

```
Pet.query.delete()

# Add pets
whiskey = Pet(name='Whiskey', species="dog")
bowser = Pet(name='Bowser', species="dog", hunger=10)
spike = Pet(name='Spike', species="porcupine")

# Add new objects to session, so they'll persist
db.session.add(whiskey)
db.session.add(bowser)
db.session.add(spike)

# Commit--otherwise, this never gets saved!
db.session.commit()
```

Coming Up

SQLAlchemy II: relationships and joins

Learning More

SQLAlchemy Docs <<http://docs.sqlalchemy.org/en/latest/>>

Flask-SQLAlchemy Docs <<https://pythonhosted.org/Flask-SQLAlchemy/>>

SQLAlchemy

[Download Demo Code <../sqla-intro-demo.zip>](#)

Goals

Learn to use object-oriented techniques with relational DBs.

Without writing any SQL.

```
>>> whiskey = Pet(name='Whiskey', species="dog", hunger=50)

>>> whiskey.hunger
50

>>> whiskey.hunger = 20
```

SQLAlchemy Intro

SQLAlchemy ORM

- Popular, powerful, Python-based ORM (object-relational mapping)
- Translation service between OOP in our server language and relational data in our database
- Can use by itself, with Flask, or other web frameworks

Installing SQLAlchemy

Need the program that lets Python speak PostgreSQL: ***psycopg2***

Need the program that provides SQLAlchemy: ***flask-sqlalchemy***

```
$ pip install psycopg2-binary
$ pip install flask-sqlalchemy
```

OO into SQL

Model

A model like this:

demo/models.py

```

class Pet(db.Model):
    """Pet."""

    __tablename__ = "pets"

    id = db.Column(db.Integer,
                    primary_key=True,
                    autoincrement=True)
    name = db.Column(db.String(50),
                     nullable=False,
                     unique=True)
    species = db.Column(db.String(30), nullable=True)
    hunger = db.Column(db.Integer, nullable=False, default=20)

```

Would turn into this SQL:

```

CREATE TABLE pets (
  id SERIAL PRIMARY KEY,
  name VARCHAR(50) NOT NULL UNIQUE,
  species VARCHAR(30),
  hunger INTEGER NOT NULL DEFAULT 20
)

```

Setup

demo/models.py

```

from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

def connect_db(app):
    """Connect to database."""

    db.app = app
    db.init_app(app)

```

demo/app.py

```

from flask import Flask, request, redirect, render_template
from models import db, connect_db, Pet

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql:///sqla_intro'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SQLALCHEMY_ECHO'] = True

```

```
Pet.query.delete()

# Add pets
whiskey = Pet(name='Whiskey', species="dog")
bowser = Pet(name='Bowser', species="dog", hunger=10)
spike = Pet(name='Spike', species="porcupine")

# Add new objects to session, so they'll persist
db.session.add(whiskey)
db.session.add(bowser)
db.session.add(spike)

# Commit--otherwise, this never gets saved!
db.session.commit()
```

Coming Up

SQLAlchemy II: relationships and joins

Learning More

SQLAlchemy Docs <<http://docs.sqlalchemy.org/en/latest/>>

Flask-SQLAlchemy Docs <<https://pythonhosted.org/Flask-SQLAlchemy/>>

demo/app.py

```
@app.route("/", methods=["POST"])
def add_pet():
    """Add pet and redirect to list."""

    name = request.form['name']
    species = request.form['species']
    hunger = request.form['hunger']
    hunger = int(hunger) if hunger else None

    pet = Pet(name=name, species=species, hunger=hunger)
    db.session.add(pet)
    db.session.commit()

    return redirect(f"/{pet.id}")
```

Demo: Detail

demo/app.py

```
@app.route("/<int:pet_id>")
def show_pet(pet_id):
    """Show info on a single pet."""

    pet = Pet.query.get_or_404(pet_id)
    return render_template("detail.html", pet=pet)
```

demo/templates/detail.html

```
<h1>{{ pet.name }}</h1>
<p>Species: {{ pet.species }}</p>
<p>Hunger: {{ pet.hunger }}</p>
<p>{{ pet.name }} says {{ pet.greet() }}!</p>
<a href="/">Go back</a>
```

