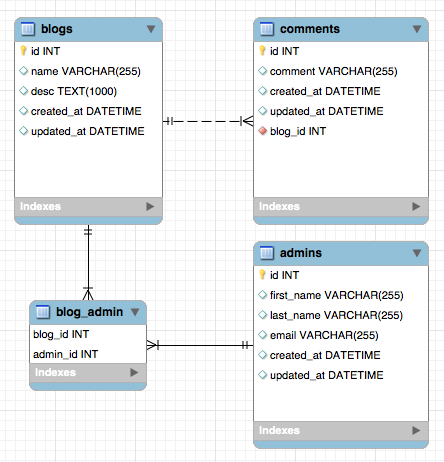
Models in the MTV architecture do all the work of data manipulation. They control which data are released to the controllers. Because of this, the phrase **skinny controllers and fat models** is often used, and is an important design principle:

Any heavy logic including database queries is the purview of a Model. If a Controller (in Django's case, a View) needs to perform logic or get information from a database, it should use a Model method to do so.

**Blogs, Comments, Admins Example**

For our blogs app, let's say that we wanted people to not only create blogs but also add comments.  Let's also allow a blog to be owned by multiple admins and where an admin can also have multiple blogs (in other words, a many to many relationship).



Note that a blog can have many comments (one to many relationship) and a blog can have many admins as well as one admin having multiple blogs (many to many relationship).

To implement this in Django, you would have the following code in your models.py:  (as Django likes to keep the model class singular, we'll follow their convention)

# Inside models.py

from \_\_future\_\_ import unicode\_literals

from django.db import models

# Create your models here.

class Blog(models.Model):

name = models.CharField(max\_length=255)

desc = models.TextField()

created\_at = models.DateTimeField(auto\_now\_add = True)

updated\_at = models.DateTimeField(auto\_now = True)

class Comment(models.Model):

comment = models.CharField(max\_length=255)

created\_at = models.DateTimeField(auto\_now\_add = True)

updated\_at = models.DateTimeField(auto\_now = True)

# Notice the association made with ForeignKey for a one-to-many relationship

# There can be many comments to one blog

blog = models.ForeignKey(Blog, related\_name = "comments")

class Admin(models.Model):

first\_name = models.CharField(max\_length=255)

last\_name = models.CharField(max\_length=255)

email = models.CharField(max\_length=255)

blogs = models.ManyToManyField(Blog, related\_name = "admins")

created\_at = models.DateTimeField(auto\_now\_add = True)

updated\_at = models.DateTimeField(auto\_now = True)

**That ForeignKey statement is the equivalent of the one-to-many relationship shown in the ERD diagram.**

**Column types**

You can find a full list of allowed column types [in the documentation](https://docs.djangoproject.com/en/1.10/ref/models/fields/#field-types), but here are some of the main ones you'll be using.

* CharField
  + Any text that a user may enter. This has one additional required parameter, max\_length, that (unsurprisingly) is the maximum length of text that can be saved.
* TextField
  + Like a CharField, but with no maximum length. Your user could copy the entire text of the Harry Potter series into the field, and it would save in the database correctly.
* IntegerField, BooleanField
  + Holds integers or booleans, respectively
* DateTimeField
  + Used for date and times, such as timestamps or when a flight is scheduled to depart. This field can take two very useful optional parameters, auto\_now\_add=True, which adds the current date/time when object is created, and auto\_now=True, which automatically updates any time the object is modified.
* ForeignKey, ManyToManyField, OneToOneField
  + Used to indicate a relationship between models (anything that would require a JOIN statement in SQL). ForeignKey is for one-to-many relationships and goes in the model on the "many" side, just as the foreign key column goes in the SQL table on the "many" side.

**Migrations**

Now think back to Flask. Creating an ERD diagram didn't actually create the database table(s). We had to *forward engineer* the diagram into SQL code, and then *run* that SQL code in a MySQL environment. Instead of using a separate program to do this Django can do the whole job for us with minimal code.

To do the equivalent of forward engineer we are going to run a couple of commands from the terminal.

> python manage.py makemigrations

> python manage.py migrate

***makemigrations*** is a kind of staging. When this command runs Django is looking at the code we wrote, finding any changes, and formulating the correct Python code to move on to the next step.

***migrate*** actually applies the changes made above. This step is where the SQL queries are actually built and executed and is equivalent to forward engineering an ERD.

The migration process is split into two steps so that Django can check and make sure you wrote code it can understand before moving on to the next step.

***Note:***

1. ***Never****delete migration files*, and **always** makemigrations and migrate anytime you change something in your models.pyfiles – that's what updates the actual database so it reflects what's in your models.
2. For now we are going to be using **SQLite** - a SQL database that comes packaged with Django and has much of the functionality of MySQL (but not all of it) and is stored as *local memory* in a file, and as such is very fast. This type of database is best used in development, like we're doing now.
3. For deployment you'll have the option to use SQLite, but we'll also show you how to add a **MySQL** database, which you've already used with Flask.
4. *Django ORM models and queries will always be the same no matter which database we are using.*

Now we've seen how to build a model and migrate in order to translate our code to SQL.