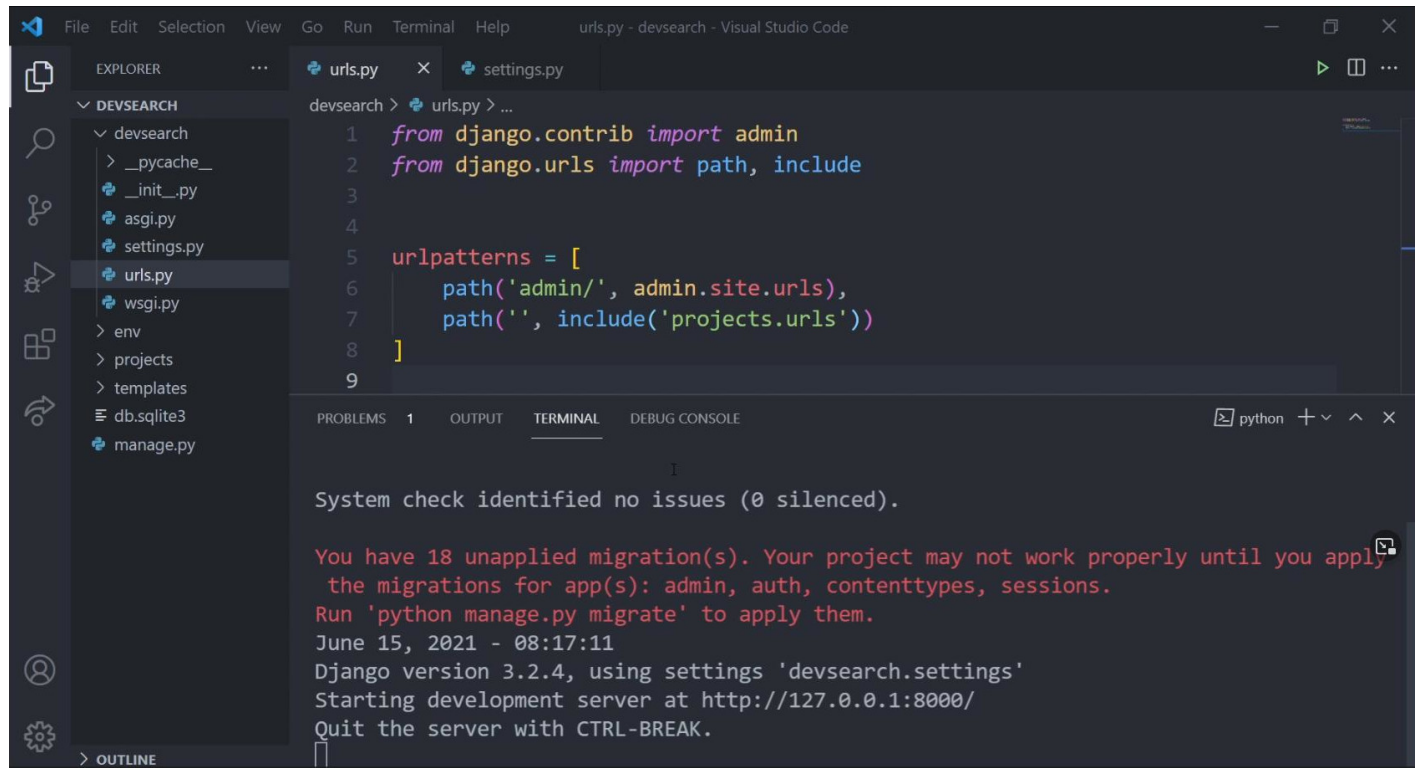


# Database Managing in Django

Possibly by now you're aware that when first starting the server on your Django project, just after creating it:

```
django-admin startproject <project_name>
python manage.py runserver
```



by default, Django uses SQLITE3 database

The screenshot shows the Visual Studio Code editor with the `settings.py` file open. The `DATABASES` configuration is visible, showing the default database as `sqlite3`. The terminal at the bottom shows the output of a Django command, indicating that migrations for `admin`, `auth`, `contenttypes`, and `sessions` are needed. The terminal text is as follows:

```

the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
June 15, 2021 - 08:17:11
Django version 3.2.4, using settings 'devsearch.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

```

We can always switch to another database by changing the configuration file

`<root_project_folder>/settings.py`

For the time being, we'll keep working with the default database SQLITE3.

