**Database setup**

Open up **mysite/settings.py**. It’s a normal Python module with module-level variables representing Django settings.

By default, the configuration uses SQLite.

SQLite is included in Python, so you won’t need to install anything else to support your database.

[**ENGINE**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-DATABASE-ENGINE) –

**'django.db.backends.sqlite3'**, or

**'django.db.backends.postgresql'**, or

**'django.db.backends.mysql'**, or

**'django.db.backends.oracle'**. etc.

[**NAME**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-NAME) –

The name of your database. If you’re using SQLite, the database will be a file on your computer; in that case, [**NAME**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-NAME) should be the full absolute path, including filename, of that file.

The default value, **os.path.join(BASE\_DIR, 'db.sqlite3')**, will store the file in your project directory.

If you are using MYSql as your database, additional settings such as [**USER**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-USER), [**PASSWORD**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-PASSWORD), and [**HOST**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-HOST) must be added.

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': 'myproject',

'USER': 'root',

'PASSWORD': 'root',

'HOST': '127.0.0.1',

'PORT': 3306'', }}

To use database, run the following command:

**$ python manage.py migrate**

The [**migrate**](https://docs.djangoproject.com/en/2.0/ref/django-admin/#django-admin-migrate) command looks at the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-INSTALLED_APPS) setting and creates any necessary database tables according to the database settings in your **mysite/settings.py**.

**Creating models**

Now we’ll define your models – essentially, your database layout, with additional metadata.

Edit the **app/models.py** file so it looks like this:

**Example**

**from django.db import models  
*# Create your models here.*class Student(models.Model):  
 student\_idno = models.IntegerField(default=10)  
 join\_date = models.DateTimeField("date joined")  
  
class Student\_Details(models.Model):  
 models.ForeignKey(Student,on\_delete=models.CASCADE)  
 student\_name = models.CharField(max\_length=50)  
 student\_course = models.DecimalField(max\_digits=10,decimal\_places=2)**

**Note**: A foreign key with **cascade** delete means that if a record in the parent table is deleted, then the corresponding records in the child table will automatically be deleted.

But first we need to tell our project that the **app**  is installed.

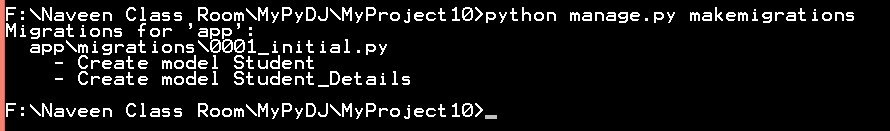
Add your app to **settings.py**

INSTALLED\_APPS = [  
 **..........**,  
 **'app'**]

Now Django knows to include the app.

Let’s run another command:

**$** python manage.py makemigrations



By running **makemigrations**, you’re telling Django that you’ve made some changes to your models (in this case, you’ve made new ones) and that you’d like the changes to be stored as a *migration*.

Table names are automatically generated by combining the name of the app (app) and the lowercase name of the model.

Primary keys (IDs) are added automatically. (You can override this, too.)

By convention, Django appends "\_id" to the foreign key field name. (Yes, you can override this, as well.)

**Introducing the Django Admin**

**Creating an admin user**

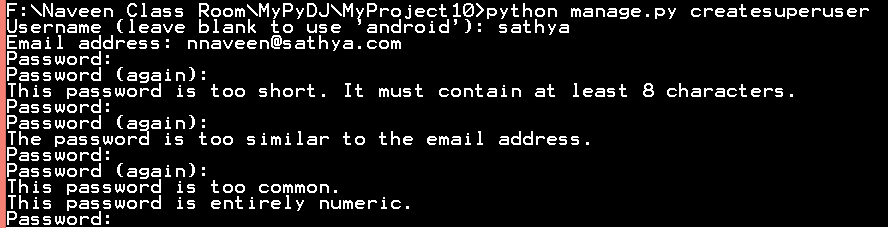
**$** python manage.py createsuperuser

Enter your desired username and press enter.

