Tutorial

Contents

Alembic provides for the creation, management, and invocation of *change management* scripts for a relational database, using SQLAlchemy as the underlying engine. This tutorial will provide a full introduction to the theory and usage of this tool.

To begin, make sure Alembic is installed as described at Installation. As stated in the linked document, it is usually preferable that Alembic is installed in the **same module / Python path as that of the target project**, usually using a Python virtual environment, so that when the `alembic` command is run, the Python script which is invoked by `alembic`, namely your project's `env.py` script, will have access to your application's models. This is not strictly necessary in all cases, however in the vast majority of cases is usually preferred.

The tutorial below assumes the `alembic` command line utility is present in the local path and when invoked, will have access to the same Python module environment as that of the target project.

The Migration Environment

Usage of Alembic starts with creation of the *Migration Environment*. This is a directory of scripts that is specific to a particular application. The migration environment is created just once, and is then maintained along with the application's source code itself. The environment is created using the `init` command of Alembic, and is then customizable to suit the specific needs of the application.

The structure of this environment, including some generated migration scripts, looks like:

```
yourproject/
    alembic/
        env.py
        README
        script.py.mako
        versions/
            3512b954651e_add_account.py
            2b1ae634e5cd_add_order_id.py
            3adcc9a56557_rename_username_field.py
```

The directory includes these directories/files:

- `yourproject` - this is the root of your application's source code, or some directory within it.
- `alembic` - this directory lives within your application's source tree and is the home of the migration environment. It can be named anything, and a project that uses multiple databases may even have more than one.
- `env.py` - This is a Python script that is run whenever the alembic migration tool is invoked. At the very least, it contains instructions to configure and generate a SQLAlchemy engine, procure a connection from that engine along with a transaction, and then invoke the migration engine, using the connection as a source of database connectivity.

  The `env.py` script is part of the generated environment so that the way migrations run is entirely customizable. The exact specifics of how to connect are here, as well as the specifics of how the migration environment are invoked. The script can be modified so that multiple engines can be operated upon, custom arguments can be passed into the migration environment, application-specific libraries and models can be loaded in and made available.

  Alembic includes a set of initialization templates which feature different varieties of `env.py` for different use cases.

- `README` - included with the various environment templates, should have something informative.
- `script.py.mako` - This is a Mako template file which is used to generate new migration scripts. Whatever is here is used to generate new files within `versions/`. This is scriptable so that the structure of each migration file can be controlled, including standard imports to be within each, as well as changes to the structure of the `upgrade()` and `downgrade()` functions. For example, the `multidb` environment allows for multiple functions to be generated using a naming scheme `upgrade_engine1()`, `upgrade_engine2()`.
- `versions/` - This directory holds the individual version scripts. Users of other migration tools may notice that the files here don't use ascending integers, and instead use a partial GUID approach. In Alembic, the ordering of version scripts is relative to directives within the scripts themselves, and it is theoretically possible to "splice" version files in between others, allowing migration sequences from different branches to be merged, albeit carefully by hand.

Creating an Environment

With a basic understanding of what the environment is, we can create one using `alembic init`. This will create an environment using the "generic" template:

```
$ cd /path/to/yourproject
$ source /path/to/yourproject/.venv/bin/activate   # assuming a local virtualenv
$ alembic init alembic
```

Where above, the `init` command was called to generate a migrations directory called `alembic`:

```
Creating directory /path/to/yourproject/alembic...done
Creating directory /path/to/yourproject/alembic/versions...done
Generating /path/to/yourproject/alembic.ini...done
Generating /path/to/yourproject/alembic/env.py...done
Generating /path/to/yourproject/alembic/README...done
Generating /path/to/yourproject/alembic/script.py.mako...done
Please edit configuration/connection/logging settings in
'/path/to/yourproject/alembic.ini' before proceeding.
```

Alembic also includes other environment templates. These can be listed out using the `list_templates` command:

```
$ alembic list_templates
Available templates:

generic - Generic single-database configuration.
async - Generic single-database configuration with an async dbapi.
multidb - Rudimentary multi-database configuration.

Templates are used via the 'init' command, e.g.:

  alembic init --template generic ./scripts
```

Changed in version 1.8: The “pylons” environment template has been removed.

