Week 18 Study Guide

Table of Contents

- Reason about the way Alembic orders your migrations; and,
- Handle branching and merging concerns
- Configuring a Flask application to use Alembic;
- Run commands to manage your database through the flask command; and,
- Instead of alembic init…
- Check the help for the rest of the commands, which are the same as Alembic
- Autogenerate migrations from your models!
- Instead of alembic migrate…

Python Unit Testing Objectives

Use the built-in unittest package to write unit tests

- unittest
  - Built in to python
  - Requires that you build a class that inherits from `unittest.TestCase`
  - test functions must start with `test_`
  - Has a collection of assertion functions

Install and use the pytest package to write unit tests

- pytest
  - Has better output than unittest
  - Just requires a test file full of test methods
  - Can also run `unittest` based tests
  - Uses python built in assert keyword

Python Environment Management Objectives

- pyenv
  - Installs versions of python inside your home directory in a `.pyenv` folder
  - Allows you to easily switch between python versions with the `pyenv global` command.
  - Closest Node.JS equivalennt would be `nvm`

Describe pip

- pip
  - Installs Python packages into python's library path folders.
  - Can use a `requirements.txt` file to install a set of packages.
  - Can be used standalone but we used it mostly by levaraging pipenv which uses it under the hood
  - Closest Node.JS equivalent would be `npm install -g`

Describe virtualenv

- virtualenv
  - Creates a virtual installation of python. Uses symbolic links and adjustments to certain environment variables to isolate python packages from one project to another.
  - Can be used standalone but we used it mostly by leveraging pipenv which uses it under the hood
  - No Node.JS equivalent

Demonstrate how to use pipenv to initialize a project and install dependencies

- pipenv
  - Combines `pip` and `virtualenv` into one command.
  - Creates a virtual environment using `virtualenv`
  - Uses `pip` internally to install packages listed in a `Pipfile`.
  - Locks packages to specific versions with a `Pipfile.lock`.
  - Uses an environment variable named `PIPENV_VENV_IN_PROJECT`. When set to `1` it causes pipenv to create the `virtualenv` inside your project directory in a folder named `.venv` instead of in your home directory
  - Will read a `.env` file and populate the environment variables inside the `virtualenv`
  - Can generate a `requirements.txt` file for use with regular `pip`
  - Closest Node.JS equivalent would be `npm`

Demonstrate how to run a Python program using pipenv using its shell

```
pipenv shell
```

This will start a new shell inside the virtual environment.

Then you can run python programs and they will run with the right set of packages and environment variables

```
python someprogram.py
```

When you are finished running commands in the virtual environment don't forget to exit the shell by issuing the `exit` command, or using Control-D.

Demonstrate how to run a Python program using pipenv using the run command

If you just need to run a single command inside the virtual environment you can use the `pipenv run` command.

```
pipenv run python someprogram.py
```

Describe how modules and packages are found and loaded from import statements

First some definitions:

Module : a single .py file or a directory with a `__init__.py` file can be considered a module

Package : a collection of modules and submodules in a directory

Submodule : a python module inside a sub directory of a module

Python Path

The Python Path is a list of directories python looks for modules in.

When you import a module, python searches these directories for a file module or directory module (with a **init**.py file in it) that matches the name you are trying to import.

You can inspect the python path from python by printing `sys.path`

You can add directories to the python path by setting the `PYTHON_PATH` environment variable.

Luckily we have tools like `virtualenv` and `pipenv` which means we do not have to worry as much about setting the Python path manually.

Exporting

Inside a python script, any variables, functions or classes are automatically exported and can be imported by name.

If you want to control which things get exported from a python module you can set the variable `__all__` equal to a list of strings representing the things to export.

The Rules

Using the `import` statement

- When you import a .py file as a module, it searches `sys.path` for a file with that name and runs that file.
- When you import a directory as a module, it also searches `sys.path` for a directory with that name and runs the `__init__.py` contained in that directory.

Using the `python` command line interpreter

- When you run a .py file it runs that file
- When you run a directory it runs `__main__.py`
- When you run a directory with the `-m` option, it searches `sys.path` for the module and runs both the `__init__.py` and the `__main__.py`

Most of the time we'll use `__init__.py` not `__main__.py` when we build our own modules.

