflow.models.slamiss.SlaMiss attribute*), 674
execution_date (*airflow.models.TaskFail attribute*), 699
execution_date (*airflow.models.taskfail.TaskFail attribute*), 674
execution_date (*airflow.models.TaskInstance attribute*), 700
execution_date (*airflow.models.taskinstance.TaskInstance attribute*), 675
execution_date (*airflow.models.TaskReschedule attribute*), 705
execution_date (*airflow.models.taskreschedule.TaskReschedule attribute*), 679
execution_date (*airflow.models.XCom attribute*), 706
execution_date (*airflow.models.xcom.XCom attribute*), 681
executor_config (*airflow.models.TaskInstance attribute*), 700
executor_config (*airflow.models.taskinstance.TaskInstance attribute*), 675
Executors (*class in airflow.executors*), 653
exists() (*airflow.contrib.hooks.azure_container_instance_hook.AzureCont method*), 542
exists() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 608
expand_role() (*airflow.contrib.hooks.aws_hook.AwsHook method*), 539
expand_role() (*airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOp method*), 470
expand_role() (*airflow.contrib.operators.sagemaker_endpoint_operator.SageMakerEnd*

method), 472
expand_role() (*airflow.contrib.operators.sagemaker_model_operator.SageMakerModelOperator method*), 472
expand_role() (*airflow.contrib.operators.sagemaker_training_operator.SageMakerTrainingOperator method*), 473
expand_role() (*airflow.contrib.operators.sagemaker_transform_operator.SageMakerTransformOperator method*), 474
expand_role() (*airflow.contrib.operators.sagemaker_tuning_operator.SageMakerTuningOperator method*), 474
export_instance() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 596
export_table() (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 639
export_to_storage_bucket() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 565
external_trigger (*airflow.models.DagRun attribute*), 696
external_trigger (*airflow.models.dagrun.DagRun attribute*), 670
ExternalTaskSensor (*class in airflow.sensors.external_task_sensor*), 306
extra (*airflow.models.Connection attribute*), 687
extra (*airflow.models.connection.Connection attribute*), 660
extra (*airflow.models.Log attribute*), 698
extra (*airflow.models.log.Log attribute*), 672
extra_dejson (*airflow.models.Connection attribute*), 687
extra_dejson (*airflow.models.connection.Connection attribute*), 660
extra_links() (*airflow.models.BaseOperator method*), 686
extra_links() (*airflow.models.baseoperator.BaseOperator method*), 659

F

fail() (*airflow.executors.base_executor.BaseExecutor method*), 645
fail() (*airflow.executors.BaseExecutor method*), 651
FAILED (*airflow.contrib.hooks.gcp_transfer_hook.GcpTransferOperationStatus attribute*), 600
FAILED_STATE (*airflow.contrib.sensors.emr_job_flow_sensor.EmrJobFlowSensor attribute*), 499
FAILED_STATE (*airflow.contrib.sensors.emr_step_sensor.EmrStepSensor attribute*), 499
failed_states (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook attribute*), 621
failed_states() (*airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor method*), 509
failed_states() (*airflow.contrib.sensors.sagemaker_endpoint_sensor.SageMakerEndpoi method*), 509
failed_states() (*airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTraining method*), 510
failed_states() (*airflow.contrib.sensors.sagemaker_transform_sensor.SageMakerTrans method*), 510
failed_states() (*airflow.contrib.sensors.sagemaker_tuning_sensor.SageMakerTuningSen method*), 511
FAILURE_STATES (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook attribute*), 535
FAILURE_STATES (*airflow.contrib.sensors.aws_athena_sensor.AthenaSensor attribute*), 492
fallback_to_default_project_id() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook static method*), 571
fetch_celery_task_state() (*in module airflow.executors.celery_executor*), 645
fetchall() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558
fetchmany() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558
fetchone() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558
fileloc (*airflow.models.dag.DagModel attribute*), 667
fileloc (*airflow.models.DagModel attribute*), 694
filename (*airflow.models.errors.ImportError attribute*), 671
filepath (*airflow.models.DAG attribute*), 689
filepath (*airflow.models.dag.DAG attribute*), 662
FileSensor (*class in airflow.contrib.sensors.file_sensor*), 499
FileToGoogleCloudStorageOperator (*class in airflow.contrib.operators.file_to_gcs*), 362
FileToWasbOperator (*class in airflow.contrib.operators.file_to_wasb*), 363
FILTER (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
filter_for_filesize() (*airflow.sensors.hdfs_sensor.HdfsSensor static method*), 307
filter_for_ignored_ext() (*airflow.sensors.hdfs_sensor.HdfsSensor static*

method), 307
FILTER_JOB_NAMES (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
filter_options() (*in module airflow.contrib.hooks.qubole_hook*), 619
FILTER_PROJECT_ID (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
find() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
find() (*airflow.models.DagRun static method*), 697
find() (*airflow.models.dagrun.DagRun static method*), 670
find_for_task_instance() (*airflow.models.TaskReschedule static method*), 705
find_for_task_instance() (*airflow.models.taskreschedule.TaskReschedule static method*), 680
flatten_list() (*in module airflow.contrib.hooks.qubole_hook*), 619
folder (*airflow.models.DAG attribute*), 689
folder (*airflow.models.dag.DAG attribute*), 663
following_schedule() (*airflow.models.DAG method*), 690
following_schedule() (*airflow.models.dag.DAG method*), 663
free_reserved_port() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 599
from_dict() (*airflow.executors.kubernetes_executor.KubernetesExecutorConfig static method*), 647
FSHook (*class in airflow.contrib.hooks.fs_hook*), 568
FTPHook (*class in airflow.contrib.hooks.ftp_hook*), 568
FTPSensor (*class in airflow.contrib.sensors.ftp_sensor*), 500
FTPSHook (*class in airflow.contrib.hooks.ftp_hook*), 570
FTPSSensor (*class in airflow.contrib.sensors.ftp_sensor*), 500
full_filepath (*airflow.models.DAG attribute*), 689
full_filepath (*airflow.models.dag.DAG attribute*), 662
FUNCTION_NAME_COMPILED_PATTERN (*in module airflow.contrib.operators.gcp_function_operator*), 378
FUNCTION_NAME_PATTERN (*in module airflow.contrib.operators.gcp_function_operator*), 378

G

G_APP_CRED (*in module airflow.contrib.operators.gcp_container_operator*), 375
GCE_INSTANCE_TEMPLATE_FIELDS_TO_SANITIZE (*in module airflow.contrib.operators.gcp_compute_operator*), 371
GCE_INSTANCE_TEMPLATE_VALIDATION_PATCH_SPECIFICATIO (*in module airflow.contrib.operators.gcp_compute_operator*), 371
GceBaseOperator (*class in airflow.contrib.operators.gcp_compute_operator*), 369
GceHook (*class in airflow.contrib.hooks.gcp_compute_hook*), 575
GceInstanceGroupManagerUpdateTemplateOperator (*class in airflow.contrib.operators.gcp_compute_operator*), 372
GceInstanceStartOperator (*class in airflow.contrib.operators.gcp_compute_operator*), 369
GceInstanceStopOperator (*class in airflow.contrib.operators.gcp_compute_operator*), 370
GceInstanceTemplateCopyOperator (*class in airflow.contrib.operators.gcp_compute_operator*), 371
GceOperationStatus (*class in airflow.contrib.hooks.gcp_compute_hook*), 575
GceSetMachineTypeOperator (*class in airflow.contrib.operators.gcp_compute_operator*), 370
GCF_SOURCE_ARCHIVE_URL (*in module airflow.contrib.operators.gcp_function_operator*), 377
GCF_SOURCE_UPLOAD_URL (*in module airflow.contrib.operators.gcp_function_operator*), 377
GCF_ZIP_PATH (*in module airflow.contrib.operators.gcp_function_operator*), 377
GcfFunctionDeleteOperator (*class in airflow.contrib.operators.gcp_function_operator*), 378
GcfFunctionDeployOperator (*class in airflow.contrib.operators.gcp_function_operator*), 376
GcfHook (*class in airflow.contrib.hooks.gcp_function_hook*), 581
GCP_CREDENTIALS_KEY_PATH (*in module airflow.contrib.hooks.gcp_sql_hook*), 596
GCP_CREDENTIALS_KEYFILE_DICT (*in module airflow.contrib.hooks.gcp_sql_hook*), 596
GCPSpeechToTextHook (*class in airflow.contrib.hooks.gcp_speech_to_text_hook*), 593
GcpSpeechToTextRecognizeSpeechOperator

(*class in airflow.contrib.operators.gcp_speech_to_text_operator*), 385

GCPTextToSpeechHook (*class in airflow.contrib.hooks.gcp_text_to_speech_hook*), 599

GcpTextToSpeechSynthesizeOperator (*class in airflow.contrib.operators.gcp_text_to_speech_operator*), 393

GcpTransferJobsStatus (*class in airflow.contrib.hooks.gcp_transfer_hook*), 600

GcpTransferOperationStatus (*class in airflow.contrib.hooks.gcp_transfer_hook*), 600

GCPTransferServiceHook (*class in airflow.contrib.hooks.gcp_transfer_hook*), 601

GcpTransferServiceJobCreateOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 394

GcpTransferServiceJobDeleteOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 396

GcpTransferServiceJobUpdateOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 395

GcpTransferServiceOperationCancelOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 398

GcpTransferServiceOperationGetOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 396

GcpTransferServiceOperationPauseOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 397

GcpTransferServiceOperationResumeOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 397

GcpTransferServiceOperationsListOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 396

GCPTransferServiceWaitForJobStatusSensor (*class in airflow.contrib.sensors.gcp_transfer_sensor*), 501

GcpTranslateSpeechOperator (*class in airflow.contrib.operators.gcp_translate_speech_operator*), 403

GCS_DATA_SINK (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601

GCS_DATA_SOURCE (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601

GCS_PREFIX_LENGTH (*airflow.contrib.operators.dataflow_operator.GoogleCloudBucketHelper attribute*), 345

generate_command() (*airflow.models.TaskInstance static method*), 701

generate_command() (*airflow.models.taskinstance.TaskInstance static method*), 676

generate_data_dict() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS class method*), 335

generate_schema_dict() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS class method*), 335

GenericTransfer (*class in airflow.operators.generic_transfer*), 282

GET (*airflow.contrib.operators.sftp_operator.SFTPOperation attribute*), 475

get() (*airflow.contrib.hooks.gcp_dataflow_hook._DataflowJob method*), 580

get() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJob method*), 580

get() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcOperation method*), 581

get() (*airflow.models.Variable class method*), 705

get() (*airflow.models.variable.Variable class method*), 680

get_active_runs() (*airflow.models.DAG method*), 691

get_active_runs() (*airflow.models.dag.DAG method*), 664

get_arraysize() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558

get_attachments_by_name() (*airflow.contrib.hooks.imap_hook.Mail method*), 613

get_autocommit() (*airflow.hooks.dbapi_hook.DbApiHook method*), 519

get_autocommit() (*airflow.hooks.mssql_hook.MsSqlHook method*), 528

get_autocommit() (*airflow.hooks.mysql_hook.MySqlHook method*), 529

get_available_fields() (*airflow.contrib.hooks.salesforce_hook.SalesforceHook method*), 626

get_bq_fields() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS class method*), 335

get_bq_mode() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS class method*), 335

get_bq_type() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS class method*), 335

get_bucket() (*airflow.hooks.S3_hook.S3Hook method*), 514 get_client() (*airflow.contrib.hooks.gcp_container_hook.GKEClusterHook method*), 578

get_client_type() (*airflow.contrib.hooks.aws_hook.AwsHook method*), 539

get_cluster() (*airflow.contrib.hooks.cassandra_hook.CassandraHook method*), 559

get_cluster() (*airflow.contrib.hooks.gcp_container_hook.GKEClusterHook method*), 579

get_cluster() (*airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook method*), 581

get_cluster_states_for_table() (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook static method*), 574

get_collection() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615

get_collection_link() (*in module airflow.contrib.hooks.azure_cosmos_hook*), 543

get_column_families_for_table() (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook static method*), 573

get_command() (*airflow.operators.docker_operator.DockerOperator method*), 280

get_conn() (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook method*), 535

get_conn() (*airflow.contrib.hooks.aws_dynamodb_hook.AwsDynamoDBHook method*), 537

get_conn() (*airflow.contrib.hooks.aws_firehose_hook.AwsFirehoseHook method*), 537

get_conn() (*airflow.contrib.hooks.aws_glue_catalog_hook.AwsGlueCatalogHook method*), 537

get_conn() (*airflow.contrib.hooks.aws_lambda_hook.AwsLambdaHook method*), 539

get_conn() (*airflow.contrib.hooks.aws_sns_hook.AwsSnsHook method*), 540

get_conn() (*airflow.contrib.hooks.aws_sqs_hook.SQSHook method*), 540

get_conn() (*airflow.contrib.hooks.azure_container_instance_hook.AzureContainerInstanceHook method*), 541

get_conn() (*airflow.contrib.hooks.azure_container_registry_hook.AzureContainerRegistryHook method*), 542

get_conn() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543

get_conn() (*airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook method*), 544

get_conn() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 545

get_conn() (*airflow.contrib.hooks.bigquery_hook.BigQueryHook method*), 547

get_conn() (*airflow.contrib.hooks.cassandra_hook.CassandraHook method*), 559

get_conn() (*airflow.contrib.hooks.cloudant_hook.CloudantHook method*), 560

get_conn() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 563

get_conn() (*airflow.contrib.hooks.dingding_hook.DingdingHook method*), 566

get_conn() (*airflow.contrib.hooks.emr_hook.EmrHook method*), 568

get_conn() (*airflow.contrib.hooks.fs_hook.FSHook method*), 568

get_conn() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569

get_conn() (*airflow.contrib.hooks.ftp_hook.FTPSHook method*), 570

get_conn() (*airflow.contrib.hooks.gcp_cloud_build_hook.CloudBuildHook method*), 574

get_conn() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 575

get_conn() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook method*), 580

get_conn() (*airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook method*), 581

get_conn() (*airflow.contrib.hooks.gcp_function_hook.GcfHook method*), 582

get_conn() (*airflow.contrib.hooks.gcp_kms_hook.GoogleCloudKMSHook method*), 583

get_conn() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 584

get_conn() (*airflow.contrib.hooks.gcp_natural_language_hook.CloudNatu method*), 585

get_conn() (*airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method*), 588

get_conn() (*airflow.contrib.hooks.gcp_speech_to_text_hook.GCPSpeechTo method*), 593

get_conn() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 594

get_conn() (*airflow.contrib.hooks.gcp_text_to_speech_hook.GCPTextToSp method*), 599

get_conn() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferService method*), 601

get_conn() (*airflow.contrib.hooks.gcp_translate_hook.CloudTranslateHoo method*), 604

get_conn() (*airflow.contrib.hooks.gcp_video_intelligence_hook.CloudVide method*), 605

get_conn() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606

get_conn() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 607

get_conn() (*airflow.contrib.hooks.grpc_hook.GrpcHook method*), 611

get_conn() (*airflow.contrib.hooks.jira_hook.JiraHook method*), 614

get_conn() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
get_conn() (*airflow.contrib.hooks.openfaas_hook.OpenFaasHook method*), 617
get_conn() (*airflow.contrib.hooks.opsgenie_alert_hook.OpsgenieAlertHook method*), 618
get_conn() (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook method*), 618
get_conn() (*airflow.contrib.hooks.redis_hook.RedisHook method*), 620
get_conn() (*airflow.contrib.hooks.redshift_hook.RedshiftHook method*), 620
get_conn() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 622
get_conn() (*airflow.contrib.hooks.salesforce_hook.SalesforceHook method*), 626
get_conn() (*airflow.contrib.hooks.segment_hook.SegmentHook method*), 628
get_conn() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 628
get_conn() (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook method*), 630
get_conn() (*airflow.contrib.hooks.spark_jdbc_hook.SparkJDBCHook method*), 632
get_conn() (*airflow.contrib.hooks.spark_sql_hook.SparkSqlHook method*), 634
get_conn() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
get_conn() (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 638
get_conn() (*airflow.contrib.hooks.ssh_hook.SSHHook method*), 640
get_conn() (*airflow.contrib.hooks.vertica_hook.VerticaHook method*), 641
get_conn() (*airflow.contrib.hooks.wasb_hook.WasbHook method*), 641
get_conn() (*airflow.contrib.hooks.winrm_hook.WinRMHook method*), 644
get_conn() (*airflow.hooks.base_hook.BaseHook method*), 518
get_conn() (*airflow.hooks.dbapi_hook.DbApiHook method*), 518
get_conn() (*airflow.hooks.docker_hook.DockerHook method*), 520
get_conn() (*airflow.hooks.druid_hook.DruidDbApiHook method*), 520
get_conn() (*airflow.hooks.hdfs_hook.HDFSHook method*), 521
get_conn() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 523
get_conn() (*airflow.hooks.hive_hooks.HiveServer2Hook method*), 525
get_conn() (*airflow.hooks.http_hook.HttpHook method*), 527
get_conn() (*airflow.hooks.jdbc_hook.JdbcHook method*), 528
get_conn() (*airflow.hooks.mssql_hook.MsSqlHook method*), 528
get_conn() (*airflow.hooks.mysql_hook.MySqlHook method*), 529
get_conn() (*airflow.hooks.oracle_hook.OracleHook method*), 530
get_conn() (*airflow.hooks.postgres_hook.PostgresHook method*), 531
get_conn() (*airflow.hooks.presto_hook.PrestoHook method*), 532
get_conn() (*airflow.hooks.S3_hook.S3Hook method*), 514
get_conn() (*airflow.hooks.samba_hook.SambaHook method*), 533
get_conn() (*airflow.hooks.sqlite_hook.SqliteHook method*), 533
get_conn() (*airflow.hooks.webhdfs_hook.WebHDFSHook method*), 534
get_conn() (*airflow.hooks.zendesk_hook.ZendeskHook method*), 535
get_conn_url() (*airflow.hooks.druid_hook.DruidHook method*), 520
get_connection() (*airflow.hooks.base_hook.BaseHook class method*), 517
get_connections() (*airflow.hooks.base_hook.BaseHook class method*), 517
get_context_from_env_var() (*in module airflow.hooks.hive_hooks*), 521
get_crc32c() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 609
get_credentials() (*airflow.contrib.hooks.aws_hook.AwsHook method*), 539
get_current() (*airflow.models.dag.DagModel class method*), 667
get_current() (*airflow.models.DagModel class method*), 694
get_current_resource_version() (*airflow.models.KubeResourceVersion static method*), 698
get_current_resource_version() (*airflow.models.kubernetes.KubeResourceVersion static method*), 672
get_cursor() (*airflow.hooks.dbapi_hook.DbApiHook method*), 519
get_dag() (*airflow.models.DagBag method*), 695
get_dag() (*airflow.models.dagbag.DagBag method*), 669
get_dag() (*airflow.models.dag.DagModel method*), 667

