Pytest Alembic

Release 0.4.0

unknown

Apr 18, 2023

CONTENTS:

CHAPTER

ONE

QUICKSTART

See the full documentation here.

1.1 Introduction

A pytest plugin to test alembic migrations (with default tests) and which enables you to write tests specific to your migrations.

```
$ pip install pytest-alembic
$ pytest --test-alembic

...
::pytest_alembic/tests/model_definitions_match_ddl <- . PASSED           [ 25%]
::pytest_alembic/tests/single_head_revision <- . PASSED                  [ 50%]
::pytest_alembic/tests/up_down_consistency <- . PASSED                   [ 75%]
::pytest_alembic/tests/upgrade <- . PASSED                               [100%]

============================== 4 passed in 2.32s ===============================
```

1.2 The pitch

Have you ever merged a change to your models and you forgot to generate a migration?

Have you ever written a migration only to realize that it fails when there's data in the table?

Have you ever written a **perfect** migration only to merge it and later find out that someone else merged also merged a migration and your CD is now broken!?

`pytest-alembic` is meant to (with a little help) solve all these problems and more. Note, due to a few different factors, there **may** be some minimal required setup; however most of it is boilerplate akin to the setup required for alembic itself.

1.2.1 Built-in Tests

- **test_single_head_revision**

  Assert that there only exists one head revision.

  We're not sure what realistic scenario involves a diverging history to be desirable. We have only seen it be the result of uncaught merge conflicts resulting in a diverged history, which lazily breaks during deployment.

- **test_upgrade**

  Assert that the revision history can be run through from base to head.

- **test_model_definitions_match_ddl**

  Assert that the state of the migrations matches the state of the models describing the DDL.

  In general, the set of migrations in the history should coalesce into DDL which is described by the current set of models. Therefore, a call to `revision --autogenerate` should always generate an empty migration (e.g. find no difference between your database (i.e. migrations history) and your models).

- **test_up_down_consistency**

  Assert that all downgrades succeed.

  While downgrading may not be lossless operation data-wise, there's a theory of database migrations that says that the revisions in existence for a database should be able to go from an entirely blank schema to the finished product, and back again.

- Experimental tests
  - all_models_register_on_metadata

    Assert that all defined models are imported statically.

    Prevents scenarios in which the minimal import of your models in your `env.py` does not import all extant models, leading alembic to not autogenerate all your models, or (worse!) suggest the deletion of tables which should still exist.

  - downgrade_leaves_no_trace

    Assert that there is no difference between the state of the database pre/post downgrade.

    In essence this is a much more strict version of `test_up_down_consistency`, where the state of a MetaData before and after a downgrade are identical as far as alembic (autogenerate) is concerned.

  These tests will need to be enabled manually because their semantics or API are not yet guaranteed to stay the same. See the linked docs for more details!

Let us know if you have any ideas for more built-in tests which would be generally useful for most alembic histories!

1.2.2 Custom Tests

For more information, see the docs for custom tests (example below) or custom static data (to be inserted automatically before a given revision).

Sometimes when writing a particularly gnarly data migration, it helps to be able to practice a little timely TDD, since there's always the potential you'll trash your actual production data.

With `pytest-alembic`, you can write tests directly, in the same way that you would normally, through the use of the `alembic_runner` fixture.

```python
def test_gnarly_migration_xyz123(alembic_engine, alembic_runner):
    # Migrate up to, but not including this new migration
    alembic_runner.migrate_up_before('xyz123')

    # Perform some very specific data setup, because this migration is sooooo complex.
    # ...
    alembic_engine.execute(table.insert(id=1, name='foo'))

    alembic_runner.migrate_up_one()
```

`alembic_runner` has a number of methods designed to make it convenient to change the state of your database up, down, and all around.

1.3 Installing

```
pip install "pytest-alembic"
```

CHAPTER

TWO

SETUP

2.1 env.py

The default `env.py` file that alembic will autogenerate for you includes a snippet like so:

```
def run_migrations_online():
    connectable = engine_from_config(
        config.get_section(config.config_ini_section),
        prefix="sqlalchemy.",
        poolclass=pool.NullPool,
    )
```

This is fine, but `pytest-alembic` needs to provide alembic with a connection at runtime. So to allow us to produce that connection in a way that `env.py` understands, modify the above snippet to resemble:

```
def run_migrations_online():
    connectable = context.config.attributes.get("connection", None)

    if connectable is None:
        connectable = engine_from_config(
            context.config.get_section(context.config.config_ini_section),
            prefix="sqlalchemy.",
            poolclass=pool.NullPool,
        )
```

2.1.1 Caplog Issues

The default `env.py` file that alembic will autogenerate for you also includes a call to `logging.config.fileConfig()`. Given that alembic tests invoke the `env.py`, and `logging.config.fileConfig()` has a default argument of `disable_existing_loggers=True`, this can inadvertently break tests which use pytest's `caplog` fixture.

To fix this, simply provide `disable_existing_loggers=False` to `fileConfig`.

Warning: Additionally, if you are a user of `logging.basicConfig()`, note that `logging.basicConfig()` "does nothing if the root logger already has handlers configured", (which is why we generally try to avoid `basicConfig`) and may cause issues for similar reasons.

Note: Python 3.8 added a `force=True` keyword to `logging.basicConfig()`, which makes it somewhat less hazardous to use.

