**Python - Django Framework for Web Development Part 2**

# **Django with MySql Database**

## **Requirements**:

1. **MySql Server (e.g. WAMP Server)**

<http://wampserver.aviatechno.net/>

1. **MySql Connector Python**

Download MySQL Connector for Python and install: <https://dev.mysql.com/downloads/connector/python/>

1. **MySqlClient:**

Open command prompt or terminal and run this command to install MySqlClient:

**pip install mysqlclient**

1. **Create Database**

Graphical user interface, text, application, chat or text message

Description automatically generated

## **Django Project Database Configuration**

1. Open **mysite/settings.py** file and find the **DATABASES = {'NAME': BASE\_DIR / 'db.sqlite3',}**
2. Replace it with the following MySQL Drivers:

DATABASES = {

    'default': {

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

        'NAME': 'mysite', #database name

        'USER': 'root',

        'PASSWORD': '',

        'HOST': '127.0.0.1',

        'PORT': '3306',

    }

}

**Sample:**

A screenshot of a computer

Description automatically generated with medium confidence

If you wish to use another database, install the appropriate [database bindings](https://docs.djangoproject.com/en/4.0/topics/install/#database-installation) and change the following keys in the [**DATABASES**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-DATABASES) **'default'** item to match your database connection settings:

* [**ENGINE**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-DATABASE-ENGINE) – Either **'django.db.backends.sqlite3'**, **'django.db.backends.postgresql'**, **'django.db.backends.mysql'**, or **'django.db.backends.oracle'**. Other backends are [also available](https://docs.djangoproject.com/en/4.0/ref/databases/#third-party-notes).
* [**NAME**](https://docs.djangoproject.com/en/4.0/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/4.0/ref/settings/#std:setting-NAME) should be the full absolute path, including filename, of that file. The default value, **BASE\_DIR / 'db.sqlite3'**, will store the file in your project directory.

If you are not using SQLite as your database, additional settings such as [**USER**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-USER), [**PASSWORD**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-PASSWORD), and [**HOST**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-HOST) must be added. For more details, see the reference documentation for [**DATABASES**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-DATABASES).

Also, note the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-INSTALLED_APPS) setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects.

By default, [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-INSTALLED_APPS) contains the following apps, all of which come with Django:

* [**django.contrib.admin**](https://docs.djangoproject.com/en/4.0/ref/contrib/admin/#module-django.contrib.admin) – The admin site. You’ll use it shortly.
* [**django.contrib.auth**](https://docs.djangoproject.com/en/4.0/topics/auth/#module-django.contrib.auth) – An authentication system.
* [**django.contrib.contenttypes**](https://docs.djangoproject.com/en/4.0/ref/contrib/contenttypes/#module-django.contrib.contenttypes) – A framework for content types.
* [**django.contrib.sessions**](https://docs.djangoproject.com/en/4.0/topics/http/sessions/#module-django.contrib.sessions) – A session framework.
* [**django.contrib.messages**](https://docs.djangoproject.com/en/4.0/ref/contrib/messages/#module-django.contrib.messages) – A messaging framework.
* [**django.contrib.staticfiles**](https://docs.djangoproject.com/en/4.0/ref/contrib/staticfiles/#module-django.contrib.staticfiles) – A framework for managing static files.

These applications are included by default as a convenience for the common case.

Some of these applications make use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

py manage.py migrate

The [**migrate**](https://docs.djangoproject.com/en/4.0/ref/django-admin/#django-admin-migrate) command looks at the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-INSTALLED_APPS) setting and creates any necessary database tables according to the database settings in your **mysite/settings.py** file and the database migrations shipped with the app.

Text

Description automatically generated

After **migrate** command:

Text

Description automatically generated

Ten (10) tables form INSTALLED\_APPS added to your database. (6 tables from Authentication apps/system)

Graphical user interface, application

Description automatically generated

## [**Models**](https://docs.djangoproject.com/en/4.0/topics/db/models/)

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you’re storing. Generally, each model maps to a single database table.