Documentation on import

- Import System
- Import Statement

Describe the purpose of and when init.py runs

When you run a directory with the `-m` option, or when you import a directory, the `__init__.py` file executes. The purpose of `__init__.py` to be able to build python packages and subdivide the packages into multiple sub-modules.

Describe the purpose of and when main.py runs

When you run a directory as a regular python program (not with `-m`) the `__main__.py` file is executed. The purpose of `__main__.py` is to allow us to execute a directory as if it was a python program.

Flask Objectives

Setup a new Flask project

Flask is a python based web application server. It is a backend framework similar to Express.js

First, you should install Flask into a virtual environment

```
pipenv install flask
```

Create a python script to start your application. This might be `app.py` or another script which imports an `app/__init.py` module.

This is the bare minimum needed to make Flask application:

```
from flask import Flask
app = Flask(__name__)
```

Flask requires that you set an environment variable called `FLASK_APP` before it will run. It needs to be set to the name of your flask application script or module. You could put this into a `.env` file and let `pipenv` load it or use the `python-dotenv` module to load a `.flaskenv` file.

Often you might use the `.flaskenv` file to load environment variables like `FLASK_APP` and checking it into source control, and reserve the `.env` file for secret information like passwords or database configurations.

Run a simple Flask web application on your computer

Once you have your application setup, you can just run it with flask.

```
pipenv run flask run
```

Utilize basic configuration on a Flask project

You can use the `app.config` dictionary to hold Flask configuration values.

An even better way to setup your flask app is to create a python module with a configuration class in it. This class just needs properties for each configuration variable. Then you can import the class, and use the `from_object()` method to load it into the app's config dictionary.

```
# config.py

class Config:
    SOME_CONFIG_VARIABLE = 'Some value'
```

```
# app.py
# Import the config class
from config import Config

app = Flask(__name__)

# Load the config into Flask.
app.config.from_object(Config)
```

You can access any config variables in your flask app by just referencing them on the `app.config` dictionary.

```
app.config['SOME_CONFIG_VARIABLE']
```

Create a static route in Flask

A static route is one that just routes to a path without any parameters.

```
# Examples
@app.route('/')
def index():
    """Put code here to execute when `/` is visited"""
    pass

@app.route('/somepath')
def some_path():
    """Put code here to execute when `/somepath` is visited"""
    pass
```

Create a parameterized route in Flask

A parameterized route uses `<>` characters to declare that part of a path should be a parameter.

```
# the <id> parameter will be captured and passed into the function as the first
# argument
@app.route('/item/<id>')
def item(id):
    return f'<h1>Item {id}</h1>'
```

```
# You can also specify the type of the parameter by prepending it with the type
# and a colon
@app.route('/item/<int:id>')
def item(id):
    return f'<h1>Item {id}</h1>'
```

Use decorators run code before and after requests

The `@app.before_request` and `@app.after_request` happen before and after every request to the server. Use them to do any initialization or cleanup you need to happen on each request

```
@app.before_request
def before_request_function():
```

```
    print("before_request is running")


@app.after_request
def after_request_function(response):
    print("after_request is running")
    return response
```

`@app.before_first_request` only happens once before the very first request to the server

```
@app.before_first_request
def before_first_function():
    print("before_first_request happens once")
```

Identify the "static" route

Don't confuse this with declaring a static route above. This is a special built in route you don't have to define at all.

If you create a folder called `static` then any requests to `/static` on your server will cause flask to serve up the files contained in this folder.

```
http://localhost:5000/static/styles/main.css
```

```
.
├── Pipfile
├── Pipfile.lock
├── app              <- directory where Flask is created
│   ├── __init__.py  <- file in which Flask is created
│   ├── routes.py
│   ├── static       <- static files served from here
│   │   └── styles
│   │       └── main.css
│   └── templates
│       └── main.html
└── app_loader.py
```

Use WTForms to define and render forms in Flask

WTForms is a python package that allows you to easily generate forms and form fields. Flask-WTF is a companion python package that allows you to parse POST data from a form and render the form fields.