2.1.2 Optional but helpful additions

Alembic comes with a number of other options to customize how the autogeneration of revisions is handled, but most of them are disabled by default. There are many good reasons your particular migrations might **not** want some of these options enabled; but if they don't apply to your setup, we think they increase the quality of the safety this library helps to provide.

Further down in your `env.py`, you'll see a configure block.

```
with connectable.connect() as connection:
    context.configure(
        connection=connection,
        target_metadata=target_metadata,
        # This is where we want to add more options!
    )

    with context.begin_transaction():
        context.run_migrations()
```

Consider enabling the following options:

- `compare_type=True`: Indicates type comparison behavior during an autogenerate operation.
- `compare_server_default=True`: Indicates server default comparison behavior during an autogenerate operation.
- `include_schemas=True`: If True, autogenerate will scan across all schemas located by the SQLAlchemy get_schema_names() method, and include all differences in tables found across all those schemas. This may only be useful if you make use of schemas.

2.2 Setting up Fixtures

We expose 2 explicitly overridable fixtures `alembic_config` and `alembic_engine`.

One should generally put the implementations of *alembic_config* and *alembic_engine* in a `conftest.py` (a special file recognized by pytest) at the root of your tests folder, typically `tests/conftest.py`.

If your tests are located elsewhere, you should use the *pytest config* to specify `pytest_alembic_tests_path` (defaults to `tests/conftest.py`), to point at your tests folder root.

Then you can define your own implementations of these fixtures

2.2.1 Setting up `alembic_config`

alembic_config is the primary point of entry for configurable options for the alembic runner. See the *API* reference for a comprehensive list. This fixture can often be omitted though, if your use of alembic is straightforward and/or uses alembic defaults.

The default implementation is:

```
from pytest_alembic.config import Config

@pytest.fixture
def alembic_config():
    """Override this fixture to configure the exact alembic context setup required.
    """
    return Config()
```

See *Config* for more details about the sort of options available on our config.

2.2.2 Setting up `alembic_engine`

alembic_engine is where you specify the engine with which the *alembic_runner* should execute your tests.

The default `alembic_engine` implementation is:

```
@pytest.fixture
def alembic_engine():
    """Override this fixture to provide pytest-alembic powered tests with a database␣
↪handle.
    """
    return sqlalchemy.create_engine("sqlite:///")
```

If you have a **very** simple database schema, you **may** be able to get away with the default fixture implementation, which uses an in-memory SQLite engine. In most cases however, SQLite will not be able to sufficiently model your migrations. Typically, DDL is where features of databases tend to differ the most, and so the **actual** database you, should likely be what your `alembic_engine` is.

2.2.3 Pytest Mock Resources

Our recommended approach is to use pytest-mock-resources, another library we have open sourced which uses Docker to manage the lifecycle of an ephemeral database instance.

This library is what `pytest-alembic` internally uses, so it's the strategy we can most easily guarantee should work.

If you use Postgres, MySQL, Redshift, or SQLite (or a database which reacts sufficiently closely) `pytest-mock-resources` can support your usecase today. For other alembic-supported databases, file an issue!

```
from pytest_mock_resources import create_postgres_fixture

alembic_engine = create_postgres_fixture()
```

2.2.4 `alembic_engine` Invariants

Note: Depending on what you want, and how your code internally produces/consumes engines there is plenty of flexibility in how `pytest-alembic` test engines interact with your own.

For example (using `pytest-mock-resources`), you can ensure that there's no interdependence between this engine and the one used by your own tests:

```
from pytest_mock_resources import create_postgres_fixture

pg = create_postgres_fixture()
alembic_engine = create_postgres_fixture()

def test_foo(pg, alembic_engine):  # two unique databases
    ...
```

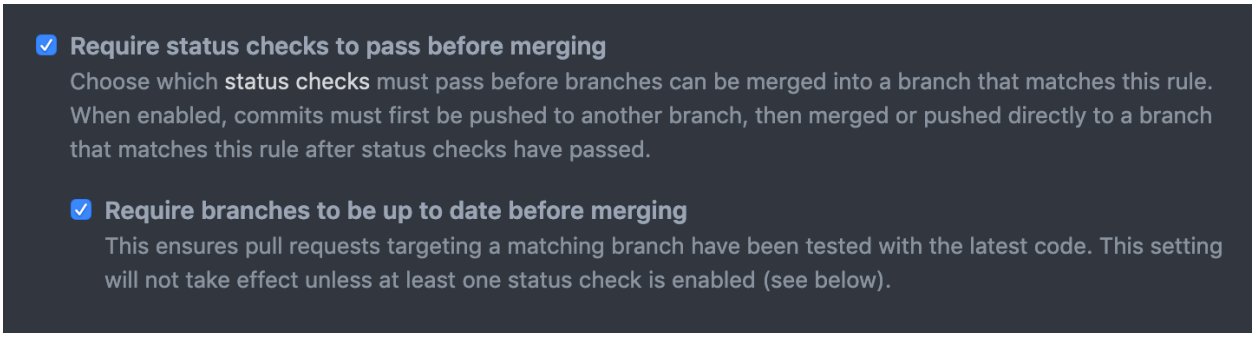
Or if you would **prefer** them to be the same, you could instead do:

```
import pytest
from pytest_mock_resources import create_postgres_fixture

pg = create_postgres_fixture()

@pytest.fixture
def alembic_engine(pg):
    return pg

def test_foo(pg, alembic_engine):
    assert pg is alembic_engine  # they're literally the same
    ...
```

Of course, you can implement whatever strategy you want. However there are a few invariants that an `alembic_engine` fixture should follow, to ensure that tests reliably pass and to avoid inter-test state issues.