The basics:

* Each model is a Python class that subclasses **[django.db.models.Model](https://docs.djangoproject.com/en/4.0/ref/models/instances/" \l "django.db.models.Model" \o "django.db.models.Model)**.
* Each attribute of the model represents a database field.
* With all of this, Django gives you an automatically generated database-access API

**Quick Example:**

This example model defines a **Person**, which has a **first\_name** and **last\_name**:

**from** **django.db** **import** models

**class** **Person**(models.Model):

first\_name = models.CharField(max\_length=30)

last\_name = models.CharField(max\_length=30)

**first\_name** and **last\_name** are [fields](https://docs.djangoproject.com/en/4.0/topics/db/models/#fields) of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above **Person** model would create a database table like this:

**CREATE** **TABLE** myapp\_person (

"id" serial **NOT** **NULL** **PRIMARY** **KEY**,

"first\_name" varchar(30) **NOT** **NULL**,

"last\_name" varchar(30) **NOT** **NULL**

);

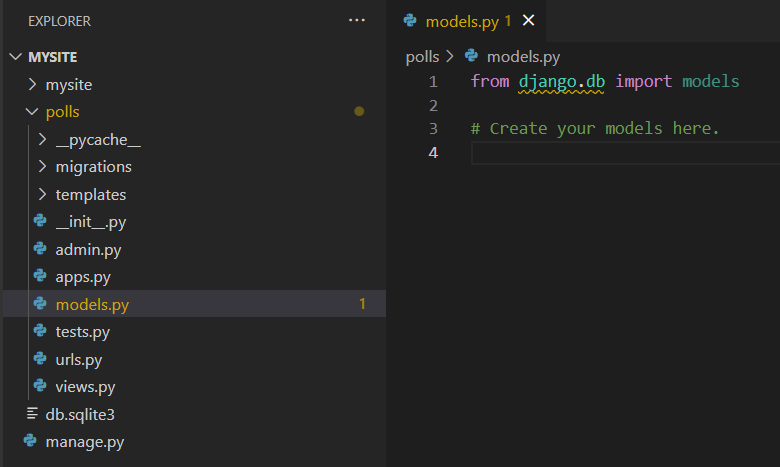
#### Creating model of your apps

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

In our poll app, we’ll create two models: **Question** and **Choice**. A **Question** has a question and a publication date. A **Choice** has two fields: the text of the choice and a vote tally. Each **Choice** is associated with a **Question**.

These concepts are represented by Python classes. Edit the **polls/models.py** file so it looks like this:

1. Open the app **“polls/models.py**”

****

1. **Add the following code for creating table “question”:**

class Question(models.Model):

    question\_id = models.AutoField(primary\_key=True)

    question\_text = models.CharField(max\_length=254)

    pub\_date = models.DateTimeField('date published')

1. **Creating table “choice”**

class Choice(models.Model):

    choice\_id = models.AutoField(primary\_key=True)

    question\_id = models.ForeignKey(Question, on\_delete=models.CASCADE)

    choice\_text = models.CharField(max\_length=200)

    votes = models.IntegerField(default=0)

Here, each model is represented by a class that subclasses **[django.db.models.Model](https://docs.djangoproject.com/en/4.0/ref/models/instances/" \l "django.db.models.Model" \o "django.db.models.Model)**. Each model has a number of class variables, each of which represents a database field in the model.

Each field is represented by an instance of a [**Field**](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.Field) class – e.g., **[CharField](https://docs.djangoproject.com/en/4.0/ref/models/fields/" \l "django.db.models.CharField" \o "django.db.models.CharField)** for character fields and [**DateTimeField**](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.DateTimeField) for datetimes. This tells Django what type of data each field holds.