Editing the .ini File

Alembic placed a file `alembic.ini` into the current directory. This is a file that the `alembic` script looks for when invoked. This file can exist in a different directory, with the location to it specified by either the `--config` option for the `alembic` runner or the `ALEMBIC_CONFIG` environment variable (the former takes precedence).

The file generated with the “generic” configuration looks like:

```
# A generic, single database configuration.

[alembic]
# path to migration scripts
script_location = alembic

# template used to generate migration file names; The default value is %%(rev)s_%%(s
# Uncomment the line below if you want the files to be prepended with date and time
# file_template = %%(year)d_%%(month).2d_%%(day).2d_%%(hour).2d%%(minute).2d-%%(rev)

# sys.path path, will be prepended to sys.path if present.
# defaults to the current working directory.
prepend_sys_path = .

# timezone to use when rendering the date within the migration file
# as well as the filename.
# If specified, requires the python>=3.9 or backports.zoneinfo library.
# Any required deps can installed by adding `alembic[tz]` to the pip requirements
# string value is passed to ZoneInfo()
# leave blank for localtime
# timezone =

# max length of characters to apply to the
# "slug" field
# truncate_slug_length = 40

# set to 'true' to run the environment during
# the 'revision' command, regardless of autogenerate
# revision_environment = false

# set to 'true' to allow .pyc and .pyo files without
# a source .py file to be detected as revisions in the
# versions/ directory
# sourceless = false

# version location specification; This defaults
# to ${script_location}/versions.  When using multiple version
# directories, initial revisions must be specified with --version-path.
# The path separator used here should be the separator specified by "version_path_se
# version_locations = %(here)s/bar:%(here)s/bat:${script_location}/versions

# version path separator; As mentioned above, this is the character used to split
# version_locations. The default within new alembic.ini files is "os", which uses os
# If this key is omitted entirely, it falls back to the legacy behavior of splitting
# Valid values for version_path_separator are:
#
# version_path_separator = :
# version_path_separator = ;
# version_path_separator = space
# version_path_separator = newline
version_path_separator = os  # Use os.pathsep. Default configuration used for new pr

# set to 'true' to search source files recursively
# in each "version_locations" directory
# new in Alembic version 1.10
# recursive_version_locations = false

# the output encoding used when revision files
# are written from script.py.mako
# output_encoding = utf-8

sqlalchemy.url = driver://user:pass@localhost/dbname

# [post_write_hooks]
# This section defines scripts or Python functions that are run
# on newly generated revision scripts.  See the documentation for further
# detail and examples

# format using "black" - use the console_scripts runner,
# against the "black" entrypoint
# hooks = black
# black.type = console_scripts
# black.entrypoint = black
# black.options = -l 79 REVISION_SCRIPT_FILENAME

# lint with attempts to fix using "ruff" - use the exec runner, execute a binary
# hooks = ruff
# ruff.type = exec
# ruff.executable = %(here)s/.venv/bin/ruff
# ruff.options = --fix REVISION_SCRIPT_FILENAME

# Logging configuration
[loggers]
keys = root,sqlalchemy,alembic

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = WARNING
handlers = console
qualname =

[logger_sqlalchemy]
level = WARNING
handlers =
qualname = sqlalchemy.engine

[logger_alembic]
level = INFO
handlers =
qualname = alembic

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(levelname)-5.5s [%(name)s] %(message)s
datefmt = %H:%M:%S
```

The file is read using Python's `ConfigParser.SafeConfigParser` object. The `%(here)s` variable is provided as a substitution variable, which can be used to produce absolute pathnames to directories and files, as we do above with the path to the Alembic script location.

This file contains the following features:

- `[alembic]` - this is the section read by Alembic to determine configuration. Alembic's core implementation does not directly read any other areas of the file, not including additional directives that may be consumed from the end-user-customizable `env.py` file (see note below). The name "alembic" can be customized using the `--name` commandline flag; see Run Multiple Alembic Environments from one .ini file for a basic example of this.

