
data-pipelines-cli

GetInData

Apr 14, 2023

CONTENTS:

| | | |
|----------|----------------------------|-----------|
| 1 | Introduction | 3 |
| 2 | Community | 5 |
| | Python Module Index | 45 |
| | Index | 47 |

INTRODUCTION

Data Pipelines CLI, also called **DP tool**, is a command-line tool providing an easy way to build and manage data pipelines based on **dbt** in an environment with **GIT**, **Airflow**, **DataHub**, **VSCode**, etc.

The tool can be used in any environment with access to shell and **Python** installed.

data-pipelines-cli's main task is to cover technical complexities and provides an abstraction over all components that take part in Data Pipelines creation and execution. Thanks to the integration with templating engine it allows Analytics Engineers to create and configure new projects. The tool also simplifies automation as it handles deployments and publications of created transformations.

COMMUNITY

Although the tools were created by [GetInData](#) and used in their project it is open-sourced and everyone is welcome to use and contribute to making it better and more powerful.

2.1 Installation

Use the package manager [pip](#) to install `data-pipelines-cli`:

```
pip install data-pipelines-cli [<flags>]
```

Depending on the systems that you want to integrate with you need to provide different flags in square brackets. You can provide comma separate list of flags, for example:

```
pip install data-pipelines-cli [gcs,git,bigquery]
```

Depending on the data storage you have you can use:

- bigquery
- snowflake
- redshift
- postgres

If you need git integration for loading packages published by other projects or publish them by yourself you will need:

- git

If you want to deploy created artifacts (docker images and DataHub metadata) add the following flags:

- docker
- datahub

These are not usually used by a person user.

If you need Business Intelligence integration you can use following options:

- looker

2.2 Setup an environment

This section is for Data Engineers who will be preparing and administrating the whole environment. It describes steps that should be done to prepare the DP tool to be used in an organization with full potential.

2.2.1 Create Data Pipeline project template

The first thing that you need to do is to create a git repository with a project template used later to create multiple projects. The template should contain the whole directory structure and files used in your projects. Additionally, it should have a connection configuration to all components in your environment, CICD, and all other aspects specific to your company. Here you can find templates examples that you can adjust to your need: <https://github.com/getindata/data-pipelines-template-example> . Based on the template The Data Pipelines CLI will ask a user a series of questions to build the final project.

Thanks to the `copier` you can leverage Jinja template syntax to create easily modifiable configuration templates. Just create a `copier.yml` and configure the template questions (read more at [copier documentation](#)).

2.2.2 Create a template to setup a local environment

Working with Data Pipelines usually requires local variables to be set to run and test avoiding messing in shared environments (DEV, STAGE, PROD). To simplify working environment preparation we also decided to use templates that will ask a series of questions and generate local configuration in a home directory.

It requires a repository with a global configuration template file that you or your organization will be using. The repository should contain `dp.yml.tpl` file looking similar to this:

```
_templates_suffix: ".tpl" _envops:
```

```
  autoescape: false block_end_string: "[%]" block_start_string: "[%]" comment_end_string:
  "#]" comment_start_string: "[#" keep_trailing_newline: true variable_end_string: "]" vari-
  able_start_string: "[["
```

```
templates:
```

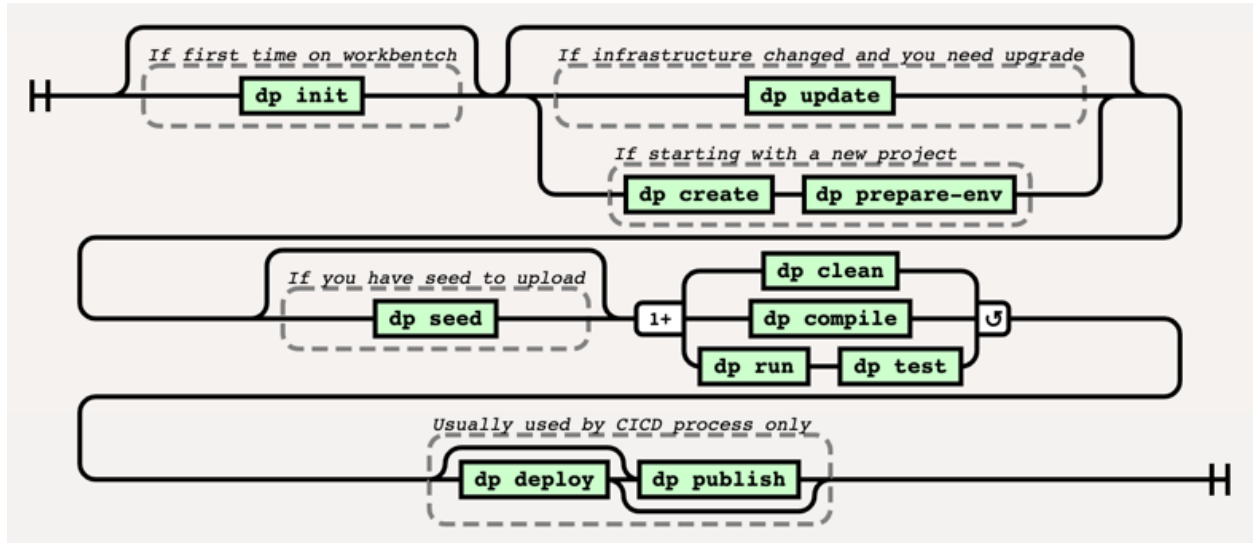
```
  my-first-template: template_name: my-first-template template_path: https://github.
  com/<YOUR_USERNAME>/<YOUR_TEMPLATE>.git
```

```
vars: username: [[ YOUR_USERNAME ]]
```

The file must contain a list of available templates. The templates will be displayed and available for selection in Data Pipelines CLI. The next section contains variables that will be passed to the project whenever running in the configured environment. The same rules apply in template creation as for project templates.

2.3 Usage

This section is for Data Pipelines CLI's users. It will present how to use the tool and how it handles interaction with the whole Data environment. Below diagram presents the sequence process how usually the tool is used and order in which different commands are executed:



2.3.1 Preparing working environment

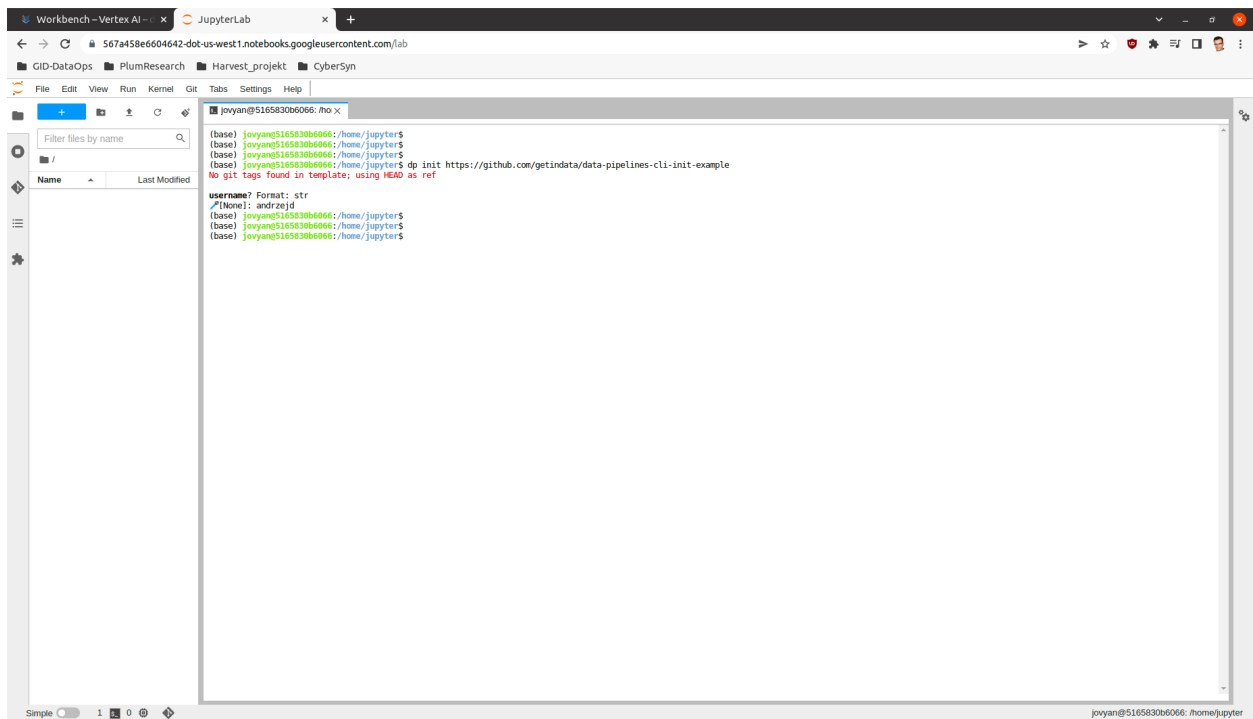
The first thing that needs to be done when starting Building Data Pipelines is to prepare the working environment. This step can be done either on a local machine on any kind of Workbench (eg. JupyterLab). You will need a link from your Data Engineer or Administrator to the template with initial configuration then, run `dp init <CONFIG_REPOSITORY_URL>` to initialize **dp**. You can also drop `<CONFIG_REPOSITORY_URL>` argument, **dp** will get initialized with an empty config.

This step is done only the first time for each working environment you want to use.

Example:

In this example only one variable you will be asked for and it is going to be *username* which is used in many **dp** commands.

```
dp init https://github.com/getindata/data-pipelines-cli-init-example
```



2.3.2 Project creation

You can use `dp create <NEW_PROJECT_PATH>` to choose one of the templates to create the project in the `<NEW_PROJECT_PATH>` directory.

You can also use `dp create <NEW_PROJECT_PATH> <LINK_TO_TEMPLATE_REPOSITORY>` to point directly to a template repository. If `<LINK_TO_TEMPLATE_REPOSITORY>` proves to be the name of the template defined in `dp`'s config file, `dp create` will choose the template by the name instead of trying to download the repository.