The name of each [**Field**](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.Field) instance (e.g. **question\_text** or **pub\_date**) is the field’s name, in machine-friendly format. You’ll use this value in your Python code, and your database will use it as the column name.

You can use an optional first positional argument to a [**Field**](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.Field) to designate a human-readable name. That’s used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn’t provided, Django will use the machine-readable name. In this example, we’ve only defined a human-readable name for **Question.pub\_date**. For all other fields in this model, the field’s machine-readable name will suffice as its human-readable name.

Some [**Field**](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.Field) classes have required arguments. **[CharField](https://docs.djangoproject.com/en/4.0/ref/models/fields/" \l "django.db.models.CharField" \o "django.db.models.CharField)**, for example, requires that you give it a **[max\_length](https://docs.djangoproject.com/en/4.0/ref/models/fields/" \l "django.db.models.CharField.max_length" \o "django.db.models.CharField.max_length)**. That’s used not only in the database schema, but in validation, as we’ll soon see.

A [**Field**](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.Field) can also have various optional arguments; in this case, we’ve set the [**default**](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.Field.default) value of **votes** to 0.

Finally, note a relationship is defined, using **[ForeignKey](https://docs.djangoproject.com/en/4.0/ref/models/fields/" \l "django.db.models.ForeignKey" \o "django.db.models.ForeignKey)**. That tells Django each **Choice** is related to a single **Question**. Django supports all the common database relationships: many-to-one, many-to-many, and one-to-one.

##### **Model Manager**

“A Manager is the interface through which database query operations are provided to Django models.”

**Advantages of model managers:**

* Readability - when we use model managers, we reduce complexity in our model.
* Ease of maintenance - model managers make our model simple and readable which also leads to ease of maintenance in the long run.

1. **Example 1 (**using **model.Manager):**

Assigning “**models.Manager**” into “**objects”** variable.

from django.db import models

# Create your models here.

# For Question Table

class Question(models.Model):

    question\_id = models.AutoField(primary\_key=True)

    question\_text = models.CharField(max\_length=254)

    pub\_date = models.DateTimeField('date published')

    objects = models.Manager

# For Choice Table

class Choice(models.Model):

    choice\_id = models.AutoField(primary\_key=True)

    question\_id = models.ForeignKey(Question, on\_delete=models.CASCADE)

    choice\_text = models.CharField(max\_length=200)

    votes = models.IntegerField(default=0)

    objects = models.Manager

1. **Example 2** (creating class manager of class **Question)**:

We created a class **QuestionManager** and created a validation method before it inserted to database. You can also add methods for your query sets or any SQL statements.

class QuestionManager(models.Manager):

    def validator(self, postData):

        errors = {}

        if len(postData['question\_text']) > 254:

            errors['question\_text'] = "Question text cannot be longer than 254 characters"

        return errors

To use the **QuestionManager**, we call and assign it to a variable “**objects**” inside **Question** Class:

class Question(models.Model):

    question\_id = models.AutoField(primary\_key=True)

    question\_text = models.CharField(max\_length=254)

    pub\_date = models.DateTimeField('date published')

    objects = QuestionManager() #added/call question’s manager

Notes: if you have no queryset inside QuestionManager class,

the **“objects = QuestionManager()”** is the same as **“objects = model.Manager”**.

1. **Example 3 (**queryset in **Manager** class**):**

class QuestionManager(models.Manager):

    def get\_queryset(self):

        return super().get\_queryset().filter(pub\_date='2022-02-04')

class Question(models.Model):

    question\_id = models.AutoField(primary\_key=True)

    question\_text = models.CharField(max\_length=254)

    pub\_date = models.DateTimeField('date published')

objects = models.Manager() #display all questions

    filtered\_objects = QuestionManager() #display questions published at 2022-02-04

#### Activating Models

That small bit of model code gives Django a lot of information. With it, Django can:

* Create a database schema (**CREATE TABLE** statements) for this app.
* Create a Python database-access API for accessing **Question** and **Choice** objects.