Username: admin

You will then be prompted for your desired email address:

Email address: admin@example.com





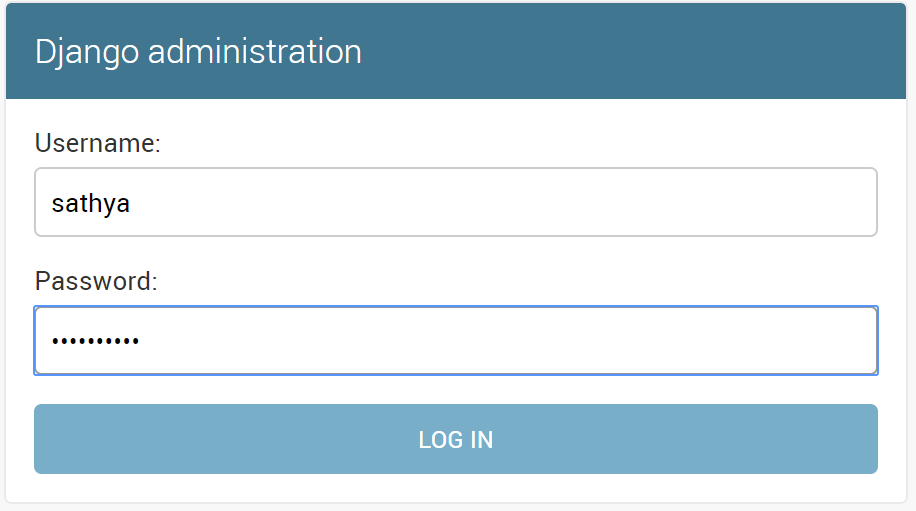
**Start the development server**

The Django admin site is activated by default. Let’s start the development server and explore it.

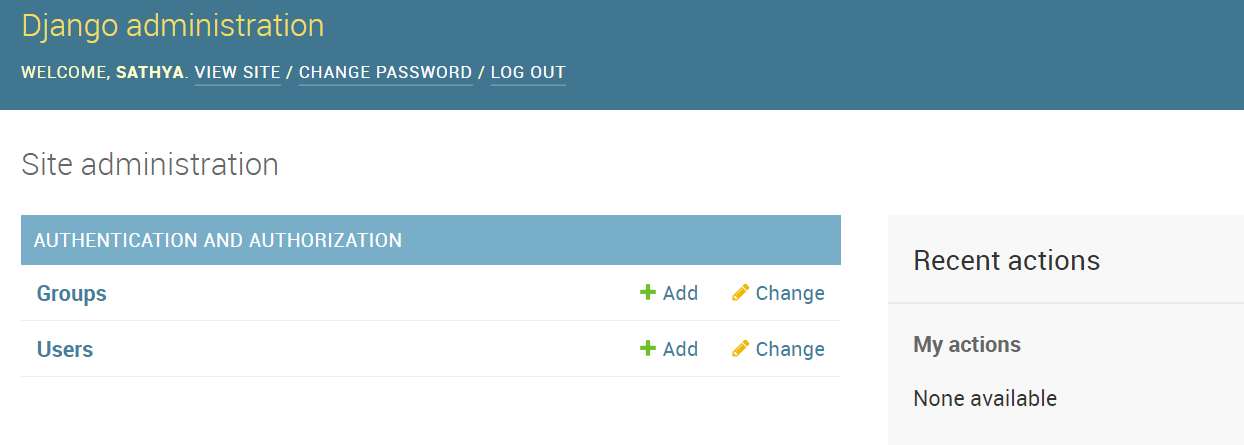
If the server is not running start it like so:

**$** python manage.py runserver

Now, open a Web browser and go to “/admin/” on your local domain – e.g.,<http://127.0.0.1:8000/admin/>. You should see the admin’s login screen:



After login The Home Screen



You should see a few types of editable content: groups and users. They are provided by **[django.contrib.auth](https://docs.djangoproject.com/en/2.0/topics/auth/" \l "module-django.contrib.auth" \o "django.contrib.auth: Django's authentication framework.)**, the authentication framework shipped by Django.