You define your form as a class that inherits from the FlaskForm base class.

```
from flask_wtf import FlaskForm

class SampleForm(FlaskForm):
```

Then inside the class use WTForm fields on properties of the class.

```
class SampleForm(FlaskForm):
    name = StringField('Name')
```

In your route, you can instantiate an instance of your form and then pass it to a view to be rendered.

```
from app.sample_form import SampleForm

# Create an instance of our form
form = SampleForm()

# And pass it to the view template
return render_template('form.html', form=form)
```

Inside the view template, you can access the fields from the form to output HTML for the form and it's fields.

```
<form action="" method="post" novalidate>
    {{ form.csrf_token }}
    <p>
        {{ form.name.label }}
        {{ form.name(size=32) }}
    </p>
    <p>{{ form.submit() }}</p>
</form>
```

The calls inside of the `{{ }}` will output HTML.

Because of some special python magic (the **call** and **str** methods on FlaskForm), you can just use the properties without calling them, or call them with extra parameters, and both will work!

Passing extra keyword parameters to the field instances will add HTML attributes for those parameters. However, because `class` is a reserved word in Python, you will have to use `class_` when you want to add a CSS class.

```
    form.name(size=32, class_='name')
```

Use WTForms to validate data in a POST with the built-in validators

To validate a form with Flask-WTF you can call the `validate_on_submit` method on your form instance. This must be done inside of a route that handles `POST` requests.

```
@app.route('/submit', methods=['POST'])
def handle_form_submit():
    if form.validate_on_submit():
        # Do something with the form data.
        # and return something
        return
    # You can put code here to handle what happens when
    # the form fails validation, like redirecting or rendering
    # the form again.
    return
```

It should be noted that `validate_on_submit` *automatically* reads the incoming parameters from the `request` object in Flask, so there's no reason to import it or use it manually.

CSRF

To protect against Cross-Site Request Forgery attacks, Flask-Wtf automatically generates and checks CSRF tokens. However we must add one of these two fields in our form in order to print out the CSRF token.

```
# This one prints out ALL the hidden fields including the CSRF that are
# defined on the form class
{{ form.hidden_tag() }}
```

or

```
# While this one only prints out the CSRF token hidden field
{{ form.csrf_token() }}
```

Use the following basic field types in WTForms

You use these by creating a class property on your class which inherits from FlaskForm

```
class MyForm(FlaskForm):
    field1 = StringField()
```

- BooleanField
- DateField
- DateTimeField
- DecimalField
- FileField
- MultipleFileField
- FloatField
- IntegerField
- PasswordField
- RadioField
- SelectField
- SelectMultipleField
- SubmitField
- StringField
- TextAreaField

Check the documentation on the specific parameters you must pass each type of field.

WTForms Field Documentation

Create a Flask Blueprint

A Flask Blueprint is a way to modularize our routes.

In a new module, import Blueprint and create one like this:

```
# admin.py
from flask import Blueprint

admin_bp = Blueprint('admin', __name__, url_prefix='/admin')
```

Register the Flask Blueprint with the Flask application

Then import it into your main Flask app file
and register it so Flask knows about the routes contained within.

```
from admin import admin_bp

app = Flask()

app.register_blueprint(admin_bp)
```

Use the Flask Blueprint to make routes

Inside the blueprint you can add routes, like you normally would, just you use the blueprint instance instead of using `app`

```
@admin_bp.route('/', methods=('GET', 'POST'))
def admin_index():
    return
```

Configure and use sessions in Flask

You must set a `SECRET_KEY` property in your flask config for sessions to work.

You can import session from flask.

```
from flask import Flask, session
```

Then simply use `session` to store things you want to be available later

```
# To set something in the session
session['key'] = value
# To get something from the session
session.get('key')
# to remove something from the session
session.pop('key')
```

Use a Jinja template as return for a Flask route with render_template

Use the `render_template` method to render the template into a string, and then return it from your route. You can give it the HTML file and keyword arguments that will be accessible as variables inside the template.

```
@app.route('/')
def index():
    return render_template('index.html', sitename='My Sample')
```

Add variables to a Jinja template with {{ }}

Then inside our HTML we can access the key

```
<title>{{ sitename }}</title>
```

Check the Jinja2 docs for all the things you can do in Jinja2 templates.