1. The engine should point to a database that must be empty. It is out of scope for pytest-alembic to manage the database state.
2. You should not expect to be able to roll back changes made by these tests. Alembic will internally perform commits, as do certain `pytest-alembic` features. Alembic is **literally** being invoked in the same way you would normally run migrations, so it's exactly as permanent.
3. The yielded engine should not be inside a (manually created) transaction. The engine is configured into Alembic itself, Alembic internals perform commits, and it will almost certainly not work if you try to manage transactional state around Alembic.

2.3 Git(hub) Settings



We highly recommend you enable "Require branches to be up to date before merging" on repos which have alembic migrations!

While this will require that people merging PRs to rebase on top of master before merging (which we think is ideal for ensuring your build is always green anyways), it guarantees that **our** tests are running against a known up-to-date migration history.

Without this option it is trivially easy to end up with an alembic version history with 2 or more heads which needs to be manually resolved.

Provider support

- Only GitLab EE supports an approximate option to GitHub's.
- Only Bitbucket EE supports an approximate option to GitHub's.

CHAPTER

THREE

RUNNING TESTS

You have two primary options for running the configured set of tests:

1. Automatically at the command-line via `--test-alembic`

   `Pytest Alembic` automatically adds a flag, `pytest --test-alembic`, which will automatically invoke the baked-in tests.

   This can be convenient if you want to exclude migrations tests most of the time, but include them for e.g. CI. By default, `pytest tests` would then, **not** run migrations tests.

   Additionally, it means you don't need to manually include the tests in a test file somewhere in your project.

   If your tests dont generally reside at/below a `tests/` directory with a `tests/conftest.py` file, you can/should set the `pytest_alembic_tests_path` option, described below.

2. You can directly import the tests you want to include at any point in your project.

   Listing 1: tests/test_migrations.py

   ```
   from pytest_alembic.tests import (
       test_model_definitions_match_ddl,
       test_single_head_revision,
       test_up_down_consistency,
       test_upgrade,
   )
   ```

   This can be convenient if you always want the migrations tests to run, or else want a reference to the tests' existence somewhere in your source code. Pytest would automatically include the tests every time you run i.e. `pytest tests`.

In either case, you can exclude migrations tests using pytest's "marker" system, i.e. `pytest -m "not alembic"`.

3.1 Configuration

3.1.1 Pytest Config

In any of the pytest config locations (`pytest.ini`, `setup.cfg`, `pyproject.toml`), you can set any of the following configuration options to alter global pytest-alembic behavior.

- `pytest_alembic_include`

  List of built-in tests to include. If specified, 'pytest_alembic_exclude' is ignored. If both are omitted, all tests are included. The tests should be listed as a comma delimited string containing the tests' names.

- `pytest_alembic_exclude`

  List of built-in tests to exclude. Ignored if 'pytest_alembic_include' is specified. The tests should be listed as a comma delimited string containing the tests' names.

- `pytest_alembic_tests_folder`

  The location under which the built-in tests will be bound. This defaults to 'tests/' (the tests themselves then being executed from tests/pytest_alembic/), the typical test location. However this can be customized if pytest is, for example, invoked from a parent directory like `pytest folder/tests`, or the tests are otherwise located at a different location, relative to the `pytest` invocation.

  Note: As of pytest-alembic version 0.8.5, this option is ignored. Instead, if you require customizing the registration location, you should use `pytest_alembic_tests_path` instead.

- `pytest_alembic_tests_path`

  Note: Introduced in v0.10.1.

  The location at which the built-in tests will be bound. This defaults to 'tests/conftest.py'. Typically, you would want this to coincide with the path at which your *alembic_engine* is being defined/registered. Note that this path must be the full path, relative to the root location at which pytest is being invoked.

  This option has replaced `pytest_alembic_tests_folder` due to changes in how pytest test collection needed to be performed in around pytest ~7.0.

  Additionally, this option is only required if you are using the `--test-alembic` flag.

3.1.2 Alembic Config

See the *Config* fixture for more detail.

3.2 Alternative to `--test-alembic`

There is **some** magic to the automatic inclusion of the built-in tests. It's not obvious, from looking at any of the test code, that these tests (sometimes) magically be included.

Also, one may want to include the built-in tests automatically, every time, without needing to specify `--test-alembic`, or by doing so conditionally in-code.

Whatever the reason, it is possible to simply import the test implementations from `pytest_alembic` directly.

Simply import the tests at whatever location you want tests to be included:

Listing 2: tests/test_migrations.py

```
from pytest_alembic.tests import test_single_head_revision
from pytest_alembic.tests import test_upgrade
from pytest_alembic.tests import test_model_definitions_match_ddl
from pytest_alembic.tests import test_up_down_consistency
```

Furthermore, doing this as well as using `--test-alembic` will cause the tests to be run twice (since they'd be considered unique tests with different paths). So generally, these methods should be considered mutually exclusive.

3.3 Multiple Alembic Histories

It may be the case that you have the histories for two separate databases (or schemas) in a single project. How should you structure your tests?