**Make the poll app modifiable in the admin**

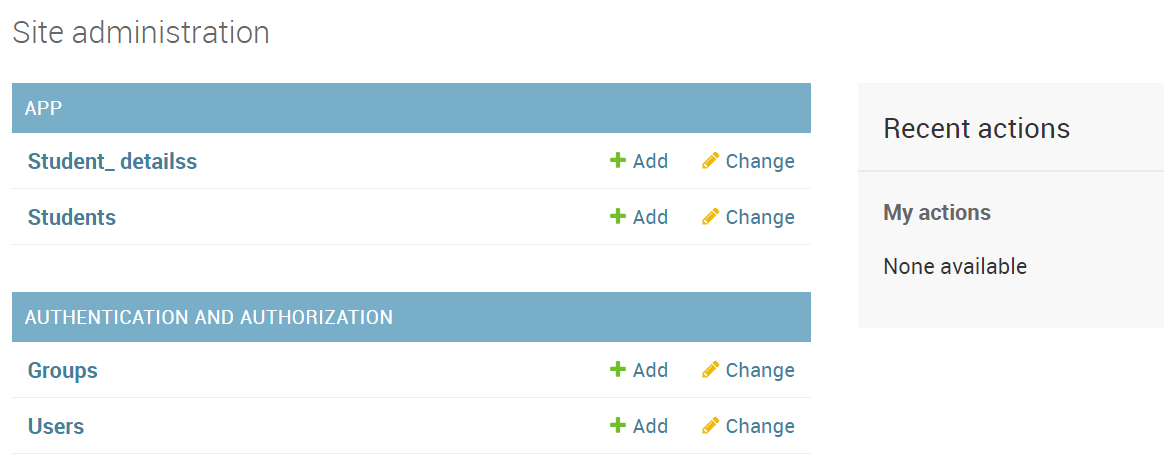
But where’s our app? It’s not displayed on the admin index page.

Just one thing to do: we need to tell the admin that **Question** objects have an admin interface. To do this, open the **app/admin.py** file, and edit it to look like this:

app/admin.py

**from django.contrib import admin  
  
from .models import Student  
from .models import Student\_Details  
  
admin.site.register(Student)  
admin.site.register(Student\_Details)**

**Note:** Save the file and refresh the browser



**Django Database Operation methods**

**Creating objects**

In Python A model class represents a database table, and an instance of that class represents a particular record in the database table.

To create an object, instantiate it using keyword arguments to the model class, then call [**save()**](https://docs.djangoproject.com/en/2.0/ref/models/instances/#django.db.models.Model.save) to save it to the database.

**Example:**

b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')

b.save()

**Saving changes to objects**

To save changes to an object that’s already in the database, use [**save()**](https://docs.djangoproject.com/en/2.0/ref/models/instances/#django.db.models.Model.save).

**>>>** b5.name = 'New name'

**>>>** b5.save()

**Retrieving objects**

To retrieve objects from your database, construct a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** via a [**Manager**](https://docs.djangoproject.com/en/2.0/topics/db/managers/#django.db.models.Manager) on your model class.

A **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** represents a collection of objects from your database. It can have zero, one or many *filters*. Filters narrow down the query results based on the given parameters. In SQL terms, a[**QuerySet**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet) equates to a **SELECT** statement, and a filter is a limiting clause such as **WHERE** or **LIMIT**.

You get a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** by using your model’s [**Manager**](https://docs.djangoproject.com/en/2.0/topics/db/managers/#django.db.models.Manager). Each model has at least one [**Manager**](https://docs.djangoproject.com/en/2.0/topics/db/managers/#django.db.models.Manager), and it’s called [**objects**](https://docs.djangoproject.com/en/2.0/ref/models/class/#django.db.models.Model.objects) by default. Access it directly via the model class, like so:

**>>>** Blog.objects

<django.db.models.manager.Manager object at ...>

**>>>** b = Blog(name='Foo', tagline='Bar')

**>>>** b.objects

Traceback:

...