Use include to share template content in Jinja

Just use the include directive to include another html inside a jinja template.

```
{% include 'file.html' %}
```

Psycopg Objectives

Connect to a PostgreSQL RDBMS using Psycopg

```
import psycopg2

CONNECTION_PARAMETERS = {
                          'dbname': 'psycopg_test_db',
                          'user': 'psycopg_test_user',
                          'password': 'password',
}

with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
    print(conn.get_dsn_parameters())
```

Open a "cursor" to perform data operations

Use the with keyword to clean up connections and database cursors

```
import psycopg2

CONNECTION_PARAMETERS = {
                          'dbname': 'psycopg_test_db',
                          'user': 'psycopg_test_user',
                          'password': 'password',
}

with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
    print(conn.get_dsn_parameters())
```

Use results performed from executing a SELECT statement on existing database entities

```
with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
    with conn.cursor() as curs:
        curs.execute('SELECT manu_year, make, model FROM cars;')
        cars = curs.fetchall()
        for car in cars:
            print(car) # (1993, 'Mazda', 'Rx7')
```

Use parameterized SQL statements to insert, select, update, and delete data

```
def print_all_cars():
    with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
        with conn.cursor() as curs:
            curs.execute('SELECT manu_year, make, model, owner_id FROM cars;')
            cars = curs.fetchall()
            for car in cars:
                print(car)


print_all_cars()
# Output:
# (1993, 'Mazda', 'Rx7', 1)
# ...additional cars
```

Specify what type Psycopg will convert the following PostgreSQL types into:

PostgreSQL	Python
NULL	None
bool	bool
double	float
integer	long
varchar	str
text	unicode
date	date

SQLAlchemy Objectives

Describe how to create an "engine" that you will use to connect to a PostgreSQL database instance

Note: When using Flask-SQLAlchemy you don't have to do this

```
from sqlalchemy import create_engine

engine = create_engine("postgresql://sqlalchemy_test:password@localhost/sqlalchemy_test")
```

Describe how the with engine.connect() as connection: block establishes and cleans up a connection to the database

Note: When using Flask-SQLAlchemy you don't have to do this

```
from sqlalchemy import create_engine

db_url = "postgresql://sqlalchemy_test:password@localhost/sqlalchemy_test"
engine = create_engine(db_url)

with engine.connect() as connection:
    result = connection.execute("""
        SELECT o.first_name, o.last_name, p.name
        FROM owners o
        JOIN ponies p ON (o.id = p.owner_id)
    """)
    for row in result:
        print(row["first_name"], row["last_name"], "owns", row["name"])

engine.dispose()
```

Describe how to create a database session from an engine

Note: When using Flask-SQLAlchemy you don't have to do this

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

db_url = "postgresql://sqlalchemy_test:password@localhost/sqlalchemy_test"
engine = create_engine(db_url)

SessionFactory = sessionmaker(bind=engine)

session = SessionFactory()

# Do stuff with the session

engine.dispose()
```

Create a mapping for SQLAlchemy to use to tie together a class and a table in the database

Mappings

Mappings with plain SQLAlchemy

With just SQLAlchemy we inherit from `Base` and we have to import all the schema objects and types manually.

```
# ponies.py
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.schema import Column, ForeignKey
from sqlalchemy.types import Integer, String

Base = declarative_base()

class Pony(Base):
    __tablename__ = 'ponies'

    id = Column(Integer, primary_key=True)
    name = Column(String(255))
    birth_year = Column(Integer)
    breed = Column(String(255))
    owner_id = Column(Integer, ForeignKey("owners.id"))
```

Mappings with Flask-SQLAlchemy

When using Flask-SQLAlchemy we inherit from `db.Model` instead of `Base` and we can use all the schema objects and types because Flask-SQLAlchemy attaches them to the `db` instance. So we just prefix them with `db.`

```
# owner.py

from .models import db

class Pony(db.Model):
    __tablename__ = 'ponies'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    birth_year = db.Column(db.Integer)
    breed = db.Column(db.String(255))
    owner_id = db.Column(db.Integer, db.ForeignKey("owners.id"))
```

Relationships

One-to-Many

Just create the proper foreign key columns on the models, and then define the relationships. *(Remember Flask-SQLAlchemy will need to preface most of these objects with `db.`)*

Remember the rule of thumb. The "Many" always has the foreign key on it.