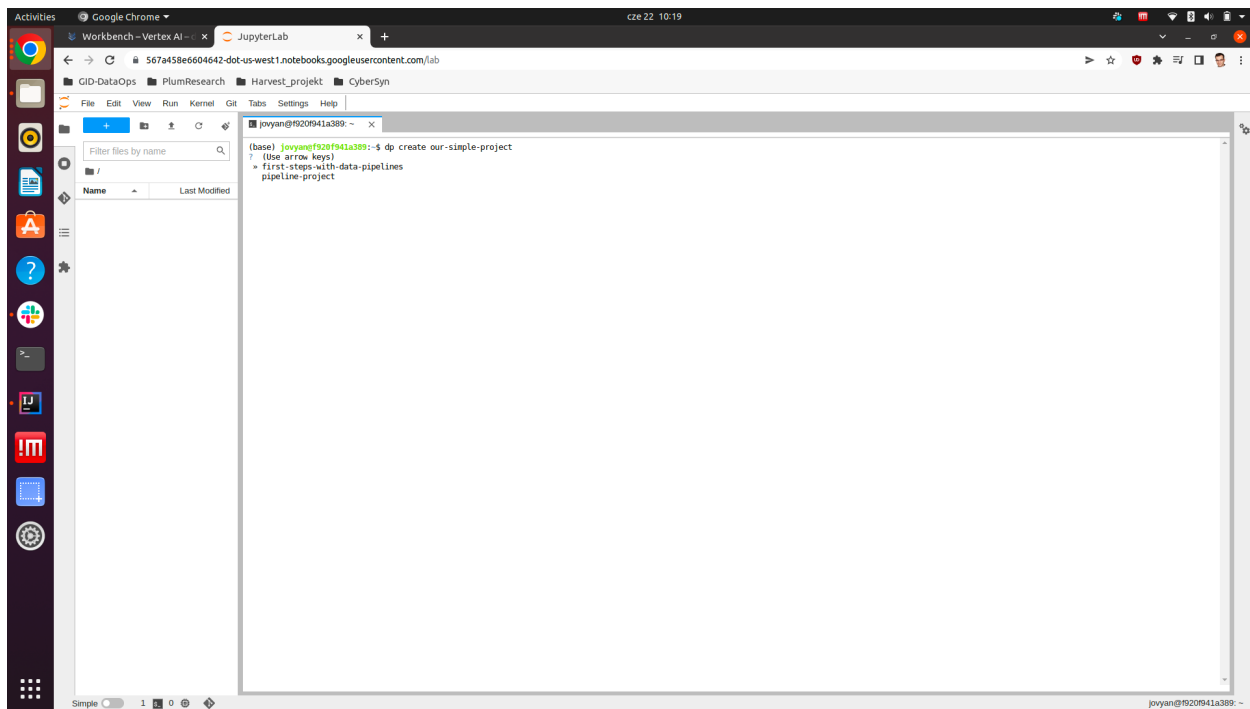
After the template selection, you will be asked a series of predefined questions in the template. Answering them all will cause a new empty project to be generated. The project will be adjusted and personalized based on answers to the questions.

Example:

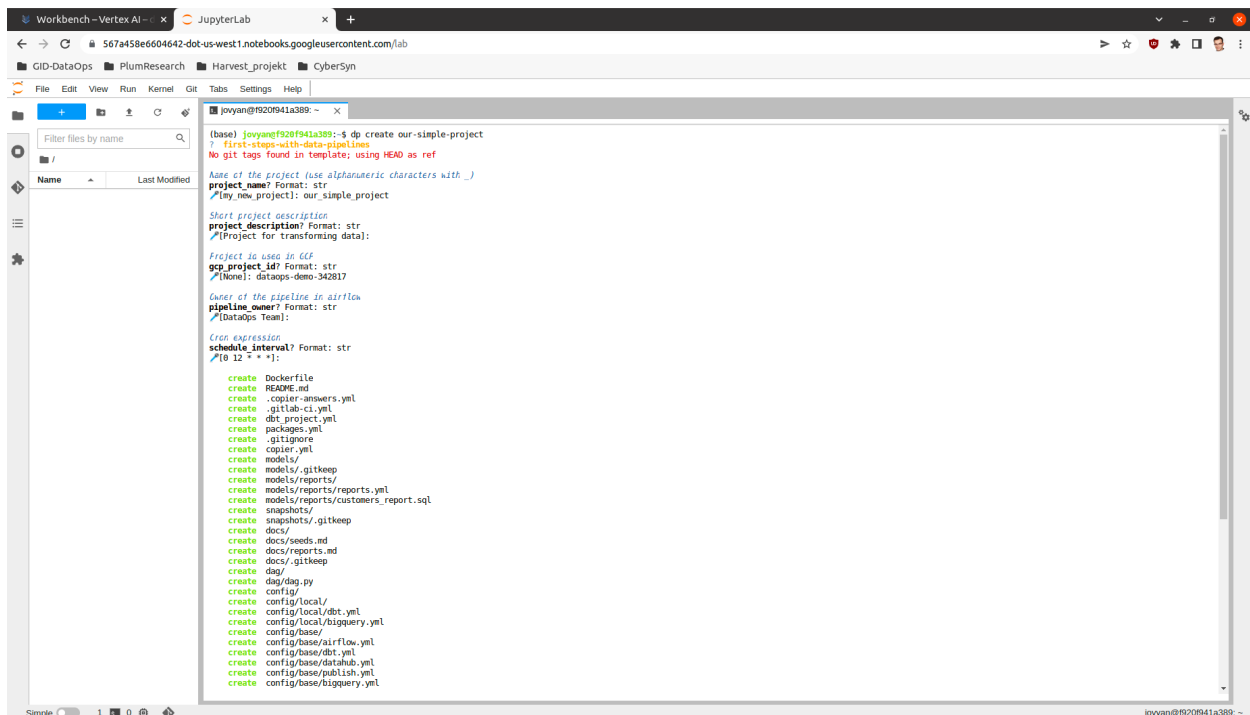
Following command starts project creation process.

```
dp create our-simple-project
```

Fist step after this command is template selection:



We can switch options by pressing up and down buttons and we can make a decision by pressing enter. After that, series of questions will be asked. Be aware that the name of the DP project should be composed of alpha-numeric signs and the _ sign. After answering these questions the tool will generate complete project.



2.3.3 Adapting working environment to VSCode

VSCode is recommended tool to work with **dbt** as you can add a plugin that makes the work more efficient. To configure the plugin or integrate it with some other standalone application you will need to generate `profiles.yml` file from the project. `dp prepare-env` prepares your local environment to be more conformant with standalone **dbt** requirements by saving `profiles.yml` in the home directory.

However, be aware that IDE usage is optional, and you can comfortably use `dp run` and `dp test` commands to interface with the **dbt** instead.

2.3.4 List all available templates

Execute `dp template-list` to list all added templates.

2.3.5 Project update

Whenever the template change you can update your project using `dp update <PIPELINE_PROJECT-PATH>` command. It will sync your existing project with the updated template version selected by `--vcs-ref` option (default HEAD).

It may be very useful when there are some infrastructure changes in your organization and you need to upgrade all created projects (there can be hundreds of them).

2.3.6 Project compilation

`dp compile` prepares your project to be run on your local machine and/or deployed on a remote one.

2.3.7 Local run

When you get your project created, you can run `dp run` and `dp test` commands.

- `dp run` runs the project on your local machine,
- `dp test` runs tests for your project on your local machine.

Both commands accept `--env` parameter to select the execution environment. The default value is `local`.

Example:

```
dp run
```

This process will look at the contents of the `models` directory and create corresponding tables or views in data storage.

```

08:40:02 Completed successfully
08:40:02 Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
(base) joyan@f920f941a389:~/our-simple-projects
(base) joyan@f920f941a389:~/our-simple-projects
(base) joyan@f920f941a389:~/our-simple-projects dp run
Copying 'config' directory to /home/joyan/our-simple-project/build/dag/config
Missing config file: /home/joyan/our-simple-project/build/dag/config/local/execution.env.yml
Missing config file: /home/joyan/our-simple-project/build/dag/config/local/execution.env.yml
The tool has run across a following error when trying to get git revision hash. Ensure your project is a Git repository (run 'git init', if not).
fatal: not a git repository (or any of the parent directories): .git

Could not get git revision hash.
Replacing <PAGE_TAG> with image tag = None
Replacing Jinja variables in /home/joyan/our-simple-project/build/dag/config/base/databus.yml.
Generating profiles.yml in /home/joyan/our-simple-project/build/profiles/local
Running dbt commands:
dbt deps
08:40:12 Running with dbt=1.0.4
08:40:13 Installing dbt-labs/dbt_utils
08:40:14 Installed from version 0.8.0
08:40:14 Updated version available: 0.8.6
08:40:14 Installing calogica/dbt_expectations
08:40:14 Installed from version 0.5.8
08:40:14 Up to date!
08:40:14 Installing dbt-labs/codegen
08:40:14 Installed from version 0.6.0
08:40:14 Up to date!
08:40:14 Installing calogica/dbt_date
08:40:14 Installed from version 0.5.7
08:40:14 Up to date!
08:40:14 Updates available for packages: ['dbt-labs/dbt_utils']
Update your versions in packages.yml, then run dbt deps
dbt compile
08:40:18 Running with dbt=1.0.4
08:40:18 Found 1 model, 1 test, 0 snapshots, 0 analyses, 583 macros, 0 operations, 2 seed files, 0 sources, 0 exposures, 0 metrics
08:40:18 Concurrency: 1 threads (target='local')
08:40:18 Done.
dbt docs generate
08:40:22 Running with dbt=1.0.4
08:40:22 Found 1 model, 1 test, 0 snapshots, 0 analyses, 583 macros, 0 operations, 2 seed files, 0 sources, 0 exposures, 0 metrics
08:40:22 Concurrency: 1 threads (target='local')
08:40:22 Done.
08:40:22 Building catalog
08:40:25 Catalog written to /home/joyan/our-simple-project/target/catalog.json
dbt source freshness
08:40:29 Running with dbt=1.0.4
08:40:29 Found 1 model, 1 test, 0 snapshots, 0 analyses, 583 macros, 0 operations, 2 seed files, 0 sources, 0 exposures, 0 metrics

```

Now after all the tables and views are created we can also check, if the models work as intended by running the tests.

dp test

```

08:41:40 Running with dbt=1.0.4
08:41:41 Installing dbt-labs/dbt_utils
08:41:42 Installed from version 0.8.0
08:41:42 Updated version available: 0.8.6
08:41:42 Installing calogica/dbt_expectations
08:41:42 Installed from version 0.5.8
08:41:42 Up to date!
08:41:42 Installing dbt-labs/codegen
08:41:42 Installed from version 0.6.0
08:41:42 Up to date!
08:41:42 Installing calogica/dbt_date
08:41:42 Installed from version 0.5.7
08:41:42 Up to date!
08:41:42 Updates available for packages: ['dbt-labs/dbt_utils']
Update your versions in packages.yml, then run dbt deps
dbt compile
08:41:45 Running with dbt=1.0.4
08:41:46 Found 1 model, 1 test, 0 snapshots, 0 analyses, 583 macros, 0 operations, 2 seed files, 0 sources, 0 exposures, 0 metrics
08:41:46 Concurrency: 1 threads (target='local')
08:41:46 Done.
dbt docs generate
08:41:50 Running with dbt=1.0.4
08:41:50 Found 1 model, 1 test, 0 snapshots, 0 analyses, 583 macros, 0 operations, 2 seed files, 0 sources, 0 exposures, 0 metrics
08:41:50 Concurrency: 1 threads (target='local')
08:41:50 Done.
08:41:50 Building catalog
08:41:53 Catalog written to /home/joyan/our-simple-project/target/catalog.json
dbt source freshness
08:41:56 Running with dbt=1.0.4
08:41:56 Found 1 model, 1 test, 0 snapshots, 0 analyses, 583 macros, 0 operations, 2 seed files, 0 sources, 0 exposures, 0 metrics
08:41:56 [WARNING]: Nothing to do. Try checking your model configs and model specification args
08:41:56 Done.
Copying dbt manifest
dbt test
08:42:00 Running with dbt=1.0.4
08:42:00 Found 1 model, 1 test, 0 snapshots, 0 analyses, 583 macros, 0 operations, 2 seed files, 0 sources, 0 exposures, 0 metrics
08:42:00 Concurrency: 1 threads (target='local')
08:42:00
08:42:00 1 of 1 START test assert_active_customers_in_every_country..... [RUN]
08:42:02 1 of 1 PASS assert_active_customers_in_every_country..... [PASS in 1.41s]
08:42:02 Finished running 1 test in 1.69s.
08:42:02 Completed successfully
08:42:02 Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
(base) joyan@f920f941a389:~/our-simple-projects

```

2.3.8 dbt sources and automatic models creation

With the help of `dbt-codegen` and `dbt-profiler`, one can easily generate `source.yml`, source's base model SQLs, and model-related YAMLs. `dp` offers a convenient CLI wrapper around those functionalities.