This is likely one of the times you want to avoid the use of the `--test-alembic` flag and the automatic insertion of tests.

Instead, you'll likely want to want to make use of `create_alembic_fixture()`.

```
from pytest_alembic import tests, create_alembic_fixture

# The argument here represents the equivalent to `alembic_config`. Depending
# on your setup, this may be configuring the "file" argument, "script_location",
# or some other way of configuring one or the other of your histories.
history_1 = create_alembic_fixture({"file": "alembic.ini"})

def test_single_head_revision_history_1(history_1):
    tests.test_single_head_revision(history_1)

def test_upgrade_history_1(history_1):
    tests.test_upgrade(history_1)

def test_model_definitions_match_ddl_history_1(history_1):
    tests.test_model_definitions_match_ddl(history_1)

def test_up_down_consistency_history_1(history_1):
    tests.test_up_down_consistency(history_1)

# The 2nd fixture, and the 2nd set of tests.
history_2 = create_alembic_fixture({"file": "history_2.ini"})

def test_single_head_revision_history_2(history_2):
    tests.test_single_head_revision(history_2)

def test_upgrade_history_2(history_2):
    tests.test_upgrade(history_2)

def test_model_definitions_match_ddl_history_2(history_2):
    tests.test_model_definitions_match_ddl(history_2)

def test_up_down_consistency_history_2(history_2):
    tests.test_up_down_consistency(history_2)
```

Due to limitations of how pytest test collection occurs, there's currently no obvious way to automatically set up and define these tests to occur against different fixtures.

3.4 Pytest Marks

Pytest-alembic automatically marks all tests which use the `alembic_runner` fixture (including all built-in tests) with the `alembic` mark.

This means you can optionally include/exclude migrations tests using the vanilla pytest mark machinery like so:

```
pytest -m 'alembic'  # Run *only* alembic tests
pytest -m 'not alembic'  # Run everything *except* alembic tests
```

CHAPTER

FOUR

CUSTOM DATA

To preempt the need to write an explicit test for every new migration, there is a mechanism built into the config to automatically insert data into the database before or upon reaching a given revision.

To that end:

```
@pytest.fixture
def alembic_config():
    return {
        "before_revision_data": {
            "fb4d3ab5f38d": [
                {"__tablename__": "foo", "id": 1, "name": "foo"},
                {"__tablename__": "bar", "id": 1, "name": "bar"},
            ],
        },
        "at_revision_data": {
            "a5a9ccc4e535": {
                "__tablename__": "foo",
                "id": 1,
                "name": "foo2",
            },
        },
    }
```

There are two, similiar but distinct options. `before_revision_data`, and `at_revision_data`.

- `before_revision_data` will insert the given data for all upgrade commands before performing the upgrade to the given revision.
- `at_revision_data` will insert the given data for all upgrade commands after having performed the upgrade to the given revision.

To be clear, you can use either of them to describe the same operation. But depending on the circumstance (if for example, you are testing a real-world situation that you are trying to semantically model more closely), one or the other option may be more appropriate.

4.1 Schema

The value for either `before_revision_data` or `at_revision_data`, should be a `dict()` where the keys are the revision for which the data is being described.

The value can either be a `dict()` (single row), or a `list()` of `dict()` (multiple rows). `__tablename__` is a special key which tells us the name of the table to insert the data into, and the rest of the spec should describe the columns and the column data for that row, similar to what you might do for a `table.insert().values(...)` call.

Alternatively, you can directly import and instantiate a `RevisionSpec` and set that as the value to either `before_revision_data` or `at_revision_data`.

4.2 Example

Given the above data, and history of:

```
23256c7bf855 > fb4d3ab5f38d -> a5a9ccc4e535 -> b835ae9ff1d1
```

We would:

- Upgrade to 23256c7bf855
- Insert `{"__tablename__": "foo", "id": 1, "name": "foo"}` and `{"__tablename__": "bar", "id": 1, "name": "bar"}`
- Upgrade to fb4d3ab5f38d
- Upgrade to a5a9ccc4e535
- Insert `{"__tablename__": "foo", "id": 1, "name": "foo2"}`
- Upgrade to b835ae9ff1d1

CHAPTER

FIVE

WRITING CUSTOM TESTS

Honestly, there's not much to it by this point!

```
from sqlalchemy import text

def test_gnarly_migration_xyz123(alembic_runner, alembic_engine):
    # Migrate up to, but not including this new migration
    alembic_runner.migrate_up_before('xyz123')

    # Perform some very specific data setup, because this migration is sooooo complex.
    # ...
    alembic_runner.insert_into('tablename', dict(id=1, name='foo'))
    # Or you can optionally accept the `alembic_engine` fixture, which is a
    # sqlalchemy engine object, with which you can do whatever setup you'd like.

    alembic_runner.migrate_up_one()

    with alembic_engine.connect() as conn:
        rows = conn.execute(text("SELECT id from foo")).fetchall()

    assert rows == [(1,)]
```

`alembic_runner` has all sorts of convenience methods for altering the state of the database for your test:

class pytest_alembic.runner.**MigrationContext**(*command_executor*, *revision_data*, *connection_executor*, *history*, *config*)

Within a given environment/execution context, executes alembic commands.

property current: `str`

Get the list of revision heads.

generate_revision(*process_revision_directives=None*, *prevent_file_generation=True*, *autogenerate=False*, ***kwargs*)

Generate a test revision.