In order to solve the issue with the database error, it's necessary to build the default database tables set up by Django. We do this by using the following commands:

```
python manage.py makemigrations
python manage.py migrate
```

we can set up a bash script to simplify this task since we'll have to do every time we do some change to the database tables by modifying any attribute in any table:

```
#!/usr/bin/bash
# migrate.sh

python manage.py makemigrations && python manage.py migrate
```

Name the script `migrate.sh`, place it at the root folder and give it execution privileges

```
chmod +x migrate.sh
```

For windows development environments:

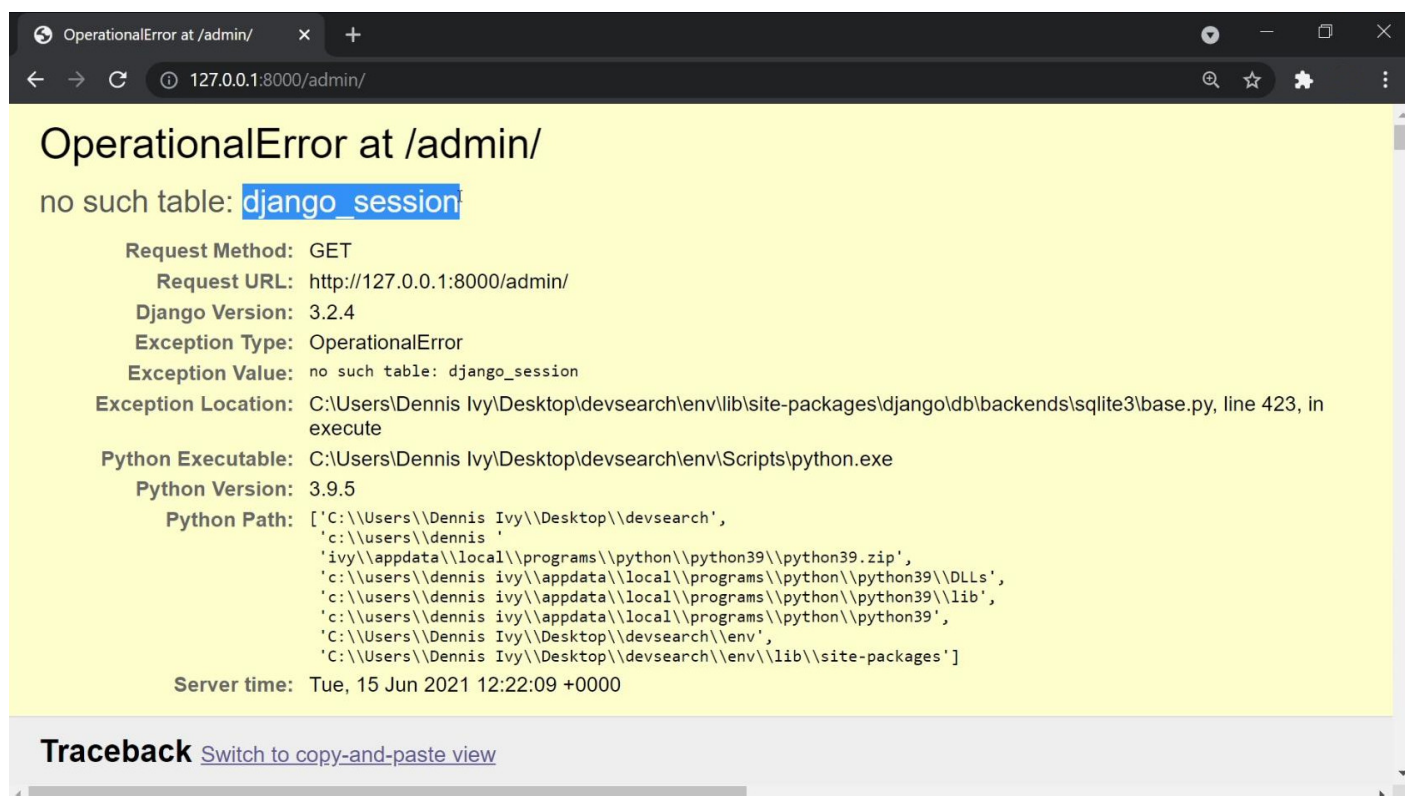
```
python manage.py makemigrations
python manage.py migrate
```

name the script migrate.bat; place it on the project root level, the same as manage.py; and give it execution privileges.

Execute every time you do modifications to any models.py file.

