DJANGO MODELS

06016322 - WEB PROGRAMMING

CONNECTING TO MYSQL DATABASE

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'webpro',
        'USER': 'root',
        'PASSWORD': 'dbpass',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

IMPORTANT COMMANDS

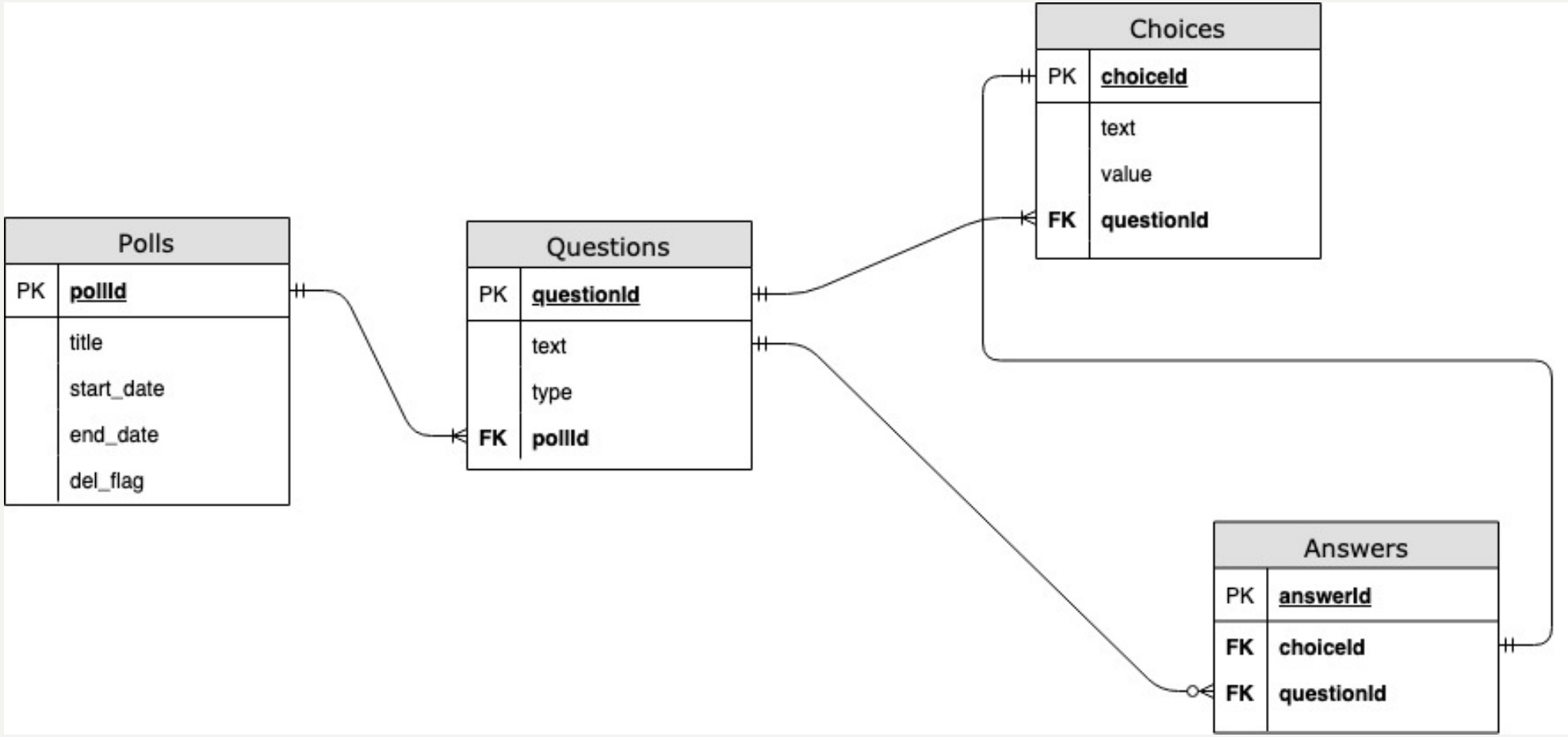
Python manage.py migrate

– to migrate changes in your model to the connected database

Python manage.py makemigrations

– to create migration files for changes in your models.py files

THE ER DIAGRAM FOR THE POLLS APP



COMMONLY USED FIELD TYPES

- **CharField**
  - CharField has one extra required argument: max_length
- **BooleanField**
- **DateTimeField**
- **IntegerField**
- **EmailField**
- **FileField**

- Field options
  - null
  - blank
  - choices
  - default
  - unique
  - primary_key

https://docs.djangoproject.com/en/2.1/ref/models/fields/

RELATIONSHIP FIELDS

- **ForeignKey**
  - A many-to-one relationship. Requires two positional arguments: the class to which the model is related and the on_delete option.

```
manufacturer = models.ForeignKey(
    'Manufacturer',
    on_delete=models.CASCADE,
)
```

  - The possible values for on_delete: CASCADE, PROTECT, SET_NULL, SET_DEFAULT, SET(), DO_NOTHING

RELATIONSHIP FIELDS

ManyToManyField

- A many-to-many relationship. Requires a positional argument: the class to which the model is related, which works exactly the same as it does for ForeignKey.
- Behind the scenes, Django creates an intermediary join table to represent the many-to-many relationship.

