**BEST PRACTICE**

**Azure Cloud Development**

# Versioning

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Author | Remarks | Version |
| 19-04-2022 | Dave Gabriël | Initial setup for document | V0.1 |
| 01-05-2022 | Dave Gabriël | Added Git best practice | V0.2 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Index

[Versioning 2](#_Toc102337127)

[Introduction 4](#_Toc102337128)

[Overview of the platform 5](#_Toc102337129)

[Ingestion: Azure Data Factory 7](#_Toc102337130)

[Ingestion 7](#_Toc102337131)

[Folders in ADF 8](#_Toc102337132)

[Naming conventions in ADF 8](#_Toc102337133)

[Transformation: dbt 9](#_Toc102337134)

[Dbt transformation in ADF 9](#_Toc102337135)

[Models in dbt 9](#_Toc102337136)

[Renewi Model Layers 13](#_Toc102337137)

[Tests and documentation in dbt 16](#_Toc102337138)

[Data Science: Azure Machine Learning 18](#_Toc102337139)

[AML pipelines in ADF 18](#_Toc102337140)

[Further topics 18](#_Toc102337141)

[Version control: Github 19](#_Toc102337142)

[Git guidelines 19](#_Toc102337143)

[Git Workflow 20](#_Toc102337144)

[Git Cheat Sheet 22](#_Toc102337145)

[Infrastructure as a code: Terraform 24](#_Toc102337146)

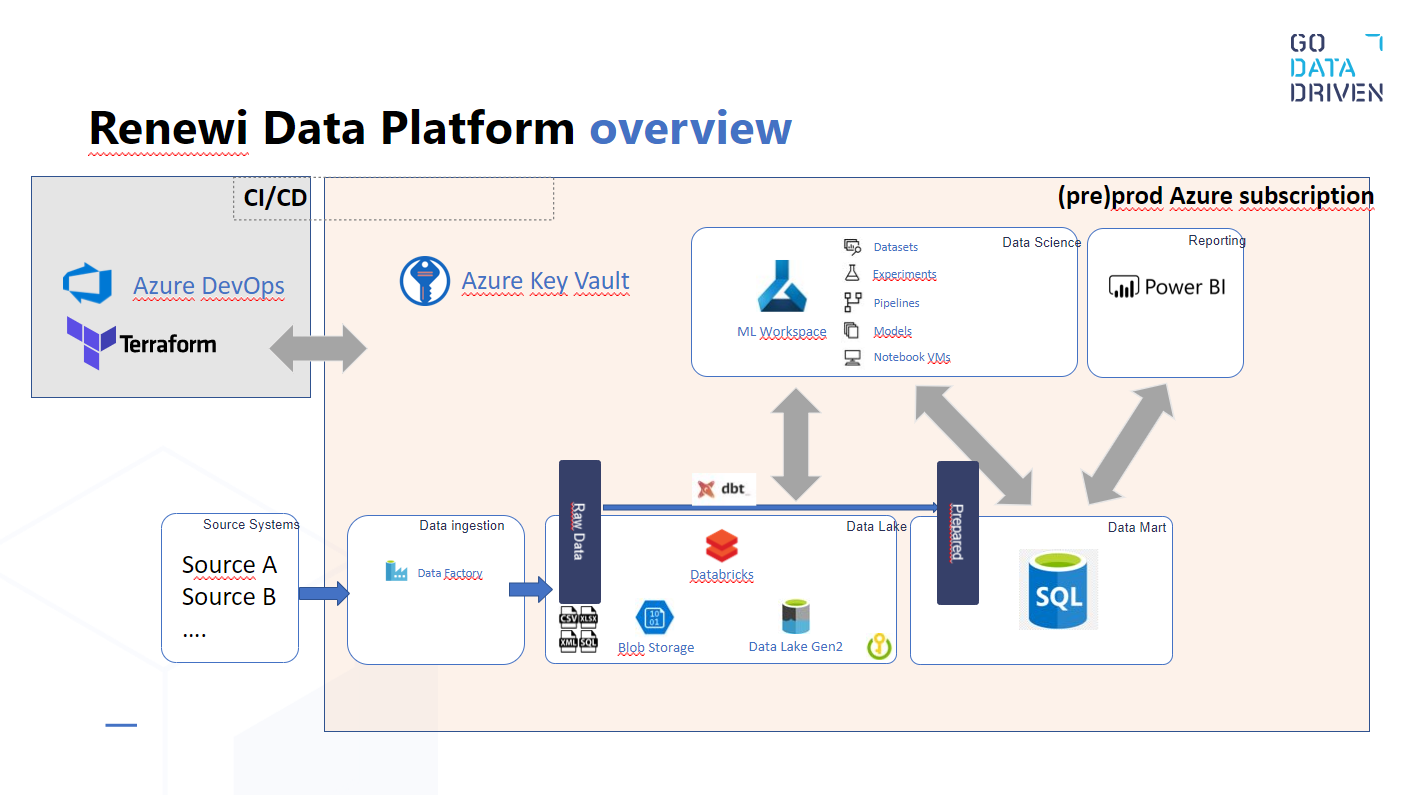
[Further Topics 24](#_Toc102337147)

# Introduction

This document is written to share the best practice in development on the Azure cloud platform implemented at Renewi. This document contains guidelines on how to setup environments for new colleagues just starting to work on the platform. Furthermore the document contains standards for development such as naming conventions, folder structures and information on how to use each part of the platform as a developer.