If *prevent_file_generation* is *True*, the final act of this process raises a *RevisionSuccess*, which is used as a sentinal to indicate the revision was generated successfully, while not actually finishing the generation of the revision file on disk.

property heads: `List[str]`

Get the list of revision heads.

Result is cached for the lifetime of the *MigrationContext*.

insert_into(*table*, *data=None*, *revision=None*)

Insert data into a given table.

Parameters

- **table** (Optional[str]) – The name of the table to insert data into
- **data** (Union[Dict, List, None]) – The data to insert. This is eventually passed through to SQLAlchemy's Table class *values* method, and so should accept either a list of *dict`s representing a list of rows, or a `dict* representing one row.
- **revision** – The revision of MetaData to use as the table definition for the insert.

managed_downgrade(*dest_revision*)

Perform an downgrade, one migration at a time.

managed_upgrade(*dest_revision*)

Perform an upgrade, one migration at a time, inserting static data at the given points.

migrate_down_before(*revision*)

Migrate down to, but not including the given *revision*.

migrate_down_one()

Migrate down by exactly one revision.

migrate_down_to(*revision*)

Migrate down to, and including the given *revision*.

migrate_up_before(*revision*)

Migrate up to, but not including the given *revision*.

migrate_up_one()

Migrate up by exactly one revision.

migrate_up_to(*revision*)

Migrate up to, and including the given *revision*.

raw_command(**args*, ***kwargs*)

Execute a raw alembic command.

refresh_history()

Refresh the context's version of the alembic history.

Note this is not done automatically to avoid the expensive reevaluation step which can make long histories take seconds longer to evaluate for each test.

Return type
AlembicHistory

roundtrip_next_revision()

Upgrade, downgrade then upgrade.

This is meant to ensure that the given revision is idempotent.

table_at_revision(*name*, ***, *revision=None*, *schema=None*)

Return a reference to a *sqlalchemy.Table* at the given revision.

Parameters

- **name** – The name of the table to produce a *sqlalchemy.Table* for.
- **revision** – The revision of the table to return.

- **schema** – The schema of the table.

CHAPTER

SIX

EXPERIMENTAL TESTS

Note: Experimental tests may be moved to the default test section at some point. As that time their name would be changed, and the old name will become deprecated (though for a time, just as a deprecation warning).

6.1 test_all_models_register_on_metadata

Diffs the set of tables registered by alembic's `env.py` versus the set of full tables we find throughout your models package/module.

6.1.1 Enabling all_models_register_on_metadata (TL;DR)

You can either enable this test with no configuration, which will attempt to identify the source module from which the `env.py` is loading its `MetaData` and automatically search in that module/package

Listing 1: pyproject.toml/setup.cfg/pytest.ini

```
# pyproject.toml
[tool.pytest.ini_options]
pytest_alembic_include_experimental = 'all_models_register_on_metadata'

# or setup.cfg/pytest.ini
[pytest]
pytest_alembic_include_experimental = all_models_register_on_metadata
```

Or you can manually import and execute the test somewhere in your own tests. Using this mechanism, you would be able to circumvent the automatic detection and provide the module/package directly.

```
from pytest_alembic import tests

def test_all_models_register_on_metadata(alembic_runner):
   tests.experimental.test_all_models_register_on_metadata(alembic_runner, 'package.
↪models')
```

6.1.2 How all_models_register_on_metadata works

The problem this test attempts to solve is best described with an example. Consider the following package structure:

```
package/
  models/
     __init__.py
     foo.py
     bar.py
     baz.py
  other_packages/
  other_modules.py

migrations/
   env.py
```

Next, a typical package containing a `MetaData` or `declarative_base` and models or tables. Yours may look superficially different than ours, but you will almost certainly define your base, and either define or import any models or tables after its definition.

Listing 2: __init__.py

```
import sqlalchemy
from sqlalchemy import Column, types
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

from package.models import (
   foo,
   bar,
)
```

The specifics of the table definitions are not particularly important, so we'll omit `bar.py` and `baz.py` (imagine they're essentially identical!), but here's `foo.py`.

Listing 3: foo.py

```
from package.models import Base

class Foo(Base):
 __tablename__ = "foo"

 id = Column(types.Integer(), autoincrement=True, primary_key=True)
```

Finally, an excerpt from what is commonly autogenerated by running `alembic init`.

Listing 4: env.py

```
...
from package.models import Base
target_metadata = Base.metadata
...
with connectable.connect() as connection:
```

(continues on next page)

(continued from previous page)

```
    context.configure(connection=connection, target_metadata=target_metadata)
    ...
```

And now we get to the crux of the problem.

A keen eye may have noticed that `baz` is not being imported above, and that's not a mistake! Elsewhere in your code (`other_packages/other_modules`, for example) you will likely import all of your models at **some** point. So when you go to actually use the models, you may not even notice that there is anything wrong.

However as far as alembic is concerned:

- It will load the `env.py`
- `env.py` only imports `package.models` (which notably omits `package.models.baz`!)
- `Base`/`Base.metadata` will therefore only have `foo` and `bar` tables registered on it.

So when you go to run `alembic revision --autogenerate`, it will be unaware of the "baz" table and either omit its creation or suggest it be dropped if you had already created it.