get_dag() (*airflow.models.DagModel method*), 694
get_dag() (*airflow.models.DagRun method*), 697
get_dag() (*airflow.models.dagrun.DagRun method*), 671
get_dagmodel() (*airflow.models.dag.DagModel static method*), 667
get_dagmodel() (*airflow.models.DagModel static method*), 694
get_dagrun() (*airflow.models.DAG method*), 691
get_dagrun() (*airflow.models.dag.DAG method*), 664
get_dagrun() (*airflow.models.TaskInstance method*), 703
get_dagrun() (*airflow.models.taskinstance.TaskInstance method*), 677
get_database() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 591
get_database() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 595
get_database_hook() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 599
get_database_link() (*in module airflow.contrib.hooks.azure_cosmos_hook*), 543
get_databases() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 524
get_dataset() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 556
get_datasets_list() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 556
get_db_hook() (*airflow.contrib.operators.bigquery_check_operator.BigQueryCheckOperator method*), 321
get_db_hook() (*airflow.contrib.operators.bigquery_check_operator.BigQueryIntervalCheckOperator method*), 322
get_db_hook() (*airflow.contrib.operators.bigquery_check_operator.BigQueryValueCheckOperator method*), 321
get_db_hook() (*airflow.contrib.operators.qubole_check_operator.QuboleCheckOperator method*), 460
get_db_hook() (*airflow.contrib.operators.qubole_check_operator.QuboleValueCheckOperator method*), 461
get_db_hook() (*airflow.operators.check_operator.CheckOperator method*), 276
get_db_hook() (*airflow.operators.check_operator.IntervalCheckOperator method*), 277
get_db_hook() (*airflow.operators.check_operator.ValueCheckOperator method*), 277
get_db_hook() (*airflow.operators.druid_check_operator.DruidCheckOperator method*), 281
get_db_hook() (*airflow.operators.presto_check_operator.PrestoCheckOperator method*), 294
get_db_hook() (*airflow.operators.presto_check_operator.PrestoIntervalCheckOperator method*), 295
get_db_hook() (*airflow.operators.presto_check_operator.PrestoValueCheckOperator method*), 295
get_default_executor() (*in module airflow.executors*), 653
get_default_exprs() (*airflow.operators.hive_stats_operator.HiveStatsCollectionOperator method*), 285
get_default_view() (*airflow.models.DAG method*), 690
get_default_view() (*airflow.models.dag.DAG method*), 663
get_default_view() (*airflow.models.dag.DagModel method*), 667
get_default_view() (*airflow.models.DagModel method*), 694
get_direct_relative_ids() (*airflow.models.BaseOperator method*), 686
get_direct_relative_ids() (*airflow.models.baseoperator.BaseOperator method*), 658
get_direct_relatives() (*airflow.models.BaseOperator method*), 686
get_direct_relatives() (*airflow.models.baseoperator.BaseOperator method*), 658
get_document() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
get_document_link() (*in module airflow.contrib.hooks.azure_cosmos_hook*), 543
get_documents() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
get_emr_response() (*airflow.contrib.sensors.emr_job_flow_sensor.EmrJobFlowSensor method*), 499
get_emr_response() (*airflow.contrib.sensors.emr_step_sensor.EmrStepSensor method*), 499
get_entity_with_name() (*air-*

flow.contrib.hooks.gcp_vision_hook.NameDeterminer method), 606
get_event_buffer() (*airflow.executors.base_executor.BaseExecutor method*), 645
get_event_buffer() (*airflow.executors.BaseExecutor method*), 651
get_extra() (*airflow.models.Connection method*), 687
get_extra() (*airflow.models.connection.Connection method*), 660
get_extra_links() (*airflow.contrib.hooks.qubole_hook.QuboleHook method*), 619
get_extra_links() (*airflow.models.BaseOperator method*), 686
get_extra_links() (*airflow.models.baseoperator.BaseOperator method*), 659
get_failed_dep_statuses() (*airflow.models.TaskInstance method*), 702
get_failed_dep_statuses() (*airflow.models.taskinstance.TaskInstance method*), 677
get_failed_reason_from_response() (*airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor method*), 509
get_failed_reason_from_response() (*airflow.contrib.sensors.sagemaker_endpoint_sensor.SageMakerEndpointSensor method*), 509
get_failed_reason_from_response() (*airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTrainingSensor method*), 510
get_failed_reason_from_response() (*airflow.contrib.sensors.sagemaker_transform_sensor.SageMakerTransformSensor method*), 510
get_failed_reason_from_response() (*airflow.contrib.sensors.sagemaker_tuning_sensor.SageMakerTuningSensor method*), 511
get_fernet() (*in module airflow.models.crypto*), 660
get_file() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 546
get_file() (*airflow.contrib.hooks.imap_hook.MailPart method*), 614
get_file() (*airflow.contrib.hooks.wasb_hook.WasbHook method*), 642
get_file_to_stream() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 546
get_file_volume() (*airflow.contrib.hooks.azure_container_volume_hook.AzureContainerVolumeHook method*), 542
get_first() (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook method*), 618
get_first() (*airflow.contrib.hooks.qubole_check_hook.QuboleCheckHook method*), 619
get_first() (*airflow.hooks.dbapi_hook.DbApiHook method*), 518
get_first() (*airflow.hooks.presto_hook.PrestoHook method*), 532
get_first() (*airflow.operators.druid_check_operator.DruidCheckOperato method*), 281
get_flat_relative_ids() (*airflow.models.BaseOperator method*), 686
get_flat_relative_ids() (*airflow.models.baseoperator.BaseOperator method*), 658
get_flat_relatives() (*airflow.models.BaseOperator method*), 686
get_flat_relatives() (*airflow.models.baseoperator.BaseOperator method*), 658
GET_FUNCTION (*airflow.contrib.hooks.openfaas_hook.OpenFaasHook attribute*), 617
get_function() (*airflow.contrib.hooks.gcp_function_hook.GcfHook method*), 582
get_hook() (*airflow.contrib.operators.aws_athena_operator.AWSAthenaO method*), 316
get_hook() (*airflow.contrib.operators.awsbatch_operator.AWSBatchOpera method*), 317
get_hook() (*airflow.contrib.operators.ecs_operator.ECSOperator method*), 361
get_hook() (*airflow.contrib.operators.jenkins_job_trigger_operator.Jenkin method*), 438
get_hook() (*airflow.contrib.operators.qubole_check_operator.QuboleChec method*), 460
get_hook() (*airflow.contrib.operators.qubole_check_operator.QuboleValu method*), 461
get_hook() (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464
get_hook() (*airflow.contrib.operators.snowflake_operator.SnowflakeOpera method*), 479
get_hook() (*airflow.contrib.sensors.aws_athena_sensor.AthenaSensor method*), 492
get_hook() (*airflow.contrib.sensors.aws_glue_catalog_partition_sensor.Av method*), 493
get_hook() (*airflow.hooks.base_hook.BaseHook class method*), 517
get_hook() (*airflow.models.Connection method*), 687
get_hook() (*airflow.models.connection.Connection method*), 660
get_hook() (*airflow.operators.docker_operator.DockerOperator method*), 280
get_hook() (*airflow.operators.hive_operator.HiveOperator method*), 284
get_hook() (*airflow.operators.pig_operator.PigOperator method*), 293
get_token() (*air-

flow.hooks.mysql_hook.MySqlHook method), 529
get_iam_token() (*airflow.hooks.postgres_hook.PostgresHook method*), 532
get_instance() (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook method*), 572
get_instance() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 590
get_instance() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 594
get_instance_group_manager() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 576
get_instance_template() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 576
get_jenkins_server() (*airflow.contrib.hooks.jenkins_hook.JenkinsHook method*), 614
get_jobs_id() (*airflow.contrib.hooks.qubole_hook.QuboleHook method*), 619
get_jobs_id() (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464
get_key() (*airflow.hooks.S3_hook.S3Hook method*), 515
get_last_dagrun() (*airflow.models.DAG method*), 690
get_last_dagrun() (*airflow.models.dag.DAG method*), 664
get_last_dagrun() (*airflow.models.dag.DagModel method*), 667
get_last_dagrun() (*airflow.models.DagModel method*), 694
get_last_dagrun() (*in module airflow.models.dag*), 661
get_latest_runs() (*airflow.models.DagRun class method*), 698
get_latest_runs() (*airflow.models.dagrun.DagRun class method*), 671
get_lb_policy() (*airflow.contrib.hooks.cassandra_hook.CassandraHook static method*), 559
get_link() (*airflow.contrib.operators.bigquery_operator.BigQueryConsoleLink method*), 323
get_link() (*airflow.contrib.operators.qubole_operator.QDSLink method*), 462
get_link() (*airflow.models.baseoperator.BaseOperatorLink method*), 659
get_log() (*airflow.contrib.hooks.qubole_hook.QuboleHook method*), 619
get_log() (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464
get_log_conn() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 622
get_logs() (*airflow.contrib.hooks.azure_container_instance_hook.AzureC method*), 541
get_many() (*airflow.models.XCom class method*), 706
get_many() (*airflow.models.xcom.XCom class method*), 681
get_md5hash() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 609
get_messages() (*airflow.contrib.hooks.azure_container_instance_hook.AzureContainerI method*), 541
get_metastore_client() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 523
get_mod_time() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 570
get_mod_time() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 629
get_model() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 585
get_num_active_runs() (*airflow.models.DAG method*), 691
get_num_active_runs() (*airflow.models.dag.DAG method*), 664
get_num_running_task_instances() (*airflow.models.TaskInstance method*), 704
get_num_running_task_instances() (*airflow.models.taskinstance.TaskInstance method*), 679
get_num_task_instances() (*airflow.models.DAG static method*), 693
get_num_task_instances() (*airflow.models.dag.DAG static method*), 666
get_object_from_salesforce() (*airflow.contrib.hooks.salesforce_hook.SalesforceHook method*), 626
get_one() (*airflow.models.XCom class method*), 706
get_one() (*airflow.models.xcom.XCom class method*), 681
get_operation() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 564
get_operation() (*airflow.contrib.hooks.gcp_container_hook.GKEClusterHook method*), 578

get_options_list() (*in module airflow.contrib.hooks.qubole_hook*), 619
get_or_create_current_kube_worker_uuid() (*airflow.models.kubernetes.KubeWorkerIdentifier static method*), 672
get_or_create_current_kube_worker_uuid() (*airflow.models.KubeWorkerIdentifier static method*), 698
get_pandas_df() (*airflow.contrib.hooks.bigquery_hook.BigQueryHook method*), 547
get_pandas_df() (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook method*), 618
get_pandas_df() (*airflow.hooks.base_hook.BaseHook method*), 518
get_pandas_df() (*airflow.hooks.dbapi_hook.DbApiHook method*), 518
get_pandas_df() (*airflow.hooks.druid_hook.DruidDbApiHook method*), 521
get_pandas_df() (*airflow.hooks.hive_hooks.HiveServer2Hook method*), 526
get_pandas_df() (*airflow.hooks.presto_hook.PrestoHook method*), 532
get_partitions() (*airflow.contrib.hooks.aws_glue_catalog_hook.AwsGlueCatalogHook method*), 537
get_partitions() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 524
get_password() (*airflow.models.Connection method*), 687
get_password() (*airflow.models.connection.Connection method*), 660
get_path() (*airflow.contrib.hooks.fs_hook.FSHook method*), 568
get_previous_dagrun() (*airflow.models.DagRun method*), 697
get_previous_dagrun() (*airflow.models.dagrun.DagRun method*), 671
get_previous_scheduled_dagrun() (*airflow.models.DagRun method*), 697
get_previous_scheduled_dagrun() (*airflow.models.dagrun.DagRun method*), 671
get_product() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606
get_product_set() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606
get_proxy_version() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner method*), 597
get_query_results() (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook method*), 536
get_query_results() (*airflow.contrib.hooks.qubole_check_hook.QuboleCheckHook method*), 619
get_records() (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook method*), 618
get_records() (*airflow.hooks.base_hook.BaseHook method*), 518
get_records() (*airflow.hooks.dbapi_hook.DbApiHook method*), 518
get_records() (*airflow.hooks.hive_hooks.HiveServer2Hook method*), 526
get_records() (*airflow.hooks.presto_hook.PrestoHook method*), 532
get_resource_type() (*airflow.contrib.hooks.aws_hook.AwsHook method*), 539
get_results() (*airflow.contrib.hooks.qubole_hook.QuboleHook method*), 619
get_results() (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464
get_results() (*airflow.hooks.hive_hooks.HiveServer2Hook method*), 525
get_run() (*airflow.models.DagRun static method*), 697
get_run() (*airflow.models.dagrun.DagRun static method*), 671
get_run_dates() (*airflow.models.DAG method*), 690
get_run_dates() (*airflow.models.dag.DAG method*), 663
GET_RUN_ENDPOINT (*in module airflow.contrib.hooks.databricks_hook*), 560
get_run_page_url() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
get_run_state() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
get_s3_key() (*airflow.contrib.operators.s3_to_sftp_operator.S3ToSFTPO static method*), 469
get_s3_key() (*airflow.contrib.operators.sftp_to_s3_operator.SFTPToS3O

static method), 477
get_sagemaker_response() (*airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor method*), 509
get_sagemaker_response() (*airflow.contrib.sensors.sagemaker_endpoint_sensor.SageMakerEndpointSensor method*), 509
get_sagemaker_response() (*airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTrainingSensor method*), 510
get_sagemaker_response() (*airflow.contrib.sensors.sagemaker_transform_sensor.SageMakerTransformSensor method*), 510
get_sagemaker_response() (*airflow.contrib.sensors.sagemaker_tuning_sensor.SageMakerTuningSensor method*), 511
get_schema() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 554
get_service() (*airflow.contrib.hooks.bigquery_hook.BigQueryHook method*), 547
get_session() (*airflow.contrib.hooks.aws_hook.AwsHook method*), 539
get_size() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 570
get_size() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 609
get_socket_path() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner method*), 597
get_sql_from_qbol_cmd() (*in module airflow.contrib.operators.qubole_check_operator*), 461
get_sqlalchemy_engine() (*airflow.hooks.dbapi_hook.DbApiHook method*), 518
get_sqlproxy_runner() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 599
get_state() (*airflow.models.DagRun method*), 697
get_state() (*airflow.models.dagrun.DagRun method*), 670
get_state_exitcode_details() (*airflow.contrib.hooks.azure_container_instance_hook.AzureContainerInstanceHook method*), 541
get_storagekey() (*airflow.contrib.hooks.azure_container_volume_hook.AzureContainerVolumeHook method*), 542
get_table() (*airflow.contrib.hooks.aws_glue_catalog_hook.AwsGlueCatalogHook method*), 538
get_table() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 524
get_table_location() (*airflow.contrib.hooks.aws_glue_catalog_hook.AwsGlueCatalogHook method*), 538
get_tabledata() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 554
get_tables() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 524
get_task() (*airflow.models.DAG method*), 692
get_task() (*airflow.models.dag.DAG method*), 665
get_task_instance() (*airflow.models.DagRun method*), 697
get_task_instance() (*airflow.models.dagrun.DagRun method*), 671
get_task_instances() (*airflow.models.BaseOperator method*), 686
get_task_instances() (*airflow.models.baseoperator.BaseOperator method*), 658
get_task_instances() (*airflow.models.DAG method*), 691
get_task_instances() (*airflow.models.dag.DAG method*), 664
get_task_instances() (*airflow.models.DagRun method*), 697
get_task_instances() (*airflow.models.dagrun.DagRun method*), 671
get_template_context() (*airflow.models.TaskInstance method*), 704
get_template_context() (*airflow.models.taskinstance.TaskInstance method*), 678
get_template_env() (*airflow.models.BaseOperator method*), 686
get_template_env() (*airflow.models.baseoperator.BaseOperator method*), 658
get_template_env() (*airflow.models.DAG method*), 691
get_template_env() (*airflow.models.dag.DAG method*), 664
get_time() (*in module airflow.contrib.sensors.gcs_sensor*), 503
get_transfer_job() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 602
get_transfer_operation() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 603
get_tunnel() (*airflow.contrib.hooks.ssh_hook.SSHHook method*), 640
get_uri() (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook method*), 618
get_uri() (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook method*), 630

get_uri() (*airflow.hooks.dbapi_hook.DbApiHook method*), 518
get_uri() (*airflow.hooks.druid_hook.DruidDbApiHook method*), 521
get_val() (*airflow.models.Variable method*), 705
get_val() (*airflow.models.variable.Variable method*), 680
get_wildcard_key() (*airflow.hooks.S3_hook.S3Hook method*), 515
GKEClusterCreateOperator (*class in airflow.contrib.operators.gcp_container_operator*), 374
GKEClusterDeleteOperator (*class in airflow.contrib.operators.gcp_container_operator*), 373
GKEClusterHook (*class in airflow.contrib.hooks.gcp_container_hook*), 577
GKEPodOperator (*class in airflow.contrib.operators.gcp_container_operator*), 375
global_operator_extra_link_dict() (*airflow.models.BaseOperator method*), 685
global_operator_extra_link_dict() (*airflow.models.baseoperator.BaseOperator method*), 657
google_cloud_to_local() (*airflow.contrib.operators.dataflow_operator.GoogleCloudBucketHelper method*), 345
GoogleCloudBaseHook (*class in airflow.contrib.hooks.gcp_api_base_hook*), 571
GoogleCloudBaseHook._Decorators (*class in airflow.contrib.hooks.gcp_api_base_hook*), 571
GoogleCloudBucketHelper (*class in airflow.contrib.operators.dataflow_operator*), 345
GoogleCloudKMSHook (*class in airflow.contrib.hooks.gcp_kms_hook*), 583
GoogleCloudStorageBucketCreateAclEntryOperator (*class in airflow.contrib.operators.gcs_acl_operator*), 422
GoogleCloudStorageCreateBucketOperator (*class in airflow.contrib.operators.gcs_operator*), 426
GoogleCloudStorageDeleteOperator (*class in airflow.contrib.operators.gcs_delete_operator*), 424
GoogleCloudStorageDownloadOperator (*class in airflow.contrib.operators.gcs_download_operator*), 424
GoogleCloudStorageHook (*class in airflow.contrib.hooks.gcs_hook*), 607
GoogleCloudStorageListOperator (*class in airflow.contrib.operators.gcs_list_operator*), 425
GoogleCloudStorageObjectCreateAclEntryOperator (*class in airflow.contrib.operators.gcs_acl_operator*), 423
GoogleCloudStorageObjectSensor (*class in airflow.contrib.sensors.gcs_sensor*), 501
GoogleCloudStorageObjectUpdatedSensor (*class in airflow.contrib.sensors.gcs_sensor*), 502
GoogleCloudStoragePrefixSensor (*class in airflow.contrib.sensors.gcs_sensor*), 502
GoogleCloudStorageToBigQueryOperator (*class in airflow.contrib.operators.gcs_to_bq*), 428
GoogleCloudStorageToGoogleCloudStorageOperator (*class in airflow.contrib.operators.gcs_to_gcs*), 430
GoogleCloudStorageToGoogleCloudStorageTransferOpera (*class in airflow.contrib.operators.gcp_transfer_operator*), 400
GoogleCloudStorageToS3Operator (*class in airflow.contrib.operators.gcs_to_s3*), 433
GoogleCloudStorageUploadSessionCompleteSensor (*class in airflow.contrib.sensors.gcs_sensor*), 503
GrpcHook (*class in airflow.contrib.hooks.grpc_hook*), 611
GrpcOperator (*class in airflow.contrib.operators.grpc_operator*), 434

H

handle_airflow_exception() (*in module airflow.contrib.operators.qubole_check_operator*), 461
handle_callback() (*airflow.models.DAG method*), 690
handle_callback() (*airflow.models.dag.DAG method*), 664
handle_failure() (*airflow.models.TaskInstance method*), 703
handle_failure() (*airflow.models.taskinstance.TaskInstance method*), 678
handle_failure_retry() (*airflow.contrib.hooks.qubole_check_hook.QuboleCheckHook static method*), 619
handle_failure_retry() (*airflow.contrib.hooks.qubole_hook.QuboleHook static method*), 619
has_attachments() (*airflow.contrib.hooks.imap_hook.Mail method*), 613
has_dag() (*airflow.models.BaseOperator method*), 685
has_dag() (*airflow.models.baseoperator.BaseOperator method*), 657
has_equal_name() (*airflow.contrib.hooks.imap_hook.MailPart method*), 614

has_mail_attachment() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 612
has_matching_name() (*airflow.contrib.hooks.imap_hook.MailPart method*), 613
has_task() (*airflow.executors.base_executor.BaseExecutor method*), 644
has_task() (*airflow.executors.BaseExecutor method*), 651
has_task() (*airflow.models.DAG method*), 692
has_task() (*airflow.models.dag.DAG method*), 665
HDFSHook (*class in airflow.hooks.hdfs_hook*), 521
HDFSHookException, 521
HdfsSensor (*class in airflow.sensors.hdfs_sensor*), 307
HdfsSensorFolder (*class in airflow.contrib.sensors.hdfs_sensor*), 504
HdfsSensorRegex (*class in airflow.contrib.sensors.hdfs_sensor*), 504
heartbeat() (*airflow.executors.base_executor.BaseExecutor method*), 644
heartbeat() (*airflow.executors.BaseExecutor method*), 651
HipChatAPIOperator (*class in airflow.contrib.operators.hipchat_operator*), 435
HipChatAPISendRoomNotificationOperator (*class in airflow.contrib.operators.hipchat_operator*), 435
Hive2SambaOperator (*class in airflow.operators.hive_to_samba_operator*), 287
HIVE_QUEUE_PRIORITIES (*in module airflow.hooks.hive_hooks*), 521
HiveCliHook (*class in airflow.hooks.hive_hooks*), 521
HiveMetastoreHook (*class in airflow.hooks.hive_hooks*), 523
HiveOperator (*class in airflow.operators.hive_operator*), 283
HivePartitionSensor (*class in airflow.sensors.hive_partition_sensor*), 308
HiveServer2Hook (*class in airflow.hooks.hive_hooks*), 525
HiveStatsCollectionOperator (*class in airflow.operators.hive_stats_operator*), 284
HiveToDruidTransfer (*class in airflow.operators.hive_to_druid*), 285
HiveToDynamoDBTransferOperator (*class in airflow.contrib.operators.hive_to_dynamodb*), 436
HiveToMySqlTransfer (*class in airflow.operators.hive_to_mysql*), 286
host (*airflow.models.Connection attribute*), 687
host (*airflow.models.connection.Connection attribute*), 659
hostname (*airflow.models.TaskInstance attribute*), 700
hostname (*airflow.models.taskinstance.TaskInstance attribute*), 675
HOURS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
HTTP_DATA_SOURCE (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
HttpHook (*class in airflow.hooks.http_hook*), 527
HttpSensor (*class in airflow.sensors.http_sensor*), 308

