Integrating a database migration framework

Jeff Trawick

January 23, 2020

TriPython — Triangle Python Users Group

Who am I?

I have been learning and enjoying Python for about eight years, using it to develop web applications, web scrapers, and other software. I had earlier mini-careers working on networking software for IBM mainframes and as a major contributor to Apache HTTP Server while working for IBM, Sun, Oracle, and as a consultant. I am employed by American Efficient, an energy-related company in Durham.

Scope of this talk/discussion

- A strange mix of high-level and low-level topics, mostly dealing with Alembic for the latter.
- Not a tutorial for exactly how to use a particular framework

What are migrations?

Common sense definition

Migration: Change to database schema or contents in order to work with the current application code.

Similar in some respects, but not a "migration": Other updates to the database contents, usually via SQL, when there is no UI and/or when they should follow the same change control process as code. Call these "table updates." (django-reversion implements an audit trail of sorts for changes in Django admin, which might be a good replacement.)

- If the database has all current "migrations" applied, the current code is expected to work reliably. If the database does not have all current migrations applied, the current code is expected to fail somehow.
- If the database does not have all "table updates" applied, the current code is expected to work reliably but will not produce the canonical query results.

Add a table

```
+    class User(MySQLAlchemyBase):
+        __tablename__ == 'j_user'
+
+        id = sqla.Column(sqla.Integer, primary_key=True)
+        email = sqla.Column(sqla.String(100), unique=True,
+                            nullable=False)
```

```
CREATE TABLE j_user (
    id SERIAL NOT NULL,
    email VARCHAR(100) NOT NULL
);
```

Add a column

```
+ is_superuser = sqla.Column(sqla.Boolean, nullable=False,
                             server_default='f')
```

```
ALTER TABLE
    j_user
ADD COLUMN
    is_superuser BOOLEAN DEFAULT 'f' NOT NULL;
```

Change column type

```
-    amount = sqla.Column(sqla.Text, default=None,
-                         nullable=True)
+    amount = sqla.Column(
+        sqla.Numeric(precision=10, scale=2),
+                     default=None, nullable=True
+    )
```

```
# ### commands auto generated by Alembic - please adjust! ###
op.alter_column(
    "j_table",
    "amount",
    existing_type=sa.TEXT(),
    type_=sa.Numeric(precision=10, scale=2),
    existing_nullable=True,
    postgresql_using='amount::numeric(10,2)',  # added manually!
)
# ### end Alembic commands ###
```

Merge two columns

A common example is merging `last_name` and `first_name` into `name`.

- In the model/schema definition, add the new column and leave the old ones.
- In the migration, add the new column and migrate existing data to it.
- Future: Remove the old column from the model/schema and drop it in a migration.

Desired migration capabilities for a project

Production

- Easily synchronize deploy of code with application of migrations while eliminating the manual application step.
- Roll back schema changes with a command if we need to revert the matching code changes.

Staging

- Easily keep the staging database up to date with no extra work.

Desired migration capabilities for a project

Developer

- Easily keep local database up to date with schema and other code-ish changes, reducing the need to download a fresh production db dump for accurate testing of code changes and/or poking around in recent migratoins to find the migration that fixes a problem symptom.
- Ability to iterate while developing schema changes without reloading the database from a prior dump or manually fixing it.

Reviewer

- Ability to roll back schema changes to the previous state after finishing a review which contains migrations.

Frameworks

A migration framework can:

- Establish the order in which migrations run
- Record in the database the successful application of migrations
- Provide a way to display the status of migrations
- Provide a way to roll back migrations (subject to how they are written)

The "migration framework" does a lot of the heavy lifting for automation.

You may need to customize it; you will need to implement integration into your deploy process.

Rollbacks

The ability to roll back the database to a prior set of schemas requires rollback logic in all migrations being rolled back. Common frameworks will auto-generate rollback logic for schema changes it handles. Developers can handle rollbacks in an appropriate manner on a case by case basis.

For migrations written manually, here are some of the obvious choices to make.

No-op rollback

This is suitable for migrations that are just updating tables with the latest information and arent tied exactly to the level of Python code; e.g., changing a factor.
The rollback code would look something like

```
def downgrade(**kwargs):
    pass
```

(sample Alembic rollback function)

Rollback that blows up

This is suitable for when something needs to be done to revert but it is too much trouble to implement it.

```
def downgrade(**kwargs):
    assert False, 'yada yada yada must be rolled back manually'
```

Heavy-handed support: Make a backup of a modified table in the upgrade path, and restore it in the downgrade path. (A future migration would remove the backup.)

Rollback that actually rolls back the change

This varies widely based on the migration step. Here are common cases:

- remove a new table
- remove a new column
- remove a new constraint
- transform data back to the previous format, if no information was lost

Conditional rollback/blow-up

Scenario: The migration is easy to roll back as long as users haven't somehow affected the tables. As rolling back usually happens soon after the migration is applied, that's not so bad.

The rollback code can check for the results of such user activity. If present, blow up with a helpful message, and leave that for manual recovery; if not, perform the rollback.

Integrating with deployment

The basic idea

Start running the new code with a matching database without blowing up.
Avoid:

- Old code still running and accessing a removed column or breaking a new constraint or ...
- New code accessing a removed column or breaking an old constraint or ...
- SQLAlchemy: old code still running and blowing up while trying to reference the Python definition of an enum value seen in db or ...

Simple

Enable a maintenance page, then completely stop the app, then apply migrations, then start the app using the new code and disable a maintenance page.
Maybe that is your bank on Sunday morning? Maybe that is you messaging your internal users during the business day that they shouldn't use the app from 10am until you say "go"?