This test is meant to be a lint against such scenarios and will fail in any case where there is no direct import of any tables defined on a *MetaData* during the course of executing the `env.py` through alembic.

Note: The original inspiration for this test was actually a refactor which changed some pre-existing imports around.

This lead to an **already created** table no longer being incidentally imported (somewhere **else** in the codebase!) during the normal course of importing our equivalent of `package.models`.

This immediately resulted in an `--autogenerate` suggesting that the table be dropped, since it was alembic assumes you've deleted the model entirely!

6.2 test_downgrade_leaves_no_trace

Attempts to ensure that the downgrade for every migration precisely undoes the changes performed in the upgrade.

6.2.1 Enabling downgrade_leaves_no_trace (TL;DR)

Listing 5: pyproject.toml/setup.cfg/pytest.ini

```
# pyproject.toml
[tool.pytest.ini_options]
pytest_alembic_include_experimental = 'downgrade_leaves_no_trace'

# or setup.cfg/pytest.ini
[pytest]
pytest_alembic_include_experimental = downgrade_leaves_no_trace
```

Or you can manually import and execute the test somewhere in your own tests. Using this mechanism, you would be able to circumvent the automatic detection and provide the module/package directly.

```
from pytest_alembic import tests

def test_downgrade_leaves_no_trace(alembic_runner):
   tests.experimental.test_downgrade_leaves_no_trace(alembic_runner)
```

6.2.2 How downgrade_leaves_no_trace works

This test works by attempting to produce two autogenerated migrations.

1. The first is the comparison between the original state of the database before the given migration's upgrade occurs, and the *MetaData* produced by having performed the upgrade.

   This should approximate the autogenerated migration that alembic would have generated to produce your upgraded database state itself.

2. The 2nd is the comparison between the state of the database after having performed the upgrade -> downgrade cycle for this revision, and the same *MetaData* used in the first comparison.

   This should approximate what alembic would have autogenerated if you **actual** performed the downgrade on your database.

In the event these two autogenerations do not match, it implies that your upgrade -> downgrade cycle produces a database state which is different (enough for alembic to detect) from the state of the database without having performed the migration at all.

Note: This isn't perfect! Alembic autogeneration will not detect many kinds of changes! If you encounter some scenario in which this does not detect a change you'd expect it to, alembic already has extensive ability to customize and extend the autogeneration capabilities.

CHAPTER

SEVEN

ASYNCIO

Support for asyncio is largely built on top of the Alembic Cookbook example, inlined here for posterity:

```
import asyncio

# ... no change required to the rest of the code

def do_run_migrations(connection):
    context.configure(connection=connection, target_metadata=target_metadata)

    with context.begin_transaction():
        context.run_migrations()


async def run_migrations_online():
    """Run migrations in 'online' mode.

    In this scenario we need to create an Engine
    and associate a connection with the context.

    """
    connectable = AsyncEngine(
        engine_from_config(
            config.get_section(config.config_ini_section),
            prefix="sqlalchemy.",
            poolclass=pool.NullPool,
            future=True,
        )
    )

    async with connectable.connect() as connection:
        await connection.run_sync(do_run_migrations)

    await connectable.dispose()


if context.is_offline_mode():
    run_migrations_offline()
else:
    asyncio.run(run_migrations_online())
```

Note that this is a prerequisite for how one gets **alembic itself** to run with an async connection, when running `alembic`

commands interatively yourself.

At this point, you just need to make sure the `alembic_engine` fixture is producing a async engine. something like

```
from sqlalchemy import create_engine
from sqlalchemy.ext.asyncio import create_engine_async, AsyncEngine

@pytest.fixture
def alembic_engine(...):
    return create_async_engine(URL(...))


@pytest.fixture
def alembic_engine(...):
    engine = create_engine(URL(...))
    return AsyncEngine(engine)


# or, for example, with pytest-mock-resources
from pytest_mock_resources import create_postgres_fixture

alembic_engine = create_postgres_fixture(async_=True)
```

7.1 A slightly more versatile setup

The above `env.py` setup comes with a caveat. It assumes execution of the migrations solely through async. Due to the way sqlalchemy/alembic async works (as evidenced by even their suggested use of `run_sync`), this can be a problem.

For pytest-alembic the only such built in test is *test_downgrade_leaves_no_trace*. For compatibility with (majority) sync alembic use, it's implemented sychronously, and internally requires performing transaction manipulation which would otherwise require re-entrant use of `asyncio.run`.

If you don't use this test, and haven't implemented any of your own which encounter this issue, then feel free to stick with the official alembic suggestion. However a slight reorganization of their suggested setup allows for both sychronous and asynchronous execution of migrations, and thus fixes *test_downgrade_leaves_no_trace*.

```
from sqlalchemy.ext.asyncio.engine import AsyncEngine

def run_migrations_online():
    connectable = context.config.attributes.get("connection", None)

    if connectable is None:
        connectable = AsyncEngine(
            engine_from_config(
                context.config.get_section(context.config.config_ini_section),
                prefix="sqlalchemy.",
                poolclass=pool.NullPool,
                future=True,
            )
        )

    # Note, we decide whether to run asynchronously based on the kind of engine we're␣
↪dealing with.
```

(continues on next page)

(continued from previous page)

```
    if isinstance(connectable, AsyncEngine):
        asyncio.run(run_async_migrations(connectable))
    else:
        do_run_migrations(connectable)


# Then use their setup for async connection/running of the migration
async def run_async_migrations(connectable):
    async with connectable.connect() as connection:
        await connection.run_sync(do_run_migrations)

    await connectable.dispose()


def do_run_migrations(connection):
    context.configure(connection=connection, target_metadata=target_metadata)

    with context.begin_transaction():
        context.run_migrations()


# But the outer layer still allows sychronous execution also.
run_migrations_online()
```

CHAPTER

EIGHT

API

8.1 Fixtures

8.1.1 alembic_runner

pytest_alembic.plugin.fixtures.**alembic_runner**(*alembic_config*, *alembic_engine*)

Produce the primary alembic migration context in which to execute alembic tests.