First, add the **dbt-codegen** package to your `packages.yml` file:

```
packages:
- package: dbt-codegen
  version: 0.5.0 # or newer
```

Then, run `dp generate source-yaml YOUR_DATASET_NAME` to generate `source.yml` file in `models/source` directory. You can list more than one dataset, divided by space. After that, you are free to modify this file.

When you want to generate SQLs for your sources, run `dp generate source-sql`. It will save those SQLs in the directory `models/staging/YOUR_DATASET_NAME`.

Finally, when you have all your models prepared (in the form of SQLs), run `dp generate model-yaml MODELS_DIR` to generate YAML files describing them (once again, you are not only free to modify them but also encouraged to do so!). E.g., given such a directory structure:

```
models
├── staging
│   ├── my_source
│   │   ├── stg_table1.sql
│   │   └── stg_table2.sql
│   └── intermediate
│       ├── intermediate1.sql
│       ├── intermediate2.sql
│       └── intermediate3.sql
└── presentation
    └── presentation1.sql
```

`dp generate model-yaml models/` will create `models/staging/my_source/my_source.yml`, `models/staging/intermediate/intermediate.yml`, and `models/presentation/presentation.yml`. Beware, however, this command WILL NOT WORK if you do not have those models created in your data warehouse already. So remember to run `dp run` (or a similar command) beforehand.

If you add the **dbt-profiler** package to your `packages.yml` file too, you can call `dp generate model-yaml --with-meta MODELS_DIR`. **dbt-profiler** will add a lot of profiling metadata to descriptions of your models.

2.3.9 Project deployment

`dp deploy` executes the deployment of a project. Depending on the configuration the command may execute different steps described in this section. Please be aware that this command is meant for the CI/CD process and usually should be avoided as manual activity.

Blob storage synchronization

The main action of the `dp deploy` command is synchronization with your bucket provider. The provider will be chosen automatically based on the remote URL. Usually, it is worth pointing `dp deploy` to a JSON or YAML file with provider-specific data like access tokens or project names. The *provider-specific data* should be interpreted as the `**kwargs` (keyword arguments) expected by a specific `fsspec`'s `FileSystem` implementation. One would most likely want to look at the `S3FileSystem` or `GCSFileSystem` documentation.

E.g., to connect with Google Cloud Storage, one should run:

```
echo '{"token": "<PATH_TO_YOUR_TOKEN>", "project_name": "<YOUR_PROJECT_NAME>"}' > gs_
↪args.json
dp deploy --dags-path "gs://<YOUR_GS_PATH>" --blob-args gs_args.json
```

However, in some cases, you do not need to do so, e.g. when using **gcloud** with properly set local credentials. In such a case, you can try to run just the `dp deploy --dags-path "gs://<YOUR_GS_PATH>"` command and let `gcsfs` search for the credentials. Please refer to the documentation of the specific `fsspec`'s implementation for more information about the required keyword arguments.

You can also provide your path in the `config/base/airflow.yml` file, as a `dags_path` argument:

```
dags_path: gs://<YOUR_GS_PATH>
# ... rest of the 'airflow.yml' file
```

In such a case, you do not have to provide a `--dags-path` flag, and you can just call `dp deploy` instead.

Docker image

`dp deploy` command builds Docker image with **dbt** and project and sends it to Docker Registry. Docker registry may be configured via Environment Variables (eg. `DOCKER_AUTH_CONFIG`) and the image repository can be configured in `execution_env.yml` file. Use `--docker-push` flag to enable docker pushing during deployment.

DataHub synchronization

The deployment also sends metadata to DataHub based on receipt created in `datahub.yml` file. Use `--datahub-ingest` flag to enable DataHub synchronization.

2.3.10 Packing and publishing

Sometimes there is a need to reuse data created in other projects and/or by a different team. The built project can be converted to a **dbt** package by calling `dp publish`. `dp publish` parses `manifest.json` and prepares a package from the presentation layer. It lists models created by transformations and they usually are a final product of a project. The models are prepared in form of **dbt** sources. Created metadata files are saved in the `build/package` directory and sent to a git repository configured in `publish.yml` file.

Publication repo usually is private for a company and appropriate permissions are required. We recommend key-based communication. You can use `--key-path` as a parameter to point to the key file with push permissions.

Using published sources

Published packages can be used as standard **dbt** packages by adding them in `packages.yml` in the following form:

```
packages:
- git: "https://{{env_var('DBT_GIT_USER_NAME', '')}}:{{env_var('DBT_GIT_SECRET_TOKEN', ' '
  ↳')}}@gitlab.com/<path to you repository>"
  subdirectory: "<upstream project name>"
```

Dependencies metadata

Created metadata files containing extra information about the project name (which can be also Airflow DAG name).

```
"source_meta": {
  "dag": "<project name>"
}
```

This way explicit dependencies can be created in the execution environment. For more information see the documentation of *dbt-airflow-factory* <<https://dbt-airflow-factory.readthedocs.io/en/latest/features.html#source-dependencies>>

2.3.11 Clean project

If needed call `dp clean` to remove compilation-related directories.

2.3.12 Load seed

One can use `dp seed` to load seeds from the project. Use `--env` to choose a different environment.

2.3.13 Serve documentation

dbt creates quite good documentation and sometimes it is useful to expose them to your coworkers on a custom port. To do that you can run `dbt docs --port <port>` command.

2.4 Project configuration

dp as a tool depends on a few files in your project directory. It must be able to find a `config` directory with a structure looking similar to this:

```
config
├── base
│   ├── dbt.yml
│   ├── bigquery.yml
│   └── ...
├── dev
│   └── bigquery.yml
├── local
│   ├── dbt.yml
│   └── bigquery.yml
└── prod
    └── bigquery.yml
```

Whenever you call **dp**'s command with the `--env <ENV>` flag, the tool will search for `dbt.yml` and `<TARGET_TYPE>.yml` files in `base` and `<ENV>` directory and parse important info out of them, with `<ENV>` settings taking precedence over those listed in `base`. So, for example, for the following files:

```
# config/base/dbt.yml
target: env_execution
target_type: bigquery

# config/base/bigquery.yml
method: oauth
project: my-gcp-project
dataset: my-dataset
threads: 1

# cat config/dev/bigquery.yml
dataset: dev-dataset
```

`dp test --env dev` will run `dp test` command using values from those files, most notably with `dataset: dev-dataset` overwriting `dataset: my-dataset` setting.

dp synthesizes `dbt's profiles.yml` out of those settings among other things. However, right now it only creates `local` or `env_execution` profile, so if you want to use different settings amongst different environments, you should rather use `{{ env_var('VARIABLE') }}` as a value and provide those settings as environment variables. E.g., by setting those in your `config/<ENV>/k8s.yml` file, in `envs` dictionary:

```
# config/base/bigquery.yml
method: oauth
dataset: "{{ env_var('GCP_DATASET') }}"
project: my-gcp-project
threads: 1
```

(continues on next page)

(continued from previous page)

```
# config/base/execution_env.yml
# ... General config for execution env ...

# config/base/k8s.yml
# ... Kubernetes settings ...

# config/dev/k8s.yml
envs:
  GCP_DATASET: dev-dataset

# config/prod/k8s.yml
envs:
  GCP_DATASET: prod-dataset
```

2.4.1 dbt configuration

The main configuration is in `config/<ENV>/dbt.yml` file. At the moment it allows setting two values: * `target` - should be set either to `local` or `env_execution` depending on where the tool is used. Local means running locally while `env_execution` means executing by the scheduler on the dev or prod environment. * `target_type` - defines which backend **dbt** will use and what file **dp** will search for additional configuration (example: `bigquery` or `snowflake`).

Additionally, the backend configuration file should be provided with a name depending on the selected `target_type` (`<target_type>.yml`). For example setting `target_type` to `bigquery` **dp** will look for `bigquery.yml` files. This file should consist of all configurations that will be used to build `profile.yml`. Example files for the production environment:

```
method: service-account
keyfile: "{{ env_var('GCP_KEY_PATH') }}"
project: gid-dataops-labs
dataset: presentation
threads: 1
timeout_seconds: 300
priority: interactive
location: europe-central2
retries: 1
```

Variables

You can put a dictionary of variables to be passed to **dbt** in your `config/<ENV>/dbt.yml` file, following the convention presented in [the guide at the dbt site](#). E.g., if one of the fields of `config/<SNOWFLAKE_ENV>/snowflake.yml` looks like this:

```
schema: "{{ var('snowflake_schema') }}"
```

you should put the following in your `config/<SNOWFLAKE_ENV>/dbt.yml` file:

```
vars:
  snowflake_schema: EXAMPLE_SCHEMA
```

and then run your `dp run --env <SNOWFLAKE_ENV>` (or any similar command).

You can also add “global” variables to your **dp** config file `$HOME/.dp.yml`. Be aware, however, that those variables get erased on every `dp init` call. It is a great idea to put *commonly used* variables in your organization’s `dp.yml.jinja` template and make **copier** ask for those when initializing **dp**. By doing so, each member of your organization will end up with a list of user-specific variables reusable across different projects on its machine. Just remember, **global-scoped variables take precedence over project-scoped ones**.

2.4.2 Airflow configuration

Airflow-related configuration is stored in `config/<ENV>/airflow.yml` file and is strongly connected to the Airflow plugin: `dbt-airflow-factory`. More information about this configuration can be found [here](#)

One important config from **dp** tool in this file is `days_path`. It sets the URL to blob storage that is responsible for storing projects DAGs with other artifacts.

2.4.3 Execution environment configuration

All configuration about how **dbt** is executed on the Airflow side is kept in `execution_env.yml` and `<env type>.yml`. More information about these settings can be found [here](#)

2.4.4 Publication configuration

`config/<ENV>/publish.yml` file contains configuration about creating **dbt** packages for downstream projects and publishing it to a git repository as a package registry.

| Parameter | Data type | Description |
|------------|-----------|---|
| repository | string | HTTPS link to repo that works as packages repository. |
| branch | string | Branch of the selected repository where packages are published. |
| username | string | User name that will be presented as package publisher in GIT. |
| email | string | Email of the package publisher. |

2.4.5 Data governance configuration

dp can send **dbt** metadata to DataHub. All related configuration is stored in `config/<ENV>/datahub.yml` file. More information about it can be found [here](#) and [here](#).