Not simple

Run migrations which add new things used by the new code, then start containers running the new code, then wait for the containers running the old code to quiesce, then run migrations which remove things used by the old code.
(s/containers/whatever/)
Generally: pre-deploy and post-deploy migrations

Not simple

Stage migrations into additions and deletions which are intended to be rolled out in separate deploys, and only merge the deletions and build images once only new code is running.

The first choice is definitive and simple, but deploys would either need to happen outside of business hours or the deployer would need to give staff staff pre-notification.

The latter two choices require migrations to be designed in two logical "before" and "after" pieces, which can place constraints on possible changes, but is usually required in a 24x7 public-facing environment.

Alternatively, we can allow some brokenness around the time of deployment.

Oops

Some migrations are time consuming, such as when certain materialized views need to be rebuilt after a schema change is applied.

- Implement a downtime mechanism regardless of the normal deployment strategy?
- Deal with that particular migration in an ad hoc manner?

To consider

- Need a way to collect output from a migration? Maybe the migration logs the result of some queries before/after?

Pitfall

Applying migrations as part of server container startup may be part of a very simple integration with deployment, but you have to avoid the possibility that more than one container is trying to migrate under any circumstance (e.g., locking).

Pitfall

I implemented an image build that pushes a `latest` tag when images are published. The container runtime is configured to pull the `latest` tag. What happens if the runtime needs to restart a container on its own, grabs the `latest`, and that new code relies on a new migration? (In this case, different tags are needed for "latest" vs. "deployable".)

Alembic migrations for SQLAlchemy

Some properties of Alembic

- Use of transactions highly configurable
- Autogeneration of migrations
  - Looks at database, and will complain if latest migration is not applied
  - Assumes that the model definitions are correct and the database is wrong, and will generate code to make the database match the model definitions
  - Will generate extra migration logic to account for other schema differences in your database; this can be easily removed from the generated migration
  - You can point Alembic to a production database as a read-only user to look for missed schema changes.
  - By default, Column Type changes won't be recognized.

Options/tricks

`compare_type=True` to track Column Type changes (e.g., `FLOAT` to `NUMERIC`)

Options/tricks

Default generated code is poorly formatted. Add Black to your virtualenv and uncomment the configuration in `alembic.ini`.

Options/tricks

The ability to edit a template (`script.py.mako`) used for new migration files is great.

- Add ignore-flake8-warning comments as appropriate so devs don't have to delete sample code?
- If you change the indention of where the Alembic-generated code goes (e.g., put it inside a transaction context), you'll have to implement a simple post-write hook to recognize where to indent further (groan!).

Options/tricks

You probably don't want your database connect string in `alembic.ini`. Just delete that, then change the context setup like this:

```
context.configure(
        url=_get_db_url(),  # implement however you want!
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
        # maybe useful, since there are two calls to configure
        **COMMON_CONTEXT_SETTINGS,
    )
```

Options/tricks

A clean way to pass arbitrary parameters down to your `upgrade` and `downgrade` functions is to declare them as accepting `**kwargs`, and pass those parameters to them in `env.py` when calling `run_migrations`. The functions can use anything they need and ignore the other arguments.

Options/tricks

Your database may have some (BUNCHES?) of things which aren't tracked by SQLAlchemy models. Trying to auto-generate a migration will then try to fix the database to match. (`DROP TABLE` and all that!)

Implement an `include_object` callback that indicates when something in the database should be ignored. (Hint: Try to use patterns in the names of these things. E.g., maybe copies or alternate versions of a table should have a recognizable date string in them, so that they can all be ignored via a regex.)

Options/tricks

Too many things called `alembic`? The sample command in the Alembic docs will call your project migration directory `alembic`. That's also the name of Alembic's package and its CLI in your virtualenv. Call your project migration directory something else to avoid awkardness when trying to import things from there while debugging or otherwise.

Options/tricks

Those hash values in filenames got you down? (e.g., `alembic_migrations/versions/a1082f966e8c_foo.py`
Configure the pattern in `alembic.ini` to include a date string so that it is easier to `ls` them in approximately the right order. (The date would reflect when the migration was written, not necessarily the order that migrations were merged to a deployable branch. Devs can rename the file to adjust the date as appropriate.)

Stuff for which I forgot to write a slide until too late!

- Flask-Migrate (what does it add?)
- The project-I-can't-remember that created SQLAlchemy column subclasses that caused accesses to fail, to ensure that code had stopped referencing columns to be removed.

References

References

- Alembic Tutorial
  https://alembic.sqlalchemy.org/en/latest/tutorial.html
- Flask-Migrate documentation
  https://flask-migrate.readthedocs.io/en/latest/
- Database section within The Flask Mega-Tutorial
  https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-iv-database
- Database section within the Django Tutorial
  https://docs.djangoproject.com/en/2.2/intro/tutorial02/

References

- Jason Myers, Introduction to SQLAlchemy and Alembic Migrations
  `https://www.youtube.com/watch?v=stpGLcX5XgM`
- Selena Deckelmann, Sane schema migrations with Alembic and Postgres
  `https://www.youtube.com/watch?v=y4EQsBssn_0`

References

- Robert Lechte Your database migrations are bad and you should feel bad `https://www.youtube.com/watch?v=xr498W8oMRo&t=1272s`
- Vineet Gopal Move fast and migrate things: how we automated migrations in Postgres `https://benchling.engineering/move-fast-and-migrate-things-how-we-automated-migratio`
- Migra A schema diff tool for PostgreSQL (Hacker News) `https://news.ycombinator.com/item?id=16675088`
- Sqitch, `https://sqitch.org/about/`

Thank you!