# Overview of the platform

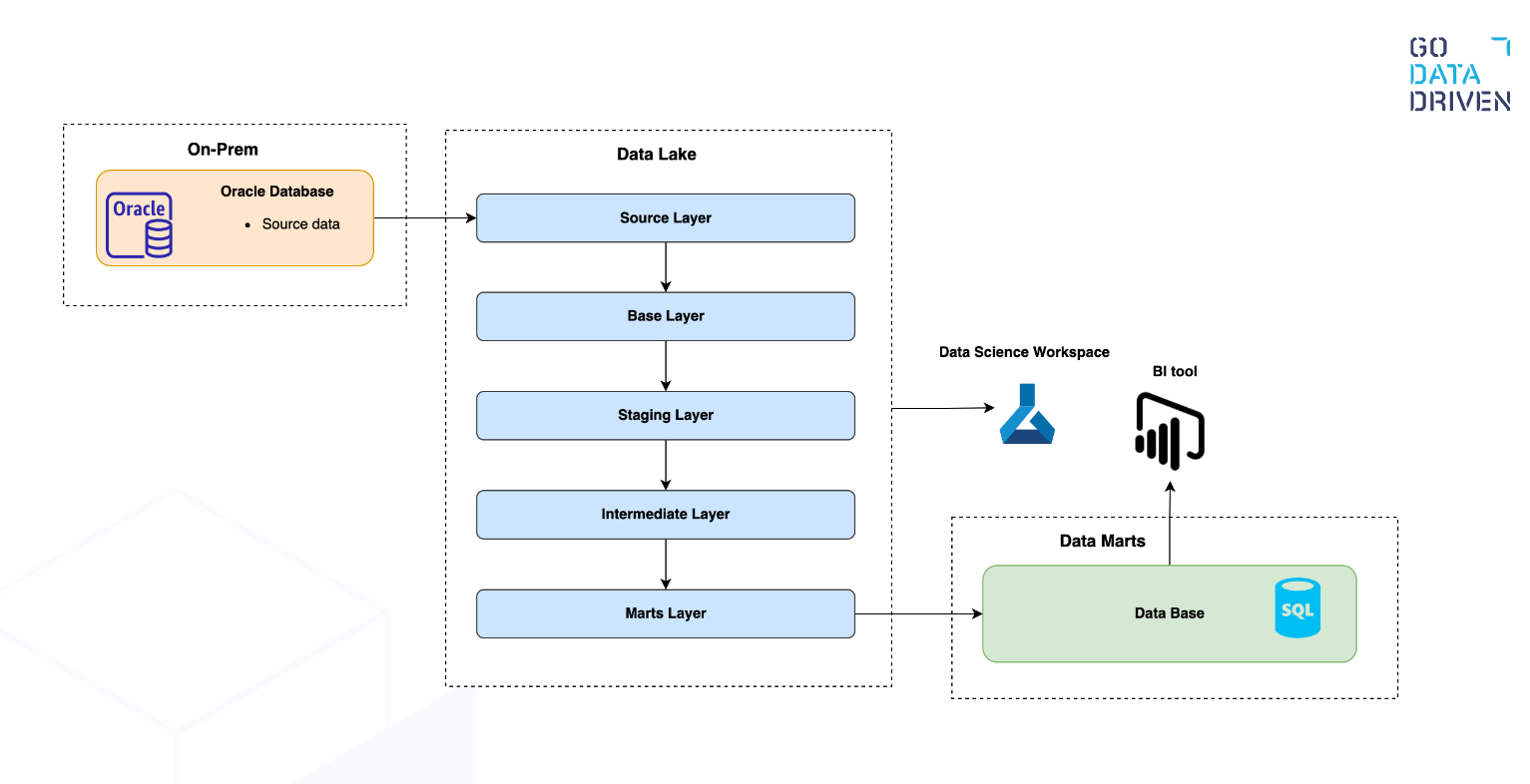
Below an overview of the Azure cloud platform as implemented by Go Data Driven.