I

id (*airflow.models.Connection attribute*), 687
id (*airflow.models.connection.Connection attribute*), 659
id (*airflow.models.DagPickle attribute*), 696
id (*airflow.models.dagpickle.DagPickle attribute*), 669
id (*airflow.models.DagRun attribute*), 696
id (*airflow.models.dagrun.DagRun attribute*), 670
id (*airflow.models.errors.ImportError attribute*), 671
id (*airflow.models.Log attribute*), 698
id (*airflow.models.log.Log attribute*), 672
id (*airflow.models.Pool attribute*), 698
id (*airflow.models.pool.Pool attribute*), 673
id (*airflow.models.TaskFail attribute*), 699
id (*airflow.models.taskfail.TaskFail attribute*), 674
id (*airflow.models.TaskReschedule attribute*), 704
id (*airflow.models.taskreschedule.TaskReschedule attribute*), 679
id (*airflow.models.Variable attribute*), 705
id (*airflow.models.variable.Variable attribute*), 680
id (*airflow.models.XCom attribute*), 706
id (*airflow.models.xcom.XCom attribute*), 681
id_for_date() (*airflow.models.DagRun class method*), 697
id_for_date() (*airflow.models.dagrun.DagRun class method*), 670
ID_FORMAT_PREFIX (*airflow.models.DagRun attribute*), 696
ID_FORMAT_PREFIX (*airflow.models.dagrun.DagRun attribute*), 670
ID_LEN (*in module airflow.models*), 681
ID_LEN (*in module airflow.models.base*), 653
ID_PREFIX (*airflow.models.DagRun attribute*), 696
ID_PREFIX (*airflow.models.dagrun.DagRun attribute*), 670
ImapAttachmentSensor (*class in airflow.contrib.sensors.imap_attachment_sensor*), 504
ImapAttachmentToS3Operator (*class in airflow.contrib.operators.imap_attachment_to_s3_operator*), 437
ImapHook (*class in airflow.contrib.hooks.imap_hook*), 612
import_from_storage_bucket() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 565

import_instance() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method*), 596
import_query() (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 638
import_table() (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 638
ImportError (*class in airflow.models.errors*), 671
IN_PROGRESS (*airflow.contrib.hooks.gcp_transfer_hook.GcpTransferOperationStatus attribute*), 600
init_log_resource() (*airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTrainingSensor method*), 510
init_on_load() (*airflow.models.TaskInstance method*), 701
init_on_load() (*airflow.models.taskinstance.TaskInstance method*), 676
init_on_load() (*airflow.models.XCom method*), 706
init_on_load() (*airflow.models.xcom.XCom method*), 681
init_run_context() (*airflow.models.TaskInstance method*), 704
init_run_context() (*airflow.models.taskinstance.TaskInstance method*), 679
insert_all() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 556
insert_bucket_acl() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 610
insert_documents() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
insert_instance_template() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 576
insert_many() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
insert_object_acl() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 610
insert_one() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
insert_rows() (*airflow.contrib.hooks.bigquery_hook.BigQueryHook method*), 547
insert_rows() (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook method*), 618
insert_rows() (*airflow.hooks.dbapi_hook.DbApiHook method*), 519
insert_rows() (*airflow.hooks.druid_hook.DruidDbApiHook method*), 521
insert_rows() (*airflow.hooks.oracle_hook.OracleHook method*), 530
insert_rows() (*airflow.hooks.presto_hook.PrestoHook method*), 533
integer_fields (*airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOp attribute*), 470
integer_fields (*airflow.contrib.operators.sagemaker_endpoint_config_operator.SageMa attribute*), 470
integer_fields (*airflow.contrib.operators.sagemaker_training_operator.SageMakerTra attribute*), 473
integer_fields (*airflow.contrib.operators.sagemaker_tuning_operator.SageMakerTunin attribute*), 474
INTERMEDIATE_STATES (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook attribute*), 535
INTERMEDIATE_STATES (*airflow.contrib.sensors.aws_athena_sensor.AthenaSensor attribute*), 492
IntervalCheckOperator (*class in airflow.operators.check_operator*), 277
InvalidFernetToken, 660
INVOKE_ASYNC_FUNCTION (*airflow.contrib.hooks.openfaas_hook.OpenFaasHook attribute*), 617
invoke_async_function() (*airflow.contrib.hooks.openfaas_hook.OpenFaasHook method*), 617
invoke_lambda() (*airflow.contrib.hooks.aws_lambda_hook.AwsLambdaHook method*), 539
is_active (*airflow.models.dag.DagModel attribute*), 667
is_active (*airflow.models.DagModel attribute*), 694
is_array_type() (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudS class method*), 335
is_attachment() (*airflow.contrib.hooks.imap_hook.MailPart method*), 613
is_backfill (*airflow.models.DagRun attribute*), 696
is_backfill (*airflow.models.dagrun.DagRun attribute*), 670
is_bucket_updated() (*air-*

flow.contrib.sensors.gcs_sensor.GoogleCloudStorageUploadSessionCompleteSensor method), 504
`is_eligible_to_retry()` (*airflow.models.TaskInstance method*), 703
`is_eligible_to_retry()` (*airflow.models.taskinstance.TaskInstance method*), 678
`is_encrypted` (*airflow.models.Connection attribute*), 687
`is_encrypted` (*airflow.models.connection.Connection attribute*), 660
`is_encrypted` (*airflow.models.crypto.NullFernet attribute*), 660
`is_encrypted` (*airflow.models.Variable attribute*), 705
`is_encrypted` (*airflow.models.variable.Variable attribute*), 680
`is_extra_encrypted` (*airflow.models.Connection attribute*), 687
`is_extra_encrypted` (*airflow.models.connection.Connection attribute*), 660
`is_fixed_time_schedule()` (*airflow.models.DAG method*), 690
`is_fixed_time_schedule()` (*airflow.models.dag.DAG method*), 663
`is_paused` (*airflow.models.DAG attribute*), 689
`is_paused` (*airflow.models.dag.DAG attribute*), 663
`is_paused` (*airflow.models.dag.DagModel attribute*), 667
`is_paused` (*airflow.models.DagModel attribute*), 694
`is_paused_at_creation` (*airflow.models.dag.DagModel attribute*), 667
`is_paused_at_creation` (*airflow.models.DagModel attribute*), 694
`is_premature` (*airflow.models.TaskInstance attribute*), 701
`is_premature` (*airflow.models.taskinstance.TaskInstance attribute*), 675
`is_record_type()` (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator class method*), 335
`is_simple_type()` (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator class method*), 335
`is_subdag` (*airflow.models.dag.DagModel attribute*), 667
`is_subdag` (*airflow.models.DagModel attribute*), 694
`is_successful` (*airflow.contrib.hooks.databricks_hook.RunState attribute*), 562
`is_terminal` (*airflow.contrib.hooks.databricks_hook.RunState attribute*), 562
`is_updated_after()` (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 609
`isbool()` (*in module airflow.contrib.hooks.qubole_check_hook*), 618
`isfloat()` (*in module airflow.contrib.hooks.qubole_check_hook*), 618
`isint()` (*in module airflow.contrib.hooks.qubole_check_hook*), 618
`issue_field_checker()` (*airflow.contrib.sensors.jira_sensor.JiraTicketSensor method*), 506

J

`JdbcHook` (*class in airflow.hooks.jdbc_hook*), 528
`JdbcOperator` (*class in airflow.operators.jdbc_operator*), 288
`jenkins_request_with_headers()` (*in module airflow.contrib.operators.jenkins_job_trigger_operator*), 437
`JenkinsHook` (*class in airflow.contrib.hooks.jenkins_hook*), 614
`JenkinsJobTriggerOperator` (*class in airflow.contrib.operators.jenkins_job_trigger_operator*), 437
`JiraHook` (*class in airflow.contrib.hooks.jira_hook*), 614
`JiraOperator` (*class in airflow.contrib.operators.jira_operator*), 439
`JiraSensor` (*class in airflow.contrib.sensors.jira_sensor*), 505
`JiraTicketSensor` (*class in airflow.contrib.sensors.jira_sensor*), 505
`JOB_COMPLETE` (*airflow.contrib.hooks.sagemaker_hook.LogState attribute*), 621
`job_id` (*airflow.models.TaskInstance attribute*), 700
`job_id` (*airflow.models.taskinstance.TaskInstance attribute*), 675
`job_type` (*airflow.contrib.operators.dataproc_operator.DataProcHadoopOp attribute*), 354
`job_type` (*airflow.contrib.operators.dataproc_operator.DataProcHiveOpera attribute*), 353
`job_type` (*airflow.contrib.operators.dataproc_operator.DataProcJobBaseOp attribute*), 351
`job_type` (*airflow.contrib.operators.dataproc_operator.DataProcPigOperat attribute*), 352
`job_type` (*airflow.contrib.operators.dataproc_operator.DataProcPySparkO attribute*), 355
`job_type` (*airflow.contrib.operators.dataproc_operator.DataProcSparkOper attribute*), 354
`job_type` (*airflow.contrib.operators.dataproc_operator.DataProcSparkSqlO attribute*), 353

K

`key` (*airflow.models.TaskInstance attribute*), 701

key (*airflow.models.taskinstance.TaskInstance attribute*), 675
key (*airflow.models.Variable attribute*), 705
key (*airflow.models.variable.Variable attribute*), 680
key (*airflow.models.XCom attribute*), 706
key (*airflow.models.xcom.XCom attribute*), 681
kill() (*airflow.contrib.hooks.qubole_hook.QuboleHook method*), 619
kill() (*airflow.contrib.hooks.spark_sql_hook.SparkSqlHook method*), 634
kill() (*airflow.hooks.hive_hooks.HiveCliHook method*), 523
kill() (*airflow.hooks.pig_hook.PigCliHook method*), 531
kill_zombies() (*airflow.models.DagBag method*), 695
kill_zombies() (*airflow.models.dagbag.DagBag method*), 669
KUBE_CONFIG_ENV_VAR (*in module airflow.contrib.operators.gcp_container_operator*), 375
KubeConfig (*class in airflow.executors.kubernetes_executor*), 647
KubeResourceVersion (*class in airflow.models*), 698
KubeResourceVersion (*class in airflow.models.kubernetes*), 672
kubernetes_section (*airflow.executors.kubernetes_executor.KubeConfig attribute*), 647
KubernetesExecutor (*airflow.executors.Executors attribute*), 653
KubernetesExecutor (*class in airflow.executors.kubernetes_executor*), 649
KubernetesExecutorConfig (*class in airflow.executors.kubernetes_executor*), 647
KubernetesJobWatcher (*class in airflow.executors.kubernetes_executor*), 647
KubernetesPodOperator (*class in airflow.contrib.operators.kubernetes_pod_operator*), 441
KubeWorkerIdentifier (*class in airflow.models*), 698
KubeWorkerIdentifier (*class in airflow.models.kubernetes*), 672

L

label_detection() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
last_expired (*airflow.models.dag.DagModel attribute*), 667
last_expired (*airflow.models.DagModel attribute*), 694
last_pickled (*airflow.models.dag.DagModel attribute*), 667
last_pickled (*airflow.models.DagModel attribute*), 694
last_scheduler_run (*airflow.models.dag.DagModel attribute*), 667
last_scheduler_run (*airflow.models.DagModel attribute*), 694
latest_execution_date (*airflow.models.DAG attribute*), 690
latest_execution_date (*airflow.models.dag.DAG attribute*), 663
LatestOnlyOperator (*class in airflow.operators.latest_only_operator*), 288
list() (*airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook method*), 545
list() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 609
list_directories_and_files() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 545
list_directory() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569
list_directory() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 628
list_keys() (*airflow.hooks.S3_hook.S3Hook method*), 514
list_prefixes() (*airflow.hooks.S3_hook.S3Hook method*), 514
list_transfer_job() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 602
list_transfer_operations() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 603
LIST_URL (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
list_versions() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 585
load_bytes() (*airflow.hooks.S3_hook.S3Hook method*), 516
LOAD_CHECK_INTERVAL (*in module airflow.operators.hive_to_druid*), 285
load_df() (*airflow.hooks.hive_hooks.HiveCliHook method*), 522
load_file() (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShare method*), 546
load_file() (*airflow.contrib.hooks.wasb_hook.WasbHook method*), 641
load_file() (*airflow.hooks.hive_hooks.HiveCliHook method*), 523

`load_file()` (*airflow.hooks.S3_hook.S3Hook method*), 516
`load_file()` (*airflow.hooks.webhdfs_hook.WebHDFSHook method*), 534
`load_file_obj()` (*airflow.hooks.S3_hook.S3Hook method*), 516
`load_stream()` (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 547
`load_string()` (*airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook method*), 546
`load_string()` (*airflow.contrib.hooks.wasb_hook.WasbHook method*), 642
`load_string()` (*airflow.hooks.S3_hook.S3Hook method*), 516
`LocalExecutor` (*airflow.executors.Executors attribute*), 653
`LocalExecutor` (*class in airflow.executors*), 652
`LocalExecutor` (*class in airflow.executors.local_executor*), 650
`LocalExecutor._LimitedParallelism` (*class in airflow.executors*), 652
`LocalExecutor._LimitedParallelism` (*class in airflow.executors.local_executor*), 650
`LocalExecutor._UnlimitedParallelism` (*class in airflow.executors*), 652
`LocalExecutor._UnlimitedParallelism` (*class in airflow.executors.local_executor*), 650
`LocalWorker` (*class in airflow.executors.local_executor*), 649
`log` (*airflow.executors.LoggingMixin attribute*), 651
`Log` (*class in airflow.models*), 698
`Log` (*class in airflow.models.log*), 672
`log` (*in module airflow.contrib.operators.mlengine_operator*), 443
`log` (*in module airflow.hooks.webhdfs_hook*), 534
`log_filepath` (*airflow.models.TaskInstance attribute*), 701
`log_filepath` (*airflow.models.taskinstance.TaskInstance attribute*), 675
`log_stream()` (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 622
`log_url` (*airflow.models.TaskInstance attribute*), 701
`log_url` (*airflow.models.taskinstance.TaskInstance attribute*), 675
`logger` (*airflow.executors.LoggingMixin attribute*), 651
`LoggingMixin` (*class in airflow.executors*), 651
`login` (*airflow.models.Connection attribute*), 687
`login` (*airflow.models.connection.Connection attribute*), 660
`LogState` (*class in airflow.contrib.hooks.sagemaker_hook*), 621
`lookup()` (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 564

M

`Mail` (*class in airflow.contrib.hooks.imap_hook*), 613
`MailPart` (*class in airflow.contrib.hooks.imap_hook*), 613
`make_query()` (*airflow.contrib.hooks.salesforce_hook.SalesforceHook method*), 626
`mark_success_url` (*airflow.models.TaskInstance attribute*), 701
`mark_success_url` (*airflow.models.taskinstance.TaskInstance attribute*), 675
`MAX_PART_COUNT` (*airflow.hooks.hive_hooks.HiveMetastoreHook attribute*), 523
`max_partition()` (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 525
`max_partition()` (*in module airflow.macros.hive*), 274
`max_tries` (*airflow.models.TaskInstance attribute*), 700
`max_tries` (*airflow.models.taskinstance.TaskInstance attribute*), 675
`MAX_XCOM_SIZE` (*in module airflow.models.xcom*), 681
`METADATA` (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
`metadata` (*in module airflow.models.base*), 653
`MetastorePartitionSensor` (*class in airflow.sensors.metastore_partition_sensor*), 309
`MINUTES` (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
`MLEngineBatchPredictionOperator` (*class in airflow.contrib.operators.mlengine_operator*), 443
`MLEngineHook` (*class in airflow.contrib.hooks.gcp_mlengine_hook*), 584
`MLEngineModelOperator` (*class in airflow.contrib.operators.mlengine_operator*), 444
`MLEngineTrainingOperator` (*class in airflow.contrib.operators.mlengine_operator*), 446
`MLEngineVersionOperator` (*class in airflow.contrib.operators.mlengine_operator*), 445
`mlsd()` (*in module airflow.contrib.hooks.ftp_hook*), 568
`MongoHook` (*class in airflow.contrib.hooks.mongo_hook*), 614
`MongoSensor` (*class in airflow.contrib.sensors.mongo_sensor*), 506
`MongoToS3Operator` (*class in airflow.contrib.operators.mongo_to_s3*), 447

MONTH (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
MsSqlHook (*class in airflow.hooks.mssql_hook*), 528
MsSqlOperator (*class in airflow.operators.mssql_operator*), 289
MsSqlToGoogleCloudStorageOperator (*class in airflow.contrib.operators.mssql_to_gcs*), 448
MsSqlToHiveTransfer (*class in airflow.operators.mssql_to_hive*), 289
multi_stream_iter() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 623
MySqlHook (*class in airflow.hooks.mysql_hook*), 529
MySqlOperator (*class in airflow.operators.mysql_operator*), 290
MySqlToGoogleCloudStorageOperator (*class in airflow.contrib.operators.mysql_to_gcs*), 450
MySqlToHiveTransfer (*class in airflow.operators.mysql_to_hive*), 291

N

name (*airflow.contrib.operators.bigquery_operator.BigQueryConsoleLink attribute*), 323
name (*airflow.contrib.operators.qubole_operator.QDSLink attribute*), 462
name (*airflow.models.baseoperator.BaseOperatorLink attribute*), 659
NAME (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
NameDeterminer (*class in airflow.contrib.hooks.gcp_vision_hook*), 606
NamedHivePartitionSensor (*class in airflow.sensors.named_hive_partition_sensor*), 310
NEGATIVE_STATUSES (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
next() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558
next_retry_datetime() (*airflow.models.TaskInstance method*), 702
next_retry_datetime() (*airflow.models.taskinstance.TaskInstance method*), 677
next_try_number (*airflow.models.TaskInstance attribute*), 701
next_try_number (*airflow.models.taskinstance.TaskInstance attribute*), 675
non_terminal_states (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook attribute*), 621
NON_TERMINAL_STATES (*airflow.contrib.sensors.emr_job_flow_sensor.EmrJobFlowSensor attribute*), 499
NON_TERMINAL_STATES (*airflow.contrib.sensors.emr_step_sensor.EmrStepSensor attribute*), 499
non_terminal_states() (*airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor method*), 509
non_terminal_states() (*airflow.contrib.sensors.sagemaker_endpoint_sensor.SageMakerEndpoi method*), 509
non_terminal_states() (*airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTraining method*), 510
non_terminal_states() (*airflow.contrib.sensors.sagemaker_transform_sensor.SageMakerTrans method*), 510
non_terminal_states() (*airflow.contrib.sensors.sagemaker_tuning_sensor.SageMakerTuningSen method*), 511
normalize_schedule() (*airflow.models.DAG method*), 690
normalize_schedule() (*airflow.models.dag.DAG method*), 664
NoteBook (*class in airflow.operators.papermill_operator*), 292
notification_sent (*airflow.models.SlaMiss attribute*), 699
notification_sent (*airflow.models.slamiss.SlaMiss attribute*), 674
NullFernet (*class in airflow.models.crypto*), 660

