03 Alembic

April 4, 2021

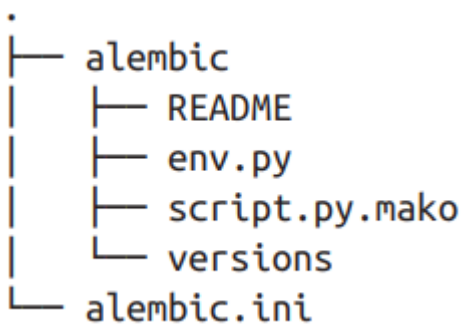
0.1 Overview

- `Alembic` is a tool for handling database changes that leverages SQLalchemy to perform migrations
- SQLALchemy will only create missing tables when we use the metadata's `create_all` method, it doesn't update the database tables to match any changes we might make to the columns
- it would also not delete tavles that were removed from the code
- Alembic uses SQLAlchemy to perform the migrations, they can be used on a wide array of backend databases
- Alembic programatically creates and perfomrs migrations to hadnle changes to the database

0.2 Getting Started with Alembic

0.2.1 Creating the Migrations Enviorment

- run `alembic init alembic` command to create the migrations enviroment
- to create migration enviorment in the `migrations/` directory, you have to use `alembic init migeations`
- this creates the migration enviorment and also creates an `alembic.ini` file with the configuration options



- the `versions/` directory will hold our migration scripts
- the `env.py` file is used by Alembic to define an dinstantiate a `SQLAlchemy` engine, connect to that engine and start a transaction, and calls the migration engine properly when you run the alembic commands
- the `script.py.mako` template is used when creating a migration, and it defines the basic structure of a migration

0.2.2 Configuring the Migration Environment

- the setting in the `alembic.ini` and `env.py` files need to be tweaked so that alembic can work with out database and application
- in the `alembic.ini` file we need to change the `sqlalchemy.url` to match our database co-nenction string
- in this case, we want to set it to connect to a `SQLite` file named `alembictest.db` in the current directory
- we need to change the `env.py` file to point to our `metadata`, which is an attribute of the `Base` instance we created in `db.py` file
- it uses the metadata to compare what it finds in the database to the models defined in SQLAlchemy
- we start by adding the current directory to the path that Python uses for locating modules so that it can see our app module
- finally we change the target metadata line in `env.py` to match our metadata object

[3]:
```
import os
import sys

class Base:
    metadata = None

sys.path.append(os.getcwd())
target_metadata = Base.metadata
```

0.3 Building migrations

- we will be exploring how to use autogenerate for adding tables and how to handcraft migrations to accomplush things that autogenerate cannot do

0.4 Generating a Base Empty Migration

- use the command `alembic revision -m "Empty Init"`
- this will create a migration file in the `versions/` subfolder

1. the migration message we specified
2. the alembic revision ID
3. the previous revision used to determine how to downgrade
4. the branch associated with this migration
5. any migrations that this one depends on

[7]:
```
'''
Empty Init # 1

Revision ID: 8a8a9d067
Revises:
Date: 2015-09-13 20:10:05.486995
'''
```

```
revision = '8a8a9d067' # 2
down_revision = None # 3
branch_labels = None # 4
depends_on = None # 5

#from alembic import op
import sqlalchemy as sa

def upgrade():
    pass

def downgrade():
    pass
```

- to run all the migrations form whatever the current database state is to the highest `Alembic` migration, we execute
  - `alembic upgrade head`

0.4.1 Autogenerating a Migration

- `alembic revision --autogenerate -m "Added User model"`
- then once the migration is generated, simply update
  - `alembic upgrade head`

Table 12-1. Schema changes that autogenerate can detect

Schema element	Changes
Table	Additions and removals
Column	Additions, removals, change of nullable status on columns
Index	Basic changes in indexes and explicitly named unique constraints, support for autogenerate of indexes and unique constraints
Keys	Basic renames

Table 12-2. Schema changes that autogenerate cannot detect

Schema element	Actions
Tables	Name changes
Column	Name changes
Constraints	Constraints without an explicit name
Types	Types like ENUM that aren't directly supported on a database backend

- changes to column type or changes in a server default will not be detected by `autogeneration's` capabilities and limitations

0.4.2 Building a Migration Manually

- Alembic cant detech a table name change, so we need to do this ourselves

1. create a nre migration that we can edit
   - `alembic revision -m "Renaming table name"`
2. with our migration created, we need to edit the migration file and add the rename operation to the upgrade and downgrade methods

```
def upgrade():
    op.rename_table('cookies', 'new_cookies')

def downgrade():
    op.rename_table('new_cookies', 'cookies')
```

3. after that we are ready to run our migrations with the alembic upgrade command
   - `alembic upgrade head`
4. to confirm what happened, go into database and see change

Table 12-3. Alembic operations

Operation	Used for
`add_column`	Adds a new column
`alter_column`	Changes a column type, server default, or name
`create_check_constraint`	Adds a new CheckConstraint
`create_foreign_key`	Adds a new ForeignKey
`create_index`	Adds a new Index
`create_primary_key`	Adds a new PrimaryKey
`create_table`	Adds a new table
`create_unique_constraint`	Adds a new UniqueConstraint
`drop_column`	Removes a column
`drop_constraint`	Removes a constraint
`drop_index`	Drops an index
`drop_table`	Drops a table
`execute`	Run a raw SQL statement
`rename_table`	Renames a table

0.5 Controlling Alembic

- we are going to learn the current migration level of the database, how to downgrade from a migration and how to mark the database at a certain migration level

0.5.1 Determining a Database's Migration Level

- to check what the last migration applied to the database is use the following command:
  - `alembic current`
- it returns the revision ID of the current migration and tells you whether it is the latest migration
- latest migration is also know as the head
- we can also confrim what the last migration changed by using
  - `alembic history`

0.5.2 Downgrading Migrations

- to downgrade, we need to choose the revision `ID` for the migration we want to go back to
- `alembic downgrade <REVISION-ID>`
- when we downgrade, it leaves us with an issue
- namely, when we use `alembic upgrade head` it will go the current revision ID and casuse issues

0.5.3 Marking the Database Migration Level

- when we want to skip a migration or restore a database, we want to explicitly mark the database as being a specific migration level to correct the issue
  - below we mark the database migration level
  - `alembic stamp <REVSION-ID>`
- stamping the database migration level does not actually run the migrations,
- it merely updates the Alembic table to reflect the migration level we supplied in the command
- this effectively skips applying the `<REVISION-ID>` we mentioned above

0.5.4 Generating SQL

- if you want to change your production database's schema with `SQL`, Alembic has that support
- this is important when we have massively distributed enviorments and need to run many different database servers
- the process is the same as performing an `online` Alembic upgrade like we did earlier
- we can specifiy both the starting and ending versions of the `SQL` script for upgrading from an empty database
- `alembic upgrade <START-REVISION-ID:END-REVISION-ID>`
- `alembic upgrade <START-REVISION-ID:END-REVISION-ID> --sql > migration.sql`
- `cat migration.sql`