Key components:

* Azure DevOps and Terraform for CI/CD and infrastructure as a code. Cegeka is aligned in maintaining the networking and infrastructure part of the platform, including the Azure Key Vault and Resource management. Next to that this includes setting up infra an firewalls to the on-premise platform for Renewi.
* Azure Key Vault is used for storing all authorization and authentication within the platform.
* Azure Data Factory is used for the ingestion of the source data to the Data Lake. Next to that it is used as the overall orchestration tool for scheduling and triggering of the dbt and ML workspaces.
* Azure Blob Storage is used as a sandbox area for new use cases. It is not used in the production process.
* Azure Data Lake Gen2 Storage(ADLS) is used for storing the data ingested from the source systems up until the transformed data is pushed to the SQL Server Datamarts. Note that there is also a Datamarts layer in the ADLS, this is the layer that the SQL Server takes it data from.
* To go threw the different stages of the data, we use dbt to have a composed, documented and version controlled way of defining our data using SQL.
* Databricks is used as the configured compute to execute the dbt SQL query’s on the ADLS.
* Azure machine learning workspace is the main component in the data science workspace. This is the environment where data scientists can explore and experiment. The environment has storage and compute separate from the datalake. However, it does have access to the ADLS for reading data.
* SQL databases are hosted on the cloud to store data coming from dbt and Azure ML environments
* Power BI is the main reporting tool using the data on the SQL databases to build reports

Below a more detailed zoom on the datalake storage is shown. Different layers are shown. Physically these layers are a folder structure on the ADLS. The data is stored as parquet-files for maximum speed and minimum usage of storage.



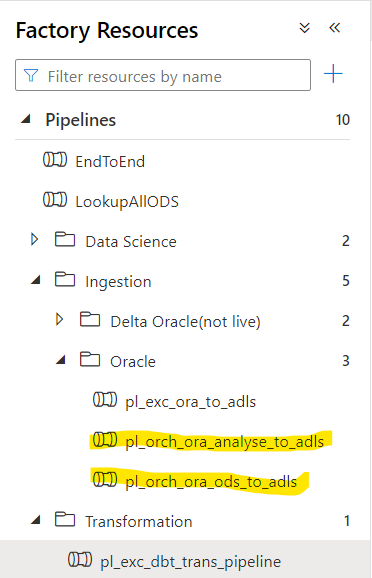
1. *Source: Contains the ingested data by ADF, no transformation on the data is allowed.*
2. *Base[optional]: If there is a very, very good reason to not go directly to staging, this layer gives the flexibility to have an intermediate step between source and staging. Still only raw source centric data is allowed here. As an example this layer can be used when combining the historical and incremental load of a source into one staging model.*
3. Staging: Source centric transformations are applied to standardize the data model.
   1. Renaming columns
   2. Casting columns
   3. Changing timestamps to be in the same timezone or a different dateformat.
4. Intermediate[optional]: It is common that some transformations are the same for different datamarts. Instead of duplication the logic and computation on the different datamarts, this logic should be applied once by using this layer.
5. Marts: From staging to marts businesscentric transformations and joins are applied to create the data models which will be pushed to the SQL databases for the consumer to access.
   1. Joins
   2. Aggregation
   3. Calculating new columns

# Ingestion: Azure Data Factory

Azure Data Factory(ADF) is being used to fulfill two goals; the first is to ingest data from different source systems into the ADLS. The other goal is to orchestrate the overall scheduling of the end to end data ingestion. This means ADF triggers all the runs from ADF’s ingestion, to dbt’s transformation and running the azure machine learning notebooks.

## Ingestion

ADF’s main purpose is getting the data in the datalake. It is adviced to do this in an manner that flexible and doesn’t cost a lot of development effort. As a standard we use 2 pipelines(1 orchestrating pipeline, 1 executing pipeline) and 1 trigger per source system, we can deviate from that when there is a difference in time when source tables are available in the source system for ingestion. Below is an example of the Oracle source. We have an ODS schema which is available for ingestion at 6 am, and an ANALYSE schema which is available at 9 am in the morning.

pl\_orch\_ora\_ods\_to\_adls is used to ingest data from the ODS schema and pl\_orch\_ora\_analyse\_to\_adls is used to ingest data from the ANALYSE schema.

Both pipelines contain 4 parameters:

* Source\_system: Unique name for the sourcesystem, used for setting folder structure in ADLS
* Source\_dataset(Optional): Extra hierarchy when splitting source tables.(For example schema names in Oracle source) Used for locating the source tables for ingestion. Also used in the folder structure in ADLS.
* Source\_tables: Table name in the source system, used for ingesting the source table. Also used in the folder structure in ADLS.
* Target\_container: The container in ADLS to write the data to. Default is Source container.

The parameters are parsed to pl\_exc pipeline threw the pl\_orch pipelines. A ForEach activity is used to walk through the array in the parameter source\_tables. It is then checked if the table is partitioned in order to choose the correct copy style(Partitioned or not). Finally the source\_table is Copied to a target folder using the other parameter values:

Code :  
  
File path= @dataset().target\_container/@concat('Source\_system=', dataset().source\_system, '/Dataset=', dataset().source\_dataset, '/Table=', dataset().source\_table, '/IngestionDate=', formatDateTime(utcnow(), 'yyyy-MM-dd'), '/IngestionTimestamp=', formatDateTime(utcnow(), 'yyyy-MM-ddTHH:mm:sszzz'))

Folder location:

Source\_system=oracle/Dataset=ANALYSE/Table=MV\_SLIM\_WEGEN/IngestionDate=2022-04-19/IngestionTimestamp=2022-04-19T07:00:43+00:00/

## Folders in ADF

Folder structure:

Ingestion/{source}/

Transformation

Data Science

## Naming conventions in ADF

Pipelines

Datasets

Linked Services

Integration Runtimes

Triggers

# Transformation: dbt

For transforming the source data ingested by ADF we use dbt as a our tool to transform and join the data into datamarts. Dbt also functions as our tool for automated testing and our tool for documenting the data.