## Admin Panel

Django provides by default all the machinery for an Administration Panel. However, if the database is not set up from the beginning, it's not possible to access the administration panel. Without setting the database, if you try to access the administration panel you get an 'OperationalError' error in your browser:



In order to solve this issue, it's necessary to implement two things:

1. Create the default Django tables in the database by running your migrate.sh or migrate.bat script, and
2. Create a super user for the administration panel.

We already reviewed the first point and how to solve it. The second point can be only done after completing the first point.

To create a superuser we need the server up, run:

```
python manage.py runserver
```

only if the server is not running. And use the command:

```
python manage.py createsuperuser
```

which will launch a scrip requesting username and password for the superuser:

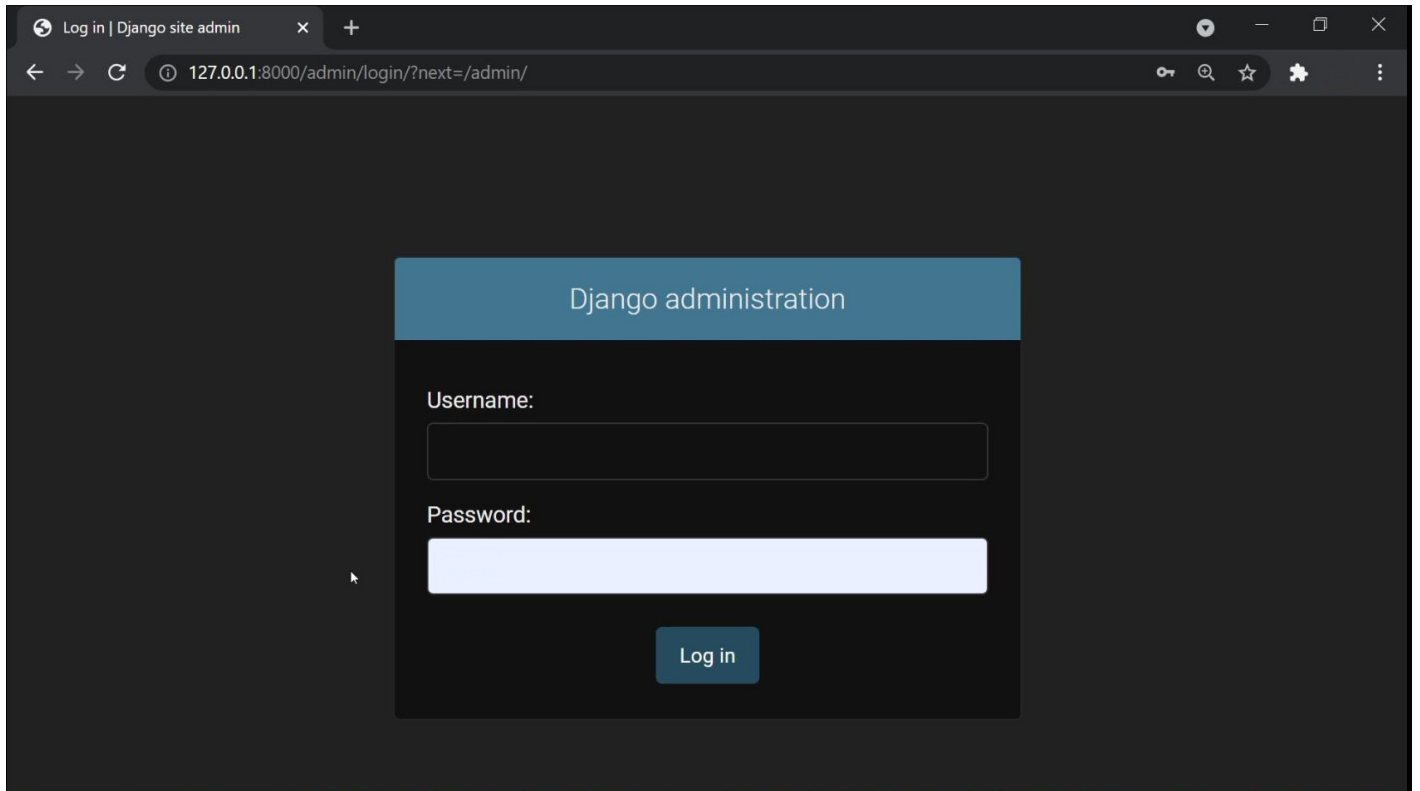
```
Username (leave blank to use '<default_user>'): <type username or leave it  
blank for <default_user> >  
Email: <type your email addresss>  
Password: <type your password here>  
Password (again): <type your password here>  
Superuser created successfully.
```

The last line will only happen if there were no errors when typing the password.