O

OBJECT_CONDITIONS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
OK_STATUS_CODE (*in module airflow.contrib.hooks.openfaas_hook*), 617
on_error() (*airflow.contrib.hooks.segment_hook.SegmentHook method*), 628
on_kill() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 637
on_kill() (*airflow.contrib.operators.aws_athena_operator.AWSAthenaOpe method*), 316
on_kill() (*airflow.contrib.operators.awsbatch_operator.AWSBatchOperato method*), 317
on_kill() (*airflow.contrib.operators.bigquery_operator.BigQueryOperator method*), 325
on_kill() (*airflow.contrib.operators.databricks_operator.DatabricksRunN method*), 341
on_kill() (*airflow.contrib.operators.databricks_operator.DatabricksSubm method*), 338
on_kill() (*airflow.contrib.operators.dataproc_operator.DataProcJobBase method*), 352
on_kill() (*airflow.contrib.operators.ecs_operator.ECSOperator method*), 361

on_kill() (*airflow.contrib.operators.qubole_operator.QuboleOperator method*), 464
on_kill() (*airflow.contrib.operators.spark_jdbc_operator.SparkJDBCOperator method*), 483
on_kill() (*airflow.contrib.operators.spark_sql_operator.SparkSqlOperator method*), 484
on_kill() (*airflow.contrib.operators.spark_submit_operator.SparkSubmitOperator method*), 486
on_kill() (*airflow.contrib.operators.sqoop_operator.SqoopOperator method*), 488
on_kill() (*airflow.models.BaseOperator method*), 685
on_kill() (*airflow.models.baseoperator.BaseOperator method*), 658
on_kill() (*airflow.operators.bash_operator.BashOperator method*), 275
on_kill() (*airflow.operators.docker_operator.DockerOperator method*), 280
on_kill() (*airflow.operators.hive_operator.HiveOperator method*), 284
on_kill() (*airflow.operators.pig_operator.PigOperator method*), 293
one_row_id (*airflow.models.KubeResourceVersion attribute*), 698
one_row_id (*airflow.models.kubernetes.KubeResourceVersion attribute*), 672
one_row_id (*airflow.models.kubernetes.KubeWorkerIdentifier attribute*), 672
one_row_id (*airflow.models.KubeWorkerIdentifier attribute*), 698
open_slots() (*airflow.models.Pool method*), 699
open_slots() (*airflow.models.pool.Pool method*), 673
OpenFaasHook (*class in airflow.contrib.hooks.openfaas_hook*), 617
OPERATIONAL_POLL_INTERVAL (*in module airflow.contrib.hooks.gcp_container_hook*), 577
OPERATIONS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
operations_contain_expected_statuses() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook static method*), 603
operator (*airflow.models.TaskInstance attribute*), 700
operator (*airflow.models.taskinstance.TaskInstance attribute*), 675
operator_extra_link_dict() (*airflow.models.BaseOperator method*), 685
operator_extra_link_dict() (*airflow.models.baseoperator.BaseOperator method*), 657
operator_extra_links (*airflow.contrib.operators.bigquery_operator.BigQueryOperator attribute*), 325
operator_extra_links (*airflow.contrib.operators.qubole_operator.QuboleOperator attribute*), 464
operator_extra_links (*airflow.models.BaseOperator attribute*), 684
operator_extra_links (*airflow.models.baseoperator.BaseOperator attribute*), 657
OpsgenieAlertHook (*class in airflow.contrib.hooks.opsgenie_alert_hook*), 617
OpsgenieAlertOperator (*class in airflow.contrib.operators.opsgenie_alert_operator*), 452
OracleHook (*class in airflow.hooks.oracle_hook*), 530
OracleOperator (*class in airflow.operators.oracle_operator*), 292
OracleToAzureDataLakeTransfer (*class in airflow.contrib.operators.oracle_to_azure_data_lake_transfer*), 453
OracleToOracleTransfer (*class in airflow.contrib.operators.oracle_to_oracle_transfer*), 454
overwrite_params_with_dag_run_conf() (*airflow.models.TaskInstance method*), 704
overwrite_params_with_dag_run_conf() (*airflow.models.taskinstance.TaskInstance method*), 678
owner (*airflow.models.DAG attribute*), 689
owner (*airflow.models.dag.DAG attribute*), 663
owner (*airflow.models.Log attribute*), 698
owner (*airflow.models.log.Log attribute*), 672
owners (*airflow.models.dag.DagModel attribute*), 667
owners (*airflow.models.DagModel attribute*), 694

P

PapermillOperator (*class in airflow.operators.papermill_operator*), 292
PARALLELISM (*in module airflow.executors.base_executor*), 644
parse_config_integers() (*airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOp method*), 470
parse_first_row() (*in module airflow.contrib.hooks.qubole_check_hook*), 619
parse_from_uri() (*airflow.models.Connection method*), 687
parse_from_uri() (*airflow.models.connection.Connection method*), 660
parse_integer() (*airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOp method*), 470
parse_netloc_to_hostname() (*in module airflow.models.connection*), 659
parse_partition_name() (*airflow.sensors.named_hive_partition_sensor.NamedHivePartitionSens static method*), 310

parse_s3_url() (airflow.hooks.S3_hook.S3Hook static method), 514
parser (in module airflow.contrib.hooks.spark_jdbc_script), 633
password (airflow.models.Connection attribute), 687
password (airflow.models.connection.Connection attribute), 660
patch_database() (airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method), 595
patch_instance() (airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook method), 594
patch_instance_group_manager() (airflow.contrib.hooks.gcp_compute_hook.GceHook method), 577
patch_table() (airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method), 550
pause_transfer_operation() (airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method), 603
PAUSED (airflow.contrib.hooks.gcp_transfer_hook.GcpTransferOperationStatus attribute), 600
PENDING (airflow.contrib.hooks.gcp_compute_hook.GceOperationStatus attribute), 575
PENDING (airflow.contrib.hooks.gcp_sql_hook.CloudSqlOperationStatus attribute), 594
pickle (airflow.models.DagPickle attribute), 696
pickle (airflow.models.dagpickle.DagPickle attribute), 669
pickle() (airflow.models.DAG method), 692
pickle() (airflow.models.dag.DAG method), 665
pickle_hash (airflow.models.DagPickle attribute), 696
pickle_hash (airflow.models.dagpickle.DagPickle attribute), 669
pickle_id (airflow.models.DAG attribute), 689
pickle_id (airflow.models.dag.DAG attribute), 662
pickle_id (airflow.models.dag.DagModel attribute), 667
pickle_id (airflow.models.DagModel attribute), 694
pickle_info() (airflow.models.DAG method), 692
pickle_info() (airflow.models.dag.DAG method), 665
pid (airflow.models.TaskInstance attribute), 700
pid (airflow.models.taskinstance.TaskInstance attribute), 675
PigCliHook (class in airflow.hooks.pig_hook), 531
PigOperator (class in airflow.operators.pig_operator), 293
PinotDbApiHook (class in airflow.contrib.hooks.pinot_hook), 618
poke() (airflow.contrib.operators.gcp_bigtable_operator.BigtableTableWaitForReplicationSensor method), 367
poke() (airflow.contrib.sensors.aws_athena_sensor.AthenaSensor method), 492
poke() (airflow.contrib.sensors.aws_glue_catalog_partition_sensor.AwsGlue method), 493
poke() (airflow.contrib.sensors.aws_redshift_cluster_sensor.AwsRedshiftClu method), 494
poke() (airflow.contrib.sensors.aws_sqs_sensor.SQSSensor method), 494
poke() (airflow.contrib.sensors.azure_cosmos_sensor.AzureCosmosDocumen method), 495
poke() (airflow.contrib.sensors.bash_sensor.BashSensor method), 495
poke() (airflow.contrib.sensors.bigquery_sensor.BigQueryTableSensor method), 496
poke() (airflow.contrib.sensors.cassandra_record_sensor.CassandraRecordS method), 497
poke() (airflow.contrib.sensors.cassandra_table_sensor.CassandraTableSen method), 497
poke() (airflow.contrib.sensors.celery_queue_sensor.CeleryQueueSensor method), 498
poke() (airflow.contrib.sensors.datadog_sensor.DatadogSensor method), 498
poke() (airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor method), 498
poke() (airflow.contrib.sensors.file_sensor.FileSensor method), 500
poke() (airflow.contrib.sensors.ftp_sensor.FTPSensor method), 500
poke() (airflow.contrib.sensors.gcp_transfer_sensor.GCPTransferServiceWa method), 501
poke() (airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectSensor method), 502
poke() (airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectUpdate method), 502
poke() (airflow.contrib.sensors.gcs_sensor.GoogleCloudStoragePrefixSensor method), 503
poke() (airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageUploadSessio method), 504
poke() (airflow.contrib.sensors.hdfs_sensor.HdfsSensorFolder method), 504
poke() (airflow.contrib.sensors.hdfs_sensor.HdfsSensorRegex method), 504
poke() (airflow.contrib.sensors.imap_attachment_sensor.ImapAttachmentSe method), 505
poke() (airflow.contrib.sensors.jira_sensor.JiraSensor method), 505
poke() (airflow.contrib.sensors.jira_sensor.JiraTicketSensor method), 506
poke() (airflow.contrib.sensors.mongo_sensor.MongoSensor method), 506
poke() (airflow.contrib.sensors.pubsub_sensor.PubSubPullSensor method), 507
poke() (airflow.contrib.sensors.python_sensor.PythonSensor method), 507

poke() (*airflow.contrib.sensors.qubole_sensor.QuboleSensor method*), 507
poke() (*airflow.contrib.sensors.redis_key_sensor.RedisKeySensor method*), 508
poke() (*airflow.contrib.sensors.redis_pub_sub_sensor.RedisPubSubSensor method*), 509
poke() (*airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor method*), 509
poke() (*airflow.contrib.sensors.sftp_sensor.SFTPSensor method*), 511
poke() (*airflow.contrib.sensors.wasb_sensor.WasbBlobSensor method*), 512
poke() (*airflow.contrib.sensors.wasb_sensor.WasbPrefixSensor method*), 512
poke() (*airflow.contrib.sensors.weekday_sensor.DayOfWeekSensor method*), 513
poke() (*airflow.sensors.base_sensor_operator.BaseSensorOperator method*), 306
poke() (*airflow.sensors.external_task_sensor.ExternalTaskSensor method*), 307
poke() (*airflow.sensors.hdfs_sensor.HdfsSensor method*), 307
poke() (*airflow.sensors.hive_partition_sensor.HivePartitionSensor method*), 308
poke() (*airflow.sensors.http_sensor.HttpSensor method*), 309
poke() (*airflow.sensors.metastore_partition_sensor.MetastorePartitionSensor method*), 310
poke() (*airflow.sensors.named_hive_partition_sensor.NamedHivePartitionSensor method*), 310
poke() (*airflow.sensors.s3_key_sensor.S3KeySensor method*), 311
poke() (*airflow.sensors.s3_prefix_sensor.S3PrefixSensor method*), 312
poke() (*airflow.sensors.sql_sensor.SqlSensor method*), 312
poke() (*airflow.sensors.time_delta_sensor.TimeDeltaSensor method*), 312
poke() (*airflow.sensors.time_sensor.TimeSensor method*), 313
poke() (*airflow.sensors.web_hdfs_sensor.WebHdfsSensor method*), 313
poke_partition() (*airflow.sensors.named_hive_partition_sensor.NamedHivePartitionSensor method*), 310
poll_job_complete() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 554
poll_job_in_queue() (*airflow.contrib.operators.jenkins_job_trigger_operator.JenkinsJobTriggerOperator method*), 438
poll_operation_until_done() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 565
poll_query_status() (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook method*), 536
pool (*airflow.models.Pool attribute*), 698
pool (*airflow.models.pool.Pool attribute*), 673
pool (*airflow.models.TaskInstance attribute*), 700
pool (*airflow.models.taskinstance.TaskInstance attribute*), 675
Pool (*class in airflow.models*), 698
Pool (*class in airflow.models.pool*), 673
pool_full() (*airflow.models.TaskInstance method*), 703
pool_full() (*airflow.models.taskinstance.TaskInstance method*), 677
Popen() (*airflow.contrib.hooks.sqoop_hook.SqoopHook method*), 638
port (*airflow.models.Connection attribute*), 687
port (*airflow.models.connection.Connection attribute*), 660
Position (*in module airflow.contrib.hooks.sagemaker_hook*), 621
POSITIONAL_ARGS (*in module airflow.contrib.hooks.qubole_hook*), 619
post_event() (*airflow.contrib.hooks.datadog_hook.DatadogHook method*), 563
post_execute() (*airflow.models.BaseOperator method*), 685
post_execute() (*airflow.models.baseoperator.BaseOperator method*), 658
PostgresHook (*class in airflow.hooks.postgres_hook*), 531
PostgresOperator (*class in airflow.operators.postgres_operator*), 293
PostgresToGoogleCloudStorageOperator (*class in airflow.contrib.operators.postgres_to_gcs_operator*), 455
pre_execute() (*airflow.models.BaseOperator method*), 685
pre_execute() (*airflow.models.baseoperator.BaseOperator method*), 657
prepare_additional_parameters() (*in module airflow.contrib.operators.gcp_vision_operator*), 422
prepare_request() (*airflow.contrib.operators.hipchat_operator.HipChatAPIOperator method*), 435
prepare_request() (*airflow.contrib.operators.hipchat_operator.HipChatAPISendRoomNotif method*), 436
prepare_template() (*airflow.models.BaseOperator method*), 686
prepare_template() (*air-*

flow.models.baseoperator.BaseOperator method), 658
prepare_template() (*airflow.operators.hive_operator.HiveOperator method)*, 284
prepare_template() (*airflow.operators.pig_operator.PigOperator method)*, 293
preprocess_body() (*airflow.contrib.operators.gcp_function_operator.ZipPathPreprocessor method)*, 377
preprocess_config() (*airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator method)*, 470
PrestoCheckOperator (*class in airflow.operators.presto_check_operator)*, 294
PrestoException, 532
PrestoHook (*class in airflow.hooks.presto_hook)*, 532
PrestoIntervalCheckOperator (*class in airflow.operators.presto_check_operator)*, 295
PrestoToMySqlTransfer (*class in airflow.operators.presto_to_mysql)*, 295
PrestoValueCheckOperator (*class in airflow.operators.presto_check_operator)*, 294
previous_execution_date_success (*airflow.models.TaskInstance attribute)*, 701
previous_execution_date_success (*airflow.models.taskinstance.TaskInstance attribute)*, 676
previous_schedule() (*airflow.models.DAG method)*, 690
previous_schedule() (*airflow.models.dag.DAG method)*, 663
previous_start_date_success (*airflow.models.TaskInstance attribute)*, 701
previous_start_date_success (*airflow.models.taskinstance.TaskInstance attribute)*, 676
previous_ti (*airflow.models.TaskInstance attribute)*, 701
previous_ti (*airflow.models.taskinstance.TaskInstance attribute)*, 676
previous_ti_success (*airflow.models.TaskInstance attribute)*, 701
previous_ti_success (*airflow.models.taskinstance.TaskInstance attribute)*, 676
priority_weight (*airflow.models.TaskInstance attribute)*, 700
priority_weight (*airflow.models.taskinstance.TaskInstance attribute)*, 675
priority_weight_total (*airflow.models.BaseOperator attribute)*, 685
priority_weight_total (*airflow.models.baseoperator.BaseOperator attribute)*, 657
process_body() (*airflow.contrib.operators.gcp_cloud_build_operator.BuildProcessor method)*, 368
process_body() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobPreprocess method)*, 394
process_error() (*airflow.executors.kubernetes_executor.KubernetesJobWatcher method)*, 647
process_file() (*airflow.models.DagBag method)*, 695
process_file() (*airflow.models.dagbag.DagBag method)*, 669
process_status() (*airflow.executors.kubernetes_executor.KubernetesJobWatcher method)*, 647
process_watcher_task() (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler method)*, 648
product_name_determiner (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook attribute)*, 606
product_set_name_determiner (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook attribute)*, 606
project_id (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseH attribute)*, 571
PROJECT_ID (*in module airflow.contrib.hooks.gcp_transfer_hook)*, 601
provide_gcp_credential_file() (*airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook._Dec static method)*, 571
publish() (*airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method)*, 588
publish_to_target() (*airflow.contrib.hooks.aws_sns_hook.AwsSnsHook method)*, 540
PubSubException, 588
PubSubHook (*class in airflow.contrib.hooks.gcp_pubsub_hook)*, 588
PubSubPublishOperator (*class in airflow.contrib.operators.pubsub_operator)*, 459
PubSubPullSensor (*class in airflow.contrib.sensors.pubsub_sensor)*, 506
PubSubSubscriptionCreateOperator (*class in airflow.contrib.operators.pubsub_operator)*, 456
PubSubSubscriptionDeleteOperator (*class in airflow.contrib.operators.pubsub_operator)*, 458
PubSubTopicCreateOperator (*class in airflow.contrib.operators.pubsub_operator)*, 456
PubSubTopicDeleteOperator (*class in air-*

flow.contrib.operators.pubsub_operator), 458
pull() (*airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook method*), 589
push_from_local() (*airflow.hooks.samba_hook.SambaHook method*), 533
PUT (*airflow.contrib.operators.sftp_operator.SFTPOperation attribute*), 475
put_records() (*airflow.contrib.hooks.aws_firehose_hook.AwsFirehoseHook method*), 537
PythonOperator (*class in airflow.operators.python_operator*), 296
PythonSensor (*class in airflow.contrib.sensors.python_sensor*), 507
PythonVirtualenvOperator (*class in airflow.operators.python_operator*), 297

Q

QDSLink (*class in airflow.contrib.operators.qubole_operator*), 462
qubole_hook_allowed_args_list (*airflow.contrib.operators.qubole_operator.QuboleOperator attribute*), 464
QuboleCheckHook (*class in airflow.contrib.hooks.qubole_check_hook*), 619
QuboleCheckOperator (*class in airflow.contrib.operators.qubole_check_operator*), 460
QuboleFileSensor (*class in airflow.contrib.sensors.qubole_sensor*), 507
QuboleHook (*class in airflow.contrib.hooks.qubole_hook*), 619
QuboleOperator (*class in airflow.contrib.operators.qubole_operator*), 462
QubolePartitionSensor (*class in airflow.contrib.sensors.qubole_sensor*), 508
QuboleSensor (*class in airflow.contrib.sensors.qubole_sensor*), 507
QuboleValueCheckOperator (*class in airflow.contrib.operators.qubole_check_operator*), 461
query_metric() (*airflow.contrib.hooks.datadog_hook.DatadogHook method*), 562
queue (*airflow.models.TaskInstance attribute*), 700
queue (*airflow.models.taskinstance.TaskInstance attribute*), 675
queue_command() (*airflow.executors.base_executor.BaseExecutor method*), 644
queue_command() (*airflow.executors.BaseExecutor method*), 651
queue_task_instance() (*airflow.executors.base_executor.BaseExecutor method*), 644
queue_task_instance() (*airflow.executors.BaseExecutor method*), 651
queued_dttm (*airflow.models.TaskInstance attribute*), 700
queued_dttm (*airflow.models.taskinstance.TaskInstance attribute*), 675
queued_slots() (*airflow.models.Pool method*), 699
queued_slots() (*airflow.models.pool.Pool method*), 673
QueuedLocalWorker (*class in airflow.executors.local_executor*), 650