2.4.6 Data ingestion configuration

Ingestion configuration is divided into two levels:

- General: `config/<ENV>/ingestion.yml`
- Ingestion tool related: e.g. `config/<ENV>/airbyte.yml`

`config/<ENV>/ingestion.yml` contains basic configuration of ingestion:

| Parameter | Data type | Description |
|-----------|-------------|--|
| enable | bool | Flag for enable/disable ingestion option in dp . |
| engine | enum string | Ingestion tool you would like to integrate with (currently the only supported value is <code>airbyte</code>). |

config/<ENV>/airbyte.yml must be present if engine of your choice is airbyte. It consists of two parts:

1. First part is required by [dbt-airflow-factory](#) and must be present in order to create ingestion tasks preceding dbt rebuild in Airflow. When you choose to manage Airbyte connections with *dp* tool, `connectionId` is unknown at the time of coding however *dp* tool is ready to handle this case. For detailed info reference example `airbyte.yml` at the end of this section.

| Parameter | Data type | Description |
|-----------------------|-------------|---|
| airbyte_connection_id | string | Name of Airbyte connection in Airflow |
| tasks | array<task> | Configurations of Airflow tasks used by dbt-airflow-factory . Allowed <i>task</i> options are documented here . |

2. Second part is used directly by *dp* tool to manage (insert or update) connections in Airbyte. It is **not** required unless you would like to manage Airbyte connections with *dp* tool.

| Parameter | Data type | Description |
|-------------|-------------------|--|
| airbyte_url | string | Https address of Airbyte deployment that allows to connect to Airbyte API |
| connections | array<connection> | Configurations of Airbyte connections that should be upserted during CI/CD. Minimal connection schema is documented below. These configurations are passed directly to Airbyte API to the <code>connections/create</code> or <code>connections/update</code> endpoint. Please reference Airbyte API reference for more detailed configuration. |

| | |
|---|---|
| YOUR_CONNECTION_NAME: string | |
| name: string | Optional name of the connection |
| sourceId: uuid | UUID of Airbyte source used for this. |
| ↪connection | |
| destinationId: uuid | UUID of Airbyte destination used for this. |
| ↪connection | |
| namespaceDefinition: enum | Method used for computing final namespace in. |
| ↪destination | |
| namespaceFormat: string | Used when namespaceDefinition is |
| ↪'customformat' | |
| status: enum | `active` means that data is flowing through. |
| ↪the connection. `inactive` means it is not | |
| syncCatalog: object | Describes the available schema (catalog). |
| streams: array | |
| - stream: object | |
| name: string | Stream's name |
| jsonSchema: object | Stream schema using Json Schema specs. |
| config: | |
| syncMode: enum | Allowed: full_refresh incremental |
| destinationSyncMode: enum | Allowed: append overwrite append_dedup |
| aliasName: string | Alias name to the stream to be used in the. |
| ↪destination | |

Example `airbyte.yml` might look like the following. Notice (highlighted lines) how connection name in `connections` array has the same name as the environmental variable in `task[0].connection_id` attribute. During CI/CD, after the connection creation in Airbyte, variable `${POSTGRES_BQ_CONNECTION}` is substituted by the received Airbyte connection UUID and passed in config to `dbt-airflow-factory` tool.

```

1 # dbt-airflow-factory configuration properties:
2 airbyte_connection_id: airbyte_connection_id
3 tasks:
4   - api_version: v1
5     asynchronous: false
6     connection_id: ${POSTGRES_BQ_CONNECTION}
7     task_id: postgres_bq_connection_sync_task
8     timeout: 600
9     wait_seconds: 3
10 # Airbyte connection managing properties:
11 airbyte_url: https://airbyte-dev.company.com
12 connections:
13   POSTGRES_BQ_CONNECTION:
14     name: postgres_bq_connection
15     sourceId: c3aa49f0-90dd-4c8e-9641-505a2f6cb65c
16     destinationId: 3f47dbf1-11f3-41b0-945f-9463c82f711b
17     namespaceDefinition: customformat
18     namespaceFormat: ingestion_pg
19     status: active
20     syncCatalog:
21       streams:
22         - stream:
23             name: raw_orders
24             jsonSchema:
25               properties:
26                 id:
27                   airbyte_type: integer
28                   type: number
29                 order_date:
30                   format: date
31                   type: string
32                 status:
33                   type: string
34                 user_id:
35                   airbyte_type: integer
36                   type: number
37             type: object
38       config:
39         syncMode: full_refresh
40         destinationSyncMode: append
41         aliasName: raw_orders

```

2.4.7 Business Intelligence configuration

BI configuration is divided into two levels:

- General: config/<ENV>/bi.yml file
- BI tool related: e.g. config/<ENV>/looker.yml

config/<ENV>/bi.yml contains basic configuration about BI integration:

| Parameter | Data type | Description |
|---------------|-----------|--|
| is_bi_enabled | bool | Flag for enable/disable BI option in dp . |
| bi_target | string | BI tool you want to working with (currently only Looker is supported). |
| is_bi_compile | bool | Whether generate BI code in compile phase? |
| is_bi_deploy | bool | Whether deploy and push BI codes? |

config/<ENV>/looker.yml contains more detailed configuration related to BI tool:

| Parameter | Data type | Description |
|----------------------------|-----------|--|
| looker_repository | string | Git repository used by Looker project you want to integrate. |
| looker_repository_username | string | Git config - username for operating with repository |
| looker_repository_email | string | Git config - user email for operating with repository |
| looker_project_id | string | Looker's project ID |
| looker_webhook_secret | string | Looker's project webhook secret for deployment |
| looker_repository_branch | string | Looker's repository branch for deploy new codes |
| looker_instance_url | string | URL for you Looker instance |

Example looker.yml file might look like this:

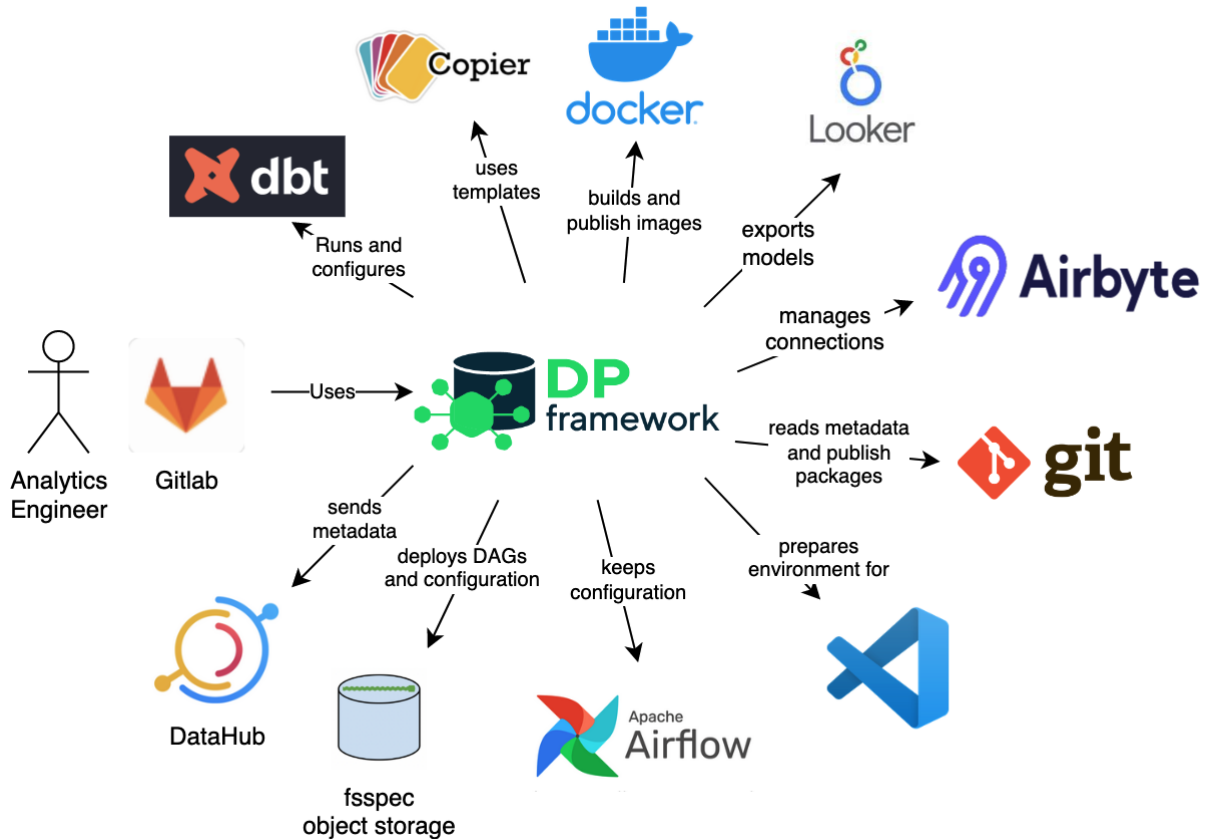
```

1 looker_repository: git@gitlab.com:company/looker/pipeline-example-looker.git
2 looker_repository_username: "{{ env_var('LOOKER_REPO_USERNAME') }}"
3 looker_repository_email: name-surname@company.com
4 looker_project_id: my_lookers_project
5 looker_webhook_secret: "{{ env_var('LOOKER_WEBHOOK_SECRET') }}"
6 looker_repository_branch: main
7 looker_instance_url: https://looker.company.com/

```

2.5 Integration with environment

Data Pipelines CLI provides some sort of abstraction over multiple other components that take part in Data Pipeline processes. The following picture presents the whole environment which is handled by our tool.



2.5.1 dbt

dbt is currently the main tool that **DP** integrates with. The purpose of the **DP** tool is to cover **dbt** technicalities including configuration and generates it on the fly whenever needed. At the same time, it gives more control over **dbt** process management by chaining commands, interpolating configuration, and providing easy environments portability.

2.5.2 Copier

DP is heavily using **Copier** as templating tool. It gives a possibility to easily create new projects that are configured automatically after a series of questions. It is also used to configure the working environment with required environment variables.

2.5.3 Docker

One of the artifacts during building and publishing Data Pipelines are **Docker's** images. Each created image contains **dbt** with its transformation and scripts to run. Created images are environment agnostic and can be deployed in any external configuration. Images are pushed to the selected Container Registry which configuration should be taken from the environment (there should be a docker client configured).

2.5.4 Git

The *Data Pipelines CLI* can also publish created **dbt** packages for downstream usage into configured [GIT](#) repository. It uses key-based authentication where the key is provided as parameter *-key-path*

2.5.5 Airflow

DP doesn't communicate directly with Airflow, it rather sends artifacts to Object storage managed by Airflow and [dbt-airflow-factory](#) library handles the rest. Created projects keep DAG and configuration required to execute on the Airflow side.

2.5.6 Object storage

Configuration, Airflow DAG, and **dbt** manifest.json file are stored in Object storage for Airflow to be picked up and executed. the **DP** uses [fsspec](#) which gives a good abstraction over different object storage providers. Currently, the tools were tested with GCS and S3.

2.5.7 DataHub

The *Data Pipelines CLI* is able to send data to [DataHub](#) based on a recipe in configuration. The tool uses DataHub CLI under the hood.

2.5.8 Visual Studio Code

[VS Code](#) is one of the recommended by us tools to work with **dbt**. **DP** tool simplify integration of the created project with the *VS Code* plugin for **dbt** management.

2.5.9 Airbyte

Data Pipelines CLI can manage Airbyte connections and execute their syncs in Airflow tasks preceding dbt build.

2.5.10 Looker

dp can generate lookML codes for your models and views, publish and deploy your [Looker](#) project

2.6 CLI Commands Reference

If you are looking for extensive information on a specific CLI command, this part of the documentation is for you.

2.7 API Reference

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

2.7.1 data_pipelines_cli package

data-pipelines-cli (dp) is a CLI tool designed for data platform.

dp helps data analysts to create, maintain and make full use of their data pipelines.

