Singular tests[​](https://docs.getdbt.com/docs/build/tests#singular-tests)

The simplest way to define a test is by writing the exact SQL that will return failing records. We call these "singular" tests, because they're one-off assertions usable for a single purpose.

These tests are defined in .sql files, **typically in your tests directory (as defined by your**[**test-paths config**](https://docs.getdbt.com/reference/project-configs/test-paths)**)**. You can use **Jinja (including ref and source**) in the test definition, just like you can when creating models.

Each .sql file contains one select statement, and it defines one test:

**Example:**

**tests/assert\_total\_payment\_amount\_is\_positive.sql**

*-- Refunds have a negative amount, so the total amount should always be >= 0.*  
*-- Therefore return records where this isn't true to make the test fail*

**select  
 order\_id,  
 sum(amount) as total\_amount  
from {{ ref('fct\_payments' )}}  
group by 1  
having not(total\_amount >= 0)**

The name of this test is the name of the file: **assert\_total\_payment\_amount\_is\_positive**. Simple enough.

Singular tests are easy to write—so easy that you may find yourself writing the same basic structure over and over, only changing the name of a column or model. By that point, the test isn't so singular! In that case**, we recommend...**

Generic tests[​](https://docs.getdbt.com/docs/build/tests#generic-tests)

Certain tests are generic: they can be reused over and over again. A generic test is defined in a test block, which contains a parametrized query and accepts arguments. It might look like:

{% test not\_null(model, column\_name) %}  
  
 select \*  
 from {{ model }}  
 where {{ column\_name }} is null  
  
{% endtest %}

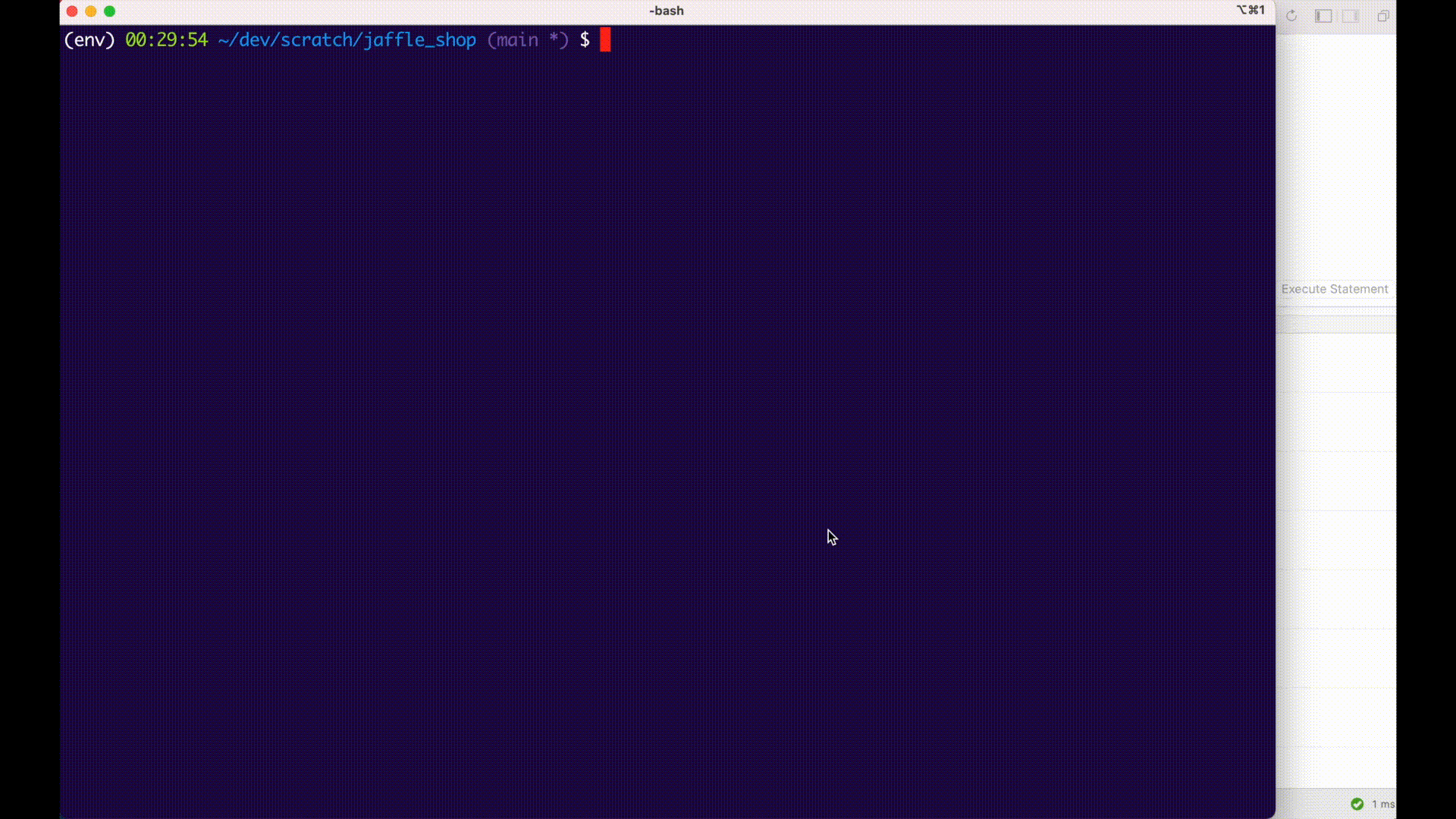
Check out the results of SQL dbt is running by either:

* + dbt Cloud: checking the **Details tab.**
  + dbt CLI: checking the **target/compiled directory**

Storing test failures[​](https://docs.getdbt.com/docs/build/tests#storing-test-failures)

Normally, a test query will calculate failures as part of its execution. If you set the optional --store-failures flag or [store\_failures config](https://docs.getdbt.com/reference/resource-configs/store_failures), dbt will first save the results of a test query to a table in the database, and then query that table to calculate the number of failures.