R

raise_error() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJob method*), 580
random() (*in module airflow.macros*), 273
ratio_formulas (*airflow.operators.check_operator.IntervalCheckOperator attribute*), 277
read_file() (*airflow.contrib.hooks.wasb_hook.WasbHook method*), 642
read_key() (*airflow.hooks.S3_hook.S3Hook method*), 515
ready_for_retry() (*airflow.models.TaskInstance method*), 703
ready_for_retry() (*airflow.models.taskinstance.TaskInstance method*), 677
recognize_speech() (*airflow.contrib.hooks.gcp_speech_to_text_hook.GCPSpeechToTextHook method*), 593
record_exists() (*airflow.contrib.hooks.cassandra_hook.CassandraHook method*), 560
RedisHook (*class in airflow.contrib.hooks.redis_hook*), 620
RedisKeySensor (*class in airflow.contrib.sensors.redis_key_sensor*), 508
RedisPublishOperator (*class in airflow.contrib.operators.redis_publish_operator*), 465
RedisPubSubSensor (*class in airflow.contrib.sensors.redis_pub_sub_sensor*), 508
RedshiftHook (*class in airflow.contrib.hooks.redshift_hook*), 620
RedshiftToS3Transfer (*class in airflow.operators.redshift_to_s3_operator*), 299
refresh_from_db() (*airflow.models.DagRun method*), 697

refresh_from_db() (*airflow.models.dagrun.DagRun method*), 670
refresh_from_db() (*airflow.models.TaskInstance method*), 702
refresh_from_db() (*airflow.models.taskinstance.TaskInstance method*), 677
REGEX_REPO_PATH (*in module airflow.contrib.operators.gcp_cloud_build_operator*), 367
remove_product_from_product_set() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
rename() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 570
render_template() (*airflow.models.BaseOperator method*), 686
render_template() (*airflow.models.baseoperator.BaseOperator method*), 658
render_template_from_field() (*airflow.models.BaseOperator method*), 686
render_template_from_field() (*airflow.models.baseoperator.BaseOperator method*), 658
render_templates() (*airflow.models.TaskInstance method*), 704
render_templates() (*airflow.models.taskinstance.TaskInstance method*), 678
replace_many() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 616
replace_one() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
REQUIRED_ATTRIBUTES (*airflow.contrib.operators.gcp_bigtable_operator.BigtableClusterUpdateOperator attribute*), 367
REQUIRED_ATTRIBUTES (*airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceCreateOperator attribute*), 364
REQUIRED_ATTRIBUTES (*airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceDeleteOperator attribute*), 365
REQUIRED_ATTRIBUTES (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableCreateOperator attribute*), 366
REQUIRED_ATTRIBUTES (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableDeleteOperator attribute*), 366
REQUIRED_ATTRIBUTES (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableWaitForReplicationSensor attribute*), 367
REQUIRED_ATTRIBUTES (*airflow.contrib.operators.gcp_bigtable_operator.BigtableValidationMix attribute*), 363
reschedule (*airflow.sensors.base_sensor_operator.BaseSensorOperator attribute*), 306
reschedule_date (*airflow.models.TaskReschedule attribute*), 705
reschedule_date (*airflow.models.taskreschedule.TaskReschedule attribute*), 680
reserve_free_tcp_port() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 599
reset_resource_version() (*airflow.models.KubeResourceVersion static method*), 698
reset_resource_version() (*airflow.models.kubernetes.KubeResourceVersion static method*), 672
resolve_template_files() (*airflow.models.BaseOperator method*), 686
resolve_template_files() (*airflow.models.baseoperator.BaseOperator method*), 658
resolve_template_files() (*airflow.models.DAG method*), 691
resolve_template_files() (*airflow.models.dag.DAG method*), 664
resource_version (*airflow.models.KubeResourceVersion attribute*), 698
resource_version (*airflow.models.kubernetes.KubeResourceVersion attribute*), 672
restart_cluster() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
RESTART_CLUSTER_ENDPOINT (*in module airflow.contrib.hooks.databricks_hook*), 560
restore_from_cluster_snapshot() (*airflow.contrib.hooks.redshift_hook.RedshiftHook method*), 620
resume_transfer_operation() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 603
retrieve_connection() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598
retrieve_file() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 569
retrieve_file() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 628

retrieve_mail_attachments() (*airflow.contrib.hooks.imap_hook.ImapHook method*), 612
rewrite() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 608
rollback() (*airflow.contrib.hooks.bigquery_hook.BigQueryConnection method*), 548
rollback() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 564
roots (*airflow.models.DAG attribute*), 690
roots (*airflow.models.dag.DAG attribute*), 663
rotate_fernet_key() (*airflow.models.Connection method*), 687
rotate_fernet_key() (*airflow.models.connection.Connection method*), 660
rotate_fernet_key() (*airflow.models.Variable method*), 705
rotate_fernet_key() (*airflow.models.variable.Variable method*), 680
ROW_DELIM (*in module airflow.contrib.hooks.qubole_check_hook*), 618
rowcount (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor attribute*), 557
run() (*airflow.contrib.hooks.grpc_hook.GrpcHook method*), 611
run() (*airflow.executors.kubernetes_executor.KubernetesJobWatcher method*), 647
run() (*airflow.executors.local_executor.LocalWorker method*), 650
run() (*airflow.executors.local_executor.QueuedLocalWorker method*), 650
run() (*airflow.hooks.base_hook.BaseHook method*), 518
run() (*airflow.hooks.dbapi_hook.DbApiHook method*), 518
run() (*airflow.hooks.http_hook.HttpHook method*), 527
run() (*airflow.hooks.presto_hook.PrestoHook method*), 532
run() (*airflow.models.BaseOperator method*), 686
run() (*airflow.models.baseoperator.BaseOperator method*), 658
run() (*airflow.models.DAG method*), 692
run() (*airflow.models.dag.DAG method*), 665
run() (*airflow.models.TaskInstance method*), 703
run() (*airflow.models.taskinstance.TaskInstance method*), 678
run_and_check() (*airflow.hooks.http_hook.HttpHook method*), 527
run_cli() (*airflow.hooks.hive_hooks.HiveCliHook method*), 522
run_cli() (*airflow.hooks.pig_hook.PigCliHook method*), 531
run_copy() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 552
run_extract() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 552
run_grant_dataset_view_access() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 555
run_id (*airflow.models.DagRun attribute*), 696
run_id (*airflow.models.dagrun.DagRun attribute*), 670
RUN_LIFE_CYCLE_STATES (*in module airflow.contrib.hooks.databricks_hook*), 562
run_load() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 553
run_next() (*airflow.executors.kubernetes_executor.AirflowKubernetesSche method*), 648
run_now() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
RUN_NOW_ENDPOINT (*in module airflow.contrib.hooks.databricks_hook*), 560
run_query() (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook method*), 535
run_query() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 551
run_query() (*airflow.contrib.hooks.datastore_hook.DatastoreHook method*), 564
run_query() (*airflow.contrib.hooks.spark_sql_hook.SparkSqlHook method*), 634
run_table_delete() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 555
run_table_upsert() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 555
run_with_advanced_retry() (*airflow.hooks.http_hook.HttpHook method*), 527
run_with_configuration() (*airflow.contrib.hooks.bigquery_hook.BigQueryBaseCursor method*), 554
RUNNING (*airflow.contrib.hooks.gcp_compute_hook.GceOperationStatus attribute*), 575
RUNNING (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlOperationStatus attribute*), 594
RunState (*class in airflow.contrib.hooks.databricks_hook*), 562

S

S3CopyObjectOperator (*class in airflow.contrib.operators.s3_copy_object_operator*), 465
S3DeleteObjectsOperator (*class in airflow.contrib.operators.s3_delete_objects_operator*), 466
S3FileTransformOperator (*class in airflow.operators.s3_file_transform_operator*),

300
S3Hook (*class in airflow.hooks.S3_hook*), 514
S3KeySensor (*class in airflow.sensors.s3_key_sensor*), 310
S3ListOperator (*class in airflow.contrib.operators.s3_list_operator*), 467
S3PrefixSensor (*class in airflow.sensors.s3_prefix_sensor*), 311
S3ToGoogleCloudStorageOperator (*class in airflow.contrib.operators.s3_to_gcs_operator*), 468
S3ToGoogleCloudStorageTransferOperator (*class in airflow.contrib.operators.gcp_transfer_operator*), 398
S3ToHiveTransfer (*class in airflow.operators.s3_to_hive_operator*), 301
S3ToRedshiftTransfer (*class in airflow.operators.s3_to_redshift_operator*), 302
S3ToSFTPOperator (*class in airflow.contrib.operators.s3_to_sftp_operator*), 469
safe_dag_id (*airflow.models.dag.DagModel attribute*), 667
safe_dag_id (*airflow.models.DagModel attribute*), 694
safe_search_detection() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607
SageMakerBaseOperator (*class in airflow.contrib.operators.sagemaker_base_operator*), 470
SageMakerBaseSensor (*class in airflow.contrib.sensors.sagemaker_base_sensor*), 509
SageMakerEndpointConfigOperator (*class in airflow.contrib.operators.sagemaker_endpoint_config_operator*), 470
SageMakerEndpointOperator (*class in airflow.contrib.operators.sagemaker_endpoint_operator*), 471
SageMakerEndpointSensor (*class in airflow.contrib.sensors.sagemaker_endpoint_sensor*), 509
SageMakerHook (*class in airflow.contrib.hooks.sagemaker_hook*), 621
SageMakerModelOperator (*class in airflow.contrib.operators.sagemaker_model_operator*), 472
SageMakerTrainingOperator (*class in airflow.contrib.operators.sagemaker_training_operator*), 472
SageMakerTrainingSensor (*class in airflow.contrib.sensors.sagemaker_training_sensor*), 510
SageMakerTransformOperator (*class in airflow.contrib.operators.sagemaker_transform_operator*), 473
SageMakerTransformSensor (*class in airflow.contrib.sensors.sagemaker_transform_sensor*), 510
SageMakerTuningOperator (*class in airflow.contrib.operators.sagemaker_tuning_operator*), 474
SageMakerTuningSensor (*class in airflow.contrib.sensors.sagemaker_tuning_sensor*), 511
SalesforceHook (*class in airflow.contrib.hooks.salesforce_hook*), 626
SambaHook (*class in airflow.hooks.samba_hook*), 533
sanitize_queries() (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance static method*), 383
SCHEDULE (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
SCHEDULE_END_DATE (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
schedule_interval (*airflow.models.BaseOperator attribute*), 684
schedule_interval (*airflow.models.baseoperator.BaseOperator attribute*), 657
schedule_interval (*airflow.models.dag.DagModel attribute*), 667
schedule_interval (*airflow.models.DagModel attribute*), 694
SCHEDULE_START_DATE (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
ScheduleInterval (*in module airflow.models.dag*), 661
scheduler_lock (*airflow.models.dag.DagModel attribute*), 667
scheduler_lock (*airflow.models.DagModel attribute*), 694
schema (*airflow.models.Connection attribute*), 687
schema (*airflow.models.connection.Connection attribute*), 659
secondary_training_status_changed() (*in module airflow.contrib.hooks.sagemaker_hook*), 621
secondary_training_status_message() (*in module airflow.contrib.hooks.sagemaker_hook*), 621
SECONDS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
SECRET_ACCESS_KEY (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
SegmentHook (*class in airflow.contrib.hooks.segment_hook*), 628
SegmentTrackEventOperator (*class in airflow.contrib.operators.segment_track_event_operator*),

475
select_key() (*airflow.hooks.S3_hook.S3Hook method*), 515
send() (*airflow.contrib.hooks.dingding_hook.DingdingHook method*), 567
send_message() (*airflow.contrib.hooks.aws_sqs_hook.SQSHook method*), 540
send_metric() (*airflow.contrib.hooks.datadog_hook.DatadogHook method*), 562
send_task_to_executor() (*in module airflow.executors.celery_executor*), 646
SequentialExecutor (*airflow.executors.Executors attribute*), 653
SequentialExecutor (*class in airflow.executors*), 652
SequentialExecutor (*class in airflow.executors.sequential_executor*), 650
set() (*airflow.models.Variable class method*), 705
set() (*airflow.models.variable.Variable class method*), 680
set() (*airflow.models.XCom class method*), 706
set() (*airflow.models.xcom.XCom class method*), 681
set_arraysize() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558
set_autocommit() (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook method*), 618
set_autocommit() (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook method*), 630
set_autocommit() (*airflow.hooks.dbapi_hook.DbApiHook method*), 519
set_autocommit() (*airflow.hooks.druid_hook.DruidDbApiHook method*), 521
set_autocommit() (*airflow.hooks.jdbc_hook.JdbcHook method*), 528
set_autocommit() (*airflow.hooks.mssql_hook.MsSqlHook method*), 528
set_autocommit() (*airflow.hooks.mysql_hook.MySqlHook method*), 529
set_common_options() (*in module airflow.contrib.hooks.spark_jdbc_script*), 633
set_dag_runs_state() (*airflow.models.DAG method*), 691
set_dag_runs_state() (*airflow.models.dag.DAG method*), 665
set_default_version() (*airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook method*), 585
set_dependency() (*airflow.models.DAG method*), 691
set_dependency() (*airflow.models.dag.DAG method*), 664
set_downstream() (*airflow.models.BaseOperator method*), 686
set_downstream() (*airflow.models.baseoperator.BaseOperator method*), 658
set_duration() (*airflow.models.TaskInstance method*), 704
set_duration() (*airflow.models.taskinstance.TaskInstance method*), 678
set_extra() (*airflow.models.Connection method*), 687
set_extra() (*airflow.models.connection.Connection method*), 660
set_is_paused() (*airflow.models.dag.DagModel method*), 668
set_is_paused() (*airflow.models.DagModel method*), 694
set_job_name() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
set_machine_type() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 576
SET_MACHINE_TYPE_VALIDATION_SPECIFICATION (*in module airflow.contrib.operators.gcp_compute_operator*), 370
set_main() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuila method*), 581
set_password() (*airflow.models.Connection method*), 687
set_password() (*airflow.models.connection.Connection method*), 660
set_python_main() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJobBuilder method*), 581
set_state() (*airflow.models.DagRun method*), 697
set_state() (*airflow.models.dagrun.DagRun method*), 670
set_state() (*airflow.models.TaskInstance method*), 702
set_state() (*airflow.models.taskinstance.TaskInstance method*), 677
set_upstream() (*airflow.models.BaseOperator method*), 686
set_upstream() (*air-*

flow.models.baseoperator.BaseOperator method), 659
set_val() (*airflow.models.Variable method*), 705
set_val() (*airflow.models.variable.Variable method*), 680
setdefault() (*airflow.models.Variable class method*), 705
setdefault() (*airflow.models.variable.Variable class method*), 680
setinputsizes() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558
setoutputsize() (*airflow.contrib.hooks.bigquery_hook.BigQueryCursor method*), 558
SETTINGS (*in module airflow.contrib.operators.gcp_sql_operator*), 385
SETTINGS_VERSION (*in module airflow.contrib.operators.gcp_sql_operator*), 385
SFTPHook (*class in airflow.contrib.hooks.sftp_hook*), 628
SFTPOperation (*class in airflow.contrib.operators.sftp_operator*), 475
SFTPOperator (*class in airflow.contrib.operators.sftp_operator*), 475
SFTPSensor (*class in airflow.contrib.sensors.sftp_sensor*), 511
SFTPToS3Operator (*class in airflow.contrib.operators.sftp_to_s3_operator*), 477
shallow_copy_attrs (*airflow.models.BaseOperator attribute*), 684
shallow_copy_attrs (*airflow.models.baseoperator.BaseOperator attribute*), 657
shallow_copy_attrs (*airflow.operators.python_operator.PythonOperator attribute*), 296
ShortCircuitOperator (*class in airflow.operators.python_operator*), 297
should_upload_function() (*airflow.contrib.operators.gcp_function_operator.ZipPathPreprocessor method*), 377
shutdown_cluster() (*airflow.contrib.hooks.cassandra_hook.CassandraHook method*), 559
SimpleHttpOperator (*class in airflow.operators.http_operator*), 287
size() (*airflow.models.DagBag method*), 695
size() (*airflow.models.dagbag.DagBag method*), 668
skip() (*airflow.models.SkipMixin method*), 699
skip() (*airflow.models.skipmixin.SkipMixin method*), 673
SkipMixin (*class in airflow.models*), 699
SkipMixin (*class in airflow.models.skipmixin*), 673
SlackAPIOperator (*class in airflow.operators.slack_operator*), 303
SlackAPIPostOperator (*class in airflow.operators.slack_operator*), 303
SlackHook (*class in airflow.hooks.slack_hook*), 533
SlackWebhookHook (*class in airflow.contrib.hooks.slack_webhook_hook*), 629
SlackWebhookOperator (*class in airflow.contrib.operators.slack_webhook_operator*), 478
SlaMiss (*class in airflow.models*), 699
SlaMiss (*class in airflow.models.slamiss*), 673
slots (*airflow.models.Pool attribute*), 698
slots (*airflow.models.pool.Pool attribute*), 673
snakebite_loaded (*in module airflow.hooks.hdfs_hook*), 521
SnowflakeHook (*class in airflow.contrib.hooks.snowflake_hook*), 630
SnowflakeOperator (*class in airflow.contrib.operators.snowflake_operator*), 479
SnsPublishOperator (*class in airflow.contrib.operators.sns_publish_operator*), 479
SOURCE_REPOSITORY (*in module airflow.contrib.operators.gcp_function_operator*), 377
spark_read_from_jdbc() (*in module airflow.contrib.hooks.spark_jdbc_script*), 633
spark_write_to_jdbc() (*in module airflow.contrib.hooks.spark_jdbc_script*), 633
SparkJDBCHook (*class in airflow.contrib.hooks.spark_jdbc_hook*), 631
SparkJDBCOperator (*class in airflow.contrib.operators.spark_jdbc_operator*), 481
SparkSqlHook (*class in airflow.contrib.hooks.spark_sql_hook*), 633
SparkSqlOperator (*class in airflow.contrib.operators.spark_sql_operator*), 483
SparkSubmitHook (*class in airflow.contrib.hooks.spark_submit_hook*), 635
SparkSubmitOperator (*class in airflow.contrib.operators.spark_submit_operator*), 485
SQL_ALCHEMY_SCHEMA (*in module airflow.models.base*), 653
SqliteHook (*class in airflow.hooks.sqlite_hook*), 533
SqliteOperator (*class in airflow.operators.sqlite_operator*), 304
SqlSensor (*class in airflow.sensors.sql_sensor*), 312
SqoopHook (*class in airflow.contrib.hooks.sqoop_hook*),

637

SqoopOperator (*class in airflow.contrib.operators.sqoop_operator*), 487

SQSHook (*class in airflow.contrib.hooks.aws_sqs_hook*), 540

SQSPublishOperator (*class in airflow.contrib.operators.aws_sqs_publish_operator*), 316

SQSSensor (*class in airflow.contrib.sensors.aws_sqs_sensor*), 494

SSHHook (*class in airflow.contrib.hooks.ssh_hook*), 640

SSHOperator (*class in airflow.contrib.operators.ssh_operator*), 488

stacktrace (*airflow.models.errors.ImportError attribute*), 671

start() (*airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOperator method*), 349

start() (*airflow.contrib.operators.dataproc_operator.DataprocClusterDeleteOperator method*), 351

start() (*airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOperator method*), 350

start() (*airflow.contrib.operators.dataproc_operator.DataprocOperationBaseOperator method*), 346

start() (*airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateInlineOperator method*), 356

start() (*airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateOperator method*), 355

start() (*airflow.executors.base_executor.BaseExecutor method*), 644

start() (*airflow.executors.BaseExecutor method*), 651

start() (*airflow.executors.celery_executor.CeleryExecutor method*), 646

start() (*airflow.executors.dask_executor.DaskExecutor method*), 646

start() (*airflow.executors.kubernetes_executor.KubernetesExecutor method*), 649

start() (*airflow.executors.local_executor.LocalExecutor method*), 650

start() (*airflow.executors.local_executor.LocalExecutor._LimitedParallelism method*), 650

start() (*airflow.executors.local_executor.LocalExecutor._UnlimitedParallelism method*), 650

start() (*airflow.executors.LocalExecutor method*), 652

start() (*airflow.executors.LocalExecutor._LimitedParallelism method*), 652

start() (*airflow.executors.LocalExecutor._UnlimitedParallelism method*), 652

start_cluster() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561

START_CLUSTER_ENDPOINT (*in module airflow.contrib.hooks.databricks_hook*), 560

start_date (*airflow.models.DagRun attribute*), 696

start_date (*airflow.models.dagrun.DagRun attribute*), 670

start_date (*airflow.models.TaskFail attribute*), 699

start_date (*airflow.models.taskfail.TaskFail attribute*), 674

start_date (*airflow.models.TaskInstance attribute*), 700

start_date (*airflow.models.taskinstance.TaskInstance attribute*), 675

start_date (*airflow.models.TaskReschedule attribute*), 705

start_date (*airflow.models.taskreschedule.TaskReschedule attribute*), 679

start_instance() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 575

start_java_dataflow() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook method*), 580

start_proxy() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner method*), 597

start_python_dataflow() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook method*), 580

start_template_dataflow() (*airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook method*), 580