OneToOneField

- Conceptually, this is similar to a **ForeignKey** with **unique=True**, but the "reverse" side of the relation will directly return a single object.

LET'S DO SOME QUERY!

- The object django.db.connection represents the default database connection.
- To use the database connection, call connection.cursor() to get a cursor object.
- Then, call cursor.execute(sql, [params]) to execute the SQL.
- Use cursor.fetchone() or cursor.fetchall() to return the resulting rows.

```
from django.db import connection

def my_custom_sql(self):
    with connection.cursor() as cursor:
        cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
        cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
        row = cursor.fetchone()

    return row
```

DJANGO DATABASE API

https://docs.djangoproject.com/en/2.1/topics/db/queries/

MODELS.PY OF EXAMPLES IN THIS SLIDE

```python
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

    def __str__(self):
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):
        return self.headline
```

RETRIEVING OBJECTS

- To retrieve objects from your database, construct a QuerySet via a Manager on your model class.
- A QuerySet represents a collection of objects from your database.
- In SQL terms, a QuerySet equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT.

RETRIEVING ALL OBJECTS

- The all() method returns a QuerySet of all the objects in the database.

```
>>> all_entries = Entry.objects.all()
```

SELECT * FROM table_name

RETRIEVING SPECIFIC OBJECTS WITH FILTERS

- The two most common ways to refine a QuerySet are:
  - filter(**kwargs)
    - Returns a new QuerySet containing objects that match the given lookup parameters.
  - exclude(**kwargs)
    - Returns a new QuerySet containing objects that do not match the given lookup parameters.

```
Entry.objects.filter(pub_date__year=2006)
```

SELECT * FROM table_name WHERE condition

RETRIEVING A SINGLE OBJECT WITH GET()

- **filter()** will always give you a QuerySet, even if only a single object matches the query - in this case, it will be aQuerySet containing a single element.

- If you know there is only one object that matches your query, you can use the get() method which returns the object directly:

```
>>> one_entry = Entry.objects.get(pk=1)
```

FIELD LOOKUPS

- Field lookups are how you specify the meat of an SQL WHERE clause.
- Basic lookups keyword arguments take the form field__lookuptype=value (That's a double-underscore).

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

COMMONLY USED LOOKUP TYPES

- exact

```
Entry.objects.get(id__exact=14)
Entry.objects.get(id__exact=None)
```

```
SELECT ... WHERE id = 14;
SELECT ... WHERE id IS NULL;
```

- iexact

```
Blog.objects.get(name__iexact='beatles blog')
Blog.objects.get(name__iexact=None)
```

- contains

```
Entry.objects.get(headline__contains='Lennon')
```

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

COMMONLY USED LOOKUP TYPES

- in

```
Entry.objects.filter(id__in=[1, 3, 4])
Entry.objects.filter(headline__in='abc')
```

- gt – greater than
- gte – greater than or equal
- lt – less than
- lte – less than or equal

```
Entry.objects.filter(id__gt=4)
```

```
SELECT ... WHERE id > 4;
```

- isnull

```
Entry.objects.filter(pub_date__isnull=True)
```

- range

```
import datetime
start_date = datetime.date(2005, 1, 1)
end_date = datetime.date(2005, 3, 31)
Entry.objects.filter(pub_date__range=(start_date, end_date))
```

LOOKUPS THAT SPAN RELATIONSHIPS

- Django offers a powerful and intuitive way to "follow" relationships in lookups, taking care of the **SQL JOINs** for you automatically, behind the scenes.

- This example retrieves all Entry objects with a Blog whose name is 'Beatles Blog':

```
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

CREATING OBJECTS

- To represent database-table data in Python objects, Django uses an intuitive system:
  - A model class represents a database table, and an instance of that class represents a particular record in the database table.

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

  - This performs an INSERT SQL statement behind the scenes.
  - Django doesn't hit the database until you explicitly call save().

SAVING CHANGES TO OBJECTS

- To save changes to an object that's already in the database, use save()

```
>>> b5.name = 'New name'
>>> b5.save()
```

- This performs an UPDATE SQL statement behind the scenes.

SAVING FOREIGN KEY AND MANY-TO-MANY FIELDS

```
>>> from blog.models import Blog, Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

- Updating a ManyToManyField works a little differently – use the add() method on the field to add a record to the relation.

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```