Subpackages

data_pipelines_cli.cli_commands package

Subpackages

data_pipelines_cli.cli_commands.generate package

Submodules

data_pipelines_cli.cli_commands.generate.generate module

data_pipelines_cli.cli_commands.generate.model_yaml module

class MacroArgName(kwargs)**

Bases: dict

arg_name: str

deps_name: str

macro_name: str

generate_model_yamls(env: str, with_meta: bool, overwrite: bool, model_paths: Sequence[pathlib.Path]) → None

data_pipelines_cli.cli_commands.generate.source_sql module

generate_source_sqls(env: str, source_yaml_path: pathlib.Path, staging_path: pathlib.Path, overwrite: bool) → None

data_pipelines_cli.cli_commands.generate.source_yaml module

generate_source_yamls(*env: str, source_path: pathlib.Path, overwrite: bool, schema_names: Sequence[str]*) → None

data_pipelines_cli.cli_commands.generate.utils module

generate_models_or_sources_from_single_table(*env: str, macro_name: str, macro_args: Dict[str, Any], profiles_path: pathlib.Path*) → Dict[str, Any]

get_macro_run_output(*env: str, macro_name: str, macro_args: Dict[str, str], profiles_path: pathlib.Path*) → str

get_output_file_or_warn_if_exists(*directory: pathlib.Path, overwrite: bool, file_extension: str, filename: Optional[str] = None*) → Optional[pathlib.Path]

Submodules**data_pipelines_cli.cli_commands.clean module**

clean() → None

Delete local working directories.

data_pipelines_cli.cli_commands.compile module

compile_project(*env: str, docker_tag: Optional[str] = None, docker_build: bool = False, docker_build_args: Optional[Dict[str, str]] = None*) → None

Create local working directories and build artifacts.

Parameters

- **env** (*str*) – Name of the environment
- **docker_tag** (*Optional[str]*) – Image tag of a Docker image to create
- **docker_build** (*bool*) – Whether to build a Docker image
- **bi_build** – Whether to generate a BI codes

Raises *DataPipelinesError* –

replace_image_settings(*image_tag: str*) → None

data_pipelines_cli.cli_commands.create module**data_pipelines_cli.cli_commands.deploy module**

class DeployCommand(*env: str, docker_push: bool, dags_path: Optional[str], provider_kwargs_dict: Optional[Dict[str, Any]], datahub_ingest: bool, bi_git_key_path: str, auth_token: Optional[str]*)

Bases: object

A class used to push and deploy the project to the remote machine.

auth_token: `Optional[str]`

Authorization OIDC ID token for a service account to communication with Airbyte instance

bi_git_key_path: `str`

blob_address_path: `str`

URI of the cloud storage to send build artifacts to

datahub_ingest: `bool`

Whether to ingest DataHub metadata

deploy() \rightarrow `None`

Push and deploy the project to the remote machine.

Raises

- *[DependencyNotInstalledError](#)* – DataHub or Docker not installed
- *[DataPipelinesError](#)* – Error while pushing Docker image

docker_args: `Optional[data_pipelines_cli.data_structures.DockerArgs]`

Arguments required by the Docker to make a push to the repository. If set to *None*, *[deploy\(\)](#)* will not make a push

env: `str`

provider_kwargs_dict: `Dict[str, Any]`

Dictionary of arguments required by a specific cloud storage provider, e.g. path to a token, username, password, etc.

[data_pipelines_cli.cli_commands.docs module](#)

docs(*env: str, port: int*) \rightarrow `None`

Generate and serve dbt documentation.

Parameters

- **env** (*str*) – Name of the environment
- **port** (*int*) – Port to serve dbt documentation on.

[data_pipelines_cli.cli_commands.init module](#)

[data_pipelines_cli.cli_commands.prepare_env module](#)

prepare_env(*env: str*) \rightarrow `None`

Prepare local environment for use with dbt-related applications.

Prepare local environment for use with applications expecting a “traditional” dbt structure, such as plugins to VS Code. If in doubt, use `dp run` and `dp test` instead.

Parameters **env** (*str*) – Name of the environment

data_pipelines_cli.cli_commands.publish module

create_package() → `pathlib.Path`

Create a dbt package out of the built project.

Raises *DataPipelinesError* – There is no model in ‘manifest.json’ file.

publish_package(*package_path: pathlib.Path, key_path: str, env: str*) → `None`

data_pipelines_cli.cli_commands.run module

run(*env: str*) → `None`

Run the project on the local machine.

Parameters *env* (*str*) – Name of the environment

data_pipelines_cli.cli_commands.seed module

seed(*env: str*) → `None`

Run the project on the local machine.

Parameters *env* (*str*) – Name of the environment

data_pipelines_cli.cli_commands.template module

list_templates() → `None`

Print a list of all templates saved in the config file.

data_pipelines_cli.cli_commands.test module

test(*env: str*) → `None`

Run tests of the project on the local machine.

Parameters *env* (*str*) – Name of the environment

data_pipelines_cli.cli_commands.update module

Submodules

data_pipelines_cli.airbyte_utils module

class *AirbyteFactory*(*airbyte_config_path: pathlib.Path, auth_token: Optional[str]*)

Bases: `object`

A class used to create and update Airbyte connections defined in config yaml file

airbyte_config_path: `pathlib.Path`

Path to config yaml file containing connections definitions

auth_token: `Optional[str]`

Authorization OIDC ID token for a service account to communication with Airbyte instance

create_update_connection(*connection_config: Dict[str, Any]*) → Any

create_update_connections() → None

Create and update Airbyte connections defined in config yaml file

static env_replacer(*config: Dict[str, Any]*) → Dict[str, Any]

static find_config_file(*env: str, config_name: str = 'airbyte'*) → pathlib.Path

request_handler(*endpoint: str, config: Dict[str, Any]*) → Union[Dict[str, Any], Any]

update_file(*updated_config: Dict[str, Any]*) → None

data_pipelines_cli.bi_utils module

class BiAction(*value*)

Bases: `enum.Enum`

An enumeration.

COMPILE = 1

DEPLOY = 2

bi(*env: str, bi_action: data_pipelines_cli.bi_utils.BiAction, key_path: Optional[str] = None*) → None

Generate and deploy BI codes using dbt compiled data.

Parameters

- **env** (*str*) – Name of the environment
- **bi_action** – Action to be run [COMPILE, DEPLOY]
- **key_path** – Path to the key with write access to git repository

Raises [`NotSupportedBIError`](#) – Not supported bi in bi.yml configuration

read_bi_config(*env: str*) → Dict[str, Any]

Read BI configuration.

Parameters **env** (*str*) – Name of the environment

Returns Compiled dictionary

Return type Dict[str, Any]

data_pipelines_cli.cli module

data_pipelines_cli.cli_configs module

find_datahub_config_file(*env: str*) → pathlib.Path

data_pipelines_cli.cli_constants module

DEFAULT_GLOBAL_CONFIG: `data_pipelines_cli.data_structures.DataPipelinesConfig = {'templates': {}, 'vars': {}}`

Content of the config file created by `dp init` command if no template path is provided

IMAGE_TAG_TO_REPLACE: `str = '<IMAGE_TAG>'`

PROFILE_NAME_ENV_EXECUTION = `'env_execution'`

Name of the dbt target to use for a remote machine

PROFILE_NAME_LOCAL_ENVIRONMENT = `'local'`

Name of the environment and dbt target to use for a local machine

get_dbt_profiles_env_name(*env: str*) → *str*

Given a name of the environment, returns one of target names expected by the `profiles.yml` file.

Parameters *env* (*str*) – Name of the environment

Returns Name of the *target* to be used in `profiles.yml`

data_pipelines_cli.cli_utils module

echo_error(*text: str, **kwargs: Any*) → *None*

Print an error message to stderr using click-specific print function.

Parameters

- **text** (*str*) – Message to print
- **kwargs** –

echo_info(*text: str, **kwargs: Any*) → *None*

Print a message to stdout using click-specific print function.

Parameters

- **text** (*str*) – Message to print
- **kwargs** –

echo_suberror(*text: str, **kwargs: Any*) → *None*

Print a suberror message to stderr using click-specific print function.

Parameters

- **text** (*str*) – Message to print
- **kwargs** –

echo_subinfo(*text: str, **kwargs: Any*) → *None*

Print a subinfo message to stdout using click-specific print function.

Parameters

- **text** (*str*) – Message to print
- **kwargs** –

echo_warning(*text: str, **kwargs: Any*) → None

Print a warning message to stderr using click-specific print function.

Parameters

- **text** (*str*) – Message to print
- **kwargs** –

get_argument_or_environment_variable(*argument: Optional[str], argument_name: str, environment_variable_name: str*) → str

Given *argument* is not None, return its value. Otherwise, search for *environment_variable_name* amongst environment variables and return it. If such a variable is not set, raise [DataPipelinesError](#).

Parameters

- **argument** (*Optional[str]*) – Optional value passed to the CLI as the *argument_name*
- **argument_name** (*str*) – Name of the CLI's argument
- **environment_variable_name** (*str*) – Name of the environment variable to search for

Returns Value of the *argument* or specified environment variable

Raises [DataPipelinesError](#) – *argument* is None and *environment_variable_name* is not set

subprocess_run(*args: List[str], capture_output: bool = False*) → subprocess.CompletedProcess[bytes]

Run subprocess and return its state if completed with a success. If not, raise [SubprocessNonZeroExitError](#).

Parameters

- **args** (*List[str]*) – List of strings representing subprocess and its arguments
- **capture_output** (*bool*) – Whether to capture output of subprocess.

Returns State of the completed process

Return type subprocess.CompletedProcess[bytes]

Raises [SubprocessNonZeroExitError](#) – subprocess exited with non-zero exit code

data_pipelines_cli.config_generation module

class DbtProfile(***kwargs*)

Bases: dict

POD representing dbt's *profiles.yml* file.

outputs: Dict[str, Dict[str, Any]]

Dictionary of a warehouse data and credentials, referenced by *target* name

target: str

Name of the *target* for dbt to run

copy_config_dir_to_build_dir() → None

Recursively copy *config* directory to *build/dag/config* working directory.

copy_dag_dir_to_build_dir() → None

Recursively copy *dag* directory to *build/dag* working directory.

generate_profiles_dict(*env*: str, *copy_config_dir*: bool) → Dict[str, *data_pipelines_cli.config_generation.DbtProfile*]

Generate and save `profiles.yml` file at `build/profiles/local` or `build/profiles/env_execution`, depending on *env* argument.

Parameters

- **env** (str) – Name of the environment
- **copy_config_dir** (bool) – Whether to copy config directory to build working directory

Returns Dictionary representing data to be saved in `profiles.yml`

Return type Dict[str, *DbtProfile*]

generate_profiles_yaml(*env*: str, *copy_config_dir*: bool = True) → pathlib.Path

Generate and save `profiles.yml` file at `build/profiles/local` or `build/profiles/env_execution`, depending on *env* argument.

Parameters

- **env** (str) – Name of the environment
- **copy_config_dir** (bool) – Whether to copy config directory to build working directory

Returns Path to `build/profiles/{env}`

Return type pathlib.Path

get_profiles_dir_build_path(*env*: str) → pathlib.Path

Returns path to `build/profiles/<profile_name>/`, depending on *env* argument.