This fixture allows authoring custom tests which are specific to your particular migration history.

Examples

```
>>> def test_specific_migration(alembic_runner):
...     alembic_runner.migrate_up_to('xxxxxxx')
...     assert ...
```

8.1.2 alembic_config

pytest_alembic.plugin.fixtures.**alembic_config**()

Override this fixture to configure the exact alembic context setup required.

The return value of this fixture can be one of a few types. :rtype: Union[Dict[str, Any], Config, *Config*]

- If you're only configuring alembic-native configuration, a `alembic.config.Config` object is accepted as configuration. This largely leaves pytest-alembic out of the setup, so depending on your settings, might be the way to go.
- If you only have a couple of options to set, you might choose to return a `Dict`.

  The following common alembic config options are accepted as keys.

  - file/config_file_name (commonly alembic.ini)
  - script_location
  - sqlalchemy.url
  - target_metadata
  - process_revision_directives
  - include_schemas

Additionally you can send a *file* key (akin to *alembic -c*), should your *alembic.ini* be otherwise named.

Note that values here, represent net-additive options on top of what you might already have configured in your *env.py*. You should generally prefer to configure your *env.py* however you like it and omit such options here.

You may also use this dict to set pytest-alembic specific features:

- before_revision_data
- at_revision_data
- minimum_downgrade_revision

- You can also directly return a *Config* class instance. This is your only option if you want to use both pytest-alembic specific features **and** construct your own `alembic.config.Config`.

Examples

```
>>> @pytest.fixture
... def alembic_config():
...     return {'file': 'migrations.ini'}
```

```
>>> @pytest.fixture
... def alembic_config():
...     alembic_config = alembic.config.Config()
...     alembic_config.set_main_option("script_location", ...)
...     return alembic_config
```

Config

class pytest_alembic.config.**Config**(*config_options=<factory>*, *alembic_config=None*, *before_revision_data=None*, *at_revision_data=None*, *minimum_downgrade_revision=None*, *skip_revisions=None*)

Pytest-alembic configuration options.

- ***config_options*: Meant to simplify the creation of `alembic.config.Config`**
  objects. Supply keys common to customization in alembic configuration. For example:
  - file/config_file_name (commonly alembic.ini)
  - script_location
  - sqlalchemy.url
  - target_metadata
  - process_revision_directives
  - include_schemas
- Both *before_revision_data* and *at_revision_data* are described in detail in *Custom data*.
- `minimum_downgrade_revision` can be used to set a lower bound on the **downgrade** migrations which are run built-in tests like `test_up_down_consistency` and `test_downgrade_leaves_no_trace`.
- `skip_revisions` can be used to avoid executing specific revisions, particularly if they are slow and you can guarantee to yourself that the difference in the resulting migrations history wont have a meaningful

effect. Note that skipping migrations can be "dangerous", because either DDL or data differences could lead to migrations which pass in tests, but fail in practice.

For example:

```
>>> import pytest
```

```
>>> @pytest.fixture
... def alembic_config():
...     return Config(minimum_downgrade_revision='abcde12345')
```

This would essentially short-circuit and avoid running the downgrade migrations **including and below** this migration.

Note: If a downgrade raises a `NotImplementedError`, it will have the same effect as a `minimum_downgrade_revision`, but will emit a warning suggesting the use of this feature instead.

---

classmethod from_raw_config(*raw_config=None*)

Adapt between pre-produced alembic config and raw config options.

Allows one to specify raw pytest-alembic config options through raw dictionary, as well as being flexible enough to allow a literal alembic Config object.

Examples

```
>>> Config.from_raw_config()
Config(config_options={}, alembic_config=None, before_revision_data=None, at_
↪revision_data=None, minimum_downgrade_revision=None, skip_revisions=None)
```

```
>>> Config.from_raw_config({'minimum_downgrade_revision': 'abc123'})
Config(config_options={}, alembic_config=None, before_revision_data=None, at_
↪revision_data=None, minimum_downgrade_revision='abc123', skip_revisions=None)
```

```
>>> Config.from_raw_config(Config(minimum_downgrade_revision='abc123'))
Config(config_options={}, alembic_config=None, before_revision_data=None, at_
↪revision_data=None, minimum_downgrade_revision='abc123', skip_revisions=None)
```

8.1.3 alembic_engine

pytest_alembic.plugin.fixtures.**alembic_engine**()

Override this fixture to provide pytest-alembic powered tests with a database handle.

8.1.4 create_alembic_fixture

pytest_alembic.plugin.fixtures.**create_alembic_fixture**(*raw_config=None*)

Create a new fixture *alembic_runner*-like fixture.