Check that the superuser has been set up correctly by accessing the administration panel in your browser at:

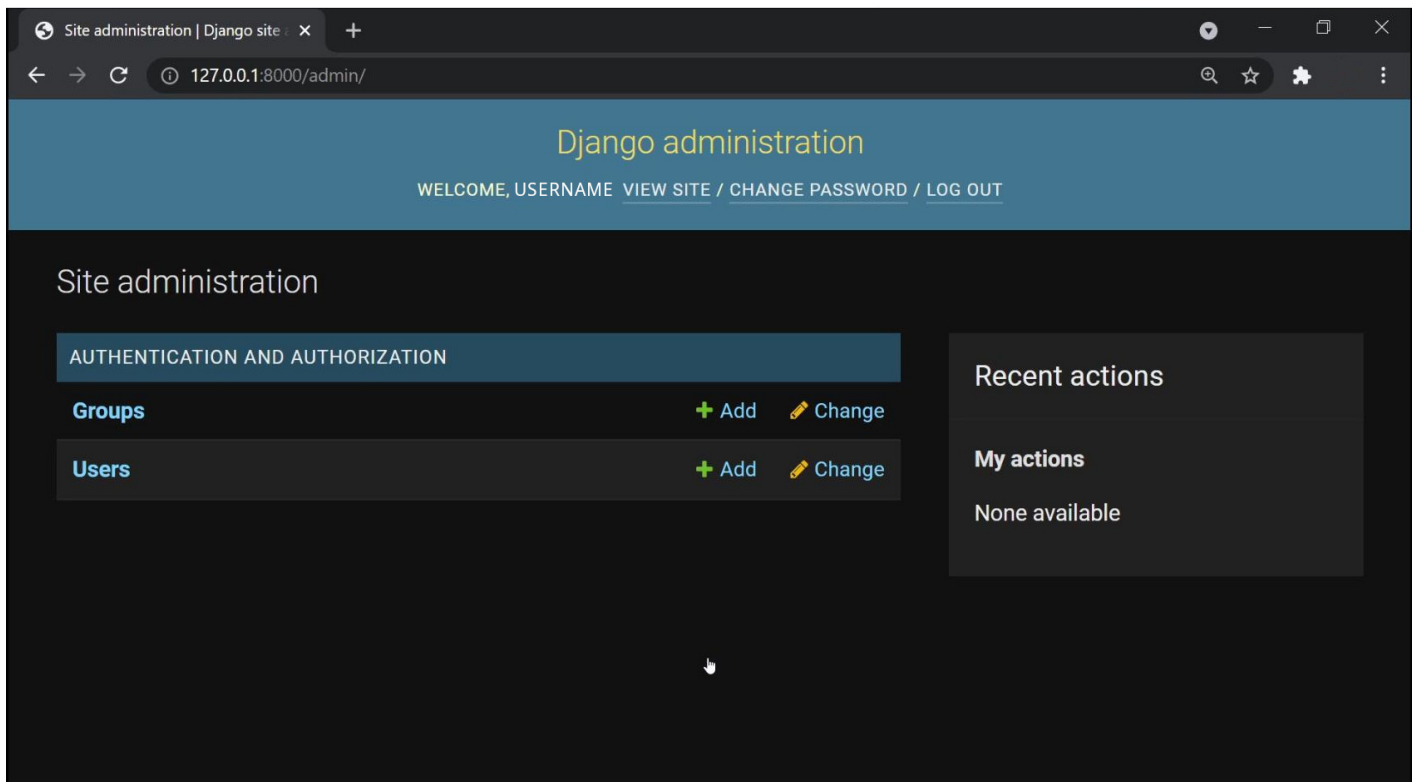
```
127.0.0.1:8000/admin
```

you should see the login page for the administration panel



though, the theme settings may vary and the background be white colored and the Django administration bar have a different color.