  Note

  The default `env.py` file included with Alembic's environment templates will also read from the logging sections `[logging]`, `[handlers]` etc. If the configuration file in use does not contain logging directives, please remove the `fileConfig()` directive within the generated `env.py` file to prevent it from attempting to configure logging.

- `script_location` - this is the location of the Alembic environment. It is normally specified as a filesystem location, either relative or absolute. If the location is a relative path, it's interpreted as relative to the current directory.

  This is the only key required by Alembic in all cases. The generation of the .ini file by the command `alembic init alembic` automatically placed the directory name `alembic` here. The special variable `%(here)s` can also be used, as in `%(here)s/alembic`.

  For support of applications that package themselves into .egg files, the value can also be specified as a package resource, in which case `resource_filename()` is used to find the file (new in 0.2.2). Any non-absolute URI which contains colons is interpreted here as a resource name, rather than a straight filename.

- `file_template` - this is the naming scheme used to generate new migration files. Uncomment the presented value if you would like the migration files to be prepended with date and time, so that they are listed in chronological order. The default value is `%%(rev)s_%%(slug)s`. Tokens available include:

  - `%%(rev)s` - revision id
  - `%%(slug)s` - a truncated string derived from the revision message

> - `%%(epoch)s` - epoch timestamp based on the create date; this makes use of the Python `datetime.timestamp()` method to produce an epoch value.
> - `%%(year)d`, `%%(month).2d`, `%%(day).2d`, `%%(hour).2d`, `%%(minute).2d`, `%%(second).2d` - components of the create date, by default `datetime.datetime.now()` unless the `timezone` configuration option is also used.

> ! *New in version 1.8:* added 'epoch'

- `timezone` - an optional timezone name (e.g. `UTC`, `EST5EDT`, etc.) that will be applied to the timestamp which renders inside the migration file's comment as well as within the filename. This option requires Python>=3.9 or installing the `backports.zoneinfo` library. If `timezone` is specified, the create date object is no longer derived from `datetime.datetime.now()` and is instead generated as:

  ```
  datetime.datetime.utcnow().replace(
    tzinfo=datetime.timezone.utc
  ).astimezone(ZoneInfo(<timezone>))
  ```

  > ! *Changed in version 1.13.0:* Python standard library `zoneinfo` is now used for timezone rendering in migrations; previously `python-dateutil` was used.

- `truncate_slug_length` - defaults to 40, the max number of characters to include in the "slug" field.
- `sqlalchemy.url` - A URL to connect to the database via SQLAlchemy. This configuration value is only used if the `env.py` file calls upon them; in the "generic" template, the call to `config.get_main_option("sqlalchemy.url")` in the `run_migrations_offline()` function and the call to `engine_from_config(prefix="sqlalchemy.")` in the `run_migrations_online()` function are where this key is referenced. If the SQLAlchemy URL should come from some other source, such as from environment variables or a global registry, or if the migration environment makes use of multiple database URLs, the developer is encouraged to alter the `env.py` file to use whatever methods are appropriate in order to acquire the database URL or URLs.
- `revision_environment` - this is a flag which when set to the value 'true', will indicate that the migration environment script `env.py` should be run unconditionally when generating new revision files, as well as when running the `alembic history` command.
- `sourceless` - when set to 'true', revision files that only exist as .pyc or .pyo files in the versions directory will be used as versions, allowing "sourceless" versioning folders.

When left at the default of 'false', only .py files are consumed as version files.

- `version_locations` - an optional list of revision file locations, to allow revisions to exist in multiple directories simultaneously. See Working with Multiple Bases for examples.
- `version_path_separator` - a separator of `version_locations` paths. It should be defined if multiple `version_locations` is used. See Working with Multiple Bases for examples.
- `recursive_version_locations` - when set to 'true', revision files are searched recursively in each "version_locations" directory.

  New in version 1.10.

- `output_encoding` - the encoding to use when Alembic writes the `script.py.mako` file into a new migration file. Defaults to `'utf-8'`.
- `[loggers]`, `[handlers]`, `[formatters]`, `[logger_*]`, `[handler_*]`, `[formatter_*]` - these sections are all part of Python's standard logging configuration, the mechanics of which are documented at Configuration File Format. As is the case with the database connection, these directives are used directly as the result of the `logging.config.fileConfig()` call present in the `env.py` script, which you're free to modify.