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

To include the app in our project, we need to add a reference to its configuration class in the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-INSTALLED_APPS) setting.

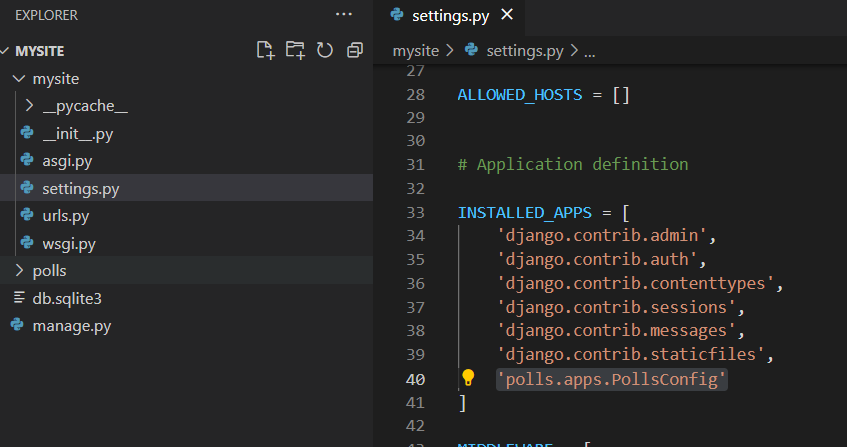
1. Open your “**polls/apps.py”** and you will see the config class (**PollsConfig**) of our polls apps.

Graphical user interface, text, application

Description automatically generated

The **PollsConfig** class is in the **polls/apps.py** file, so its dotted *path* is **'polls.apps.PollsConfig'**

1. Open your “**mysite/settings.py”** inside **INSTALLED\_APPS,** replace **'polls'** into **'polls.apps.PollsConfig'**



1. Now Django knows to include the **polls** app. Let’s run another command:

py manage.py makemigrations polls

Text

Description automatically generated

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*.

Migrations are how Django stores changes to your models (and thus your database schema) - they’re files on disk.

1. **Migrate** Command

py manage.py migrate

Text

Description automatically generated

The **Questions** & **Choices** in your polls’ model will now be added to your database.

Graphical user interface, application

Description automatically generated

## **CRUD**

* 1. Create a template for adding questions.

A screenshot of a computer

Description automatically generated with medium confidence

* 1. Create a method at **views.py** and render the template.

Text

Description automatically generated

To use the model **question**, you need to import it in your views

from polls.models import Question

* 1. Add path on your **polls/urls.py**

path('add\_question', views.add\_question),

* 1. Run your project

Graphical user interface, text, application

Description automatically generated

**Form Method**

* **POST Method**

if request.method == 'POST':

question\_text = request.POST["question\_text"]

pub\_date = datetime.now()

* **Save**

question = Question.objects.create(question\_text = request.POST["question\_text"], pub\_date = datetime.now())

question.save()

To use the model **Question**, you need to import it in your views

* **Full Code for views**

from datetime import datetime

from django.http import HttpResponse

from django.shortcuts import render

from polls.models import Question

# Create your views here.

def add\_question(request):

    if request.method == 'POST':

        question = Question.objects.create(question\_text = request.POST["question\_text"], pub\_date = datetime.now())

        question.save()

    context = {

        "page" : "Create Poll's Question."