START_TIME_OF_DAY (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601

STARTING (*airflow.contrib.hooks.sagemaker_hook.LogState attribute*), 621

state (*airflow.models.DagRun attribute*), 696

state (*airflow.models.dagrun.DagRun attribute*), 670

state (*airflow.models.TaskInstance attribute*), 700

state (*airflow.models.taskinstance.TaskInstance attribute*), 675

state_from_response() (*airflow.contrib.sensors.emr_job_flow_sensor.EmrJobFlowSensor static method*), 499

state_from_response() (*airflow.contrib.sensors.emr_step_sensor.EmrStepSensor static method*), 499

state_from_response() (*airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor method*), 509

state_from_response() (*airflow.contrib.sensors.sagemaker_endpoint_sensor.SageMakerEndpoi method*), 510

state_from_response() (*airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTraining method*), 510

state_from_response() (*airflow.contrib.sensors.sagemaker_transform_sensor.SageMakerTrans method*), 510

state_from_response() (*airflow.contrib.sensors.sagemaker_tuning_sensor.SageMakerTuningSensor method*), 511
STATUS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
STATUS1 (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
status_code (*airflow.executors.AirflowException attribute*), 651
stop_instance() (*airflow.contrib.hooks.gcp_compute_hook.GceHook method*), 575
stop_proxy() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlProxyRunner method*), 597
stop_query() (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook method*), 536
store_file() (*airflow.contrib.hooks.ftp_hook.FTPHook method*), 570
store_file() (*airflow.contrib.hooks.sftp_hook.SFTPHook method*), 629
sub_dag() (*airflow.models.DAG method*), 692
sub_dag() (*airflow.models.dag.DAG method*), 665
SubDagOperator (*class in airflow.operators.subdag_operator*), 305
subdags (*airflow.models.DAG attribute*), 690
subdags (*airflow.models.dag.DAG attribute*), 663
submit() (*airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook method*), 581
submit() (*airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook method*), 636
submit_indexing_job() (*airflow.hooks.druid_hook.DruidHook method*), 520
submit_jdbc_job() (*airflow.contrib.hooks.spark_jdbc_hook.SparkJDBCHook method*), 632
submit_run() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561
SUBMIT_RUN_ENDPOINT (*in module airflow.contrib.hooks.databricks_hook*), 560
SUCCESS (*airflow.contrib.hooks.gcp_transfer_hook.GcpTransferOperationStatus attribute*), 600
success() (*airflow.executors.base_executor.BaseExecutor method*), 645
success() (*airflow.executors.BaseExecutor method*), 651
SUCCESS_STATES (*airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook attribute*), 535
SUCCESS_STATES (*airflow.contrib.sensors.aws_athena_sensor.AthenaSensor attribute*), 492
supports_autocommit (*airflow.contrib.hooks.pinot_hook.PinotDbApiHook attribute*), 618
supports_autocommit (*airflow.contrib.hooks.snowflake_hook.SnowflakeHook attribute*), 630
supports_autocommit (*airflow.contrib.hooks.vertica_hook.VerticaHook attribute*), 641
supports_autocommit (*airflow.hooks.dbapi_hook.DbApiHook attribute*), 518
supports_autocommit (*airflow.hooks.druid_hook.DruidDbApiHook attribute*), 520
supports_autocommit (*airflow.hooks.jdbc_hook.JdbcHook attribute*), 528
supports_autocommit (*airflow.hooks.mssql_hook.MsSqlHook attribute*), 528
supports_autocommit (*airflow.hooks.mysql_hook.MySqlHook attribute*), 529
supports_autocommit (*airflow.hooks.oracle_hook.OracleHook attribute*), 530
supports_autocommit (*airflow.hooks.postgres_hook.PostgresHook attribute*), 531
supports_autocommit (*airflow.hooks.sqlite_hook.SqliteHook attribute*), 533
sync() (*airflow.executors.base_executor.BaseExecutor method*), 644
sync() (*airflow.executors.BaseExecutor method*), 651
sync() (*airflow.executors.celery_executor.CeleryExecutor method*), 646
sync() (*airflow.executors.dask_executor.DaskExecutor method*), 646
sync() (*airflow.executors.kubernetes_executor.AirflowKubernetesScheduler method*), 648
sync() (*airflow.executors.kubernetes_executor.KubernetesExecutor method*), 649
sync() (*airflow.executors.local_executor.LocalExecutor method*), 650
sync() (*airflow.executors.local_executor.LocalExecutor._LimitedParallelism method*), 650
sync() (*airflow.executors.local_executor.LocalExecutor._UnlimitedParalleli method*), 650
sync() (*airflow.executors.LocalExecutor method*), 652
sync() (*airflow.executors.LocalExecutor._LimitedParallelism method*), 652
sync() (*airflow.executors.LocalExecutor._UnlimitedParallelism method*), 652
sync() (*airflow.executors.sequential_executor.SequentialExecutor*

method), 651
sync() (*airflow.executors.SequentialExecutor method*), 652
sync_to_db() (*airflow.models.DAG method*), 693
sync_to_db() (*airflow.models.dag.DAG method*), 666
synthesize_speech() (*airflow.contrib.hooks.gcp_text_to_speech_hook.GCPTextToSpeechHook method*), 599

T

table_exists() (*airflow.contrib.hooks.bigquery_hook.BigQueryHook method*), 548
table_exists() (*airflow.contrib.hooks.cassandra_hook.CassandraHook method*), 559
table_exists() (*airflow.hooks.hive_hooks.HiveMetastoreHook method*), 525
TAILING (*airflow.contrib.hooks.sagemaker_hook.LogState attribute*), 621
tar_and_s3_upload() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 621
task_id (*airflow.models.Log attribute*), 698
task_id (*airflow.models.log.Log attribute*), 672
task_id (*airflow.models.SlaMiss attribute*), 699
task_id (*airflow.models.slamiss.SlaMiss attribute*), 674
task_id (*airflow.models.TaskFail attribute*), 699
task_id (*airflow.models.taskfail.TaskFail attribute*), 674
task_id (*airflow.models.TaskInstance attribute*), 700
task_id (*airflow.models.taskinstance.TaskInstance attribute*), 675
task_id (*airflow.models.TaskReschedule attribute*), 704
task_id (*airflow.models.taskreschedule.TaskReschedule attribute*), 679
task_id (*airflow.models.XCom attribute*), 706
task_id (*airflow.models.xcom.XCom attribute*), 681
task_ids (*airflow.models.DAG attribute*), 689
task_ids (*airflow.models.dag.DAG attribute*), 662
task_type (*airflow.models.BaseOperator attribute*), 685
task_type (*airflow.models.baseoperator.BaseOperator attribute*), 657
TaskFail (*class in airflow.models*), 699
TaskFail (*class in airflow.models.taskfail*), 674
TaskInstance (*class in airflow.models*), 700
TaskInstance (*class in airflow.models.taskinstance*), 675
TaskReschedule (*class in airflow.models*), 704
TaskReschedule (*class in airflow.models.taskreschedule*), 679
tasks (*airflow.models.DAG attribute*), 689
tasks (*airflow.models.dag.DAG attribute*), 662
template_ext (*airflow.contrib.operators.aws_athena_operator.AWSAthen attribute*), 316
template_ext (*airflow.contrib.operators.bigquery_check_operator.BigQue attribute*), 320
template_ext (*airflow.contrib.operators.bigquery_check_operator.BigQue attribute*), 321
template_ext (*airflow.contrib.operators.bigquery_operator.BigQueryOper attribute*), 325
template_ext (*airflow.contrib.operators.bigquery_to_bigquery.BigQueryT attribute*), 332
template_ext (*airflow.contrib.operators.bigquery_to_gcs.BigQueryToClou attribute*), 333
template_ext (*airflow.contrib.operators.cassandra_to_gcs.CassandraToG attribute*), 334
template_ext (*airflow.contrib.operators.dataproc_operator.DataProcHive attribute*), 353
template_ext (*airflow.contrib.operators.dataproc_operator.DataProcPig attribute*), 352
template_ext (*airflow.contrib.operators.dataproc_operator.DataProcSpa attribute*), 353
template_ext (*airflow.contrib.operators.druid_operator.DruidOperator attribute*), 360
template_ext (*airflow.contrib.operators.emr_add_steps_operator.EmrAd attribute*), 361
template_ext (*airflow.contrib.operators.emr_create_job_flow_operator.E attribute*), 362
template_ext (*airflow.contrib.operators.emr_terminate_job_flow_operat attribute*), 362
template_ext (*airflow.contrib.operators.gcp_spanner_operator.CloudSpa attribute*), 383
template_ext (*airflow.contrib.operators.gcp_spanner_operator.CloudSpa attribute*), 382
template_ext (*airflow.contrib.operators.gcp_spanner_operator.CloudSpa attribute*), 384
template_ext (*airflow.contrib.operators.gcp_sql_operator.CloudSqlQuery attribute*), 392
template_ext (*airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageT attribute*), 430
template_ext (*airflow.contrib.operators.hive_to_dynamodb.HiveToDyna attribute*), 436
template_ext (*airflow.contrib.operators.jenkins_job_trigger_operator.Jen attribute*), 438
template_ext (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleClo attribute*), 449
template_ext (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleClo attribute*), 451
template_ext (*airflow.contrib.operators.postgres_to_gcs_operator.Postgre attribute*), 455
template_ext (*airflow.contrib.operators.qubole_check_operator.QuboleC attribute*), 460
template_ext (*airflow.contrib.operators.qubole_check_operator.QuboleV attribute*), 461
template_ext (*airflow.contrib.operators.qubole_operator.QuboleOperator attribute*), 464

template_ext (airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator attribute), 470
template_ext (airflow.contrib.operators.snowflake_operator.SnowflakeOperator attribute), 479
template_ext (airflow.contrib.operators.sns_publish_operator.SnsPublishOperator attribute), 480
template_ext (airflow.contrib.operators.spark_sql_operator.SparkSqlOperator attribute), 484
template_ext (airflow.contrib.operators.ssh_operator.SSHOperator attribute), 488
template_ext (airflow.contrib.operators.vertica_operator.VerticaOperator attribute), 489
template_ext (airflow.contrib.operators.vertica_to_hive.VerticaToHiveTransfer attribute), 490
template_ext (airflow.contrib.operators.vertica_to_mysql.VerticaToMySqlTransfer attribute), 490
template_ext (airflow.contrib.sensors.aws_athena_sensor.AthenaSensor attribute), 492
template_ext (airflow.contrib.sensors.emr_job_flow_sensor.EmrJobFlowSensor attribute), 499
template_ext (airflow.contrib.sensors.emr_step_sensor.EmrStepSensor attribute), 499
template_ext (airflow.contrib.sensors.qubole_sensor.QuboleSensor attribute), 507
template_ext (airflow.contrib.sensors.sagemaker_endpoint_sensor.SageMakerEndpointSensor attribute), 509
template_ext (airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTrainingSensor attribute), 510
template_ext (airflow.contrib.sensors.sagemaker_transform_sensor.SageMakerTransformSensor attribute), 510
template_ext (airflow.contrib.sensors.sagemaker_tuning_sensor.SageMakerTuningSensor attribute), 511
template_ext (airflow.models.BaseOperator attribute), 684
template_ext (airflow.models.baseoperator.BaseOperator attribute), 656
template_ext (airflow.operators.bash_operator.BashOperator attribute), 275
template_ext (airflow.operators.check_operator.CheckOperator attribute), 276
template_ext (airflow.operators.check_operator.IntervalCheckOperator attribute), 277
template_ext (airflow.operators.check_operator.ValueCheckOperator attribute), 277
template_ext (airflow.operators.docker_operator.DockerOperator attribute), 280
template_ext (airflow.operators.email_operator.EmailOperator attribute), 282
template_ext (airflow.operators.generic_transfer.GenericTransfer attribute), 283
template_ext (airflow.operators.hive_operator.HiveOperator attribute), 283
template_ext (airflow.operators.hive_to_druid.HiveToDruidTransfer attribute), 286
template_ext (airflow.operators.hive_to_mysql.HiveToMySqlTransfer attribute), 286
template_ext (airflow.operators.hive_to_samba_operator.Hive2SambaOp
attribute), 287
template_ext (airflow.operators.http_operator.SimpleHttpOperator attribute), 288
template_ext (airflow.operators.jdbc_operator.JdbcOperator attribute), 288
template_ext (airflow.operators.mssql_operator.MsSqlOperator attribute), 289
template_ext (airflow.operators.mssql_to_hive.MsSqlToHiveTransfer attribute), 290
template_ext (airflow.operators.mysql_operator.MySqlOperator attribute), 290
template_ext (airflow.operators.mysql_to_hive.MySqlToHiveTransfer attribute), 291
template_ext (airflow.operators.oracle_operator.OracleOperator attribute), 292
template_ext (airflow.operators.pig_operator.PigOperator attribute), 293
template_ext (airflow.operators.postgres_operator.PostgresOperator attribute), 294
template_ext (airflow.operators.presto_to_mysql.PrestoToMySqlTransfer attribute), 296
template_ext (airflow.operators.redshift_to_s3_operator.RedshiftToS3Tra
attribute), 299
template_ext (airflow.operators.s3_file_transform_operator.S3FileTrans
attribute), 300
template_ext (airflow.operators.s3_to_hive_operator.S3ToHiveTransfer attribute), 302
template_ext (airflow.operators.s3_to_redshift_operator.S3ToRedshiftTra
attribute), 303
template_ext (airflow.operators.sqlite_operator.SqliteOperator attribute), 304
template_ext (airflow.sensors.sql_sensor.SqlSensor attribute), 312
template_fields (airflow.contrib.operators.adls_list_operator.AzureDataLakeStorageList
attribute), 314
template_fields (airflow.contrib.operators.adls_to_gcs.AdlsToGoogleCloudStorageOpera
attribute), 315
template_fields (airflow.contrib.operators.aws_athena_operator.AWSAthenaOperator attribute), 316
template_fields (airflow.contrib.operators.aws_sqs_publish_operator.SQSPublishOperat
attribute), 316
template_fields (airflow.contrib.operators.awsbatch_operator.AWSBatchOperator attribute), 317
template_fields (airflow.contrib.operators.azure_container_instances_operator.AzureCo
attribute), 319

template_fields (*airflow.contrib.operators.azure_cosmos_operator.AzureCosmosInsertDocumentOperator attribute*), 320

template_fields (*airflow.contrib.operators.bigquery_check_operator.BigQueryCheckOperator attribute*), 320

template_fields (*airflow.contrib.operators.bigquery_check_operator.BigQueryIntervalCheckOperator attribute*), 321

template_fields (*airflow.contrib.operators.bigquery_check_operator.BigQueryValueCheckOperator attribute*), 321

template_fields (*airflow.contrib.operators.bigquery_get_data.BigQueryGetDataOperator attribute*), 323

template_fields (*airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyDatasetOperator attribute*), 330

template_fields (*airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyTableOperator attribute*), 327

template_fields (*airflow.contrib.operators.bigquery_operator.BigQueryCreateExternalTableOperator attribute*), 329

template_fields (*airflow.contrib.operators.bigquery_operator.BigQueryDeleteDatasetOperator attribute*), 330

template_fields (*airflow.contrib.operators.bigquery_operator.BigQueryOperator attribute*), 325

template_fields (*airflow.contrib.operators.bigquery_table_delete_operator.BigQueryTableDeleteOperator attribute*), 331

template_fields (*airflow.contrib.operators.bigquery_to_bigquery.BigQueryToBigQueryOperator attribute*), 332

template_fields (*airflow.contrib.operators.bigquery_to_gcs.BigQueryToCloudStorageOperator attribute*), 333

template_fields (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator attribute*), 334

template_fields (*airflow.contrib.operators.databricks_operator.DatabricksRunNowOperator attribute*), 340

template_fields (*airflow.contrib.operators.databricks_operator.DatabricksSubmitRunOperator attribute*), 338

template_fields (*airflow.contrib.operators.dataflow_operator.DataFlowJavaOperator attribute*), 343

template_fields (*airflow.contrib.operators.dataflow_operator.DataFlowPythonOperator attribute*), 345

template_fields (*airflow.contrib.operators.dataflow_operator.DataflowTemplateOperato attribute*), 344

template_fields (*airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOpe attribute*), 349

template_fields (*airflow.contrib.operators.dataproc_operator.DataprocClusterDeleteOpe attribute*), 351

template_fields (*airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOper attribute*), 350

template_fields (*airflow.contrib.operators.dataproc_operator.DataProcHadoopOperator attribute*), 354

template_fields (*airflow.contrib.operators.dataproc_operator.DataProcHiveOperator attribute*), 353

template_fields (*airflow.contrib.operators.dataproc_operator.DataProcPigOperator attribute*), 352

template_fields (*airflow.contrib.operators.dataproc_operator.DataProcPySparkOperato attribute*), 355

template_fields (*airflow.contrib.operators.dataproc_operator.DataProcSparkOperator attribute*), 354

template_fields (*airflow.contrib.operators.dataproc_operator.DataProcSparkSqlOperato attribute*), 353

template_fields (*airflow.contrib.operators.dataproc_operator.DataprocWorkflowTempla attribute*), 356

template_fields (*airflow.contrib.operators.dataproc_operator.DataprocWorkflowTempla attribute*), 355

template_fields (*airflow.contrib.operators.dingding_operator.DingdingOperator attribute*), 358

template_fields (*airflow.contrib.operators.discord_webhook_operator.DiscordWebhookO attribute*), 359

template_fields (*airflow.contrib.operators.druid_operator.DruidOperator attribute*), 360

template_fields (*airflow.contrib.operators.ecs_operator.ECSOperator attribute*), 361

template_fields (*airflow.contrib.operators.emr_add_steps_operator.EmrAddStepsOperat attribute*), 361

template_fields (*airflow.contrib.operators.emr_create_job_flow_operator.EmrCreateJol attribute*), 362 template_fields (*airflow.contrib.operators.emr_terminate_job_flow_operator.EmrTerminateJobFlowOperator attribute*), 362

template_fields (*airflow.contrib.operators.file_to_gcs.FileToGoogleCloudStorageOperator attribute*), 363

template_fields (*airflow.contrib.operators.file_to_wasb.FileToWasbOperator attribute*), 363

template_fields (*airflow.contrib.operators.gcp_bigtable_operator.BigtableClusterUpdateOperator attribute*), 367

template_fields (*airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceCreateOperator attribute*), 365

template_fields (*airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceDeleteOperator attribute*), 365

template_fields (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableCreateOperator attribute*), 366

template_fields (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableDeleteOperator attribute*), 366

template_fields (*airflow.contrib.operators.gcp_bigtable_operator.BigtableTableWaitForReplicationSensor attribute*), 367

template_fields (*airflow.contrib.operators.gcp_cloud_build_operator.CloudBuildCreateBuildOperator attribute*), 368

template_fields (*airflow.contrib.operators.gcp_compute_operator.GceInstanceGroupManagerUpdateTemplateOperator attribute*), 373

template_fields (*airflow.contrib.operators.gcp_compute_operator.GceInstanceStartOperator attribute*), 369

template_fields (*airflow.contrib.operators.gcp_compute_operator.GceInstanceStopOperator attribute*), 370

template_fields (*airflow.contrib.operators.gcp_compute_operator.GceInstanceTemplateCopyOperator attribute*), 372

template_fields (*airflow.contrib.operators.gcp_compute_operator.GceSetMachineTypeOperator attribute*), 371

template_fields (*airflow.contrib.operators.gcp_container_operator.GKEClusterCreateOperator attribute*), 375

template_fields (*airflow.contrib.operators.gcp_container_operator.GKEClusterDeleteOperator attribute*), 374

template_fields (*airflow.contrib.operators.gcp_container_operator.GKEPodOperator attribute*), 375

template_fields (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeleteOp attribute*), 378

template_fields (*airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOp attribute*), 377

template_fields (*airflow.contrib.operators.gcp_natural_language_operator.CloudLangu attribute*), 379

template_fields (*airflow.contrib.operators.gcp_natural_language_operator.CloudLangu attribute*), 379

template_fields (*airflow.contrib.operators.gcp_natural_language_operator.CloudLangu attribute*), 380

template_fields (*airflow.contrib.operators.gcp_natural_language_operator.CloudLangu attribute*), 381

template_fields (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance attribute*), 384

template_fields (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance attribute*), 383

template_fields (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance attribute*), 382

template_fields (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance attribute*), 384

template_fields (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance attribute*), 382

template_fields (*airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstance attribute*), 382

template_fields (*airflow.contrib.operators.gcp_speech_to_text_operator.GcpSpeechToTe attribute*), 385

template_fields (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceCreateOpe attribute*), 387

template_fields (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseC attribute*), 388

template_fields (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseI attribute*), 390

template_fields (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDatabaseI attribute*), 389

template_fields (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceDeleteOpe attribute*), 388