For starting up with just a single database and the generic configuration, setting up the SQLAlchemy URL is all that's needed:

```
sqlalchemy.url = postgresql://scott:tiger@localhost/test
```

Create a Migration Script

With the environment in place we can create a new revision, using `alembic revision`:

```
$ alembic revision -m "create account table"
Generating /path/to/yourproject/alembic/versions/1975ea83b712_create_accoun
t_table.py...done
```

A new file `1975ea83b712_create_account_table.py` is generated. Looking inside the file:

```
"""create account table

Revision ID: 1975ea83b712
Revises:
Create Date: 2011-11-08 11:40:27.089406

"""

# revision identifiers, used by Alembic.
revision = '1975ea83b712'
down_revision = None
branch_labels = None

from alembic import op
import sqlalchemy as sa

def upgrade():
    pass

def downgrade():
    pass
```

The file contains some header information, identifiers for the current revision and a "downgrade" revision, an import of basic Alembic directives, and empty `upgrade()` and `downgrade()` functions. Our job here is to populate the `upgrade()` and `downgrade()` functions with directives that will apply a set of changes to our database. Typically, `upgrade()` is required while `downgrade()` is only needed if down-revision capability is desired, though it's probably a good idea.

Another thing to notice is the `down_revision` variable. This is how Alembic knows the correct order in which to apply migrations. When we create the next revision, the new file's `down_revision` identifier would point to this one:

```
# revision identifiers, used by Alembic.
revision = 'ae1027a6acf'
down_revision = '1975ea83b712'
```

Every time Alembic runs an operation against the `versions/` directory, it reads all the files in, and composes a list based on how the `down_revision` identifiers link together, with the `down_revision` of `None` representing the first file. In theory, if a migration environment had thousands of migrations, this could begin to add some latency to startup, but in practice a project should probably prune old migrations anyway (see the section Building an Up to Date Database from Scratch for a description on how to do this, while maintaining the ability to build the current database fully).

We can then add some directives to our script, suppose adding a new table `account`:

```
def upgrade():
    op.create_table(
        'account',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('name', sa.String(50), nullable=False),
        sa.Column('description', sa.Unicode(200)),
    )

def downgrade():
    op.drop_table('account')
```

`create_table()` and `drop_table()` are Alembic directives. Alembic provides all the basic database migration operations via these directives, which are designed to be as simple and minimalistic as possible; there's no reliance upon existing table metadata for most of these directives. They draw upon a global "context" that indicates how to get at a database connection (if any; migrations can dump SQL/DDL directives to files as well) in order to invoke the command. This global context is set up, like everything else, in the `env.py` script.

An overview of all Alembic directives is at Operation Reference.

Running our First Migration

We now want to run our migration. Assuming our database is totally clean, it's as yet unversioned. The `alembic upgrade` command will run upgrade operations, proceeding from the current database revision, in this example `None`, to the given target revision. We can specify `1975ea83b712` as the revision we'd like to upgrade to, but it's easier in most cases just to tell it "the most recent", in this case `head`:

```
$ alembic upgrade head
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
INFO  [alembic.context] Running upgrade None -> 1975ea83b712
```

Wow that rocked! Note that the information we see on the screen is the result of the logging configuration set up in `alembic.ini` - logging the `alembic` stream to the console (standard error, specifically).

The process which occurred here included that Alembic first checked if the database had a table called `alembic_version`, and if not, created it. It looks in this table for the current version, if any, and then calculates the path from this version to the version requested, in this case `head`, which is known to be `1975ea83b712`. It then invokes the `upgrade()` method in each file to get to the target revision.