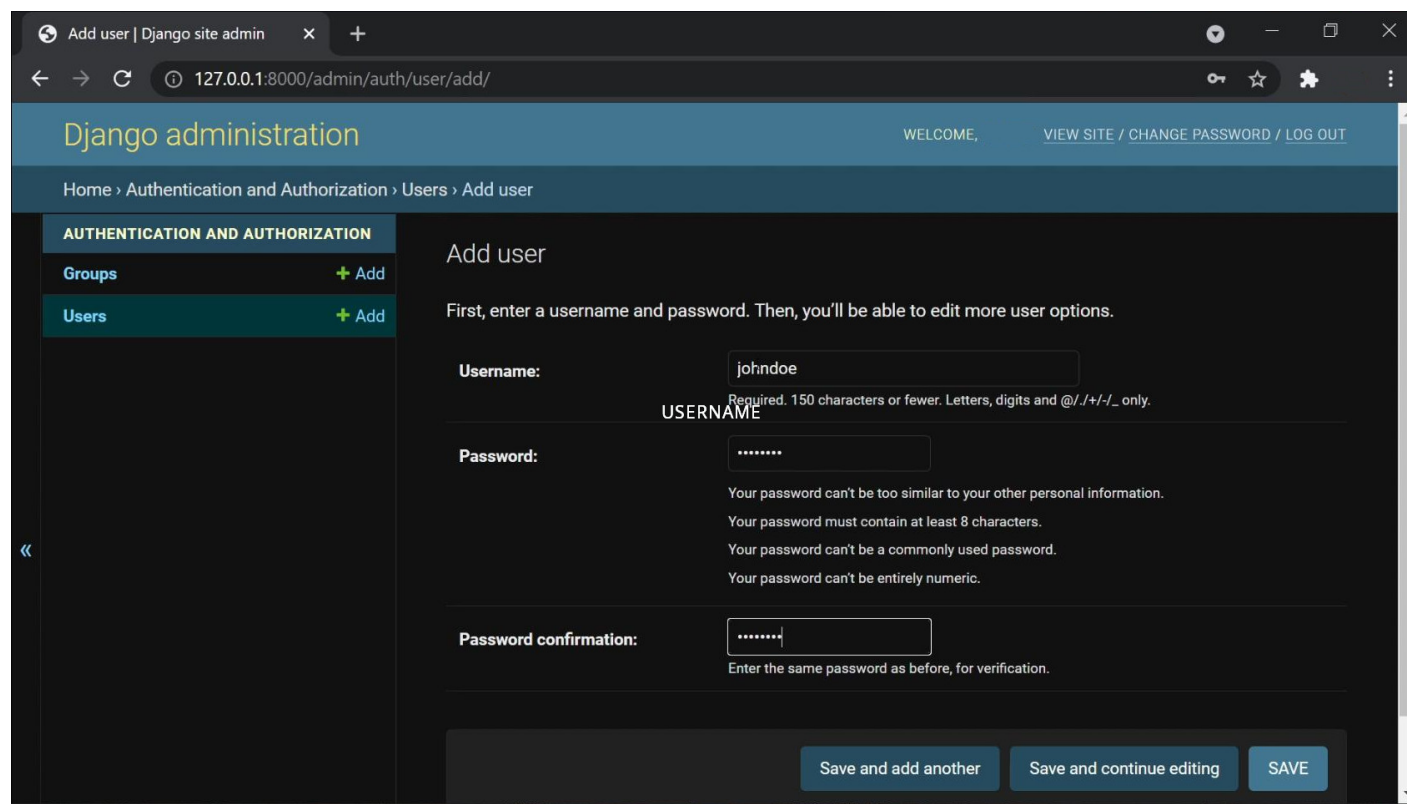
Enter your superuser name and password and you'll ingress to the administration panel



This is the database management area. Here it's possible to interact with the database and do CRUD operations directly into the database tables. On the initial phase, there are only two tables set up by Django: the Groups and the Users tables.

The Users table will contain a single user, the superuser. We should be able to add new users or modify the superuser directly from this panel.

Play around with the Users table and add a new user or change the data for your superuser

The screenshot shows a web browser window with the Django administration interface. The address bar shows the URL '127.0.0.1:8000/admin/auth/user/add/'. The page title is 'Django administration'. The left sidebar has a menu for 'AUTHENTICATION AND AUTHORIZATION' with links for 'Groups' and 'Users', both with '+ Add' buttons. The main content area is titled 'Add user' and contains a form. The form has fields for 'Username' (with the value 'johndoe') and 'Password' (with masked characters). Below the password field are four lines of password requirements: 'Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only.', 'Your password can't be too similar to your other personal information.', 'Your password must contain at least 8 characters.', and 'Your password can't be a commonly used password.' There is also a 'Password confirmation' field with masked characters and a note 'Enter the same password as before, for verification.' At the bottom of the form are three buttons: 'Save and add another', 'Save and continue editing', and 'SAVE'.