Parameters **env** (str) – Name of the environment

Returns

Return type pathlib.Path

read_dictionary_from_config_directory(*config_path*: Union[str, os.PathLike[str]], *env*: str, *file_name*: str) → Dict[str, Any]

Read dictionaries out of *file_name* in both *base* and *env* directories, and compile them into one. Values from *env* directory get precedence over *base* ones.

Parameters

- **config_path** (Union[str, os.PathLike[str]]) – Path to the *config* directory
- **env** (str) – Name of the environment
- **file_name** (str) – Name of the YAML file to parse dictionary from

Returns Compiled dictionary

Return type Dict[str, Any]

data_pipelines_cli.data_structures module

```
class DataPipelinesConfig(**kwargs)
```

Bases: dict

POD representing *.dp.yml* config file.

templates: Dict[str, [data_pipelines_cli.data_structures.TemplateConfig](#)]

Dictionary of saved templates to use in *dp create* command

vars: Dict[str, str]

Variables to be passed to dbt as *--vars* argument

```
class DbtModel(**kwargs)
```

Bases: dict

POD representing a single model from 'schema.yml' file.

columns: List[[data_pipelines_cli.data_structures.DbtTableColumn](#)]

description: str

identifier: str

meta: Dict[str, Any]

name: str

tags: List[str]

tests: List[str]

```
class DbtSource(**kwargs)
```

Bases: dict

POD representing a single source from 'schema.yml' file.

database: str

description: str

meta: Dict[str, Any]

name: str

schema: str

tables: List[[data_pipelines_cli.data_structures.DbtModel](#)]

tags: List[str]

```
class DbtTableColumn(**kwargs)
```

Bases: dict

POD representing a single column from 'schema.yml' file.

description: str

meta: Dict[str, Any]

name: str

quote: bool

tags: List[str]

tests: List[str]

class DockerArgs(*env: str, image_tag: Optional[str], build_args: Dict[str, str]*)

Bases: object

Arguments required by the Docker to make a push to the repository.

Raises *DataPipelinesError* – repository variable not set or git hash not found

build_args: Dict[str, str]

docker_build_tag() → str

Prepare a tag for Docker Python API build command.

Returns Tag for Docker Python API build command

Return type str

image_tag: str

An image tag

repository: str

URI of the Docker images repository

class TemplateConfig(***kwargs*)

Bases: dict

POD representing value referenced in the *templates* section of the *.dp.yml* config file.

template_name: str

Name of the template

template_path: str

Local path or Git URI to the template repository

read_env_config() → *data_pipelines_cli.data_structures.DataPipelinesConfig*

Parse *.dp.yml* config file, if it exists. Otherwise, raises *NoConfigFileError*.

Returns POD representing *.dp.yml* config file, if it exists

Return type *DataPipelinesConfig*

Raises *NoConfigFileError* – *.dp.yml* file not found

data_pipelines_cli.dbt_utils module

read_dbt_vars_from_configs(*env: str*) → Dict[str, Any]

Read *vars* field from dp configuration file (*\$HOME/.dp.yml*), base dbt.yml config (config/base/dbt.yml) and environment-specific config (config/{env}/dbt.yml) and compile into one dictionary.

Parameters *env* (str) – Name of the environment

Returns Dictionary with *vars* and their keys

Return type Dict[str, Any]

run_dbt_command(*command*: *Tuple[str, ...]*, *env*: *str*, *profiles_path*: *pathlib.Path*, *log_format_json*: *bool* = *False*, *capture_output*: *bool* = *False*) → *subprocess.CompletedProcess[bytes]*

Run dbt subprocess in a context of specified *env*.

Parameters

- **command** (*Tuple[str, ...]*) – Tuple representing dbt command and its optional arguments
- **env** (*str*) – Name of the environment
- **profiles_path** (*pathlib.Path*) – Path to the directory containing *profiles.yml* file
- **log_format_json** (*bool*) – Whether to run dbt command with *-log-format=json* flag
- **capture_output** (*bool*) – Whether to capture stdout of subprocess.

Returns State of the completed process

Return type *subprocess.CompletedProcess[bytes]*

Raises

- ***SubprocessNotFound*** – dbt not installed
- ***SubprocessNonZeroExitError*** – dbt exited with error

data_pipelines_cli.docker_response_reader module

class DockerReadResponse(*msg*: *str*, *is_error*: *bool*)

Bases: object

POD representing Docker response processed by *DockerResponseReader*.

is_error: *bool*

Whether response is error or not

msg: *str*

Read and processed message

class DockerResponseReader(*logs_generator*: *Iterable[Union[str, Dict[str, Union[str, Dict[str, str]]]]]*)

Bases: object

Read and process Docker response.

Docker response turns into processed strings instead of plain dictionaries.

cached_read_response:

Optional[List[data_pipelines_cli.docker_response_reader.DockerReadResponse]]

Internal cache of already processed response

click_echo_ok_responses() → *None*

Read, process and print positive Docker updates.

Raises ***DockerErrorResponseError*** – Came across error update in Docker response.

logs_generator: *Iterable[Union[str, Dict[str, Union[str, Dict[str, str]]]]]*

Iterable representing Docker response

read_response() → List[*data_pipelines_cli.docker_response_reader.DockerReadResponse*]

Read and process Docker response.

Returns List of processed lines of response

Return type List[*DockerReadResponse*]

data_pipelines_cli.errors module

exception AirflowDagsPathKeyError

Bases: *data_pipelines_cli.errors.DataPipelinesError*

Exception raised if there is no dags_path in *airflow.yml* file.

message: str

explanation of the error

submessage: Optional[str]

additional informations for the error

exception DataPipelinesError(message: str, submessage: Optional[str] = None)

Bases: Exception

Base class for all exceptions in data_pipelines_cli module

message: str

explanation of the error

submessage: Optional[str]

additional informations for the error

exception DependencyNotInstalledError(program_name: str)

Bases: *data_pipelines_cli.errors.DataPipelinesError*

Exception raised if certain dependency is not installed

message: str

explanation of the error

submessage: Optional[str]

additional informations for the error

exception DockerErrorResponseError(error_msg: str)

Bases: *data_pipelines_cli.errors.DataPipelinesError*

Exception raised if there is an error response from Docker client.

message: str

explanation of the error

submessage: Optional[str]

additional informations for the error

exception DockerNotInstalledError

Bases: *data_pipelines_cli.errors.DependencyNotInstalledError*

Exception raised if 'docker' is not installed

```

message: str
    explanation of the error
submessage: Optional[str]
    additional informations for the error
exception JinjaVarKeyError(key: str)
    Bases: data_pipelines_cli.errors.DataPipelinesError
message: str
    explanation of the error
submessage: Optional[str]
    additional informations for the error
exception NoConfigFileError
    Bases: data_pipelines_cli.errors.DataPipelinesError
    Exception raised if .dp.yml does not exist
message: str
    explanation of the error
submessage: Optional[str]
    additional informations for the error
exception NotAProjectDirectoryError(project_path: str)
    Bases: data_pipelines_cli.errors.DataPipelinesError
    Exception raised if .copier-answers.yml file does not exist in given dir
message: str
    explanation of the error
submessage: Optional[str]
    additional informations for the error
exception NotSupportedBIError
    Bases: data_pipelines_cli.errors.DataPipelinesError
    Exception raised if there is no target_id in bi.yml
message: str
    explanation of the error
submessage: Optional[str]
    additional informations for the error
exception SubprocessNonZeroExitError(subprocess_name: str, exit_code: int, subprocess_output:
    Optional[str] = None)
    Bases: data_pipelines_cli.errors.DataPipelinesError
    Exception raised if subprocess exits with non-zero exit code
message: str
    explanation of the error
submessage: Optional[str]
    additional informations for the error

```

exception SubprocessNotFound(*subprocess_name: str*)

Bases: `data_pipelines_cli.errors.DataPipelinesError`

Exception raised if subprocess cannot be found

message: `str`

explanation of the error

submessage: `Optional[str]`

additional informations for the error

data_pipelines_cli.filesystem_utils module

class LocalRemoteSync(*local_path: Union[str, os.PathLike[str]], remote_path: str, remote_kwargs: Dict[str, str]*)

Bases: `object`

Synchronizes local directory with a cloud storage's one.

local_fs: `fsspec.spec.AbstractFileSystem`

FS representing local directory

local_path_str: `str`

Path to local directory

remote_path_str: `str`

Path/URI of the cloud storage directory

sync(*delete: bool = True*) \rightarrow `None`

Send local files to the remote directory and (optionally) delete unnecessary ones.

Parameters **delete** (*bool*) – Whether to delete remote files that are no longer present in local directory

data_pipelines_cli.io_utils module

git_revision_hash() \rightarrow `Optional[str]`

Get current Git revision hash, if Git is installed and any revision exists.

Returns Git revision hash, if possible.

Return type `Optional[str]`

replace(*filename: Union[str, os.PathLike[str]], pattern: str, replacement: str*) \rightarrow `None`

Perform the pure-Python equivalent of in-place *sed* substitution: e.g., `sed -i -e 's/{pattern}/{replacement}' '{filename}'`.

Beware however, it uses Python regex dialect instead of *sed*'s one. It can introduce regex-related bugs.

data_pipelines_cli.jinja module

replace_vars_with_values(*templated_dictionary: Dict[str, Any], dbt_vars: Dict[str, Any]*) → Dict[str, Any]

Replace variables in given dictionary using Jinja template in its values.

Parameters

- **templated_dictionary** (*Dict[str, Any]*) – Dictionary with Jinja-templated values
- **dbt_vars** (*Dict[str, Any]*) – Variables to replace

Returns Dictionary with replaced variables

Return type Dict[str, Any]

Raises *JinjaVarKeyError* – Variable referenced in Jinja template does not exist

data_pipelines_cli.looker_utils module

deploy_lookML_model(*key_path: str, env: str*) → None

Write compiled lookML to Looker's repository and deploy project to production

Parameters

- **key_path** (*str*) – Path to the key with write access to git repository
- **env** (*str*) – Name of the environment

generate_lookML_model() → None

Generate lookML codes based on compiled dbt project.

read_lookmer_config(*env: str*) → Dict[str, Any]

Read Looker configuration.

Parameters **env** (*str*) – Name of the environment

Returns Compiled dictionary

Return type Dict[str, Any]

data_pipelines_cli.vcs_utils module

2.8 Changelog

2.8.1 Unreleased

2.8.2 0.24.2 - 2023-04-14

- Added Airbyte integration documentation

2.8.3 0.24.1 - 2023-03-15

Fixed

- `dp` commands failing when BI config was missing.

2.8.4 0.24.0 - 2022-12-16

- Airbyte integration
- `dp deploy` is able to add / update connections on Airbyte instance
- `dp deploy` is able to create DAG at the beginning of dbt builds that will execute ingestion tasks
- `dp deploy` accept additional attribute `auth-token` that can be used to authorize access to cloud services
- Bump packages

2.8.5 0.23.0 - 2022-10-19

2.8.6 0.22.1 - 2022-10-11

- Looker integration
- `dp compile` is able generate lookML project for Looker
- `dp deploy` is able to publish lookML codes in Looker's repo and deploy project.

2.8.7 0.22.0 - 2022-08-22

- `dp compile` default environment has been set to `local`
- GitPython is not required anymore
- Installation documentation upgrade

2.8.8 0.21.0 - 2022-07-19

- Documentation improvements

2.8.9 0.20.1 - 2022-06-17

Fixed

- `dp seed`, `dp run` and `dp test` no longer fail when we are not using git repository.