Running our Second Migration

Let’s do another one so we have some things to play with. We again create a revision file:

```
$ alembic revision -m "Add a column"
Generating /path/to/yourapp/alembic/versions/ae1027a6acf_add_a_column.py...
done
```

Let’s edit this file and add a new column to the `account` table:

```
"""Add a column

Revision ID: ae1027a6acf
Revises: 1975ea83b712
Create Date: 2011-11-08 12:37:36.714947

"""

# revision identifiers, used by Alembic.
revision = 'ae1027a6acf'
down_revision = '1975ea83b712'

from alembic import op
import sqlalchemy as sa

def upgrade():
    op.add_column('account', sa.Column('last_transaction_date', sa.DateTime))

def downgrade():
    op.drop_column('account', 'last_transaction_date')
```

Running again to `head`:

```
$ alembic upgrade head
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
INFO  [alembic.context] Running upgrade 1975ea83b712 -> ae1027a6acf
```

We’ve now added the `last_transaction_date` column to the database.

Partial Revision Identifiers

Any time we need to refer to a revision number explicitly, we have the option to use a partial number. As long as this number uniquely identifies the version, it may be used in any command in any place that version numbers are accepted:

```
$ alembic upgrade ae1
```

Above, we use `ae1` to refer to revision `ae1027a6acf`. Alembic will stop and let you know if more than one version starts with that prefix.

Relative Migration Identifiers

Relative upgrades/downgrades are also supported. To move two versions from the current, a decimal value "+N" can be supplied:

```
$ alembic upgrade +2
```

Negative values are accepted for downgrades:

```
$ alembic downgrade -1
```

Relative identifiers may also be in terms of a specific revision. For example, to upgrade to revision `ae1027a6acf` plus two additional steps:

```
$ alembic upgrade ae10+2
```

Getting Information

With a few revisions present we can get some information about the state of things.

First we can view the current revision:

```
$ alembic current
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
Current revision for postgresql://scott:XXXXX@localhost/test: 1975ea83b712 -> ae1027
```

`head` is displayed only if the revision identifier for this database matches the head revision.

We can also view history with `alembic history`; the `--verbose` option (accepted by several commands, including `history`, `current`, `heads` and `branches`) will show us full information about each revision:

```
$ alembic history --verbose

Rev: ae1027a6acf (head)
Parent: 1975ea83b712
Path: /path/to/yourproject/alembic/versions/ae1027a6acf_add_a_column.py

    add a column

    Revision ID: ae1027a6acf
    Revises: 1975ea83b712
    Create Date: 2014-11-20 13:02:54.849677

Rev: 1975ea83b712
Parent: <base>
Path: /path/to/yourproject/alembic/versions/1975ea83b712_add_account_table.py

    create account table

    Revision ID: 1975ea83b712
    Revises:
    Create Date: 2014-11-20 13:02:46.257104
```

Viewing History Ranges

Using the `-r` option to `alembic history`, we can also view various slices of history. The `-r` argument accepts an argument `[start]:[end]`, where either may be a revision number, symbols like `head`, `heads` or `base`, `current` to specify the current revision(s), as well as negative relative ranges for `[start]` and positive relative ranges for `[end]`:

```
$ alembic history -r1975ea:ae1027
```

A relative range starting from three revs ago up to current migration, which will invoke the migration environment against the database to get the current migration:

```
$ alembic history -r-3:current
```

Note

As illustrated above, to use ranges that start with a negative number (i.e. a dash), due to a bug in argparse , either the syntax `-r-<base>:<head>`, without any space, must be used as above:

```
$ alembic history -r-3:current
```

or if using `--rev-range`, an equals sign must be used:

```
$ alembic history --rev-range=-3:current
```

Using quotes or escape symbols will not work if there's a space after the argument name.

View all revisions from 1975 to the head:

```
$ alembic history -r1975ea:
```

Downgrading

We can illustrate a downgrade back to nothing, by calling `alembic downgrade` back to the beginning, which in Alembic is called `base`:

```
$ alembic downgrade base
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
INFO  [alembic.context] Running downgrade ae1027a6acf -> 1975ea83b712
INFO  [alembic.context] Running downgrade 1975ea83b712 -> None
```

Back to nothing - and up again:

```
$ alembic upgrade head
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
INFO  [alembic.context] Running upgrade None -> 1975ea83b712
INFO  [alembic.context] Running upgrade 1975ea83b712 -> ae1027a6acf
```

Next Steps

The vast majority of Alembic environments make heavy use of the “autogenerate” feature. Continue onto the next section, Auto Generating Migrations.