```
# The one
class Owner(db.Model):
    __tablename__ = "owners"

    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(255))
    last_name = db.Column(db.String(255))
    email = db.Column(db.String(255))

    # ponies belong to an owner
    ponies = db.relationship("Pony", back_populates="owner")

# The Many
class Pony(db.Model):
    __tablename__ = "ponies"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    birth_year = db.Column(Integer)
    breed = db.Column(db.String(255))
    # The pony contains an owner_id foreign key
    owner_id = db.Column(db.Integer, db.ForeignKey("owners.id"))
    # An owner has many ponies
    owner = db.relationship("Owner", back_populates="ponies")
```

Many-to-Many

Remember that a Many-to-Many relationship is really two One-to-Many relationships with a join table in the middle.

You must create a `Table()` object and not a model for your join table.

```
# We define the foreign keys on our join table, which joins the Ponies
# to thier Handlers.
pony_handlers = db.Table(
    "pony_handlers",
    db.Column("pony_id", db.ForeignKey("ponies.id"), primary_key=True),
    db.Column("handler_id", db.ForeignKey("handlers.id"), primary_key=True)
```

Then setup the relationships on each Model making sure to define a "secondary" keyword argument is set to the table we just made.

```
    # Inside the Pony class...
    handlers = db.relationship("Handler",
                           secondary=pony_handlers,
                           back_populates="ponies")

    # Inside the Handler class...
    ponies = db.relationship("Pony",
                         secondary=pony_handlers,
                         back_populates="handlers")
```

On backpopulates

If you leave out the backpopulates parameter, then when you create an object and add related data, the opposite relationship won't be populated. For instance assume we have an `Owner` instance and we add a `Pony` instance to it.

```
owner.ponies.append(pony)
```

If we do not have `backpopulates` set to the `owner` propery of the Pony class, then if you try to look at the owner of the pony like this:

```
print(pony.owner) # Returns None
```

Then it will still be None. If you set `backpopulates` to the owner, then this will get populates and stay in sync.

IMPORTANT: backpopulates just controls what happens with the objects *BEFORE* we commit them to the database.

It's always a good idea to setup your `backpopulates` properly so you aren't surprised.

Add data to the database, both single entities as well as related data

```
you = Owner(first_name="your first name",
            last_name="your last name",
            email="your email")

your_pony = Pony(name="your pony's name",
                 birth_year=2020,
                 breed="whatever you want",
                 owner=you)

# Note, id will be None until we commit
print(you.id)         # > None
print(your_pony.id)   # > None

session.add(you)      # Connects you and your_pony objects
session.commit()      # Saves data to the database

# After commiting the ids exist
print(you.id)         # > 4 (or whatever the new id is)
print(your_pony.id)   # > 4 (or whatever the new id is)
```

Using session with Flask-SQLAlchemy

We use this exactly the same as above but we get the session from the `db` instance.

```
db.session.add(you)      # Connects you and your_pony objects
db.session.commit()      # Saves data to the database
```

IMPORTANT don't confuse this session with the Flask session. This is a *database* session while flask session is the *browser* session.

Update data in the database

```
print(your_pony.birth_year)    # > 2020

# Updating is just like setting a property
your_pony.birth_year = 2019

# The pony instance updates immediately
print(your_pony.birth_year)    # > 2019

# but the database doesn't update until we commit!
session.commit()

print(your_pony.birth_year)    # > 2019
```

Delete data from the database (including cascades!)

Know how to use and specify the "delete-orphan" cascading strategy

```
# Just passing the owner instance to delete, deletes it, but....
db.session.delete(you)
# It doesn't actually change the database until you commit!
db.session.commit()
```

```
class Owner(db.Model):
    __tablename__ = 'owners'

    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(255))
    last_name = db.Column(db.String(255))
    email = db.Column(db.String(255))