AttributeError: "Manager isn't accessible via Blog instances."

The [**Manager**](https://docs.djangoproject.com/en/2.0/topics/db/managers/#django.db.models.Manager) is the main source of **QuerySets** for a model. For example, **Blog.objects.all()** returns a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** that contains all **Blog** objects in the database.

**Retrieving all objects**

To do this, use the [**all()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.all)method on a [**Manager**](https://docs.djangoproject.com/en/2.0/topics/db/managers/#django.db.models.Manager):

**>>>** all\_entries = Entry.objects.all()

The [**all()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.all) method returns a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** of all the objects in the database.

**Retrieving specific objects with filters**

The **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** returned by [**all()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.all) describes all objects in the database table. Usually, though, you’ll need to select only a subset of the complete set of objects.

To create such a subset, you refine the initial **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)**, adding filter conditions. The two most common ways to refine a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** are:

**filter(\*\*kwargs)**

Returns a new **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** containing objects that match the given lookup parameters.

**exclude(\*\*kwargs)**

Returns a new **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** containing objects that do *not* match the given lookup parameters.

For example, to get a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** of blog entries from the year 2006, use [**filter()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.filter) like so:

Entry.objects.filter(pub\_date\_\_year=2006)

**Retrieving a single object with get()**

[**filter()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.filter) will always give you a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)**, even if only a single object matches the query - in this case, it will be a **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** containing a single element.

If you know there is only one object that matches your query, you can use the [**get()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.get) method on a[**Manager**](https://docs.djangoproject.com/en/2.0/topics/db/managers/#django.db.models.Manager) which returns the object directly:

**>>>** one\_entry = Entry.objects.get(pk=1)

**Limiting QuerySets**

Use a subset of Python’s array-slicing syntax to limit your **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** to a certain number of results. This is the equivalent of SQL’s **LIMIT** and **OFFSET** clauses.

For example, this returns the first 5 objects (**LIMIT 5**):

**>>>** Entry.objects.all()[:5]

This returns the sixth through tenth objects (**OFFSET 5 LIMIT 5**):

**>>>** Entry.objects.all()[5:10]

Negative indexing (i.e. **Entry.objects.all()[-1]**) is not supported.

**Field lookups**

Field lookups are how you specify the meat of an SQL **WHERE** clause. They’re specified as keyword arguments to the **[QuerySet](https://docs.djangoproject.com/en/2.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)** methods [**filter()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.filter), [**exclude()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.exclude) and [**get()**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.get).

Basic lookups keyword arguments take the form **field\_\_lookuptype=value**. (That’s a double-underscore).

For example:

**>>>** Entry.objects.filter(pub\_date\_\_lte='2006-01-01')

roughly SQL:

**SELECT** \* **FROM** blog\_entry **WHERE** pub\_date <= '2006-01-01';

[**exact**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#std:fieldlookup-exact)

An “exact” match. For example:

**>>>** Entry.objects.get(headline\_\_exact="Cat bites dog")

Would generate SQL along these lines:

**SELECT** ... **WHERE** headline = 'Cat bites dog';

the following two statements are equivalent:

**>>>** Blog.objects.get(id\_\_exact=14)

**>>>** Blog.objects.get(id=14)

[**iexact**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#std:fieldlookup-iexact)

A case-insensitive match. So, the query:

**>>>** Blog.objects.get(name\_\_iexact="beatles blog")

Would match a **Blog** titled **"Beatles Blog"**, **"beatles blog"**, or even **"BeAtlES blOG"**.

[**contains**](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#std:fieldlookup-contains)

Case-sensitive containment test. For example:

Entry.objects.get(headline\_\_contains='Lennon')

Roughly translates to this SQL:

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

Note this will match the headline **'Today Lennon honored'** but not **'today lennon honored'**.

https://docs.djangoproject.com/en/2.0/intro/tutorial03/