2.8.10 0.20.0 - 2022-05-04

- `--docker-args` has been added to `dp compile`

2.8.11 0.19.0 - 2022-04-25

Added

- `dp seed` command acting as a wrapper for `dbt seed`.

2.8.12 0.18.0 - 2022-04-19

Added

- `dp docs-serve` command acting as a wrapper for `dbt docs serve`.

2.8.13 0.17.0 - 2022-04-11

Added

- `pip install data-pipelines-cli[ADAPTER_PROVIDER]` installs adapter alongside **dbt-core**, e.g. `pip install data-pipelines-cli[bigquery]`.

Changed

- `dp compile` accepts additional command line argument `--docker-tag`, allowing for custom Docker tag instead of relying on Git commit SHA. Moreover, if `--docker-tag` is not provided, **dp** searches for tag in `build/dag/config/<ENV>/execution_env.yml`. If it is present instead of `<IMAGE_TAG>` to be replaced, **dp** chooses it over Git commit SHA.

2.8.14 0.16.0 - 2022-03-24

Added

- `dp generate source-yaml` and `dp generate model-yaml` commands that automatically generate YAML schema files for project's sources or models, respectively (using `dbt-codegen` or `dbt-profiler` under the hood).
- `dp generate source-sql` command that generates SQL representing sources listed in `source.yml` (or a similar file) (again, with the help of `dbt-codegen`).

2.8.15 0.15.2 - 2022-02-28

Changed

- Bumped dbt to 1.0.3.

2.8.16 0.15.1 - 2022-02-28

Fixed

- Pinned MarkupSafe==2.0.1 to ensure that Jinja works.

2.8.17 0.15.0 - 2022-02-11

- Migration to dbt 1.0.1

2.8.18 0.14.0 - 2022-02-02

2.8.19 0.13.0 - 2022-02-01

2.8.20 0.12.0 - 2022-01-31

- `dp publish` will push generated sources to external git repo

2.8.21 0.11.0 - 2022-01-18

Added

- `dp update` command
- `dp publish` command for creation of dbt package out of the project.

Changed

- Docker response in `deploy` and `compile` gets printed as processed strings instead of plain dictionaries.
- `dp compile` parses content of `datahub.yml` and replaces Jinja variables in the form of `var` or `env_var`.
- `dags_path` is read from an env'd `airflow.yml` file.

2.8.22 0.10.0 - 2022-01-12

Changed

- Run `dbt deps` at the end of `dp prepare-env`.

Fixed

- `dp run` and `dp test` are no longer pointing to `profiles.yml` instead of the directory containing it.

2.8.23 0.9.0 - 2022-01-03

Added

- `--env` flag to `dp deploy`.

Changed

- Docker repository URI gets read out of `build/config/{env}/k8s.yml`.

Removed

- `--docker-repository-uri` and `--datahub-gms-uri` from `dp compile` and `dp deploy` commands.
- `dp compile` no longer replaces `<INGEST_ENDPOINT>` in `datahub.yml`, or `<DOCKER_REPOSITORY_URL>` in `k8s.yml`

2.8.24 0.8.0 - 2021-12-31

Changed

- `dp init` and `dp create` automatically adds `.git` suffix to given template paths, if necessary.
- When reading `dbt` variables, global-scoped variables take precedence over project-scoped ones (it was another way around before).
- Address argument for `dp deploy` is no longer mandatory. It should be either placed in `airflow.yml` file as value of `dags_path` key, or provided with `--dags-path` flag.

2.8.25 0.7.0 - 2021-12-29

Added

- Add documentation in the style of [Read the Docs](#).
- Exception classes in `errors.py`, deriving from `DataPipelinesError` base exception class.
- Unit tests to massively improve code coverage.
- `--version` flag to `dp` command.

- Add `dp prepare-env` command that prepares local environment for standalone **dbt** (right now, it only generates and saves `profiles.yml` in `$HOME/.dbt`).

Changed

- `dp compile`:
 - `--env` option has a default value: `base`,
 - `--datahub` is changed to `--datahub-gms-uri`, `--repository` is changed to `--docker-repository-uri`.
- `dp deploy`'s `--docker-push` is not a flag anymore and requires a Docker repository URI parameter; `--repository` got removed then.
- `dp run` and `dp test` run `dp compile` before actual **dbt** command.
- Functions raise exceptions instead of exiting using `sys.exit(1)`; `cli.cli()` entrypoint is expecting exception and exits only there.
- `dp deploy` raises an exception if there is no Docker image to push or `build/config/dag` directory does not exist.
- Rename `gcp` to `gcs` in requirements (now one should run `pip install data-pipelines-cli[gcs]`).

2.8.26 0.6.0 - 2021-12-16

Modified

- **dp** saves generated `profiles.yml` in either `build/local` or `build/env_execution` directories. **dbt** gets executed with `env_execution` as the target.

2.8.27 0.5.1 - 2021-12-14

Fixed

- `_dbt_compile` is no longer removing replaced `<IMAGE_TAG>`.

2.8.28 0.5.0 - 2021-12-14

Added

- `echo_warning` function prints warning messages in yellow/orange color.

Modified

- Docker image gets built at the end of `compile` command.
- `dbt`-related commands do not fail if no `$HOME/.dp.yml` exists (e.g., `dp run`).

Removed

- Dropped `dbt-airflow-manifest-parser` dependency.

2.8.29 0.4.0 - 2021-12-13

Added

- `dp run` and `dp test` commands.
- `dp clean` command for removing build and target directories.
- File synchronization tests for Google Cloud Storage using `gcp-storage-emulator`.
- Read vars from config files (`$HOME/.dp.yml`, `config/$ENV/dbt.yml`) and pass to `dbt`.

Modified

- `profiles.yml` gets generated and saved in build directory in `dp compile`, instead of relying on a local one in the main project directory.
- `dp dbt <command>` generates `profiles.yml` in build directory by default.
- `dp init` is expecting `config_path` argument to download config template with the help of the copier and save it in `$HOME/.dp.yml`.
- `dp template list` is renamed as `dp template-list`.
- `dp create` allows for providing extra argument called `template-path`, being either name of one of templates defined in `.dp.yml` config file or direct link to Git repository.

Removed

- Support for manually created `profiles.yml` in main project directory.
- `dp template new` command.
- `username` field from `$HOME/.dp.yml` file.

2.8.30 0.3.0 - 2021-12-06

- Run `dbt deps` alongside rest of `dbt` commands in `dp compile`

2.8.31 0.2.0 - 2021-12-03

- Add support for GCP and S3 syncing in `dp deploy`

2.8.32 0.1.2 - 2021-12-02

- Fix: do not use styled `click.secho` for Docker push response, as it may not be a `str`

2.8.33 0.1.1 - 2021-12-01

- Fix Docker SDK for Python's bug related to tagging, which prevented Docker from pushing images.

2.8.34 0.1.0 - 2021-12-01

Added

- Draft of `dp init`, `dp create`, `dp template new`, `dp template list` and `dp dbt`
- Draft of `dp compile` and `dp deploy`

PYTHON MODULE INDEX

d

- `data_pipelines_cli`, 23
- `data_pipelines_cli.airbyte_utils`, 26
- `data_pipelines_cli.bi_utils`, 27
- `data_pipelines_cli.cli_commands`, 23
- `data_pipelines_cli.cli_commands.clean`, 24
- `data_pipelines_cli.cli_commands.compile`, 24
- `data_pipelines_cli.cli_commands.deploy`, 24
- `data_pipelines_cli.cli_commands.docs`, 25
- `data_pipelines_cli.cli_commands.generate`, 23
- `data_pipelines_cli.cli_commands.generate.generate`, 23
- `data_pipelines_cli.cli_commands.generate.model_yaml`, 23
- `data_pipelines_cli.cli_commands.generate.source_sql`, 23
- `data_pipelines_cli.cli_commands.generate.source_yaml`, 24
- `data_pipelines_cli.cli_commands.generate.utils`, 24
- `data_pipelines_cli.cli_commands.prepare_env`, 25
- `data_pipelines_cli.cli_commands.publish`, 26
- `data_pipelines_cli.cli_commands.run`, 26
- `data_pipelines_cli.cli_commands.seed`, 26
- `data_pipelines_cli.cli_commands.template`, 26
- `data_pipelines_cli.cli_commands.test`, 26
- `data_pipelines_cli.cli_configs`, 27
- `data_pipelines_cli.cli_constants`, 28
- `data_pipelines_cli.cli_utils`, 28
- `data_pipelines_cli.config_generation`, 29
- `data_pipelines_cli.data_structures`, 31
- `data_pipelines_cli.dbt_utils`, 32
- `data_pipelines_cli.docker_response_reader`, 33
- `data_pipelines_cli.errors`, 34
- `data_pipelines_cli.filesystem_utils`, 36
- `data_pipelines_cli.io_utils`, 36
- `data_pipelines_cli.jinja`, 37
- `data_pipelines_cli.looker_utils`, 37

A

airbyte_config_path (*AirbyteFactory* attribute), 26
 AirbyteFactory (class in *data_pipelines_cli.airbyte_utils*), 26
 AirflowDagsPathKeyError, 34
 arg_name (*MacroArgName* attribute), 23
 auth_token (*AirbyteFactory* attribute), 26
 auth_token (*DeployCommand* attribute), 24

B

bi() (in module *data_pipelines_cli.bi_utils*), 27
 bi_git_key_path (*DeployCommand* attribute), 25
 BiAction (class in *data_pipelines_cli.bi_utils*), 27
 blob_address_path (*DeployCommand* attribute), 25
 build_args (*DockerArgs* attribute), 32

C

cached_read_response (*DockerResponseReader* attribute), 33
 clean() (in module *data_pipelines_cli.cli_commands.clean*), 24
 click_echo_ok_responses() (*DockerResponseReader* method), 33
 columns (*DbtModel* attribute), 31
 COMPILER (*BiAction* attribute), 27
 compile_project() (in module *data_pipelines_cli.cli_commands.compile*), 24
 copy_config_dir_to_build_dir() (in module *data_pipelines_cli.config_generation*), 29
 copy_dag_dir_to_build_dir() (in module *data_pipelines_cli.config_generation*), 29
 create_package() (in module *data_pipelines_cli.cli_commands.publish*), 26
 create_update_connection() (*AirbyteFactory* method), 27
 create_update_connections() (*AirbyteFactory* method), 27