    }

    return render(request, 'add\_question.html', context)

* **Select**

Views

**Text

Description automatically generated**

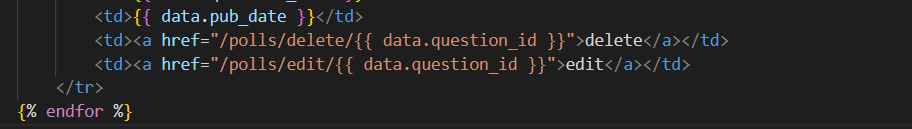
Template

A screenshot of a computer

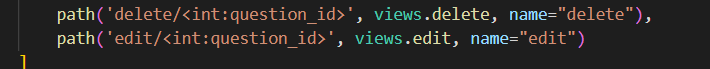
Description automatically generated with medium confidence

* **Update**

Template: view\_question.html, add the delete & edit button:



Urls.py



Views.py

A computer screen with colorful text

Description automatically generated

Template

A computer screen with text

Description automatically generated

* **Delete**

question = Question.objects.get(question\_id = 25)

question.delete()

A computer code on a black background

Description automatically generated

## **Convert existing Database to model**

If you have an existing database file and you want to create the model classes (models.py).

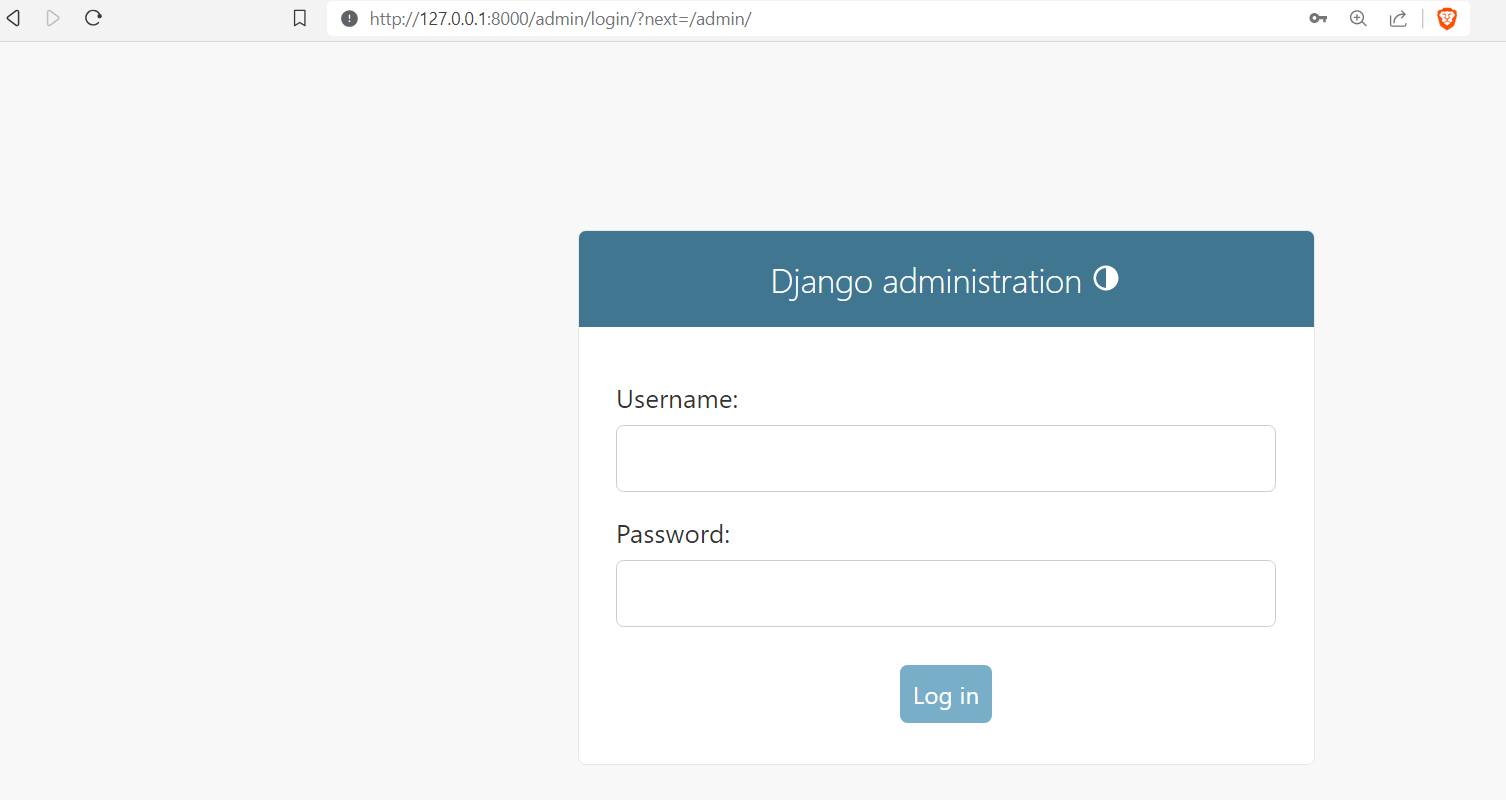
Before running this, you will have to configure your database in the **settings.py** file. The result will be a file containing a model for each table.