Demo: Seeding

demo/seed.py

```
"""Seed file to make sample data for pets db."""

from models import Pet, db
from app import app

# Create all tables
db.drop_all()
db.create_all()

# If table isn't empty, empty it
```

```
connect_db(app)
```

- **SQLALCHEMY_DATABASE_URI** - Where is your database?
- **SQLALCHEMY_TRACK_MODIFICATIONS** - Set this to false or SQLAlchemy will yell at you
- **SQLALCHEMY_ECHO** - Print all SQL statements to the terminal (helpful for debugging)
- Can talk to SQLite, PostgreSQL, MySQL, and more
- You (almost) never have to change code if you change databases

Models

Our Model

demo/models.py

```
class Pet(db.Model):  
    """Pet."""  
  
    __tablename__ = "pets"  
  
    id = db.Column(db.Integer,  
                   primary_key=True,  
                   autoincrement=True)  
    name = db.Column(db.String(50),  
                     nullable=False,  
                     unique=True)  
    species = db.Column(db.String(30), nullable=True)  
    hunger = db.Column(db.Integer, nullable=False, default=20)
```

- All models should subclass **db.Model**

- Specify the tablename with `__tablename__`

demo/models.py

```
class Pet(db.Model):  
    """Pet."""  
  
    __tablename__ = "pets"  
  
    id = db.Column(db.Integer,  
                   primary_key=True,  
                   autoincrement=True)  
    name = db.Column(db.String(50),  
                     nullable=False,  
                     unique=True)  
    species = db.Column(db.String(30), nullable=True)
```

```
hunger = db.Column(db.Integer, nullable=False, default=20)
```

- Have to specify the type of column
- Columns can contain **NULL** unless `nullable=False`
- Can specify **default, unique, primary_key, autoincrement**

Creating the Database

```
$ ipython3
In [1] %run app.py

In [2] db.create_all()
```

- Create all the tables using this database connection
- Only have to do once
 - No effect if tables already exist
- If you change table schema
 - drop table & re-run

Note: Do I always have to drop the table?

Dropping all of your tables may seem like an extreme move every time you make a change to your schema. There are tools that can help you update your schema more smoothly. *Database migrations* are a common way to do this, but this topic is beyond our scope.

Using our Model

```
>>> fluffy = Pet(name='Fluffy', species="Pet")
>>> fluffy.hunger
20

>>> db.session.add(fluffy)      # required to add to database!
>>> db.session.commit()         # commit the transaction
```

You only have to use `db.session.add()` to add a new object once – you don't need to keep adding it to the session each time you change it.

Note: Transactions

Database management systems (Postgres included) support the concept of **transactions**. The idea here is that you may want to update multiple parts of the database simultaneously, and if any piece of the update fails, the entire transaction fails.

Flask-SQLAlchemy

- Add-on product to integrate Flask and SQLAlchemy
- Benefits
 - Ties SQLAlchemy session to Flask response
 - Simplifies finding things in SQLAlchemy API
 - Simplifies querying by allowing on class

Differences

- With Flask-SQLAlchemy, all useful methods are on **db**.
 - With vanilla SQLAlchemy, stuff is spread all over
 - There are useful web-related methods, like `Pet.objects.get_or_404(pk)`

Demo

Demo: Listing

demo/app.py

```
@app.route("/")
def list_pets():
    """List pets and show add form."""

    pets = Pet.query.all()
    return render_template("list.html", pets=pets)
```

demo/templates/list.html

```
<ul>
  {% for pet in pets %}
  <li><a href="/{{ pet.id }}">{{ pet.name }}</a></li>
  {% endfor %}
</ul>
```

Demo: Adding

demo/templates/list.html

```
<form method="POST">
  <p>Name:      <input name="name"></p>
  <p>Species:   <input name="species"></p>
  <p>Hunger:    <input name="hunger"></p>
  <button>Save</button>
</form>
```


Note: Class Methods

- Most methods are “instance methods”
 - These are called on an instance of a class (ie, a single cat)
 - They can refer to/change attributes of that instance
- Some methods are “class methods”
 - They are called on the **class itself**
 - They can't refer to instance attributes
 - Often used as “factories” to return instances

```
class Pet(db.Model):  
  
    @classmethod  
    def get_by_species(cls, species):  
        """Get all pets matching that color."""  
  
        return cls.query.filter(Pet.species == species).all()
```

```
>>> Pet.get_by_species("dog")  
[<Pet ...>, <Pet ...>]
```