In many cases, this function should not be strictly necessary. You **can** generally rely solely on the `--test-alembic` flag, automatic insertion of tests, and the *alembic_runner()* fixture.

However this may be useful in some situations:

- If you would generally prefer to avoid the `--test-alembic` flag and automatic test insertion, this is the function for you!
- If you have multiple alembic histories and therefore require more than one fixture, you will **minimally** need to use this for the 2nd history (if not both)

Examples

```
>>> from pytest_alembic import tests
>>>
>>> alembic = create_alembic_fixture()
>>>
>>> def test_upgrade_head(alembic):
...     tests.test_upgrade_head(alembic)
>>>
>>> def test_specific_migration(alembic):
...     alembic_runner.migrate_up_to('xxxxxxx')
...     assert ...
```

Config can also be supplied similarly to the *alembic_config()* fixture.

```
>>> alembic = create_alembic_fixture({'file': 'migrations.ini'})
```

8.2 Alembic Runner

The object yielded into a test from an *alembic_runner* fixture is the `MigrationContext`

class pytest_alembic.runner.**MigrationContext**(*command_executor*, *revision_data*, *connection_executor*, *history*, *config*)

Within a given environment/execution context, executes alembic commands.

property current: str

Get the list of revision heads.

generate_revision(*process_revision_directives=None*, *prevent_file_generation=True*, *autogenerate=False*, ***kwargs*)

Generate a test revision.

If *prevent_file_generation* is *True*, the final act of this process raises a *RevisionSuccess*, which is used as a sentinal to indicate the revision was generated successfully, while not actually finishing the generation of the revision file on disk.

property heads: List[str]

Get the list of revision heads.

Result is cached for the lifetime of the *MigrationContext*.

insert_into(*table*, *data=None*, *revision=None*)

Insert data into a given table.

Parameters

- **table** (Optional[str]) – The name of the table to insert data into
- **data** (Union[Dict, List, None]) – The data to insert. This is eventually passed through to SQLAlchemy's Table class *values* method, and so should accept either a list of *dict`s representing a list of rows, or a `dict* representing one row.
- **revision** – The revision of MetaData to use as the table definition for the insert.

managed_downgrade(*dest_revision*)

Perform an downgrade, one migration at a time.

managed_upgrade(*dest_revision*)

Perform an upgrade, one migration at a time, inserting static data at the given points.

migrate_down_before(*revision*)

Migrate down to, but not including the given *revision*.

migrate_down_one()

Migrate down by exactly one revision.

migrate_down_to(*revision*)

Migrate down to, and including the given *revision*.

migrate_up_before(*revision*)

Migrate up to, but not including the given *revision*.

migrate_up_one()

Migrate up by exactly one revision.

migrate_up_to(*revision*)

Migrate up to, and including the given *revision*.

raw_command(**args*, ***kwargs*)

Execute a raw alembic command.

refresh_history()

Refresh the context's version of the alembic history.

Note this is not done automatically to avoid the expensive reevaluation step which can make long histories take seconds longer to evaluate for each test.

Return type
AlembicHistory

roundtrip_next_revision()

Upgrade, downgrade then upgrade.

This is meant to ensure that the given revision is idempotent.

table_at_revision(*name*, ***, *revision=None*, *schema=None*)

Return a reference to a *sqlalchemy.Table* at the given revision.

Parameters

- **name** – The name of the table to produce a *sqlalchemy.Table* for.
- **revision** – The revision of the table to return.
- **schema** – The schema of the table.

class pytest_alembic.history.**AlembicHistory**(*map*, *revisions*, *revision_indices*, *revisions_by_index*)

classmethod parse(*revision_map*)

Extract the set of migration revision hashes from alembic's notion of the history.

Return type
AlembicHistory

class pytest_alembic.revision_data.**RevisionData**(*before_revision_data*, *at_revision_data*)

Describe the data which should exist at given revisions when performing upgrades.

classmethod from_config(*config*)

Produce a *RevisionData* from raw configuration from `alembic_config()`.

get_at(*revision*)

Yield the individual data insertions which should occur upon reaching the given revision.

Return type
Union[Dict, List[Dict]]

get_before(*revision*)

Yield the individual data insertions which should occur before the given revision.

Return type
Union[Dict, List[Dict]]

class pytest_alembic.revision_data.**RevisionSpec**(*data*)

Describe a set of valid database data at a set of revisions.

get(*revision*)

Get the database data described at a particular revision.

Return type
Union[Dict, List[Dict]]

classmethod parse(*data*)

Parse a raw dict structure into a *RevisionSpec*.

CHAPTER

NINE

CONTRIBUTING

9.1 Prerequisites

If you are not already familiar with Poetry, this is a poetry project, so you'll need this!

9.2 Getting Setup

Note, while the project itself provisionally runs on python 3.6, test dependencies including pytest-mock-resources, coverage, and black, have minimum python versions of 3.7. So local development of pytest-alembic itself requires. Additionally this means we dont test 3.6 support, so supporting it is best effort until it becomes inconvenient.

See the `Makefile` for common commands, but for some basic setup:

```
# Installs the package with all the extras
make install
```

And you'll want to make sure you can run the tests and linters successfully:

```
# Runs CI-level tests, with coverage reports
make test lint
```

9.3 Need help

Submit an issue!

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

T