## Dbt transformation in ADF

Dbt has its own scheduling environment to trigger jobs that run the models that our developed on the environment. However we use ADF as our orchestration tool and in order to maintain dependencies and keep the end-to-end run on one place the dbt jobs are triggered from ADF using the pipeline pl\_orch\_dbt\_trans\_job. This job first makes sure the account id and token are retrieved from the Azure key vault and validated and parsed to the dbt cloud. Then dbt runs the jobs in its own cloud environment using API connection with ADF. ADF runs an intervalled check on the status of the dbt job. Whenever the status is failed of succeeded the pipeline moves on.

## Models in dbt

Model naming:

Our models fit in to two main categories: staging and marts. Where a specific source can possibly have one or more base models. The marts category contains an intermediate layer with models that can also be used in other marts. Below is a treemap of an example structure:

├── dbt\_project.yml

└── models

├── marts

| └── pricing  
 | ├── intermediate

| | ├── tst\_intermediate.yml

| | ├── int\_customers\_unioned.sql

| | ├── int\_customers\_grouped.sql

| └── tst\_pricing.yml

| └── prc\_dim\_customers.sql

| └── prc\_fct\_orders.sql

└── staging

└── oracle

└── pricing

├── base

| ├── tst\_base.yml  
 | ├── prc\_base\_invoices.sql

├── tst\_pricing.yml

├── prc\_stg\_clear\_customers.sql

└── prc\_stg\_clear\_invoices.sql

* All objects should be plural, such as: prc\_stg \_invoices
* Base tables are prefixed with <source>\_base\_, such as: <source>\_base\_<object>
* Intermediate tables should end with a past tense verb indicating the action performed on the object, such as: customers\_unioned
* Marts are categorized between fact (immutable, verbs) and dimensions (mutable, nouns) with a prefix that indicates either. Before the dim/fct prefix there is another prefix showing the folder the object is in. Such as: prc\_fct\_\_orders or prc\_dim\_\_customers. We use the folder prefix first so the objects are grouped together in the consumer SQL database.

Model configuration:

* Model-specific attributes (like sort/dist keys) should be specified in the model.
* If a particular configuration applies to all models in a directory, it should be specified in the dbt\_project.yml file.
* Marts should always be configured as tables
* In-model configurations should be specified like this:

{{  
 config  
(  
 materialized = 'table',  
 sort = 'id',  
 dist = 'id'  
 )  
}}

Naming and field conventions:

* Schema, table and column names should be in snake\_case.
* Use names based on the business terminology, rather than the source terminology.
* Each model should have a primary key.
* The primary key of a model should be named <object>\_id, e.g. account\_id – this makes it easier to know what id is being referenced in downstream joined models.
* For base/staging models, fields should be ordered in categories, where identifiers are first and timestamps are at the end.
* Date’s should be in YYYY-MM-DD format
* More??
* Price/revenue fields should be in decimal currency (e.g. 19.99 for $19.99; many app databases store prices as integers in cents). If non-decimal currency is used, indicate this with suffix, e.g. price\_in\_cents.
* Consistency is key! Use the same field names across models where possible, e.g. a key to the customers table should be named customer\_id rather than user\_id.

Common Table Expressions:

Complex models often include multiple Common Table Expressions (CTEs). In dbt, you can instead separate these CTEs into separate models that build on top of each other. It is often a good idea to break up complex models when:

* A CTE is duplicated across two models. Breaking the CTE into a separate model allows you to reference the model from any number of downstream models, reducing duplicated code.
* A CTE changes the grain of a the data it selects from. It's often useful to test any transformations that change the grain (as in, what one record represents) of your data. Breaking a CTE into a separate model allows you to test this transformation independently of a larger model.
* The SQL in a query contains many lines. Breaking CTEs into separate models can reduce the cognitive load when another dbt user (or your future self) is looking at the code. A model should not contain more than 80 lines of code.

Guidelines on CTE’s:

* All {{ ref('...') }} and {{ source('...', '...') }} statements should be placed in CTEs at the top of the file.
* Where performance permits, CTEs should perform a single, logical unit of work.
* CTE names should be as verbose as needed to convey what they do.
* CTEs with confusing or noteable logic should be commented.
* CTEs that are duplicated across models should be pulled out into their own model, we use the intermediate layer for this.
* create a final or similar CTE that you select from as your last line of code. This makes it easier to debug code within a model (without having to comment out code!)
* For more information about why we use so many CTEs, check out [this link](https://discourse.getdbt.com/t/why-the-fishtown-sql-style-guide-uses-so-many-ctes/1091).
* CTEs should be formatted like this:

With  
events as   
(  
...  
),  
-- CTE comments go here  
filtered\_events as   
(  
...  
)  
  
select \* from filtered\_events

SQL style guide:

* Lines of SQL should be no longer than 80 characters
* Field names and function names should all be lowercase
* The as keyword should be used when aliasing a field or table
* Fields should be stated before aggregates / window functions
* Aggregations should be executed as early as possible before joining to another table.
* Ordering and grouping by a number (eg. group by 1, 2) is preferred over listing the column names (see [this](https://blog.getdbt.com/write-better-sql-a-defense-of-group-by-1/) rant for why). Note that if you are grouping by more than a few columns, it may be worth revisiting your model design.
* Prefer union all to union \*
* Avoid table aliases in join conditions (especially initialisms) – it's harder to understand what the table called "c" is compared to "customers".
* If joining two or more tables, always prefix your column names with the table alias. If only selecting from one table, prefixes are not needed.
* Be explicit about your join (i.e. write inner join instead of join). left joins are normally the most useful, right joins often indicate that you should change which table you select from and which one you join to.
* DO NOT OPTIMIZE FOR A SMALLER NUMBER OF LINES OF CODE. NEWLINES ARE CHEAP, BRAIN TIME IS EXPENSIVE

**Example SQL**

With  
my\_data as  
(  
 select \* from {{ ref('my\_data') }}  
),

some\_cte as  
(  
 select \* from {{ ref('some\_cte') }}  
),

some\_cte\_agg as   
(  
 select  
 id,  
 sum(field\_4) as total\_field\_4,  
 max(field\_5) as max\_field\_5  
 from some\_cte  
 group by 1  
),

final as   
(  
 select [distinct]  
 my\_data.field\_1,  
 my\_data.field\_2,  
 my\_data.field\_3,  
  
 -- use line breaks to visually separate calculations into blocks

case  
 when my\_data.cancellation\_date is null  
 and my\_data.expiration\_date is not null  
 then expiration\_date  
 when my\_data.cancellation\_date is null  
 then my\_data.start\_date + 7  
 else my\_data.cancellation\_date  
 end as cancellation\_date,

some\_cte\_agg.total\_field\_4,  
 some\_cte\_agg.max\_field\_5  
 from my\_data  
 left join some\_cte\_agg   
 on my\_data.id = some\_cte\_agg.id  
 where my\_data.field\_1 = 'abc'  
 and (  
 my\_data.field\_2 = 'def' or  
 my\_data.field\_2 = 'ghi'  
 )  
 having count(\*) > 1  
)

select \* from final

* Your join should list the "left" table first (i.e. the table you are selecting from):

Select  
 trips.\*,  
 drivers.rating as driver\_rating,  
 riders.rating as rider\_rating  
from trips  
left join users as drivers  
 on trips.driver\_id = drivers.user\_id  
left join users as riders  
 on trips.rider\_id = riders.user\_id

Guidelines for dbt\_project.yml file:

Configure groups of models, by specifying configurations in your dbt\_project.yml file.

Configure your materializations in the project file

* Views are faster to build, but slower to query compared to tables.
* Incremental models provide the same query performance as tables, are faster to build compared to the table materialization, however they introduce complexity into a project.
* Use views by default
* Use ephemeral models for lightweight transformations that shouldn't be exposed to end-users
* Use tables for models that are queried by BI tools
* Use tables for models that have multiple descendants
* Use incremental models when the build time for table models exceeds an acceptable threshold

Renewi Model Layers

Source:

For each source and subsource we use a source\*.yml file to define the source tables and their location. Each src\_\*.yml contains: [Source](https://docs.getdbt.com/docs/building-a-dbt-project/using-sources) definitions, tests, and documentation. Source objects have a description, columns have a description and the primary key has not null and unique tests defined.

For oracle we ingest data from two schema’s and so we use a source.yml file for each schema. The files are named src\_<source\_system>\_<source\_dataset>.yml, example: src\_ora\_analyse.yml.

Currently we only load data from the current day into dbt. This is done in the identifier for the tables:

version: 2

sources:  
 - name: ORA\_ODS  
 description: This source is a reference to the 'analyse' schema in the oracle database used for  
 ingestion data in ADLS.  
 schema: parquet  
 tables:

- name: VGI\_CL\_ODS\_FACTUURREGEL  
 identifier: '`dbfs:///mnt/source/Source\_system=oracle/Dataset=ODS/  
 Table=VGI\_CL\_ODS\_FACTUURREGEL/  
 IngestionDate={{ modules.datetime.date.today() }}`'  
 description: Meaningfull text?  
# columns:  
# - name: abc  
# description: Meaningfull text?  
# tests:  
# - unique  
# - not\_null

Staging:

The staging layer is source centric but with a business focus. This means that for each source table there is a staging table. However the name and structure of the table and its columns are cleansed and transformed into the corporate business format. The goal of this layer is to standardize the model from different sources. Staging models take raw data, and clean and prepare them for further analysis. A staging model means that:

* Fields have been renamed and recast in a consistent way.
* Datatypes, such as timezones, are consistent.
* Light cleansing, such as replacing empty string with NULL values, has occurred.
* If useful, flattening of objects might have occurred.
* There is a primary key that is both unique and not null (and tested).

