Assume we have a database as our main one (production) called suppliers. And it has data in it. We want to create its ORM without losing any data. So these steps are the ones we need.

1. We already have the db created so we run this command to generate the ORM models

Simply run the file create\_models.py and it takes care of it.

1. Make sure that Base class in imported from the database.py inside the models.py.
2. So expectedly now ORM models are all created and using these classes you must be able to add or delete records in the database.
3. Now we want to build the migrations to be able to add or delete or some other changes to the ORM and ORM models get converted into migrations and take care of the latest version of the database.
4. Make sure that alembic is installed by typing .
5. creates a folder called migrations\_for\_db inside which we have . This command also creates out of that folder.
6. In the file make
7. In file make
8. . This command inspects the state of the connected database and the state of the target metadata and then creates a migration that brings the database in line with metadata.

If you face the error that models module does not exist, then you have a problem with sys.path and you can resolve it by running

The part after ; is the path to your project.

1. Now take a look at the file created inside migrations\_for\_db/versions. This file is the migration file. The upgrade() and downgrade() are just passing for our scenario. The reason is that we already have the database and all the tables on our local machine (or on somewhere else) and **alembic does not notice any changes between ORM models and the database**. So it just passes through these functions.
2. Now the migraions are ready. Check the endpoints you must be able to do CRUD using both ORM model object and CLI of database.

……………………………………………………………………………………………………………………………………………………

**Scenario 1**

Now Let’s assume that we want to add a new column for a table. And let us assume that this table is called Parts and is an edge case and is in a many-to-many relationship with another table called Vendors and also in a one-to-one relationship with table partDrawing.

After adding a new column to model Parts, all the queries using this class fail and announce that the new column does not exist. The reason is that migrations are not up to date so we make them up to date so that they change the real database accordingly.

1. Now a new file (that its last part name is the argument given after –m to the command above meaning adding\_a\_new\_column\_to\_class\_parts).

So we should have in that file.

"""Adding a new column to class Parts  
  
Revision ID: 5d028b75242a  
Revises: 1ced9c0cbbca  
Create Date: 2020-12-28 19:42:42.648149  
  
"""  
*from* alembic *import* op  
*import* sqlalchemy *as* sa  
  
  
# revision identifiers, used by Alembic.  
revision = '5d028b75242a'  
down\_revision = '1ced9c0cbbca'  
branch\_labels = *None*depends\_on = *None  
  
  
def* upgrade():  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.add\_column('parts', sa.Column('part\_desc', sa.String(length=255), nullable=*False*))  
 # ### end Alembic commands ###  
  
  
*def* downgrade():  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.drop\_column('parts', 'part\_desc')  
 # ### end Alembic commands ###

1. And finally we have to run command below to make alembic change our database to the latest changes.
2. After running this command we get an error that says “sqlalchemy.exc.IntegrityError: (psycopg2.errors.NotNullViolation) column "part\_desc" of relation "parts" contains null values”.

The reason is: When you try to add a NOT NULL constraint onto a column, it will be executed on PostgreSQL as an atomic operation like: ALTER TABLE table\_name ALTER COLUMN column\_name SET NOT NULL; As a consequence, PostgreSQL will fully scan the table to check that the constraint is valid on all the rows.

Just set a default value to that not null column and we are good to go.

So Now you have 2 ways ahead of us.   
1- Delete the migration regarding adding a new column (that is deleting the file with *adding\_a\_new\_column\_to\_class\_parts*) and then change the model Parts that is responsible for that new column and make it have a default value.  
ATTENTION: This does not work. The code that gets generated by alembic does NOT take care of the default values for models.

So use the following way that works.

2- Directly add a default value to the migration file (that is the file with *adding\_a\_new\_column\_to\_class\_parts*) and then add a default value to the column of the model Parts just for the sake of consistency between the migration code and the model code we have for the column part\_desc.

*def* upgrade():  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.add\_column('parts', sa.Column('part\_desc', sa.String(length=255), server\_default='Hi',  
 nullable=*False*))  
 # ### end Alembic commands ###

and in the models we change the class Parts to have.

part\_desc = Column(String(255), nullable=*False*, default='Hi')

1. After doing so you can run the command and your database changes and has a default value for the column part\_desc. Now all the endpoints using ORM classes and CLI of the database must be working and the database is up to date.