## How To Add Tables To The Database And Connect Them With The Admin Panel

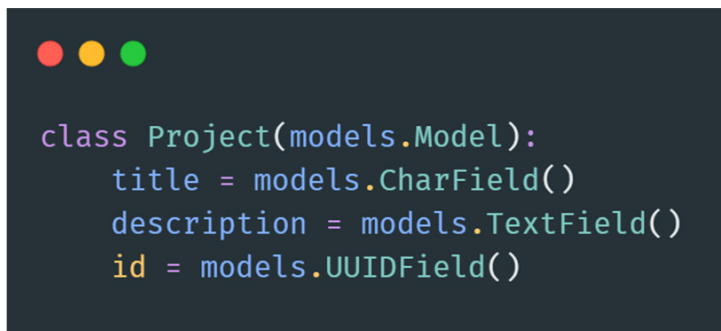
Django has a modular philosophy for adding tables to the database. Each Django application in the project is in charge of creating its own tables and internal relationships. Then, the application's tables are connected to the general database via the administration panel through a registration process of the application.

Each Django application has its own directory `<django_application_name>` and in the directory there's a `models.py` file:

```
<django_application_name>/models.py
```

which contains the Django's Object Relational Mapper (**ORM**) models for that particular Django application. A Django **ORM** model is a Python derived class from `models.Model` class that represents a single table record inside a Table with the same name as the class

## Python Class (Inherits from Django Models)



```
class Project(models.Model):
    title = models.CharField()
    description = models.TextField()
    id = models.UUIDField()
```



Database Table		
ID	TITLE	DESC.
1	...	...
2	...	...
3	...	...
4	...	...
5	...	...

**models.<Type>Field** are member-property classes in charge of specific features required on the database for each table attribute (**table.column**). A complete list of member properties can be found [here](#)

**IMPORTANT NOTE:** The term *member-property* class is used here to refer to a somewhat confusing usage of properties in Python. If you take a quick look at the image above, you'll realize that the variable definitions are in the place of a **class variable** (a variable that has a scope in the range of all objects instantiated from the **Class.variable\_name**). Here, however, the variables are **NOT** class variables, they are **object variables**, meaning they have the same usage as any variable declared as **self.variable\_name**.

After a mouthful of technical terms, let's go back to how to construct a record model using the Django **ORM** classes and field types

Let's create a model for our project table:

```
from django.db import models      # This is setup by Django by default
import uuid

class Project(models.Model):      # All model classes derive from the Model
class in the models module
    title = models.CharField(max_length=200)
    description = models.TextField(null=True, blank=True)
    demo_link = models.CharField(max_length=200, null=True, blank=True)
    source_link = models.CharField(max_length=200, null=True, blank=True)
    created_on = models.DateTimeField(auto_add_now=True, editable=False)
    id = models.UUIDField(unique=True, uuid=uuid4, primary_key=True,
    editable=False)

    # Only for the time being.
    # def __str__(self):
    #     return self.title
```

Important point from the code above:

1. There are some Fields that have required construction parameters, for example:

```
CharField(max_length=200)
```

requires that the *max\_length* parameter always defines an argument with the length of the character field.

2. Some construction parameters define specific behaviors for databases, for example, *unique* and *primary\_key* in

```
UUIDField(unique=True, primary_key=True, ... )
```

define specific behaviours for the *<table\_name>.id* attribute. In detail, *unique* indicates that every row in the column can't be repeated, all rows have different values. The *primary\_key* indicates that the column is used to index each record (every row) and that the column can be used by other tables to reference the specific record with the id column.

3. The parameter *auto\_add\_now* and *editable* in

```
DateTimeField(auto_add_now=True, editable=False)
```

tells that the field should take the timestamp at the record's creation time and *editable=False* tells that the column won't accept changes to the value later on (it becomes non-editable)

4. The *null* and *blank* parameters seem to have related actions, but their actions are applied on different areas:

```
TextField(null=True, ...)
CharField(null=True, ...)
```

tells the database should accept NULL values in the attribute, the default is *null=False*.

```
TextField(blank=True, ...)
CharField(blank=True, ...)
```



tells any Form (View for input) that it should accept empty strings or values for the attribute, the default is blank=False or that the FORM **should not accept empty values**.