In order to create full lineage over the model we use the source function after the FROM statement of the query:

with dates as   
(  
 select \* from {{ source('ORA\_ANALYSE', 'POR\_DATES') }}  
)

select \* from dates  
  
Each staging directory contains at a minimum:

* One staging model for each source object that is useful for analytics:
  + Named <source>\_stg\_<object>.
  + Generally materialized as a view (unless performance requires it as a table).
* A tst\_<layer>\_<source>.yml file which contains
  + [Tests and documentation](https://docs.getdbt.com/docs/building-a-dbt-project/tests) for models in the same directory

Because we ingest data from different sources we use multiple folders in the staging layer.

├── dbt\_project.yml  
 └── models  
 ├── marts  
 └── staging  
 └── oracle  
 └── portal

Base(optional):

The base layer is optional and only used when object from the same source and with the same structure need to be joined. For example a historical load with an incremental load. Or like in the example below a table with failed and succeeded payments.

├── dbt\_project.yml  
 └── models  
 ├── marts  
 └── staging  
 └── oracle  
 └── portaal  
 ├── base  
 | ├── tst\_bse\_portaal.yml  
 | ├── base\_portaal\_\_failed\_payments.sql  
 | └── base\_portaal\_\_succeeded\_payments.sql  
 ├── tst\_stg\_portaal.yml  
 └── stg\_portaal\_payments.sql

The base models are always placed as a subfolder within the sourcefolder they belong to.

Marts:

Marts are stores of models that describe business entities and processes. They are often grouped by business unit: marketing, finance, product. Models that are shared across an entire business are grouped in a core directory.

├── dbt\_project.yml  
 └── models  
 ├── marts  
 | ├── core  
 | ├── finance  
 | ├── marketing  
 | └── pricing  
 └── staging

Marts are divided in facts and dimensions:

* <model>\_fct\_<verb>: A tall, narrow table representing real-world processes that have occurred or are occurring. The heart of these models is usually an immutable event stream: sessions, transactions, orders, stories, votes.
* <model>\_dim\_<noun>: A wide, short table where each row is a person, place, or thing; the ultimate source of truth when identifying and describing entities of the organization. They are mutable, though slowly changing: customers, products, candidates, buildings, employees.

Where the work of staging models is limited to cleaning and preparing, fact tables are the product of substantive data transformation: choosing (and reducing) dimensions, date-spining, executing business logic, and making informed, confident decisions.

This layer of modeling is considerably more complex than creating staging models, and the models we design are highly tailored to the analytical needs of an organization. As such, we have far less convention when it comes to these models. However fct\_ and dim\_ models should be materialized as tables within a warehouse to improve query performance. As a default, we use the table materialization, and where performance requires it, we use the incremental materialization.

Intermediate(optional):

Intermediate transformations may be required to get to a fact or dimension model. Typically the layer is used to define reusable CTE’s so that the code will not be placed in different marts multiple times. The models are placed in a nested marts/<mart>/intermediate directory. They are named int\_<useful\_name>\_<transformation\_in\_past\_tense>.sql, example: customers\_unioned. The int prefix and use of double underscores indicates that these are intermediate models.

Models are tested and documented in a tst\_<layer>\_<model>.yml file in the same directory as the models.

A marts directory may therefore end up looking like:

├── dbt\_project.yml  
 └── models  
 ├── marts  
 │ ├── core  
 │ │ ├── tst\_core.md  
 │ │ ├── tst\_core.yml  
 │ │ ├── dim\_customers.sql  
 │ │ ├── fct\_orders.sql  
 │ │ └── intermediate  
 │ │ ├── int\_customer\_orders\_grouped.sql  
 │ │ ├── int\_customer\_payments\_grouped.sql  
 │ │ ├── tst\_int\_core.yml  
 │ │ └── int\_order\_payments\_joined.sql  
 │ ├── finance  
 │ ├── marketing  
 │ └── product  
 └── staging

## Tests and documentation in dbt

Tests are assertions you make about your models and other resources in your dbt project (e.g. sources, seeds and snapshots). When you run dbt test, dbt will tell you if each test in your project passes or fails.

Like almost everything in dbt, tests are SQL queries. In particular, they are select statements that seek to grab "failing" records, ones that disprove your assertion. If you assert that a column is unique in a model, the test query selects for duplicates; if you assert that a column is never null, the test seeks after nulls. If the test returns zero failing rows, it passes, and your assertion has been validated.

For each model we use generic tests where not null and duplicate on the primary key are mandatory.

Every subdirectory should contain a .yml file, in which each model in the subdirectory is tested and documented. For each folders, the naming structure should be tst\_<layer>\_<model>.yml.

Documentation is another very important part of our dbt code. The lakehouse is to be the single point of truth and corporate source for retrieving the data within Renewi. It makes it very important to describe the data in a proper manner so all consumers use the same definitions moving forward. At a minimum Objects and columns need to be described in the .yml files. Also business rules need to be described whenever applied on a column.