    # This is a relationship between Ponies and Owner.
    # We have set it to cascase and delete orphans so
    # when we delete an owner all the ponies related to
    # that owner will be deleted
    ponies = db.relationship("Pony",
                         back_populates="owner",
                         cascade="all, delete-orphan")
```

Describe the purpose of a Query object

When you use SQLAlchemy's querying API, you're not actually immediately executing SQL against the database. Instead, all of the specifications that you add to the query are saved up into a single object that you then use to have SQL executed against the database. This allows you to make decisions at runtime about how you want to apply filters to the query. This will become clearer as you read about how to query and apply filters in the following sections. The important thing to note is that a Query object will not actually do anything with the database unless you explicitly tell it to do something.

Use a Session object to query the database using a model

With plain SQLAlchemy

```
pony_query = session.query(Pony)
print(pony_query)
```

```
pony_id_4_query = session.query(Pony).get(4)
```

With Flask SQLAlchemy

Flask SQLAlchemy attaches the `session.query` to the Model directly.

So you can re-write any call to `session.query` as `<Model>.query`.

```
# This plain SQLAlchemy query:
pony = session.query(Pony).get(4);

# Can be re-written as:
pony = Pony.query.get(4)
```

How to order your results

```
owner_query = Owner.query(Owner.first_name, Owner.last_name)
                    .order_by(Owner.last_name)
print(owner_query)
```

Use the filter method to find just what you want

```
pony_query = Pony.query.filter(Pony.name.like("%u%"))

pony_query = Pony.query
                .filter(Pony.name.ilike("%u%"))
                .filter(Pony.birth_year < 2015)
```

Use instance methods on the Query object to return a list or single item

- `all` - returns a list
- `first` - returns a single object
- `one` - returns a single object or raises an exception
- `one_or_none` - returns a single object or None

```
ponies = Pony.query.all()
for pony in ponies:
    print(pony.name)
```

Use the count method to … count

```
pony_query = Pony.query
print(pony_query.count())
```

Query objects with criteria on dependant objects

```
hirzai_owners = Owner.query \
                    .join(Pony)  \
                    .filter(Pony.breed == "Hirzai")

for owner in hirzai_owners:
    print(owner.first_name, owner.last_name)
```

Lazily load objects

```
for owner in Owner.query:
    print(owner.first_name, owner.last_name)
    for pony in owner.ponies:
        print(pony.name)
```

Eagerly load objects

```
owners_and_ponies = Owner.query.options(joinedload(Owner.ponies))

for owner in owners_and_ponies:
    print(owner.first_name, owner.last_name)
    for pony in owner.ponies:
        print(pony.name)
```

Install the Flask-SQLAlchemy extension to use with Flask

```
pipenv install Flask psycopg2-binary \
               SQLAlchemy Flask-SQLAlchemy
```

Configure SQLAlchemy using Flask-SQLAlchemy

Create a `SQLALCHEMYDATABASE_URI` property in your Flask app config

Then you can pass your app to SQLAlchemy for super simple apps

```
from config import Config
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object(Config)
# We are creating the DB in app.py after creating the app.
# So we can just pass our app to SQLAlchemy
db = SQLAlchemy(app)
```

However, if you've defined your db object BEFORE your app is created in another module, you must use the `init_app` method on db to configure Flask-SQLAlchemy

```
# models.py
from flask_sqlalchemy import SQLAlchemy

# notice we create the db instance without passing it app
db = SQLAlchemy()
```

```
# app.py
from flask import Flask
from .config import Configuration
# The act of importing this creates the db instance
from .models import db

# We create our app here
app = Flask(__name__)
app.config.from_object(Configuration)
# We use init_app and pass it the app
db.init_app(app)
```

Use the convenience functions and objects Flask-SQLAlchemy provides you to use in your code

Flask-SQLAlchemy adds the query object to every instance of a Model.

```
Pony.query.get(4)
```

It has some Flask specific things such as `get_or_404`, which just throws a
404 error if there's no Pony coming back from the database. There is also a similar `first_or_404` method.

```
Pony.query.get_or_404(4)
```

Flask-SQLAlchemy also adds the `session` object to the `db` instance.