Continuing with our example, we need to construct and link the new model with the administration panel.

Construct the model by running:

```
python manage.py makemigrations
python manage.py migrate
```

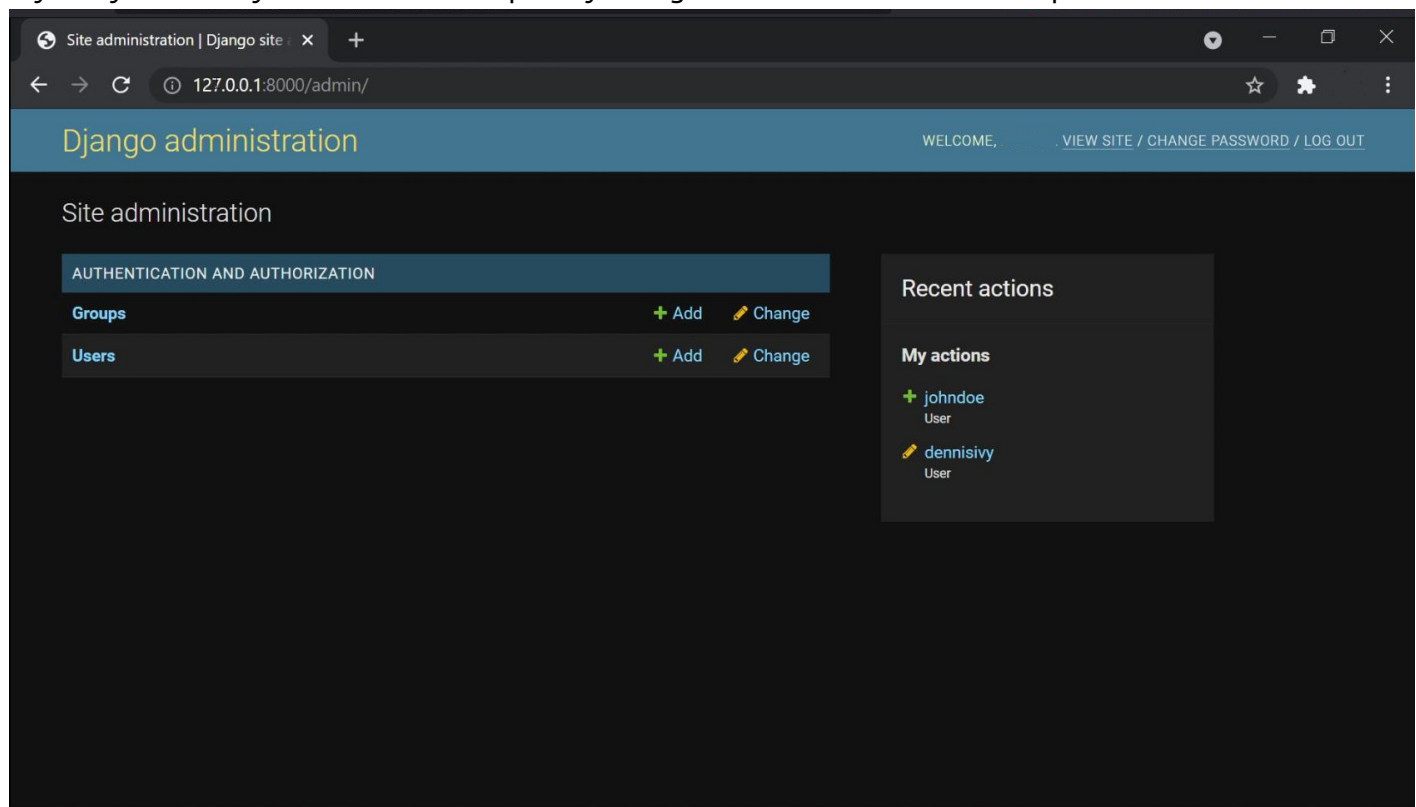
or if you have your script already:

```
./migrate.sh
```

or if in windows:

```
migrate
```

If you try to access your administration panel you might find that it doesn't show up



To correct that, we need to connect the new model to the Administration panel by "registering" the model.