YAML style guide

* Indents should be two spaces
* List items should be indented
* Use a new line to separate list items that are dictionaries where appropriate
* Lines of YAML should be no longer than 80 characters

**Example YAML**

version: 2

models:  
 - name: events  
 columns:  
 - name: event\_id  
 description: This is a unique identifier for the event  
 tests:  
 - unique  
 - not\_null

- name: event\_time  
 description: "When the event occurred in UTC (eg. 2018-01-01 12:00:00)"  
 tests:  
 - not\_null

- name: user\_id  
 description: The ID of the user who recorded the event  
 tests:  
 - not\_null  
 - relationships:  
 to: ref('users')  
 field: id

# Data Science: Azure Machine Learning

For running machine learning and data science models on the data.

## AML pipelines in ADF

Explanation of pipelines to be added.

## Further topics

# Version control: Github

For version control we use Github on all our tooling and code. It is used for ADF, dbt and Azure ML. For all tooling the same best practice applies. There are some best practices that we keep in mind and there is a workflow to follow when developing.

All our coding and development tasks are logged in a GIT repository configured in Azure DevOps. There are four(?) repositories defined:

* DBT Analytics
  + Git repo for changes made in the dbt project ACC.
* Data Analytics Platform
  + Git repo for the IaaC scripts controlling the infrastructure on Azure.
* ADF Ingestion
  + Git repo containing all the JSON files that together represent the ADF environment.
* Data Science
  + A git repository for machine learning scripts.

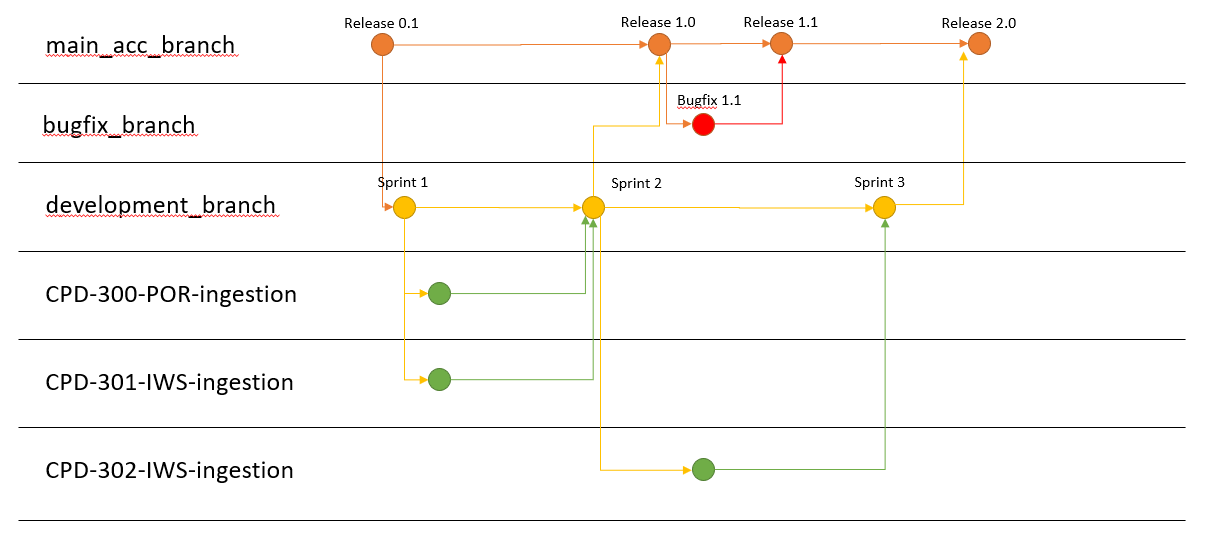
## Git guidelines

* U are free to use your own coding tool. If you are not familiar with coding/command tools then working with Visual Code Studio is a free and nice tool to use.
* Use the JIRA userstory code when creating a branch. <jira code>\_<short description>
  + For example ‘git branch CDB-334-ingest-POR’Afbeelding met tekst

    Automatisch gegenereerde beschrijving
* Commit related changes
  + A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other developers to understand the changes and roll them back if something went wrong. With tools like the staging area and the abillity to stage only parts of a file, Git makes it easy to create very granular commits.
* Commit often
  + Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it‘s easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard to solve conflicts.
* Do not commit half-done work
  + You should only commit code when it‘s completed. As stated above, this doesn‘t mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature‘s implementation into logical chunks and remember to commit early and often. But don‘t commit just to have something in the repository before leaving the office at the end of the day. If you‘re tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) use Git‘s ‘Stash’ feature instead.
* Test code before you commit
  + Resist the temptation to commit something that you ‘think’ is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing half-baked things in your local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing/sharing your code with others.
* Write good commit messages
  + Write your message with a short summary of your changes (up to 50 characters as a guideline).
* Use Branches
  + Branches are the perfect tool to help you avoid mixing up different lines of development. We use branches extensively in your development workflows: for new features, bug fixes, ideas….