Better Representation

```
>>> Pet.query.filter(Pet.species == 'dog').all()  
[<__main__.Pet object at ...>, <__main__.Pet object at ...>]
```

Yeah, but *which pet* is that?

demo/models.py

```
class Pet(db.Model):    # ...  
    def __repr__(self):  
        """Show info about pet."""  
  
        p = self  
        return f"<Pet {p.id} {p.name} {p.species} {p.hunger}>"
```

```
>>> Pet.query.get(1)  
<Pet 1 Whiskey dog 10>
```

SQLAlchemy with Flask

The most common example is a bank transfer: imagine Abby is trying to send \$20 to Barbara, and we want to record this fact in a database. So we deduct \$20 from Abby's account, but before we can increase Barbara's balance by \$20, there's a power failure. In this case, the whole transaction should be cancelled. Otherwise, Abby would be out \$20!

In PostgreSQL, we can begin a transaction with `BEGIN TRANSACTION`. Inside of our transaction, any SQL we write won't make permanent changes to the database. If we make a change we don't like, we can cancel the transaction with the `ROLLBACK` command.

But more importantly for our present purposes, if there's a change we do like, we need to `COMMIT` the transaction.

Here's a small example you can explore in the demo code:

demo/colors.sql

```
-- from the terminal run:
-- psql < colors.sql

DROP DATABASE IF EXISTS colors;

CREATE DATABASE colors;

\c colors

CREATE TABLE colors
(
    id SERIAL PRIMARY KEY,
    name TEXT
);

INSERT INTO colors (name) VALUES ('red'), ('blue'), ('green');

BEGIN TRANSACTION;
    DELETE FROM colors;
    SELECT * FROM colors;
    -- no colors are left!
ROLLBACK;

SELECT * FROM colors;
-- all the colors are still here!
-- we only removed them in a rolled back transaction.

BEGIN TRANSACTION;
    DELETE FROM colors;
    SELECT * FROM colors;
    -- no colors are left!
COMMIT;

SELECT * FROM colors;
-- Since we committed the transaction,
```

```
-- all of the colors are gone.
```

Querying

```
Pet.query.all()
```

```
SELECT *  
FROM pets
```

```
Pet.query.get(1)
```

```
SELECT *  
FROM pets  
WHERE id = 1
```

```
Pet.query.filter_by(species='dog').all()
```

```
SELECT *  
FROM pets  
WHERE species = 'dog'
```

```
Pet.query.filter(Pet.species == 'dog').all()
```

```
Pet.query.filter(Pet.hunger < 10).all()
```

```
SELECT *  
FROM pets  
WHERE hunger < 10
```

```
Pet.query.filter(Pet.hunger < 15,  
                 Pet.species == 'dog').all()
```

```
SELECT *  
FROM pets  
WHERE hunger < 15  
      AND species = 'dog'
```

Fetching Records

.get(pk)

Get single record with that primary key value

.all()

Get all records as a list

.first()

Get first record or **None**

.one()

Get first record, error if 0 or if > 1

.one_or_none()

Get first record, error if >1, None if 0

Methods

demo/models.py

```
class Pet(db.Model):
    """Pet."""

    __tablename__ = "pets"

    id = db.Column(db.Integer,
                    primary_key=True,
                    autoincrement=True)
    name = db.Column(db.String(50),
                     nullable=False,
                     unique=True)
    species = db.Column(db.String(30), nullable=True)
    hunger = db.Column(db.Integer, nullable=False, default=20)

    def greet(self):
        """Greet using name."""

        return f"I'm {self.name} the {self.species or 'thing'}"

    def feed(self, units=10):
        """Nom nom nom."""

        self.hunger -= units
        self.hunger = max(self.hunger, 0)
```

Using Our Methods

```
>>> fluffy.greet()
'I am Fluffy the cat'

>>> fluffy.feed()
>>> fluffy.hunger
>>> db.session.commit()    # save new hunger
```