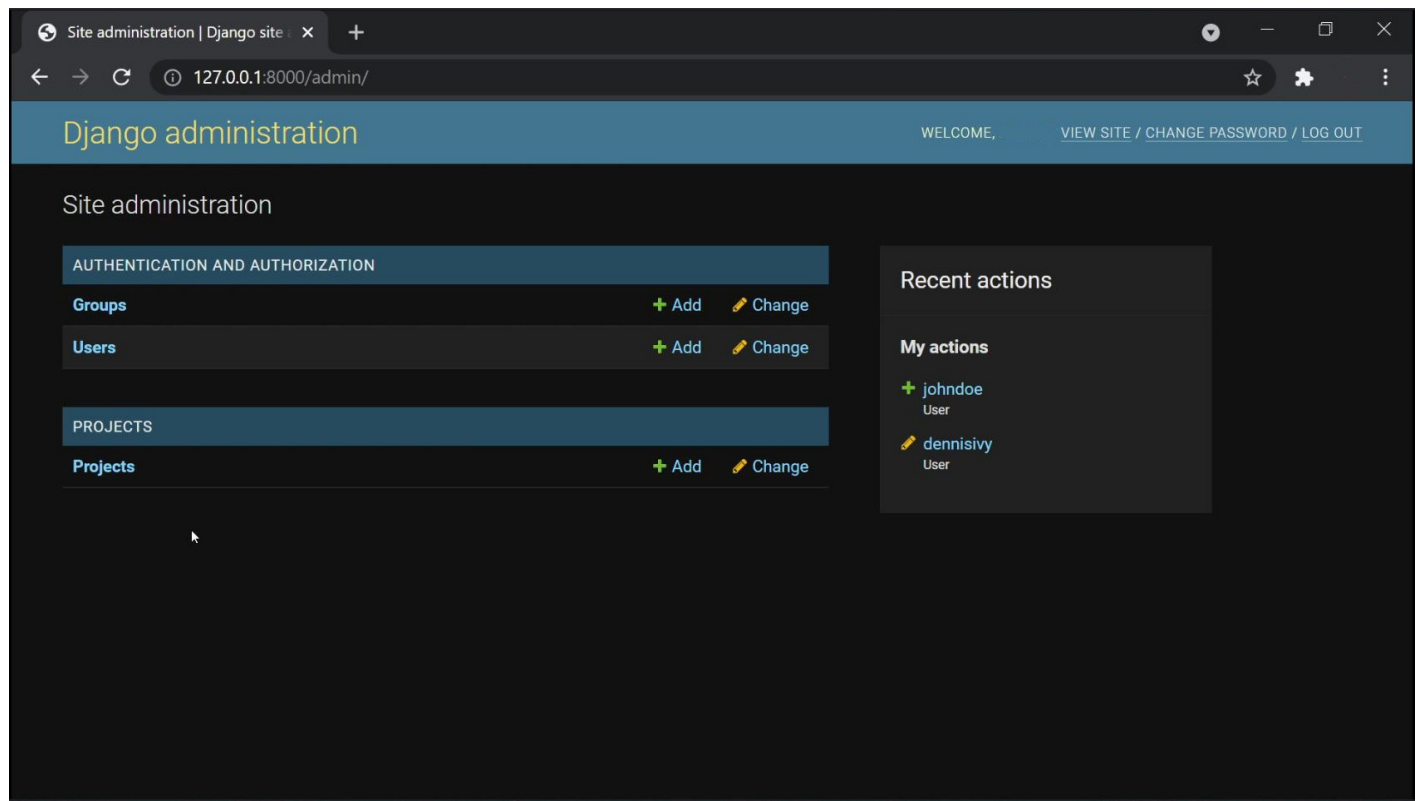
1. Access the projects/admin.py file
2. Import the projects.models module to the file and register the model:

```
from django.contrib import admin

# Register your models here.
from .models import Project          # Both files are in the same
directory, that why the .

admin.site.register(Project)        # Registering your model
```

Now, if we refresh the administration panel in the browser, you should see the new model in the panel:



## Summary

There are three main steps involved in creating a new Database model in Django:

1. Create your model in your django application by including the model in the `<application_name>/models.py`. Use a class derived from `models.Model` base class. Describe your attributes using the **ORM** Field types.
2. Create the tables or field modifications in the database by migrating the database to its new state. Use the commands:

```
python manage.py makemigrations  
python manage.py migrate
```

3. Register the models into the Administration panel in the <application\_name>/admin.py file

```
from django.contrib import admin  
from .models import <Your_Model_Class>  
  
# Register your models here  
admin.site.register(<Your_Model_Class>)
```