`template_fields` (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceExportOperator attribute*), 390
`template_fields` (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstanceImportOperator attribute*), 391
`template_fields` (*airflow.contrib.operators.gcp_sql_operator.CloudSqlInstancePatchOperator attribute*), 387
`template_fields` (*airflow.contrib.operators.gcp_sql_operator.CloudSqlQueryOperator attribute*), 392
`template_fields` (*airflow.contrib.operators.gcp_text_to_speech_operator.GcpTextToSpeechSynthesizeOperator attribute*), 393
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobCreateOperator attribute*), 395
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobDeleteOperator attribute*), 396
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobUpdateOperator attribute*), 395
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationCancelOperator attribute*), 398
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationGetOperator attribute*), 396
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationPauseOperator attribute*), 397
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationResumeOperator attribute*), 398
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationsListOperator attribute*), 397
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.GoogleCloudStorageToGoogleCloudStorageTransferOperator attribute*), 401
`template_fields` (*airflow.contrib.operators.gcp_transfer_operator.S3ToGoogleCloudStorageTransferOperator attribute*), 400
`template_fields` (*airflow.contrib.operators.gcp_translate_operator.CloudTranslateTextOperator attribute*), 402
`template_fields` (*airflow.contrib.operators.gcp_translate_speech_operator.GcpTranslateSpeechOperator attribute*), 404
`template_fields` (*airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVideoExplicitContentOperator attribute*), 405
`template_fields` (*airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoI attribute*), 405
`template_fields` (*airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoI attribute*), 406
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionAddProduct attribute*), 417
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionAnnotateIma attribute*), 415
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectDocum attribute*), 420
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectImage attribute*), 421
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectImage attribute*), 422
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectTextO attribute*), 419
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductCrea attribute*), 411
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductDele attribute*), 415
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductGetO attribute*), 412
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetC attribute*), 407
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetD attribute*), 410
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetG attribute*), 408
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetU attribute*), 410
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionProductUpd attribute*), 414
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionReferenceIm attribute*), 416
`template_fields` (*airflow.contrib.operators.gcp_vision_operator.CloudVisionRemoveProd attribute*), 418

template_fields (*air-flow.contrib.operators.gcs_acl_operator.GoogleCloudStorageBucketCreateAclEntryOperator attribute*), 423
template_fields (*air-flow.contrib.operators.gcs_acl_operator.GoogleCloudStorageObjectCreateAclEntryOperator attribute*), 423
template_fields (*air-flow.contrib.operators.gcs_delete_operator.GoogleCloudStorageDeleteOperator attribute*), 424
template_fields (*air-flow.contrib.operators.gcs_download_operator.GoogleCloudStorageDownloadOperator attribute*), 425
template_fields (*air-flow.contrib.operators.gcs_list_operator.GoogleCloudStorageListOperator attribute*), 426
template_fields (*air-flow.contrib.operators.gcs_operator.GoogleCloudStorageCreateBucketOperator attribute*), 427
template_fields (*air-flow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBigQueryOperator attribute*), 430
template_fields (*air-flow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator attribute*), 432
template_fields (*air-flow.contrib.operators.gcs_to_s3.GoogleCloudStorageToS3Operator attribute*), 433
template_fields (*air-flow.contrib.operators.grpc_operator.GrpcOperator attribute*), 434
template_fields (*air-flow.contrib.operators.hipchat_operator.HipChatAPISendRoomNotificationOperator attribute*), 436
template_fields (*air-flow.contrib.operators.hive_to_dynamodb.HiveToDynamoDBTransferOperator attribute*), 436
template_fields (*air-flow.contrib.operators.imap_attachment_to_s3_operator.ImapAttachmentToS3Operator attribute*), 437
template_fields (*air-flow.contrib.operators.jenkins_job_trigger_operator.JenkinsJobTriggerOperator attribute*), 438
template_fields (*air-flow.contrib.operators.jira_operator.JiraOperator attribute*), 439
template_fields (*air-flow.contrib.operators.kubernetes_pod_operator.KubernetesPodOperator attribute*), 443
template_fields (*air-flow.contrib.operators.mlengine_operator.MLEngineBatchPredictionOperator attribute*), 444
template_fields (*air-flow.contrib.operators.mlengine_operator.MLEngineModelOperator attribute*), 445
template_fields (*air-flow.contrib.operators.mlengine_operator.MLEngineTrainingOperat attribute*), 447
template_fields (*air-flow.contrib.operators.mlengine_operator.MLEngineVersionOperato attribute*), 446
template_fields (*air-flow.contrib.operators.mongo_to_s3.MongoToS3Operator attribute*), 448
template_fields (*air-flow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOp attribute*), 449
template_fields (*air-flow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOp attribute*), 451
template_fields (*air-flow.contrib.operators.opsgenie_alert_operator.OpsgenieAlertOperat attribute*), 453
template_fields (*air-flow.contrib.operators.oracle_to_azure_data_lake_transfer.OracleTo attribute*), 454
template_fields (*air-flow.contrib.operators.oracle_to_oracle_transfer.OracleToOracleTra attribute*), 454
template_fields (*air-flow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleC attribute*), 455
template_fields (*air-flow.contrib.operators.pubsub_operator.PubSubPublishOperator attribute*), 459
template_fields (*air-flow.contrib.operators.pubsub_operator.PubSubSubscriptionCreateO attribute*), 458
template_fields (*air-flow.contrib.operators.pubsub_operator.PubSubSubscriptionDeleteO attribute*), 459
template_fields (*air-flow.contrib.operators.pubsub_operator.PubSubTopicCreateOperator attribute*), 456
template_fields (*air-flow.contrib.operators.pubsub_operator.PubSubTopicDeleteOperator attribute*), 458
template_fields (*air-flow.contrib.operators.qubole_check_operator.QuboleCheckOperator attribute*), 460
template_fields (*air-flow.contrib.operators.qubole_check_operator.QuboleValueCheckOp attribute*), 461
template_fields (*air-flow.contrib.operators.qubole_operator.QuboleOperator attribute*), 464
template_fields (*air-flow.contrib.operators.redis_publish_operator.RedisPublishOperator attribute*), 465

template_fields (*airflow.contrib.operators.s3_copy_object_operator.S3CopyObjectOperator attribute*), 466

template_fields (*airflow.contrib.operators.s3_delete_objects_operator.S3DeleteObjectsOperator attribute*), 467

template_fields (*airflow.contrib.operators.s3_list_operator.S3ListOperator attribute*), 467

template_fields (*airflow.contrib.operators.s3_to_gcs_operator.S3ToGoogleCloudStorageOperator attribute*), 469

template_fields (*airflow.contrib.operators.s3_to_sftp_operator.S3ToSFTPOperator attribute*), 469

template_fields (*airflow.contrib.operators.sagemaker_base_operator.SageMakerBaseOperator attribute*), 470

template_fields (*airflow.contrib.operators.segment_track_event_operator.SegmentTrackEventOperator attribute*), 475

template_fields (*airflow.contrib.operators.sftp_operator.SFTPOperator attribute*), 476

template_fields (*airflow.contrib.operators.sftp_to_s3_operator.SFTPToS3Operator attribute*), 477

template_fields (*airflow.contrib.operators.snowflake_operator.SnowflakeOperator attribute*), 479

template_fields (*airflow.contrib.operators.sns_publish_operator.SnsPublishOperator attribute*), 480

template_fields (*airflow.contrib.operators.spark_sql_operator.SparkSqlOperator attribute*), 484

template_fields (*airflow.contrib.operators.spark_submit_operator.SparkSubmitOperator attribute*), 486

template_fields (*airflow.contrib.operators.sqoop_operator.SqoopOperator attribute*), 487

template_fields (*airflow.contrib.operators.ssh_operator.SSHOperator attribute*), 488

template_fields (*airflow.contrib.operators.vertica_operator.VerticaOperator attribute*), 489

template_fields (*airflow.contrib.operators.vertica_to_hive.VerticaToHiveTransfer attribute*), 490

template_fields (*airflow.contrib.operators.vertica_to_mysql.VerticaToMySqlTransfer attribute*), 490

template_fields (*airflow.contrib.operators.wasb_delete_blob_operator.WasbDeleteBlob attribute*), 491

template_fields (*airflow.contrib.operators.winrm_operator.WinRMOperator attribute*), 491

template_fields (*airflow.contrib.sensors.aws_athena_sensor.AthenaSensor attribute*), 492

template_fields (*airflow.contrib.sensors.aws_glue_catalog_partition_sensor.AwsGlueCa attribute*), 493

template_fields (*airflow.contrib.sensors.aws_redshift_cluster_sensor.AwsRedshiftCluster attribute*), 494

template_fields (*airflow.contrib.sensors.aws_sqs_sensor.SQSSensor attribute*), 494

template_fields (*airflow.contrib.sensors.azure_cosmos_sensor.AzureCosmosDocumentSe attribute*), 495

template_fields (*airflow.contrib.sensors.bash_sensor.BashSensor attribute*), 495

template_fields (*airflow.contrib.sensors.bigquery_sensor.BigQueryTableSensor attribute*), 496

template_fields (*airflow.contrib.sensors.cassandra_record_sensor.CassandraRecordSen attribute*), 497

template_fields (*airflow.contrib.sensors.cassandra_table_sensor.CassandraTableSensor attribute*), 497

template_fields (*airflow.contrib.sensors.emr_job_flow_sensor.EmrJobFlowSensor attribute*), 499

template_fields (*airflow.contrib.sensors.emr_step_sensor.EmrStepSensor attribute*), 499

template_fields (*airflow.contrib.sensors.file_sensor.FileSensor attribute*), 500

template_fields (*airflow.contrib.sensors.ftp_sensor.FTPSensor attribute*), 500

template_fields (*airflow.contrib.sensors.gcp_transfer_sensor.GCPTransferServiceWaitF attribute*), 501

template_fields (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectSensor attribute*), 501

template_fields (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectUpdatedS attribute*), 502

template_fields (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStoragePrefixSensor attribute*), 503
template_fields (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageUploadSessionCompleteSensor attribute*), 504
template_fields (*airflow.contrib.sensors.imap_attachment_sensor.ImapAttachmentSensor attribute*), 505
template_fields (*airflow.contrib.sensors.jira_sensor.JiraTicketSensor attribute*), 505
template_fields (*airflow.contrib.sensors.mongo_sensor.MongoSensor attribute*), 506
template_fields (*airflow.contrib.sensors.pubsub_sensor.PubSubPullSensor attribute*), 506
template_fields (*airflow.contrib.sensors.python_sensor.PythonSensor attribute*), 507
template_fields (*airflow.contrib.sensors.qubole_sensor.QuboleSensor attribute*), 507
template_fields (*airflow.contrib.sensors.redis_key_sensor.RedisKeySensor attribute*), 508
template_fields (*airflow.contrib.sensors.redis_pub_sub_sensor.RedisPubSubSensor attribute*), 508
template_fields (*airflow.contrib.sensors.sagemaker_endpoint_sensor.SageMakerEndpointSensor attribute*), 509
template_fields (*airflow.contrib.sensors.sagemaker_training_sensor.SageMakerTrainingSensor attribute*), 510
template_fields (*airflow.contrib.sensors.sagemaker_transform_sensor.SageMakerTransformSensor attribute*), 510
template_fields (*airflow.contrib.sensors.sagemaker_tuning_sensor.SageMakerTuningSensor attribute*), 511
template_fields (*airflow.contrib.sensors.sftp_sensor.SFTPSensor attribute*), 511
template_fields (*airflow.contrib.sensors.wasb_sensor.WasbBlobSensor attribute*), 512
template_fields (*airflow.contrib.sensors.wasb_sensor.WasbPrefixSensor attribute*), 512
template_fields (*airflow.models.BaseOperator attribute*), 684
template_fields (*airflow.models.baseoperator.BaseOperator attribute*), 656
template_fields (*airflow.operators.bash_operator.BashOperator attribute*), 275
template_fields (*airflow.operators.check_operator.CheckOperator attribute*), 276
template_fields (*airflow.operators.check_operator.IntervalCheckOperator attribute*), 277
template_fields (*airflow.operators.check_operator.ValueCheckOperator attribute*), 277
template_fields (*airflow.operators.dagrun_operator.TriggerDagRunOperator attribute*), 278
template_fields (*airflow.operators.docker_operator.DockerOperator attribute*), 280
template_fields (*airflow.operators.email_operator.EmailOperator attribute*), 282
template_fields (*airflow.operators.generic_transfer.GenericTransfer attribute*), 283
template_fields (*airflow.operators.hive_operator.HiveOperator attribute*), 283
template_fields (*airflow.operators.hive_stats_operator.HiveStatsCollectionOperator attribute*), 285
template_fields (*airflow.operators.hive_to_druid.HiveToDruidTransfer attribute*), 286
template_fields (*airflow.operators.hive_to_mysql.HiveToMySqlTransfer attribute*), 286
template_fields (*airflow.operators.hive_to_samba_operator.Hive2SambaOperator attribute*), 287
template_fields (*airflow.operators.http_operator.SimpleHttpOperator attribute*), 288
template_fields (*airflow.operators.jdbc_operator.JdbcOperator attribute*), 288
template_fields (*airflow.operators.mssql_operator.MsSqlOperator attribute*), 289
template_fields (*airflow.operators.mssql_to_hive.MsSqlToHiveTransfer attribute*), 290
template_fields (*air-*

flow.operators.mysql_operator.MySqlOperator attribute), 290

template_fields (*airflow.operators.mysql_to_hive.MySqlToHiveTransfer attribute*), 291

template_fields (*airflow.operators.oracle_operator.OracleOperator attribute*), 292

template_fields (*airflow.operators.pig_operator.PigOperator attribute*), 293

template_fields (*airflow.operators.postgres_operator.PostgresOperator attribute*), 294

template_fields (*airflow.operators.presto_to_mysql.PrestoToMySqlTransfer attribute*), 296

template_fields (*airflow.operators.python_operator.PythonOperator attribute*), 296

template_fields (*airflow.operators.redshift_to_s3_operator.RedshiftToS3Transfer attribute*), 299

template_fields (*airflow.operators.s3_file_transform_operator.S3FileTransformOperator attribute*), 300

template_fields (*airflow.operators.s3_to_hive_operator.S3ToHiveTransfer attribute*), 302

template_fields (*airflow.operators.s3_to_redshift_operator.S3ToRedshiftTransfer attribute*), 303

template_fields (*airflow.operators.slack_operator.SlackAPIPostOperator attribute*), 304

template_fields (*airflow.operators.sqlite_operator.SqliteOperator attribute*), 304

template_fields (*airflow.sensors.external_task_sensor.ExternalTaskSensor attribute*), 307

template_fields (*airflow.sensors.hdfs_sensor.HdfsSensor attribute*), 307

template_fields (*airflow.sensors.hive_partition_sensor.HivePartitionSensor attribute*), 308

template_fields (*airflow.sensors.http_sensor.HttpSensor attribute*), 309

template_fields (*airflow.sensors.metastore_partition_sensor.MetastorePartitionSensor attribute*), 310

template_fields (*airflow.sensors.named_hive_partition_sensor.NamedHivePartitionSens attribute*), 310

template_fields (*airflow.sensors.s3_key_sensor.S3KeySensor attribute*), 311

template_fields (*airflow.sensors.s3_prefix_sensor.S3PrefixSensor attribute*), 312

template_fields (*airflow.sensors.sql_sensor.SqlSensor attribute*), 312

template_fields (*airflow.sensors.web_hdfs_sensor.WebHdfsSensor attribute*), 313

terminate() (*airflow.executors.base_executor.BaseExecutor method*), 645

terminate() (*airflow.executors.BaseExecutor method*), 652

terminate() (*airflow.executors.dask_executor.DaskExecutor method*), 646

terminate() (*airflow.executors.kubernetes_executor.AirflowKubernetesSch method*), 649

terminate_cluster() (*airflow.contrib.hooks.databricks_hook.DatabricksHook method*), 561

TERMINATE_CLUSTER_ENDPOINT (*in module airflow.contrib.hooks.databricks_hook*), 560

test_cycle() (*airflow.models.DAG method*), 693

test_cycle() (*airflow.models.dag.DAG method*), 667

test_hql() (*airflow.hooks.hive_hooks.HiveCliHook method*), 522

text_detection() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 607

TIME_TO_SLEEP_IN_SECONDS (*in module airflow.contrib.hooks.gcp_cloud_build_hook*), 574

TIME_TO_SLEEP_IN_SECONDS (*in module airflow.contrib.hooks.gcp_compute_hook*), 575

TIME_TO_SLEEP_IN_SECONDS (*in module airflow.contrib.hooks.gcp_function_hook*), 581

TIME_TO_SLEEP_IN_SECONDS (*in module airflow.contrib.hooks.gcp_sql_hook*), 593

TIME_TO_SLEEP_IN_SECONDS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 600

TimeDeltaSensor (*class in airflow.sensors.time_delta_sensor*), 312

TimeSensor (*class in airflow.sensors.time_sensor*), 312

timestamp (*airflow.models.errors.ImportError attribute*), 671

timestamp (*airflow.models.SlaMiss attribute*), 699

timestamp (*airflow.models.slamiss.SlaMiss attribute*), 674

timestamp (*airflow.models.XCom attribute*), 706

timestamp (*airflow.models.xcom.XCom attribute*), 681
timezone (*airflow.models.dag.DagModel attribute*), 667
timezone (*airflow.models.DagModel attribute*), 694
to_csv() (*airflow.hooks.hive_hooks.HiveServer2Hook method*), 526
to_json() (*airflow.models.Pool method*), 699
to_json() (*airflow.models.pool.Pool method*), 673
topological_sort() (*airflow.models.DAG method*), 691
topological_sort() (*airflow.models.dag.DAG method*), 665
TRANSFER_JOB (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
TRANSFER_JOB_FIELD_MASK (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
TRANSFER_JOBS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
TRANSFER_OPERATIONS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
TRANSFER_OPTIONS (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
TRANSFER_SPEC (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601
TransferJobPreprocessor (*class in airflow.contrib.operators.gcp_transfer_operator*), 394
TransferJobValidator (*class in airflow.contrib.operators.gcp_transfer_operator*), 394
transform() (*airflow.contrib.operators.mongo_to_s3.MongoToS3Operator static method*), 448
transient_errors (*airflow.contrib.sensors.ftp_sensor.FTPSensor attribute*), 500
translate() (*airflow.contrib.hooks.gcp_translate_hook.CloudTranslateHook method*), 604
tree_view() (*airflow.models.DAG method*), 692
tree_view() (*airflow.models.dag.DAG method*), 665
trigger_tasks() (*airflow.executors.base_executor.BaseExecutor method*), 645
trigger_tasks() (*airflow.executors.BaseExecutor method*), 651
trigger_tasks() (*airflow.executors.celery_executor.CeleryExecutor method*), 646
TriggerDagRunOperator (*class in airflow.operators.dagrun_operator*), 278
try_number (*airflow.models.TaskInstance attribute*), 700
try_number (*airflow.models.taskinstance.TaskInstance attribute*), 675
try_number (*airflow.models.TaskReschedule attribute*), 705
try_number (*airflow.models.taskreschedule.TaskReschedule attribute*), 679
ts_function() (*in module airflow.contrib.sensors.gcs_sensor*), 502
tunnel() (*airflow.contrib.operators.ssh_operator.SSHOperator method*), 488
type_map (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudSto attribute*), 451
type_map() (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudS class method*), 449
type_map() (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresTo class method*), 455
type_map() (*airflow.contrib.operators.vertica_to_hive.VerticaToHiveTrans class method*), 490
type_map() (*airflow.operators.mssql_to_hive.MsSqlToHiveTransfer class method*), 290
type_map() (*airflow.operators.mysql_to_hive.MySqlToHiveTransfer class method*), 291
type_name (*airflow.operators.papermill_operator.NoteBook attribute*), 292