$ python manage.py inspectdb > models.py

The output file will be saved to your current directory. Move that file to the correct app and you have a good starting point for further customizations.

## **Django Admin**

<http://your-server-ip:8000/admin/>



### **Create Superuser**

python manage.py createsuperuser

**Output**

Username (leave blank to use 'root'): admin\_user

Email address: juan@example.com

Password:

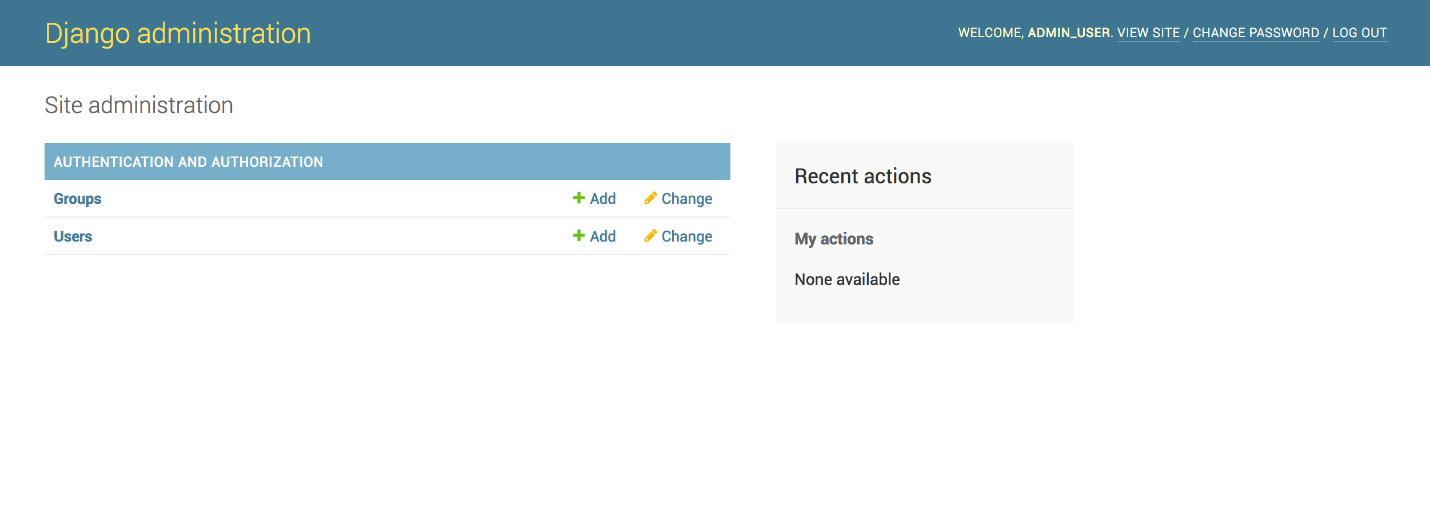
Password (again):

At this point, we now have an admin account with the username admin\_user and the password admin123.