```
db.session.add(owner)
db.session.commit()
```

Alembic Learning Objectives

Install Alembic into your project

```
pipenv install alembic
pipenv run alembic init <directory-name>
```

Configure Alembic to talk to your database and not have silly migration names

Add environment variable to `env.py`

Import the `os` module

```
import os
```

before `run_migrations_offline` add this line

```
config.set_main_option("sqlalchemy.url", os.environ.get("DATABASE_URL"))
```

Making better migration file names

You can set this in `alembic.ini` so your migration files will have dates in the names.

```
file_template = %%(year)d%%(month).2d%%(day).2d_%%(hour).2d%%(minute).2d%%(second).2d_%%(slug)s
```

Control Alembic's ability to migrate your database

Generating a migration (revision)

```
pipenv run alembic revision -m "create the owners table"
```

Running a migration (upgrading to a revision)

```
pipenv run alembic upgrade head
```

Rolling back a migration (downgrading to a revision)

```
pipenv run alembic downgrade <revision hash>
```

Rolling back all migrations (downgrading to base)

```
pipenv run alembic downgrade base
```

Viewing your migration history (revision history)

```
pipenv run alembic history
```

Reason about the way Alembic orders your migrations; and,

Alembic treats migrations like a linked list. It does not use the dates in the
filenames to decide which migrations to run and which order they get run.

Instead each revision has a revision hash, and each revision has a 'down_revision'
property that points at the previous revision. (except for the first revision which of course will have it's down_revision set to None)

```
revision = 'ddbf30c38165'
down_revision = 'e363377eb6d7'
```

Handle branching and merging concerns

If two teammates both commit new revisions, then you will end up with a conflict
in the down_revisions. Your revision linked list might look like this:

```
                    -- ae1027a6acf (Team A's most recent)
                   /
<-- 1975ea83b712 <--
                   \
                    -- 27c6a30d7c24 (Team B's most recent)
```

and you'll get an error like this:

```
FAILED: Multiple head revisions are present for given argument 'head';
please specify a specific target revision, '<branchname>@head' to
narrow to a specific head, or 'heads' for all heads
```

you can solve this with a merge specifying the two revisions

```
pipenv run alembic merge -m "merge contracts and devices" ae1027 27c6a
```

Configuring a Flask application to use Alembic;

```
pipenv install alembic Flask-Migrate
```

```
# app/__init__.py
from app.models import db
from flask import Flask
from config import Config
# We have to import flask_migrate
from flask_migrate import Migrate
import os

app = Flask(__name__)
# Load our config, make sure you set DATABASE_URL as flask migrate
# uses it as well
app.config.from_object(Config)
db.init_app(app)
# And we have to do this to configure Flask Migrate. It needs to know about
# both our app and our db object
Migrate(app, db)
```

Run commands to manage your database through the flask command; and,

When we use Flask-Migrate we run the commands through the flask command.

Instead of alembic init…

```
pipenv run flask db init
```

Check the help for the rest of the commands, which are the same as Alembic

```
pipenv run flask db --help

Usage: flask db [OPTIONS] COMMAND [ARGS]...

  Perform database migrations.

Options:
  --help  Show this message and exit.

Commands:
  branches   Show current branch points
  current    Display the current revision for each database.
  downgrade  Revert to a previous version
  edit       Edit a revision file
  heads      Show current available heads in the script directory
  history    List changeset scripts in chronological order.
  init       Creates a new migration repository.
  merge      Merge two revisions together, creating a new revision file
  migrate    Autogenerate a new revision file (Alias for 'revision...
  revision   Create a new revision file.
  show       Show the revision denoted by the given symbol.
  stamp      'stamp' the revision table with the given revision; don't run...
  upgrade    Upgrade to a later version
```

Autogenerate migrations from your models!

Instead of alembic migrate…

```
pipenv run flask db migrate -m "create owners table"
```

flask db migrate does **magic** now, it reads your models and tries to
autogenerate the migration files based on the model.

IMPORTANT always check the autogenerated migration though, as there's only
so much flask migrate can do and it might not get everything perfectly correct, but it is a time saver!