…………………………………………………………………………………………………………………………………………………

**Scenario 2**

**Assume a new developer want to set up their own local database instead and update it using the main database so that all the state of both databases is equal.**

The steps below are doing that.

1. Create a local database using postgres client tools.

Create database local**S**uppliers;

1. point your in alembic.ini to this new db.

sqlalchemy.url = postgresql://postgres:password@localhost/local**s**uppliers

1. Run . This tells sqlalchemy that the current migration represents the state of the database- so next time you run it will begin from this migration.
2. Run -. This will create your desired bulk migration that brings a fresh db in line with the current one.

Attention: You have NOT changed the DATABASE\_URL so all the CRUD are still working as we use the main database.  
Attention2: The new database is up to data in terms of the columns, tables and alembic\_version that is equal for both databases after running all the command mentioned above. But the new database does inherit any data from the main database.

…………………………………………………………………………………………………………………………..

**Scenario 3**

**Creating a new database sharing the same state and inherits the data from the source database.**

1. Copy the database using the source database by a syntax like.

create database localSuppliers  
with template suppliers;

1. Steps 2 through 5 of **Scenario 2**.
2. Change the LOCAL\_DATABASE to be of the localsuppliers (your local database rather than the source one). So that all the data that you delete or remove just affects your local db.

Attention: ANY changes to the database now must affect only localsuppliers and the main database is going to be unaffected.

Now let us assume that we wanted to add a new column to class Parts (the most challenging class) and call it score.

1. Change the models accordingly.

part\_score = Column(Integer, nullable=*True*)

1. alembic stamp head
2. alembic revision --autogenerate -m "Adding a new column score"
3. Check the new revision created to make sure all looks good.
4. Alembic upgrade head
5. Now your local database has a new column.
6. In case you want the source database to reflect the same changes as the local database (in terms of the columns or tables added, deleted, renamed) you have to make sure that the source database has the same status as the local database. Do these steps to make that happen. **Scenario 2** steps 2 through 5 with some changes.
7. sqlalchemy.url = postgresql://postgres:password@localhost/*suppliers*
8. alembic stamp head
9. alembic revision –autogenerate -m "Make the source db in line with the local"
10. alembic upgrade head
11. DATABASE\_URL = 'postgresql://postgres:password@localhost/*suppliers*'

Now the source code also has the new column score.

…………………………………………………………………………………………………………………………………………………

Attention: In case that you want to delete the migrations folder and run the command below again.

you may get

ERROR [alembic.util.messaging] Can't locate revision identified by 'e6863d1a9a3d'

FAILED: Can't locate revision identified by 'e6863d1a9a3d'.

The reason is: You deleted the migration directory but the version has been saved in the database, so you have to delete the version info in the database, run

in psql to delte alembic\_version relation from your database.

…………………………………………………………………………………………………………………………………………………..

Attention: The code generated by alembic according to SQLAlchamy models (ORM) is NOT so much trustable.

Changing a column name may make you lose all the data if you do not adjust the code generated by alembic.

…………………………………………………………………………………………………………………………………..

After adding or deleting a column to the db do the following.

1. Alembic stamp head
2. Alembic revision –autogenerate -m “Did sth on table X”
3. Check the revision created and ADJUST it as it may make you lose the data
4. Alembic upgrade head

............................................................................................................................

The target\_metadata collection may also be defined as a sequence if an application has multiple **[MetaData](https://docs.sqlalchemy.org/en/13/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "(in SQLAlchemy v1.3))** collections involved:

**from** **myapp.mymodel1** **import** Model1Base

**from** **myapp.mymodel2** **import** Model2Base

target\_metadata = [Model1Base.metadata, Model2Base.metadata]

The sequence of **[MetaData](https://docs.sqlalchemy.org/en/13/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "(in SQLAlchemy v1.3))** collections will be consulted in order during the autogenerate process. Note that each **[MetaData](https://docs.sqlalchemy.org/en/13/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "(in SQLAlchemy v1.3))** must contain **unique** table keys (e.g. the “key” is the combination of the table’s name and schema); if two **[MetaData](https://docs.sqlalchemy.org/en/13/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "(in SQLAlchemy v1.3))** objects contain a table with the same schema/name combination, an error is raised.