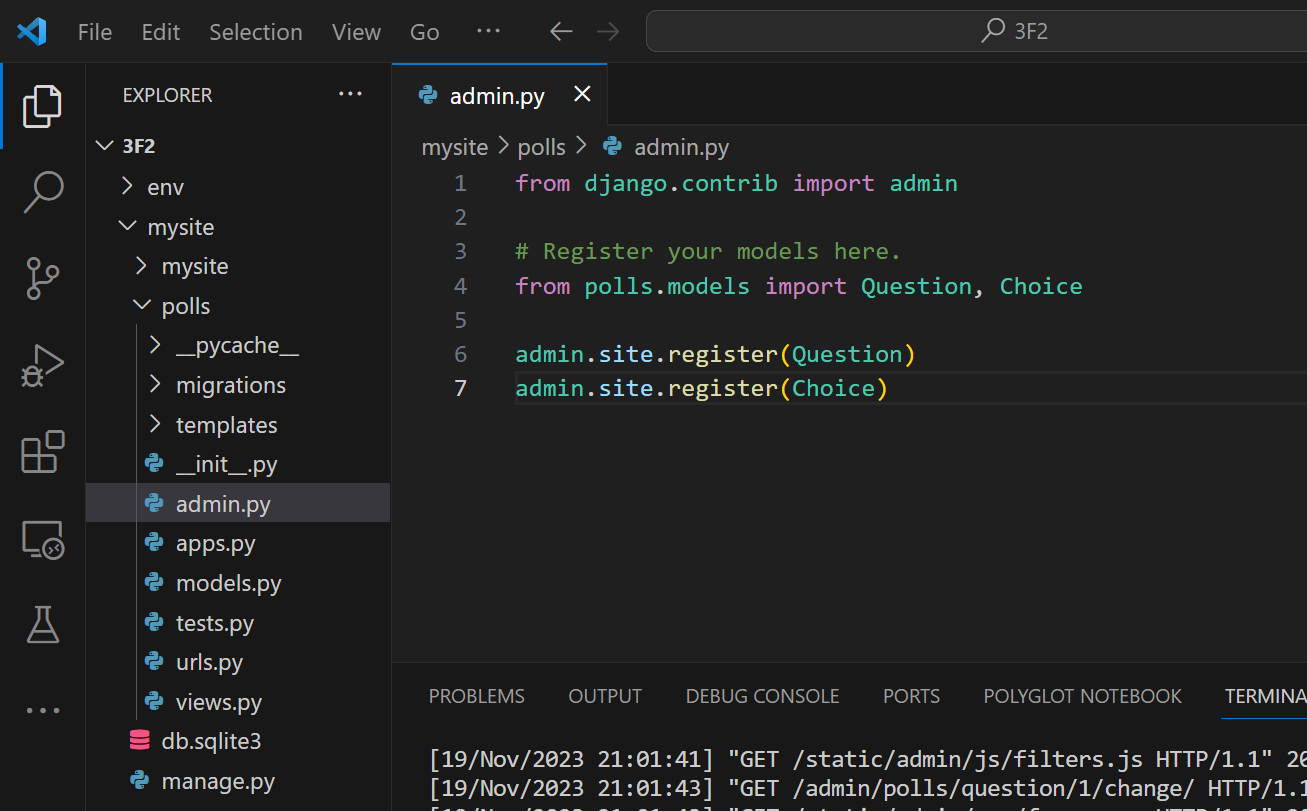
Let’s log in and investigate what exists on our admin page.

If needed, run the Django app again with python manage.py runserver 0.0.0.0:8000 and then navigate once more to the URL http://your-server-ip:8000/admin/ to get to the admin login page. Then log in with the username and password and password you just created.