This workflow allows you to query and examine failing records much more quickly in development:

[](https://docs.getdbt.com/docs/build/tests)

Store test failures in the database for faster development-time debugging.

Note that, if you elect to store test failures:

* **Test result tables are created in a schema suffixed or named dbt\_test\_\_audit, by default. It is possible to change this value by setting a schema config.**

models:

  - name: dim\_location

    description: Check whether the values of fact\_id is unique or not

    config:

      tags: ['tag']

    columns:

      - name: location\_id

        tags: ['loc\_tag']

        tests:

          - unique:

              store\_failures: true

              schema: "loc\_test\_uniq"

* **A test's results will always replace previous failures for the same test.**

Commands:

* **In test config file:**

        tests:

          - unique:

              store\_failures: true

              schema: "loc\_test\_uniq"

* **in cli flag**

**Dbt test –m abc –-store-failure**

**Test selection examples**

Test selection works a little differently from other resource selection. This makes it very easy to:

* run tests on a particular model
* run tests on all models in a subdirectory
* run tests on all models upstream / downstream of a model, etc.

Like all resource types, tests can be selected **directly**, by methods and operators that capture one of their attributes: their name, properties, tags, etc.

# Direct selection

* Run generic tests only:

$ dbt test --select test\_type:generic

* Run singular tests only:

$ dbt test --select test\_type:singular

In both cases, test\_type checks a property of the test itself. These are forms of "direct" test selection.

# Indirect selection

**Syntax examples**[**​**](https://docs.getdbt.com/reference/node-selection/test-selection-examples?indirect-selection-mode=eager#syntax-examples)

Setting indirect\_selection can also be specified in a [yaml selector](https://docs.getdbt.com/reference/node-selection/yaml-selectors" \l "indirect-selection).

The following examples should feel somewhat familiar if you're used to executing dbt run with the --select option to build parts of your DAG:

***# Run tests on a model (indirect selection)***  
$ dbt test --select customers  
  
***# Run tests on all models in the models/staging/jaffle\_shop directory (indirect selection)***  
$ dbt test --select staging.jaffle\_shop  
  
***# Run tests downstream of a model (note this will select those tests directly!)***  
$ dbt test --select stg\_customers+  
  
***# Run tests upstream of a model (indirect selection)***  
$ dbt test --select +stg\_customers  
  
***# Run tests on all models with a particular tag (direct + indirect)***  
$ dbt test --select tag:my\_model\_tag  
  
***# Run tests on all models with a particular materialization (indirect selection)***  
$ dbt test --select config.materialized:table

The same principle can be extended to tests defined on other resource types. In these cases, we will execute all tests defined on certain sources via the source: selection method:

***# tests on all sources***  
$ dbt test --select source:\*  
  
***# tests on one source***  
$ dbt test --select source:jaffle\_shop  
  
***# tests on one source table***  
$ dbt test --select source:jaffle\_shop.customers  
  
***# tests on everything \_except\_ sources***  
$ dbt test --exclude source:\*

**More complex selection**[**​**](https://docs.getdbt.com/reference/node-selection/test-selection-examples?indirect-selection-mode=eager#more-complex-selection)

Through the combination of direct and indirect selection, there are many ways to accomplish the same outcome. Let's say we have a data test named assert\_total\_payment\_amount\_is\_positive that depends on a model named payments. All of the following would manage to select and execute that test specifically:

$ dbt test --select assert\_total\_payment\_amount\_is\_positive ***# directly select the test by name***  
$ dbt test --select payments,test\_type:**singular *# indirect selection, v1.2***  
$ dbt test --select payments,test\_type:data ***# indirect selection, v0.18.0***  
$ dbt test --select payments --data  ***# indirect selection, earlier versions***

As long as you can select a common property of a group of resources, indirect selection allows you to execute all the tests on those resources, too. In the example above, we saw it was possible to test all table-materialized models. This principle can be extended to other resource types, too:

***# Run tests on all models with a particular materialization***  
$ dbt test --select config.materialized:table  
  
***# Run tests on all seeds, which use the 'seed' materialization***  
$ dbt test --select config.materialized:seed  
  
***# Run tests on all snapshots, which use the 'snapshot' materialization***  
$ dbt test --select config.materialized:snapshot

**Run tests on tagged columns**[**​**](https://docs.getdbt.com/reference/node-selection/test-selection-examples#run-tests-on-tagged-columns)

Because the column order\_id is tagged my\_column\_tag, the test itself also receives the tag my\_column\_tag. Because of that, this is an example of direct selection.

models/<filename>.yml

version: 2  
  
models:  
 - name: orders  
 columns:  
 - name: order\_id  
 tests:  
 tags: [my\_column\_tag]  
 - unique

$ dbt test --select tag:my\_column\_tag

Currently, tests "inherit" tags applied to columns, sources, and source tables. They do *not* inherit tags applied to models, seeds, or snapshots. In all likelihood, those tests would still be selected indirectly, because the tag selects its parent. This is a subtle distinction, and it may change in future versions of dbt.

**Run tagged tests only**[**​**](https://docs.getdbt.com/reference/node-selection/test-selection-examples#run-tagged-tests-only)

This is an even clearer example of direct selection: the test itself is tagged my\_test\_tag, and selected accordingly.

models/<filename>.yml

version: 2  
  
models:  
 - name: orders  
 columns:  
 - name: order\_id  
 tests:  
 - unique:  
 tags: [my\_test\_tag]

$ dbt test --select tag:my\_test\_tag