U

ui_color (*airflow.contrib.operators.adls_list_operator.AzureDataLakeStora attribute*), 314
ui_color (*airflow.contrib.operators.adls_to_gcs.AdlsToGoogleCloudStorage attribute*), 315
ui_color (*airflow.contrib.operators.aws_athena_operator.AWSAthenaOper attribute*), 316
ui_color (*airflow.contrib.operators.aws_sqs_publish_operator.SQSPublishO attribute*), 316
ui_color (*airflow.contrib.operators.awsbatch_operator.AWSBatchOperator attribute*), 317
ui_color (*airflow.contrib.operators.azure_cosmos_operator.AzureCosmosIn attribute*), 320
ui_color (*airflow.contrib.operators.bigquery_get_data.BigQueryGetDataOp attribute*), 323
ui_color (*airflow.contrib.operators.bigquery_operator.BigQueryCreateEmp attribute*), 330
ui_color (*airflow.contrib.operators.bigquery_operator.BigQueryCreateEmp attribute*), 327
ui_color (*airflow.contrib.operators.bigquery_operator.BigQueryCreateExte attribute*), 329
ui_color (*airflow.contrib.operators.bigquery_operator.BigQueryDeleteData attribute*), 330
ui_color (*airflow.contrib.operators.bigquery_operator.BigQueryOperator attribute*), 325
ui_color (*airflow.contrib.operators.bigquery_table_delete_operator.BigQue attribute*), 331
ui_color (*airflow.contrib.operators.bigquery_to_bigquery.BigQueryToBigQ attribute*), 332
ui_color (*airflow.contrib.operators.bigquery_to_gcs.BigQueryToCloudStora attribute*), 333
ui_color (*airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleC attribute*), 334

ui_color (*airflow.contrib.operators.databricks_operator.DatabricksRunNowOperator attribute*), 340
ui_color (*airflow.contrib.operators.databricks_operator.DatabricksSubmitRunOperator attribute*), 338
ui_color (*airflow.contrib.operators.dataflow_operator.DataFlowJavaOperator attribute*), 343
ui_color (*airflow.contrib.operators.dataflow_operator.DataflowTemplateOperator attribute*), 344
ui_color (*airflow.contrib.operators.dataproc_operator.DataProcHadoopOperator attribute*), 354
ui_color (*airflow.contrib.operators.dataproc_operator.DataProcHiveOperator attribute*), 353
ui_color (*airflow.contrib.operators.dataproc_operator.DataProcPigOperator attribute*), 352
ui_color (*airflow.contrib.operators.dataproc_operator.DataProcPySparkOperator attribute*), 355
ui_color (*airflow.contrib.operators.dataproc_operator.DataProcSparkOperator attribute*), 354
ui_color (*airflow.contrib.operators.dataproc_operator.DataProcSparkSqlOperator attribute*), 353
ui_color (*airflow.contrib.operators.dingding_operator.DingdingOperator attribute*), 358
ui_color (*airflow.contrib.operators.ecs_operator.ECSOperator attribute*), 360
ui_color (*airflow.contrib.operators.emr_add_steps_operator.EmrAddStepsOperator attribute*), 361
ui_color (*airflow.contrib.operators.emr_create_job_flow_operator.EmrCreateJobFlowOperator attribute*), 362
ui_color (*airflow.contrib.operators.emr_terminate_job_flow_operator.EmrTerminateJobFlowOperator attribute*), 362
ui_color (*airflow.contrib.operators.gcp_transfer_operator.GoogleCloudStorageToGoogleCloudStorageTransferOperator attribute*), 401
ui_color (*airflow.contrib.operators.gcp_transfer_operator.S3ToGoogleCloudStorageTransferOperator attribute*), 400
ui_color (*airflow.contrib.operators.gcs_download_operator.GoogleCloudStorageDownloadOperator attribute*), 425
ui_color (*airflow.contrib.operators.gcs_list_operator.GoogleCloudStorageListOperator attribute*), 426
ui_color (*airflow.contrib.operators.gcs_operator.GoogleCloudStorageCreateBucketOperator attribute*), 427
ui_color (*airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBigQueryOperator attribute*), 430
ui_color (*airflow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator attribute*), 432
ui_color (*airflow.contrib.operators.gcs_to_s3.GoogleCloudStorageToS3Operator attribute*), 433
ui_color (*airflow.contrib.operators.hipchat_operator.HipChatAPISendRoomNotificationOperator attribute*), 436
ui_color (*airflow.contrib.operators.hive_to_dynamodb.HiveToDynamoDBTransferOperator attribute*), 436
ui_color (*airflow.contrib.operators.jenkins_job_trigger_operator.JenkinsJobTriggerOperator attribute*), 438
ui_color (*airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOperator attribute*), 449
ui_color (*airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStor attribute*), 451
ui_color (*airflow.contrib.operators.oracle_to_azure_data_lake_transfer.Or attribute*), 454
ui_color (*airflow.contrib.operators.oracle_to_oracle_transfer.OracleToOra attribute*), 454
ui_color (*airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGo attribute*), 455
ui_color (*airflow.contrib.operators.pubsub_operator.PubSubPublishOperat attribute*), 459
ui_color (*airflow.contrib.operators.pubsub_operator.PubSubSubscriptionCr attribute*), 458
ui_color (*airflow.contrib.operators.pubsub_operator.PubSubSubscriptionD attribute*), 459
ui_color (*airflow.contrib.operators.pubsub_operator.PubSubTopicCreateOp attribute*), 456
ui_color (*airflow.contrib.operators.pubsub_operator.PubSubTopicDeleteOp attribute*), 458
ui_color (*airflow.contrib.operators.qubole_operator.QuboleOperator attribute*), 464
ui_color (*airflow.contrib.operators.s3_list_operator.S3ListOperator attribute*), 467
ui_color (*airflow.contrib.operators.s3_to_gcs_operator.S3ToGoogleCloudS attribute*), 469
ui_color (*airflow.contrib.operators.sagemaker_base_operator.SageMakerB attribute*), 470
ui_color (*airflow.contrib.operators.segment_track_event_operator.Segment attribute*), 475
ui_color (*airflow.contrib.operators.snowflake_operator.SnowflakeOperator attribute*), 479
ui_color (*airflow.contrib.operators.spark_submit_operator.SparkSubmitOp attribute*), 486
ui_color (*airflow.contrib.operators.sqoop_operator.SqoopOperator attribute*), 488
ui_color (*airflow.contrib.operators.vertica_operator.VerticaOperator attribute*), 489
ui_color (*airflow.contrib.operators.vertica_to_hive.VerticaToHiveTransfer attribute*), 490
ui_color (*airflow.contrib.operators.vertica_to_mysql.VerticaToMySqlTrans attribute*), 490
ui_color (*airflow.contrib.sensors.aws_athena_sensor.AthenaSensor attribute*), 492
ui_color (*airflow.contrib.sensors.aws_glue_catalog_partition_sensor.AwsG attribute*), 493
ui_color (*airflow.contrib.sensors.bigquery_sensor.BigQueryTableSensor attribute*), 496
ui_color (*airflow.contrib.sensors.datadog_sensor.DatadogSensor attribute*), 498
ui_color (*airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor attribute*), 498
ui_color (*airflow.contrib.sensors.file_sensor.FileSensor attribute*), 500
ui_color (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectSen attribute*), 502

ui_color (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectUpdatedSensor attribute*), 502
ui_color (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStoragePrefixSensor attribute*), 503
ui_color (*airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageUploadSessionCompleteSensor attribute*), 504
ui_color (*airflow.contrib.sensors.pubsub_sensor.PubSubPullSensor attribute*), 506
ui_color (*airflow.contrib.sensors.redis_key_sensor.RedisKeySensor attribute*), 508
ui_color (*airflow.contrib.sensors.redis_pub_sub_sensor.RedisPubSubSensor attribute*), 509
ui_color (*airflow.contrib.sensors.sagemaker_base_sensor.SageMakerBaseSensor attribute*), 509
ui_color (*airflow.models.BaseOperator attribute*), 684
ui_color (*airflow.models.baseoperator.BaseOperator attribute*), 656
ui_color (*airflow.operators.bash_operator.BashOperator attribute*), 275
ui_color (*airflow.operators.check_operator.CheckOperator attribute*), 276
ui_color (*airflow.operators.check_operator.IntervalCheckOperator attribute*), 277
ui_color (*airflow.operators.check_operator.ValueCheckOperator attribute*), 277
ui_color (*airflow.operators.dagrun_operator.TriggerDagRunOperator attribute*), 278
ui_color (*airflow.operators.dummy_operator.DummyOperator attribute*), 281
ui_color (*airflow.operators.email_operator.EmailOperator attribute*), 282
ui_color (*airflow.operators.generic_transfer.GenericTransfer attribute*), 283
ui_color (*airflow.operators.hive_operator.HiveOperator attribute*), 283
ui_color (*airflow.operators.hive_stats_operator.HiveStatsCollectionOperator attribute*), 285
ui_color (*airflow.operators.hive_to_mysql.HiveToMySqlTransfer attribute*), 286
ui_color (*airflow.operators.http_operator.SimpleHttpOperator attribute*), 288
ui_color (*airflow.operators.jdbc_operator.JdbcOperator attribute*), 288
ui_color (*airflow.operators.latest_only_operator.LatestOnlyOperator attribute*), 288
ui_color (*airflow.operators.mssql_operator.MsSqlOperator attribute*), 289
ui_color (*airflow.operators.mssql_to_hive.MsSqlToHiveTransfer attribute*), 290
ui_color (*airflow.operators.mysql_operator.MySqlOperator attribute*), 290
ui_color (*airflow.operators.mysql_to_hive.MySqlToHiveTransfer attribute*), 291
ui_color (*airflow.operators.oracle_operator.OracleOperator attribute*), 292
ui_color (*airflow.operators.pig_operator.PigOperator attribute*), 293
ui_color (*airflow.operators.postgres_operator.PostgresOperator attribute*), 294
ui_color (*airflow.operators.presto_to_mysql.PrestoToMySqlTransfer attribute*), 296
ui_color (*airflow.operators.python_operator.PythonOperator attribute*), 296
ui_color (*airflow.operators.redshift_to_s3_operator.RedshiftToS3Transfer attribute*), 299
ui_color (*airflow.operators.s3_file_transform_operator.S3FileTransformO attribute*), 300
ui_color (*airflow.operators.s3_to_hive_operator.S3ToHiveTransfer attribute*), 302
ui_color (*airflow.operators.s3_to_redshift_operator.S3ToRedshiftTransfer attribute*), 303
ui_color (*airflow.operators.slack_operator.SlackAPIPostOperator attribute*), 304
ui_color (*airflow.operators.sqlite_operator.SqliteOperator attribute*), 304
ui_color (*airflow.operators.subdag_operator.SubDagOperator attribute*), 305
ui_color (*airflow.sensors.base_sensor_operator.BaseSensorOperator attribute*), 306
ui_color (*airflow.sensors.external_task_sensor.ExternalTaskSensor attribute*), 307
ui_color (*airflow.sensors.hdfs_sensor.HdfsSensor attribute*), 307
ui_color (*airflow.sensors.hive_partition_sensor.HivePartitionSensor attribute*), 308
ui_color (*airflow.sensors.metastore_partition_sensor.MetastorePartitionSen attribute*), 310
ui_color (*airflow.sensors.named_hive_partition_sensor.NamedHivePartitio attribute*), 310
ui_color (*airflow.sensors.sql_sensor.SqlSensor attribute*), 312
ui_fgcolor (*airflow.contrib.operators.databricks_operator.DatabricksRun attribute*), 340
ui_fgcolor (*airflow.contrib.operators.databricks_operator.DatabricksSubm attribute*), 338
ui_fgcolor (*airflow.contrib.operators.qubole_check_operator.QuboleChec attribute*), 460
ui_fgcolor (*airflow.contrib.operators.qubole_check_operator.QuboleValu attribute*), 461
ui_fgcolor (*airflow.contrib.operators.qubole_operator.QuboleOperator attribute*), 464
ui_fgcolor (*airflow.models.BaseOperator attribute*), 684
ui_fgcolor (*airflow.models.baseoperator.BaseOperator attribute*), 657
ui_fgcolor (*airflow.operators.subdag_operator.SubDagOperator attribute*), 305
UNIX_PATH_MAX (*in module air-*

flow.contrib.hooks.gcp_sql_hook), 593
unixname (*airflow.models.TaskInstance attribute*), 700
unixname (*airflow.models.taskinstance.TaskInstance attribute*), 675
UNKNOWN (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlOperationStatus attribute*), 594
update_cluster() (*airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook static method*), 573
update_database() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 592
update_endpoint() (*airflow.contrib.hooks.sagemaker_hook.SageMakerHook method*), 624
UPDATE_FUNCTION (*airflow.contrib.hooks.openfaas_hook.OpenFaasHook attribute*), 617
update_function() (*airflow.contrib.hooks.gcp_function_hook.GcfHook method*), 582
update_function() (*airflow.contrib.hooks.openfaas_hook.OpenFaasHook method*), 617
update_instance() (*airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook method*), 591
update_many() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
update_one() (*airflow.contrib.hooks.mongo_hook.MongoHook method*), 615
update_product() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606
update_product_set() (*airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook method*), 606
update_state() (*airflow.models.DagRun method*), 697
update_state() (*airflow.models.dagrun.DagRun method*), 671
update_transfer_job() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 602
upload() (*airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook method*), 608
upload_file() (*airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook method*), 544
upload_function (*airflow.contrib.operators.gcp_function_operator.ZipPathPreprocessor attribute*), 377
upload_function_zip() (*airflow.contrib.hooks.gcp_function_hook.GcfHook method*), 582
upsert_document() (*airflow.contrib.hooks.azure_cosmos_hook.AzureCosmosDBHook method*), 543
upstream_list (*airflow.models.BaseOperator attribute*), 685
upstream_list (*airflow.models.baseoperator.BaseOperator attribute*), 657
upstream_task_ids (*airflow.models.BaseOperator attribute*), 685
upstream_task_ids (*airflow.models.baseoperator.BaseOperator attribute*), 657
used_slots() (*airflow.models.Pool method*), 699
used_slots() (*airflow.models.pool.Pool method*), 673
USER_AGENT_HEADER (*in module airflow.contrib.hooks.databricks_hook*), 560

V

val (*airflow.models.Variable attribute*), 705
val (*airflow.models.variable.Variable attribute*), 680
valid_modes (*airflow.sensors.base_sensor_operator.BaseSensorOperator attribute*), 306
validate_body() (*airflow.contrib.operators.gcp_transfer_operator.TransferJobValidator method*), 394
validate_response() (*airflow.contrib.hooks.datadog_hook.DatadogHook method*), 562
validate_socket_path_length() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598
validate_ssl_certs() (*airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook method*), 598
value (*airflow.models.XCom attribute*), 706
value (*airflow.models.xcom.XCom attribute*), 681
ValueCheckOperator (*class in airflow.operators.check_operator*), 276
Variable (*class in airflow.models*), 705
Variable (*class in airflow.models.variable*), 680
verify_integrity() (*airflow.models.DagRun method*), 697
verify_integrity() (*airflow.models.dagrun.DagRun method*), 671
VerticaHook (*class in airflow.contrib.hooks.vertica_hook*), 640
VerticaOperator (*class in airflow.contrib.operators.vertica_operator*), 488
VerticaToHiveTransfer (*class in airflow.contrib.operators.vertica_to_hive*), 489

VerticaToMySqlTransfer (*class in airflow.contrib.operators.vertica_to_mysql*), 490
Volume (*in module airflow.contrib.operators.azure_container_instances_operator*), 318

W

wait() (*airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook method*), 581
wait_for_done() (*airflow.contrib.hooks.gcp_dataflow_hook._Dataflow method*), 580
wait_for_done() (*airflow.contrib.hooks.gcp_dataflow_hook._DataflowJob method*), 580
wait_for_done() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcJob method*), 580
wait_for_done() (*airflow.contrib.hooks.gcp_dataproc_hook._DataProcOperation method*), 581
wait_for_operation() (*airflow.contrib.hooks.gcp_container_hook.GKEClusterHook method*), 578
wait_for_transfer_job() (*airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook method*), 603
WAIT_IN_PROGRESS (*airflow.contrib.hooks.sagemaker_hook.LogState attribute*), 621
WasbBlobSensor (*class in airflow.contrib.sensors.wasb_sensor*), 511
WasbDeleteBlobOperator (*class in airflow.contrib.operators.wasb_delete_blob_operator*), 491
WasbHook (*class in airflow.contrib.hooks.wasb_hook*), 641
WasbPrefixSensor (*class in airflow.contrib.sensors.wasb_sensor*), 512
WebHDFSHook (*class in airflow.hooks.webhdfs_hook*), 534
WebHdfsSensor (*class in airflow.sensors.web_hdfs_sensor*), 313
WILDCARD (*in module airflow.contrib.operators.gcs_to_gcs*), 430
WinRMHook (*class in airflow.contrib.hooks.winrm_hook*), 643
WinRMOperator (*class in airflow.contrib.operators.winrm_operator*), 491
worker_uuid (*airflow.models.kubernetes.KubeWorkerIdentifier attribute*), 672
worker_uuid (*airflow.models.KubeWorkerIdentifier attribute*), 698
write_batch_data() (*airflow.contrib.hooks.aws_dynamodb_hook.AwsDynamoDBHook method*), 537
write_object_to_file() (*airflow.contrib.hooks.salesforce_hook.SalesforceHook method*), 627

X

XCom (*class in airflow.models*), 705
XCom (*class in airflow.models.xcom*), 681
xcom_pull() (*airflow.models.BaseOperator method*), 686
xcom_pull() (*airflow.models.baseoperator.BaseOperator method*), 659
xcom_pull() (*airflow.models.TaskInstance method*), 704
xcom_pull() (*airflow.models.taskinstance.TaskInstance method*), 679
xcom_push() (*airflow.models.BaseOperator method*), 686
xcom_push() (*airflow.models.baseoperator.BaseOperator method*), 659
xcom_push() (*airflow.models.TaskInstance method*), 704
xcom_push() (*airflow.models.taskinstance.TaskInstance method*), 678
XCOM_RETURN_KEY (*in module airflow.models*), 706
XCOM_RETURN_KEY (*in module airflow.models.xcom*), 681
XCOM_RUN_ID_KEY (*in module airflow.contrib.operators.databricks_operator*), 335
XCOM_RUN_PAGE_URL_KEY (*in module airflow.contrib.operators.databricks_operator*), 335

Y

YEAR (*in module airflow.contrib.hooks.gcp_transfer_hook*), 601

Z

ZendeskHook (*class in airflow.hooks.zendesk_hook*), 534
ZipPathPreprocessor (*class in airflow.contrib.operators.gcp_function_operator*), 377