## Git Workflow

Currently we have a main branch for the acc environment and a main branch for the prd environment. Because the acc environment is cloned automatically to the prd environment it is not desired that developers merge their branches into the main branch. We want to merge our branches into the ‘Development’ branch and after each sprint the ‘Development’ branch is merged into the main branch as a release, containing all features that have been delivered by the team. Remember this has to be done for all Repositories. In a diagram this would look as follows:



1. **Clone the repository**

When starting to work on one of the repo’s select the repository In the toolbar and click on ‘Clone’

Afbeelding met tekst, schermafbeelding, computer

Automatisch gegenereerde beschrijving

Select your coding tool of choice and click ‘Clone in …’. There will be a local clone of the repository in your coding environment.

1. **Start a new branch**

‘git branch <new-branch-name>’  
 Whenever you start a new feature, a new experiment or a new bugfix, you should create a new branch. In Git, this is extremely fast and easy: just call “git branch ” and you have a new, separate context.

1. Commit changes

Commit whenever a small set of relates changes is finished. For example when finishing a stg script in dbt, or after adding a linked service in ADF.

1. Pull latest changes

When your development is done. Before you merge your changes in the local main branch be sure to pull the remote main branch into the local branch. This can be done with the command: ‘git pull origin master’

1. Merge your branch

Merge your branch into your local main branch. This is done by switching into the local main branch: ‘git swith main’ and using the merge command: ‘git merge CPD-304-POR-ingestion’.

1. Resolve any conflicts

When resolving conflicts be sure to discuss any doubts with a peer-developer.

1. Push your branch to the remote

When your branch and the remote main branch are merged on your local repository u can push the feature branch to the remote server using: ‘git push -u origin <local-branch-name>’ Using -u makes sure your local branch is ‘tracked’ with the remote branch. For future pulls and pushes u just need to type ‘git pull’ and ‘git push’. Be sure to have to local branch checked out.

## Git Cheat Sheet

**General commands**

|  |  |
| --- | --- |
| git --version | Check installed git version |
| git init | Create empty git repository |
| git status | Check working directory & staging area status |
| git log | Display all commits of current branch |
| git ls-files | List all data in the staging area |

**Stage & Commit**

|  |  |
| --- | --- |
| git add <filename> | Add single file to staging area |
| git add . | Add all Working Directory to staging area |
| git commit -m “<message>” | Create new commit on current branch |
| git blame <file> | Track who changed what and when in <file> |

**Branches**

|  |  |
| --- | --- |
| git branch | List all local branches |
| git branch <branchname> | Create new branch |
| git branch -D <branchname> | Delete branch |
| git switch <branchname> | Go to branch |
| git checkout <commitid> | Checkout commit(detached head) of current branch |
| git checkout -b <branchname> | Create a new branch and checkout the branch |
| git merge <branche\_to\_pull\_in> | Bring other branch’s changes to current branch |
| git rebase <branche to pull in> | Bring commits from other branch in current branch(these commits will get new hashes so only use this LOCAL) |
| git cherry-pick <commit-hash> | Bring mentioned commit in to current branch |

**Remote**

|  |  |
| --- | --- |
| git branch -r | Show all remote tracking branches |
| git branch -a | Show all existing branches |
| git branch -vv | Show information on tracking local-remote branches |
| git fetch origin | Fetch all remote tracking branches(nothing local yet) |
| git pull origin master | Pull the remote branch commit linked to the local branch |
| git push origin <branch> | Push local branch to remote |
| git push -u origin <branch> | Push local branch to remote and enable upstream tracking |
| git push origin –delete <branch> | Delete remote branch |
| git branch –track <remotebranchname> | Create a tracked local version of a remote branch |

**Stash**

|  |  |
| --- | --- |
| git stash | Stash uncommited changes and return to last commit |
| git stash apply | Bring changes into WD from latest stash |
| git stash list | List all stashes |
| git stash apply {1} | Go to stash number 1 |

**Tags**

|  |  |
| --- | --- |
| git tag | Show all tags |
| git tag <tagname> <commit-hash> | Bring changes into WD from latest stash |
| git tag -a <tagname> -m “text” | Create tag of latest commit with message |
| git show <tagname> | Show named tag |
| git tag -d <tagname> | Delete tag |

**Undo Unstaged**

|  |  |
| --- | --- |
| git restore <filename> | Revert changes in a unstaged file |
| git restore . | Revert changes in all unstaged files |
| git clean -dn | List all new files not part of a commit yet |
| git clean -df | Delete new files not part of a commit yet |

**Undo Staged**

|  |  |
| --- | --- |
| git restore –staged <filename> | Remove file from staging area |
| git restore –staged . | Remove all files from staging area |

**Undo commits**

|  |  |
| --- | --- |
| git reset --soft HEAD~2 | Reset commit 2 commits back but keep WD and staged changes |
| git reset –hard HEAD~3 | Reset commit 3 commits back and reset WD and staging aswell |
| git reflog | List all commits(included lost/deleted ones) |
| git reset –hard <commit-hash> |  |

# Infrastructure as a code: Terraform

Dasfd

## Further Topics

kjk