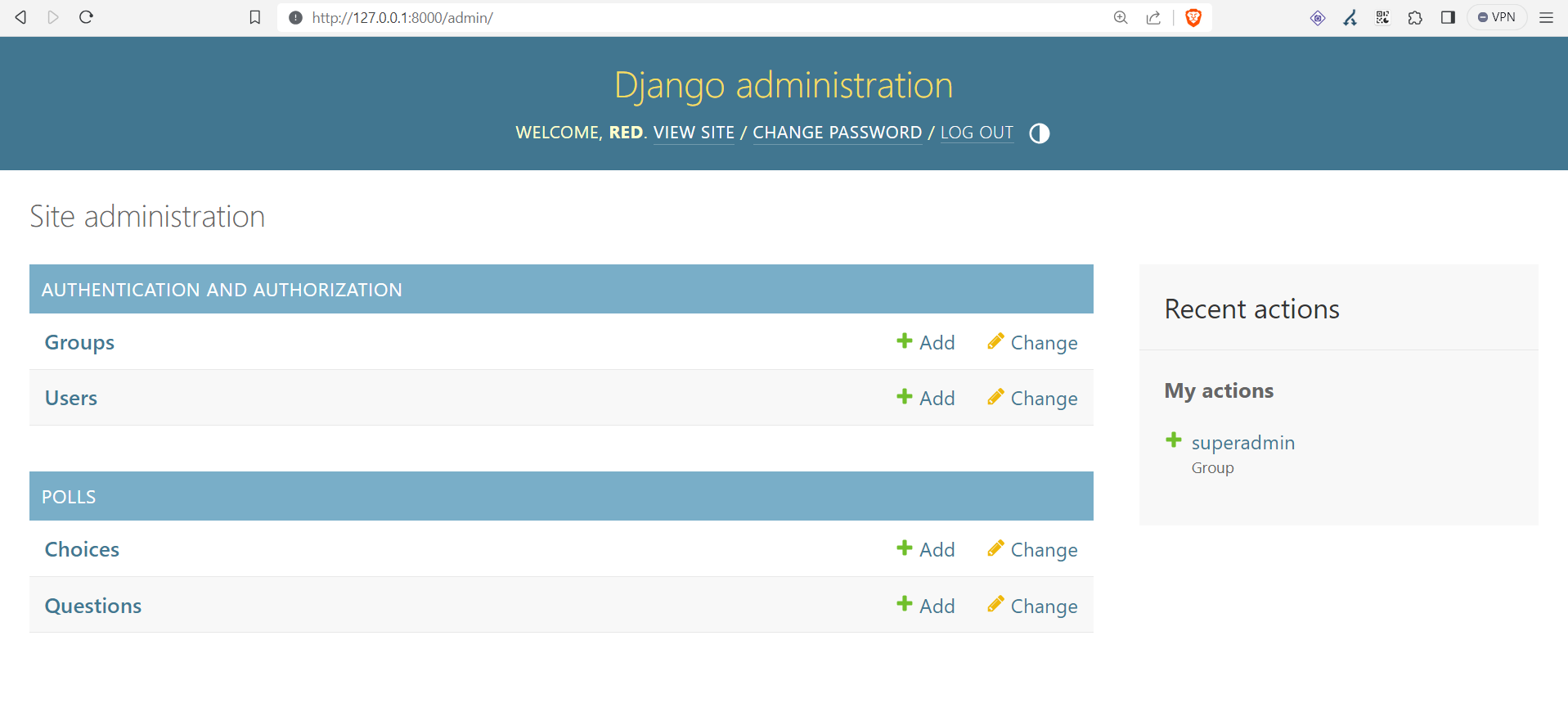
After a successful login, you’ll receive the following page.



### **Register your Model**



Now, your model will appear on the admin page.



# [**REFERENCES:**](https://docs.djangoproject.com/en/4.0/)

<https://docs.djangoproject.com/en/4.0/>

<https://learndjango.com/tutorials/django-best-practices-projects-vs-apps#:~:text=Apps%20A%20Django%20app%20is%20a%20small%20library,another%20app%20called%20payments%20to%20charge%20logged-in%20subscribers>.