D

data_pipelines_cli

module, 23
 data_pipelines_cli.airbyte_utils
 module, 26
 data_pipelines_cli.bi_utils
 module, 27
 data_pipelines_cli.cli_commands
 module, 23
 data_pipelines_cli.cli_commands.clean
 module, 24
 data_pipelines_cli.cli_commands.compile
 module, 24
 data_pipelines_cli.cli_commands.deploy
 module, 24
 data_pipelines_cli.cli_commands.docs
 module, 25
 data_pipelines_cli.cli_commands.generate
 module, 23
 data_pipelines_cli.cli_commands.generate.generate
 module, 23
 data_pipelines_cli.cli_commands.generate.model_yaml
 module, 23
 data_pipelines_cli.cli_commands.generate.source_sql
 module, 23
 data_pipelines_cli.cli_commands.generate.source_yaml
 module, 24
 data_pipelines_cli.cli_commands.generate.utils
 module, 24
 data_pipelines_cli.cli_commands.prepare_env
 module, 25
 data_pipelines_cli.cli_commands.publish
 module, 26
 data_pipelines_cli.cli_commands.run
 module, 26
 data_pipelines_cli.cli_commands.seed
 module, 26
 data_pipelines_cli.cli_commands.template
 module, 26
 data_pipelines_cli.cli_commands.test
 module, 26
 data_pipelines_cli.cli_configs
 module, 27
 data_pipelines_cli.cli_constants

module, 28
 data_pipelines_cli.cli_utils
 module, 28
 data_pipelines_cli.config_generation
 module, 29
 data_pipelines_cli.data_structures
 module, 31
 data_pipelines_cli.dbt_utils
 module, 32
 data_pipelines_cli.docker_response_reader
 module, 33
 data_pipelines_cli.errors
 module, 34
 data_pipelines_cli.filesystem_utils
 module, 36
 data_pipelines_cli.io_utils
 module, 36
 data_pipelines_cli.jinja
 module, 37
 data_pipelines_cli.looker_utils
 module, 37
 database (*DbtSource* attribute), 31
 datahub_ingest (*DeployCommand* attribute), 25
 DataPipelinesConfig (class in *data_pipelines_cli.data_structures*), 31
 DataPipelinesError, 34
 DbtModel (class in *data_pipelines_cli.data_structures*), 31
 DbtProfile (class in *data_pipelines_cli.config_generation*), 29
 DbtSource (class in *data_pipelines_cli.data_structures*), 31
 DbtTableColumn (class in *data_pipelines_cli.data_structures*), 31
 DEFAULT_GLOBAL_CONFIG (in *data_pipelines_cli.cli_constants*), 28
 DependencyNotInstalledError, 34
 DEPLOY (*BiAction* attribute), 27
 deploy() (*DeployCommand* method), 25
 deploy_lookML_model() (in *data_pipelines_cli.looker_utils*), 37
 DeployCommand (class in *data_pipelines_cli.cli_commands.deploy*), 24
 deps_name (*MacroArgName* attribute), 23
 description (*DbtModel* attribute), 31
 description (*DbtSource* attribute), 31
 description (*DbtTableColumn* attribute), 31
 docker_args (*DeployCommand* attribute), 25
 docker_build_tag() (*DockerArgs* method), 32
 DockerArgs (class in *data_pipelines_cli.data_structures*), 32
 DockerErrorResponseError, 34
 DockerNotInstalledError, 34
 DockerReadResponse (class in *data_pipelines_cli.docker_response_reader*), 33
 DockerResponseReader (class in *data_pipelines_cli.docker_response_reader*), 33
 docs() (in *data_pipelines_cli.cli_commands.docs*), 25
E
 echo_error() (in *data_pipelines_cli.cli_utils*), 28
 echo_info() (in *data_pipelines_cli.cli_utils*), 28
 echo_suberror() (in *data_pipelines_cli.cli_utils*), 28
 echo_subinfo() (in *data_pipelines_cli.cli_utils*), 28
 echo_warning() (in *data_pipelines_cli.cli_utils*), 28
 env (*DeployCommand* attribute), 25
 env_replacer() (*AirbyteFactory* static method), 27
F
 find_config_file() (*AirbyteFactory* static method), 27
 find_datahub_config_file() (in *data_pipelines_cli.cli_configs*), 27
G
 generate_lookML_model() (in *data_pipelines_cli.looker_utils*), 37
 generate_model_yamls() (in *data_pipelines_cli.cli_commands.generate.model_yaml*), 23
 generate_models_or_sources_from_single_table() (in *data_pipelines_cli.cli_commands.generate.utils*), 24
 generate_profiles_dict() (in *data_pipelines_cli.config_generation*), 29
 generate_profiles_yaml() (in *data_pipelines_cli.config_generation*), 30
 generate_source_sqls() (in *data_pipelines_cli.cli_commands.generate.source_sql*), 23
 generate_source_yamls() (in *data_pipelines_cli.cli_commands.generate.source_yaml*), 24
 get_argument_or_environment_variable() (in *data_pipelines_cli.cli_utils*), 29
 get_dbt_profiles_env_name() (in *data_pipelines_cli.cli_constants*), 28
 get_macro_run_output() (in *data_pipelines_cli.cli_commands.generate.utils*), 24

get_output_file_or_warn_if_exists() (in module *data_pipelines_cli.cli_commands.generate.utils*), 24
 get_profiles_dir_build_path() (in module *data_pipelines_cli.config_generation*), 30
 git_revision_hash() (in module *data_pipelines_cli.io_utils*), 36
 I
 identifier (*DbtModel* attribute), 31
 image_tag (*DockerArgs* attribute), 32
 IMAGE_TAG_TO_REPLACE (in module *data_pipelines_cli.cli_constants*), 28
 is_error (*DockerReadResponse* attribute), 33
 J
 JinjaVarKeyError, 35
 L
 list_templates() (in module *data_pipelines_cli.cli_commands.template*), 26
 local_fs (*LocalRemoteSync* attribute), 36
 local_path_str (*LocalRemoteSync* attribute), 36
 LocalRemoteSync (class in *data_pipelines_cli.filesystem_utils*), 36
 logs_generator (*DockerResponseReader* attribute), 33
 M
 macro_name (*MacroArgName* attribute), 23
 MacroArgName (class in *data_pipelines_cli.cli_commands.generate.model_yaml*), 23
 message (*AirflowDagsPathKeyError* attribute), 34
 message (*DataPipelinesError* attribute), 34
 message (*DependencyNotInstalledError* attribute), 34
 message (*DockerErrorResponse* attribute), 34
 message (*DockerNotInstalledError* attribute), 34
 message (*JinjaVarKeyError* attribute), 35
 message (*NoConfigFileError* attribute), 35
 message (*NotAProjectDirectoryError* attribute), 35
 message (*NotSupportedBIError* attribute), 35
 message (*SubprocessNonZeroExitError* attribute), 35
 message (*SubprocessNotFound* attribute), 36
 meta (*DbtModel* attribute), 31
 meta (*DbtSource* attribute), 31
 meta (*DbtTableColumn* attribute), 31
 module
 data_pipelines_cli, 23
 data_pipelines_cli.airbyte_utils, 26
 data_pipelines_cli.bi_utils, 27
 data_pipelines_cli.cli_commands, 23
 data_pipelines_cli.cli_commands.clean, 24
 data_pipelines_cli.cli_commands.compile, 24
 data_pipelines_cli.cli_commands.deploy, 24
 data_pipelines_cli.cli_commands.docs, 25
 data_pipelines_cli.cli_commands.generate, 23
 data_pipelines_cli.cli_commands.generate.generate, 23
 data_pipelines_cli.cli_commands.generate.model_yaml, 23
 data_pipelines_cli.cli_commands.generate.source_sql, 23
 data_pipelines_cli.cli_commands.generate.source_yaml, 24
 data_pipelines_cli.cli_commands.generate.utils, 24
 data_pipelines_cli.cli_commands.prepare_env, 25
 data_pipelines_cli.cli_commands.publish, 26
 data_pipelines_cli.cli_commands.run, 26
 data_pipelines_cli.cli_commands.seed, 26
 data_pipelines_cli.cli_commands.template, 26
 data_pipelines_cli.cli_commands.test, 26
 data_pipelines_cli.cli_configs, 27
 data_pipelines_cli.cli_constants, 28
 data_pipelines_cli.cli_utils, 28
 data_pipelines_cli.config_generation, 29
 data_pipelines_cli.data_structures, 31
 data_pipelines_cli.dbt_utils, 32
 data_pipelines_cli.docker_response_reader, 33
 data_pipelines_cli.errors, 34
 data_pipelines_cli.filesystem_utils, 36
 data_pipelines_cli.io_utils, 36
 data_pipelines_cli.jinja, 37
 data_pipelines_cli.looker_utils, 37
 msg (*DockerReadResponse* attribute), 33
 N
 name (*DbtModel* attribute), 31
 name (*DbtSource* attribute), 31
 name (*DbtTableColumn* attribute), 31
 NoConfigFileError, 35
 NotAProjectDirectoryError, 35
 NotSupportedBIError, 35
 O
 outputs (*DbtProfile* attribute), 29
 P
 prepare_env() (in module *data_pipelines_cli.cli_commands.prepare_env*), 25

PROFILE_NAME_ENV_EXECUTION (in module *data_pipelines_cli.cli_constants*), 28
PROFILE_NAME_LOCAL_ENVIRONMENT (in module *data_pipelines_cli.cli_constants*), 28
provider_kwargs_dict (*DeployCommand* attribute), 25
publish_package() (in module *data_pipelines_cli.cli_commands.publish*), 26

Q

quote (*DbtTableColumn* attribute), 31

R

read_bi_config() (in module *data_pipelines_cli.bi_utils*), 27
read_dbt_vars_from_configs() (in module *data_pipelines_cli.dbt_utils*), 32
read_dictionary_from_config_directory() (in module *data_pipelines_cli.config_generation*), 30
read_env_config() (in module *data_pipelines_cli.data_structures*), 32
read_looker_config() (in module *data_pipelines_cli.looker_utils*), 37
read_response() (*DockerResponseReader* method), 33
remote_path_str (*LocalRemoteSync* attribute), 36
replace() (in module *data_pipelines_cli.io_utils*), 36
replace_image_settings() (in module *data_pipelines_cli.cli_commands.compile*), 24
replace_vars_with_values() (in module *data_pipelines_cli.jinja*), 37
repository (*DockerArgs* attribute), 32
request_handler() (*AirbyteFactory* method), 27
run() (in module *data_pipelines_cli.cli_commands.run*), 26
run_dbt_command() (in module *data_pipelines_cli.dbt_utils*), 32

S

schema (*DbtSource* attribute), 31
seed() (in module *data_pipelines_cli.cli_commands.seed*), 26
submessage (*AirflowDagsPathKeyError* attribute), 34
submessage (*DataPipelinesError* attribute), 34
submessage (*DependencyNotInstalledError* attribute), 34
submessage (*DockerErrorResponseError* attribute), 34
submessage (*DockerNotInstalledError* attribute), 35
submessage (*JinjaVarKeyError* attribute), 35
submessage (*NoConfigFileError* attribute), 35
submessage (*NotAProjectDirectoryError* attribute), 35
submessage (*NotSupportedBIErrors* attribute), 35

submessage (*SubprocessNonZeroExitError* attribute), 35
submessage (*SubprocessNotFound* attribute), 36
subprocess_run() (in module *data_pipelines_cli.cli_utils*), 29
SubprocessNonZeroExitError, 35
SubprocessNotFound, 35
sync() (*LocalRemoteSync* method), 36

T

tables (*DbtSource* attribute), 31
tags (*DbtModel* attribute), 31
tags (*DbtSource* attribute), 31
tags (*DbtTableColumn* attribute), 32
target (*DbtProfile* attribute), 29
template_name (*TemplateConfig* attribute), 32
template_path (*TemplateConfig* attribute), 32
TemplateConfig (class in *data_pipelines_cli.data_structures*), 32
templates (*DataPipelinesConfig* attribute), 31
test() (in module *data_pipelines_cli.cli_commands.test*), 26
tests (*DbtModel* attribute), 31
tests (*DbtTableColumn* attribute), 32

U

update_file() (*AirbyteFactory* method), 27

V

vars (*DataPipelinesConfig* attribute), 31